A User Centric Security Model for Tamper-Resistant Devices

Raja Naeem Akram

Thesis submitted to the University of London
for the degree of Doctor of Philosophy

Information Security Group
Department of Mathematics
Royal Holloway, University of London

2012

To my father,
Raja Muhammad Akram

Declaration

These doctoral studies were conducted under the supervision of Dr. Konstantinos Markantonakis.

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Mathematics as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Raja Naeem Akram
January, 2012

Publications

A number of papers resulting from this work have been presented in refereed conferences.

R. N. Akram, K. Markantonakis, and K. Mayes, "Application Management Framework in User Centric Smart Card Ownership Model," in The 10th International Workshop on Information Security Applications (WISA09), ser. LNCS, H. Y. Youm and M. Yung, Eds., vol. 5932/2009. Busan, Korea: Springer, August 2009.

R. N. Akram, K. Markantonakis, and K. Mayes, "Location Based Application Availability," in On the Move to Meaningful Internet Systems: OTM 2009 Workshops, ser. LNCS, R. Meersman, P. Herrero, and T. Dillon, Eds., vol. 5872/2009. Vilamoura, Portugal: Springer, November 2009.

R. N. Akram, K. Markantonakis, and K. Mayes, "A Paradigm Shift in Smart Card Ownership Model," in Proceedings of the 2010 International Conference on Computational Science and Its Applications (ICCSA 2010), B. O. Apduhan, O. Gervasi, A. Iglesias, D. Taniar, and M. Gavrilova, Eds. Fukuoka, Japan: IEEE Computer Society, March 2010.

R. N. Akram, K. Markantonakis, and K. Mayes, "Firewall Mechanism in a User Centric Smart Card Ownership Model," in Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, ser. LNCS, D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, Eds., vol. 6035/2010. Passau, Germany: Springer, April 2010.

R. N. Akram, K. Markantonakis, and K. Mayes, "A Dynamic and Ubiquitous Smart Card Security Assurance and Validation Mechanism," in 25th IFIP International Information Security Conference (SEC 2010), ser. IFIP AICT Series, K. Rannenberg and V. Varadharajan, Eds. Brisbane, Australia: Springer, September 2010.

R. N. Akram, K. Markantonakis, and K. Mayes, "Simulator Problem in User Centric Smart Card Ownership Model," in 6th IEEE/IFIP International Symposium on Trusted Computing and Communications (TrustCom-10), H. Y. Tang and X. Fu, Eds. HongKong, China: IEEE Computer Society, December 2010.

R. N. Akram, K. Markantonakis, and K. Mayes, "Application-Binding Protocol in the User Centric Smart Card Ownership Model," in the 16th Australasian Conference on Information Security and Privacy (ACISP), ser. LNCS, U. Parampalli and P. Hawkes, Eds. Melbourne, Australia: Springer, July 2011.

R. N. Akram, K. Markantonakis, and K. Mayes, "User Centric Security Model for Tamper-Resistant Devices," in 8th IEEE International Conference on e-Business Engineering (ICEBE 2011), J. Li and J.-Y. Chung, Eds. Beijing, China: IEEE Computer Science, October 2011.

R. N. Akram, K. Markantonakis, and K. Mayes, “Cross-Platform Application Sharing Mechanism,” in 10th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-11), H. Wang, S. R. Tate, and Y. Xiang, Eds. Changsha, China: IEEE Computer Science, November 2011.

R. N. Akram, K. Markantonakis, and K. Mayes, "Pseudorandom Number Generation in Smart Cards: An Implementation, Performance and Randomness Analysis", in 5th International Conference on New Technologies, Mobility and Security (NTMS), Antonio Mana, and Marek Klonowski, eds., Istanbul, Turkey, IEEE Computer Science, May 2012.

R. N. Akram, K. Markantonakis, and K. Mayes, “A Privacy Preserving Application Acquisition Protocol,” in 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-12), Geyong Min, Felix Gomez Marmol, Eds. Liverpool, United Kingdom: IEEE Computer Science, June 2012.

Acknowledgements

This thesis would not have been possible without the guidance, support, and encouragement of a number of people.

My initial and foremost thanks go to my supervisors Dr. Konstantinos Markantonakis and Prof. Keith Mayes. I am especially thankful to Dr. Markantonakis for his support, interest, patience, and guidance throughout my Masters and Doctoral studies at Royal Holloway. During my Ph.D studies, in the very first meeting with Dr. Markantonakis, he told me that a Ph.D is a life-changing experience and I should take it as such. This came true in many ways — not just restricted to academic aspects.

I would like to offer my gratitude to Sheila Osborn, Hisham Abbasi, and Gerhard Hancke for excellent reviews of my early draft of the thesis. Any remaining mistakes are due solely to my negligence and omission.

During my stay at Royal Holloway, I met wonderful people who not only inspired me academically but also personally. No doubt they have left their mark on my personality. Among these remarkable people, I thank Alam Hussain, Babur Mahmood, Salman Bashir, Takreem Anwar, Hassan Sher, Xuefei Leng, Yasir Abbasi, Abdul Qaddoos, Fahad Mehmood, and Kashif Munir. I would also like to convey my thanks to Kashif Riaz, without whom I may not have developed an interest in computer science and would have lost my way ages ago.

I would like to thank Min Chen for the support, encouragement, and sense of exploration she instilled in me. I thank her for reviewing my early drafts that had gigantic proportion of mistakes, but she still corrected them with a smile. I would also like to thank Qianqian Tang, Nhung Nguyen, Margaret Ronia, Jie Lie, Sun Beilei, and Olive Cheung.

I want to convey my deepest respect and appreciation for my uncle Raja Muhammad Azam for his tireless help, teaching, and support. I thank him for developing my interest in mathematics and engineering along with making me an inquisitive person. I would like to pay tribute to my late father Raja Muhammad Akram for his encouragement, training, and guidance. I would also like to thank my late brother Raja Qaiser Mehmood for his ever-present support and for giving me lot of cherished memories.

Finally, I could not have reached at this place without the support and help of my brother Raja Nadeem Ashraf and my mom. Without my brother's support I would have completed my M.Sc at Royal Holloway, and may not have got the opportunity to do a Ph.D with Dr. Markantonakis. I am thankful to both of you for the opportunities you have created for me and patience you have shown for my transgressions.

Abstract

In this thesis we propose a design for a ubiquitous and interoperable device based on the smart card architecture to meet the challenges of privacy, trust, and security for traditional and emerging technologies like personal computers, smart phones and tablets. Such a device is referred a User Centric Tamper-Resistant Device (UCTD). To support the smart card architecture for the UCTD initiative, we propose the delegation of smart card ownership from a centralised authority (i.e. the card issuer) to users. This delegation mandated a review of existing smart card mechanisms and their proposals for modifications/improvements to their operation.

Since the inception of smart card technology, the dominant ownership model in the smart card industry has been refer to as the Issuer Centric Smart Card Ownership Model (ICOM). The ICOM has no doubt played a pivotal role in the proliferation of the technology into various segments of modern life. However, it has been a barrier to the convergence of different services on a smart card. In addition, it might be considered as a hurdle to the adaption of smart card technology into a general-purpose security device.

To avoid these issues, we propose citizen ownership of smart cards, referred as the User Centric Smart Card Ownership Model (UCOM). Contrary to the ICOM, it gives the power of decision to install or delete an application on a smart card to its user. The ownership of corresponding applications remains with their respective application providers along with the choice to lease their application to a card or not. In addition, based on the UCOM framework, we also proposed the Coopetitive Architecture for Smart Cards (CASC) that merges the centralised control of card issuers with the provision of application choice to the card user.

In the core of the thesis, we analyse the suitability of the existing smart card architectures for the UCOM. This leads to the proposal of three major contributions spanning the smart card architecture, the application management framework, and the execution environment. Furthermore, we propose protocols for the application installation mechanism and the application sharing mechanism (i.e. smart card firewall). In addition to this, we propose a framework for backing-up, migrating, and restoring the smart card contents.

Finally, we provide the test implementation results of the proposed protocols along with their performance measures. The protocols are then compared in terms of features and performance with existing smart cards and internet protocols. In order to provide a more detailed analysis of proposed protocols and for the sake of completeness, we performed mechanical formal analysis using the CasperFDR.

Contents

List of Figures

List of Tables

List of Abbreviations

AAC	Application Assurance Certificate
AAM	Application Abstract Machine
ABPD	Application Binding Protocol - Distributed
ABPL	Application Binding Protocol - Local
AID	Application Identifier
ALP	Application Lease Policy
AMS	Application Management Server
API	Application Programming Interface
ARM	Application Resource Manager
ASAS	Application Services Authentication Server
ATP	Attestation Protocol
CAMS	Card Application Management System
CASC	Coopetitive Architecture for Smart Cards
CC	Common Criteria
CDAM	Cross-Device Application Sharing Mechanism
CDAS	Cross-Device Application Sharing
CIB	Card Issuing Bank
CR	Cardholder's Requirement
CRP	Challenge-Response Pair
DAP	Data Authentication Pattern
DSA	Digital Signature Algorithm
GR	General Requirement
ICOM	Issuer Centric Smart Card Ownership Model
IMA	Integrity Measurement Authorisation
JCRE	Java Card Runtime Environment
JCVM	Java Card Virtual Machine
JVM	Java Virtual Machine
MNO	Mobile Network Operator
MPM	Mobile Phone Manufacturer
MTM	Mobile Trusted Module
NFC	Near Field Communication
PAC	Platform Assurance Certificate
PAN	Primary Account Number
PBP	Platform Binding Protocol
PRNG	Pseudo-Random Number Generators
PRNG	Pseudorandom Number Generator
PUF	Physical Unclonable Function
RAS	Remote Application Server
SAP	Service Access Point
SCM	Smart Card Manufacturer
SCOS	Smart Card Operating System
SCP	Secure Channel Protocol
SCR	Smart Card Requirement
SCRT	Smart Card Runtime Environment
SCWS	Smart Card Web Server
SIO	Shareable Interface Object
SOG	Security and Operational Goal
SP	Service Provider
STCP	Secure and Trusted Channel Protocol
TEM	Trusted Environment & Execution Manager
TPM	Trusted Platform Module
TSM	Trusted Service Manager
TSO	Transport Service Operator
UCOM	User Centric Smart Card Ownership Model
UCSC	User Centric Smart Card
UCTD	User Centric Tamper-Resistant Device
VSC	Virtual Smart Card

Chapter 1

Introduction

Contents

In this chapter, we discuss the past, present, and possible future of smart card technology and its operational infrastructures. We explain the motivation behind the thesis and the contribution it makes to the user centric approach to the management of tamper-resistant devices. The chapter concludes by outlining the structure of the thesis and briefly describing the contents of each of the subsequent chapters.

1.1 Setting the Scene

We open the discussion in this chapter by exploring the evolution of the smart card from its beginnings to the present. This is followed by a discussion of the reasons for having a user centric approach to the management of a security-critical device like a smart card and the challenges this approach involves. We then discuss the contributions of the thesis, and outline its structure.

1.2 A Brief History of Smart Cards

Card-based transactions originated in the USA, starting with a system which came to be known as *metal money*. This was a metal card issued by Western Union[1] as part of a deferred payment scheme [2]. In 1946, John Biggins, a banker at Flatbush National Bank of Brooklyn, issued a banking card to his customers called *Charg-It* [3]. Customers used their *Charg-It* cards to pay for groceries at local shops. In 1951, New York's Franklin National Bank issued the first credit cards [4] to gain a competitive advantage over rival banks. During the same period, an exclusive club known as the *Diners Club* issued the first plastic cards [5]. These cards reflected the high status of the individuals who used them. Instead of using cash, cardholders would use these cards to pay for services at selected hotels and restaurants. This was the beginning of plastic money as we know it; however, the rapid proliferation of plastic cards came when Visa[2] and MasterCard[3] entered the field [5].

These early cards spread from the USA to Europe and within a few years to the rest of the world. They had a very simple mechanism to store user-specific data and secure it against forgery. These cards carried the name of the cardholder and a unique card number printed or embossed on the card along with the card issuer's logo and a signature panel. The signature panel was used as a security mechanism to link the card to its cardholder. When used at a merchant's premises, the merchant had to verify the printed/embossed features of the card and ask the cardholder to sign the receipt. To verify the cardholder's right to use the card, the merchant could then match the signature on the receipt with the one on the signature panel [5]. The system relied heavily on the competence of the person at the Point of Sale (POS). This system worked for a while on a limited scale, but as the use of plastic cards increased, banks soon realised that a machine-readable and automated system would benefit all parties including cardholders, merchants, and banks [6].

[1] Western Union is a US-based financial company that provides person-to-person money transfer, business and commercial services.

[2] Visa: Trademark of Visa Inc, San Francisco, California, USA. A global payment technology and transaction management company that provides financial services to banks.

[3] MasterCard: Trademark of the MasterCard Worldwide that provides technology and architecture to support the relationship between financial institutions, merchants, and consumers for monetary transactions.

The next big innovation in the plastic card's evolution was the introduction of magnetic stripe cards. The magnetic stripe was used by banks to store digital information regarding the card and its cardholder that supplemented the visual features of the card. The storage of data on the magnetic stripe, along with the introduction of magnetic stripe readers at each merchant's site, automated the payment process and eliminated the tiresome handling of paper receipts. A feature of this innovation that outlived the magnetic stripe initiative and superseded the cardholder's signature is referred as a Personal Identification Number (PIN), which was used to identify the cardholder [6]. At a POS, a cardholder had to provide her card along with the PIN. If the card-issuing bank verified the PIN, the transaction would go ahead [5, 7]. These cards are still used in many places around the world, especially as student cards, hotel room keys, and rail tickets.

In subsequent years, the use of magnetic stripe cards started to strain the banking infrastructure. There were two reasons for this: first, a malicious user with suitable equipment could copy, modify, or write new data values onto the magnetic stripe; second, the early use of magnetic stripe cards used online connections to connect to the card-issuing bank's computer system (back-office system). This incurred substantial costs for data transmission, which in most cases were paid by merchants. In addition, requiring an online connection with the back office system put extra demands on the availability of the payment-by-card service. Remote areas and international call dialling rates soon decentralised the payment processing systems managed by two big American financial services companies, VISA and MasterCard. However, even the introduction of local points of clearance for payments did not make things much easier for merchants, and they still had to bear the burden of calling the transaction clearance server of the bank that issued the card.

A minor improvement to the magnetic stripe cards came in the shape of optical-storage (holographic) cards that provided a much larger storage capacity. However, the costs of manufacturing and writing or reading data from the optical-storage cards were higher and they still had the same shortcomings as magnetic stripe cards [8], so there was no particular incentive to change over to them.

The next breakthrough in the card-based services sector came in the 1970s, not from the USA but from Germany and France. This breakthrough was fuelled mainly by progress in microelectronics, which led to the ability to build a single silicon chip with different logical components to store and process logic data, revolutionising the card industry.

Nevertheless, it took a decade before chip-based cards were widely deployed; the French Postes, télégraphes et téléphones (PTT) first used them as telephone cards [5]. German telephone cards soon followed. These deployments provided a testing ground for the new technology, which was later exported to other industries, as chip-based cards provided much greater reliability and security than the magnetic stripe or holographic cards. Initially,

chip cards known as memory cards were based on fixed logic and limited storage capacity. However, later in the 1990s microprocessor cards emerged on the scene. These cards can store and dynamically process information without relying on hard-wired fixed logic, as was the case in early chip cards. The German Post Office conducted initial trials of chip cards for their analogue mobile telephone network. The success of these trials resulted in the deployment of the microprocessor cards in the GSM[4] networks. At the time, telecommunication companies all over the world were rapidly adopting microprocessor cards, mainly to prevent phone cloning. However, the banking networks of the time did not embrace the new technology as quickly.

The development of smart cards coincided with another revolution in the field of system security. The discipline of cryptography was emerging from government and military secrecy. The security provided by sophisticated (cryptographic) mathematical concepts, and improved designs in hardware and software programming paved the way for the use of cryptography in new technologies such as smart cards. This gave smart cards an edge over magnetic stripe cards in the banking sector, and this was soon acknowledged [5]. As in the case of the initial innovation of chip card technology, French banks pioneered the adoption of the smart cards as payment cards. After long negotiations and development, the widespread adoption of smart cards in the banking sector came in 1994, when Europay, MasterCard and Visa published their payment card specification (i.e. the EMV specification [9]).

The initial attempts to have multiple functionality on a smart card were made in 1996 in Austria with the introduction of a smart card, which allowed banking (e.g. POS services), an electronic purse and optional value-added services [5]. However, this initiative cannot be considered a true multi-application smart card, because it was a multi-functional smart card [10] that had a single application with multiple functionality.

At the same time, another concept termed the generic soft mask [5] was taking centre stage. In "generic soft mask" a card manufacturer implements a Smart Card Operating System (SCOS) on a non-mutable memory of the smart card. This operating system is independent of applications like banking or transport. To support these applications, the card manufacturer implements the Application Programming Interfaces (APIs) to facilitate individual application. These APIs were stored on the mutable memory rather than on a non-mutable memory where traditionally the bulk of the SCOS was stored. This innovation simplified the development of smart card applications: card manufacturers proposed using generic soft masks for different types of applications. Implementing the concept of the generic soft mask requires a minimum operating system and some customized Application Programming Interfaces (APIs) for any particular application. The application developers utilised these APIs to develop their applications.

[4]Global System for Mobile Communication (GSM) is a standard for the mobile Telecom industry that is developed and promoted by the GSM Association (GSMA).

The introduction of the soft mask also enabled the smart card developers to have a single smart card which had multiple application. These were fundamentally different from multi-functional smart cards because each of the functionalities/services had a separate application in the smart card. One example of an initial soft mask-based multiple application smart card is the French banking card. It had the old B0' application [11], EMV [9] banking application and a French (electronic) purse called Moneo [12, 13]. When a smart card user presented his/her smart card at a terminal, the card first checked whether or not the terminal supported EMV. If it did not, then it could opt for the B0' French banking application. Although these smart cards had separate functional applications, we cannot term them true multi-application smart cards because of the rigidity of the smart card architecture. Once these smart cards were issued, not even the card issuers could update them or install new applications on them. Therefore, how can we define a true multi-application smart card?

A multi-application smart card is one that supports the features listed below [5, 6, 14, 15]:

1. A separate context for each application on the card (e.g. storage and execution isolation), ensuring a secure and reliable application segregation mechanism.
2. Post-issuance application installation, deletion, and management (update/modification).
3. The ability for terminals to select an application directly and independently of other on-card applications.
4. The management, updating, modification, and deletion of each application without affecting other applications.
5. Delegation of the management of an application to an entity, which is not necessarily the card issuer. If an application is managed by such an entity, then the card issuer cannot access the application context. The only possible authority a card issuer might have is to block and/or remove the application without accessing its contents.
6. Secure and reliable inter-application communication.

A large number of smart cards deployed today are single task devices which can only execute one application at a time, and do not support the simultaneous execution of multiple applications. However, innovation in the hardware design and in the SCOS/platforms have begun to explore the concept of multi-threading [16]. These developments will surely make smart cards into powerful and secure computing devices which can support different tasks concurrently.

Over the last ten years, smart card technology has rapidly moved out of its traditional businesses such as banking and telecommunication. It has spread to transport, health care, identity cards, travel cards, access control, leisure passes, and other fields. Smart cards are becoming synonymous with everyday activities and they are deployed in almost every aspect of modern life. In the smart card industry the technology has changed its shape regularly. It has moved from plastic cards to magnetic stripes to chip cards and from single application to multi-application smart cards. Each innovation has taken approximately ten years to arrive and become commercially viable. With each new step, the emphasis has been on greater flexibility, operability, security and reliability, and on the value that smart cards bring to customers and businesses.

1.3 Motivation and Challenges

A wide range of computing devices are being introduced that can perform the same tasks. For example, traditionally mobile phones were only for voice communication and later, for text messaging. However, when it comes to Internet access the advent of smart phones has blurred the line between a desktop computer and hand-held devices. In addition, computing devices like tablets are making headway in market; thus general-purpose computers, mobile phones, and tablets are providing similar services. This multiplies the potential for customers to become victims of security breaches or privacy violations as their data is on multiple computing devices with different platforms and varying levels of security safeguards.

At the same time, the smart card industry, that until the start of the 21st century was reluctant to adopt the multi-application smart card initiative, is considering a convergence of different services on to a single device. The idea of multi-application smart cards has been well known since the late 1990s but after its initial advocacy, it did not gain any momentum. However, recent innovations such as the Near Field Communication (NFC) [17] technology and secure elements[5] in mobile phones have set off a renewed interest in multi-application smart cards. The NFC enables a contactless data exchange between a chip (i.e. a smart card) and a terminal. It is also extended to enable mobile phones to emulate contactless smart cards. As a result, an NFC-enabled mobile phone can use the existing infrastructures of different industries (i.e. banking, transport and access control) that support contactless smart cards.

The security and privacy concerns are increasing with the increasing number of different portals (i.e. devices) through which users are accessing associated data/functions. There

[5] A secure element is an electronic chip which can securely store and execute programs. Examples are the Universal Integrated Circuit Card (UICC), the Embedded Secure Element, and Secure Memory Cards. Throughout this thesis, the terms "secure element" and "smart card" are used interchangeably.

are several proposals to provide a hardware-based security and privacy protection, and they differ in operation and capability from one computing device to another. What we mean by this is that the proposal becomes specific to the target computing device for which the trust, security, or privacy architecture is proposed. For example, the difference between the Trusted Platform Module (TPM) [18] and Mobile Trusted Module (MTM) [19] is that they target two different computing devices, namely general-purpose computers and mobile phones respectively. Similarly, to provide protection to mobile devices including mobile phones and tablets, architectures like AEGIS [20], ARM TrustZone [21], M-Shield [22], GlobalPlatform Trusted Execution Environment (TEE) [23] are proposed. These proposals, along with the TPM and MTM, have created a wide range of options that provide the same services namely trust, security, and privacy.

Having a wide range of choices is encouraging, but it also means that a Service Provider (SP) that offers a service that requires trust, security and privacy support has to implement or support a wide range of technologies. For example, an internet identity application could be on a desktop computer and mobile phone. For a desktop computer, the protection technology might use TPM and for a mobile phone, it might depend on MTM or M-Shield. This diversity could not only create complexity for the SP to manage and provision its services but also for the consumer to use different architectures on individual devices. Furthermore, most of the proposed architectures like TPM or MTM are physically bound to the computing devices, thus reducing the ubiquity and inter-operability of the same services on different devices. The same is true if a user has an identity application as part of her smart card. Therefore, a user that has a computer, a mobile phone, a tablet and smart cards, will be using the same service on each device in isolation.

A possible solution might be to have a device that can support a unified, ubiquitous, and interoperable architecture for security and privacy services incorporated across different computing environments (e.g. desktop computers, embedded devices, tablets, and mobile phones etc.). We refer to such a device as a User Centric Tamper-Resistant Device (UCTD). Application developers, whether they are targeting smart cards, hand-held devices and/or traditional computing devices, can utilise the UCTD architecture that provides a single unified framework. The reason we focus on having a user centric architecture is to provide maximum interoperability, flexibility, and integration between different services and operational architectures.

The UCTD provides hardware level security coupled with a robust software platform that will enable an SP to design their services for a single platform (i.e. a UCTD) so that consumers can use the service seamlessly on any of their computing devices. For example, a banking application on a UCTD can provide a secure online payment scheme for a computer, mobile phone or tablet along with provision to pay at a POS or withdraw money at an Automated Teller Machine (ATM). The bank does not need to worry about which

computing device the consumer is using. The user and the bank get security and privacy protection from the UCTD regardless of the device (e.g. computer, mobile phone, tablet, and POS, etc.) from wherever they are connecting to the payment network. Therefore, feature-rich computing devices can have applications that rely on the services provided by the UCTD to enable a security- and privacy-preserving framework.

For such a device, we consider that smart cards offer the most promising architecture. In our opinion, the rigorous design and analysis constituted in the smart card industry can benefit other computing environments by providing security, privacy, and reliability services. If we port the smart card architecture as a generic tamper-resistant device that can interface with different computing environments then it can provide a ubiquitous, interoperable, flexible, dynamic, secure, and reliable architecture that can store and execute security- and privacy-sensitive applications. In this thesis, we use the term smart card as inclusive of the technology (both hardware and software architecture) and without any restriction on form factors. The smart card architecture can only be realised as a UCTD if the associated ownership issues are resolved.

The most prominent ownership model in the smart card industry is centred on the organisation, which acquires smart cards from card manufacturers and issues them to the customers. Such organisations are referred to as card issuers and in this thesis this ownership model is called the Issuer Centric Smart Card Ownership Model (ICOM). This model provided much needed momentum in the smart card industry, driving the technological and infrastructural improvements to provide better, more secure and reliable services to customers. It also enabled the initial motivation for standardising the smart card technology (e.g. ISO 7816 [24], ISO 14443 [25]) and its applications for specific fields (e.g. GSM [26], EMV [9] and ITSO [27], etc.).

The ICOM architecture is restrictive and might not be suitable for smart cards if they are to be adapted as UCTDs. Therefore, we propose a model that provides a more flexible, and dynamic platform which also gives control of the smart card to its users. This model is referred to as the User Centric Smart Card Ownership Model (UCOM). The term *ownership* (control) in the proposed model means "*freedom of choice*" that gives a UCTD owner the privilege of installing or deleting any application as they desire. However, this does not mean that they have the ownership of individual applications installed on the device [10]. The application(s) installed on the smart card will always be under the total control of the application issuers (i.e. the SPs) and the user will be entitled to use these applications under sanction from their respective SPs. Furthermore, the choice about whether to lease an application to a card (user) resides solely with the relevant SP. Therefore, we can define a UCTD as a device whose architecture is based on the smart card technology that supports the UCOM framework.

The challenges presented to the UCOM proposal are rooted in the history of the smart card technology. Any architecture or framework proposed in the smart card industry was designed with the underlying requirement of supporting centralised control (i.e. ICOM). Most notable examples are Java Card [28], Multos [29], and GlobalPlatform [30]. The design of these technologies is based on the strong assumption that the smart cards will always be under the control of a trusted centralised authority.

When we move the smart card ownership to its users, the traditional notion of trust does not hold. Therefore, most of the ICOM-based architectures did not provide the same level of security and reliability that compelled us to choose smart cards for the UCTD proposal. However, we do not propose that existing well-established architectures like Java Card and GlobalPlatform are incapable of supporting a user centric approach. What we propose is that they require some modifications that will enable them to support an architecture that supports the user ownership of the smart cards. In this way, UCOM not only makes a smart card into a general-purpose security device (i.e. UCTD) [31] but also resolves the ownership issues in the smart card industry that are hindering the adoption of having multiple services from different organisation on a single smart card [32].

Furthermore, for the UCTDs, to support feature rich environments (e.g. desktop computers) while supporting only a single user might not be sufficient. Some of these environments might include administrative oversight as part of the corporate administration, or parental control. Therefore, we extended the UCOM architecture to enable administrative management while strictly adhering to the user ownership, platform security, and reliability requirements. This extension is referred as the Coopetitive[6] Architecture for Smart Cards (CASC).

In the rest of the thesis, the terms UCTD, smart card, and secure element will be used interchangeably unless specified otherwise.

1.4 Contributions

In this thesis, we set out to analyse whether user ownership is technically and operationally possible for a tamper-resistant device based on the smart card architecture. After showing that it is possible, we consider what changes must be made to traditional smart card architecture and service infrastructure. These questions are the focus of this thesis.

The contributions of this thesis are spread over different stages of the UCTD lifecycle, from

[6]The term coopetitive is borrowed from game theory [33]–[35]; where it stands for arrangements in which competitors collaborate with each other to share the common cost and compete where they see that they might have competitive advantage.

the UCTD manufacturing, to the application download and execution, to decommissioning at the end of the UCTD's lifetime. The main contribution of the thesis is the development of a user centric framework for tamper-resistant security-sensitive devices. To accomplish this, we propose several changes to existing smart card architecture, including changes to the application management framework, the application download protocols, the smart card firewall mechanism, and finally the application execution environment.

We propose a new architecture for the smart card platform, including the remote attestation and security assurance mechanism. These changes will enable an SP to ascertain whether the current state of a platform is trustworthy. Furthermore, we propose a framework to securely backup the contents of a smart card and restore them (when required) to any other smart card. This latter mechanism allows a rapid recovery if the existing smart card is stolen or corrupted (i.e. cannot work), and also facilitates migration from one device to another.

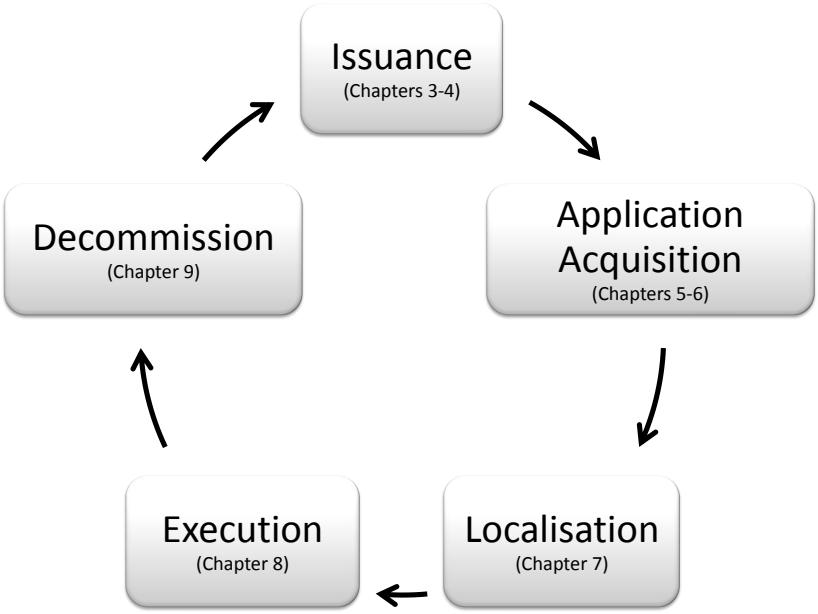

Figure 1.1: Life cycle of UCTDs in relation to a user and an application

The life cycle stages of a UCTD in relation to its provision of different functionality or features to the respective user or SP are shown in figure 1.1. Each depicted stage of the UCTD also has a corresponding chapter or chapters in the thesis.

The first stage in the life cycle of a UCTD is the issuance phase that includes manufacturing and issuance of the UCTDs along with the ownership acquisition by the users.

In the second stage referred as "Application Acquisition", the user requests an SP to lease its application(s) to the acquired UCTD. This process encompasses the dynamic establishment of a trust relationship between the SP and the UCTD along with the downloading of the (requested) application.

The next stage is the "Localisation" stage, in which the downloaded application registers its services with the UCTD. In addition, if the downloaded application shares resources with other (installed) applications then it will also establish an application sharing relationship with them.

Once an application is localised, it will utilise the runtime environment to execute its services. This phase of the UCTD in relation to an application is termed as the "Execution" phase.

Finally, at the end of the life cycle of a UCTD, an application retires from service, which might be due to damage to the device, loss, theft, or to the user acquiring a more feature-rich UCTD. This phase is termed as "Decommission" and it requires the inclusion of architecture which can take a backup of the existing contents of a UCTD and transfer it to another UCTD.

During the course of this thesis, the term UCTD is used to indicate a tamper-resistant device, whereas, UCOM refers to the ownership model that we have proposed for the UCTDs. Furthermore, when we refer to an honest user we use third-person singular pronoun "she" and for a malicious user we use "he".

1.5 Structure of the Thesis

The remainder of this thesis is structured as follows:

In chapter 2, we begin the discussion by emphasising that tamper-resistant devices can provide a secure, reliable, and trusted execution environment even when the device is in the possession of an adversary. With the ever-growing use of different computing devices (i.e. mobile phones, tablets, and embedded devices), the potential for compromising the security and privacy of an individual is increased. The Trusted Platform Module (TPM) is restricted to integrity measurement and cryptographic operations, which is crucial in its own right. On the other hand, smart cards provide a general-purpose execution environment, but traditionally they are under centralised control, which if extended to generic tamper-resistant devices may not be appropriate. Therefore, in this chapter we analyse the rationale for a general-purpose user centric tamper-resistant device based on the smart card architecture, and its applications in different computing environments.

Chapter 3, opens the discussion with the ICOM and extends it to include the security and operational assumptions adopted in this model. Next, we provide a succinct description of the widely deployed smart card frameworks that support the ICOM. After discussing the

traditional smart card ownership model and associated architectures, we move on to discuss the concept of giving control of application selection to cardholders. We then discuss the major components of the UCOM framework.

To provide a secure and reliable architecture for the UCTD we need to make adequate modifications to the smart card platform. Therefore, in chapter 4, we discuss the security and operational architecture of the UCOM-supported platform, termed as the User Centric Smart Card (UCSC). We detail the reasons behind the architectural changes to the traditional smart card which are needed to accommodate the philosophy of the UCOM. Subsequently, we discuss the mechanism that provides security assurance of the UCOM platform to the requesting entity. We also describe the ownership acquisition process through which a user takes ownership of an UCSC and how she can verify the claims articulated (e.g. assurances about security and reliability) by the UCSC.

After describing the smart card platform architecture, we move to a description of the framework that supports the application acquisition and management. Hence, in chapter 5, we discuss the card management architectures that are widely accepted and deployed in the smart card industry: GlobalPlatform and Multos. We explain why these architectures are not fully compatible with the user centric architecture. Subsequently, we describe the card management architecture for the UCOM. Finally, we discuss two new issues raised by the proposed architecture.

Chapter 6, begins with a discussion on the secure channel protocols that are used for entity authentication and key establishment. We discuss the security and operational goals that a secure channel protocol has to accomplish in the UCTD environment. Subsequently, we discuss different protocols which have been proposed for Internet and smart card environments, and these protocols are used to provide a comparison to the ones we propose. We propose two protocols that closely adhere to the UCOM philosophy and a protocol related to the CASC model. An informal analysis is provided of all proposed protocols. For the sake of completeness, we subject the proposed protocols to mechanical formal analysis using CasperFDR. Finally, we discuss the test implementation and performance measures of the proposed protocols.

After an application is installed on a smart card, it might want to communicate with other applications or services available on the card. To do so, an application will utilise the provision of an application sharing mechanism. In chapter 7, we begin the discussion with a description of the two contrasting frameworks for application sharing deployed by Java Card and Multos, followed by an explanation for why we need to extend the existing techniques for the UCOM framework. Subsequently, we discuss the architectural framework of an application sharing mechanism for the UCTD. Later, we extend the proposed application sharing mechanism between applications installed on different UCTDs, referring

to it as Cross-Device Application Sharing (CDAS). The application sharing mechanism for UCTD requires entity authentication and trust validation, along with key generation to secure the sharing of resources between applications. To do so, we propose adequate protocols that accomplish the listed goals of the UCTD application sharing mechanism. Furthermore, we provide an informal analysis of the protocols along with a comparison with existing protocols. Subsequently, mechanical formal analysis based on the CasperFDR, and the test implementation experience, is presented.

Once an application is installed, and has registered itself with different applications and platform services, it will execute to provide services to the user. Therefore, chapter 8 discusses the UCTD execution environment in which the downloaded applications will execute. We articulate the threat model for the execution environment in the ICOM architecture. We then examine the potential aggravation of the threat model to the proposed UCOM because of its openness. Subsequently, we look at countermeasures that can be deployed to provide a secure and reliable execution platform. The discussed countermeasures are then analysed in terms of their suitability and performance.

In chapter 9, we analyse the content backup and restoration mechanism that allows a user to securely backup her smart card. This mechanism enables a user to retain the same set of applications if she loses her smart card or wants to move to a new one. Subsequently, we detail the application deletion process that ensures that an application is removed without affecting the reliability of the UCTD platform.

Finally, in chapter 10, we conclude the thesis by summarising its contributions and providing suggestions for future work.

Chapter 2

User Centric Tamper-Resistant Device

Contents

In this chapter, we begin by discussing the notion that tamper-resistant devices can provide a secure, reliable, and trusted execution environment even when they are in the possession of an adversary. We survey security and privacy issues in different computing environments including smart cards, mobile devices, and traditional computers. Subsequently, we analyse the rationale for a general-purpose user centric tamper-resistant device based on smart card architecture, and its applications in different computing environments.

2.1 Introduction

The adoption of mobile phones and tablet-based computing platforms (e.g. iPads) is increasing. To some extent, the security and privacy issues of personal computers, including insecure execution environments, also apply to hand-held devices. As reliance on these devices increases, so will threats to the security and privacy of the platform and its users. For example, a healthcare mobile application, if it is badly designed and gets compromised, it may reveal user's sensitive medical information.

A possible solution is to have a tamper-resistant execution environment that executes a program in a trusted, secure, reliable, and fault-tolerant environment. Among widely deployed tamper-resistant devices, two are most prominent: Trusted Platform Module (TPM) [36], and smart cards [5].

The TPM provides a platform's integrity measurement with cryptographic protection in contrast to smart cards that provide a generic execution environment, in which an application can execute and store application code and data. This landscape maps from smart cards, mobile phones, tablets and general-purpose computers through to Machine-to-Machine communication and the Internet of Things [37]. It would be beneficial to have an interoperable unified architecture that provides a secure and reliable execution and storage environment for different computing devices.

Structure of the Chapter: In section 2.2, we discuss the rationale for having a secure, reliable, trusted, dynamic, and ubiquitous architecture for a generic tamper-resistant device that is under the user's control. Such a device is referred as a User Centric Tamper-Resistant Device (UCTD). Section 2.3 briefly surveys the proposals that provide secure and trusted services to different computing devices. In this section, we also compare the discussed devices for their suitability as UCTDs. Subsequently, in section 2.4 we discuss the reasoning behind the choice of smart card architecture for a proposed unified architecture based on UCTDs. Finally, section 2.5 provides three case studies based on the adoption of the UCTD in different computing environments.

2.2 Rationale for a User Centric Tamper-Resistant Device

The motivation for having a generic tamper-resistant device that is under the control of its user rather than a centralised authority comes from three distinct but interrelated computing fields, discussed individually in subsequent sections.

2.2.1 Smart Card Environment

As pointed out by Porter [38], the crucial elements that stimulate competition and innovation in an industry can be: a) the threat of new entrants, b) the threat of substitute products or devices, and c) consumer power (culture). For the smart card industry, these elements are present in a multitude of forms. The provision of having applications on a mobile phone has enabled new entrants to venture into the traditionally monopolised industries like the payment sector. Companies like PayPal, Google or any other third party can offer a mobile payment service. In addition smart phones, with inclusion of Near Field Communication (NFC) functionality, can provide a substitute for traditional smart card applications like transport ticketing and access control [39]. Technology savvy consumers require more features on a device, a need [40], which is successfully fulfilled by high-end smart phones (e.g. the iPhone). Smart cards are lagging behind in providing such possibilities. Nevertheless, the NFC technology provides an opportunity for the convergence of different services on a single smart card.

In NFC trials around the world [41], the prominent framework that is deployed is an extension of the ICOM model and is referred as the Trusted Service Manager (TSM) [42]. It has gained support from the banking and telecom sectors [43, 44].

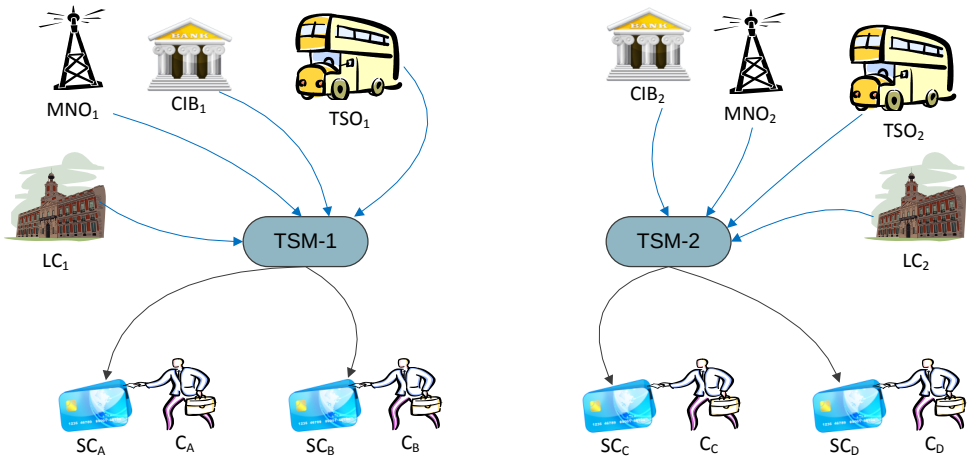

Figure 2.1: Trusted Service Manager (TSM) architecture

The TSM architecture is illustrated in figure 2.1 in which we have two TSM networks: namely TSM-1 and TSM-2. Each network has a Mobile Network Operator (MNO), a Card Issuing Bank (CIB), a Transport Service Operator (TSO) and a Leisure Centre (LC). A customer C_A receives a smart card (SC_A) from the TSM-1. The customer C_A would only be able to have applications on the SC_A from the MNO_1, CIB_1, TSO_1, and LC_1. However, if C_A does banking with the CIB_2 that is associated with TSM-2 then she has to either acquire a new smart card from TSM-2 or change banks, effectively creating market segmentation.

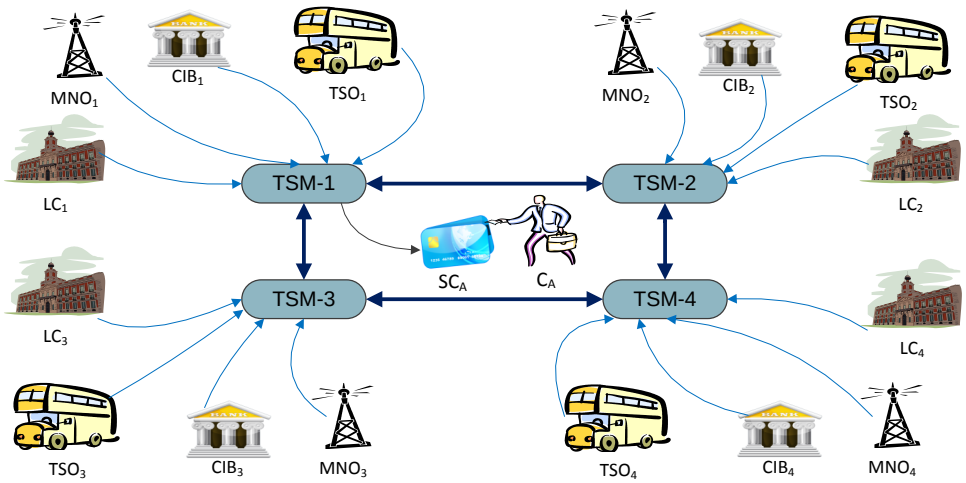

Figure 2.2: Possible interaction between TSMs for scalability

One possible option to reduce market segmentation is to have all application providers maintain a relationship with all or most of the TSMs. However, this option might not be practically feasible. We propose to resolve market segmentation by introducing a syndicated scheme, which can be termed as the Dynamic Contractual Syndicated TSM (DCS-TSM) in which multiple TSMs participate to provide services to their customers. In this model, a user (C_A) can request to install an application from an application provider (e.g. MNO, CIB, TSO, and LC) that is a member of any TSM, which participates in the DCS-TSM. The application installation is still authorised by the scheme manager — for example, TSM-1 in figure 2.2 is the scheme manager for the SC_A as it has issued the smart card to the customer C_A. It could be argued that this scenario is workable, but the DCS-TSM framework also suffers from limited scalability, flexibility, and ubiquitousness.

The limited scalability arises because a) not all application providers can establish or manage a relationship with every possible TSM and b) not all TSMs will be part of the same DCS-TSM. In addition, to be part of a collaborative scheme offered by a TSM, application providers might be required to pay a subscription fee. Therefore, small or medium scale organisations like local libraries, universities, and health centres may not be able to afford to be associated with a TSM. We consider that such a barrier would reduce a scheme's flexibility. Furthermore, it lacks true ubiquitousness as different countries might opt for having their own independent TSMs. Therefore, tourists or business travellers would face difficulties in acquiring applications (i.e. applications from a TSO) in a foreign country. Other issues include ownership privileges, customer loyalty, customer relationship management, card surface marketing, and potential revenue generation opportunities that are discussed in [11, 32, 45, 46].

Innovation and success in a competitive environment is dependent on the core competence of an organisation [47] in a particular field (e.g. business, technology and culture). There is no universally accepted concession on who should be taking the role of the TSM from

possible contenders such as smart card manufacturers (SCMs), MNOs, CIBs, mobile phone manufacturers (MPMs) and independent/trusted third parties (e.g. post office). With reference to trust and brand awareness, SCMs do not have a market presence, as since the inception of the smart card technology their brand has seldom been part of the final product. Whereas, the core competence of MNOs or CIBs is not chip manufacturing, but a strong branding and an existing customer base. The MPMs can extend their core competence to secure-element designing/manufacturing, and they also have a strong brand and customer base. Nevertheless, no one has a clear competitive advantage. There is an underlying fear that this entire process might be the repeat of the multi-application smart card initiative, which inspired an initial fervour that later died down due to the conflicting business objectives of different organisations. In this entire process, one stakeholder that is crucial to the survival of all other entities in the ecosystem is missing: the users (consumers) of the system, which we consider might be a gross oversight. An amicable solution to all stakeholders could be the UCOM initiative.

2.2.2 Hand-held Devices

In this thesis, we use the term hand-held devices to refer to mobile phones and tablets. The reason for grouping them together is the similarity in the application lifecycles of these devices and a growing convergence between their form-factors and underlying platforms.

The mobile phone platform has come a long way from being just a medium of communication. It has developed into a social construct that has affiliations and emotional attachments for individual users along with being an entertainment hub, and a medium to connect with the world through social media sites [48, 49]. With the ever-increasing trend of convergence of different technologies/services in smart phones, they are becoming attractive targets for adversaries who want to compromise the security and privacy of users.

The so-called "App Culture" promoted by Apple Inc., which enables users to seamlessly download any application they desire has opened up the mobile phone application market to a wide range of companies [40]. New ideas are being tested; for example, Starbucks customers can pay for coffees using a "Starbucks' Card Mobile App" on their iPhones. This indicates that there can be additional services/organisations which develop mobile applications that perform sensitive processing like banking or healthcare, which have traditionally required a strong security and privacy architecture. Predominantly, mobile phone platforms are not extensively evaluated for their security and privacy services, as is normal in high-end smart cards. In addition, most of the smart phones do not have a tamper-resistant execution environment [50] (except for the secure element). In addition, lack of Mobile Trusted Module (MTM) adoption leaves application developers with no choice but to de-

velop the applications that will run on a non-evaluated and possibly insecure/compromised device.

If we analyse the mobile phone environment, the application download concept resembles the ICOM model. For example, on an iPhone an application cannot be installed unless it is in conformance with the Apple's stated regulations[1] that are enforced by mandatory review by the Apple's App Store. However, it has been possible to write malicious applications that can bypass the Apple's App Store review [51]. Therefore, a conservative view of handheld devices will see them as potential military, corporate espionage, and civilian attack targets. UCTDs will enable an application developer for handheld devices to store and execute security- and privacy-related code and data on a secure and trusted device.

Likewise, tablet devices are gaining market share and have a similar product lifecycle to a mobile phone. Therefore, the possibilities and issues we discussed above regarding the mobile phone platform are also true for the tablet platform.

2.2.3 Traditional Computing Devices

These computing platforms are used in personal and corporate spheres, and they provide access to a wide range of services. Most of these services require the security and privacy of the user, and SPs (i.e. banks, and corporate servers, etc.) need to be able to authenticate the user who is accessing their services. It can be argued that the Trusted Platform Module (TPM) provides an adequate security and privacy service. We are not contesting this notion — we are suggesting that by having a tamper-resistant device that can store applications and execute them within its bounds, can go a step further and provide a secure execution environment that individual applications can utilise for their security sensitive code. The problem with this proposal is that most of the tamper-resistant devices capable of executing applications are under centralised control (as in the smart card industry) [11, 32], or they are stripped down to cryptographic services [18, 20, 21, 52, 53]. A UCTD avoids such issues while providing an open and dynamic execution environment.

2.3 Candidates for User Centric Tamper-Resistant Device

In hand-held and traditional computer devices, proposals like TPM [36], MTM [19], AEGIS [20], ARM TrustZone [21] and GlobalPlatform Trusted Execution Environment (TEE) [54, 55], already exist. Most of these devices: a) are limited to a particular computing

[1] There are other ways of installing applications on a smart phone (e.g. iPhone) and most of them are referred as jailbreaking [51]

environment (e.g. TPM [18] and MTM [19]), b) only provide execution protection (e.g. AEGIS), c) have limited application execution without user control (i.e. TEE), d) have limited scalability regarding the support for different application and platform scenarios, e) do not provide dynamic trust validation and assurance [56] and require an implicit trust, f) do not require third party (security) evaluation, and g) do not provide user ownership/control (e.g. smart cards [32]). We discuss these technologies individually below and analyse their suitability for the UCTD architecture in table 2.1.

2.3.1 Trusted Platform Module

The Trusted Computing Group (TCG) [36] started an initiative for providing a tamper-resistant device referred as the Trusted Platform Module (TPM) [18]. The mission statement of the TCG commits it to providing authentication, data protection, network security, and disaster recovery services [36]. A Trusted Platform Module (TPM) will measure the integrity matrixes referred to as Platform Configuration Registers (PCRs) that are securely sealed with cryptographic keys. If the TPM finds any discrepancies in the future integrity-measurements then it will flag the problem. A TPM does not decide whether this discrepancy is authorised by the user or whether it is due to a malicious entity.

A TPM is a tamper-resistant device with a low footprint that is utilised as a root of trust to support the trusted computing platform. The concept of trust as defined by the Trusted Computing Group (TCG) is the evaluation of platform results as expected by the requesting entity [36]. A TPM is not concerned with whether the evaluated state is secure or not as long as the evaluation result is trusted by the requesting hall. Therefore, we can say that a TPM is specifically designed (or restricted) to be a trusted component, which will be physically bound (soldered) to a platform. The fundamental function of a TPM is to provide secure, trusted, and tamper-resistant root of (trusted) measurements on which the integrity measurement of the rest of the platform is dependent. A TPM is typically under the control of the platform user, and it has a secure and reliable software/hardware platform. However, it is not a general-purpose execution environment in which an arbitrary code can be executed and neither is it portable, unless a smart card is used to behave like a TPM [57]–[59]. In this chapter, we treat TPM and MTM together even though there are subtle differences between them.

2.3.2 AEGIS

AEGIS is a single-chip secure processor that is designed to build trusted systems and is secure against physical and software attacks [20]. Therefore, we can consider AEGIS as a processor with a limited memory that stores processor identification information along

with possible cryptographic parameters (i.e. private key of the public key pair). AEGIS has two processing modes: Tamper-Evident Processing (TE) and Private Tamper-Resistant Processing (PTR). In the TE environment, AEGIS ensures the integrity of an executing program whereas in PTR it also protects the privacy of the code or data. One fundamental difference between TPM and AEGIS is that TPM relies on static integrity measurements whereas AEGIS provides a dynamic mechanism that measures an application's integrity at different stages of execution. It is apparent that TPM has better performance than AEGIS, and it can be argued that static integrity measurement is good enough for the job. Another distinguishing feature of the AEGIS is that it uses a Physical Unclonable Function (PUF) [60] to securely store the cryptographic keys inside the AEGIS processor chip [61].

2.3.3 ARM TrustZone

Similar to the MTM, the ARM TrustZone also provides an architecture for a trusted platform specifically for mobile devices. The underlying concept is the provision of two virtual processors with hardware-level segregation and access control [21, 62]. This enables the ARM TrustZone to define two execution environments termed as Secure world and Normal world. The Secure world executes the security and privacy sensitive components of applications and normal execution takes place in the Normal world. The ARM processor manages the switch between the two worlds. The ARM TrustZone is implemented as a security extension to the ARM processors (e.g. ARM1176JZ(F)-S, Cortes-A8, and Cortex-A9 MPCore) [21], which a developer can opt to utilise if required.

2.3.4 M-Shield

Texas Instruments has designed the M-Shield as a secure execution environment for the mobile phone market [22]. Unlike ARM TrustZone, the M-Shield is a standalone secure chip, and it provides a secure execution and limited non-volatile memory. Furthermore, it has internal memory to store runtime execution data [63] and this makes it less susceptible to attacks on off-chip memory or communication buses [64]. The memory and communication buses that we mention here are part of the platform, main memory and communication buses between a TPM and other components on a motherboard, rather than the on-chip memory and communication buses.

2.3.5 GlobalPlatform Trusted Execution Environment (TEE)

The TEE is GlobalPlatform's initiative [23, 54, 65] for mobile phones, set-top boxes, utility meters, and payphones. GlobalPlatform defines a specification for interoperable secure hardware, which is based on the GlobalPlatform's experience in the smart card industry. It does not define any particular hardware, which can be based on either a typical secure element or any of the previously discussed tamper-resistant devices. The rationale for discussing the TEE as part of the candidate devices is to provide a complete picture. The underlying ownership of the TEE device still predominantly resides with the issuing authority, which is similar to the GlobalPlatform's specification for the smart card industry [30].

2.3.6 Trusted Personal Devices

The term Trusted Personal Devices (TPD) was coined by the **In**tegrated **s**ecure **p**latform for the **i**nteractive T**r**usted P**e**rsonal **D**evices (InspireD) project [66]. Similar to our proposal, the architecture for the TPD is based on smart card technology. The architecture of the TPD is similar to that of the smart card, with the exception that is has different form factors that include SIM cards, Secure Digital (SD) cards, and Universal Serial Bus (USB) memory sticks [66]. However, the InspireD project recommended that the TPD to be under the ownership of a centralised authority (i.e. card issuer) and users get the privilege of choosing whether to use the device or not. Users cannot request installation or deletion of an application. Therefore, we can say that TPD was in conformance with the ICOM framework.

2.3.7 Comparative Analysis

In this section, we analyse three questions: i) why use a tamper-resistant device?, ii) why have a user centric ownership architecture?, and finally iii) why do we not just opt for the TPM (or other devices discussed above)?

In most of the scenarios, a tamper-resistant device is assumed to be in the possession of a malicious user [5, 6]. This assumption is natural for banking, transport, and healthcare cards. Therefore, a tamper-resistant device has a physical protection layer to avoid any intrusion attacks. In addition, these devices require an adequate hardware protection and self-protect mechanism to safeguard them from accidental or intentional damage. Therefore, a tamper-resistant device provides a secure and reliable platform that can remain trustworthy even in the possession of a malicious user. However, just focusing on the tam-

per resistance is not the complete picture when we discuss the UCTD and other measures related to application and platform design are also required to complement the hardware level protection [67].

The rationale for emphasising the user ownership of the UCTD is: a) to enable an open, dynamic, and ubiquitous system, b) individual developers (application providers) do not need to convince the scheme managers (as is required, for example, in the ICOM- or TSM-based models for smart cards [32, 68]) to gain permission to install their applications onto a UCTD, c) UCTD users (owners) will get the choice to install or delete an application, and finally d) to facilitate the interoperability and scalability of the UCTD framework (i.e. users can use their UCTDs in conjunction with any of their devices like mobile phones, tablets, and computers, etc.).

Finally, why not just use TPM? After all it is already in the user's control, the TPM specifications [18] require tamper-resistance and the TPM acts as a root-of-trust in hand-held and traditional computing devices. The rationale behind not choosing the TPM is: a) the TPM is designed to support the trusted computing platform initiative [18] that is focused on the integrity of the platform, rather than on an execution platform on which a general application code can execute, b) the design of the TPM is specific to a particular platform as there are two different specifications for traditional computers and mobile phones: TPM [18] and MTM [19] respectively, c) the basic functionalities of a TPM are protected capabilities, integrity measurement and reporting; it does not make decisions but merely reports the integrity measurements to the requesting entity, and finally d) TPM is required to be bound to the relevant platform.

Below is the list of requirements for the UCTD architecture that we use to compare different candidate technologies in table 2.1.

1. Execution protection: Defined commands related to security and privacy sensitive processing are executed in a secure and reliable environment.
2. Storage protection (Volatile): The device has a secure volatile memory on the chip to store temporary data and code related to the executing application.
3. Storage protection (Non-Volatile): The device provides non-volatile storage on the chip.
4. Tamper-resistant: The device provides tamper-resistant protection that is based on hardware techniques.
5. Tamper-evident: The device has the capability to detect potential tampering with the hardware and respond in a pre-defined manner.

Table 2.1: Comparison of different candidate devices for the UCTD proposal

Criteria	TPM	AEGIS	ARM TrustZone	M-Shield	TEE	TPD	Smart Card
1. Execution protection	Yes	Yes	Yes	Yes	Yes	Yes	Yes
2. Storage protection (Volatile)	-Yes	No	Yes	Yes	Yes	Yes	Yes
3. Storage protection (Non-Volatile)	-Yes	-Yes	Yes	Yes	Yes	Yes	Yes
4. Tamper-resistant	Yes	Yes	Yes	Yes	Yes	Yes	Yes
5. Tamper-evident	Yes	Yes	Yes	Yes	Yes	Yes	Yes
6. Scalability	Yes	Yes	Yes	Yes	No	No	Yes
7. Interoperable architecture	No	NA	NA	NA	Yes	Yes	Yes
8. Dynamic relation	Yes	NA	NA	No	No	No	Yes
9. User ownership	Yes	NA	NA	NA	No	No	Yes
10. Administrative architecture	Yes	NA	Yes*	Yes*	No	No	Yes
11. Open design	-Yes	-Yes	No	No	-Yes	-Yes	Yes
12. Secure execution platform	No	Yes	Yes	Yes	Yes	Yes	Yes
13. Independent security evaluation	Yes	No	No	No	No	-Yes	Yes

Note. In the above table, "Yes" indicates that that the device completely supports the criterion, "-Yes" means that the device generally supports the criterion but there are instances where it does not (e.g. in the case of criterion 16, (U)SIM are not required to be independently evaluated whereas in the case of EMV cards it is mandatory), "Yes*" means that the device can support the criterion with adequate design. The notation "No" means not supported, and "NA" means that the given criterion is not applicable as it is not the design requirement of the device.

6. Scalability: The architecture of the device is scalable so that it can provide services to any application or application provider, and does not require authorisation/authentication from a centralised authority.

7. Interoperable architecture: The architecture deals with the idea that the candidate device (e.g. TPM, smart card, etc.) can be interoperable with different computing devices (i.e. mobile phones, tablets, and personal computers).

8. Dynamic relation: A third party can establish a direct relationship based on the security and reliability of the device. The dynamic relation requires that an application provider can trust a device without requiring it to be part of a syndicated scheme (i.e. TSM or one adopted by Apple App Store, etc.) and vice versa.

9. User ownership: The device is in the control of its user and she can install, delete, and execute any application she desires.

10. Administrative architecture: The device also provides for administrative controls as might be required in a corporate network or in the case of parental control. This option is to accommodate different deployment scenarios. For example, an MNO can subsidise (locked) mobile phones under contract to a user; in this case, the MNO is an administrative authority that gives the user the privilege of using the mobile devices. The administrative architecture by no means restricts the user's *freedom of choice*; therefore, it is an extension of the UCOM (section 3.6).

11. Open design: The design should not be proprietary; it should be in the public domain.

12. Secure execution platform: The device allows the execution of an application code (from third parties) in a secure and reliable manner as long as it complies with its requirements.

13. Independent security evaluation: As part of the design, the device is subjected to a third party (e.g. the Common Criteria [69]) security and reliability analysis.

In table 2.1, the term smart card refers to one that supports the UCOM. Therefore, based on the comparisons shown in table 2.1, it is easy to see that the UCOM-based smart card architecture is suitable for UCTDs. Obviously, smart card technology requires suitable modification if it is to be used as part of the UCOM framework. Whereas, the TPM as expected to display support for most of the UCTD requirements listed above except for requirements twelve and thirteen, which unfortunately are the cornerstones of the UCTD framework. Likewise, AEGIS supports seven requirements out of thirteen.

The GlobalPlatform TEE, ARM TrustZone, and M-Shield meet an equal number of requirements, as their design focuses on the mobile platform that imposes requirements similar to those of the UCTD framework. The TPD design base was the traditional smart

card that supported the ICOM; therefore, it does not support as many requirements as a UCOM-based smart card. We can say that table 2.1 also provides a comparison between UCOM- and ICOM-based smart cards and their suitability for the UCTD initiative.

2.4 The User Centric Tamper-Resistant Device

As is apparent from the comparison in table 2.1, a multi-application smart card architecture has the potential to serve as the underlying framework for the UCTD. The crucial point that has to be taken into account is that smart card architecture is traditionally under a stringent centralised control, whereas the UCTD requires a more diverse architecture which also accommodates the user's ownership. Therefore, the concept of the User Centric Smart Card Ownership Model (UCOM) becomes synonymous with the UCTD. In addition to the UCOM framework for smart cards, for the UCTD initiative the form factor of smart cards is also diversified as shown in figure 2.3.

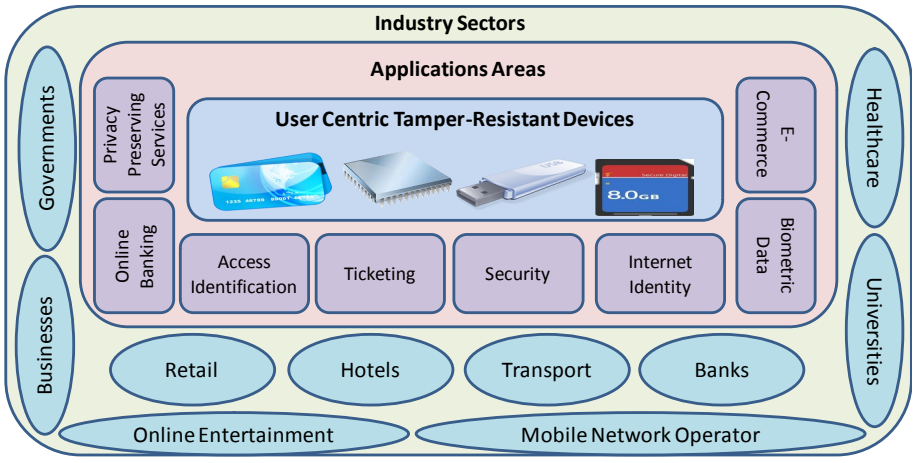

Figure 2.3: Illustration of UCTD form factors, application areas, and industry sectors

Figure 2.3 shows different possible form factors for the UCTD, various applications that it can host, and different industries that can use the provided functionality. In subsequent sections, we briefly introduce multiple application smart cards along with different management architectures. This discussion serves as a foundation for the concept of multi-application smart cards, their management architecture, and potential issues with them.

Before we dive into the UCOM proposal in chapter 3, in subsequent sections we briefly survey different management initiatives in the smart card industry and also discuss some earlier attempts at user centric smart card management.

2.4.1 Smart Card Management Initiatives

Since the beginning of the multi-application smart card initiative, there has been a debate over different possible business models. With each proposed business model, new opportunities and issues have emerged. However, for the most part the business model for smart card-based services has not changed. Nevertheless, one of the main reasons behind the failure to welcome the evolution of business models is the primary purpose of a smart card: to be a security token. However, we postpone this discussion until the next chapter and in this section, we concentrate on the initial proposals for different business models.

When the multi-application smart card initiative was proposed, the predominant belief was that continuing the previously successful business model would be the key to its success. It was envisaged that, like single-application smart cards, multi-application smart cards would also be entirely under the control of the card issuer [14]. The card issuers would have supervisory authority over their smart cards and would decide which applications should be installed or deleted. Other companies (e.g. application providers) wishing to share the platform with the card issuer would have to negotiate and agree on their terms and conditions. This might be suitable for certain business models and industries, but the failure of such architecture to achieve wide-scale deployment suggests that in the past it did not enjoy overwhelming support from the business community.

An alternative ownership model could be to allow smart card users to purchase a smart card. They could then contact the different service providers (companies which used smart card-based applications for the services they offered), acquire their applications and install them on their smart cards. However, some issues [14, 45] were anticipated a decade ago for this possible ownership model, and to some extent most of them are valid concerns [11].

One such issue is whether the card issuers (application providers) will be willing to give their applications to a platform for which they had no operational or security assurance. This proposal can be referred to as the "open card" initiative discussed in section 2.4.2.1. In addition, Deville et al. [14] suggest the possibility of having a dual scheme in which users can install non-security critical applications onto their smart cards but the card is issued/controlled by a centralised authority, the card issuer.

Another scheme proposed by Deville et al. is to have a certification authority, which evaluates and guarantees the security and reliability of the card platform and the installed applications. This model is to some extent deployed by the Multos [29] architecture and it has the potential to be incorporated in the TSM-based architecture discussed in 2.2.1.

Since the multi-application smart card initiative was first proposed, the only surviving and successful model has been the issuer control model, which we term the ICOM. However,

recently a substantial number of trials [41] have been made of the ownership model that includes a certification authority (similar to the model proposed by Deville et al. above), which is in fact an extension of the ICOM. The certification authority in these trials is termed the TSM (section 2.2.1). The second proposed model (i.e. open card) was always considered highly insecure, unreliable, problematic, and not feasible from a commercial standpoint [11, 14, 45, 46, 70]. Nevertheless, the open card proposal was the first concrete effort to introduce user centricity in the smart card industry, later on which we based the UCOM architecture.

2.4.2 User Centricity in the Smart Card Industry

In this section, we discuss the open card and virtual smart card initiative that (to some extent) gave control of the smart card to its user.

2.4.2.1 Open Card Initiative

It is difficult to give an exact definition of open cards. In general, however, the term "open card" is used to refer to blank smart cards that a user can purchase from a supplier. After purchasing the smart card, the user can perform the role previously performed by the card issuer and either accept or buy applications from different application providers. These applications can be installed onto the user's card and used to access any associated services. The whole card is under the user's control similar to the card issuer in the ICOM. Therefore, we can say that the open card initiative is an ICOM framework with the user replacing the card issuer.

Traditional smart card frameworks like Java Card, Multos, and GlobalPlatform were considered suitable for such a usage scenario. Most of these frameworks were built to support the ICOM, and by making the user an issuer, they did not require any substantial changes. However, as implied by Pierre Girard [46], such a mechanism would require an application provider to issue their application to users to install on their smart card. This would require the application provider to trust user not to reverse engineer or corrupt the application.

Such a scenario does not ensure the security, protection of intellectual property, and reliability of an application, as an application provider does not have any control on the destination smart card that hosts its application. The main reason for this lack of control on the part of the application provider was the unavailability of any guarantees regarding the security and operational behaviour of the smart cards. Similar security issues are raised by Chaumette and Sauveron in [70] and they make the open card initiative in its current form unsuitable for the user centric framework.

2.4.2.2 Virtual Smart Cards

One of our proposals concerning the smart card ownership architecture proposes a Virtual Smart Card (VSC). The VSC enables a user to utilise a service by connecting to an associated remote application, based on her location information [68]. The VSC does not require the installation of an application onto a secure element, and this removes the need for implicit trust and ownership of the platform. The remote application is hosted on a Remote Application Server (RAS) that is in total control of a Service Provider (SP) such as a bank or transport operator. The secure element supports the VSC model independently of its owner (cardholder or user). The security of the secure element is ensured by the manufacturer, making the ownership issues irrelevant in the VSC model.

The secure element only has a secure binding with an RAS that enables the mobile phone to connect with it and use the associated services. The mobile phone connects through the internet provided by the mobile operators, as soon as the user enters the associated service zone. The service zone is identified with the aid of the Global Positing System (GPS) [71], in conjunction with the published Service Access Points (SAPs).

The customers are only required to carry their mobile phones, and access to services (i.e. banking, and transport, etc.) would be made available on demand. The location of a user plays an important role in this model. The secure element decides whether to connect with an RAS, depending upon the services available in close vicinity. The VSC framework is illustrated in figure 2.4 and described as below:

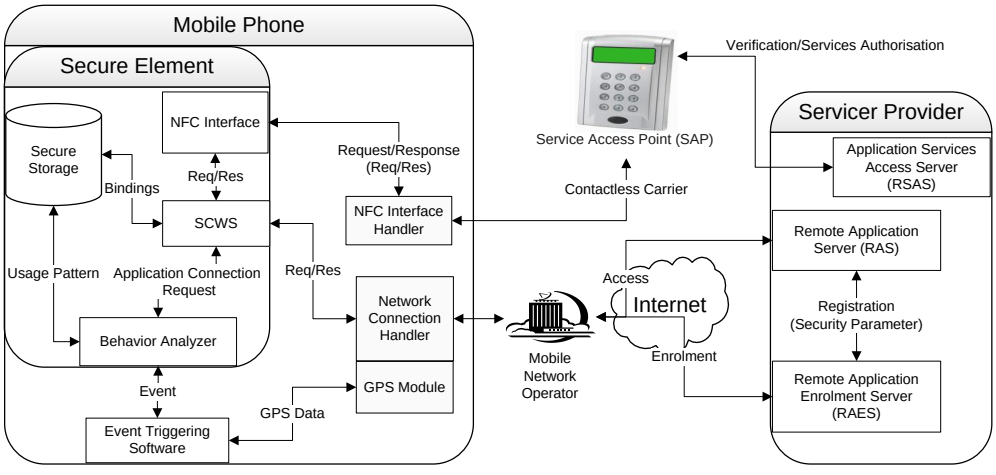

Figure 2.4: Location based virtual smart card architecture

The mobile phone provides an interface that enables a user to enter an SP's registration credentials. The SP's registration credentials are issued by the relevant SP after the user is registered with them. From the SP's registration credentials, the secure element will initiate

an enrolment process with the Remote Application Enrolment Server (RAES). The RAES enables the secure element to establish a secure binding for the Remote Application Server (RAS). The secure binding will be used in future to establish a connection with the RAS to access the remote application(s).

As a user enters the vicinity of a SAP, the event-triggering software sends an alert to the behaviour analyser. The behaviour analyser calculates the probability of the user accessing the service. If the user is predicted to use the service, the behaviour analyser requests the Smart Card Web Server (SCWS) to establish a connection over the internet with the corresponding RAS and act as a bridge between the terminal and the remote application(s).

When the user waves the mobile phone near the SAP to access service(s), the SAP might challenge the mobile phone to authenticate the user (application). The challenge is sent to the remote application by the secure element. Once the remote application has authenticated the user, the terminal will provide the requested service(s).

2.5 Case Studies

In this section, we discuss a non-exhaustive list of case studies where UCTD can provide flexibility, and ubiquity to the existing security and privacy architectures.

2.5.1 One Card - All Services

User ownership enables a user to establish relationships with SPs ubiquitously, which is referred as dynamism. Consider a scenario in the proposed TSM model in which a user who travels extensively around the world (for business or pleasure) acquires a smart card from a TSM in her country of origin. When she travels, she wants to access services that are specific to the visiting country, but she cannot download them onto her smart card. The reasons behind this might be that the services in the visiting country are not associated with (i.e. are not part of) the TSM from which she acquired the smart card. To further explain the scenario, consider Oyster Card [72], which a user can use in London to access local transport services but at the time of writing, it cannot be used as an e-purse. In contrast, the Octopus Card [73], which a user can use in Hong Kong for local transport services, can also be used at groceries, confectioneries, and restaurants. Our user from London would still have to queue to get the Octopus Card when she reached Hong Kong. As we pointed out, she travels extensively around the world and she may not derive a great deal of benefit from the TSM architecture [42, 74].

In summary, in the UCTD framework the user does not have to acquire a new smart card to gain access to new services. The open and dynamic nature of the UCTD allows a user to install or delete any application to which she is entitled, enabling her to download any application when she reaches the visiting country and to continue using the same device. This dynamism is a cornerstone of the UCTD design philosophy, enabling users to match the product to their requirements rather than adapt their requirements to the product.

2.5.2 Authentication Gateway (Single Sign On)

An authentication gateway provider (e.g. Microsoft Passport, Liberty Alliance) can issue an application to its registered users. The application is downloaded onto the user's UCTD. In subsequent sessions, the user can utilise the UCTD to provide an entity authentication service while accessing the gateway provider's services. In addition to the authentication data, the installed application on the UCTD can also provide secure storage for the user's related digital identifiers (i.e. unique data items to identify the user), privacy, and security policies. In such a scenario, the authentication gateway provider does not have to store user-related sensitive data, as this data can be decentralised and stored on the appropriate UCTD. Furthermore, authentication gateways can be implemented to use the biometric data of the user. The biometric data can be stored and matching processes can be performed on the UCTD, thus providing a decentralised architecture for biometric identification.

2.5.3 E-Commerce

The UCTD can be used to provide a dynamic, robust, secure, and reliable authentication mechanism for e-commerce transactions. The model currently propagated for e-commerce by Visa and MasterCard is referred as 3D secure [75], and is a glorified single-sign-on (SSO). In this model, a user registers with a bank that in return issues her a smart card. The user then opts-in for the 3D secure by setting up her credentials (i.e. password or pass-phrase). During online shopping, merchants can authenticate the user by opting for the 3D secure that enables the card issuer to verify the user (i.e. user's consent) for the concerned transaction.

In place of using passwords or pass-phrases, a bank can issue its e-commerce application that is designed as a SSO application (similar to the one discussed in the previous section) and issue it to individual customers. During e-commerce transactions (and for online banking), the user plugs the UCTD into her intended platform (as discussed before, a UCTD can be in any form-factor and pluggable to any computing device). The merchant can then facilitate the communication between the user's UCTD and the bank (that has

issued the application). The bank can opt for certain user credentials (e.g. PIN, password, or biometric) that the bank's application on the UCTD can ask the user to authenticate herself. Therefore, authentication details do not need to be communicated over the internet. The UCTD then provides dynamic authentication and if required can provide a transaction certificate to the merchant as is done in POS transactions [9], effectively avoiding poor technical-security and security-usability along with privacy issues discussed in [76]–[78].

2.5.4 Online Gaming

In April 2011, the security breach of the Sony PlayStation Network and Qriocity services that revealed private information regarding an estimated 70 million users [79] was in the news. This breach has shown that big networks that store user's private data are the prime targets for malicious users. In this section, we are not going to provide a solution to the problems faced by Sony in this security breach but look at how a UCTD can reduce the clustering of large data at one point (i.e. on SP's servers) which provides a potential motivation for attack (i.e. economics of attack[2]) [80].

Therefore, for this case study, we consider a Company A that offers an online gaming platform, and games store to its customers. The objectives of Company A are: (1) to ensure that customers can be uniquely identified and their credentials can be validated, (2) to ensure customers get the services for which they are authorised, and finally (3) to ensure that customers can make purchases while being logged onto the games store or online-gaming platform.

Company A offers an application that a user can download onto her UCTD. The download application is personalised to the user. It has the user's name, email address, and postal address (if necessary). The user identity at Company A's server is identified by a unique user identity (i.e. it is a pseudo-identity that does not have any obvious link to the user). The user has her password stored on her UCTD rather than on Company A's server; therefore, when the user tries to access Company A's resources, she provides her password to the UCTD. We do not delve into the details of how the user identification and authentication will be carried out using a UCTD in our case study, but similar mechanisms are already in operation, for example the EMV Dynamic Data Authentication (DDA) or Combined DDA (CDA) mechanisms [9] (i.e. card transactions at a POS). This will allow Company A to identify and authenticate their customer. In addition, the service privileges associated with the user are also stored in the application. Therefore, Company A does not have to store any of these details at a centralised location as they are already stored on a tamper-resistant device as part of the Company A's application.

[2]Cost-benefit comparison of a potential attacks and outcome from it is referred as economics of attack.

Furthermore, to perform online monetary transactions the user does not have to register her credit card with Company A which would then have to invest a huge amount to safeguard its security. In our case study, the user also has a banking application (issued by her bank) installed on the UCTD. When a user makes a purchase online, the UCTD application of Company A will communicate the purchase request to the banking application that will then process the transaction (e.g. 3D-Secure [75] but it asks the user to enter her password, not at the bank's website, but to the UCTD application). This removes any need to register the card on Company A's website.

In this case study, we have decentralised the data storage that is related to the user's identification and online payment (i.e. credit card details). It is comparatively easy to embed a (small) secure application and get it certified by an independent third party rather than having to implement and secure a large database of user's credit card details. We do not suggest that in our proposed case study all attacks are eliminated. Adversaries can perform attacks, but the financial rewards of such attacks are limited and in comparison less attractive than the invasion of a centralised database (in the breach of Sony's Servers in April 2011, data related to approximately 70 million users was compromised).

2.6 Summary

In this chapter, we discussed the motivation behind the User Centric Tamper-Resistant Device (UCTD). The motivation came from three distinct but continuously converging technologies: smart cards, hand-held devices, and general purpose computing platforms. We briefly discussed different architectures for tamper-resistant devices that can be considered as possible candidates for the UCTD architecture. We then compared these architectures with the smart card technology in different stated aspects of the UCTD. Finally, we discussed the UCTD and its requirements that compel modification to certain aspects of smart card technology. We briefly introduced the concept of multi-application smart cards, its related management models, and its issues. This was followed by a description of selected case studies that utilise UCTDs to provide security and privacy services to existing frameworks.

Chapter 3

Smart Card Ownership Models

Contents

In this chapter, we open the discussion with a description of the ICOM framework and the security and operational assumptions adopted in the ICOM, along with a short introduction to the widely deployed smart card frameworks which support the ICOM. We then provide an overview of the User Centric Smart Card Ownership Model (UCOM) and its major stakeholders. The discussion then moves to the security and operational requirements of UCOM stakeholders. We then extend the UCOM model to include an administrative authority that manages the UCTD while providing the user with freedom of choice. *This model is referred to as the Coopetitive Architecture for Smart Cards (CASC).*

3.1 Introduction

The ICOM has played a major role in the spread of smart card technology to every aspect of modern life. Card issuers see smart cards as a conduit for customer loyalty, rather than as a mere electronic device used to access services. Smart cards have become a market presence, a means of customer outreach and even in certain circumstances a status symbol (i.e. privilege cards). Given all the above, it is obvious why surrendering control of smart cards is a difficult decision for any organisation to contemplate. In this chapter, we describe the ICOM architecture along with prominent platforms that support it.

The aim of a UCTD is to provide security, trust, and privacy services while being interoperable with diverse computing devices (e.g. computers, mobile phones, and tablets). The ownership model for the UCTD has to strike a balance between the user's *freedom of choice* and the SP's requirements of security, intellectual property protection, control, and reliability of their application. For this purpose, we propose the UCOM because it takes into account the ownership requirements of the UCTD framework. Therefore, in this chapter we discuss the UCOM architecture, its main stakeholders, and their requirements.

In some cases, there is a need to have an administrative authority that manages a computing platform. Two examples of such an authority can be parents and MNOs. On a computing platform used by children, the respective parents would like to manage the overall platform while giving the children the right to install or delete any application that does not violate the policy defined by the parents. Similarly, an MNO might provide a mobile handset to a customer in return for signing a fixed term contract. During the contract period, the MNO might be involved with the UCTD that came with the mobile handset and they might want to have the administrative rights to it. We propose a model that accommodates the requirement of administrative authority on a UCTD (smart card), while adhering to the UCOM. Such a model will protect the security and privacy of the user, while implementing the usage policy defined by the administrative authority. We refer to this model as Coopetitive Architecture for Smart Cards (CASC), which is also discussed in this chapter.

Structure of the Chapter: Section 3.2 discusses the ICOM and its advantages and disadvantages. In section 3.3, we briefly introduce prominent platforms that support the ICOM framework. In section 3.4, we describe the ICOM and its major components. The security and operational requirements of individual UCOM stakeholders are discussed in section 3.5. The extension to the UCOM framework to include provision for an administrative authority is described in section 3.6. Finally, we conclude the chapter in section 3.7.

Before diving into the chapter, we want to explain that the discussion of different management models in the previous chapter and in this chapter is necessary to appreciate our

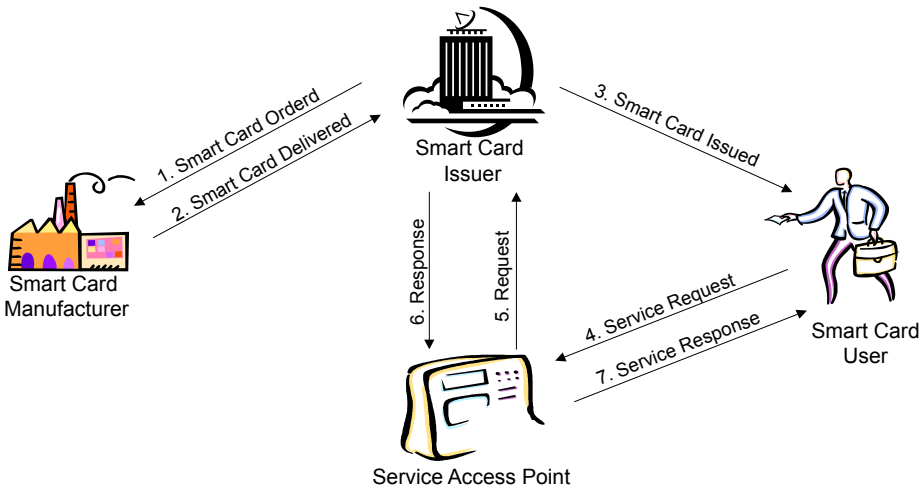

Figure 3.1: Overview of the Issuer Centric Smart Card Ownership Model (ICOM)

contribution. Different business, management, and ownership models in the smart card industry have shaped how the technical mechanisms are implemented. We have to consider the changes in perceptions about control and ownership of smart cards in the smart card industry to justify the modifications that we propose in this dissertation.

3.2 Issuer Centric Smart Card Ownership Model (ICOM)

In figure 3.1, smart card issuers are companies (e.g. in the banking, transport and telecom sectors), which use smart cards to provide services to their customers. The card issuers order smart cards from a card manufacturer. The card manufacturer delivers them to the card issuer, which in turn issues them to individual cardholders. A cardholder presents her smart card at a Service Access Point (SAP) to use the services provided by the card issuer. A SAP can be an ATM (Automated Teller Machine), a mobile phone or a simple card reader; it acts as a gateway to a card issuer's services.

In this framework, the control of the issued smart cards lies with the card issuer, who decides what application(s) will be installed on the cards. If a card issuer has a business agreement with any other company, then the cardholder may get a smart card with multiple applications, as in the case of the Barclaycard's OnePulse card [81]. Barclaycard[1] is the card issuer and it has an established business relationship with Transport for London[2] (issuer of Oyster cards [72]). Therefore, with its bankcard, Barclays provides the Oyster card functionality.

[1]Barclaycard: Barclaycard is a trading name for the banking card sector of Barclays Bank PLC, United Kingdom. Web address: `http://www.barclaycard.co.uk`

[2]Transport for London (TfL) is a publicly owned company that provides transport services to Greater London, United Kingdom. Web address: `http://www.tfl.gov.uk`

The ICOM requires a trusted centralised authority to be set up which will have supervisory authority over smart cards. This centralised authority can be either the card issuer or a certifying authority. For convenience, we use the term card issuer to refer to any centralised authority in regard to the ICOM. The role of the authority is to enforce the security policy, which enables all the applications on a smart card to behave in a predefined manner. The predefined manner is negotiated between the application provider concerned and the card issuer. This agreement defines the parameters under which the application provider may access different services on the card issuer's smart cards. Furthermore, the card issuer will have the authority to grant or deny access to any particular application provider.

Smart cards in the ICOM are acquired by the card issuer, which is in a position to choose their operational and security functionality. This gives assurance to the purchasing company (the card issuer) that the smart cards that carry its applications are secure to their required standard. If the card issuer required a third party evaluation of the smart card product, the card manufacturer might provide the Common Criteria [69] evaluation certificate (a paper based certificate) as a means of assurance.

To summarise the ICOM framework, the privileges or rights that a card issuer receives as part of the ICOM are listed below:

1. Privilege to install an application.
2. Privilege to delete an application.
3. Control over card issuance to individual users. This enables the issuers to decide who receives their smart cards (and effectively control their application).
4. Power to define the security and operational requirements for the smart cards.
5. Enforcement of the security policy.
6. Control over who can access their services using the issued smart cards.

The security and operational assumptions discussed above are the cornerstones of the ICOM. These assumptions have given a strong impetus to the ICOM framework, particularly in the business community. Later in section 3.4, we discuss what privileges the UCOM takes from the card issuer and gives to individual users. This will give an indication of the difference between the ICOM and UCOM, and provide the rationale for suitable modifications to the ICOM platforms discussed in section 3.3.

3.2.1 Advantages of the ICOM

Since the inception of smart card technology, issuers have adopted the ICOM as their model of preference for smart card based services. The advantages of this approach are presented below:

Issuance Control. Only a legitimate issuer (organisation) can offer smart cards to its customers. The card issuer controls the availability of the cards and the card/application management lifecycle. The centralised control of applications and the card management lifecycle stand out as crucial elements in the acceptability of the ICOM. Centralised control has meant that the card issuer treats smart cards in a way similar to the way business assets are treated. Therefore, issuers preferred smart cards to be under their control and the ICOM fitted well with such commercial attitudes. In theory, by ensuring centralised control an issuer can increase the revenue streams by renting out space in a multi-application smart card. However, although such ideas are considered workable to some extent, the adoption of this model is not widespread.

Security Control. A smart card is often deployed as a security token, which provides secure and reliable access to certain services. Most organisations prefer to retain control of the security mechanisms for access to their services, which are implemented on a smart card. This ensures that only the applications installed on their (issued) smart cards can access sanctioned services, which maintains the provided services. As the installed applications are designed by the card issuer, it is considered safe to connect with the services provided by the card issuer. Any compromise of the smart card's security will result in loss for an organisation whose application is installed on the smart card, both financially and in relation to the brand image. To remain secure and confident that the smart cards meet an organisation's security requirements, the organisation will prefer that the cards remains under its control as provided by the ICOM.

Modification Control. Once a smart card is issued, only its issuer or trusted partners may modify the installed applications. Therefore, a malicious user can neither install a new application nor modify existing applications. As the installation, modification or updating of an application is under control of either the card issuer or their trusted partners, it can be assumed with confidence that no application on the smart cards will be malicious. This assumption that centralised control guarantees security led to a realistic but simple approach to numerous smart card security mechanisms; such as the smart card firewall [82], application installation mechanism/protocol [83], virtual machine [84] and platform assurance [56, 85, 86].

Communication Control. A smart card can communicate on different interfaces (e.g. contact, contactless) and protocols as web enabling protocols [87], T1, T2 [88], and NFC-WI [89]. The issuer can regulate the mechanism through which an application on their smart card can communicate with off-card entities. For example, the Oyster application on the Barclaycard OnePulse card will not communicate with any off-card entity through a contact based interface. However, it does communicate via a contactless interface.

Marketing and Customer Loyalty. The plastic card surface is considered to be marketing real estate by the card issuers. For most deployments, the card surface is used to print the company names and logos. In the banking industry for example, a typical smart card will have the issuer bank's name and logo. It might also have the insignia of the payment clearing system (i.e. VISA, MasterCard or American Express, etc.). In addition, the concept that having a smart card with a particular brand translates into the customer loyalty to the organisation has its roots in the initial smart card deployment (i.e. Dinner's Club card). Encouraging customer loyalty definitely had its benefits in some industries like banking, but it might be less beneficial in industries like mobile telecom here the smart card module is hidden inside the mobile phone.

3.2.2 Drawbacks of the ICOM

The ICOM has been successfully deployed over the past decade, but it has minor drawbacks that are listed below:

Card Handling. Usually, an individual will require a number of smart cards[3] [91] for: train or bus journeys, mobile phones, office building access, internet/office-network access, banking/shopping and health services, and other purposes. With increasing numbers of industries relying on smart cards to provide their services to customers, the customer's wallet is becoming crowded with smart cards. Users of smart card-based services already have to carry a large number of smart cards and with each new service they enrol for, they get more. To maintain and manage these cards sometimes becomes troublesome to cardholders who have to use diverse services.

Stringent Model. The concept of the smart card as a medium for promoting customer loyalty and as a marketing avenue took centre stage in business strategy; different card issuers started to consolidate their customer base and this in turn created a situation in

[3] A Survey [90] conducted in 2008 by Federal Reserve Bank of Boston showed that an average American consumer has 5.4 banking cards (i.e. prepaid, credit, and debit cards).

which it was difficult to bring different organisations to share the same smart card platform. The change from perceiving the smart card as a security token to seeing it as a loyalty and marketing medium imposed additional restrictions on it. Therefore, users cannot choose to put an application on their smart cards; the privilege of an installing applications was zealously and solely retained by the card issuers, leaving users with a restricted use of the smart card platform.

Service Roll-out. With the ICOM, a card issuer has to acquire smart cards from a card manufacturer and then either develop application(s) itself or acquire them from the card manufacture or a third party. Once the cards are acquired and they have the card issuer's application, they are posted to individual customers. This process is cyclic: the card issuer may have to reissue new smart cards because of expiry of old ones or it may want to introduce new services or meet new regulatory/legal requirements. Furthermore, it takes a long time to offer new services in the ICOM, since an issuer has to order new smart cards and install new applications on them and then has to issue these smart cards to individual customers. Generally, new services are issued gradually at the time when the issued smart cards are nearing the end of their lifecycle.

Costly. With the ICOM the cost for card issuers is incurred in two ways. The first is acquiring smart cards and getting them certified[4] (third party evaluation of security) to meet any regulatory, standardisation or legal requirements. The second is the loss of possible revenue in the service roll-out period or in the process of issuing a replacement smart card. For example, if a cardholder loses a smart card and requests a new card it usually takes from three days to a week (or sometimes more) in the case of the banking industry before he/she receives it. In industries like telecom and transport the user might acquire the card immediately from designated outlets. However, smart cards deployed in the health sector or national identity cards might have longer re-issue waiting periods. During this period, the customer cannot use the service(s) of the particular card issuer, and this might result in loss of revenue and inconvenience for the user.

3.3 Frameworks for the ICOM

In this thesis, we analyse in some detail the different components of the ICOM frameworks as required, to contrast them with those of the UCOM. For articulation of our arguments, we are not going to dive into the technical details of each ICOM framework in subse-

[4]Security Certification: For certain industries like banking, smart cards offered by the card manufacturers have third party security evaluation as a standard product requirement. However, other industries like telecom and transport often does not require such evaluations.

quent sections. We leave in-depth analysis to later chapters where they are discussed and compared alongside the UCOM-based proposals. Therefore, in this section we will briefly introduce the best-known ICOM-based smart card architectures.

3.3.1 Multos

In 1997, a consortium of companies (MAOSCO) supported the development of a Smart Card Operating System (SCOS) called Multos [29], with one aim: to provide a high level of security and reliability. They required a single operating system which could be implemented on any silicon chip and which had an application written for it that was independent of the underlying hardware. Their vision anticipated the creation of a multi-application smart card. From the beginning, Multos was developed as a secure multi-application SCOS that achieved ITSEC[5] Assurance Level E6 [93](comparable to the Common Criteria EAL7 [69, 94]), which is the highest level attained by any SCOS [6].

The MAOSCO Consortium defines the Multos specifications, and is the license issuer and operator of the certification service for Multos. It has made most of its specifications available to the SCOS developers provided they sign an NDA (Non Disclosure Agreement), and pay licence and royalty fees. A restriction in the Multos specification is its inflexibility with respect to adding new Application Programming Interfaces (APIs). The license agreement with Multos restricts smart card manufacturers from enhancing their product by including new APIs to the specification.

With the advent of the Java Card technology, a Multos card division called StepNexus [95] has made available the Multos SmartDeck environment free of charge [96]. The Multos SmartDeck is a complete high-level development environment which enables application developers to design applications easily for Multos-based cards.

The Multos card architecture is illustrated in figure 3.2. At the top in figure 3.2 is the application layer that contains three applications (namely A, B, and C); each application has its own space, which is protected by the card's firewall mechanism. The next layer is the Application Abstract Machine (AAM), which also includes different APIs. The Multos operating system presides over the hardware and provides services such as communication, memory management, the handling of loading and deleting of applications, together with APDU commands and responses. At the bottom of the figure is the hardware, which supports the SCOS. Functions that access this layer are written in native language, but are accessed by a fully specified virtual machine, which is the same no matter what the hardware.

[5]Information Technology Security Evaluation Criteria (ITSEC) is an international security assurance evaluation criteria [92].

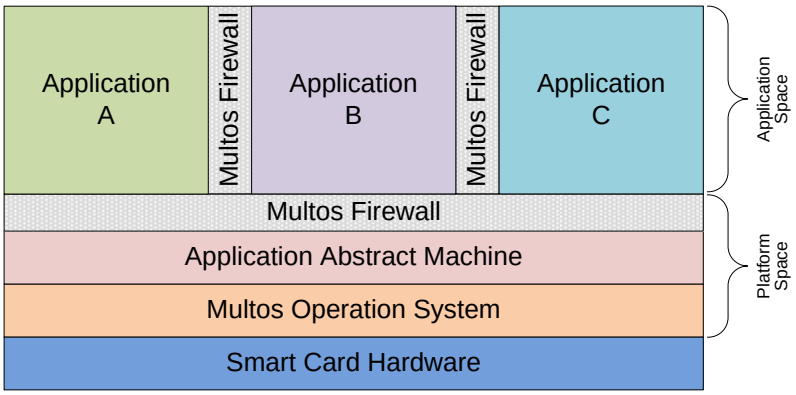

Figure 3.2: Generic representation of the Multos card architecture

The application installation and deletion mechanism proposed by the Multos specification has stringent centralised architecture [97]. Every time an application is to be installed, an application provider will request an "Application Load Certificate" from the Multos Certification Authority through the appropriate card issuer. Because it has such a stringent architecture and a mandatory requirement for a crypto co-processor, this highest security evaluation level smart card platform is not considered the industry's leading specification. This title goes to the Java Card technology, which has proliferated in the smart card industry because of its flexibility and robustness, and its readily available pool of experienced developers.

3.3.2 Java Card

By 1990, Sun Microsystems had started a project to develop a language that generated a program once that could then be executed on any micro-controller. Their only consideration was the micro-controllers used in electronic appliances (i.e. toasters, washing and coffee machines, etc.). However, with the emergence of the internet came an increasing need to deliver rich contents on the heterogeneous devices connected to the internet. The Java language that accommodated these changes was invented by Sun Microsystems and it can be adapted to the ever-growing personal computer market. Soon Java became a de facto standard language for internet applications.

In 1996, engineers at the IT technology provider Schlumberger at Austin (TX, USA) developed the Java Card, which is a smart card that supports a subset of Java language [98]. When this idea was made public, the smart card industry immediately became interested. Later, Sun Microsystems arranged a meeting to gather input and explore the dynamics of the smart card industry. All the major smart card manufacturers attended this meeting. This was the beginning of the Java Card forum, which is an independent forum for collaboration between different industrial players. Membership of the Java Card forum

includes smart card manufacturers and application/solution providers. This is another important aspect of Java Card technology, which differentiates it from Multos in its constant consultation with the industrial players.

Due to these consultations, the Java Card technology has changed considerably. The Java Card has progressed from the initial release which supported only limited functionality (i.e. primitive data types such as boolean, byte and short) to the more recently released Java Card specification 3.0 [16] which includes the TCP/IP stack [99] along with SSL/TLS [100] and HTTP [101] /HTTPS [16, 87, 102]. The Java Card can behave as an internet device in either a server or client capacity. The architecture of a Java Card is illustrated in the following figure 3.3 and is described below:

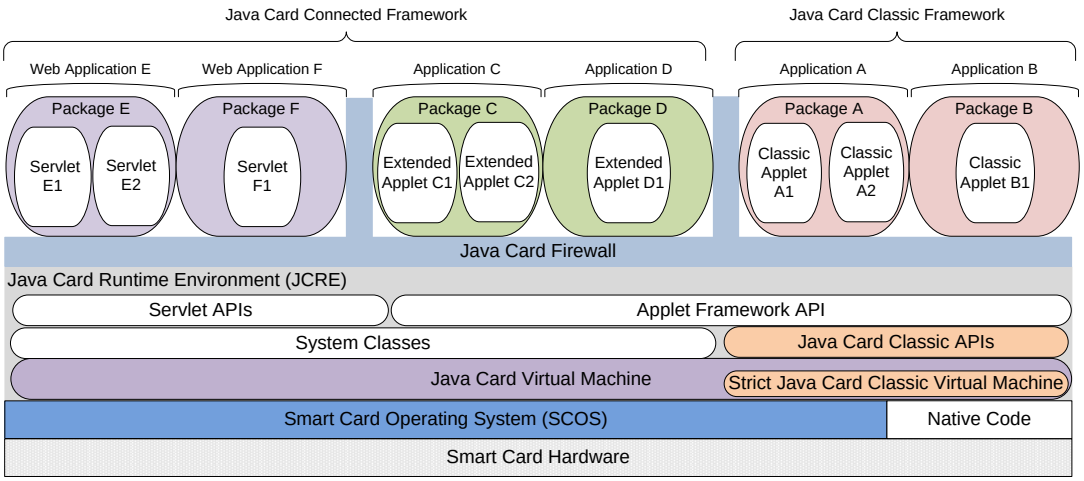

Figure 3.3: Generic representation of the Java Card 3 architecture

In comparison to Multos, Java Card is better termed a platform rather than an operating system. Due to this distinction, above the smart card hardware layer, Java Card Virtual Machine (JCVM) and native methods are all available. The native methods section can also be considered a native operating system developed by each card manufacturer to support its implementation of the JCVM. Furthermore, as Java will take longer to execute than the native code, the native method segment is also the crucial point for implementing the cryptographic algorithms. Above this layer, we have the Java Card Runtime Environment (JCRE), which provides different services in the shape of Application Programming Interfaces (APIs) and System Classes to the residing applications. The Java Card APIs provide a well-structured framework to access the system-level services in a secure and reliable manner. The segregation on a Java Card between platform-application and application-application is enforced by the Java Card firewall.

The Java Card specification leaves decisions regarding the mechanism for installing, deleting, updating, and managing multiple applications on a smart card to the card manufacturer. The industry appreciated this move, as it allowed greater flexibility than Multos

which is rigid in comparison. However, it was soon realised that for application management tasks it would be beneficial for all the players in the smart card industry to have a unified specification. Sun Microsystems did not get involved in defining such a specification but gave space to the industry's players to decide on a specification. The proposed application management framework came in the form of the GlobalPlatform card specification, which is the topic of the next section.

3.3.3 GlobalPlatform

Towards the end of the 1990s, the smart card technology was being adopted on a large scale. It was soon realised by card manufacturers, card issuers, and application providers that to manage such a complex and technically complicated infrastructure, it would be beneficial to share a unified and universal card management system which freed them from the demands of the smart card hardware, platform, application service and card issuer's requirements. Visa gave the impetus to this idea by transferring their Open Platform initiative to a consortium of card issuers, application providers, and smart card manufacturers, later known as GlobalPlatform.

GlobalPlatform is a non-profit organisation which provides a vendor-neutral specification of different components of smart card-based business operations. The GlobalPlatform card specification provides a standardised view of smart cards, card terminals, and smart card-based infrastructure management systems. The specification which is of most relevance to this thesis is the GlobalPlatform card architecture. We provide detailed descriptions of the different components of the GlobalPlatform card specification as required throughout the thesis to clarify our discussion.

The GlobalPlatform card specification is a card architecture-neutral specification which does not require/specify any particular Runtime Environment (RTE). However, at present most smart cards which support the GlobalPlatform specifications actually call for a Java Card Runtime Environment (JCRE). Technically, it is possible to have GlobalPlatform architecture on a Multos card, but at the time of writing this thesis, the author was not aware of any such implementation in the public domain. Nevertheless, as the Multos card already has a well-defined application management framework, there is no particular need to complement a Multos card with the GlobalPlatform implementation.

The architecture illustrated in figure 3.4 has applications from the card issuer, application providers (partners of the card issuer) and a global service application, which provides services to all the applications installed on the smart card. The applications are managed and controlled by the mechanism of security domains. A security domain has an association with one of the application(s) which it manages and enforces the security policies of the

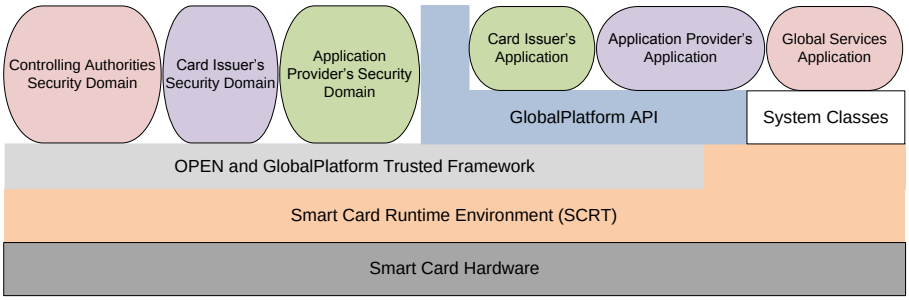

Figure 3.4: Generic representation of the GlobalPlatform card architecture

owner of the domain. The security domain also provides separate cryptographic keys to the card issuer and the application providers to manage their respective domains/applications. The security domain also manages key handling, encryption, decryption, digital signature, and the verification of (hosted) applications (i.e. only at the time of installation [30, 103]). The card issuer generates the security domain (application domain) on the card and then gives control of the application domains to the card issuer's partners (application providers). These application providers can then manage their applications independently of the card issuer's involvement.

The OPEN framework defined in the GlobalPlatform specification handles/controls the downloading and installation of applications. The Trusted framework enables different services such as inter-application communications; however, the "GlobalPlatform Card Security Requirement Specification" [1] states that GlobalPlatform relies on the underlying platform's (e.g. Java Card, and Multos) implementation of the firewall mechanism..

The crucial component of the GlobalPlatform card specification is termed the Card Manager. This is a generic term used for such services as OPEN, the issuer security domain and Cardholder verification method services. The Card Manager actively controls the smart card environment. Furthermore, the smart card issuer cannot access any of the application domains because they are protected by the cryptographic keys (access keys) and these keys are shared only between an application domain and an application provider. However, if a particular application provider violates the agreement with the card issuer, or they no longer have a partnership to provide services, then the card issuer can block or delete the application provider's application.

In this section, we have provided a short description of the GlobalPlatform card specification which in no way defines all the functions of the specification. However, we continue to refer to the GlobalPlatform card specification in subsequent chapters and give detailed descriptions of its components as required. It is noteworthy that GlobalPlatform has shown the capacity and willingness to adapt to the industry's trends. It has published "GlobalPlatform Card Remote Application Management over HTTP Card Specification v2.2" in response to Java Card 3 and "GlobalPlatform's Proposition for NFC Mobile: Secure

Element Management and Messaging" [104]; both of these specifications accommodate the current trend towards the NFC mobile phone-based services.

3.3.3.1 Why not GlobalPlatform for UCTD?

This question comes to mind, as the GlobalPlatform card specification provides an accepted and reliable way to manage applications on smart cards in pre- or post-issuance stages, so why not just have a GlobalPlatform-based smart card whose ownership is with the cardholder?

This option is workable in a limited scenario where the applications are less critical. The cardholder would have the same rights as the card issuer in the ICOM. However, the security issues raised due to the delegation of the ownership that are discussed in the rest of this thesis are not adequately addressed in the GlobalPlatform card specification. The reason for this is the underlying assumption in the card specification — that the card issuer (or in user centric cards, the cardholder) is a trusted entity and any other application provider has to trust them. The security mechanisms implemented on smart cards are also based on the similar assumption that there is a trusted entity which we can term as the root of trust. In the smart card industry, the root of trust is usually an organisation that acts as a smart card issuer. If we give the smart card ownership to the user under the traditional framework, then the root of trust would be the individual user. The assumption that each user is trustworthy, might not be easy to ascertain. Therefore, GlobalPlatform, along with other frameworks of the ICOM, are not only useful in the ICOM but also in the UCOM. However, they require modification so that they can securely support the UCOM's requirements.

Similarly, there is an argument that having a TSM-based architecture can provide user control by making the user the TSM. All application providers are connected with the user who then installs their applications onto her smart card(s). In reality, this idea is similar to the open card initiative discussed in section 2.4.2.1. The user-based TSM concept suffers from the same issues, including trusting the user, application provider inability to control the destination smart card, assurance of security and reliability of the application.

3.3.4 Other Proposals

In this section, we discuss initiatives that were not taken up as enthusiastically as were those in the previous sections.

3.3.4.1 Windows for Smart Cards

At the time Java Card was proposed, Microsoft also ventured into the smart card business [105]. They took the same sort of approach as they took in the PC domain and developed the whole architecture of the smart card operating system without any consultation with the smart card industry.

The Windows for Smart Cards (WfSC) was an ISO7816 [24] compliant smart card with a FAT file-system [106, 107] and rule-based Access Control [108]. The design of the virtual machines was rooted in Intel 8048 and the associated APIs were compact versions of the Windows (Win32) APIs [105]. As noted by Jurgensen and Guthery [105], the WfSC has one of the best-designed Virtual Machine architectures, similar to the Multos. Applications for the WfSC can be written in Visual Basic and Visual C++.

Due to Microsoft's design-in-isolation approach, the WfSC was not adopted as quickly as other frameworks from the beginning and Microsoft soon had to shelve the project [6]. In contrast the Java Card took the approach of consultation and open specification, which give it enough of an advantage to outdo powerful initiatives such as Multos and WfSC.

3.3.4.2 Smartcard .Net

Although Microsoft's own attempt to enter the smart card market did not pay off, Hive-Minded Inc. (since 2006 owned by StepNexus Inc.) later developed a smart card framework based on Microsoft's .NET. Its main aim was to allow its developers the freedom to choose any programming language (i.e. C#, VB.NET, J#, and Jscript.NET, etc.). Like Java Card, it supported all data types except floating point and 64-bit integers.

The later acquisition of Hive-Minded by the Multos manufacturer Step-Nexus put an end to the smart card .Net initiative. However another smart card manufacturer, Gemalto, has since offered .Net products [109] with the caveat that these smart cards are natively supported by Microsoft Windows Vista and Windows 7. However, it is yet to be seen how far the smart card .Net framework will spread.

3.3.4.3 Multi-application BasicCard

BasicCards were available even before the Java Card was proposed [110]. Initially they supported only single applications but since 2004, the BasicCard manufacturer ZeitControl has started to issue multi-application BasicCards (e.g. MultiApplication BasicCard

ZC6.5). Like WfSC, they also support the FAT file system, but unlike any other smart card framework, they support floating-point numbers natively [6]. Although these are less expensive than other options available to customers, they have not seen an exponential growth such as Java Card.

3.4 User Centric Smart Card Ownership Model (UCOM)

The UCOM provides to different entities the architectural, operational, and security framework needed to support the delegation of smart card ownership to its users. In this section, a detailed description of the UCOM is provided, defining the basic working principles of the UCOM along with a description of the UCOM components. Figure 3.5 illustrates the basic architecture of the model.

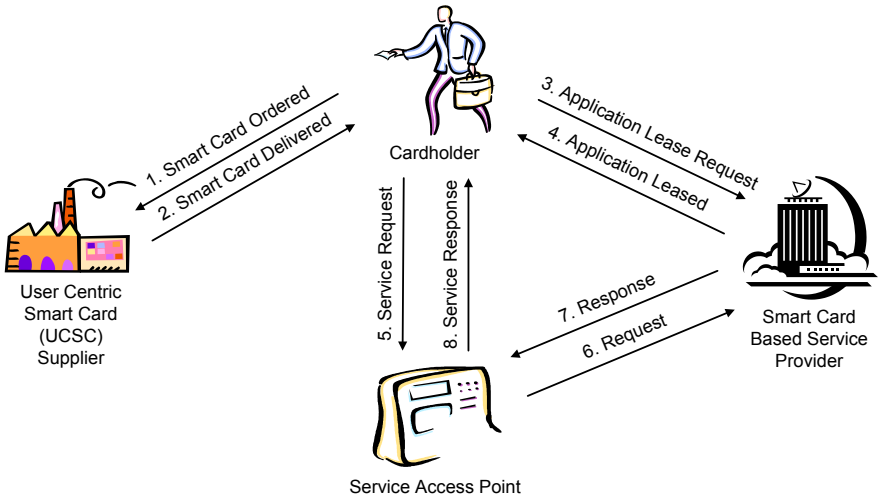

Figure 3.5: Overview of the User Centric Smart Card Ownership Model (UCOM)

In the UCOM, a card issuer is denoted as a Service Provider (SP). An SP and a card issuer represent the same entity in different contexts of UCOM and ICOM, respectively. The main difference between an issuer and an SP is that a card issuer provides a smart card's hardware and application(s) to their customers, whereas an SP only offers smart card application(s) that can be downloaded to a customer's smart card on request.

The aim of the UCOM is not to replace users with card issuers as the open card initiative (see section 2.4.2.1) does. The UCOM ensures that the same level of security and application control is provided to an SP as in the ICOM, while provisioning the *freedom of choice* to individual cardholders. Going back to the list of privileges for ICOM (section 3.2), the UCOM transfers the privileges (rights) one, two and four to the smart card users.

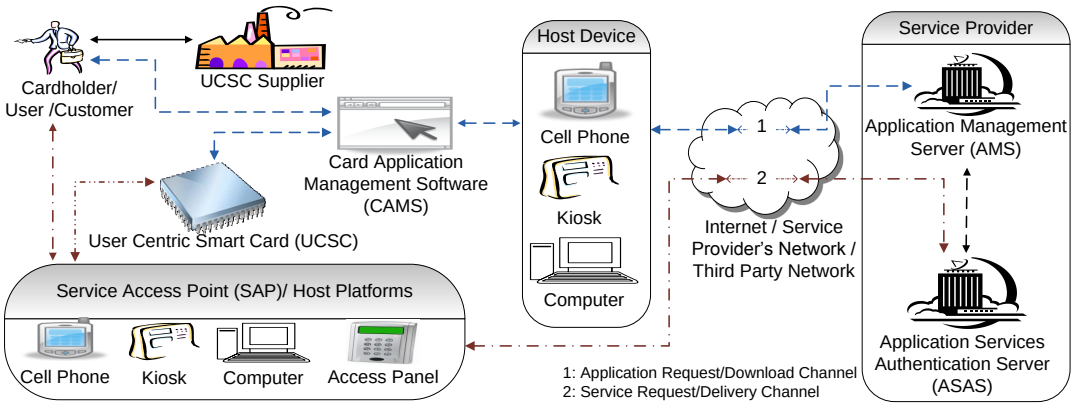

Figure 3.6: Illustration of the UCOM components and their interactions

Therefore, the role of security policy enforcer is taken up by the smart card itself and cardholders only have the privilege to install and delete applications. The ICOM enables a card issuer to control the issuance of its smart card to individual users, which is translated as the privilege to lease its application in the UCOM, whereas installed applications will always be in total control of the SPs, and users will be entitled to use them under the lease policy of their respective SPs.

The architecture of a UCOM consists of seven main components, as shown in figure 3.6: the User Centric Smart Card (UCSC) supplier, the cardholder, the UCSC, the Card Application Management Software (CAMS), the host devices, the Service Provider (SP) and the Service Access Points (SAPs). A smart card that supports a user's ownership is called a UCSC and we use this term only in this chapter to differentiate between ICOM-based smart cards and cards that support the UCOM.

A cardholder acquires a UCSC from a UCSC supplier. After acquiring the UCSC, the cardholder requests an SP to lease their application. The cardholder presents her card to a host device. The host device then enables the cardholder to use Card Application Management Software (CAMS) that establishes an interface between UCSC and the SP's Application Management Server (AMS). After authentication of the cardholder and security validation of the UCSC, the AMS leases the application(s).

Once the application is installed, the cardholder can present her card to a SAP to access services. The SAP will establish a connection between UCSC and the SP's Application Services Authentication Server (ASAS). After being authenticated by ASAS, the user can use the designated service. The architecture of the application lease and usage is explained in section 3.4.6.2. In subsequent sections, we discuss the UCOM components shown in figure 3.6.

3.4.1 Supplier

A supplier is an organisation that sells UCSCs. A smart card manufacturer, an SP, or a third party vendor can be the supplier. The suppliers ensure that the UCSCs supplied to a user have a reliable and secure platform that supports the UCOM and fulfils the requirements of a UCSC, as stipulated in section 3.5.3.

3.4.2 Cardholder

A cardholder is not just a user of the UCSC, but she is also the owner of the card. Cardholders would have the ability to install and delete any application they require. A cardholder would also be a registered customer of the respective SPs. The cardholder could install an application on the UCSC after being authorised by the corresponding SP. After installation, the cardholder could use the application to access associated services.

From a UCOM's perspective, cardholders do not have to be technically literate (about the underlying architecture of the platform) and do not have to be trusted users. In subsequent chapters, it will be shown that we adopt the default assumption that the cardholder may be malicious.

3.4.3 User Centric Smart Card (UCSC)

The UCSC is the cornerstone of the UCOM proposal. It provides a seamless framework for application installation, management, and deletion to the cardholder. The ownership management and delegation (i.e. the transfer of ownership between different users) is also provided by the UCSC while preserving the integrity and security of the platform, and the privacy of the cardholder. Furthermore, the UCSC manages secure communication with the respective SP to request the lease of the application. An SP does not have to trust the cardholder, but they need to trust the smart cards. The UCSC supports mechanisms that can provide dynamic and ubiquitous security assurance and validation to the requesting entity. It ensures that during the lifetime of the smart card, the entire platform along with the installed applications will be secure and reliable.

Henceforth, we will be using the term UCSC and smart card interchangeably, unless otherwise specified.

3.4.4 Card Application Management Software (CAMS)

The CAMS acts as an interface between a smart card, an SP's Application Management Server (AMS), and a cardholder, as illustrated by figure 3.6. The cardholder uses this interface to authenticate with the SP's AMS and to perform smart card management tasks (e.g. application installation, deletion and state change) [10]. In addition, it can also provide protocol translation services to avoid any incompatibilities between the smart card capability and the respective SP's AMS (e.g. in a scenario where a smart card does not support the TCP/IP protocol [111]. Therefore, the CAMS will translate the TCP/IP protocol to one supported by the smart card). The CAMS communicates directly with a smart card, but it is hosted on the host devices that are discussed in the next section.

3.4.5 Host Device

Host devices are electronic devices that hold the smart card and facilitate it in establishing a secure channel to an SP's AMS for application management tasks. These devices can be categorised into mobile phones, kiosks, and computer-based host devices. There is no specific security requirement on the host device. It is advisable to consider the host device as insecure while implementing a solution supporting UCOM.

3.4.6 Service Provider (SP)

An SP is an organisation that offers smart card-based services. It develops applications that support different smart card platforms (e.g. Java Card [28], and Multos [29]). A cardholder can easily download the chosen application, and use it to access the SP's services.

To install an application and access the services provided by an SP, users have to register with the SP. This registration mechanism is already in place in different industrial sectors (such as banking, and telecom, etc.). After the successful completion of the registration, an SP will send the account details to the user. The user will use these account details to gain access to a server that provides the functionality to maintain the SP's application. This server is called the Application Management Server (AMS). After an application is installed, the user can access the services provided by the SP. To access these services, the application on the smart card has to be authenticated by the SP's Application Services Authentication Server (ASAS).

3.4.6.1 Application Management Server (AMS)

An AMS is implemented and maintained by an SP to support the UCOM. The AMS's main function is to facilitate authorised cardholders to ubiquitously manage the SP's applications on their cards.

The account details provided by the SP to its customers contain the AMS access credentials. Using these credentials, the user can access and install the SP's application(s). The exact mechanism of the user registration and credential issuance, and the usage mechanism which controls how a user's credential will be verified (authenticated) are specific to each SP.

The main function of an AMS is to maintain the SP's application(s) and to ensure that the application is only leased to a smart card if it satisfies the SP's Application Lease Policy (ALP).

3.4.6.2 Application Lease Policy (ALP)

An ALP defines the minimum requirement of an SP that a smart card has to satisfy before the SP will lease its application. The ALP is defined by an SP, and it could have the following requirements.

1. Minimum smart card hardware requirement.
2. Minimum Smart Card Operating System (SCOS) or platform (e.g. Java Card) requirements.
3. Minimum application memory requirement.
4. Minimum Common Criteria Security Evaluation Level [69].
5. Maximum number of smart cards that can hold the lease of the application.
6. Cryptographic key generation requirements.
7. Secure communication channel requirements.
8. Application lease limits and restrictions (if applicable).

In addition to the abovementioned points in the ALP, an SP can define some additional criteria for its application. During the application installation process [10], a smart card tries to satisfy the SP's requirements, and if it succeeds, the SP will lease the application to the smart card; otherwise the request will be declined.

3.4.6.3 Application Services Authentication Server (ASAS)

An ASAS authenticates the valid lease of an application that the requesting card holds. In the UCOM, multiple smart cards of a single user may have an SP's application, depending on the particular SP's ALP. If the ALP allows multiple smart cards to have a valid lease of its applications, an ASAS is necessary to verify the validity of the lease to the smart card. Once a smart card is authenticated as holding the valid leased application, it can access services offered by the SP, subject to successful authentication by the user (user's application).

The ASAS is already implemented in the ICOM, and is referred as a back-office or transaction clearance system. Each industry has its unique way of implementing the ASAS and part of the design philosophy of the UCOM is that we do not require a modification (in most cases) to the existing architecture of the ASAS in the ICOM.

3.4.7 Service Access Point (SAP)

Any device that a cardholder can use to access services provided by an SP is called a Service Access Point (SAP). A SAP can be a mobile phone, a kiosk, a computer, or an access panel. The main function of all these devices is to connect with the SP through a smart card and provide services to the cardholder. We consider that SAPs do not have to be secure; therefore, during the course of this thesis SAPs will be treated as insecure terminals.

3.5 Security and Operational Requirements of the UCOM

The UCOM delegates control of the smart card to its user. This scenario introduces unique requirements that were not present in the ICOM.

3.5.1 General Requirements

In this section, we discuss the requirements that are not specific to a particular entity in the UCOM but are instead common to the overall architecture.

GR1. **Control:** The UCOM should provide a mechanism(s) that enables cardholders to manage applications on their smart cards. It should also ensure that only the autho-

rised cardholder can execute any privileged commands. These privileged commands can change the state of the smart card (e.g. alter the application installation and deletion commands).

GR2. **Security:** This requirement stipulates the need to provide protection against attacks that can violate the security requirements of each of the UCOM components. Therefore, the UCOM should provide an adequate security mechanism to protect all the components and their communications.

GR3. **Privacy:** The privacy requirement is essential in the UCOM, since smart cards are used as a secure token to access some personal information or monetary services. Therefore, the UCOM should provide privacy services to those components that require it.

GR4. **Interoperability:** The UCOM should not prefer any particular platform, SCOS, or hardware configuration. The aim is to provide an unrestricted scalability to the overall UCOM model. A smart card would present the list of supported functionalities to the requesting SP and then it would be up to the SP's discretion whether it leased its application or not. From the point of view of the smart card, the SP and the UCOM architecture, there should be no preference. If a smart card supports an SP's requirements and supports the security and operational functionality required by the SP's ALP, then the SP would lease its application on request, unless there is some genuine reason not to do so. In the event of a valid reason, the SP should inform the requesting cardholder of the main reason for not leasing the application.

GR5. **Ease of Maintenance:** The framework should be simple to use and maintain for SPs. In addition, it should not require any extensive modification to the existing infrastructure.

GR6. **Impartiality:** The smart card supplier could be a smart card manufacturer, an SP, or a third party vendor. Regardless of the supplier, the smart card should not favour any particular application or set of applications. This would be possible if an SP supplies the smart cards, and then they might be tempted to give additional privileges to its own application. Therefore, the UCOM should provide guarantees that all applications will have the equivalent privileges to suit their operations.

The major components of the UCOM are cardholders, smart cards, and SPs. In subsequent sections, the operational and security requirements of these major components are discussed.

3.5.2 Cardholder's Requirements

A cardholder is an entity that uses a smart card to access authorised services. In the UCOM, the control of a smart card is with its user. Therefore, cardholders have complete control over the choice of applications on their smart cards. They will have the flexibility to change the installed applications on their smart cards. Furthermore, they could install or delete any applications they are entitled to at their convenience. The framework will provide the mechanism that ensures the secure control and the ubiquitous management of applications on smart cards. A cardholder's requirements in UCOM are listed below:

CR1. **Security:** If a smart card is inherently insecure, or if it becomes vulnerable to new threats, it can affect the security of applications installed on the card. We cannot expect that each cardholder is technically capable of ensuring and managing the security of the smart card; therefore, a cardholder would require an assurance that the card platform will be secure and reliable even if it is in the possession of an illiterate or malicious user.

CR2. **Privacy:** Applications installed on a smart card represent the identities of the cardholder in different contexts. For example a college card, a health card and a credit card represent a cardholder's identity as a student, a patient, and a consumer respectively. These identities are in the form of applications that have some unique characteristics (e.g. student ID, patient ID, and Primary Account Number: PAN) to identify a particular user. Therefore, applications on a smart card can be treated as the identities of the cardholder. In the ICOM, these identities may not have any connections with each other. However, in the UCOM, any or all of these identities could be on the same card, creating a privacy issue if one application becomes aware of the existence of others on a smart card. Therefore, the identities on a particular card should not have any links between them. For example, a college application should not be able to find out about a medical application(s) installed on the same card.

CR3. **Least Interaction (Seamless Framework):** Most users do not understand the technology behind a particular product (i.e. mobile phone applications). Therefore, the framework should not be based on the assumption that an average user can perform technically challenging tasks. The UCOM should be seamless and should perform all necessary tasks by itself, and only involve the user when required.

CR4. **Interoperability:** The smart card user will not want to buy a separate smart card for each application. Smart card suppliers should provide cards that support most of the available functionalities and SPs should offer applications in many formats to support as many different execution environments as possible.

CR5. **Ownership Mechanism:** A mechanism is required that securely authenticates the owner of the smart card and facilitates the exercise of her privileges (i.e. installing and deleting applications).

3.5.3 User Centric Smart Card's Requirements

The security and operational requirements of smart cards are listed below, and most of these requirements should be implemented by smart card suppliers:

SCR1. **Security Assurance:** A smart card should have a mechanism(s) that will provide assurance to a requesting SP that adequate security and privacy measures have been implemented to ensure the security and privacy requirements of the application(s).

SCR2. **Security Evaluation:** A smart card should be able to evaluate the downloaded applications and verify that they do not pose any threat to the safe execution of the other application(s) on the card.

SCR3. **Interoperability:** A smart card should have the capability to support a wide range of applications and communication interfaces/channels.

SCR4. **Runtime Environment Fairness:** A smart card should require that no application installed on it tries to monopolise the runtime environment.

SCR5. **Application Management:** A smart card should have mechanisms to securely manage the applications. The management of the applications includes application downloading, installation, and deletion.

SCR6. **Application Lease Management:** A smart card should require adequate mechanisms to manage an application lease. The lease of an application may have certain limits or restrictions that a smart card has to satisfy over the lifetime of the application. For instance, the limit could be an expiry date or the number of times used, and restrictions may be the runtime environment's configuration. The mechanism should be able to provide assurance to the SP that their application will be deleted if the limit is reached, or cease to execute if lease restrictions are violated. The application leased from the SP is governed by an application lease policy described in section 3.4.6.2.

SCR7. **Ownership Validation:** A smart card should support a mechanism to authenticate its owner using some security parameters (i.e. Personal Identification Numbers: PIN, password, pass-phrase, or biometric, etc.). There should be the functionality to reset these values securely.

SCR8. **Feature Interaction Problem Avoidance:** The smart card needs to have a mechanism that prevents any possible feature interaction problems. Feature interaction problems are caused by dependencies between the software and hardware. The mechanism will record any dependencies of an application before it is installed. Therefore, when there is a change that affects the dependencies of the application, it can either be deleted or cease its execution.

SCR9. **Malicious Application/User Problem:** The smart card should implement effective security and privacy measures to counter a malicious user or application. The smart card platform should be able to resist the introduction of malicious applications or hardware-based intrusions to breach security. To avoid application-level breaches, the card should have the capability to perform application code verifications on the card. In addition it should have a conservative execution environment (i.e. a defensive virtual machine [112]) that ceases execution of an application if there is a violation of the card's security.

SCR10. **Application Scanning Attack:** Each application on a smart card acts as an identity of the card owner in some context, as discussed in the previous section. Each application on a smart card has unique Application Identifier (AID) [24]. A malicious user can use the application identifier to scan the applications installed on a particular card. It will not only violate the privacy requirement of the user, but may also enable the attacker to create/modify the attack. A malicious user may choose the weakest application in the smart card to initiate an attack. The smart card should have a security mechanism to avoid such an attack.

3.5.4 Service Provider's Requirements

Service providers use smart cards as secure tokens to offer their services, in a secure manner, to their customers. If this secure token is compromised, they have to bear both financial and brand losses. Therefore, from a business point of view, SPs have even more at stake than card issuers, both financially and in relation to brand image, and they would be reluctant to adopt the UCOM, if they had doubts about its security. The requirements of SPs in the UCOM are listed below:

SPR1. **Transmission Security:** SPs will lease their applications to a smart card through the internet or a third party intranet. It is essential that applications are not tampered with during the transmission.

SPR2. **Installation Security:** After an application is downloaded to a smart card, it will be decrypted. During this process, no on-card or off-card entity should be able to gain access to the application code or data.

SPR3. **Maintenance Security:** SPs require access privileges for their applications to update application data files or to update the applications themselves. This access to the application should be secure, and it should not involve cardholders. However, if the SP desires, the cardholder's consent can also be requested for application update/modification processes.

SPR4. **Intellectual Property Protection:** In the UCOM, SPs lease their applications to the user's smart card. A malicious user can simulate a smart card on a device (e.g. computer) and then request the application from an SP. If the SP leases the application to a simulated environment, the malicious user can reverse engineer the application. This could reveal the secret information (e.g. cryptographic keys, and algorithms) contained in the application. Therefore, SPs require that adequate security protection is implemented to safeguard their confidentiality and integrity of their applications.

SPR5. **Application Code Confidentiality:** The applications from SPs would need to comply with certain standards. However, these standards will not prevent them from using proprietary algorithms. The SPs would require that the code of their applications and its inner workings should remain confidential. The smart card provides adequate security to prevent an adversary from gaining knowledge of the SP's application. Application Code Confidentiality will be jointly provided by mechanisms that satisfy the abovementioned requirements.

SPR6. **Application Control:** In the UCOM, although users own the smart cards, the ownership of the application still resides with the SP. SPs only lease applications to their customers (UCSCs). The SP has the power to revoke the application lease, or to block or modify the application. The lease of an application is governed by the ALP. In addition, the SP controls all operations that a cardholder can request on its smart card. These operations are application installation, application deletion, and state change (i.e. application block and unblock operations), and they cannot be performed unless authorised by the relevant application's SP.

SPR7. **Protection Against Monopolies:** In the UCOM, multiple applications may be installed on a smart card from different SPs. They all share and use the same smart card hardware and card operating system. There is a possibility that a smart card operating system can favour certain applications. As a result, these applications can monopolise the card. Therefore, SPs will require assurances that such scenarios will not be possible.

SPR8. **Ease of Implementation:** SPs have made substantial investments to their existing infrastructure that provides services to their customers. Therefore, the UCOM should not impose unnecessary changes to the existing infrastructure. The basic idea is to implement the UCOM as another layer on top of the existing infrastructure, which implements UCOM without extensive modification.

SPR9. **Feature Interaction Management:** The SPs require that any changes to the smart card platform that can affect their application's execution should be avoided. In the best-case scenario, the smart card platform follows the SP's delegated process that removes the interdependencies between applications in such situations. However, if this does not solve the problem, then the card should simply block the application or possibly delete it (with the card owner's consent).

SPR10. **Protection from Malicious Users:** The SPs would like to have their applications protected by the underlying smart card platform. Therefore, even if the application were issued to the malicious user, he would not be able to obtain any sensitive information about the application.

3.6 Coopetitive Architecture

A UCTD can be a single-user, and/or a multi-user device, depending upon its deployment architecture. In a multi-user architecture, a device (e.g. a computer or tablet) might be used by multiple users. Furthermore, a UCTD might be part of a corporate network, as in the case of a mobile phone issued to employees by an organisation. The organisation (referred as an administrative authority) would like to retain the control of the UCTD issued as part of the mobile phone, while giving freedom to the user to have/manage the UCTD independently of the organisation. It is necessary to consider these deployment scenarios in order to achieve true scalability that will enable small to medium-sized organisations like leisure facilities, local libraries, schools, surgeries and colleges, as well as large-scale organisations like banks, MNOs and transport operators, to provide their services through a UCTD to a user without any restrictions from a centralised authority (e.g. card issuer).

To accommodate the role of an administrative authority on a UCTD, we extend the UCOM architecture and propose the Coopetitive Architecture for Smart Cards (CASC). In the CASC a cardholder retains the application choice but under the provision of an administrative authority (e.g. TSM). Nevertheless, the baseline architecture of the UCTD is based the UCOM, as the CASC is an extension of the UCOM architecture that provides centralised ownership of the device while preserving the user's *freedom of choice*.

The CASC combines the TSM architecture with the openness, scalability, and flexibility of the UCOM architecture. In this architecture, users get their choice of selecting which application they want on their smart cards, and administrative authorities (or the TSM, the administrator of the corporate network) have a permanent presence on the cards along with possibly being part of the revenue loop. The CASC requires all the necessary modifications to the existing smart card architecture (i.e. deployed in the ICOM) to achieve its goals. So while we focus on the UCOM in this thesis, we are implicitly also catering to the

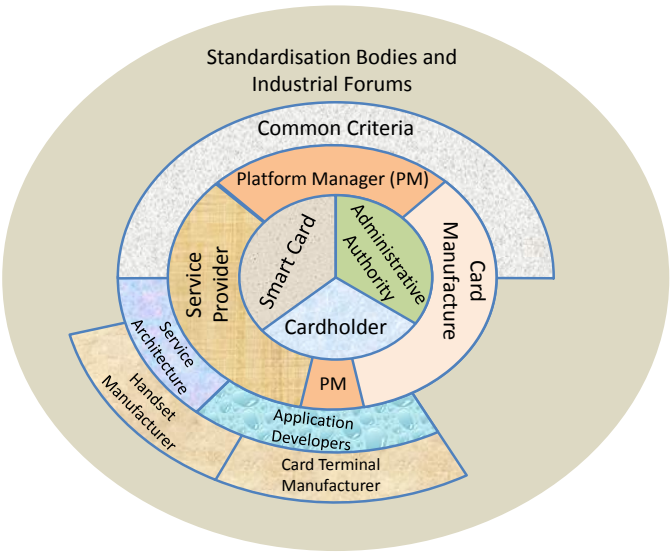

Figure 3.7: Ecosystem of the Coopetitive Architecture for Smart Cards (CASC)

requirements of the CASC.

The ecosystem of the CASC is illustrated in figure 3.7, and at its centre there are three main entities: the administrative authority (the card issuer, TSM, and corporate authority etc.), cardholder, and the smart card. The administrative authority issues the smart cards to its customers. The cardholder would have the choice to install or delete any application they would require. The management of the smart card application installation, deletion, and application/card lifecycle management is handled by the Platform Manager (PM) (discussed in section 4.2). The PM facilitates both the administrative authority and the cardholder to perform their sanctioned tasks.

As an example, consider a scenario in which a user enrols into the multi-application smart card service architecture through a Mobile Network Operator (MNO). In this scenario, the MNO plays the role of an administrative authority. As the customer of the MNO, the user can receive an NFC-enabled mobile phone (possibly under a fixed period contract) and UCTDs. In certain cases, MNOs subsidise the mobile phone in return for a fixed period contract with their customers. The phone is under MNO lock and it can only be used on the issuing MNO's network. At the end of the contract, the customer can request the MNO to unlock the mobile phone. The acquired secure element(s) would have the MNO's application installed by default. In addition, if the user is a customer of any other organisations that are associated partners of the MNO in the TSM scheme, then she may get their applications pre-installed on the secure element. This secure element would enable the user to request installation or deletion of any application she chooses, except

for the MNO's application. At the end of the contract, the MNO would not only unlock the mobile phone but also the TSM. From this point forward, the user can either use the secure element under UCOM architecture or register their secure element with any other TSM (or continue with the original MNO).

Similarly, other entities like card issuing banks, transport service operators, smart card and mobile phone manufacturers, or independent third parties, can participate by offering competitive products that adhere to the CASC. The security and reliability of the coopetitive smart cards would be a key issue, which is dealt with separately in the ICOM and UCOM scenarios.

3.7 Summary

In this chapter, we have discussed the Issuer Centric Smart Card Ownership Model along with different smart card frameworks that have been proposed to support the ICOM. In addition, this chapter serves as a basic introduction to the UCOM framework and its main stakeholders. We have also defined the roles of each of these stakeholders along with their security and operational requirements. Furthermore, we have extended the UCOM architecture to represent a more viable and dynamic architecture for the UCTD that might require administrative control.

Chapter 4

User Centric Smart Card Architecture

Contents

In this chapter, we discuss the security and operational architecture of the UCOM supported platform, termed the User Centric Smart Card (UCSC). Subsequently, we detail the inclusion of a trusted computing platform for smart cards that we refer as the Trusted Environment & Execution Manager (TEM). This is followed by the rationale behind the changes to the traditional smart card architecture to accommodate the remote security assurance and validation mechanism. We propose an attestation protocol that provides an online security validation of a smart card by its manufacturer. Finally, the attestation protocol is informally analysed, and its test implementation and performance are presented.

4.1 Introduction

The ecosystem of the UCOM is centred around smart cards that have to implement adequate security and operational functionality to support a) enforcement of security policies stipulated by the card platform and individual SPs for their respective applications, and b) operational functionality that enables an SP to manage its application(s), and a cardholder to manage her ownership privileges. The smart card architecture has to represent this change in ownership architecture. For this purpose, we require a trusted module as part of the smart card architecture. The module would validate the current state of the platform to requesting entities in order to establish the trustworthiness of a smart card in the UCOM architecture.

In the UCOM, the card manufacturers make sure that smart cards have adequate security and operational functionality to support user ownership. In addition, the cardholder manages her relationship with individual SPs. These relationships enable her to request installation of their applications. Before leasing an application, SPs will require an assurance of the smart card's security and reliability. This assurance will be achieved through a third party security evaluation of the smart cards before they are issued to individual users. Furthermore, to provide a dynamic security validation, the evaluated smart cards implement an attestation mechanism. The attestation mechanism should accommodate remote validation, as in the UCOM an SP will not always have physical access to the smart card. In addition, the attestation mechanism will certify that the current state of the smart card is as evaluated by the independent third party. Therefore, the trust architecture in the UCOM is based on the adequacy of the third party evaluation, and the security and reliability of the remote attestation mechanism.

Structure of the Chapter: Section 4.2, discusses the UCTD architecture and its major components. To provide security and reliability assurance to remote entities we define the role of the Trusted Environment & Execution Manager (TEM) in section 4.3. Subsequently, we extend the discussion to the security evaluation in section 4.4, followed by the remote attestation mechanism in section 4.5. In section 4.6, we discuss different types of UCTD ownership and how an off-card entity can acquire them. In section 4.7 we propose an attestation protocol; in section 4.8 we detail an informal analysis and test implementation results of the attestation protocol.

4.2 Platform Architecture

The proposed architecture for a UCTD is depicted in figure 4.1 and this architecture satisfies the requirements of the UCOM discussed in section 3.5.3.

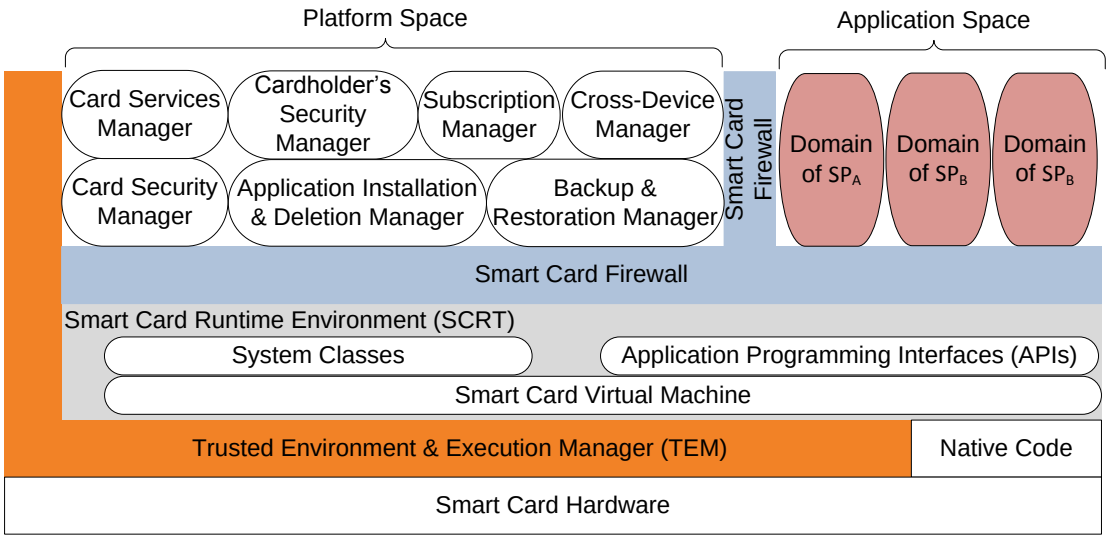

Figure 4.1: User Centric Smart Card (UCSC) architecture

Most of the components shown in figure 4.1 are either an improvement to the existing framework or an addition to the GlobalPlatform architecture. We use GlobalPlatform as the base architecture for the components in this section. These components modify the GlobalPlatform card specification to accommodate the UCOM philosophy. Please note that in this section, we make repeated references to the security and operational requirements discussed in section 3.5.

Furthermore, certain components that are shown as part of the UCTD architecture in figure 4.1 are discussed in later chapters where they are described in detail. These components include the application installation & deletion manager (chapters 5 and 9), the backup & restoration mechanism (chapter 9), the cross-device manager and smart card firewall (chapter 7), and the Smart Card Runtime Environment (chapter 8). We delay their discussion to later chapters for the sake of argument flow and logical placement as we compare them with the existing smart card architectures (e.g. Java Card, Multos and GlobalPlatform).

4.2.1 Spaces

A "space" is a memory container that holds collections of services or applications (i.e. domains). As depicted in figure 4.1 there are two spaces: platform space and application space. The platform space is owned by the smart card platform itself so that users do not have any control over the services installed in the platform space. The application space can be under the control of an off-card entity that may be a centralised authority (i.e. TSM or card issuer) or the smart card user. In addition, there can be multiple application spaces on a smart card accommodating a centralised authority and the respective card

user (section 3.6). The owner of an application space has the right to install or delete any application they choose within their respective space. With the concept of spaces, we can even extend the smart card's capability to accommodate multiple users. An example is a home personal computer used by family members where each member of the family has her or his own profile (account) on the computer. In such a scenario, applications on a UCTD belonging to individual family members should also be securely segregated, and this can be accomplished by creating an application space for each individual user.

A logical set of memory locations, associated with a single SP, is called a domain and it is under the complete and independent control of that SP. The domain provides a simple mechanism in which each application has a secure compartment that is independently managed by the SP. Domain ownership is delegated independently of any off-card entity (e.g. card manufacturer) to the SP during the application installation process that is discussed in chapter 6. A point to note is that the concept of domains is widely deployed by the GlobalPlatform card specification [30] and we simply adapt it to the UCOM architecture.

The managers shown in the platform space of figure 4.1 are collectively represented by the term Platform Manager (PM), illustrated in figure 3.7.

4.2.2 Card Security Manager

The card security manager is the hub for the different security and operational services that a smart card provides.

During the application installation process, the card security manager will facilitate the generation of an SP's domain and oversee the transfer of control of the domain to the appropriate SP. For each application belonging to an SP, there will be a separate domain allocated to the SP that will only have one application in it. This is to allow an SP to manage its individual applications on a smart card individually. Furthermore, this also simplifies the deletion, and blocking/disabling of applications. The card security manager can delete entire domain and any associated privileges to applications installed in the domain — without affecting other applications in the domain. The card security manager facilitates the transfer of domain control to the appropriate SP. This transfer includes the generation of cryptographic keys that the SP will use them to authenticate itself to the domain and perform related management tasks (e.g. application installation, deletion, blocking, unblocking and update). The card security manager would also ensure that the keys generated during the application installation process are not revealed to any third party (e.g. card manufacturer or cardholder).

If an installed application violates the security policy of a smart card, the card security

manager can take action and restrict the application by either blocking it so it cannot execute, or by deleting it. In the GlobalPlatform card specification, such a mechanism requires the card issuer's permission whereas in the UCOM the card security manager wants to delete an application it only requires permission from the cardholder.

In addition, when the ownership of a smart card is changed or if the card is decommissioned, the card security manager is responsible for resetting the smart card configuration. This process includes the deletion of all installed applications and any data related to applications/users. The resetting operation will set the smart card to the default factory setting, as a blank card. Such a mechanism does not exist in the GlobalPlatform card specification and it is discussed as part of the decommissioning of the UCTD in chapter 9.

The card security manager provides functionality that ensures the platform is in conformance with the requirements CR1, CR2, CR5, SCR5, SCR6, SPR1, SPR2, and SPR6 that are listed in section 3.5.

4.2.3 Card Services Manager

Services provided by the smart card platform are under the control of the card services manager. The services include the off-card interface, the runtime Application Programming Interface (API), and default applications. The access rights to these services are designated (requested) by the respective application's SP and the card services manager enforces them. This functionality enables an SP to manage the behaviour of its application(s) on a smart card.

Furthermore, a smart card might have multiple applications from different SPs that provide the same service, like banking applications from distinct banks. In such a situation, the user would have the option of making one application the default application of the group to which it belongs. The card services manager deals with a list of default applications when a smart card is presented at a Services Access Point (SAP). If the SAP only requests an application that belongs to a particular group (e.g. transport, banking, telecom or access control) without specifying a particular member of that group, the card services manager selects the default application for the group. However, if the SAP wants to select a specific application, which may not be the default application of its group the SAP has to request that application explicitly.

The card services manager ensures that the platform satisfies requirements CR3, CR4, SCR5, SPR6, and SPR7 (defined in section 3.5).

4.2.4 Cardholder's Security Manager

The cardholder's security manager maintains services that facilitate an effective and secure management of the smart card contents by its user (cardholder).

At the time a UCTD is delivered to a user, it might be a blank card, which is under the default ownership of the smart card manufacturer. The cardholder's security manager facilitates a cardholder to acquire the control of the smart card (section 4.6), which will enable her to install or delete any application she desires.

Furthermore, when a user requests any privilege services (e.g. application installation, application deletion, a list of installed applications), she has to authenticate herself to the cardholder's security manager. On successful authentication, the cardholder's security manager will proceed with the requested service.

When a user takes the ownership of a smart card, the card contents (e.g. cryptographic keys and certificates) are specific to the user. Therefore, when the ownership changes hands, the cardholder's security manager requests the card security manager (section 4.2.2) to initiate the clean-up command that deletes all applications and data, returning the smart card to the default ownership (card manufacturer's ownership). This process is referred to as decommissioning and is discussed in chapter 9.

The cardholder's security manager provides functionality to satisfy requirements CR1, CR2, SCR1, and SCR7.

4.2.5 Subscription Manager

The subscription manager handles the registration of a smart card with an administrative authority. The authority can be a corporate and home-network administrator and/or a centralised scheme manager like a card issuer or TSM. These entities might be registered before the card was issued or the user might choose to register her smart card to a particular authority to get better services.

The subscription manager facilitates the registered administrative authority to manage their application space on the UCTD. In addition, if a user is allowed to evict the administrative authority then the subscription manager will proceed with the removal process. This process will include deleting the associated space and all applications (domains) in the respective space, along with revoking any privileges delegated to the administrative authority on the UCTD. In carrying out this process, the subscription manager is similar

to the cardholder's security manager except that the cardholder's security manager caters to a user's requirements whereas the subscription manager caters to an administrative authority. The subscription manager is an optional manager and is only required if a user wants to be part of an administrative authority or if the UCOM platform is issued by an organisation that wants to keep control of its interests for example, a mobile network operator that subsidises UCTDs to its customers.

4.3 Trusted Environment & Execution Manager

On a typical smart card, several mechanisms are in place to test and verify the state of the platform (both software and hardware). At the software level, GlobalPlatform card specification has proposed the controlling authority (termed CA in the GlobalPlatform card specification) [74] and the Mandated Data Authentication Pattern (Mandated DAP) mechanism [30, 74]. In the DAP mechanism, an off-card entity (controlling authority) signs applications that are being loaded onto a smart card, and this approval of the applications is verified by an oncard entity referred to as the GlobalPlatform card manager [30]. At the hardware level, the Known Answer Test (KAT) for cryptographic modules mandated by FIPS [113] and similar mechanism are deployed by the smart card manufacturer (i.e. RAM test, and checking checksum of non-volatile memory, etc.) [5].

At the time of writing (September 2011), the Trusted Computing Group (TCG) was initiating a working group to devise specifications for a trusted module for embedded devices [114]. The working group has not released any specifications regarding the trusted module for embedded devices. We propose the Trusted Environment & Execution Manager (TEM) as a trusted module for embedded devices like smart cards. The TEM is fundamentally different from the Trusted Platform Module (TPM) [18] and Mobile Trusted Module (MTM) [19] in two respects. Firstly, the TEM implements a self-test mechanism that includes hardware parameters to provide remote attestation and a dynamically configurable integrity measurement mechanism that is based on a challenge-response framework. Secondly, the TEM is not based on a static architecture; in fact, it enforces platform security policies during the application execution rather than just generating the hash (once) at the start of the application execution. The architecture of the TEM is illustrated in figure 4.2.

The concept of TEM is to group/provide similar and enhanced functionality that provides assurance and validation of the platform to requesting on-card or off-card entities. The TEM is independent of the platform configuration that is mainly concerned with the smart card runtime environment, which can be based on a technology such as Java Card or Multos. A TEM does not have to be implemented in hardware; it can be software-based

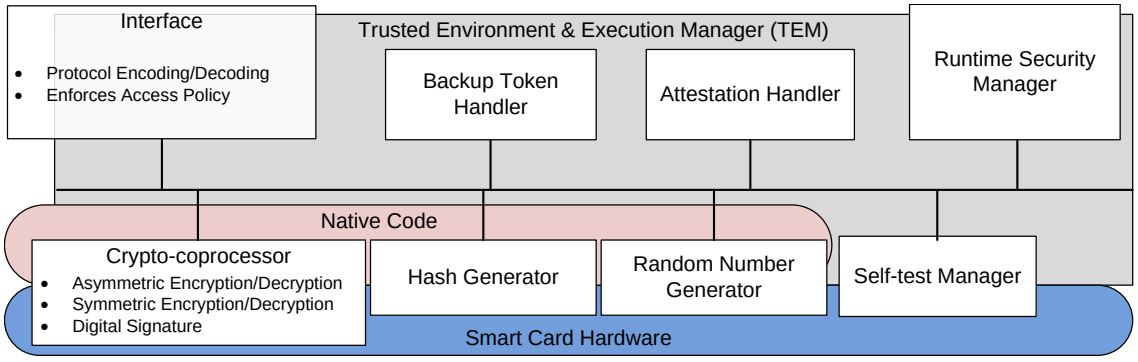

Figure 4.2: Architecture for the Trusted Environment & Execution Manager

and utilise the smart card's cryptographic hardware (the crypto co-processor). The TEM requires access to the crypto co-processor for encryption/decryption, signature generation and verification, and random number generation.

4.3.1 Interface

The TEM interface manages the communication between the TEM and on-card entities (e.g. platform services and applications) or off-card entities (e.g. SPs). The TEM interface does not replace the off-card interface discussed in section 4.2.3, it only implements the communication service that a TEM uses to communicate with on-card applications and (off-card) SPs.

The TEM interface implements the attestation protocol discussed in section 4.7. Furthermore, it also provides a state validation service (section 4.4.3) to installed applications during the application sharing process. The state validation of an application can only be performed by the TEM if it is explicitly requested to do so by that application. Therefore, for state validation, the TEM establishes a shared secret with an application (at the time of application installation). When the installed application (refer to it as App_A) needs to provide state validation to another application (refer to it as App_B), for example during the application sharing process, the TEM will only provide the state valuation of App_A to App_B if App_A explicitly requests the TEM with App_B identity using the shared secret (for a more detailed discussion, please see section 7.3.2).

4.3.2 Backup Token Handler

The backup token handler acts as a repository that stores the restoration tokens of individual applications (if sanctioned by their respective SPs) on a smart card. When a user registers with a backup server or wants to transfer the installed applications from

one smart card to another, the backup & restoration manager retrieves these tokens from the backup token handler, encrypts them, and communicates to the intended entity (e.g. backup server or new smart card). The details of this mechanism are further elaborated in chapter 9.

4.3.3 Runtime Security Manager

The runtime security manager deals with the enforcement of the platform policies regarding the smart card runtime environment. These policies may deal with the security and reliability of an application execution, and they ensure that an application executes in a trustworthy manner. The runtime security manager is discussed in detail in chapter 8 where we examine the threats to the smart card runtime environment and related countermeasures.

4.3.4 Attestation Handler

The attestation handler and the self-test manager are part of the security assurance and validation mechanism discussed in section 4.4. The difference between these two modules (i.e. the attestation handler and the self-test manager) of the TEM is that one focuses on the software and the other on the hardware. However, in the proposed attestation mechanism (section 4.5) they complement each other to provide proof that a smart card is secure, reliable and trustworthy.

During the application installation process, the attestation handler will verify the current state of the platform runtime environment (e.g. security and operationally sensitive parts of the SCOS) and affirm to the appropriate SP that the platform is as secure and reliable as it is claimed to be the evaluation certificate discussed in section 4.4. Once the application is installed the relevant SP can ask the TEM to generate the state validation of an application (e.g. signed hash of the application), ensuring that the application is downloaded without any errors onto the platform. This function of the TEM is similar to the DAP [30, 74].

Furthermore, SPs can request the state validation of their applications at any time during the lifetime of the applications on a smart card. In addition, as part of the application sharing mechanism the TEM also provides application state validation to the applications that share each other's resources (discussed in chapter 7).

4.3.5 Self-test Manager

The self-test mechanism checks whether the smart card is tamper-resistant as certified by a trusted third party evaluation. The aim of the self-test mechanism is to provide a remote hardware validation framework in a way that enables a requesting entity (e.g. an SP) to independently verify it. As our focus and expertise is not the hardware end of the smart card, we do not propose any hardware-based mechanism in this thesis, which is one of the possible directions for future research.

A self-test mechanism in the UCTD should provide the properties that are listed below:

1. Robustness: On input of certain data, it should always produce associated output.
2. Independence: When the same data is input to a self-test mechanism implemented on two different devices, they should output different (random) values.
3. Pseudo-randomness: The generated output should be computationally difficult to distinguish from a pseudo-random function.
4. Tamper-evidence: Any attack aiming to access the function should cause irreversible changes which render the device dead.
5. Unforgeable: It should be computationally difficult to simulate the self-test mechanism and mimic the actual deployed function on a device.
6. Assurance: the function should provide assurance (either implicitly or explicitly) to independent verifiers. It should not require an active connection with the device manufacturer to provide the assurance.

There are several possibilities for a self-test mechanism in a UCTD including using active (intelligent) shield/mesh [115], the Known Answer Test (KAT) [113], and the Physical Unclonable Function (PUF) [116].

To provide protection against invasive attacks, smart card manufacturers implement an active shield/mesh around the chip. If a malicious user removes the active shield then the chip will be disabled. The self-test mechanism can be associated with this shield to provide a limited assurance that the protective measures of the chip are still in place and active.

Furthermore, Hash-based Message Authentication Code (HMAC) can be deployed with a hard-wired key that would be used to generate a checksum of randomly selected memory addresses that have non-mutable code related to the SCOS. This mechanism requires the involvement of the device manufacturer, as the knowledge of the correct HMAC key would be a secret known only to the manufacturer and its smart cards.

Another potential protection strategy is to utilise Physical Unclonable Functions (PUFs) [116] to provide hardware validation. It is difficult to find a single and consistent definition of PUF in the literature [117]. However, a property description definition of the PUF is provided by Gassend et al. in [116]. Usual applications of the PUF described in the literature are in anti-counterfeiting [118], Intellectual Property protection [119]–[121], tamper-evident hardware [122], hardware based cryptography [60, 123]–[125] and secure/trusted processors [126].

Based on the above listed features, table 4.1 shows the comparison between different possible functions that can act as the self-test mechanism. Although the debate regarding the viability, security, and reliability of the PUFs is still open in both academic circles and industry [127]; for completeness, we use them as a self-test mechanism in our proposals because they meet most of the requirements listed in table 4.1.

Table 4.1: Comparison of different proposals for self-test mechanism

Features	**Active-Shield**	**Keyed-HMAC**	**PRNG**	**PUF**
Robustness	Yes	Yes	Yes	Yes
Independence	No	No	Yes	Yes
Pseudo-randomness	No	Yes	Yes	Yes
Tamper-evidence	Yes	–	Yes*	Yes
Unforgeable	No	Yes	Yes*	Yes
Assurance	Yes	No	Yes	Yes*

Note. "Yes" means that the mechanism supports the feature. "No" indicates that the mechanism does not support the required feature. The entry "Yes*" means that it can supports this feature if adequately catered for during the design.

If a manufacturer maintains separate keys for individual smart cards that support the HMAC then it can provide the independence feature. However the HMAC key is hard-wired and this makes it difficult for it to be different on individual smart cards of the same batch. Furthermore, it requires other features to provide tamper evidence, like active-shield. On the other hand, PUFs and adequately designed Pseudo-Random Number Generators (PRNGs) can provide assurance that the platform state and the tamper-resistant protections of a UCTD are still active.

Before we discuss how a self-test manager and an attestation handler can be implemented based on PUF and/or PRNG, we first discuss the overall framework that is responsible for providing security assurance and validation of a smart card.

4.4 Security Assurance and Validation Mechanism

The UCOM requires a mechanism that supports a dynamic and remote security assurance and validation process which is based on the TEM coupled with a third party evaluation. The third party evaluation certificate provides a security assurance and TEM provides the validation that the assurance is correct at the time of request. However, in the UCOM environment, applications are not required to be evaluated by third parties, and so evaluations can be costly, and may discourage small and medium-scale organisations from opting for the UCTD-based architecture. To verify the security and reliability of an application, a smart card can employ on-card verification mechanisms like bytecode verification [128].

In this thesis, we refer to the Common Criteria (CC) evaluators for third party evaluation as it is one of the most accepted and deployed evaluation mechanisms in the smart card industry.

4.4.1 Common Criteria

In late 1990s, the Common Criteria (CC) was released, and they were later adopted as a multi-part ISO/IEC standard (ISO/IEC 15408 [129]), that is internationally accepted under the Common Criteria Recognition Agreement (CCRA) [69].

The CC scheme defines the methodology for expressing the security requirements, conformance claims, evaluations process, and finally, certification of the product. The security requirements for a product at an abstract level are stipulated by Protection Profiles (PPs). A Security Target (ST) details these security requirements and makes the conformance claims for a product or its sub-component(s), generally referred to as Target of Evaluation (TOE).

The Evaluation Assurance Levels (EALs) are predefined assurance packages that have a set of security requirements. There are seven packages defined in the Common Methodology for Information Technology Security Evaluation (CEM) [130] that are referred as EAL 1 to EAL 7 with level seven being the most comprehensive security evaluation. The CC proposes an evaluation methodology, which defines the procedures that an evaluator should follow when processing conformance claims regarding a TOE under a particular ST, PP and desired EAL. This evaluation methodology is published in the CEM [130].

In the literature, some reservations are expressed regarding the validity and the process efficiency of the CC [67, 86, 131]. However, the CC has taken a strong hold in the smart card industry, especially in high-security areas like banking and IDS/passports, as the security

evaluation-standard of choice. CC evaluation has a well-established security requirement specification [69] and evaluation methodology [130]. Furthermore, card issuers, application providers, and most smart card manufacturers have extensive experience of the CC evaluation scheme.

In subsequent sections, we discuss how CC plays the role of trusted third party (evaluator) in the UCOM security assurance and validation process.

4.4.2 Assurance Phase

This section describes the pre-issuance security evaluation. It is divided into two subsections: smart card evaluation and application evaluation.

4.4.2.1 Smart Card Evaluation

In this phase, the card manufacturer would get their smart cards evaluated to the defined EAL. If the evaluation of the smart card is successful, the CC Certification Body (CB) would issue a cryptographic certificate [132], referred to as the Platform Assurance Certificate (PAC). The main components of the certificate include a PAC identifier, a unique reference to the product's ST, PP, and list of hardware security mechanisms and a hash of the immutable (security and reliability critical) part of the SCOS.

Smart cards could be subjected to extensive evaluation by the manufacturer, evaluation labs, or the academic community even after the issuance of the card's PAC; therefore, if such evaluations discover vulnerabilities in a particular product, SPs can disable their application leases to them, preventing the smart cards from accessing the sanctioned services. Furthermore, the CB may downgrade their PAC assurance level or include the card on a certification revocation list, prohibiting such smart cards from downloading applications in the future.

In addition, a PAC can also have the manufacturer's ID, the evaluator's (Commercial Licensed Evaluation Facility: CLEF) ID, the manufacturer's signature verification key [132], and the validity period. The validity period is determined by the CC evaluators and it represents an estimated period that a given product is expected to remain secure. The manufacturer's ID uniquely identifies the smart card manufacturer, and similarly the CLEF ID identifies the evaluation body that has carried out the evaluation. Finally, the certificate would also certify the manufacturer's signature key pair.

The manufacturer would use the signature key certified by the PAC to issue certificates to

the individual smart cards that have the same validity as the associated PAC.

4.4.2.2 Application Evaluation Phase.

An SP would create an ST according to its security requirements and get it evaluated by the CLEF. If the SP's application is approved, the CB would issue a cryptographically signed Application Assurance Certificate (AAC) that would contain the EAL level achieved by the SP's application and the hash of its immutable application code.

The structure of the AAC is similar to the PAC, except for few changes. Details of data fields included in the AAC are: the SP's ID, the evaluator's (CLEF) ID, reference to the evaluation target documents (PPs and ST), a digest of immutable application code, the SP's signature key, and the certificate's validity period.

The certificate chain traversal and verification of the individual certificates in the chain are comparatively easy for the SPs as they have more computational power than a smart card and independent access to an external network (i.e. the internet). To perform such tasks would no doubt be challenging for a smart card; therefore, it would request the SP to provide the certificate hierarchy that leads back either to the smart card manufacturer or the third party evaluator of the smart card (i.e. to an entity that is considered trusted by the smart card). In this way, the smart card can easily verify the certificate as the root of the certificate chain provided the SP's certificate chain has entities (certificate issuers) that the smart card trusts.

4.4.3 Validation Phase

This phase deals with the process that provides a dynamic and remote attestation of the current state of the smart card or applications. The attestation mechanism combines the self-test manager and attestation handler of the TEM to provide the state validation of the UCTD. For validation of applications the TEM attestation handler only generates the hash of the application in question, but the attestation mechanism for UCTD validation has two modes of operation: offline and online. In the offline mode, the validation process is independent of the card manufacturer and the smart card provides a security validation message to the requesting entity. In the online mode the card manufacturer provides the security validation message (i.e. a signed message from the card manufacturer) to the requesting entity.

4.5 Attestation Mechanisms

In this section, we discuss the two attestation mechanisms based on non-simulatable PUFs and pseudorandom number generators that combine the functionality attestation handler and self-test manager discussed in section 4.3.

4.5.1 Non-simulatable PUFs

A non-simulatable PUF is a PUF that is computationally difficult to simulate by either the device manufacturer or a malicious entity. This property has made non-simulatable PUFs a candidate for true/pseudo random number and secret key generators [123, 133, 134].

Based on non-simulatable PUFs, we describe two algorithms 4.1 and 4.2 that take into account the offline and online modes of the attestation mechanism.

Algorithm 4.1: Self-test algorithm for offline attestation based on a PUF

Input : l; list (array) of selected memory addresses.
Output : S; signature key of the smart card.
Data: *seed*; temporary seed value for the PRNG set to zero.
n; number of memory addresses in the list l.
i; counter set to zero.
a; memory address.
k; secret key used to encrypt the signature key of the smart card.
S_e; encrypted signature key using a symmetric algorithm with key k.
Notation:
x ⟵ y+z: first the operation on the right of the arrow will be performed and the result will be stored in x. This notation is common for all algorithms in this thesis.

```
1  SelfTestOffline (l) begin
2      while i < n do
3          a ⟵ ReadAddressList (l,i)
4          seed ⟵ Hash (ReadMemoryContents (a), seed)
5          i ⟵ i+1
6      if seed ≠ ∅ then
7          k ⟵ nmPUF (seed)
8      else
9          return testfailed
10     S ⟵ DecryptionFunction (k, S_e)
11     return S
```

The offline algorithm is based on the function `SelfTestOffline` that takes a list of selected memory addresses (l) stored on the card by the card manufacturer. This list has memory addresses of security and reliability critical components of the smart card platform. The

function `SelfTestOffline` iterates through the l and generates a hash of the contents of the given memory location. The generated hash value is then stored as a *seed*. After traversing through the l, the `SelfTestOffline` checks the value of the *seed*. If the value is zero then throw test fail exception; otherwise, proceed. The generated *seed* value is then input to the PUF that produces a sequence referred as k in algorithm 4.1. Using the generated k, the `SelfTestOffline` will decrypt the signature key for the given device, then return the signature key to the attestation handler. The handler will generate a signature and send it to the requesting entity (e.g. the SP) along with the relevant cryptographic certificate. If the signature verifies then the smart card state is in conformance to the evaluation state.

Algorithm 4.2: Self-test algorithm for online attestation based on a PUF

Input :
c; challenge sent by the card manufacturer.
n; random number send by the card manufacturer.
Output :
r; hash value generated on selected memory addresses, set at zero.
p; response part of the CRP for the implemented PUF.
Data:
$seedfile$; seed file that has a list of non-zero values.
$seed$; temporary seed value for the PRNG set to zero.
ns; number of entries in a $seedfile$.
s; unique reference to an entry in the $seedfile$.
nc; number of bytes in the n.
i; counter set to zero.
l; upper limit of memory address defined by the card manufacturer.
m; memory address.
mK; shared secret between a smart card and respective card manufacturer.
Notation:
x % y: represents x modulo y. This notation is common for all algorithms in this thesis.

```
1  SelfTestOnline (c, n) begin
2      mK ⟵ nmPUF(c)
3      while i < nc do
4          s ⟵ ReadSingleByte(n, i) % ns
5          seed ⟵ ReadSeedFile(seedfile, s)
6          m ⟵ GenPRNG(seed) % l
7          r ⟵ Hash(ReadMemoryContents(m), r, mK)
8          if (nc − i) = 1 then
9              p ⟵ nmPUF(r)
10         i ⟵ i+1
11     return r, p
```

For online attestation, the card manufacturers will have to generate (limited) Challenge-Response Pairs (CRPs) discussed in section 4.5.3, which will be unique to a device. The

rationale behind this is based on the design of a non-simulatable PUF in which the designer tries make the CRP space sufficiently large to make it difficult for an adversary to simulate the PUF [123, 135]. This design decision even makes it difficult for the card manufacturer to simulate the PUF. The limited set of generated CRPs will lead to a limited number of device validations (before they start to repeat), which is not a desirable situation. Therefore, we use a rolling update mechanism in which at the end of each successful device validation (section 4.7) a new CRP will be generated for future use. A valid CRP response can also help the card manufacturer ascertain that the device is not counterfeit as only the issued device's CRPs are registered in its CRP database.

The PUF-based online attestation mechanism represented in algorithm 4.2 implements a function `SelfTestOnline` that takes two parameters: a challenge 'c' and random number 'n' from the respective card manufacturer. The challenge 'c' is input to the PUF at line two and a response is generated, which is the response to the challenge 'c' and we treat it as a shared secret (mK). The function `SelfTestOnline` then treats the random number 'n' as a collection of bytes, reading one byte at a time and taking modulus of the byte with the length of the $seedfile$. By doing so, we generate an index to the $seedfile$ and in the next step we read a $seed$ value from that index. The $seed$ value is used to generate a new random number, whose modulus with upper memory limit (l) defined by the manufacturer gives us a memory location. In the next step (line seven), we read and hash the memory contents from the memory location, and the result is stored in "r". This process is repeated for the number of bytes the random number 'n' has, which is represented by the nc. At $nc - 1$ iteration, the "r" is input to the PUF again to generate a new CRP.

In function `SelfTestOnline`, the generated 'r' and 'p' are then securely communicated back to the smart card manufacturer, which can verify the generated 'r' and stores the CRP. The card manufacturer can verify the 'r' by executing instructions from lines three to seven of the algorithm 4.2. Similarly, the function `SelfTestOnline` does not send the challenge which was used to generate the response 'p' because the card manufacturer can also generate the value of 'r' at iteration $ns - 1$.

4.5.2 Pseudorandom Number Generator

In the second option, we propose the use of a Pseudorandom Number Generator (PRNG) to provide the device authentication, validation, and implicit anti-counterfeit functionality. Unlike non-simulatable PUFs, PRNGs are emulatable and their security relies on the protection of their internal state (e.g. input seed values, and/or secret keys, etc.).

Unlike PUFs, the PRNGs implemented in one device will be the same as they are in other devices and given the same input, they will produce the same output. Therefore, the

manufacturer will populate the PRNG seed file with unique values in each smart card. The seed file is a collection of inputs that is fed to the PRNG to produce a random number, and it is updated constantly by the PRNG [136]. This will enable a card manufacturer to emulate the PRNG and generate valid CRPs for a particular device. The PRNG mechanism is not tamper-evident and it relies on the tamper-resistant mechanisms of the smart card to provide physical security.

Based on the PRNG, algorithms 4.3 and 4.4 show the offline and online attestation mechanism, respectively.

Algorithm 4.3: Self-test algorithm for offline attestation based on a PRNG

Input : l; list of selected memory addresses.
Output: S; signature key of the smart card.
Data:
seed; temporary seed value for the PRNG set to zero.
n; number of memory addresses in the list l.
i; counter set to zero.
a; memory address.
k; secret key used to encrypt the signature key of the smart card.
S_e; encrypted signature key using a symmetric algorithm with key k.

```
1  SelfTestOffline (l) begin
2      while i < n do
3          a ⟵ ReadAddressList (l,i)
4          seed ⟵ Hash (ReadMemoryContents (a), seed)
5          i ⟵ i+1
6      if seed ≠ ∅ then
7          k ⟵ GenPRNG (seed)
8      else
9          return testfailed
10     S ⟵ DecryptionFunction (k, S_e)
11     return S
```

The `SelfTestOffline` takes a list of selected memory addresses l that is illustrated in algorithm 4.1. The function iterates through the l reading one memory address at a time, and then generating a hash of the contents stored at the given memory address. In the next step at line six, the function `SelfTestOffline` checks the value of *seed* and if it is not zero it will proceed; otherwise, it will throw a test fail exception. If the *seed* value is not zero then the *seed* is input to the PRNG and a sequence k is generated. The k is used to encrypt the smart card signature key, and if the input to the PRNG at line seven is as expected the signature key will be correctly decrypted.

The algorithm returns the signature key, which is used by the attestation handler to sign a message. The requesting entity will verify the signed message and if the state of the platform is in conformance with the evaluated state then the signature will be verified;

otherwise, it will fail. The signature verification will fail because the decrypted signature key will be different as the input to the PRNG at line seven of the algorithm was different. Therefore, we can assume that if the state is changed, signature key will change, and the generated signature will not verify.

Algorithm 4.4: Self-test algorithm for online attestation based on a PRNG

Input : c; randomly generated challenge sent by the card manufacturer.
Output: r; hash value generated on selected memory addresses.
Data:
$seedfile$; seed file that has a list of non-zero values.
$seed$; temporary seed value for the PRNG set to zero.
ns; number of entries in a seed file.
s; unique reference to an entry in the $seedfile$.
nc; number of bytes in the c.
i; counter set to zero.
l; upper limit of memory address defined by the card manufacturer.
m; memory address.
mK; HMAC key shared between a smart card and respective card manufacturer

```
1 SelfTestOnline (c) begin
2     while i < nc do
3         s ⟵ ReadChallenge(c, i) % ns
4         seed ⟵ ReadSeedFile(seedfile, s)
5         m ⟵ GenPRNG(seed) % l
6         r ⟵ r ⊕ Hash(ReadMemoryContents(m), mK)
7         i ⟵ i + 1
8     return r
```

The PRNG-based online attestation mechanism is illustrated in algorithm 4.4. The function `SelfTestOnline` takes the challenge c from the card manufacturer as input. The received challenge is treated as a collection of bytes and individual bytes of the challenge c are used to generate indexes to $seedfile$; values stored on these indexes are used to generate memory addresses (within the range specified by the card manufacturer). The contents of generated memory addresses are then HMACed and the result is securely sent to the card manufacturer. The SP can use the same process described in algorithm 4.4 to generate the HMAC result and if the result matches with the one sent by the smart card, then the card manufacturer can ascertain that the current state of the card is trustworthy. At line six of the algorithm 4.4, we update the $seedfile$ with the value stored in 'm'. This update is necessary to avoid generation of the same 'r' if the card manufacturer sends the same challenge 'c'.

In the implementation of the attestation protocol (section 4.7), we only use the PRNGs as we have neither the technical expertise to design a PUF nor adequate means to do so. However, in section 4.8.3, we emulate a PUF to provide the test performance measurements.

4.5.3 Challenge-Response Pair Generation

In the case of the mechanism based exclusively on the PRNG as depicted in algorithm 4.4, the card manufacturer will provide a set of seed values that is referred as the seed file. The seed file has a limited set of seeds and with the PRNG designed to update, the seed file will keep the internal state of the PRNG difficult to emulate by an adversary.

On the other hand, if the online attestation mechanism is based on PUFs then the card manufacturer requests the smart card to generate a limited set of CRPs. A new CRP is generated on every successful online attestation; therefore, the card manufacturer does not need to maintain an exhaustive set of CRPs for individual smart cards.

4.6 Device Ownership

An off-card entity can have one of two types of ownerships on a UCTD. These are discussed in subsequent sections.

4.6.1 Administrative Ownership

This ownership privilege is enabled in the UCTD to accommodate the requirements of an IT infrastructure in a corporate, government or public institution (i.e. schools, library, etc.) to manage hand-held and traditional computing platforms. In addition, the administrative ownership enables the scheme in which an organisation that is referred as administrative authority (i.e. MNOs, CIBs, TSOs, SCMs, and MPMs, etc.) can issue smart cards to its customers and may charge either the application provider or the user on each application download (section 3.6).

An entity with administrative privileges can install an application in the administrator space on a UCTD. The administrator space is an "application space" (section 4.2.1) on a UCTD that is under the control of the administrative authority. The user of the UCTD will not have any privilege to install or delete an application from the administrator space; the user only has the right to use these applications to acquire sanctioned services. The administrator space can enable the administrative authority to install certain protection applications (i.e. applications related to network/system user policy, firewall and antivirus definitions, and content filters, etc.). Furthermore, administrative ownership does not give the administrative authority the privilege to install, delete or use/access any application that is installed by a user in her "application space".

4.6.2 User Ownership

User ownership is associated with individual users that acquire a smart card either from a supplier or an administrative authority. This ownership gives the privilege to a user to install, delete, and use applications installed in her "application space". There are two scenarios in user ownership: 1) the UCTD is subscribed with an administrative authority (discussed in the previous section), and 2) there is no administrative authority on the UCTD, as in UCOM initiative [32].

In the first case, the user has to abide by the terms and conditions of the administrative authority. However, in the second case, there is no administrative authority and the user has complete freedom on the UCTD. Therefore, in the second case we can say that the user is the administrator and user at the same time.

4.6.3 Ownership Acquisition & Delegation

A UCTD in its pre-issuance state is under the default ownership of the UCTD manufacturer. When an entity, whether an administrative authority or a user takes control of the smart card, it will initiate an ownership acquisition process. The first step of the acquisition is to select whether the UCTD will be under administrative control or not. If it will be, then the administrative authority takes the administrative ownership and then issues the smart cards to individual users. Whether the UCTD is under administrative control or not, the user will then acquire the ownership privileges. The ownership acquisition process is same whether it is initiated by an administrator or a normal user; therefore, we will use the term user to indicate administrator and normal user during this section. The process is described below:

1. The user initiates the ownership acquisition process through the Card Application Management Software (CAMS). At this stage, the user will indicate the type of ownership (e.g. administrative or user) and CAMS will select the appropriate manager of the UCTD. For administrative ownership, it will select subscription manager (section 4.2.5) and for user ownership, it will select cardholder's security manager (section 4.2.4). In case, the UCTD will only have one owner then the smart card will disable the administrative ownership, Unless explicitly instructed not to do so by the cardholder.

2. The UCTD requests the default ownership credentials, which are communicated to the user by the card manufacturer. In response the user will provide the relevant default credentials.

3. On verification of the credentials, the UCTD checks the mode of platform assurance and validation selected by the user. The supported modes are offline and online attestation (section 4.3.5). Depending upon the user's choice the UCTD proceeds with the security attestation process.

4. Once the assurance validation is communicated to the CAMS, the user can compare the smart card features with those stated by the card manufacturer at the time of purchase. If satisfied, the user will provide her credentials and they are used to authenticate the user to the UCTD for management operations (e.g. application installation, deletion, and registration with an administrative authority) discussed in chapter 5. The credentials can be based on a Personal Identification Number (PIN), a password, a pass-phrase, or biometric data [137]–[139] depending upon the card manufacturer, and user's requirements.

The ownership delegation process is used when a user relinquished control of a UCTD to re-sell or scrap the device. The process is similar to ownership acquisition but this time the user requests ownership delegation that will delete the user's space and any applications she has installed in it.

4.6.4 Key Generation

Individual smart cards have a unique set of cryptographic keys that the card uses for different protocols/mechanisms during its lifetime. Therefore, after the hardware fabrication and masking of the SCOS is completed [5] the card manufacturer initiates the key generation process.

Each smart card will generate a signature key pair that does not change for the lifetime of the smart card. The smart card signature key pair is certified by the card manufacturer, and it is used to provide offline attestation (section 4.5). Furthermore, in the certificate hierarchy shown in figure 4.3, the smart card signature key pair is linked with the PAC via the card manufacturer's certificate. The reason for this is that a malicious user might copy a PAC that belongs to a genuine device and put it on his tampered device and when an SP requests security assurance from the tampered device, it provides the (copied) PAC of a (trusted) genuine device. By ensuring the PAC is tied to genuine devices by the certificate hierarchy shown in figure 4.3 we can avert such scenarios.

As discussed in section 4.4.2.1, the evaluation authority issues a certificate (e.g. a PAC) which certifies that the signature key of the card manufacturer is valid only for the evaluated product. If an adversary can get hold of the manufacturer's signature key pairs then he

can successfully masquerade as the smart card; either as a dumb device or by simulating the smart card on a powerful device like a computer.

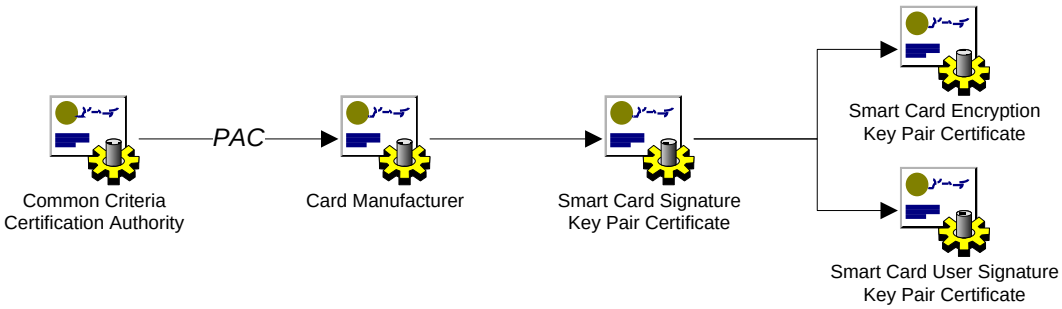

Figure 4.3: Certificate hierarchy in the UCOM

The smart card will also generate a public encryption key pair that is certified by the smart card signature key. The smart card user signature key pair is used to identify the owner of the device and to provide proof of ownership (see chapter 6). This signature key is unique to the individual user and it is generated on the successful completion of ownership acquisition process (section 4.6.3).

Finally, the smart card and card manufacturer share an encryption key for symmetric algorithms (e.g. TDES, AES) and a MAC key. These keys will be used to encrypt and for MAC communication messages between the smart card and the card manufacturer.

4.7 Attestation Protocol

The attestation protocol, referred as Attestation Protocol (ATP), involves the card manufacturer in the security assurance and validation framework by using the online attestation mechanisms. The aim of the protocol is to provide an assurance to a remote SP that the current state of the smart card is not only secure but also dynamically attested by the card manufacturer. The card manufacturer generates a security validation message that testifies to the requesting SP that its product is safe and still in compliance with the security evaluation indicated by the associated PAC.

4.7.1 Protocol Prerequisites

Before the execution of the attestation protocol, the prerequisites for the proposed protocol are listed below:

PPR-1 Third Party Evaluation: The smart card is independently evaluated by a third

party that certifies the security and reliability features of the device.

PPR-2 Attestation Mechanism: An attestation mechanism is implemented that provides an effective assurance of the tamper-evidence and conformance with the evaluated state of the smart card.

PPR-3 Authenticated & Valid CRP: To provide online device authentication and validation, the card manufacturer maintains a valid CRP database corresponding to individual smart cards.

PPR-4 Unique Identifier: Each smart card has a unique identifier that it can use to authenticate itself to the card manufacturer.

PPR-5 Pseudo Public Identifier: Each smart card has a dynamic pseudo public identifier that it uses to connect with the card manufacturer. Before issuing the smart cards to individual users, the card manufacturer will generate a unique pseudo identity for each card that will be updated on each successful execution of the attestation protocol.

PPR-6 Smart Card Signature Key Pair: Each smart card will have a unique signature key pair that is bound to the attestation mechanism.

PPR-7 Encryption & MAC Keys: The smart card manufacturer shares a unique encryption and MAC key with each of their individual smart cards. These keys are used to encrypt and MAC the communication messages between the smart card and its manufacturer.

4.7.2 Protocol Goals

The goals for the attestation protocol are listed as below:

PG-1 Secrecy: During the attestation protocol, the communication messages are adequately protected.

PG-2 Privacy: In the attestation protocol, the identity smart card owner (user) should not be revealed to any eavesdropper or the card manufacturer.

4.7.3 Intruder's Capabilities

The aim of an adversary $\mathcal{A}$ could be to retrieve enough information to enable him to successfully masquerade as a card manufacturer or as a smart card. Therefore, we assume

an adversary $\mathcal{A}$ is able to intercept all messages communicated between a smart card and its manufacturer. In addition, $\mathcal{A}$ can modify, change, replay, and delay the intercepted messages.

If $\mathcal{A}$ is able to masquerade as a card manufacturer then $\mathcal{A}$ can issue fake attestation certificates to individual smart cards, which might compromise the security and privacy of the user and related SPs. On the other hand, if $\mathcal{A}$ is able to compromise the smart card then he can effectively simulate the smart card environment (discussed in detail in section 5.5.1). This will enable him to reverse engineer the downloaded applications and retrieve sensitive data related to the user and application (e.g. intellectual property of the SP).

4.7.4 Protocol Notation and Terminology

Table 4.2 summarises the notation used in the proposed attestation protocol.

Table 4.2: Protocol notation and terminology

Notation	Description
$\mathcal{SC}$	Denotes a smart card.
$\mathcal{SP}$	Denotes a Service Provider.
$\mathcal{CM}$	Denotes the respective card manufacturer of the $\mathcal{SC}$.
$\mathcal{CC}$	Denotes the respective Common Criteria evaluation laboratory that evaluates the $\mathcal{SC}$.
SID	Session identifier that is used as an authentication credential and to avoid Denial of Service (DoS) attacks. The SID generated during the protocol run 'n' is used in the subsequent protocol run (i.e. n+1).
X_i	Indicates the identity of an entity X.
N_X	Random number generated by entity X.
$h(Z)$	The result of applying a hash algorithm (e.g. SHA-256) on data Z.
K_{X-Y}	Long term encryption key shared between entities X and Y.
mK_{X-Y}	Long term MAC key shared between entities X and Y.
B_X	Private decryption key associated with an entity X.
V_X	Public encryption key associated with an entity X.
$e_K(Z)$	Result of encipherment of data Z with symmetric key K.
$f_K(Z)$	Result of applying MAC algorithm on data Z with key K.
$Sign_X(Z)$	Is the signature on data Z with the signature key belonging to an entity X using a signature algorithm like DSA or based on the RSA function. In this thesis the message $Sign_X(Z)$ can be interpreted as either "signature with appendix" [140] or "signature with message recovery" [141]. The choice of a particular scheme is left to the discretion of communicating entities.

Notation	Description
$CertS_{X \leftarrow Y}$	Is the certificate for the signature key belonging to an entity X, issued by an entity Y.
$CertE_{X \leftarrow Y}$	Certificate for the public encryption key belonging to an entity X, issued by an entity Y.
VM	The Validation Message (VM) issued by the respective $\mathcal{CM}$ to a $\mathcal{SC}$ representing that the current state of the $\mathcal{SC}$ is as secure as at the time of third party evaluation, which is evidenced by the PAC (section 4.4.2.1).
$X \rightarrow Y : C$	Entity X sends a message to entity Y with contents C.
$X \| \| Y$	Represents the concatenation of data items X and Y.

4.7.5 Protocol Description

In this section, we describe the attestation protocol, and each message is represented by **ATP-n**, where n represents the message number. We use the same representation to describe each message of proposed protocols in this thesis. The structure of this representation would be the protocol acronym (i.e. ATP for attestation protocol) followed by the message number.

$$\begin{aligned} \textbf{ATP-1.} \qquad \mathcal{SC} \quad &: \quad mE = e_{k_{SC-CM}}(SC_i || N'_{SC} || CM_i || ReqVal) \\ \mathcal{SC} \rightarrow \mathcal{CM} \quad &: \quad SC_{i'} || mE || f_{mk_{SC-CM}}(mE) || SID \end{aligned}$$

Before issuing the smart card to the user, the $\mathcal{SC}$ and $\mathcal{CM}$ will establish two long term secret keys; encryption key K_{SC-CM} and MAC key mK_{SC-CM}. The $\mathcal{SC}$ and $\mathcal{CM}$ can use these long-term shared keys to generate the session encryption key k_{SC-CM} and the MAC key mk_{SC-CM}. The method deployed to generate session keys is left to the sole discretion of the card manufacturer. Each $\mathcal{SC}$ has a unique identifier $\mathcal{SC}_i$ that is the identity of the smart card. To provide privacy to each smart card (and its user) the identity of the $\mathcal{SC}$ is not communicated in plaintext. Therefore, the pseudo-identifier $SC_{i'}$ is used in the ATP-1, which is generated by the $\mathcal{SC}$ and corresponding $\mathcal{CM}$ on the successful completion of the previous run of the attestation protocol. We will discuss the generation of $SC_{i'}$ and SID in subsequent messages, as the generated $SC_{i'}$ and SID during this message will be used in the next execution of the attestation protocol. A point to note is that for the very first execution of the attestation protocol, the smart card uses the pseudo-identifier ($SC_{i'}$) that was generated by the card manufacturer and stored on the smart card before the card was issued to the user. The SID is used for two purposes: firstly to authenticate the $\mathcal{SC}$ and secondly, to prevent a Denial of Service (DoS) attack on the attestation server. The $ReqVal$ is the request for attestation process.

On receipt of the first message, the $\mathcal{CM}$ will check whether it has the correct values of

$SC_{i'}$ and SID. If these values are correct, it will then proceed with verifying the MAC. If satisfied, it will then decrypt the encrypted part of the message.

ATP-2. $\mathcal{CM}$: $mE = e_{k_{SC-CM}}(CM_i||N'_{SC}||N_{CM}||Challenge)$
$\mathcal{CM} \rightarrow \mathcal{SC}$: $mE||f_{mk_{SC-CM}}(mE)||SID$

The $\mathcal{CM}$ generates a random number N_{CM} and a *Challenge*. In case of the PRNG-based attestation mechanism, the *Challenge* would also be a random number; however, in case of PUF-based attestation mechanism it would be the pre-calculated challenge part of the CRP.

ATP-3. $\mathcal{SC}$: $mE = e_{k_{SC-CM}}(N'_{SC}||N_{CM}||N_{SP}||N_{SC}||Response||Optional)$
$\mathcal{SC} \rightarrow \mathcal{CM}$: $mE||f_{mk_{SC-CM}}(mE)||SID$

After generating the *Response* using the PRNG- or PUF-based algorithms discussed in section 4.5, the $\mathcal{SC}$ will proceed with message three. It will concatenate the random numbers generated by the $\mathcal{SC}$, $\mathcal{CM}$, and $\mathcal{SP}$, with the *Response*. The rationale for including the random number from the SP in message three is to request $\mathcal{CM}$ to generate a validation message that can be independently checked by the $\mathcal{SP}$ to ensure it is fresh and valid. The function of the *Optional* element is to accommodate the CRP updates if the $\mathcal{CM}$ implements a PUF-based attestation process.

While the $\mathcal{SC}$ was generating the *Response* based on the *Challenge*, the $\mathcal{CM}$ also calculates the correct attestation response. When the $\mathcal{CM}$ receives message three, it will check the values and if they match then it will issue the validation message. Otherwise the attestation process has failed and $\mathcal{CM}$ does not issue any validation message (VM).

ATP-4. $\mathcal{CM}$: $VM = Sign_{CM}(CM_i||SC_i||N_{SP}||N_{SC}||PAC)$
$\mathcal{CM}$: $mE = e_{k_{SC-CM}}(N'_{SC}||VM||SC^+_{i'}||SID^+||CertS_{CM})$
$\mathcal{CM} \rightarrow \mathcal{SC}$: $mE||f_{mk_{SC-CM}}(mE)||SID$

If the attestation response is successful then the $\mathcal{CM}$ will take the random numbers generated by the $\mathcal{SP}$ and the $\mathcal{SC}$ (e.g. N_{SP} and N_{SP}) during the Secure and Trusted Channel Protocols (STCPs) discussed in chapter 6 and include the identities of the $\mathcal{SC}$ and $\mathcal{CM}$. All of these items are then concatenated with the $\mathcal{SC}$'s evaluation certificate PAC and then signed by the $\mathcal{CM}$. The signed message is then communicated to the $\mathcal{SC}$.

In the ATP-4, the $\mathcal{CM}$ will also generate a SID and $SC_{i'}$ that will used in the subsequent execution of the attestation protocol between the $\mathcal{SC}$ and $\mathcal{CM}$. The SID and $SC_{i'}$ for the subsequent run of the attestation protocol is represented as SID^+ and $SC^+_{i'}$. The SID^+ is basically a (new) random number that is associated with the pseudo-identifier of the smart card that it will use to authenticate in the subsequent attestation protocol. Furthermore,

the $SC_{i'}^{+}$ is generated as $SC_{i'}^{+} = f_{mK_{CM}}(CM_i||N_{SC}||N_{CM}||SID)$, where mK_{CM} is the MAC key that the $\mathcal{CM}$ does not share

4.8 Protocol Analysis

In this section, we analyse the proposed attestation protocol for given goals and provide details of the test performance results.

4.8.1 Informal Analysis

In order to meet the goals PG-1 and PG-2, all messages communicated between the $\mathcal{SC}$ and $\mathcal{CM}$ are encrypted and MACed using long term secret encryption and MAC keys; K_{SC-CM} and mK_{SC-CM}, respectively. The $\mathcal{A}$ has to compromise these keys in order to violate the PG-1. If we consider that the symmetric algorithm used (e.g. AES) is sufficiently strong to avert any exhaustive key search and robust enough to thwart any cryptanalysis then it is difficult for the $\mathcal{A}$ to break the protocol by attacking the used symmetric algorithms. A possibility can be to perform side-channel analysis of the smart card and attempt to retrieve the cryptographic keys; however, most modern smart cards have adequate security to prevent this attack, and third party evaluation will endorse and evaluate these mechanisms. Nevertheless, these assurances can only be against the state-of-the-art attack methodologies at the time of manufacturing/evaluation. Any attacks which surface after manufacture and evaluation will render both the assurance and validation mechanisms useless.

The smart card identity is not used as plaintext during the communication between the $\mathcal{SC}$ and the $\mathcal{CM}$. Instead of using the $\mathcal{SC}_i$, the $\mathcal{SC}$ uses a pseudo-identity $\mathcal{SC}_{i'}$ which changes on every successful completion of communication with the respective $\mathcal{CM}$. Therefore, a particular $\mathcal{SC}$ will only use $\mathcal{SC}_{i'}$ once during its lifetime.

4.8.2 Protocol Verification by CasperFDR

The CasperFDR approach is adopted to test the soundness of the proposed protocol under the defined security properties. In this approach, the Casper compiler [142] takes a high-level description of the protocol, together with its security requirements. It then translates the description into the process algebra of Communicating Sequential Processes (CSP) [143]. The CSP description of the protocol can be machine verified using the Failures-Divergence Refinement (FDR) model checker [144]. A short introduction to the

CasperFDR approach to mechanical formal analysis is provided in appendix B.1. The intruder's capability modelled in the Casper script (appendix B.2) for the proposed protocol is as below:

1. An intruder can masquerade as any entity in the network.
2. It can read the messages transmitted by each entity in the network.
3. An intruder cannot influence the internal process of an agent in the network.

The security specifications for which the CasperFDR evaluates the network are as shown below. The listed specifications are defined in the # Specification section of appendix B.2:

1. The protocol run is fresh and both applications are alive.
2. The key generated by a smart card is known only to the card manufacturer.
3. Entities mutually authenticate each other and have mutual key assurance at the conclusion of the protocol.
4. Long term keys of communicating entities are not compromised.

The CasperFDR tool evaluated the protocol and did not find any attack(s). A point to note is that in this thesis, we provide mechanical formal analysis using CasperFDR for the sake of completeness and we do not claim expertise in the mathematical base of the formal analysis.

4.8.3 Implementation Results & Performance Measurements

The test protocol implementation and performance measurement environment in this thesis consists of a laptop with a 1.83 GHz processor, 2 GB of RAM running on Windows XP. The off-card entities execute on the laptop and for on-card entities, we have selected two distinct 16bit Java Cards referred as C1 and C2. Each implemented protocol is executed for 1000 iterations to adequately take into account the standard deviation between different protocol runs, and the time taken to complete an iteration of protocol was recorded. The test Java Cards (e.g. C1 and C2) were tested with different numbers of iterations to find out a range, which we could use as a common denominator for performance measurements in this thesis. As a result, the figure of 1000 iterations was used because after 1000 iterations, the standard deviation becomes approximately uniform.

Regarding the choice of cryptographic algorithms we have selected Advance Encryption Standard (AES) [145] 128-bit key symmetric encryption with Cipher Block Chaining (CBC) [146] without padding for both encryption and MAC operations. The signature algorithm is based on the Rivest-Shamir-Aldeman (RSA) [146] 512-bit key. We use SHA-256 [147] for hash generation. For Diffie-Hellman key generation we used a 2058-bit group with a 256-bit prime order subgroup specified in the RFC-5114 [148]. The average performance measurements in this thesis is rounded up to the nearest natural number.

The attestation mechanism implemented for emulating the practical performance is based on the PRNG design. The PRNG for our experiments was based on the HMAC-SHA256 [149] and it has been implemented such that it allows us to input the seed file. For completeness, we have taken the measurement of PUF-based algorithms in which all other instructions were executed on a Java Card and PUF execution time from [135] was added later. The performance measures taken from two different 16-bit Java Cards are listed in table 4.3. The offline attestation mechanism based on PRNG and PUF take in total (excluding PRNG seed file) 2084 and 2292 bytes respectively. Similarly, the online attestation mechanism and associated attestation protocol based on PRNG and PUF take in total (excluding PRNG seed file) 5922 and 6392 bytes respectively.

Table 4.3: Test performance measurement (milliseconds) for the attestation protocol

Measures	**Offline Attestation**				**Attestation Protocol**			
	PRNG		PUF		PRNG		PUF	
Card Specification	C1	C2	C1	C2	C1	C2	C1	C2
Average	408.63	484.55	532	584	1008	1284	1128	1284
Best time	367	395	506	495	930	1075	992	1075
Worse time	532	638	749	838	1493	1638	1312	1638
Standard Deviation	41.82	59.43	53.22	83.31	87.68	92.29	103.62	112.72

4.8.4 Related Work

The basic concept of remote attestation and ownership acquisition came from the TCG's specifications [36]. The user takes the ownership of the TPM and in return, the TPM generates a unique set of keys that are associated with the respective user. The remote attestation mechanism described in the TPM specification [18] provides a remote system attestation (only software). The attestation mechanism is designed so that if the software state is modified, the TPM cannot generate a valid report.

The TPM does not provide an attestation that includes the hardware state. Furthermore, the attestation defined in the TPM specification is more like the offline attestation. However, the offline attestation mechanism (algorithm 4.3) is different to the one used by TPM,

whereas the online attestation is not part of the TPM specifications.

Similarly, other proposals concentrate on the software attestation without binding it to a particular hardware. Such proposals include SCUBA [150], SBAP [151], and SWATT [152]. These protocols utilise execution time as a parameter in the attestation process. This is difficult to guarantee remotely, even with the delegation of time measurement to neighbouring trustworthy nodes [150]. Other mechanisms that use trusted hardware are proposed by Schellekens et al. [153] and PUF-based protocols [123, 135, 154].

There is no such proposal for remote attestation in smart card frameworks like Java Card, Multos, or GlobalPlatform. The nearest thing is the Data Authentication Pattern (DAP) in the GlobalPlatform card specification that checks the signature on the downloaded application (if the application provider chooses this option). Furthermore, we have opted out of having execution measurement as part of the attestation process as it is difficult to ascertain the trustworthiness of the remote device that measures it. However, unlike other proposed protocols we have an explicit requirement that third party evaluation is used to provide an implicit trust in the attestation process. Furthermore, our proposal binds the software attestation with the hardware protection (tamper-evident) mechanism to provide added assurance.

4.9 Summary

In this chapter, we discussed the overall architecture of the UCTD and its components and the ways in which the UCTD is different from mainstream smart card proposals. We also extended the discussion to the security assurance and validation framework that requires a third party evaluation and an attestation process. The attestation process includes hardware validation with the traditional software attestation. We proposed two modes for the attestation process: offline and online attestation. In designing the attestation processes, we based our proposal on two different architectures. First proposal is based on the PRNG and the second approach includes the PUFs in the device attestation process. To have an online attestation, we proposed the attestation protocol that communicates with the card manufacturer to get a dynamic certificate of assurance (a signed message from the card manufacturer) that the smart card is still secure and reliable. We implemented offline and online attestation mechanisms, along with an attestation protocol on 16-bit Java Cards. We also detailed the performance measurements of the implemented mechanisms and protocols.

Chapter 5

Smart Card Management Architecture

Contents

In this chapter, we discuss two of the most widely accepted and deployed smart card management architectures in the smart card industry, namely GlobalPlatform and Multos. We explain how these architectures do not fully comply with the UCOM. We then describe our novel card management architecture designed for the UCTD framework. Finally, we discuss three new security issues raised by the proposed architecture.

5.1 Introduction

Existing multi-application smart card platforms (e.g. Java [7], Multos [8]) support the installation of applications remotely (after issuance of the card). Standardisation efforts to manage an application remotely like GlobalPlatform [9] have been effective in the ICOM. The advent of NFC technology and the growing convergence of different services to mobile phones has prompted GlobalPlatform and the GSMA[1] to propose new management architectures (e.g. TSM) [43, 50, 155, 156]. Similarly, Multos has a strong card and application management architecture that is heavily issuer centric and it can be argued that it can easily be adapted to the TSM architecture.

The GlobalPlatform and Multos card and application management architectures provide two contrasting views of the smart card industry. We limit our discussion of traditional card management architectures with these two examples. As for the Java Card, it does not have any associated management architecture and in most commercial deployments it is coupled with the GlobalPlatform management architecture.

The device management architecture in the UCTD framework has to consider the contrasting needs of the administrative authority and cardholder. It must determine the ownership requirements of each of these entities and then articulate how a UCTD framework will manage them. In addition, the management architecture proposed in this chapter deals with application issuance (lease), application domain provision on the smart card, installation, deletion, and application/domain management. This chapter serves as the framework to the proposed protocols in the subsequent chapter.

As the UCTD management architecture brings different views on smart card management, it also brings new security issues. These issues concern the device and its owner. They include the simulator problem, the user ownership issue, and the parasite application problem.

Structure of the Chapter: The GlobalPlatform card management framework is discussed in section 5.2 followed by the Multos card management framework in section 5.3. The proposed framework of UCTD management is described in section 5.4, along with various types of relationships a user and an SP can have in the UCOM. In section 5.5, we discuss the issues that are raised due to the proposed UCTD management framework and related countermeasures.

[1]The GSM Association (GSMA) is an association that represents the interest of mobile operators worldwide along with developing and prompting the Global System for Mobile Communication (GSM) specification.

5.2 GlobalPlatform Card Management Framework

In this section we discuss the GlobalPlatform card management framework along with how it supports TSM-based card management.

5.2.1 Architecture Overview

The GlobalPlatform card security requirement specification [1], specifies nine entities that perform various tasks in the overall card management architecture. The overall architecture is depicted in figure 5.1, which is a simplistic representation of the architecture described in [1]. The figure is then explained.

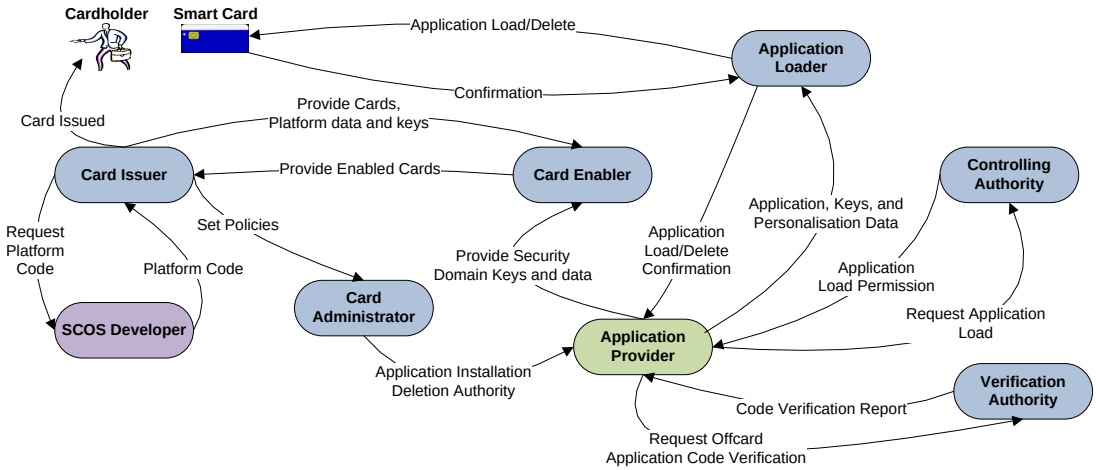

Figure 5.1: GlobalPlatform card management architecture [1]

The shaded entities in figure 5.1 have different titles and roles, but together they form the card issuer as defined in this thesis. The term "card issuer" as defined by GlobalPlatform in [1] is restrictive, so that they only have the responsibility to acquire the smart cards, set security policy, and issue them to individual cardholders. The card administrator then manages the cards once they are issued to individual customers. If application providers want to issue their applications, they first have to get the applications verified by the verification authority. The verification authority performs an off-card application code verification to ascertain whether the given code conforms to the security policy set by the card issuer. Once the verification is performed, the application provider requests the controlling authority to give permission to load the application. The controlling authority checks the verification authority's verification and issues the permission to load the application. Now, if the application is going to be loaded at the pre-issuance stage then the domain keys and data will be sent to the card issuer [157] through the card enabler. Otherwise, the domain keys and data will be sent to the application loader. In figure 5.1, we

opt for the pre-issuance model. Finally, the application provider will send its application to the application loader, which will load it onto the smart cards of individual customers.

In figure 5.1 the security domain keys that are used by the application provider to manage its domain are loaded onto the smart card through the card enabler (i.e. card issuer) [30]. However, a later addendum to the GlobalPlatform card specification [30] permitted the generation and loading of keys without the active involvement of a card issuer [158]. This extends the role of the Controlling Authority (GP-CA)[2] by giving it an on-card controlling entity (i.e. Controlling Authority Security Domain: CASD) that will be responsible for generating and/or loading the application provider's cryptographic keys. The GlobalPlatform specification supports two models: the push model in which the cryptographic keys are sent to the CASD by the application provider, and the pull model that generates the cryptographic keys on the card and then sends them to the application provider. The GlobalPlatform card specification [30] and its amendment [158] provides a secure and reliable way to load the application provider's keys onto a smart card in a confidential way. In all fairness, the proposal of CASD did not make any difference to the original GlobalPlatform card specification where the card enabler was loading the keys (pre-issuance loading) or in the GlobalPlatform card specification amendment A [158]. It is the CASD (post-issuance loading) that is under the control of the GP-CA. Both roles, card enabler and GP-CA, are predominantly played by the card issuer. Nevertheless, the amendment provides a base to accommodate the TSM architecture.

5.2.2 Support for Trusted Service Manager Architecture

To provide a standardised architecture and facilitate the adoption of NFC-enabled mobile phones for various services, GlobalPlatform proposed a framework for the management of secure elements in NFC mobile phones [155].

The role defined for the TSM by GlobalPlatform [50, 155] is to manage relationships between various actors in the ecosystem. It does not handle any key management or provide any trusted services [155]. GlobalPlatform proposes a new entity termed as the Confidential Key Loading Authority (CKLA) that will provide the initial key set in the smart cards in a confidential way. It does not specify who will take the role of the CKLA. Additionally, it breaks down the role of GP-CA so it can be performed by two different (independent) actors. This role includes managing the CKLA and enabling the Mandated Data Authentication Pattern (Mandated DAP) Authority. The CKLA will facilitate the generation and loading of keys through Over-the-Air (OTA) architecture. The DAP allows

[2] A Controlling Authority is an off-card entity (e.g. card issuer) that has a security domain on the GlobalPlatform smart card. Its role as defined in the GlobalPlatform card specification [30] is to enforce the card issuer's security policy. In the GlobalPlatform card specification, the GP-CA has the power to sanction or evict any application from a smart card.

the application provider to sign their applications before they are loaded onto the smart card. The Mandated DAP Authority will verify the signature and notify the application provider.

One thing to note is that in any framework, whether it is pre-issuance or post-issuance application loading in the GlobalPlatform card specification [30] or application loading via OTA in the NFC mobile phone [155], the loading of cryptographic keys is dependent on an entity (e.g. GP-CA or CKLA). The application provider has to trust these entities and their aim is to provide the key material for application loading and management to the respective application provider without revealing it to any malicious entity. Therefore, such entities (e.g. GP-CA or CKLA) which in most cases belongs to an off-card actor (i.e. card issuer), cannot be entertained in the UCOM proposal.

5.3 Multos Card Management Framework

In this section, we discuss Multos architecture for card management operations along with the possibility that the architecture can be accommodated into the TSM-based framework for NFC mobile phones.

5.3.1 Architecture Overview

The card management architecture for Multos is more straightforward than GlobalPlatform (section 5.2.1). An overview of the Multos card management architecture is illustrated in figure 5.2 and discussed below:

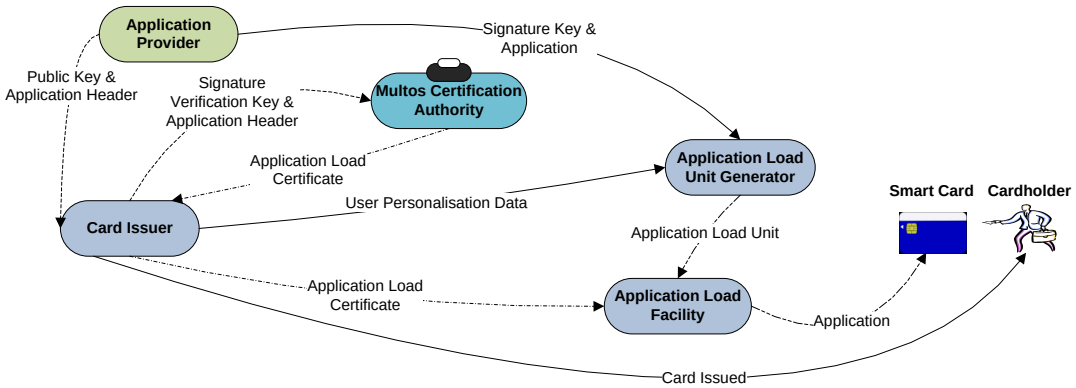

Figure 5.2: Multos card management architecture

The shaded entities in figure 5.2 represent various roles, but traditionally they reside with a single entity, for example the card issuer [97, 159]. An application provider will generate

a signature key pair and application code; the private key of the application provider along with the application code is securely communicated to the application load unit generator. The signature key is used to generate a cryptographically protected application load unit (i.e. downloadable application). The application provider will also send its signature verification key and application to the card issuer, which forwards it to the Multos Certification Authority (M-CA). The application header is a data structure that contains information regarding the respective application, which includes the application identity, hash of the application code, and code and data size [159]. The M-CA can be either the card manufacturer, or an authorised entity of the Multos consortium [29]. The role of the M-CA is to issue an application load or delete certificate to the card issuer for the application. In addition, the M-CA also provides the list of public keys for individual Multos cards that the card issuer has issued to its customers. This list of public keys is stored by the application load unit generator as the keys are used to encrypt individual applications. This transfer of public keys is marked as user personalisation data in figure 5.2. The application load unit generator will create individual application load units for individual smart cards, if the load unit has to be encrypted. Finally the load unit will have a signed digest of the application, using the application provider's signature key and if required the application encrypted by the respective smart card's public key. The application load facility now has the application load units and associated M-CA's issued certificates that it will use to download the application to individual smart cards.

As it is apparent from figure 5.2 that application providers have to communicate their application (in plaintext) and private key to the application load unit generator (i.e. card issuer). Furthermore, the application management tasks (i.e. installation, deletion, and updating etc.) have to be performed through the card issuer and/or M-CA. Unlike the GlobalPlatform, Multos specifications do not provide independent application management architecture. To delete an application, the process is similar to the Multos application installation except there is no need to generate the application load unit — only an application delete certificate that is similar to application load certificate is required from the M-CA.

5.3.2 Support for Trusted Service Manager Architecture

To date we have not seen any official proposal on how to incorporate Multos into the proposed TSM architecture for NFC mobile phones. However, in a centralised environment a TSM can take the role of the M-CA and application load unit generator.

5.4 Proposed Smart Card Management Framework

In the UCTD proposal, the management framework is divided into two categories based on whether the device is under administrative control or not. Therefore, these two categories are referred as administrative and user management, where administrative management corresponds to the CASC architecture and user management corresponds to the UCOM architecture.

5.4.1 Administrative Management Architecture

In the administrative management architecture, a smart card is under the shared ownership of an administrative authority and the respective cardholder (section 3.6). The framework is shown in figure 5.3 and the dotted lines in this figure represent optional messages.

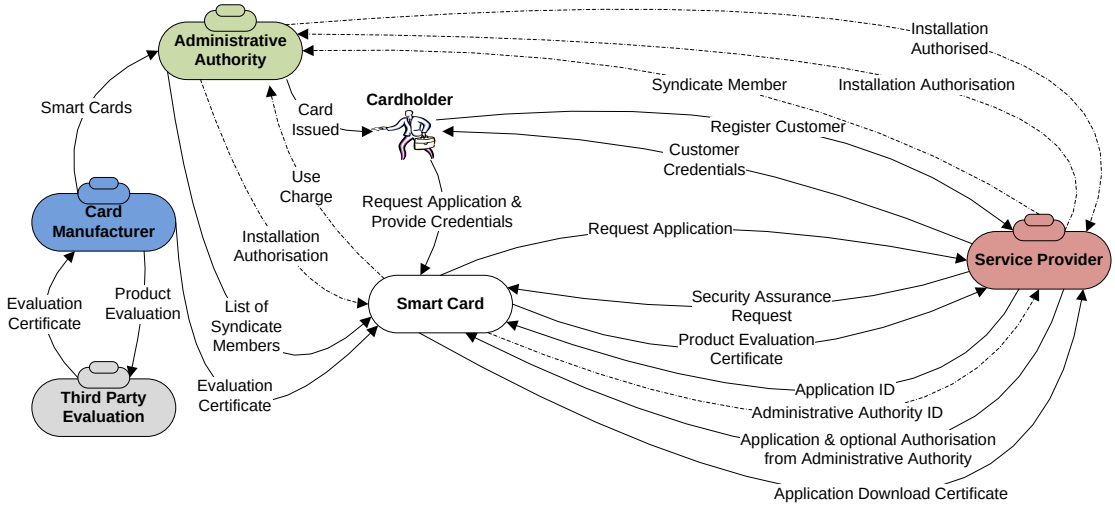

Figure 5.3: Administrative card management framework (CASC: section 3.6)

The card manufacturer gets its product evaluated by a third party that issues an evaluation certificate. The smart cards are then acquired by the administrative authority that takes administrative control and issues the cards to individual cardholders. The cardholder then has ownership, which is delegated to the cardholder under certain terms and conditions. The cardholder has to register with the relevant SP to gain access to their application. The registration process generates customer credentials that are issued by the SP and used by cardholders to download the application(s) onto their smart cards. The cardholder then provides these credentials to the smart card along with the details of the SP's application server (section 3.4.6.1). Before the SP leases its application, it requests the smart card to provide a security assurance, which is furnished by providing the evaluation certificate and a validation proof (section 4.4.3). The SP then sends the application identity to the smart card, which will check whether the application belongs to the administrative authority's

partner: partners are referred to as syndicated members. The smart card has the list of syndicated members that is provided by the administrative authority. If the SP is registered as a syndicated member of the administrative authority, then it will reveal the identity of the administrative authority. Under the scenario in which the SP is a syndicated member, the SP will then contact the administrative authority to authorise the installation. On successful authorisation, the application is leased to the smart card and installed in the administrative authority's space (section 4.2.1). If the SP is not a syndicated member, then the application is installed under the authorisation of the cardholder and she might be charged for it, which is represented by a "use charge" message sent from the smart card to the administrative authority. Subsequently, the administrative authority processes the request and issues an application "installation authorisation" message to the smart card. On receipt of this message, the smart card will allow the application to execute. It will generate an "application download certificate" that acts as a contract between the smart card and the SP. The contract signifies that the application was downloaded properly onto the smart card and it is activated to communicate with the SP.

The administrative management architecture can easily be adapted into the TSM architecture by replacing the administrative authority with the TSM. However, the shared ownership principle has to be accommodated by the TSM architecture to comply with the CASC.

In chapter 6, the Application Acquisition and Contractual Agreement Protocol is based on the administrative management architecture.

5.4.2 User Management Architecture

In the user management architecture there is no administrative authority. The user management architecture is shown in figure 5.4 and is described subsequently.

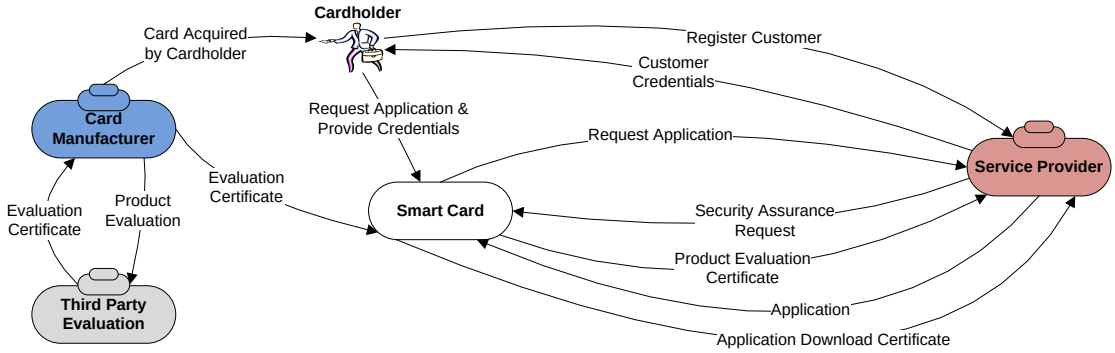

Figure 5.4: User card management framework (UCOM: section 3.4)

The smart card establishes a connection with the SP and this is a secure communication

channel, to provide smart card security and reliability assurance to the SP, facilitate in generating domain management credentials, and download the application.

In the UCTD, whether it is in administrative or user management, each SP gets its own domain. The SP's domain management credentials are mutually generated by the SP and smart card without involving any off-card entity (e.g. including the card manufacturer). Applications are directly downloaded to the SP's domain using the cryptographic keys mutually generated by the smart card and the SP, in contrast to GlobalPlatform that uses either a push or pull model for key sharing (section 5.2.1), or Multos that requires an application provider to reveal its application code and signature key (section 5.3.1).

In chapter 6, the two variants of the Secure and Trusted Channel Protocol (STCP) referred as STCP_{SP} and STCP_{SC} are based on the user management architecture.

5.4.3 Types of Application Leases

An application lease refers to issuance of an application to the requesting smart card under some terms and conditions that are stipulated by the Application Lease Policy (section 3.4.6.2). In this section, we discuss the various types of application leases that an SP can issue.

1. Card Bound Application Lease: In this lease, an SP issues its application to a specific smart card and that instance of lease is bound to it. Therefore, an SP will only issue one lease per user, which she can have on any of her smart cards; examples of such a lease may be credit card and (U)SIM card applications.

2. User Bound Application Lease: This lease is bound to the user, not to her smart card. She can install the given application on any number of her smart cards. Examples of such a lease may be Internet Identity applications [77, 160].

3. Open Application Lease: The open application lease does not bind the lease to either a user or a smart card. Any smart card, and any user can download this application. Examples of such applications may be pre-paid mobile telephone usage, pre-paid calling cards, hotel room access cards, and transport cards. One thing to note is that the examples are only valid if they do not require any registration of the user before and after issuance of the application.

5.4.4 Possible Relationships between a Cardholder and an SP

The lease issued to a cardholder discussed in the previous section would be based on a relationship that an SP has with a particular cardholder. In this section, we discuss various possible relationships that can exist between an SP and a cardholder.

1. Pre-Registration: This scenario deals with applications that are only issued to registered and pre-authorised customers. Such applications can be for banking, health centre, identity, travel documents, and telecom (e.g. post-pay accounts) that require proof that the requesting user is actually the current owner of the smart card. This relationship is valid for the card- and user-bound application leases discussed in the previous section.

2. Post-Registration: The post-registration relationship allows a cardholder to download an application without being a registered customer. However, the application does not go into service unless the user registers herself with the application (or its respective SP). This type of relationship can be valid for all three types of the application leases.

   There are two possible cases: a) the SP is only concerned with the security assurance and validation of the smart card platform and does not require user registration (anybody can download and use their application) or b) at least during the application lease process, the SP is not concerned with the user registration. However, once the application is downloaded the SP can initiate the user registration process. Option 'b' is like a user registering for a service for the first time. Examples of applications that can be downloaded in this scenario include pre-paid telecom applications, transport, and hotel room access applications.

3. No-Registration: This option does not require any registration, before or after the application is issued. It is suitable for the open application lease category. Examples include hotel room access cards, fixed pre-paid calling cards, and pre-paid gift cards.

5.4.5 Application Installation

In this section, the processes that support the secure transmission and installation of an application are discussed. The installation process discussed in this section builds additional checks around the application installation protocols (discussed in chapter 6).

The installation request will initiate the process of acquiring an application from an SP's application server (AMS discussed in section 3.4.6) and installing it on a smart card. The

entire process can be divided into three sub-processes: 1) Downloading, 2) Localisation, and 3) Application Registration. These sub-processes are explained as below.

1. Downloading: The downloading of an application is initiated by the smart card, through a secure channel protocol (chapter 6). At the conclusion of the secure channel protocol, both entities generate a set of keys for application download and domain management. The smart card then generates an SP's domain, provided it has enough space to accommodate it. The SP and smart card will then start the application downloading process. The SP will first generate a signature on the application, then encrypt and MAC it before sending it to the smart card.

   The smart card checks the generated MAC, decrypts the application, and verifies the signature. A decrypted application is not a fully installed application — it is the equivalent of copying an application to a memory location.

   The next step is to verify whether the application complies with the smart card's operational and security policy. For this purpose an on-card byte code verification is performed [161], which is already mandated by the Java Card 3 [16]; this can be based on the well-defined on-card byte code verification proposals [128, 161]–[163]. Furthermore, additional runtime checks are performed that are discussed in chapter 8.

   The UCTD does not mandate the security evaluation of an application. However, certain applications require evaluation due to government or industry regulations (e.g. EMV applications). In these cases, an SP's application(s) provides an evaluation certificate (e.g. AAC). To verify the certificate the smart card would have to calculate the hash of the downloaded application and compare it with the AAC.

2. Localisation: First, the application will be personalised by the SP. Depending upon the relationship between the cardholder and the SP, with the SP's discretion the personalisation can include acquiring user details (in post and no-registration scenarios), and cryptographic key generation. Furthermore, if the SP is issuing a card-bound lease then it will make sense to generate the on-card cryptographic keys as they will automatically become device identifiers because each lease of the application will have a various set of keys. After personalisation, the downloaded application establishes connection with various on-card services (i.e. shareable resources) that are provided by partner applications. To access a partner's application services, the downloaded application will establish an application sharing relationship that is discussed in detail in chapter 7.

3. Application Registration: The final stage of an application installation is the application registration by the SP. The registration will allow the particular instance of the application to access sanctioned services. Once the SP registers (sanctions) the downloaded application, the smart card will also make it selectable to an off-card

entity. By making an application selectable, the smart card allows the application to execute, access on-card services and communicate with off-card entities.

5.4.6 Application Deletion

The application deletion process has similar steps to the application installation but they are taken in the opposite direction. The installed application will first establish a connection with the SP and signal the deletion. It will initiate the de-registration process that will restrict the leased application's access to the SP's services. The smart card will also make it un-selectable for off-card entities; in addition, any interdependencies will be resolved. As most of the interdependencies that the deleted application might have are the result of the application sharing mechanism, the smart card firewall mechanism will cascade the deletion event to the related (partner) applications (chapter 7). The interdependencies that might exist between various applications on a smart card may end up creating the feature interaction problem that is discussed later in section 9.3. Finally, the SP's domain key material, and registration with various card services are deleted by the Card Security Manager (section 4.2.2).

5.5 Card Management-Related Issues

In this section, we discuss the potential issues related to card management architecture introduced by the UCTD framework.

5.5.1 Simulator Problem

In the context of the UCTD framework, the simulator problem refers to a possible scenario in which a malicious user could remotely install an application onto a smart card simulator. One thing to note is that the simulator problem is only related to remote installation and not to on-site installation. In remote installation, a smart card is not present at an SP's site, and the application is downloaded over the internet. Therefore, an SP needs a way of making sure that its application is not installed on a simulated device.

It can be asserted that simulators are used in a number of different environments, especially mobile application development, and do not present a substantial security issue in the mobile application environment. Nevertheless, the nature of an application installed on a smart card is different to an application on a mobile phone. The smart card application might represent the identity of the user, along with serving as a security token to

access some services (including financial services). Furthermore, the service or business environment that a smart card application deals with is substantially different from that of a mobile phone application.

In the ICOM, the simulator problem is not relevant, as applications are predominantly installed by the card issuer before the smart cards are issued to individual users. This stage in the smart card lifecycle is also referred as the pre-issuance stage. The GlobalPlatform card specification provides the framework for application installation under the application provider's control, after the smart cards are issued to customers. GlobalPlatform defines a secure entity on the smart card referred to as the Card Manager, along with associated domains [30]. The SP requires symmetric keys in order to gain access to the domains (application domains) and install applications. The assumption in the ICOM is that malicious users cannot access or retrieve these keys. The basis of this assumption is the tamper-resistance properties of the smart card hardware — that is, the assumption is based on trust in the card manufacturer or a third party evaluator (e.g. Common Criteria [69] evaluation laboratory).

In the UCOM, the problem is not only verifying the existence of a smart card, but also validating that it is in a secure and reliable state. In a simulator attack, a malicious user has a stand-alone simulator that enables him to simulate the UCTD environment. To do so, the adversary has to have knowledge of the cryptographic keys (section 4.6.4) and any related attestation mechanism (regardless of whether it is based on PRNGs or PUFs: section 4.5). If the attestation mechanism is based on PUFs then the adversary should be able to emulate the PUF for a genuine smart card. He can then try to acquire an application from an SP, install on the simulator, and attack it; that may include reverse engineering the application, retrieving the sensitive user and application data (e.g. cryptographic keys).

5.5.1.1 Countermeasure to the Simulator Problem

The countermeasure to the simulator problem has to deal with physical and side-channel attacks along with the risk of compromising the communication protocol(s). The countermeasure is based on three aspects of the UCTD architecture that are listed as below:

1. Security evaluation of the smart card platform, which certifies that the smart card is tamper-resistant, and effective against state-of-the-art attacks (section 4.4.2).
2. Self and remote attestation mechanism (section 4.4.3).
3. A secure and reliable entity authentication and key sharing protocol (chapter 6).

The evaluation and validation of the smart card provides (time limited) assurance against simulator attack. If during the lifetime of the smart card, an attack vector is discovered that can compromise the card's security and make the simulator attack feasible, the evaluation authority (and the card manufacturer) can revoke the certificates, and the SP can blacklist the affected smart cards. Because of such attacks, smart cards can be rendered useless. Therefore, card manufacturers, even today, are compelled to build a strong product or otherwise security issues would damage the reputation of their brand and the same would be true in the case of the UCTDs. Finally, the secure channel protocol should be designed in a way that would make it impossible for an adversary to retrieve the shared keys between a smart card and an SP.

To provide protection against simulator attacks, SPs can request for device attestation as described in section 4.7. The device attestation is based on a protocol, which involves the card manufacturer at the time of application lease attesting that its smart card is secure and reliable as stated by the evaluation certificate (section 4.4). Therefore, an evaluation certificate from an independent third party will confirm that the smart card is secure against complete and partial simulations. The online attestation mechanism provides a validation that the smart card is still in conformance with the state in which it was evaluated. Finally, by integrating the smart card assurance and validation mechanism into the secure channel protocols, the UCTD can avoid simulator problems.

5.5.2 User Ownership Issues

In this section, we discuss an issue that is related to the identity of the smart card owner and the authorised user that can download an application from an SP.

This issue arises in the pre-registration relationship (section 5.4.4) between an SP and a cardholder. During an application installation, a cardholder will provide her credentials to the SP that leases the application. In this situation, the SP requires that the application can only be downloaded to a smart card that is under the ownership of the authorised user.

The aim of an adversary may be to acquire the credentials of an authorised user for a particular SP to use them to download the application onto his smart card. If the SP does not issue card-bound leases (section 5.4.3), both entities (the authorised user and the adversary) can download the application.

To avoid this situation, the SP could require proof of ownership to be produced by the smart card for the given user during the application download process. The proof of ownership can be based on a signature key pair belonging to the user, which is certified by

the smart card itself or by its card manufacturer. The issue is that a similar certificate can also be requested by an adversary for the given authorised user if he knows enough personal information relating to the genuine user. Having the smart card sign the certificate is easy, and it also allows the user independence to sell/give her smart card to other users. On the other hand, including the card manufacturer in the user identification and issuance of certificates to individual users provides similar protection as the previous proposal, without effectively increasing the overall security. Therefore, we prefer the smart card to sign the certificate rather than the card manufacturer, and this approach is adopted in this thesis.

Another possibility is that the smart card user can register her smart card physically (offline) with the SP. The smart card can later access the SP server through the internet to download the application. If there are no adequate checks during the offline registration, an adversary could perform the same process and then use the credentials associated with the user and request the application lease. Furthermore, this solution also complicates the relationship between the user and the SP as there may not always be a possibility to have physical access to the SP's office (e.g. online gaming website).

An optimum solution can be based on one-time credentials issued by SPs. We assume that an SP issues a one-time credential (e.g. password) to a user to download its application to her smart card. Therefore, even if an adversary was to gain access to the credentials, he cannot use them to download the application. Furthermore, if an application is already leased to a user, an SP can reject any subsequent requests for an application lease unless the user either deletes the previous lease or loses the smart card. Therefore, in the case that the application is deleted and the user wants to install the application on her new smart card. The SP will issue a new one-time credential to the user, to download the respective application.

A malicious user cannot fake the deletion of the application, as in the UCOM deletion process (section 9.3) an application communicates with the SP in order to notify it of the deletion and also to perform any required housekeeping tasks. Therefore, the SP knows beforehand that the application is deleted so for a malicious user it is difficult to fake application deletion. Furthermore, if a user loses her smart card then she can use the authorisation token (issued by the SP and discussed in section 9.2) to acquire the application. An authorisation token is a short encrypted structure issued by an SP and it acts as a authorisation credential to download an application. If she does not have the authorisation token, then the user can contact the SP and request the issuance of new credentials. This is similar to the process after a user loses her smart card in the ICOM, where she has to contact the card issuer to get a new smart card.

5.5.3 Parasite Application Problem

In the parasite application problem, an installed application masquerades as the UCTD on which it is installed. This is possible because a UCTD allows an installed application to request the platform/application state validation from the TEM (section 4.3).

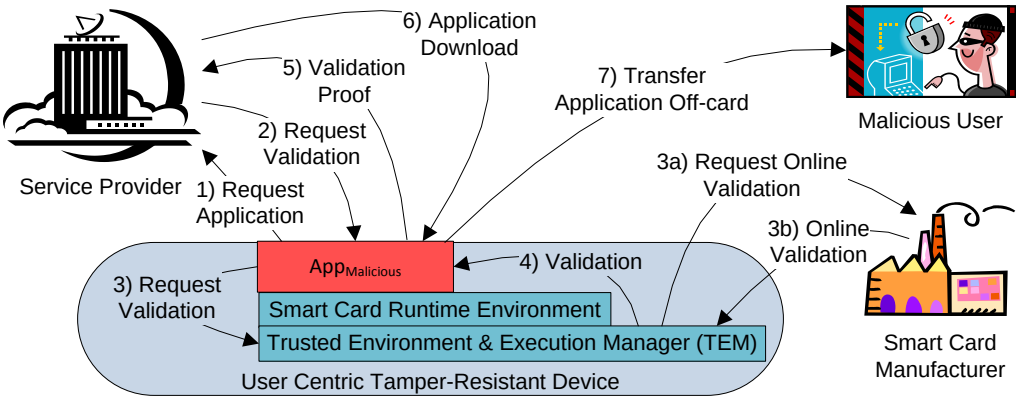

Figure 5.5: Illustration of parasite application problem

In this attack as shown in figure 5.5, an adversary ($\mathcal{A}$) installs his malicious application ($App_{Malicious}$) on a UCTD. The $App_{Malicious}$ implements the application download protocols that are discussed in chapter 6. The $\mathcal{A}$ then requests the installation of an application from its SP through the $App_{Malicious}$, represented by message one in the figure 5.5. In response, the SP asks the UCTD of security validation to avoid a simulator attack and this request is sent to the $App_{Malicious}$. The $App_{Malicious}$ then asks the TEM for the security validation (section 4.4.3). At the successful conclusion of the security validation process, the card manufacturer produces a certificate that for privacy reasons does not include the identity of the requesting SP as described in the online attestation protocol (section 4.7). Therefore, the $App_{Malicious}$ can communicate this certificate to the requesting SP as validation in message five (figure 5.5) and may be able to start the application download process. The $\mathcal{A}$ might have designed the application as if it will communicate the downloaded application off-card, which will enable $\mathcal{A}$ to retrieve the application code and data.

To avoid this problem, there are four possible solutions: 1) restrict the security assurance validation request to off-card entities and any installed applications should not be allowed to request it, 2) include the identity of the requesting SP in the security assurance validation certificate during the application installation process, 3) include the identity of the application requesting the security assurance validation during the application installation, the request is initiated by the card security manager discussed in section 4.2.2, or 4) avoid generating the signature as part of the security validation if it is requested by an application, but use the shared keys between the TEM and the application.

From the above listed options, we consider that option four is appropriate for the UCTD environment to prevent the parasite application problem. In option four, the TEM will only sign the security validation proof if it is requested by the card security manager and not by an application. Therefore, an application installed on a UCTD cannot request a signed security validation proof that would have enabled the application to masquerade as the respective UCTD.

5.6 Summary

In this chapter, we discussed the card management architectures for two contrasting frameworks: GlobalPlatform and Multos. GlobalPlatform is more open to independent application management by application providers, whereas Multos is a hardcore ICOM architecture that requires an authorisation from a centralised authority (i.e. Multos Certification Authority) before an application can be installed or deleted. Furthermore, Multos also requires that application providers should reveal their application codes to the Multos application load unit generator.

We then described the card management architecture for the UCTD followed by the application lease types, and the various kinds of relationships a user can have with an SP. Next, we discussed the application installation and deletion approach. Finally, we discussed new security issues including simulator, user ownership, and parasite application problems.

Chapter 6

Secure and Trusted Channel Protocol

Contents

In this chapter, we begin with a discussion of secure channel protocols that are used for entity authentication and key establishment for internet services. We discuss the security and operational goals that a secure channel protocol has to accomplish in the UCTD environment. We propose two protocols that closely adhere to the UCOM philosophy and a protocol related to the CASC that involves an administrative authority (e.g. TSM). An informal analysis is provided for the proposed protocols followed by a mechanical formal analysis using CasperFDR. Finally, we describe a prototype implementation of the proposed protocols, and give performance measurements obtained from this implementation.

6.1 Introduction

Secure Channel Protocols (SCPs) are designed to provide a secure communications channel. They typically start by providing entity authentication and authenticated key establishment. There are many different protocols proposed for internet and smart card environments that satisfy different (pre-defined) design goals. Not all protocols can be used for every possible scenario; if this was possible, we would not have the diversity of SCPs that we have today.

The UCTD architecture has its own set of security and operational goals for an SCP has to satisfy. These goals range from traditional SCP ones like entity authentication and mutual key generation, to UCOM-specific requirements like smart card state validation. In this chapter, we examine a non-exhaustive list of security and operational goals for the UCTD environment. The defined list is considered adequate to gauge the basic security requirements of the UCTD and related stakeholders (section 3.5).

An SCP in the UCTD can take many different forms, and we discuss three possible variants. The first two variants are based on the UCOM architecture and the differences between them is determined by who initiates the protocol, and whether it is a smart card or an SP. The third protocol caters to the CASC environment and involves an administrative authority during the protocol execution.

The proposed protocols are informally analysed, based on the pre-defined security and operational requirements. We also provide a comparison between the proposed protocols and a set of protocols ranging from the internet and smart card environments. We also provide a formal-mechanical proof for the proposed protocols using the CasperFDR tool, along with test implementation and performance measurements.

Structure of the Chapter: The chapter begins with a discussion of the rationale behind the SCPs in section 6.2. In this section we also discuss minimum security and operational requirements stipulated for the UCTD environment. Section 6.3, discusses the proposed SCP that is initiated by an SP, and section 6.4 details the SCP initiated by a smart card. The SCP that focuses on the administrative management architecture of the UCTD (section 5.4.1) is described in section 6.5. Section 6.6 provides the informal analysis, mechanical formal analysis and test implementations of the proposed protocols.

6.2 Secure Channel Protocols

In this section we explore the rationale behind SCPs for the UCTD, and then discuss the relevant work in the field of SCPs. This discussion forms the basis for work later in this thesis.

6.2.1 Rationale

By definition, an SCP provides either (or both) entity authentication or key exchange between communicating parties, referred to as end points. The SCP preserves the confidentiality and integrity of the messages communicated on the channel but does not necessarily assure the same security at the end points after the messages are received. Despite this, there can be implicit confidence in the integrity and security of the end points in the ICOM as articulated by ETSI TS 102 412 [164, section 4.5.2]. This states that the smart cards are a secure end point under the assumption that it is a tamper-resistant device.

This implicit assumption is valid for the traditional smart card environment because smart cards are issued by a "trusted" card issuer. This became the fundamental assumption in most of the smart card-based SCPs. For the ICOM, this assumption makes sense as the strict control of application installation on a smart card will effectively restrict the SCP to only execute with an entity that: a) has prior authorisation from the card issuer, or b) is initiated by an on-card authorised entity (e.g. installed application).

In the ICOM, there is a centralised authority that controls issued smart cards and their application management, enabling an implicit assurance attainable for the smart card security and reliability. However, in the UCOM, there is no such authority, hence the assumption of implicit assurance is no longer valid. The UCTD is required to provide an explicit assurance of its integrity and security to the requesting SP to satisfy requirements GR2, CR1, SCR1, SCR6, and SPR1–5 (section 3.5).

A trusted channel is a secure channel that is cryptographically bound to the current state of the communicating parties [165]. This state can be a hardware and/or software configuration, and ideally, it will require a trustworthy component to validate that it is the same as claimed. Such a component is in most instances a Trusted Platform Module (TPM) [18] as demonstrated by Zhou and Zhang [166], and Armknecht et al. [167].

The SP will probably not have any prior trust relationship with a smart card in the UCTD environment (an exception might exist in the CASC framework when the SP is a syndicated member of the administrative authority). Therefore, the traditional smart card SCPs will

fail to provide: a) assurance that an SP is communicating with a genuine smart card platform and not a simulator, b) assurance that the smart card security and operational environment is certified by a reputed third party evaluation, c) assurance that the security and operational environment state is still valid, as it was at the time of evaluation, and d) assurance that the smart card is owned by the user who is requesting the application download (user/card-platform binding authentication).

We define the Secure and Trusted Channel Protocol (STCP) in the context of the UCTD environment as a protocol providing a secure and reliable communication channel between a smart card and an SP, coupled with an assurance of security and integrity concerning the communicating smart card. The STCP can be used during: a) application installation/deletion processes, and b) when the application communicates with its respective SP, and vice versa.

6.2.2 Related Work

In this section, we restrict ourselves to a discussion of the protocols that are specifically proposed for the smart card environment and/or are being used as points of comparison in later discussions. Detailed descriptions of the discussed protocols is provided in appendix A, and this section will introduce these protocols.

Ever since the possibility arose that two computing devices could communicate with each other, there has been research work on SCPs. An early discussion on various proposed protocols can be found in [146]. A detailed comparison of authentication protocols for the mobile network environment is presented in [168].

Early smart card protocols were based on the symmetric key crypto-system like SCP01 of the GlobalPlatform specification [30] (this protocol is deprecated in the GlobalPlatform card specification version 2.2). Other protocols specified by the GlobalPlatform specification are: SCP02 (based on Triple-DES), SCP10 (based on asymmetric key crypto-system) [30], SCP81 (based on SSL/TLS) [169], SCP03 (based on AES) [170], and SCP80 for the mobile telecom industry (based on symmetric key crypto-system) [171]. In addition to this, entity authentication, key exchange, and application download protocols for the smart card environment are proposed by [83, 172, 173].

The concept of trusted channel protocols was put forward by Gasmi et al. [165] along with the adaptation of the TLS protocol [100] to meet the trusted channel requirements. Armknecht et al. [167] propose another adaptation of OpenSSL to accommodate the concept of the trusted channel, as do Zhou and Zhang [166]. However, at the time of writing we were unable to find any work that relates to the concept of the trusted channels for the

smart card environment.

In section 6.6, we compare the proposed STCP with the existing protocols. These protocols include the Station-to-Station (STS) protocol [174], the Aziz-Diffie (AD) protocol [175], the ASPeCT protocol [176, 177], Just-Fast-Keying (JFK) [178], trusted TLS (T2LS) [165], SCP81 [169], Markantonakis-Mayes (MM) protocol [83], and the Sirett-Mayes (SM) protocol [173].

For brevity and clarity, details of these protocols are provided in appendix A except for the GlobalPlatform SCP10. A point to note is that GlobalPlatform SCP10 provides the guidelines on how to implement a public key-based SCP for smart cards and not the actual protocol. The guidelines stipulated by the GlobalPlatform SCP10 are the core design requirement of the MM protocol and for this reason we have chosen this protocol.

The selection of the listed protocols is intentionally kept broad to include well-established protocols like STS, Aziz-Diffie (AD) and JFK. Also included is the ASPeCT protocol, which is designed specifically for the mobile network's value-added services. The T2LS is based on the concept of trusted channels, whereas SCP81, SM, and MM protocols are specific to smart cards. As a common criterion, we have only selected protocols whose design is rooted in asymmetric crypto-systems.

6.2.3 Minimum Security and Operational Goals

For a protocol to support the UCTD framework, it should meet at minimum the security and operational requirements listed below:

SOG-1. *Mutual Entity Authentication*: A smart card and an SP authenticate to each other to avoid masquerading by a malicious entity.

SOG-2. Exchange of certified public keys between the entities to facilitate the key generation and entity authentication process.

SOG-3. Mutual Key Agreement: Communicating parties will agree on the generation of a key during the protocol run.

SOG-4. Joint *Key Control*: Communicating parties will mutually control the generation of new keys to avoid one party choosing weak keys or predetermining any portion of the session key.

SOG-5. *Key Freshness*: The generated key will be fresh to the protocol session to protect replay attacks.

SOG-6. Mutual *Key Confirmation*: Communicating parties will provide implicit or explicit confirmation that they have generated the same keys during a protocol run.

SOG-7. Known-Key Security: If a malicious user is able to obtain the session key of a particular protocol run, it should not enable him to retrieve long-term secrets (*private keys*) or *session keys* (future and past).

SOG-8. Unknown *Key* Share Resilience: In the event of an unknown key share attack, an entity $\mathcal{X}$ believes that it has shared a key with $\mathcal{Y}$, where the entity $\mathcal{Y}$ mistakenly believes that it has shared the key with entity $\mathcal{Z} \neq \mathcal{X}$. Proposed protocols should adequately protect against this attack.

SOG-9. *Key* Compromise Impersonation (KCI) Resilience: If a malicious user retrieves the long-term key of an entity $\mathcal{Y}$, it will enable him to impersonate $\mathcal{Y}$. Nevertheless, key compromise should not enable him to impersonate other entities to $\mathcal{Y}$ [179].

SOG-10. *Perfect Forward Secrecy*: If the long-term keys of communicating entities are compromised, this will not enable a malicious user to compromise previously generated session keys.

SOG-11. Mutual *Non-Repudiation*: Communicating entities will not be able to deny that they have executed a protocol run with each other.

SOG-12. Partial Chosen Key (PCK) Attack Resilience: Protocols that claim to provide joint key control are susceptible to this type of attack [180]. In this type of attack, if two entities provide separate values to the key generation function then one entity has to communicate its contribution value to the other. The second entity can then compute the value of its contribution in such a way that it can dictate its strength (i.e. it is able to generate a partially weak key). However, this attack depends upon the computational capabilities of the second entity. Therefore, proposed protocols should adequately prevent PCK attack.

SOG-13. Trust Assurance (Trustworthiness): The communicating parties not only provide security and operation assurance but also validation proofs that are dynamically generated during the protocol execution [56].

SOG-14. Denial-of-Service (DoS) Prevention: The protocol should not require the server (in our case the SP's application server) to allocate the resources before authenticating and validating the state of the requesting entity (a smart card) or verifying the credentials of the authorised user.

SOG-15. Privacy: A third party should not be able to know the identities of the user or her smart card, over either the internet or Over-the-Air (OTA). In addition, during the trust validation and assurance process, the requesting SP should not be able to gain any additional information about the platform (e.g. applications installed on a smart card).

SOG-16. Simulator Attack Resilience: This attack discussed in [85] allows a malicious user to masquerade as a smart card platform on a computer (as a simulation). Such a possibility would enable the malicious user to download an application onto a simulated platform and then perform reverse engineering on the downloaded application, revealing proprietary and sensitive data of the application. Therefore, the proposed protocols should take into account the simulator attack and support countermeasures.

SOG-17. Platform & Application User Separation (PAU) Attack Resilience: This attack is discussed in [85]. A malicious user provides the access credentials of a genuine user to an SP and downloads an application onto his or her smart card. Any protocol should tie a platform with its card-owner (user) to avoid platform & application user separation attack.

SOG-18. Contractual Agreement: On the successful execution of the protocol, the communicating entities will mutually sign a contractual agreement. This will act as proof that a particular application was installed on a smart card.

SOG-19. Proof of Transaction: The smart card will notify the TSM about the application installation. Depending upon the TSM's policy, it will charge the user's account and notify the smart card to activate the application so it can execute.

For a formal definition of the terms (italicised) used in the above list, readers are advised to refer to [146]. The requirements listed above are later used as a point of reference to compare the proposed protocols in table 6.2 (section 6.6.3). From an operational point of view, an STCP for the user management architecture (section 5.4.2) for the UCTD environment has two variants: STCP_{SP} and STCP_{SC} discussed in sections 6.3 and 6.4, respectively. On the other hand, in an STCP for the CASC, the administrative management architecture (section 5.4.1) requires the inclusion of an administrative authority and to accommodate this, we propose an Application Acquisition and Contractual Agreement STCP (STCP_{ACA}) in section 6.5.

6.2.4 Protocol Notation and Terminology

The notation used in the protocol description is listed in table 6.1 below. This notation is an extension of the notation described in table 4.2.

Table 6.1: Protocol notation and terminology

Notation	Description
U	Denotes a smart card owner (user).
AD	Denotes the administrative authority (section 5.4.1).

Notation	Description
g^{r_X}	Denotes the Diffie-Hellman exponential generated by the entity X with random number r_X. We use this notation to represent the $g^{r_X}\ mod\ p$, where g and p are system parameters that are represented by the Diffie-Hellman group selected by the entity X. For further discussion please refer to [132, 146, 181]
K	Denotes the shared secret generated by the communicating entities using the Diffie-Hellman scheme. Keys for application download and session keys are generated from this shared secret.
ek_{X-Y}	Denotes the session encryption key shared between entities X and Y to be used with a symmetric algorithm.
mk_{X-Y}	Denotes the session MAC key shared between entities X and Y.
$h(Z)$	Represents the result of generating a hash of data Z by a hash function (e.g. SHA256 [147]).
U_{Cre}	User authentication credential (e.g. login and password) associated with a particular SP.
X_{Sup}	Denotes the supported features of entity X that include Diffie-Hellman groups [148], user authentication mechanisms (i.e. login/password), symmetric and signature algorithms.
AU_X	A signed message from an entity X that authenticates it to other entities.
SI	Session cookie generated by the respective SP. It indicates the session information and facilitates protection against DoS attacks, possibly along with providing the facility of protocol session resumption.
VR	Validation request sent by an SP to a smart card. In response, the smart card provides the security and reliability assurance to the SP.
ADP	The Application Download Protocol (ADP) will include appropriate parameters for the application download protocol, which in the context of this thesis is the GlobalPlatform application download process based on the symmetric key cryptosystem (e.g. SCP03) [170].
ALP	The Application Lease Policy (ALP) specifies the minimum security, reliability and operational requirements imposed by the SP on a smart card. The ALP also includes the relevant application's details that include application size and functionality support requirements.
hs & hp	hs is a hash message generated by the SC on data including its identity, generated Diffie-Hellman exponentials and random numbers. Similarly, hp is generated by the SP. Both these messages aim to avoid a man-in-the-middle attack on the proposed protocols.

Notation	Description
OP	Optional message.
OC	Optional certificate.

6.2.5 Pre-protocol Process

A smart card user can initiate a protocol in two ways depending upon how an SP makes its applications available to users. As depicted in figure 3.6, an SP can offer its application through different computing devices including dedicated kiosk machines. For example, a transport application can be acquired through kiosk machines installed on train stations or bus stops. The user can connect directly to an SP's Application Management Server (AMS: discussed in section 3.4.6) through the kiosk and request the application download. Another option is that an SP offers the application download through the internet and for this, the SP will provide the AMS details (e.g. Universal Resource Locator: URL).

On another note, it is difficult to protect the user's credentials if the host device is compromised. The aim of the STCPs is to provide security assurance to the smart card, not to the host device (e.g. mobile phone, desktop computer). From an SP's point of view, it can choose different ways to protect the credentials that it has issued to its customers — for example, by using one-time passwords for application download. Once a user has downloaded the application, the password expires. A new one-time password will only be issued to the user once she deletes the previously leased application. In addition, certain measures can be taken to protect the host device but that will require additional hardware and software support from the host device, which is beyond the scope of the STCPs.

6.2.6 Protocol Assumptions

The assumptions which apply during the execution of the proposed protocols between entity $\mathcal{SC}$ and $\mathcal{SP}$ that generates a session key K_{SC-SP} are listed below.

PrA-1 Attestation Mechanism: The $\mathcal{SC}$ provides a valid and trusted attestation mechanism, both offline and online depending upon the requirement of the $\mathcal{SP}$ (section 4.4).

PrA-2 Pseudorandomness: Random numbers generated by $\mathcal{SC}$ and $\mathcal{SP}$ are indistinguishable from truly random numbers to all parties (except for the $\mathcal{SC}$ and $\mathcal{SP}$) and the malicious entity that has compromised either the session between $\mathcal{SC}$ and $\mathcal{SP}$ or one of the communicating entities (e.g. $\mathcal{SC}$ and $\mathcal{SP}$).

PrA-3 Secure Cryptographic Algorithms: The cryptographic algorithms used in the protocol that include symmetric, asymmetric, and signature algorithms are secure against a computationally bound adversary.

6.3 Secure and Trusted Channel Protocol — Service Provider

In this section, we begin the discussion with a description of the proposed STCP_{SP} along with the rejection messages.

6.3.1 Protocol Prerequisites

The prerequisites to the STCP_{SP} are listed below. This is an extension to the prerequisite list in section 4.7.1.

PPR-8 User Signature Key Pair: When a user has taken ownership of a smart card and on the successful conclusion of this process, the smart card generates a user signature key pair. This key pair is used to provide proof of ownership during the STCP_{SP}.

PPR-9 Authorised Customer: The user is a registered customer of the SP, which means that the SP has sanctioned the user to download (lease) their application.

PPR-10 Established Connection: The user has the knowledge of the respective SP's application server (AMS: figure 3.6) that the SP has provided to the smart card. The smart card in return connects with the SP. Furthermore, the SP has knowledge of the smart card's Internet Protocol (IP) address.

6.3.2 Protocol Description

In this protocol, the SP takes the role of the protocol initiator. The design of this STCP variant is inspired by the requirements of user authentication as discussed in section 5.4.4.

STCP_{SP}-1.

$$
\begin{aligned}
SP &: SI = f_{k_{SP}}(g^{r_{SP}}||N_{SP}||SC_{IP}) \\
SP \rightarrow SC &: SP_i||VR||N_{SP}||g^{r_{SP}}||SP_{Sup}||SP_{Sel}||SI \\
SC &: K = (g^{r_{SP}})^{r_{SC}} \ mod\ p \\
SC &: ek_{SC-SP} = H_K(N_{SP}||N_{SC}||'1') \\
SC &: mk_{SC-SP} = H_K(N_{SP}||N_{SC}||'2')
\end{aligned}
$$

The SP generates a random number N_{SP} and computes the Diffie-Hellman exponential $g^{r_{SP}}$. The SP_{Sup} deals with the capabilities of the SP along with the details of how it will authenticate the user (e.g. password, biometric, or token based authentication, etc.). These details communicate to the smart card the way the SP would like to perform the user authentication. The MAC $f_{k_{SP}}(g^{r_{SP}}||N_{SP}||SC_{IP})$ serves as a session cookie (SI), and it is appended with each subsequent message sent by the smart card. It indicates the session information and facilitates protection against DoS attacks. Finally, the SP will request the smart card to provide an assurance that its current state is the same as it was at the time of third party evaluation by sending the VR. The VR indicates whether the SP requires an offline or online attestation (section 4.5) to be performed by the smart card.

If the smart card does not support the Diffie-Hellman group selected by the SP (SP_{Sel}), then it will send a rejection message, including a list of groups supported by the smart card (SC_{Sup}). If the smart card supports the selected group (i.e. SP_{Sel}), then it will proceed with the second message. The SC generates a random number, and a Diffie-Hellman exponential $g^{r_{SC}}$. It can then calculate the K which is the shared secret from which the rest of session keys (k_{SC-SP} and mk_{SC-SP}) will be generated. Furthermore, in a similar manner, we can generate more session keys for the application download protocol [170].

STCP$_{\mathbf{SP}}$-2.

$$
\begin{aligned}
SC &: hs = h(SC_i||SP_i||g^{r_{SC}}||g^{r_{SP}}||N_{SP}||N_{SC}) \\
SC &: AU_{SC} = Sign_{SC}(SC_i||SP_i||VM||hs) \\
SC &: mE = ek_{SC-SP}(AU_{SC}||Cert_{SC}) \\
SC \rightarrow SP &: g^{r_{SC}}||N_{SC}||SC_{Config}||mE||f_{mk_{SC-SP}}(mE)||SI
\end{aligned}
$$

The type of validation mechanism the TEM executes will depend upon the choice of the SP, which will generate a (valid) signed message if the attestation is successful. In the case of online attestation, the SC receives a validation message (VM) from the respective CM, and it will include VM in the AU_{SC} message. If the SP selects offline attestation, then the VM will not be included. Beside VM, the signed message also includes the identities of the smart card and the SP, along with the hs. The hs includes the identities of the communicating entities, the generated Diffie-Hellman exponentials and random numbers. The hs verifies to the SP that SC has used the same values (e.g. Diffie-Hellman exponentials and random numbers) as the SP, thereby avoiding potential man-in-the-middle attacks. The signed message AU_{SC} will be different if the state of the platform is modified; therefore, by verifying the signature the SP can ascertain the current state of the platform (in offline attestation mode). In the case of online attestation, the CM will not issue VM to the SC and the SC will not be able to proceed with the protocol. In the message, the SC includes SC_{config} that provides the SP with the configuration of the SC including supported cryptographic algorithms and APIs.

On receipt of message two, the SP will check the hs to avoid main-in-the-middle and replay attacks, and it will then check whether the SC_{Config} satisfies SP's ALP. Subsequently, it

will generate the session keys, verify the MAC and decrypt the message, and then verify the smart card certificate. The smart card certificate gives the required assurance that the smart card platform has had a third party evaluation, and the SP will then proceed with verifying the signature. The assumption here is that if the third party evaluation has concluded that the smart card is a tamper-resistant device with an effective validation mechanism, then it will be difficult for a malicious user to obtain the TEM keys. Hence, in the presence of a tamper-evidence mechanism only a genuine TEM can generate the correct signature. However, if the SP cannot verify the signature, then the current state of the SC has been modified and it is different to the one for which it was evaluated.

As a next step, the SP ascertains whether it has already issued an application lease to the stated smart card. If there is an application lease to the SC for the requested application, the protocol will terminate. Otherwise, the SP will proceed to the next step.

STCP$_{\text{SP}}$-3.

$$
\begin{aligned}
SP &: hp = h(SP_i||SC_i||g^{r_{SP}}||g^{r_{SC}}||N_{SP}||N_{SC}) \\
SP &: AU_{SP} = Sign_{SP}(SP_i||SC_i||hp||ALP) \\
SP &: mE = ek_{SC-SP}(AU_{SP}||ADP||CertS_{SP}) \\
SP \rightarrow SC &: mE||f_{mk_{SC-SP}}(mE)||SI
\end{aligned}
$$

The SP will then sign the identities of both the SC and SP along with the ALP and hp. The hp is similar to the hs but it is generated by the SP and provides the necessary evidence to the SC that the message is not a replay or mirror message while at the same time avoiding man-in-the-middle attack. The signed message is appended to the ADP and SP's certificate.

On receipt, the SC will verify whether it can support the listed requirements in the ALP. The most important requirement is whether the SC has enough memory space to accommodate the SP's application. Furthermore, the SC will also check the hp to prevent man-in-the-middle and replay attacks.

STCP$_{\text{SP}}$-4.

$$
\begin{aligned}
SC &: AU_U = Sign_U(SC_i||SP_i||U_i||hp||hs) \\
SC &: mE = ek_{SC-SP}(U_{Cre}||AU_U||CertS_U) \\
SC \rightarrow SP &: mE||f_{mk_{SC-SP}}(mE)||SI
\end{aligned}
$$

The SC requests the cardholder to provide the SP's authentication credentials as requested by the SP in the SP_{Sup}. After the user provides those credentials they are packaged as U_{Cre} and are concatenated with a signed message (AU_U) containing the identities of the SC, SP and user along with the hp and hs. The reason behind including the hp and hs in the signature is to provide a proof, signed by the user's signature key pair, that she initiated the protocol session. The signed message along with the certificate and the user's credentials are then encrypted and MACed. The MAC is generated on the encrypted message.

The SP verifies the U_{Cre} and if the user is authenticated then the SP will proceed with the protocol. Otherwise, it terminates after a limited number of user authentication retries. Subsequently, it will verify whether the user (owner) identity referred in the $CertS_U$ is the identity of an authorised and authenticated user. If so, then the SP will verify the signature. Furthermore, the U_{Cre} will provide the SP with an assurance that the user is cryptographically bound with the smart card (i.e. has the ownership of the smart card).

6.4 Secure and Trusted Channel Protocol — Smart Card

In this section, STCP_{SC} is described along with the rejection messages. Before we provide a description of the STCP_{SC}, a point to consider is that we adopt the protocol prerequisites discussed in section 6.3.1 and 4.7.1 with the exception of PPR-8, and PPR-9. The STCP_{SC} does not require PPR-9. However, if an SP needs to authenticate a user, the SP can implement the user authentication into their application and execute once the application is installed and active on the smart card. Based on this user authentication, the SP can then personalise the application with respective user's data. Furthermore, as the SC initiates the protocol a connection is not necessary between the SC and SP before the SC sends the first message as required by the PPR-10.

6.4.1 Protocol Description

In this protocol, an SC takes the initiator's role, with the respective SP as a responder. The protocol details and a description of the messages involved are presented below:

$$\begin{array}{lrcl} \textbf{STCP}_{\textbf{SC}}\textbf{-1.} & SC & : & cm = f_{N_{SC}}(g^{r_{SC}}||N_{SC}) \\ & SC \rightarrow SP & : & cm||SC_{Sup} \end{array}$$

An SC generates a Diffie-Hellman exponential ($g^{r_{SC}}$) and a random number (N_{SC}). Subsequently, it generates the MAC of the $g^{r_{SC}}||N_{SC}$ using the generated random number as the MAC key. The reason for generating the MAC and sending it instead of the random number and Diffie-Hellman exponential is to avoid a partial chosen key attack by only providing a commitment to the SP. The SC_{Sup} lists the Diffie-Hellman groups, cryptographic algorithms and attestation mechanism supported by the SC.

On receipt of the first message, the SP will verify the features listed in the SC_{Sup}. If they satisfy the SP's requirements then it will proceed with the protocol.

STCP$_{SC}$-2.

$$
\begin{aligned}
SP &: SI = f_{k_{SP}}(g^{r_{SP}}||N_{SP}||cm||SC_{IP}) \\
SP \rightarrow SC &: VR||g^{r_{SP}}||SP_i||N_{SP}||ALP||SP_{Sel}||SI \\
SC &: K = (g^{r_{SP}})^{r_{SC}} \ mod\ p \\
SC &: ek_{SC-SP} = H_K(N_{SP}||N_{SC}||'1') \\
SC &: mk_{SC-SP} = H_K(N_{SP}||N_{SC}||'2')
\end{aligned}
$$

The SP will also generate a Diffie-Hellman exponential and a random number. Finally, it will calculate the SI which includes similar elements to those discussed in STCP$_{SP}$ except for the inclusion of the commitment cm from the SC. The entire message is then appended with the VR.

On receipt of this message, the SC verifies the ALP. If the SC can accommodate the requirements then it will proceed with the protocol. The SC can now generate the shared secret "K", which is used to generate the session encryption and MAC keys. Furthermore, depending upon the decision of the SP as to whether it requests for an offline or online attestation, the SC will proceed with the appropriate attestation mechanism.

STCP$_{SC}$-3.

$$
\begin{aligned}
SC &: hs = (SC_i||SP_i||g^{r_{SC}}||g^{r_{SP}}||N_{SC}||N_{SP}) \\
SC &: AU_{SC} = Sign_{SC}(SC_i||SP_i||hs||VM) \\
SC &: mE = ek_{SC-SP}(AU_{SC}||CertS_{SC}) \\
SC \rightarrow SP &: g^{r_{SC}}||N_{SC}||SC_{Config}||mE||f_{mk_{SC-SP}}(mE)||SI \\
SP &: cmc = f_{N_{SC}}(g^{r_{SC}}||N_{SC})
\end{aligned}
$$

The SC will reveal the $g^{r_{SC}}$ and N_{SC}, which is appended by a message that is encrypted and MACed using the session keys. The encrypted and MACed message contains a signature generated on the identities of SC and SP, hs, along with VM. If the SP requests the online attestation then the AU_{SC} will contain VM generated by the respective card manufacturer; whereas, in case of offline attestation the AU_{SC} will not include VM.

On receipt of the STCP$_{SC}$-3, the SP will generate a commitment similar to the SC in message one, which we term as cmc. If the cmc is equal to the cm, then the SP will generate the shared secret, along with session encryption and MAC keys. The SP verifies the MAC and decrypts the message. It validates the $CertS_{SC}$, and then verifies the signature. If the SP accepts the current state of the smart card as secure then it will proceed with the next message.

STCP$_{SC}$-4.

$$
\begin{aligned}
SP &: hp = (SP_i||SC_i||g^{r_{SP}}||g^{r_{SC}}||N_{SP}||N_{SC}) \\
SP &: AU_{SP} = Sign_{SP}(SP_i||SC_i||hp||ADP) \\
SP &: mE = ek_{SC-SP}(AU_{SP}||CertS_{SP}) \\
SP \rightarrow SC &: mE||f_{mk_{SC-SP}}(mE)||SI
\end{aligned}
$$

The SP will generate an authentication message AU_{SP} that contains the identities of the

SP and SC, hp, and the ADP. On receipt of this message the SC first verifies the signature and ADP. Subsequently, the SC will initiate the application download process using the application download protocols (e.g. SCP03 [170]).

6.5 Application Acquisition and Contractual Agreement Protocol

In this section, we detail the STCP that, unlike the two protocols discussed above, includes the administrative authority (section 5.4.1). We begin the discussion with the enrolment phase that enables an administrative authority to be part of the architecture. Later we describe the STCP_{ACA} and discuss rejection messages.

6.5.1 Enrolment Phase

The enrolment phase deals with the inclusion (registration) of an administrative authority with the respective smart card. The registration can be either pre-acquisition or post-acquisition of the smart card by its user. In both cases, the process will generate a cryptographic certificate issued by the respective administrative authority to the registered smart cards. This will change the certificate hierarchy discussed in section 4.6.4, which is shown in figure 6.1.

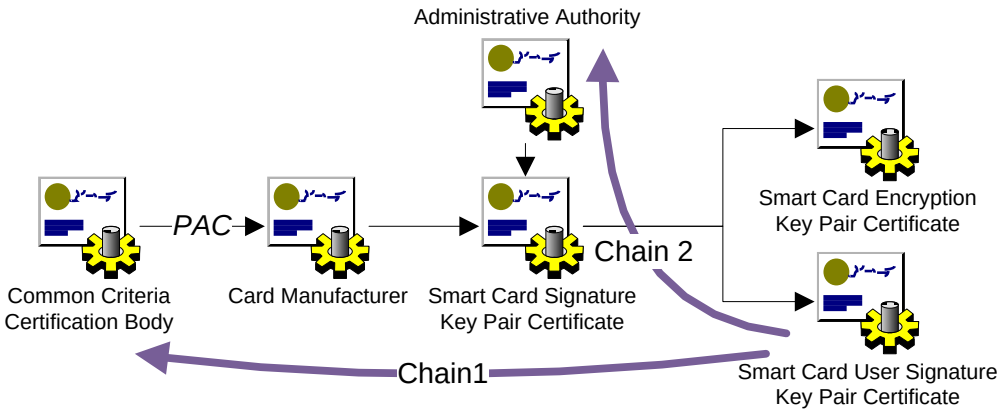

Figure 6.1: Certificate Hierarchy in the CASC

There are two roots in this hierarchy as illustrated by chain one and chain two in figure 6.1: the CC certificate authority, and the administrative authority. The reasons for having two separate roots are: a) to provide privacy protection to users who do not want to reveal the identity of their administrative authorities, and b) some smart cards may not be permanently bound with a particular administrative authority. If the administrative authority was the only root, then it would be difficult to satisfy the second and fourth

requirements listed in section 3.6 as they require a mechanism that is independent of the administrative authority.

During the STCP_{ACA}, depending upon the relationship between an SP and the administrative authority of a smart card, the appropriate chain of certificate will be used. Therefore, if the SP is not an associate of the administrative authority then the certificate chain (chain 1 in figure 6.1) with the CC certification body as a root will be used; otherwise, chain 2 of the figure 6.1 will be used.

6.5.2 Protocol Prerequisites

In this section, we extend the protocol prerequisites for the STCP_{ACA} from the ones discussed in section 6.3.1 and 4.7.1.

PPR-11 Administrative Authority Registration: The smart card is registered with an administrative authority.

PPR-12 Long Term Keys: Both the smart card and the associated administrative authority share long-term encryption and MAC keys. These keys are generated at the time of the smart card's registration with an administrative authority.

PPR-13 List of Syndicated Members: When a card registers with an administrative authority, the authority may provide the smart card two lists: a list of subscription charges and a list of associated SPs. The first list contains details on how the user will be charged on installation of the individual applications. For example, charging mechanisms can either be based on fixed charges per installation or according to the size of the application. The associated SPs list includes the details of individual SPs that are associated with the administrative authority. If a user requests installation of any of these applications, the installation goes through the administrative authority and user may not be charged.

6.5.3 Protocol Description

In the STCP_{ACA}, an SP takes the initiator's role so it can be considered as an extension of the STCP_{SP}. The protocol details and message description are as follows:

STCP$_{\text{ACA}}$-1.

$$
\begin{aligned}
SP &: SI = f_{k_{SP}}(g^{r_{SP}}||N_{SP}||SC_{IP}) \\
SP \rightarrow SC &: SP_i||N_{SP}||g^{r_{SP}}||SP_{Sup}||VR||ALP||SI \\
SC &: K = (g^{r_{SP}})^{r_{SC}} \ mod \ p \\
SC &: k_{SC-SP} = H_K(N_{SP}||N_{SC}||'1') \\
SC &: mk_{SC-SP} = H_K(N_{SP}||N_{SC}||'2')
\end{aligned}
$$

The SP will initiate the STCP$_{\text{ACA}}$ by generating a random number (N_{SP}) and Diffie-Hellman exponential ($g^{r_{SP}}$). It appends the generated values with the SP_{Sup} and associated ALP.

When the SC receives the message, it check whether it can meet the SP's ALP and support features from the SP_{Sup} list. The SC will then generate the shared secret and required session encryption and MAC keys.

STCP$_{\text{ACA}}$-2.

$$
\begin{aligned}
SC &: hs = h(SC_i||SP_i||g^{r_{SC}}||g^{r_{SP}}||N_{SC}||N_{SP}) \\
SC &: AU_{SC} = Sign_{SC}(VM||SC_i||SP_i||hs) \\
SC &: mE = ek_{SC-SP}(AU_{SC}||OP||CertS_{SC}) \\
SC \rightarrow SP &: N_{SC}||g^{r_{SC}}||SC_{Config}||mE||f_{mk_{SC-SP}}(mE)||SI
\end{aligned}
$$

The SC will generate a random number (N_{SC}) and Diffie-Hellman exponential ($g^{r_{SC}}$). Subsequently, the SC will proceed with generating the AU_{SC} and may include VM depending upon the SP's requirement (e.g. online or offline attestation) pointed out in the SP_{Sup}. If the SP is a member of the AD syndicate, the SC will include an OP containing the certificate issued to the SC by the AD. Each SC has a list of members associated with the respective AD, which can be regularly updated by the AD.

On receipt of the STCP$_{\text{ACA}}$-2, the SP will first verify the session cookie and the SC's capabilities listed in SC_{Config}. The SP will then generate the shared secret and session keys similar to the SC. Subsequently, it will verify the MAC and decrypt the message. The SP then verifies the generated signature and VM (if required); if successful the SP will proceed with the protocol.

STCP$_{\text{ACA}}$-3.

$$
\begin{aligned}
SP &: hp = h(SP_i||SC_i||g^{r_{SC}}||g^{r_{SP}}||N_{SC}||N_{SP}) \\
SP &: AU_{SP} = Sign_{SP}(SP_i||SC_i||App_i||hp) \\
SP &: mE = ek_{SC-SP}(AU_{SP}||CertS_{SP}||OC) \\
SP \rightarrow SC &: mE||f_{mk_{SC-SP}}(mE)||SI
\end{aligned}
$$

The SP will generate an encrypted and MACed message that contains AU_{SP}, SP's certificate, and an optional certificate OC. The optional certificate field is used by the SP if its application also has a third party evaluation certificate (AAC: section 4.4.2). The AU_{SP} includes the identities of the SP and the respective application along with hp.

On receipt of the STCP_{ACA}-3, the SC will check whether the SP 's identity is included in the associated SP's list (section 6.5.1). If the SP's identity is in the list then the response message the SC will include the administrative authority's identity as an optional parameter (OP).

STCP_{ACA}-4.

$$\begin{aligned} SC &: AU_U = Sign_U(SC_i||SP_i||U_i||hs) \\ SC &: mE = ek_{SC-SP}(U_{Cre}||AU_U||CertS_U||OP||ADP) \\ SC \rightarrow SP &: mE||f_{mk_{SC-SP}}(mE)||SI \end{aligned}$$

The SC will generate a user authentication message AU_U, which contains the identities of the SC, SP and user along with hs . The AU_U is appended with the user's certificate and then encrypted and MACed using the generated session keys.

After receiving message four, the SP will check whether the U_{Cre} belongs to an authorised user or not. If it does, then it will verify the signature, which will act as proof of ownership from the user. If user is authenticated then the SP will initiate the application download. Once the application download is completed, the STCP_{ACA} will proceed with the next message.

STCP_{ACA}-5.

$$\begin{aligned} SC &: sca = Sign_U(h(App)||SP_i||App_i||ALP||SC_i||U_i||hp) \\ SC &: mE = ek_{SC-SP}(sca) \\ SC \rightarrow SP &: mE||f_{mk_{SC-SP}}(mE)||SI \end{aligned}$$

Once the application download is completed, the SC will generate a message that acts as an SC to SP contract. This message contains the hash of the downloaded application, identities of the SP, SC, user, and the application along with the ALP under which the application was leased.

The SP will verify the signature and generated digest on its leased application. This will ensure that the application is downloaded properly on to the SC.

STCP_{ACA}-6.

$$\begin{aligned} SP &: amE = ek_{SP-AD}(SP_i||SC_i||U_i||App_i||hs||hp) \\ SP \rightarrow AD &: amE||f_{mk_{SP-AD}}(amE) \\ AD &: ActApp = ek_{SC-AD}(AD_i||SC_i||U_i||SP_i||App_i||hs) \\ AD \rightarrow SP &: OP = Sign_{AD}(AD_i||SP_i||ActApp||hp)||CertS_{AD} \\ SP &: spc = Sign_{SP}(SC_i||U_i||SP_i||hs||hp||OP) \\ SP &: mE = ek_{SC-SP}(spc||CertS_{SP}) \\ SP \rightarrow SC &: mE||f_{mk_{SC-SP}}(mE)||SI \end{aligned}$$

To activate an application, the SC requires the AD's authorisation. If the SP is associated with the AD then it will send the identities of the SC, the user, and the hs and the downloaded application to the respective AD. The AD in reply will generate the $ActApp$.

The *ActApp* acts as an application activation message and it will be included in message six as an optional parameter (OP). In this scenario, the last two messages will be redundant and will not be executed. The session keys k_{SP-AD} and mk_{SP-AD} are generated from long terms keys shared between the SP and AD. Similarly, the session key k_{SC-AD} is also generated from the long term key shared between the SC and AD

The SP will generate the contract message (*spc*) that certifies to the SC that the SP is satisfied with the current state of the SC and the downloaded application.

The SC will verify the *spc*. Subsequently, if the SP is a member of the AD syndicate, then it will verify the OP. If the SP is not a member of the AD syndicate then the SC will proceed with the following messages.

STCP$_{\text{ACA}}$-7.

$$\begin{aligned} SC &: mE = ek_{SC-AD}(AD_i||SC_i||U_i||AppDoD||N'_{SC}) \\ SC \rightarrow AD &: SC_{i'}||mE||f_{mk_{SC-AD}}(mE)||SID_{AD-SC} \end{aligned}$$

When the SP is not a member of the AD, the user requires the AD to issue the *ActApp*. The SC will request the AD to issue *ActApp* by sending message seven. The SC will use a one-time pseudo card identity (SC_i) so that an eavesdropper would not be able to retrieve the SC_i. The SC will encrypt the message containing the identities of AD, SC, and user. It then appends the application details (*AppDoD*) and a new random number generated by the SC. The *AppDoD* will not have any details of the application that can help the AD to uniquely identify either the SP or the application. It will include the memory occupied by the application along with a pseudo identity, and if the AD charges the user according to the space usage then this data will be used to calculate the charge. Finally, the SC uses the one-time SID_{AD-SC} that is generated in previous protocol runs with the AD, to provide authentication credentials and possibly avoid a DoS attack on the AD's server. The SID is an abbreviation for session identifier and we have discussed it in section 4.7.5.

On receipt, the AD verifies the $SC_{i'}$ and associated SID_{AD-SC}. After verification, it will retrieve the long-term shared keys, verify the MAC, and decrypt the message. Depending upon the AD's policy, it will proceed with the charge that might include billing the user's account or credit/debit card.

STCP$_{\text{ACA}}$-8.

$$\begin{aligned} AD &: ActApp = AppDoD||AD_i||SC_i||U_i||N_{AD}||N'_{SC} \\ AD &: pd = chm||chv||pm \\ AD &: tc = Sign_{AD}(pd||ActApp) \\ AD &: SC_{i'} = h(AD_i||SC_i||N'_{SC}||N_{AD}) \\ AD &: SID'_{SC-AD} = f_{k_{AD}}(SC_{i'}||AD_i||SC_i) \\ AD &: mE = ek_{SC-AD}(tc||CertS_{AD}||SC_{i'}||SID'_{AD-SC}) \\ AD \rightarrow SC &: mE||f_{mk_{SC-AD}}(mE)||SID_{AD-SC} \end{aligned}$$

The AD will sign the message that includes the transaction certificate of the charge applied by the AD. The payment details (pd) includes the charge method (chm), charge value (chv) and payment method (pm). Finally, the AD also generate the SID (SID'_{AD-SC}) and $SC_{i'}$ to be used in the subsequent session.

After the SC receives the $ActApp$, it will activate the application and notify the cardholder about the successful outcome of the application installation, and any charge that was incurred by the AD. The charging mechanism for the individual transactions is at the sole discretion of the AD. This message also acts as proof of the transaction.

6.6 Analysis of the Proposed Protocols

In this section, we discuss the proposed protocols in terms of informal, and formal mechanical analysis using CasperFDR. Later, we detail the test implementations and experimental results.

6.6.1 Informal Analysis of the Proposed Protocols

In this section, we informally discuss the requirements for the STCPs namely STCP_{SP}, STCP_{SC} and STCP_{ACA}.

6.6.1.1 One to Twelve

In this section, we consistently refer to the protocol requirements and goals in section 6.2.3 with their respective numbers as listed in the same section. Therefore, from here onward, any reference to a goal or requirement number refers to the listed item in section 6.2.3.

During the STCP protocols, the message AU_X where $X = SC$, SP and U, authenticates communicating entities satisfying the SOG-1. To satisfy the SOG-2, all communicating entities exchange cryptographic certificates that also facilitate in entity authentication process.

The proposed STCPs satisfy requirements SOG3–5 and SOG12 by first requiring the SP to generate the Diffie-Hellman exponentials as it is computationally more powerful than the smart card. If the smart card generates the exponential before the SP then it can choose a weak key; however, as smart cards are computationally restricted devices they cannot perform such tasks. After generation of session keys, communicating entities use them to

securely communicate with each other. One exception to this is the STCP_{SC}. The SC generates the Diffie-Hellman exponential before the SP but it does not reveal the values until it receives the Diffie-Hellman exponential from the SP. In this way, they satisfy requirements SOG-3 to SOG-5 and SOG-12 like the other two STCPs; because generating the Diffie-Hellman exponential before SP is not a problem as long as it is not revealed to the SP.

All communicating parties in the STCPs use the generated session keys to securely communicate with each other, which gives an implicit mutual key confirmation, satisfying the SOG-6.

In the STCPs, session keys generated in one session have no link with the session keys generated in other sessions, even when a session is established between the same entities. This enables the protocol to provide resilience against the known-key security (SOG-7). This unlinkability of session keys is because each entity not only generates a new Diffie-Hellman exponential but also a random number, both of which are used during the STCP to generate new session keys. Therefore, even if an adversary "$\mathcal{A}$" finds out about the exponentials and random numbers of a session, it would not enable him to generate past or future session keys.

Furthermore, to provide unknown key share resilience (SOG8) the STCPs include the Diffie-Hellman exponentials and random numbers along with identities of individual entities in a message (e.g. hs and hp) that is then signed by all communicating entities. Therefore, the receiving entity can then ascertain the identity of the entity with which it has shared the key by verifying the signature and parameters used to generate the session keys (e.g. Diffie-Hellman exponentials and random numbers).

The STCPs can be considered KCI-resilient (SOG9) protocols, as the protection against the KCI is based on the digital signatures. In addition, the cryptographic certificates of each signature key include its association with a particular SP or smart card. Therefore, if $\mathcal{A}$ has the knowledge of the signature key of a smart card (or an SP) then it can masquerade the smart card to other entities but not other entities to the smart card. Another point to note is that during the STCPs, all signed messages and certificates are encrypted using the session key. This facilitates the STCPs in meeting the requirements SOG-8 and SOG-9, as an adversary cannot substitute the certificate or signature.

The STCPs also meet the perfect forward secrecy (SOG10) by making the key generation process independent of any long-term keys. The session keys are generated using fresh values of Diffie-Hellman exponentials and random numbers, regardless of the long-term keys like the smart card, user, and SP signature keys. Therefore, even if $\mathcal{A}$ finds out the signature key of any entity, this knowledge will not enable him to find out past session keys.

Communicating entities in the STCPs share signed messages with each other that include the session information, thus providing mutual non-repudiation (SOG-11).

6.6.1.2 Trust Assurance (Trustworthiness)

One of the requirements was to establish a trusted channel between a smart card and an SP. It is apparent that the required and proposed trusted channel is unidirectional in relation to the trust assurance and validation. Only smart cards provide the assurance that their current state is secure and trustworthy to SPs, not the other way around. The reason behind this is the deployment environment of the UCTD where smart cards are considered inherently untrustworthy, SPs are not. In the UCOM, a UCTD assumes that an SP can be malicious but it will result in the lease of a malicious application(s). Therefore, security and reliability analysis (e.g. bytecode verification [128, 161]) of the downloaded application and not of the SP which supplied it is adequate to protect the UCTD. Furthermore, an adequate protection mechanism implemented by the UCTD runtime environment avoids malicious runtime activities (discussed in chapter 8)

Establishing a trusted channel between a smart card and an SP is based on the security and trustworthiness of the SC (section 4.4). The trust in the established protocol session comes from the assurance that the smart card complies with the evaluated state, which is certified to be secure and trustworthy by a third party evaluation. The respective SP has implicit or explicit trust in the third party evaluation.

6.6.1.3 Denial-of-Service Protection

The aim of DoS protection is to provide a level of assurance that the proposed protocols cannot be used to mount a DoS attack against an SP. This is achieved by a) adding a session cookie to the protocol messages that serve as the session identifier, which includes the smart card's IP address, and b) by not requiring the SP to perform any public key operations unless it receives user or platform authentication.

The session cookie is generated by the SP and it is the smart card's responsibility to include the cookie in every message. On receiving a message from a smart card, the SP verifies the session cookie and if it belongs to an active session, then it can ascertain that the message came from a genuine host and not from an entity that is trying to mount a DoS attack.

The second feature requires that the smart card has to provide a signed message (with either the user- or platform-key) before the SP has to perform any heavy computations.

This is necessary to avoid the SP committing memory and computational resources, unless the communicating smart card is authenticated to the SP.

6.6.1.4 Privacy

The privacy preservation goal (SOG-15) requires that the privacy of the user is protected. This requirement does not include privacy for the SP as part of their business model is to advertise their presence and identity (i.e. web servers). Therefore, the privacy requirement is restricted to the preservation of the user's identity and her smart card's identity. The smart card's identity is protected to avoid traceability. By traceability, we mean that if a user acquires an application from a malicious SP then the malicious SP knows the identity of the smart card and the user. In the future, if the user tries to acquire an application from another SP using the same smart card, the malicious SP can trace ownership of the card back to the user. In the proposed protocol, we do not send any information that can be uniquely attached to a particular user or a smart card in plaintext. All communications that include the identities and cryptographic certificates are encrypted.

However, if a user always gets online through a permanent connection (i.e. a fixed Internet Protocol address) then a malicious user can trace the communication to a user, but only if the malicious user has previously recorded the association of the IP address with the respective user. In such a scenario, privacy preservation is difficult to maintain in the restricted framework of the secure channel protocol; therefore, the proposed STCP does not provide protection against traceability under fixed, uniquely associated IP addresses to users.

6.6.1.5 Simulator Attack Resilience

The proposed protocols provide protection in relation to the simulator attack by relying on the smart card attestation (section 4.4.3), trustworthiness, and effectiveness of the evaluation laboratory. The certification ensures that the smart card is tamper-resistant, and it is highly unlikely that a malicious user can retrieve the smart card signature key pair. Therefore, the validation proof cannot be generated by a simulated environment, and so if an SP receives a genuine validation proof then it can be certain that it is generated by a genuine smart card. Furthermore, in the online attestation mechanism the card manufacturer dynamically verifies the current state of the smart card and issues a compliance certificate that the smart card sends to the respective SP.

This will in theory give the assurance to the SP that the smart card with which it is communicating is not a simulator, and that the current state of the smart card is as it

was at the time of evaluation. This does not mean that it will always be secure or that a malicious user is not able to simulate the environment with a genuine signature key pair. It only gives the assurance that the smart card is secure against attacks as evaluated by the third party and stated in the issued certificate [56], and that it is a state-of-the-art tamper-resistant device at the time of evaluation. Therefore, if the evaluation certificate does not meet the SPs requirements or it out-dates the current attacker capability then the SP should decline the application lease. As stated earlier, granting an application lease is at the sole discretion of the SP, so if they are not satisfied with a smart card, they should not lease the application to it.

6.6.1.6 Platform & Application User Separation Attack

In this attack, as discussed in section 6.2.3 and 5.5.2, a malicious user tries to install an application that belongs to some other user on his smart card. Therefore, the identity of the card owner and the leaseholder of the application are different.

Among the proposed protocols in this chapter, only the STCP_{SP} and STCP_{ACA} provide assurances against this attack as they include the smart card owner's identity and ownership proof (i.e. U_i and signed message with a certificate) in the message. The ownership proof comes from the signature generated using the smart card owner's signature key pair. This signature and the certificate are associated with the smart card, providing a cryptographic binding between the smart card and its current owner.

In the STCP_{SC}, we intentionally omitted the inclusion of the user specific details in the protocol. The rationale behind it is that in this protocol, the respective SP does not require the user identification and the user wants to keep his or her privacy. This is necessary when a user downloads applications that normally do not require user details.

6.6.1.7 Contractual Agreement

In the STCP_{ACA}, the smart card generates and sends a contractual agreement to the SP. Therefore, the smart card commits to the SP that it has downloaded the application but this does not mean that the application is in the active state. The smart card will wait for the SP to verify the contract message sent by the smart card and to check the hash value of the downloaded application to correctly corresponds to the SP's application. Once the SP verifies these parameters, it generates the contractual agreement message to the smart card that includes the validation message (VM) of the smart card (if online attestation was requested by the SP) and/or hash of the downloaded application. The SP will proceed

with the activation of the downloaded application only after this message is received by the smart card.

The SP will only register the leased application to access the SP's services once it is activated by the smart card. On activation, the application dials back to the SP's server. On receipt of the confirmation that the application is active, the SP will sanction the application to access the provided services. The contractual agreement messages provide the assurance that a smart card and an SP have communicated with each other through the STCP_{ACA}. During this protocol, the smart card assures the SP about its security and reliability mechanisms, and they are accepted by the SP. The SP has then leased its application, which was downloaded onto the smart card without any error.

6.6.2 CasperFDR Analysis of the Proposed Protocols

The intruder's capability modelled in the Casper scripts (appendices B3, B4, and B5) for the proposed protocol is as below:

1. An intruder can masquerade as any application's identity in the network.
2. An intruder is not allowed to masquerade as an SP or TEM.
3. An intruder application has a trust relationship with the TEM.
4. An intruder can read the messages transmitted by each entity in the network.
5. An intruder cannot influence the internal processes of a communicating entity (agent) in the network.

The security specification for which the CasperFDR evaluates the network is as shown below. The listed specifications are defined in the #Specification section of appendices B3, B4, and B5:

1. The protocol run is fresh and both applications were alive.
2. The key generated by the SP and SC is not known to the intruder.
3. Entities undergo mutual authentication and key assurance at the conclusion of the protocol.
4. The long term keys of communicating entities are not compromised.
5. The user's identity is not revealed to the intruder.

The CasperFDR tool evaluated the protocols and did not find any feasible attack(s).

6.6.3 Revisiting the Requirements and Goals

In this section, we take the security goals and requirements stipulated in section 6.2.3 and provide a comparison of the proposed protocols with the selected protocols 6.2.2.

As shown in the table 6.2, the STS protocol meets the first 11 goals along with goal 15. The remaining goals are not met by the STS because of the design architecture and the deployment environment, which did not require these goals. Similarly, the AD protocol does not meet goals 6, 10 and 13–19. In the AD protocol, the user reveals her identity by sending the user certificate as plaintext along with the no mutual key confirmation.

The most promising results were from the ASPeCT and JFK protocols that meet a large set of goals. Both of these protocols can be easily modified to provide the trust assurance (requiring additional signature). However, both of these protocols are vulnerable to partial chosen key attacks, but in the table 7.3 we opt for the possibility that the JFK can be modified to overcome this problem. The reason behind this is based on the entity that takes the initiator's role. Therefore, if in the JFK we opt for the assumption that an SP will always take the initiator's role then this goal is met by the JKF.

The T2LS protocol meets the trust assurance goal by default. However, because it is based on the TLS protocol, which does not meet most of the requirements of the STCP, the T2LS also does not meet them. A note in favour of the SCP81, MM, and SM protocols is that they were designed with the assumption that an application provider has a prior trusted relationship with the smart card issuer; thus, they implicitly trust the respective smart card. This assumption, which is fundamentally incompatible with the UCOM, is why these protocols fail to support a large number of the listed goals. Most of these protocols to some extent have an architecture similar to the one with which a server generates the key and then communicates that key to the client (i.e. read smart card). They do not provide non-repudiation because they do not use signatures in the protocol run. Nevertheless, the proposed STCP_{ACA} protocol meets all the listed goals. Table 6.2 provides a comparison between the listed protocols in section 6.2.2 with the proposed protocols under the required goals (see section 6.2.3).

In table 6.2, we show that STCP_{SC} does not meet the goal 17 beside the fact that SPs in STCP_{SC} does not require user identification. Therefore, a malicious user can install an application that does not belong to him. On the other hand, if the SP does require the user's identification during the application personalisation (after the application is installed) then the personalisation process [10] should take into account the platform &

Table 6.2: Protocol comparison based on the stated goals (see section 6.2.3)

SOG	Protocols										
	STS	AD	ASPeCT	JFK	T2LS	SCP81	MM	SM	STCP_{SP}	STCP_{SC}	STCP_{AGA}
1. Mutual Entity Authentication	$*$	$*$	$*$	$*$	$*$	$*$	$-*$	$-*$	$*$	$*$	$*$
2. Exchange Certificates	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$-*$	$*$	$*$	$*$
3. Mutual Key Agreement	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$-*$	$*$	$*$	$*$
4. Joint Key Control	$*$	$*$	$*$	$*$	$*$	$*$			$*$	$*$	$*$
5. Key Freshness	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$-*$	$*$	$*$	$*$
6. Mutual Key Confirmation	$*$		$*$	$*$			$*$	$-*$	$*$	$*$	$*$
7. Known-Key Security	$*$	$*$	$*$	$*$	$*$	$*$	$*$		$*$	$*$	$*$
8. Unknown Key Share Resilience	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$-*$	$*$	$*$	$*$
9. KCI Resilience	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
10. Perfect Forward Secrecy	$*$		$*$	$*$	$*$	$*$			$*$	$*$	$*$
11. Mutual Non-Repudiation	$*$	$(*)$	$+*$	$*$	$*$	$*$	$+*$	$+*$	$*$	$*$	$*$
12. PCK Attack Resilience	$(*)$	$(*)$		$(*)$	$(*)$	$(*)$			$*$	$*$	$*$
13. Trust Assurance					$*$	$-*$			$*$	$*$	$*$
14. DoS Prevention				$*$					$*$	$*$	$*$
15. Privacy	$(*)$		$*$	$*$					$*$	$*$	$*$
16. Simulator Attack Resilience					$-*$				$*$	$*$	$*$
17. PAU Attack Resilience									$*$		$*$
18. Contractual Agreement									$+*$	$+*$	$*$
19. Proof of Transaction			$*$		$+*$	$+*$	$+*$	$+*$	$+*$	$+*$	$**$

Note: $*$ means that the protocol meets the stated goal, $**$ indicates that the protocol meets the SOG if required by the communicating entities, $(*)$ shows that the protocol can be modified to satisfy the requirement, $+*$ shows that protocol can meet the stated goal but requires an additional pass or extra signature generation, and $-*$ means that the protocol (implicitly) meets the requirement not because of the protocol messages but because of the prior relationship between the communicating entities.

application user separation attack.

As is apparent from the table 6.2, the proposed STCPs satisfies all goals that were described in section 6.2.3. The protocols that are proposed specifically for the smart card environment (i.e. ICOM) only meet half of the stated goals because the security requirements for the UCOM are more stringent than for the ICOM [32]. Nevertheless, we still consider that the proposed STCP should be deployed even in the ICOM and especially with any future ownership model that supports multi-applications on a smart card under the Trusted Service Manager (TSM) architecture.

6.6.4 Implementation Results and Performance Measurements

For comparison, we have selected the performance of SSL [182], TLS [183], and public key-based Kerberos [184] implemented on 32-bit smart cards. We selected the SSL and TLS because they form the bases of the GlobalPlatform SCP81. Kerberos closely relates to the card management architecture of Multos (section 5.3). The Multos Certification Authority acts as a Trusted Third Party (TTP), and the public key-based Kerberos can be implemented to accommodate the Multos card management framework. The Kerberos discussed in the performance measures is also implemented on 32bit smart cards [184].

Table 6.3: Protocol performance measurement (milliseconds)

Measures	**SSL**	**TLS**	**Kerberos**	**STCP$_{SC}$**		**STCP$_{SP}$**		**STCP$_{ACA}$**	
				C1	C2	C1	C2	C1	C2
Average	4200	4300	4240	2998	3091	3395	3532	5843	6098
Best Run	NA	NA	NA	2906	3031	3343	3359	5485	5688
Worse Run	NA	NA	NA	3922	4344	3875	6797	9734	7329
Std Deviation	NA	NA	NA	117.54	96.28	69.82	134.91	191.62	171.13

Note: the above mentioned measurement values for SSL are taken from [182], TLS [183] and Kerberos [184]. C1 and C2 are 16bit Java Cards. We have rounded up the values to the nearest natural number except for the standard deviation.

t

For performance measurements, we use the same test bed configuration described in section 4.8.3. For the STCP$_{SC}$ and STCP$_{SP}$ we implement two entities: a smart card and an SP. For the STCP$_{ACA}$ we implement an additional entity of administrative authority. Both an SP and an administrative authority are implemented on a laptop with 1.83 GHz, and 2GB RAM running on Windows XP. The Java Card implementation of the STCP$_{SP}$, STCP$_{SC}$, and STCP$_{ACA}$ took 11102, 10382, and 13364 bytes, respectively. The performance measures listed in the table 6.3 do not include the attestation process, which is listed in table 4.3.

For the STCP_{ACA}, we need to provide the digest of the downloaded application. To do this we emulated the performance measure by monitoring the time it took to generate hash on a 256 bytes array. The hash generation on the 256 bytes took 31 milliseconds on the test smart cards. The performance measures of hash generation on the test smart cards with different sizes of the download application are shown in figure 6.2

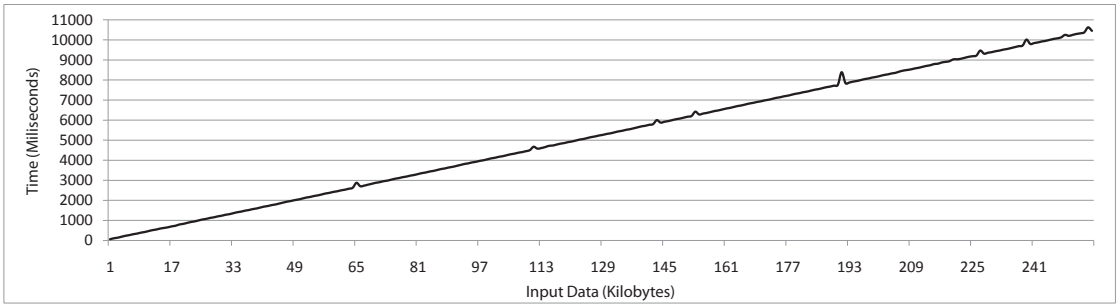

Figure 6.2: Performance measurements of hash generation on test smart cards

The STCP_{ACA} can be divided in to three distinct phases that are listed in the table 6.4 along with the breakdown of the performance measure. The breakdown provides a rough guide how much extra time these protocols will take if the protocols STCP_{SC} and STCP_{SP} are extended to provide SOG-18 and SOG-19.

Table 6.4: Breakdown of performance measurement (milliseconds) of the STCP_{ACA}

Phases	**Measures**	**STCP_{ACA}**	
		C1	C2
AKA Phase (STCP_{ACA}-1 – 4)	Average	3182	3334
Contract Phase (STCP_{ACA}-5 – 6)	Average	1253	1294
Charge Phase (STCP_{ACA}-7 – 8)	Average	1407	1470
Total (STCP_{ACA}-1 —8)	Average	5843	6098

The performance measures are only for the reference of our implementation, as the actual performance will vary depending the attestation process, the application size, and the communication speed (i.e. Internet bandwidth).

6.7 Summary

In this chapter, we discussed Secure Channel Protocols (SCPs) and their role in the UCTD. In addition, we provided the rationale behind proposing new SCPs. This was followed by an account of the related work in the field. We discussed security and operational goals for the proposed protocols. We then proposed three protocols that satisfy varying levels of security and operational goals, along with the user's and SP's requirements. These protocols were then analysed informally for a limited set of security goals and compared with a set of selected protocols. We subjected the proposed protocols to mechanical formal

analysis using the CasperFDR, which they passed without any evidence of feasible attack. Finally, we discussed the test implementation and their performance measures, comparing them with a set of selected protocols.

Chapter 7

Application Sharing Mechanisms

Contents

In this chapter, we describe two contrasting frameworks for application sharing, namely those deployed by Java Card and Multos; followed by an explanation of our reasoning for deciding that we need to extend the existing techniques for UCTDs. We then discuss the rationale behind our proposal for the application sharing mechanism in the UCTD environment. This sharing mechanism requires entity authentication, trust validation, and key generation to securely share resources between applications. To do so, we propose protocols that achieve the listed goals of the UCTD application sharing mechanism. Furthermore, we provide an informal analysis of the protocol along with a comparison with existing protocols. Subsequently, we present a mechanical formal analysis the based on the CasperFDR, and we report on our experience from developing and experimenting with a prototype implementation.

7.1 Introduction

Multi-application smart cards enable the co-existence of interrelated and cooperative applications that augment each other's functionality. This enables applications to share their data as well as their functionality with other applications, achieving optimised memory usage, and data and service sharing between applications [14].

A major concern arising from application sharing mechanisms is the possibility of unauthorised inter-application communication. A framework that ensures that application sharing is secure and reliable even in adverse conditions (i.e. malicious applications, developer's mistakes, or design oversight, etc.) is referred as a smart card firewall [185]. In this chapter, the terms firewall and smart card firewall are used interchangeably.

The dynamic and decentralised nature of the UCOM may lead to unauthorised application communication and the associated privacy concerns. Existing techniques deployed by the smart card industry are not adequate to provide security and reliability to the application sharing mechanism on a user centric device. The issues involved are: a) an inability to dynamically authenticate an application on a smart card, b) difficulty in ascertaining the security and reliability of the current state of an application, c) an inability to verify and restrict application sharing (privilege-based access), d) no provision for privacy preservation for cardholders, and e) no cryptographic binding between applications. Therefore, in this chapter, we discuss the proposed firewall mechanism [186] that provides an extension to the traditional mechanisms deployed in Multos and Java Card, in order to deal with the listed issues.

There may also be a requirement to allow applications executing on different UCTDs to intercommunicate. Thereby, we further extend the architecture of the proposed firewall [186] to accommodate application sharing among applications that are installed on different UCTDs. This extension is referred as the Cross-Device Application Sharing Mechanism (CDAM).

To meet the requirements for a UCOM firewall mechanism, we propose three protocols, analyse them against a predefined set of stated goals, validate them using mechanical formal analysis using CasperFDR, and finally describe a prototype implementation and performance measurements.

Structure of the Chapter: Section 7.2 discusses the application sharing mechanisms deployed by Java Card and Multos, along with the rationale behind the proposal for a UCOM firewall mechanism. In section 7.3, we describe the architecture of the proposed firewall mechanism. To provide entity authentication, application state assurance, and secure application binding we propose an Application Binding Protocol (ABP) in section

7.4. We then propose two protocols for the CDAM framework in section 7.5 and 7.6. In section 7.7, the proposed protocols are analysed for their security and performance.

7.2 Application Sharing Mechanism

In this section, we describe the application sharing mechanism implemented in Java Card and Multos. The reason for choosing Java Card and Multos is twofold: a) they represent two contrasting architectures to implement the firewall mechanism, and b) they are the two most deployed smart card platforms. Furthermore, the firewall mechanisms deployed in the ICOM are mature [28, 29, 185, 187, 188] and have been extensively studied [189]–[192], which cannot be claimed for the UCOM.

7.2.1 Firewall Mechanism in Java Card

The generic architecture of a Java Card is shown in figure 7.1. The Java Card Runtime Environment (JCRE) sits on top of the smart card hardware and manages the on-card resources, applet execution, and applet security [28]. The JCRE has APIs (e.g. `APDU`, `Util` and `Shareable`) that an application can use to access JCRE services. The JCRE also has system classes that are integral to its functions and these classes are not visible to applets. The firewall mechanism separates individual applications from each other and from the JCRE. In Java Card, an application is a collection of applets grouped together as a package — for example, packages A and B in figure 7.1;

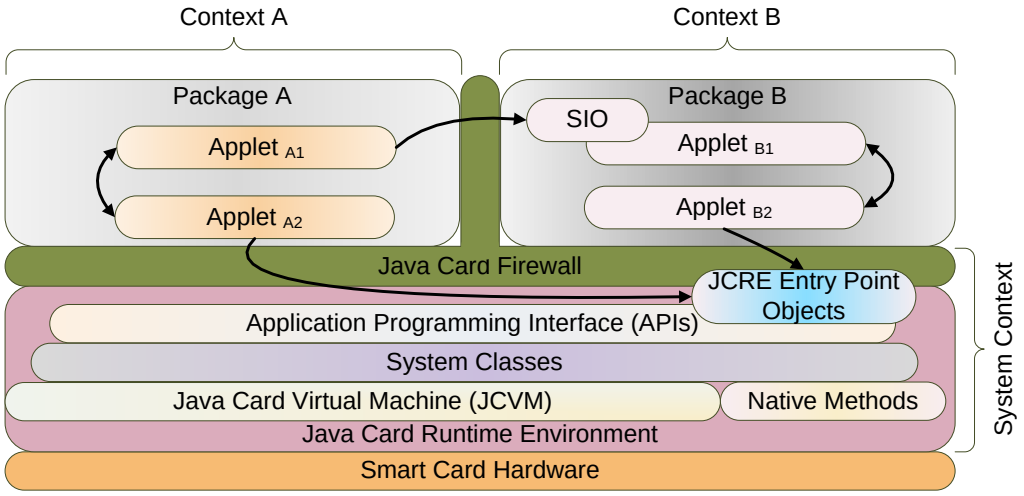

Figure 7.1: The Java Card firewall mechanism

Each instance of an applet has a unique Application Identifier (AID) [28]. An instantiated representation of an applet is termed an object. Each object is associated with a context,

including the JCRE objects (System Context). The Java Card Virtual Machine (JCVM) only allows an object to execute if the current "Active" context is the one to which it belongs. In figure 7.1 an object of $\text{Applet}_{\text{B1}}$ will only execute if the "Active" context is context B. The firewall restricts all cross context communication except for object sharing mechanisms that include JCRE Entry Point Objects and Shareable Interface Objects (SIOs). All applets in a package have the same context so there is no firewall between them.

The JCRE Entry Point Objects are instances of the Java Card APIs that can be used by applications to access platform services. These objects are accessible to all applets, and they enable non-privileged (applets) applications to execute privileged commands. The JCRE Entry Point Objects are implemented by the Java Card manufacturer, which is responsible for their security and reliability.

An SIO enables an application to share its resources with other authorised application(s). To utilise an SIO functionality, an application should implement the shareable interface (`javacard.framework.Shareable`) — the implemented functionality as part of the class that implements the shareable interface will be shareable with other applets.

When an object requests either an SIO or JCRE Entry Point Object, the JCVM saves the current "Active" context and invokes the requested object along with the associated context. Therefore, a shareable object always executes in its own context, enabling it to access any applet from the package it belongs to. By taking into account figure 7.1 when $\text{Applet}_{\text{A1}}$ calls an SIO of $\text{Applet}_{\text{B1}}$, the JCVM saves context A and invokes context B and also initiates the execution of an SIO. An SIO can then call any method in package B. Furthermore, it can also call any JCRE Entry Point Objects. When an SIO completes its execution, the JCVM restores the previous context (context A).

7.2.2 Firewall Mechanism in Multos

Multos [29] takes a different approach to Java Card in implementing a smart card firewall. The Multos Card Operating System (COS) resides over the smart card hardware as illustrated in figure 3.2. The Multos COS administers communication, resource management, and the virtual machine [29]. Applications do not have direct access to the Multos COS services; instead they utilise the Application Abstract Machine that is a set of standard APIs consisting of instructions and built-in functions. These APIs are used by applications to communicate with the COS and request privileged services. The top layer is the application space, and similar to Java Card the application segregation is implemented by the Multos firewall.

In Multos, application delegation is implemented to enable application resource sharing. The application that initiates the process is called the delegator and the application that is initiated is called the delegate. The process of delegation works as described below and shown in figure 7.2:

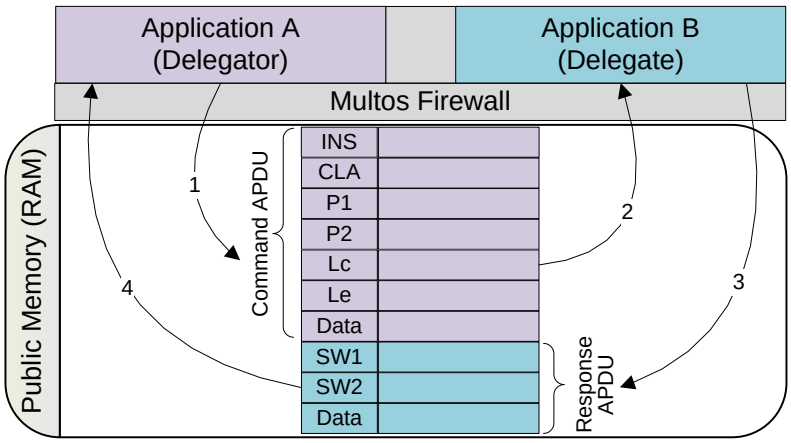

Figure 7.2: The Multos card firewall mechanism

1. Application A (delegator) creates an APDU in the public memory and invokes the delegate command. The APDU consists of application B's AID, requested data or function and the delegator's AID.

2. The Multos COS initiates the execution of B that looks for the APDU in the public memory. It reads the APDU and processes it.

3. On completion, B creates a response APDU within the public memory.

4. The Multos COS switches back to A that then retrieves B's APDU.

7.2.3 Rationale for User Centric Smart Card Firewall

Traditional smart card firewall mechanisms are fit-for-purpose in the ICOM environment but they do not provide adequate security to the UCTD environment. The operational and security requirements a UCOM firewall has to satisfy are:

FiR-1 No Off-card Security Assumption: The firewall mechanisms discussed in previous sections are designed with the implicit assumption that the smart card will be under the card issuer's control. The security of the platform is ensured not only by the on-card mechanisms but most importantly by the off-card agreements, which prevent installation of a malicious application and prevent unauthorised application access.

FiR-2 Application Authentication: In both Java Card and Multos, applications rely on the AID to locate, request and access shareable resources/data. The AID is a 5–16 byte identifier that consists of two components: a Registered Identifier (RID) and a Proprietary Application Identifier Extension (PIX). The RID is 5 bytes long and compulsory; on the other hand the PIX can be 0–9 bytes long and it is optional. If you are developing an application that will be used either nationally or internationally, you need to get a RID from designated authorities (i.e. national standardisation authorities). AIDs are issued by a designated authority but there is no enforcement mechanism that prevents an adversary from masquerading as an application on a smart card. In the ICOM, this situation does not arise because application installation is only authorised by the card issuer and even this is to a restricted group of trusted application providers. However, in the UCTD environment such a measure is difficult.

FiR-3 Application State Validation: An application "App_A" might be modified either intentionally or accidentally. This might have a malign affect on applications that share resources with, or use the resources of App_A. Therefore, before establishing sharing it would be beneficial to ascertain the current state of App_A. In addition, the firewall should also notify the server application if the client application is modified (i.e. if there have been application updates), so if the client application wants, it can revoke the sharing, and vice versa.

FiR-4 Access Control: The firewall should facilitate a flexible mechanism that enables a server application to implement a hierarchical access-level firewall. In such a firewall, a server application assigns shareable resources according to different access levels. In addition, it can also revoke, upgrade, or demote the existing privileges of a client application.

FiR-5 Application Binding: Two applications that share each other's resources should be able to bind the sharing instance (cryptographic binding) in order to provide authentication, confidentiality and reliability to all future communication.

FiR-6 Application-Platform Communication. This requirement deals with bi-directional communication between an application and a smart card platform and it is subdivided into two sections as listed below.

(a) Application to Platform Communication: Platforms make their services available to applications either through Entry Point Objects [28] or standard APIs [29]. In both cases, applications may have access to more platform services than required. That would not be desirable in the UCTD [10]. In the UCTD, applications are only given access to those platform services that are authorised by their SPs. The firewall ensures that an application cannot have access to any other services from the platform for which it is not authorised. This

allows the SPs to control their applications' behaviours, especially in terms of on-card and off-card communication.

(b) Platform to Application Communication: Java Card (like other multi-application smart cards) provides global access rights to the platform. The global access rights mean that an object of the JCRE System Context can access any method (object) in any of the application contexts. However, the Java Card specification explicitly notes that the platform should only access certain methods (`select`, `process`, `deselect`, or `getShareableInterfaceObject`) from an applet context [28]. In the UCOM, the firewall should ensure that a platform cannot have access to methods that are not sanctioned by the application SPs. Furthermore, it should enable an object or method to verify the requesting source. For example if the source is the platform, and it is trying to access an object or method not sanctioned by the corresponding SP, then it should throw a security exception.

FiR-7 Sharing Revocation: A server (or client) application can revoke a privilege, even after the server and the client have established a sharing relationship with each other. In Multos and Java Card, the only way to revoke privileges is to modify the server and/or client-application code. If a server application does not want to share resource with the client application, then the server application has to implement adequate checks to throw an error or exception when the client application accesses the resources. From the client application's point of view, the SP has to modify the client application so that it cannot use the shareable resources.

FiR-8 User's Privacy: The firewall mechanism should not allow an application to discover the existence of other applications, because such a privilege could be used to profile a user, perhaps for marketing or fraudulent purposes. In Java Card, `public static AID lookupAID` can be used to list the installed applications. It is not an issue in the ICOM as there is a central authority (card issuer) that has prior knowledge of installed applications and (to some extent)their functionality. However, it is a potential privacy threat in the UCTD environment.

The comparison between Java Card, Multos and the proposed firewall mechanism is illustrated in table 7.1.

7.2.3.1 Why Cross-Device Application Sharing?

With increasing interconnectivity between different computing environments, applications installed on different UCTDs can enable new service models by having a secure and reliable resource sharing mechanism. These are referred to as Cross-Device Application Sharing Mechanism (CDAM). Some of the possible applications of CDAM are listed below:

Table 7.1: Comparison between different firewall mechanisms

FiR	Multos	Java Card	UCTD
1. No Off-card Security Assumption	No	No	Yes
2. Application Authentication	Yes*	Yes*	Yes
3. Application State Validation	No	No	Yes
4. Access Control	No	No	Yes
5. Application Binding	No	No	Yes
6. Application-Platform Communication	No	No	Yes
7. Sharing Revocation	Yes*	Yes*	Yes
8. User's Privacy	No	No	Yes

Note. "Yes" means that it totally supports the given requirement, "Yes*" stands for limited support, and "No" means that it does not support the given requirement.

1. A mobile handset may have multiple UCTDs, which under the CDAM architecture behave as a single virtual device. Removing the need for a user to install applications that share each other's resources on the same UCTD makes the management of the multiple UCTDs flexible and user-friendly.

2. The CDAM can facilitate the installation of an internet identity application [193] on a UCTD that can be accessible to other UCTDs and the host platform. For example, if a user installs an internet identity application (i.e. which may act as a single sign on) on a UCTD then it may be used to authenticate the user when visiting online services (e.g. online gaming, social and network sites, etc.) or by applications (e.g. network access, online banking, and online ticketing, etc.) on (other) UCTDs.

3. An accounting application on a UCTD may opt for automated receipt collections and updates to the user's accounting software. For example, a user might have a financial system on her Personal Computer (PC) that she uses to track her expenditure. To enhance the mobile payment scheme, the mobile payment SP may collaborate with an accounting software developer in a way that means the payment application might record the transaction details that are later synchronised with the accounting software. The user would have the accounting software installed on her PC, with an associated application installed on the UCTD. Afterwards, the user synchronises the transaction details to her financial software. The synchronisation would be carried out by means of the UCTD of the mobile phone and the UCTD of the PC. Thereby, the CDAM provides security, reliability, and privacy to this system.

4. Internet of Things [37, 160] is an internet-like structure comprising a set of smart physical devices (e.g. toys, healthcare products, thermostats, and environment sensors, etc.) that communicate with each other. As an individual device may not have enough computational and storage resources, they would not have a complex or large set of services. However, the CDAM can enable comparatively complex and rich featured systems in an Internet of Things. Individual devices may either have a unique service in the set or even a subset of a particular service. Each connected device then

utilises CDAM to create a single virtual device comprising heterogeneous devices and a UCTD-based architecture will enable the efficient replacement of a service if the host device goes out of the network, by requesting installation of the service on an alternative available device.

7.3 UCTD Firewall

In this section, we discuss the architecture of the proposed firewall mechanism for UCTDs.

7.3.1 Firewall Architecture

The proposed firewall mechanism is based on the Java Card firewall mechanism as illustrated in figure 7.3 that is discussed subsequently.

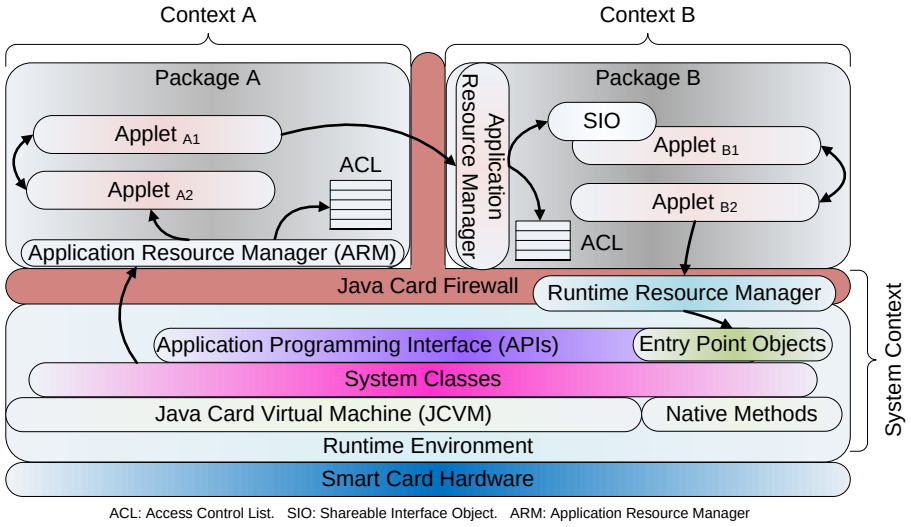

Figure 7.3: Architecture of the UCTD firewall mechanism

The request for an application's shareable resource is handled by the application's Application Resource Manager (ARM) and the Runtime Resource Manager (RRM) handles the access to the platform's resources (APIs): see figure 7.3.

The RRM controls the access to the entry point objects that are used to access platform services. The resource manager will enforce the security policy for applications as defined by the respective SPs, limiting access to the platform resources as stipulated by the policy.

For each application (package), an Application Resource Manager (ARM) is introduced. This component will act as the authentication and resource allocation point. A client

application will request a server application's ARM to enable the sharing of resources. The ARM will decide whether to grant the request based upon the client's credentials (associated privileges). At the time of application installation, the ARM also establishes a shareable interface connection with the platform, enabling the application to access methods that are essential for the application execution. The platform can access any method in the application context only after authorisation from the application's SP. The ARM also receives information regarding the requesting application. If the request is from the system context for a method that is not allowed to be accessed by the platform, then the ARM will throw a security exception.

An Access Control List (ACL) is a private list and it is used to facilitate the implementation of a hierarchical access mechanism and privilege revocation. An ACL can be updated remotely by its corresponding SP (when the application connects with the SP's servers, the SP can update the ACL), changing the behaviour of its application's sharing mechanism. The ACL holds lists of granted permissions, received permissions (permissions to access other application's resources) and a cryptographic certificate revocation list of client applications. The structure of an ACL is under the sole discretion of its SP and it is stored as part of the ARM.

The operations of the firewall can be sub-divided into two distinctive phases. In phase one, a binding is established between the client and the server applications. This process includes authentication of the client's credentials and access privileges by the server's ARM. In the second phase, the client application requests resources in line with the privileges sanctioned by the ARM. In both these phases, the firewall mechanism facilitates individual authorised applications to accomplish the application sharing, while prohibiting unauthorised applications from accessing the resources of an application.

7.3.2 Application Binding

In the UCTD-based firewall mechanism, a client application (App_C) establishes a secure connection with the relevant server application (App_S) to authenticate and verify the current state of the App_S, and to establish a secure binding with the App_S for future communications. Similarly, App_S can also authenticate and verify the current state of the App_C. Therefore, an application binding indicates that a client and server application have authenticated each other and trust each other's state to be secure (and trustworthy).

When a client application requests shareable resources, the firewall invokes the ARM of the server application. The ARM then verifies and validates the client application's credentials, and current state as secure for sanctioning the application sharing (as part of the ABP). If the request is successful, the ARM issues the shareable resources to the requesting

application. There are two ways the current state of individual applications might be verified: both of the SPs can opt for a third party evaluation of their respective applications, or they can issue a certificate to each other's application that contains the hash of the application. In both of these cases, the certified state of the application is treated as a trusted state by the other entity.

The state validation of individual applications is carried out by the Trusted Environment & Execution Manager (TEM), requesting the application and issued certificates. Consider the scenario of application sharing between App_C and App_S. When App_C is installed onto a smart card, the relevant TEM establishes a secure relationship (shared key: K_{App_C-TEM}) with the installed application. The TEM does not calculate the application state validation message (i.e. hash of the application), unless it is authorised to do so by the application itself, or by the application's SP. When an application authorises the TEM to generate its hash value, it generates a message encrypted with the shared symmetric key. The authorisation message generated by an application is referred to as an Integrity Measurement Authorisation (IMA) message. The IMA sent by the client application will be $E_{App_C-TEM}(App_C||App_S||RandomNumber_{App_S})$. The contents of the messages are the identity of the client and server application, along with a random number generated by the server application. The state validation message would be $E_{App_S-TEM}(App_C||App_S||RandomNumber_{App_S}||hash(App_C))$ that is encrypted with the TEM and the server application's shared key as this message is intended for the server application. The server application can match the hash calculated by the TEM with the one in the client application's certificate that is issued either by a third party evaluator or by the server application's SP. If they match, the server application can securely ascertain that the state of the client application is secure. A similar process can be performed in the opposite direction where a client application verifies the state of the server application.

After applications authenticate and validate their states to each other, they generate a cryptographic key that is referred to as the application binding key. This key is used in all future communications between the applications.

7.3.3 Using Shareable Resources

A client application can request to use the server application's shareable resources if required (subject to valid access permissions) as illustrated by figure 7.4.

The request message sent to the corresponding ARM consists of a ClientAID, an authenticator (message encrypted with the application binding key), access permission, the required resource and a random number to provide freshness [132]. By verifying the authenticator, the ARM ascertains the origin of the message — that is, the client application.

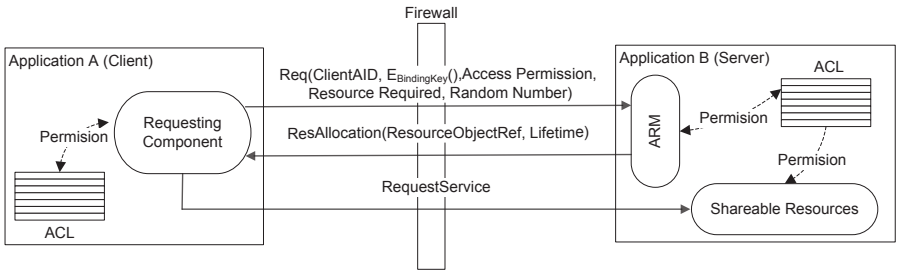

Figure 7.4: Application shareable resource access request process

Subsequently it checks the access permission for the client application (from the server application's ACL). If the client application is authorised to access the requested resource, the ARM will return the resource's object reference along with the sharing lifetime.

There are two lifetime modes, permanent grant of access to an object or temporary. In permanent mode, the server application grants the ownership of the object to the client application as proposed by the Java Card 3.1 connected edition firewall [16]. In temporary mode access is limited to individual sessions and ownership of the object is retained by the server application.

7.3.4 Privilege Modification

The SP of a server application can modify the privileges of a client application by updating the ACLs. The ARM of the server application verifies the initiator's (SP's) identity and credentials, before allowing the update of the ACL(s). The implementation of the privilege modification is at the sole discretion of the SP. Such an update could be similar to application update mechanisms already deployed, notably Over-The-Air updates in (U)SIM application [6].

7.3.5 Application-Platform Communication

At the time of installation, an application establishes bidirectional resource sharing with the platform. The application can access those platform APIs that are stipulated in the SP's Application Lease Policy (ALP) discussed in section 3.4.6, and the platform obtains the shared resources of the applications that are necessary to initiate the application execution. The platform security context does not have global access in the UCTDs. This is to avoid any possible exploitation of the platform that could lead to information leakage (data or code) from an application. The resource-sharing delegation is disabled in the platform-application communication and the firewall will deny such requests to avoid any illegal access to the APIs by an application through resource sharing delegation.

7.3.6 Cross-Device Application Sharing

In the Cross-Device Application Sharing (CDAS) architecture, a smart card acts like a node that is registered with a centralised system. The centralised system in our proposal is software running on a computer, mobile phone, or tablet, which is referred as Card Application Management Software (CAMS) [32]. For a simplistic illustration, figure 7.5 shows two possibilities for the CDAS network.

In figure 7.5a, a mobile phone has three UCTDs and all of them are connected to a CAMS hosted on the mobile phone. The CAMS can be hosted on an insecure platform and it provides discoverability and interconnectivity to an individual UCTD connected to the CAMS. By discoverability, we mean that a platform registers itself with the CAMS and thus it becomes discoverable to all other platforms in the network. The interconnectivity deals with the communication channel established between two (or more) UCTDs. Therefore, figure 7.5a depicts a scenario in which multiple UCTDs are connected to a mobile phone, and their interconnectivity and discoverability is handled by the CAMS installed on it.

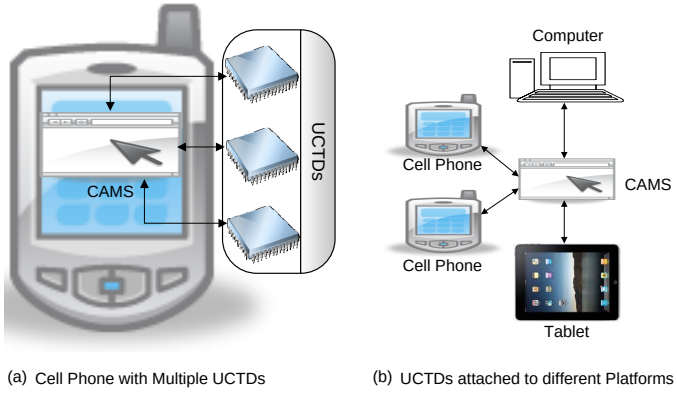

Figure 7.5: Cross-Device Application Sharing network

On the other hand, figure 7.5b shows a situation in which different computing devices (e.g. computers, mobile phones, and tablets) are connected with each other through their CAMS. Each individual device may have multiple UCTDs that are registered to their respective CAMS. Although figure 7.5b depicts the situation as if there is a single centralised CAMS, this is incorrect, as each host device has its own CAMS and there are no centralised CAMS. Therefore, if a particular device is not available, other devices can still communicate with each other. In this scenario, a host device will discover and register other computing devices. There are two possible situations. In the first, each individual host device advertises the connected smart cards to the entire network. In second case, each host device only provides the details of its CAMS and not the smart cards registered with it. For our proposed CDAM, we prefer the second arrangement as it provides better privacy for individual smart cards.

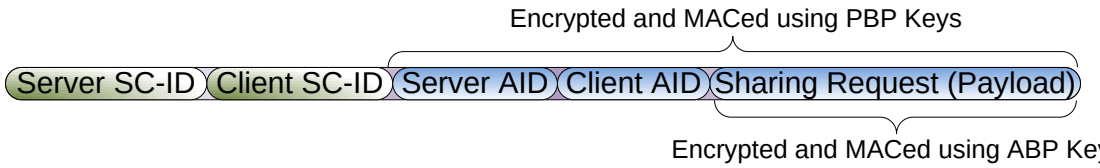

Figure 7.6: Cross-Device Application Sharing message

The provision of whether an application supports cross-device application sharing is at the sole discretion of the respective SP. The UCTD architecture will provide two levels of application sharing: a) localised sharing, and b) cross-device sharing. The first option restricts the application sharing to the smart card on which the application is installed. Any client application that is not installed on it will not be able to access the shareable resources of the server application. This scenario is implemented in traditional smart card firewalls and supported by the UCTD. The second option allows application sharing with a client application, whether or not it is installed on the same platform.

In a succinct manner, we can divide the CDAM process into four steps listed as below:

1. Registration: The first step involves registration of individual applications with their respective smart cards, and individual smart cards with their respective CAMS.
2. Platform Binding: A new smart card that is recently registered with the CAMS will ask the CAMS to provide a list of available (registered) smart cards. The CAMS provides the pseudo identities of other associated smart cards. Each smart card then initiates a dialogue referred as "platform binding" with other smart cards in the network and establishes a secure relationship (e.g. by means of a shared cryptographic key) that is termed as platform binding key. The platform binding is similar to the application binding discussed in section 7.3.2 but is between smart cards, rather than applications. After the acknowledgement of the platform binding, each smart card will register the other in its list of bound platforms.
3. Application Binding in CDAM: In the third step, client and server applications proceed with the application binding process that on its successful conclusion binds them in a way that enables them to share resources in a secure and reliable manner.
4. Application Sharing in CDAM: The client application requests its host smart card regarding the status of the server application's host smart card. If the server application is in the network then it will proceed with the application sharing. The application binding key is used to generate the session key to communicate with the server application. The structure of the message is illustrated in figure 7.6.

The resource access can be based on two mechanisms: Java Card [28] or Multos [29] inter-application communication architecture. These two mechanisms represent the synchronous

and asynchronous application sharing respectively. Synchronous application sharing enables two applications to communicate in real-time, and they can be considered as real-time distributed systems. In case of the asynchronous application sharing the communication between applications can occur when both are available (live in the network). Both schemes have benefits and drawbacks; therefore, it is at the sole discretion of the individual set of client-server applications to decide which mode they will employ. For further explanation and to provide a contrast between these two mechanisms consider the following examples.

Consider two applications, one of them provides access to certain internet services, and the second application is an internet identity [193] that acts like a Single-Sign-On (SSO). Every time a user logs on to the Internet services through the first application, a connection has to be established with the second application to provide the internet identity for user authentication and authorisation. This access to the SSO application has to take place when the first application connects with the internet services. Such an access is termed as synchronous access in the UCTD and is based on the Java Card architecture [186]. Such an access can be based on the Java Card Remote Method Invocation (RMI) [28].

For asynchronous access, we also take an example of two applications. One application provides electronic wallet functionality, and the second is a loyalty application. Every time the electronic wallet application is used, the cardholder earns loyalty points. The loyalty application does not have to be live. The electronic wallet application batches the loyalty point update task and when the loyalty application becomes live, it can proceed with updating it. For this purpose, a simple mechanism deployed by the Multos application sharing would suffice. The electronic wallet application batches the APDUs to update the loyalty application. When the loyalty application comes online, all batched APDUs can be communicated to it.

In the next section, we describe the goals and requirements of the proposed protocols to establish application and platform binding, before we look into the details of Application Binding Protocol — Local (ABPL) that focuses on the application binding between two applications on the same UCTD. Later, we discuss the Platform Binding Protocol (PBP), and Application Binding Protocol — Distributed (ABPD) that are proposed to support the CDAM.

7.3.7 Minimum Goals and Requirements for the Proposed Protocols

The goals and requirements for the proposed protocols that facilitate the establishment of application and platform binding are listed below. This list is an extension to the list in section 6.2.3, with the exception of requirements SOG-17 to SOG-19. Later, we will revisit these goals for the protocol comparison in section 7.7.2.

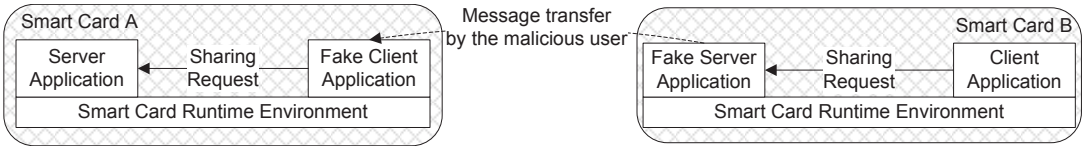

Figure 7.7: Application masquerading and relay attack scenario

SOG-20 Application Masquerading. In this scenario, a malicious application can masquerade as a server or client application. For example, in Java Card when a client application sends a request for application sharing it generates the request that contains the server application's AID. Now if a malicious application is masquerading as a server application, it only has to inform the firewall that it accepts the application sharing request without validating that it has the knowledge of the shared secret. Thus the client application thinks that it is accessing the shared resource of the server application, whereas in fact it is communicating with a malicious application. Now the fake server application can resend the application sharing request message to a genuine server application on another smart card and gain access to shared resources; this scenario is illustrated in figure 7.7.

SOG-21 Different User's Applications. Consider a scenario in which we have two users and two applications. One is a malicious user M_u while the other is an authorised user A_u. The two applications are App_A (server application) and App_B (client application) that have a client-server relationship. Both users are authorised to download application App_A, however M_u is not authorised to download application App_B. Now at some point, the M_u obtains the App_B's credentials for the A_u and manages to download App_B onto his or her smart card. The application sharing between the M_u's App_A and the A_u's App_B can be established. This can lead to some financial benefits for the M_u to which he or she is not entitled.

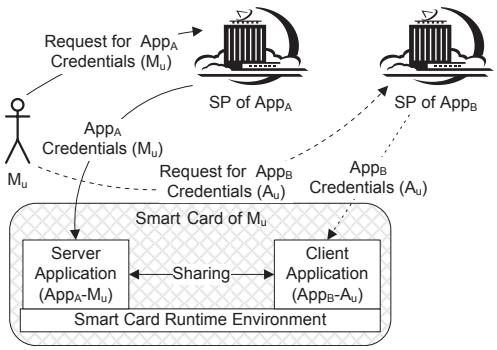

Figure 7.8: Application sharing among different user's applications

Table 7.2: Protocol notation and terminology

Notation	Description
SE	Represents the server application.
CL	Represents the client application.
TEM	Represents the TEM on a smart card.
F	Represents the UCTD firewall on a smart card.
K_{A-B}	Long term symmetric key shared between entity A and B.
K^t_{S-C}	Session key generated by the TEM.
$E_K(Z)$	Represents symmetric encryption of the data "Z" with the key "K"
$N_X + num$	Random number of entity X is incremented by the value of num, where num = 0, 1, 2, 3, ...
$X\|Y$	Represents the XOR binary operation on the data items X, Y.
IMA_X	Integrity Measurement Authorisation message generated by entity X.
VRE_X	Application assurance validation response generated by the TEM for entity X.
AP	Represents the authentication process a server application requires from the respective client application, when requesting the shareable resources.
OR	Represents the object reference to the server application's resource manager (i.e. ARM).

7.3.8 Protocol Notation and Terminology

In this section, we list the notation used to describe the protocols in this chapter. The notation listed in table 7.2 is an extension to the notation described in tables 4.2 and 6.1.

7.3.9 Enrolment Process

During the enrolment process, the SPs of a client and server application agree on the business and technical terms for sharing their application resources on a UCTD.

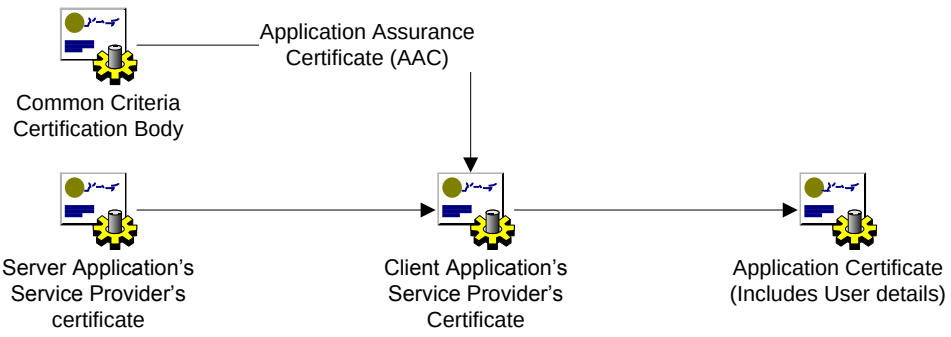

Figure 7.9: Hierarchy of a client application's certificate

In this process, an SP of a client application provides assurance and validation from a

third party evaluation [69] to an SP of a server application, and vice versa. If third party evaluation is not available then both client and server application's SPs can decide on any other adequate way of establishing trust in each other's application and its functionality. During this process, they decide the details of the ABP, such as how to perform an on-card verification and validation of applications. One possible way could be that the SP of a server application issues a certificate to a client application, and vice versa.

The certificate hierarchy in the ABP is illustrated in figure 7.9. In the absence of CC evaluation, the certificate hierarchy shown in figure 7.9 will not include "Common Criteria Certification Authority". The client application certificate has the hash value of the application. Similar contents will also be included in the server application's certificate that is issued by the SP of the client application. Basically, the enrolment process defines the restrictions and mechanisms (i.e. certificates, and cryptographic algorithms, etc.) that a client/server application's SPs agree on for the ABP.

7.4 Application Binding Protocol — Local

In this section, we begin the discussion by explaining the protocol prerequisite followed by the protocol description.

7.4.1 Protocol Prerequisites

The prerequisites for the ABPL are listed below, and are an extension to the prerequisites listed in sections 4.7.1, 6.3.1 and 6.5.2 with exception of prerequisites PPR-8 to PPR-13.

PPR-14 Off-Card Relationship: The SPs of individual applications trust each other. The roles of the server and client are predefined along with the privilege each client application is allocated.

PPR-15 Certificated Application State: A client application either has a certificate that is issued by a third party evaluation authority or by the server application's SP. This certificate has the hash value of the secure (trusted) state of the client application as considered by the third party evaluation authority or the server application's SP. A similar situation applies for the server application's certificate that is issued by the client application's SP or third party evaluators.

PPR-16 Trustworthy TEM: Applications trust the TEM and they have established a long-term shared secret key with it at the time of application installation (section 7.3.2).

7.4.2 Protocol Description

The aim of the Application-Binding Protocol — Local (ABPL) is to facilitate both the client and server applications on the same device to authenticate each other and verify their current states to be secure. The ABPL also enables applications to establish the application binding for future communications. The ABPL message description is as below:

ABPL-1.
$$\begin{aligned} CL &: IMA_{CL} = e_{K_{CL-TEM}}(CL_i||SE_i||N_{CL}) \\ CL \rightarrow F &: CL_i||SE_i||Sign_{CL}(CL_i||SE_i||N_{CL}||IMA_{CL})||CertS_{CL} \end{aligned}$$

The request message contains the identities of the client and server applications together with a random number generated by the CL. In addition, the client application creates an IMA message (section 7.3.2) for the TEM. The client application signs the message and appends its certificate.

ABPL-2.
$$F \rightarrow SE : CL_i||SE_i||Sign_{CL}(CL_i||SE_i||N_{CL}||IMA_{CL})||CertS_{CL}$$

The firewall F receives the application-binding request and it will query the SE. If the server application wants to proceed with the ABPL, it forwards the message; otherwise, it registers an exception.

ABPL-3.
$$\begin{aligned} SE &: IMA_{SE} = e_{K_{SE-TEM}}(SE_i||CL_i||N_{SE}) \\ SE \rightarrow TEM &: CL_i||SE_i||IMA_{CL}||IMA_{SE} \end{aligned}$$

The SE verifies the client's signature. If successful, it generates an IMA message for the CL. The SE then sends the message to the TEM that contains the identities and IMA messages of both the CL and SE.

ABPL-4.
$$\begin{aligned} TEM \rightarrow SL &: VRE_{CL} = e_{K_{CL-TEM}}(h(SE)||K^t_{SE-CL}||N_{CL}+1) \\ TEM \rightarrow SE &: VRE_{SE} = e_{K_{SE-TEM}}(h(CL)||K^t_{SE-CL}||N_{SE}+1) \end{aligned}$$

The TEM verifies the IMA messages from both the CL and SE. Then it will calculate the hash value of the SE, encrypt it with the shared key K_{CL-TEM} and send it to the CL. Similarly, the TEM will calculate the hash value of the CL, encrypt it with the shared key K_{SE-TEM} and send it to the SE. The encrypted messages also contain a session key generated by the TEM; this key is valid only during the ABPL run.

ABPL-5.
$$\begin{aligned} SE &: skm = e_{K^t_{SE-CL}}(e_{SE-CL}||N_{CL}+2||N_{SE}) \\ SE &: au = e_{K_{SE-CL}}(AP||OR||N_{CL}||N_{SE}) \\ SE &: tc = Sign_{SE}(SE_i||CL_i||N_{SE}||au) \\ SE \rightarrow CL &: SE_i||CL_i||IMA_{CL}||skm||tc||CertS_{SE} \end{aligned}$$

Following message four (ABPL-4); the SE verifies the hash value of the CL to be the same as the value listed either by the SE's SP or by a CC evaluation authority. It then generates an application-binding key and encrypts it with the session key. In addition, the message contains the object reference to the SE's shared resources and access permissions. The CL directly calls the SE's shared resource in all subsequent requests, using the binding key for authentication and authorisation.

ABPL-6. $CL \rightarrow SE \quad : \quad CL_i||SE_i||e_{K_{SE-CL}}(AP||(N_{CL}|N_{SE})+1)$

This message gives the assurance to the SE that the CL also has the same key, thus achieving mutual key confirmation (SOG-6: section 6.2.3).

7.5 Platform Binding Protocol

The Platform Binding Protocol (PBP) is executed between two smart cards that are listed as SCA and SCB. Both smart cards can be part of the same CAMS or they may be associated with two different CAMS and this is accommodated by the protocol described in section 7.5.2.

7.5.1 Protocol Prerequisite

The protocol prerequisite for the PBP is fundamentally different from the ones discussed before, as most of them focused on the smart card applications whereas the PBP is focused on the smart card itself. The prerequisite for the PBP is listed below:

PPR-17 Syndicated Members: Both smart cards are registered with a CDAM network, either directly to the same CAMS or two different CAMS on separate devices (e.g. mobile phones, personal computers).

7.5.2 Protocol Description

The protocol can be initiated by any smart card; however, in this section we take SCA as the initiator of the PBP.

PBP-1. $SCA \quad : \quad SCA_{cm} = h(N_{SCA}||g^{r_{SCA}}||SCB'_i)$

$SCA \rightarrow SCB \quad : \quad SCA'_i||SCB'_i||N_{SCA}||SCA_{cm}||SCA_{Sup}$

The first message (PBP-1) contains the pseudo identities of individual smart cards (e.g. SCA and SCB), along with a random number generated by the SCA (N_{SCA}). In addition, the SCA will generate a Diffie-Hellman exponential g^{rSCA} but to prevent a possible partial key chosen attack (see section 6.2.3) it does not send the g^{rSCA}. Instead, it sends a commitment that is basically a hash generated on the g^{rSCA}, random number and the recipient's pseudo identity.

PBP-2.

$$\begin{aligned} SCB &: SCB_{cm} = h(N_{SCB}||g^{rSCB}||SCA'_i) \\ SCB \rightarrow SCA &: SCB'_i||SCA'_i||N_{SCB}||SCB_{cm}||SCB_{Sup} \end{aligned}$$

In response, the SCB will select a Diffie-Hellman group that it can support and include the selection as SCB_{Sup}. The SCB will also generate its commitment (SCB_{cm}) similar to the SCA in the first message, and sends it to the SCA including the SCB_{Sup}. The commitments are made by both communicating entities and now in subsequent messages they can send the generated Diffie-Hellman exponential.

PBP-3.

$$\begin{aligned} SCA \rightarrow SCB &: g^{rSCA}||SCA'_i||SCB'_i||N_{SCA}||N_{SCB} \\ SCB &: K_{DH} = (g^{rSCA})^{rSCB} \ mod\ n \\ SCB &: K_{SCA-SCB} = f_{K_{DH}}(N_{SCA}||N_{SCB}||0) \\ SCB &: mK_{SCA-SCB} = f_{K_{DH}}(N_{SCB}||N_{SCA}||0) \end{aligned}$$

The SCA will send the Diffie-Hellman exponential to the SCB along with pseudo-identities and random numbers generated in previous messages.

On receipt, the SCB will generate the Diffie-Hellman secret (K_{DH}). The SCB generates the PBP master keys ($eK_{SCA-SCB}$ and $mK_{SCA-SCB}$) that are used to generate session keys for the current (e.g. $k_{SCA-SCB}$ and $mk_{SCA-SCB}$) and all future sessions.

PBP-4.

$$\begin{aligned} SCB &: cfb = h(N_{SCA}||g^{rSCB}||g^{rSCA}) \\ SCB &: mE = e_{k_{SCA-SCB}}(VR||SCA'_i||SCB_i||cfb||CertS_{SCB}) \\ SCB \rightarrow SCA &: g^{rSCB}||N_{SCB}||mE||f_{mk_{SCA-SCB}}(mE) \end{aligned}$$

In response, the SCB will ask the platform for assurance and validation proof (i.e. VR) from the SCB. Furthermore, the pseudo identity of the SCA is appended with the true identity of the SCB along with the commitment hash generated (cfb) by the SCA, Diffie-Hellman exponential and cryptographic certificate of the SCB. The entire message, except for the Diffie-Hellman Exponential and the generated random number, is encrypted and MACed using the generated session keys.

On receipt of the message four (PBP-4), the SCA will also generate the Diffie-Hellman secret along with session keys similar to the SCB. It will then verify the SCB's cryptographic certificate. If both smart cards are being evaluated by the same laboratory then

this process will be simple as SCA already trusts that particular evaluation laboratory. Otherwise, it will request the CAMS to traverse the certificate chain to find out whether the SCA's evaluation laboratory is part of that certificate chain. Even if this fails, the SCA can request its card manufacturer to decide whether it should proceed with the binding or not depending upon the provided certificate. Therefore, only if SCA can successfully ascertain the validity of the certificate provider of the SCB's certificate will it proceed with the protocol.

PBP-5.

$$\begin{aligned} SCA &: cfa = h(g^{r_{SCB}}||g^{r_{SCA}}||N_{SCB}||N_{SCA}) \\ SCA &: Val_{SCA} = Sign_{SCA}(cfa||SCA_i||SCB_i||U_i) \\ SCA &: mE = e_{k_{SCA-SCB}}(VR||Val_{SCA}||CertS_{SCA}) \\ SCA \rightarrow SCB &: mE||f_{mk_{SCA-SCB}}(mE) \end{aligned}$$

In response, the SCA will proceed with a platform assurance and validation mechanism (section 4.4). On successful completion, the SCA will generate a message Val_{SCA}. In the message Val_{SCA}, the SCA appends identities of both smart cards and user along with cfa. The signed message (Val_{SCA}) is appended by the certificate and encrypted and MACed by the session keys.

The SCB verifies the SCA's signature and then validates the $CertS_{SCA}$. To verify the certificate chain, the SCB will iteratively employ a similar procedure to SCA discussed as part of the message four. The SCB will also verify the identity of the user of the SCA. The SCB will record whether the user's identity is the same for both smart cards or not. This information will be used by applications to decide whether they would like to establish a communication link with an application installed on a different user's smart card.

PBP-6.

$$\begin{aligned} SCB &: Val_{SCB} = Sign_{SCB}(cfb||SCA||SCB||U_i) \\ SCB &: mE = e_{k_{SCA-SCB}}(Val_{SCB}) \\ SCB \rightarrow SCA &: mE||f_{mk_{SCA-SCB}}(mE) \end{aligned}$$

The SCB will initiate the platform assurance and validation mechanism which generates the hash value of the critical components of the SCB. It will append the Diffie-Hellman exponentials, random numbers and identities of communicating smart cards and the current owner of the SCB. The entire message is signed by the SCB then encrypted and MACed by the session keys.

7.6 Application Binding Protocol — Distributed

The Application Binding Protocol — Distributed (ABPD) is similar the ABP that we discussed in section 7.4. The subtle difference between these two protocols is that the

ABPL uses symmetric cryptography to establish the keying material whereas the ABPD uses asymmetric cryptography.

7.6.1 Protocol Prerequisite

The protocol prerequisite for the ABPD is an extension to the prerequisites of the ABPL that are PPR-14 to PPR-16. The extension of the protocol prerequisite is listed below:

PPR-18 Platform Binding: A platform binding is established between the smart cards, whose applications want to establish an application binding. This means that both smart cards have executed the PBP, described in the previous section.

7.6.2 Protocol Description

The ABPD is executed between two applications that have an application sharing engagement. The protocol listed in this section accommodates the ABP when the two applications are installed on two distinct devices. In the protocol discussed below, the CL resides on the SCA and SE on SCB.

ABPD-1.

$$
\begin{aligned}
CL &: au = f_{K_{SE \rightarrow CL}}(CL_i||SE_i||SP_i^{CL}||SP_i^{SE}||g^{r_{CL}}||N_{CL}) \\
CL \rightarrow SE &: g^{r_{CL}}||N_{CL}||au||DH_{Group} \\
SE &: K_{DH} = (g^{r_{CL}})^{r_{SE}} \ mod \ n \\
SE &: K_{SE-CL} = f_{K_{DH}}(N_{SC^a}||N_{SC^b}||1) \\
SE &: mK_{SE-CL} = f_{K_{DH}}(N_{SC^a}||N_{SC^b}||2)
\end{aligned}
$$

The protocol is initiated by the CL, which generates a Diffie-Hellman exponential ($g^{r_{CL}}$) and a random number. In addition, the application CL also generates a keyed hash of identities of the participating applications, and their respective SPs along with $g^{r_{CL}}$ and N_{CL}. The rationale behind the generation of the keyed hash value is to avoid a man-in-the-middle attack on the first message. To mount this attack, a malicious user has to gain knowledge of identities of individual applications and associated SPs, and the secret key ($K_{SE \rightarrow CL}$). This message cannot prevent replay attacks, which we deal with in subsequent messages. The message is then appended with the DH_{Group}, which details the supported Diffie-Hellman group used to generate the $g^{r_{CL}}$.

On receipt of message one, the SE will verify the authentication credentials (i.e. keyed hash) and on a successful outcome, it will continue with the protocol. The application SE will generate a Diffie-Hellman exponential and a random number. The application SE

will then generate the session keys and long-term encryption and MAC keys in a similar manner to that used in message three of PBP (see section 7.5.2).

ABPD-2.

$$
\begin{aligned}
SE &: h_{SE} = h(SE_i||CL_i||SP_i^{CL}||SP_i^{SE}||g^{r_{SE}}||g^{r_{CL}}||N_{CL}||N_{SE}) \\
SE &: cfs = Sign_{SE}(CL_i||SE_i||h_{SE}) \\
SE &: mE = e_{K_{SE-CL}}(VR||ReqUserID||cfs||CertS_{SE}) \\
SE \rightarrow CL &: g^{r_{SE}}||N_{SE}||mE||f_{mK_{SE-CL}}(mE)||DH_{GroupSel}
\end{aligned}
$$

The SE will generate a signature on the message containing the Diffie-Hellman exponential generated by both the SE and CL, and their identities along with those of the respective SP's concatenated with generated random numbers. The signed message is appended to the SE's certificate along with a request for the CL's state validation and user authentication. The user authentication is an optional parameter in the $ABPD$, which depends upon whether a server application allows application sharing with client applications that are issued to different users. If the SE allows application sharing with different user's CL then the parameter can be omitted. Otherwise, $ReqUserID$ will request the CL to provide the details of the registered owner of the SCA. The message is then encrypted and MACed using the session keys.

On receipt of message two (ABPD-2), the application CL will generate the session and long-term encryption and MAC keys. After this, it will proceed with verifying the MAC and decrypt message two. Subsequently, it will validate the certificate $CertS_{SE}$ and then verify the signature.

ABPD-3.

$$
\begin{aligned}
SCB &: aub = Sign_{SCA}(h(CL)||SCA_i||U_i||N_{CL}||N_{SE}) \\
CL &: h_{CL} = h(CL_i||SE_i||g^{r_{CL}}||g^{r_{SE}}||N_{CL}||N_{SE}) \\
CL &: auc = Sign_{CL}(CL_i||SE_i||h_{CL}) \\
CL &: mE = e_{K_{SE-CL}}(VR||aub||auc||CertS_{CL}||CertS_{SCB}) \\
CL \rightarrow SE &: mE||f_{mK_{SE-CL}}(mE)
\end{aligned}
$$

The CL will then ask the host smart card (SCA) to provide a validation proof. The SCA will generate the hash of the application and then sign it. The signed message also includes the identities of the smart card and its owner (if requested by the SE in message two), and random numbers generated by both applications. In addition, the CL generates a signature on the message containing Diffie-Hellman exponentials generated by communicating applications along with the identities of their respective SPs and generated random numbers. The signing of the message by the smart card provides security assurance and validation of the client application, and the signing of the message by the client application provides entity authentication to the server application.

After the server application SE receives message three, it will first verify whether the generated hash of the application is the same as that certified by either the SP of the SE

or by a third party evaluator. If successful, it will verify the signature generated by the CL. In cases where the SE asks for the user's identity in message two it will also check whether the user identity provided by the message three is as required.

ABPD-4.

$$\begin{aligned} SCB &: aua = Sign_{SCB}(h(SE)||SCA_i||SCB_i||U_i||N_{CL}||N_{SC}) \\ SE &: mE = e_{K_{SE-CL}}(aua||Certs_{SCB}||RL) \\ SE \rightarrow CL &: mE||f_{mK_{SE-CL}}(mE) \end{aligned}$$

In the final message of ABPD, the SE will ask the host smart card SCB to generate the hash. The hash is appended with the identities of the smart cards and user (if required) along with generated random numbers. The resource locator referred as RL provides a handle to the shareable resources provided by the SE. The RL uniquely identifies the smart card on which the server application is installed, and the name of the resources that are being shared (e.g. SIO or RMI object).

On receipt of the final message, the CL will verify the state of the application SE and if required it will verify the identity of the current owner of the SCB.

7.7 Analysis of the Proposed Protocols

In this section, we give details of an informal analysis, followed by mechanical formal analysis based on the CasperFDR. Finally, we describe the test implementation experience with performance measures.

7.7.1 Informal Analysis of the Proposed Protocols

In this section, we consider the proposed protocol and analyse it with respect to the protocol requirements listed in section 7.3.7

• SOG-13: Although an application may have genuine credentials its current state might be modified since it was last evaluated by SP(s) or the CC evaluation laboratory. To verify whether the state of an application is secure enough to initiate application sharing, the ABP requires the TEM to generate a hash of both applications and encrypt them with the corresponding keys. The applications have no influence on the outcome of the hash generation; so they cannot fake their current state. If the current state is considered to have deviated from the stated secure state in the application certificate [56], the recipient can then decide whether to continue the protocol or not.

• SOG-20: A malicious application can be installed with either a server or a client application's AID. However, the ABP does not allow a malicious application to masquerade as a server or client application because to prove the identity of an application, the ABP does not rely on the AIDs. It has a dynamic mechanism with bi-directional exchanges of messages that ascertain the entity and check its credentials (based on cryptographic certificate and signature generation/verification). Therefore, it might be difficult for a masquerading application to match the cryptographic hash (generated by the TEM) and have the signature key of the genuine application.

A malicious user can relay the binding request messages, but when these messages are forwarded to the TEM to generate the hash of the client and server application, a malicious application's hash will not match the certified hash of the client and server application. This is equivalent to violating the 2nd pre-image property of the hash functions [146]. In addition, IMA messages include random numbers that effectively prevent any replay attacks.

The server and client applications authenticate one another. The authentication is achieved through signing the messages along with communicating the application's certificate. The authentication gives an assurance to each of the participant applications that the other application is genuine (effectively avoiding masquerading).

• SOG-21: The application certificate contains details of the user to whom the application was issued. Therefore, if a client application tries to establish an application sharing with a server application, but their customer credential does not match, the request is denied. This avoids application sharing between two applications from different users.

7.7.2 Revisiting the Requirements and Goals

In this section, we only discuss SOG-20 and SOG-21. For SOG-1 to SOG19 refer to sections 6.6.1 and 6.6.3.

All selected protocols for comparison can be adapted to support SOG-20 in the way discussed in section 7.7.1. Furthermore, the PBP does not support both SOG-19 and SOG-20 as it is a protocol designed for establishing a binding relationship between UCTDs, not their applications, whereas, SOG-19 and SOG-20 focus on application binding rather than platform binding. A point to note is that ABPL does not support a number of the SOGs, for the reason that ABPL key generation is based on a symmetric cryptosystem. The ABPL do uses signature algorithms for entity authentication, and they do not play any role in key generation. Furthermore, the key generation in the ABPL is performed mainly by the TEM and server application, without any input from other communicating entities.

Table 7.3: Protocol comparison on the basis of stated goals (see sections 7.3.7 and 6.2.3)

SOG	Protocols										
	STS	**AD**	**ASPeCT**	**JFK**	**T2LS**	**SCP81**	**MM**	**SM**	**PBP**	**ABPD**	**ABPL**
1. Mutual Entity Authentication	*	*	*	*	*	*	−*	−*	*	*	*
2. Exchange Certificate	*	*	*	*	*	*	*	−*	*	*	*
3. Mutual Key Agreement	*	*	*	*	*	*	*	−*	*	*	*
4. Joint Key Control	*	*	*	*	*	*			*	*	
5. Key Freshness	*	*	*	*	*	*	*	−*	*	*	
6. Mutual Key Confirmation	*		*	*			*	−*	*	*	*
7. Known-Key Security	*	*	*	*	*	*	*		*	*	
8. Unknown Key Share Resilience	*	*	*	*	*	*	*	−*	*	*	*
9. KCI Resilience	*	*	*	*	*	*	*	*	*	*	*
10. Perfect Forward Secrecy	*		*	*	*	*			*	*	
11. Mutual Non-Repudiation	*	(*)	+*	*	*	*	+*	+*	*	*	*
12. PCK Attack Resilience	(*)	(*)		(*)	(*)	(*)			*	*	
13. Trust Assurance					*	−*			*	−*	−*
14. DoS Prevention				*					*	*	*
15. Privacy	(*)		*	*					*	*	
16. Simulator Attack Resilience					−*				*	*	*
20. Application Masquerading	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)		*	*
21. Different User's Application										*	*

Note: * means that the protocol meets the stated goal, ** indicates that the protocol meets the SOG if required by the communicating entities, (*) shows that the protocol can be modified to satisfy the requirement, +* shows that protocol can meet the stated goal but requires an additional pass or extra signature generation, and −* means that the protocol (implicitly) meets the requirement not because of the protocol messages but because of the prior relationship between the communicating entities.

7.7.3 CasperFDR Analysis of the Proposed Protocols

The intruder's capability modelled in the Casper scripts (appendices B.6, B.7, and B.8) for the proposed protocol is shown below:

1. An intruder can masquerade as any entity in the network.
2. An intruder can read the messages transmitted by each entity in the network.
3. An intruder cannot influence the internal process of an agent in the network.

The security specification for which the CasperFDR evaluates the network is shown below. The listed specifications are defined in the #Specification section of appendices B.6, B.7, and B.8:

1. The protocol run is fresh and both applications/smart cards are alive.
2. The keys generated during the protocol run are known only to the authenticated participants of the protocol and an adversary cannot retrieve the session keys.
3. Entities mutually authenticate each other and have mutual key assurance at the conclusion of the protocol.
4. Long-term keys of communicating entities are not compromised.

The protocol description defined in the Casper scripts is a simplified representation of the proposed protocols. The off-card agents like the SPs of client and server applications are not modelled in the Casper script as they do not play an active role in the protocol run. The CasperFDR tool evaluated the protocol and did not find any attack(s).

7.7.4 Implementation Results and Performance Measurements

The overall architecture of the test-bed is the same as the architecture discussed in section 4.8.3, consisting of a laptop and two Java Cards (e.g. C1 and C2). We executed individual protocol for 1000 iterations to get the performance measurements.

Our implementation model for the ABPL is based on three applets taking the roles of the TEM, client, and server application on a Java Card (16bit smart card) that take in total 8938 bytes. At the time of testing, we did not have access to an SCOS that

Table 7.4: Performance measurement (milliseconds) of the ABPL

Measures	SSL	TLS	Kerberos	ABPL	
				C1	C2
Average Time	4200	4300	4240	2484	2726
Best Time	NA	NA	NA	2243	2634
Worse Time	NA	NA	NA	2554	2945
Standard Deviation	NA	NA	NA	64.53	76.28

would have enabled us to implement the TEM at the underlying operating system level. We implemented the TEM at the application level and considered that similar or better performance can be attained if the TEM is implemented as part of the platform. Because the application-level implementation of the TEM cannot have memory access to measure the hash values of the client and server applications, we generated the hash of a fixed array of size 556 bytes to represent an application state. The performance of the hash algorithm is based on the size of the input data and in real deployment of the protocol scenario it will depend on the size of the applications. The performance measurements for the ABPL are listed in table 7.4.

The protocols (PBP and ABP) were executed on 16-bit Java Cards, and the implementation took 9799 bytes for the PBP and 8374 bytes for the ABP. The performance measurements were taken from two different sets of 16-bit Java Cards, and an average of recorded measurements for each sets is listed in table 7.5.

Table 7.5: Performance measurement (milliseconds) of the PBP and ABPD

Measures	PBP		ABPD	
	Set One (C1)	Set Two (C2)	Set One (C1)	Set Two (C2)
Average Time	4436.23	4628.35	2998.71	3091.38
Best Time	4078	4235	2906	3031
Worse Time	5469	5875	3922	4344
Standard Deviation	127.89	133.48	96.32	117.71

Note: Set One (C1) means two Java Cards that are similar to the card C1 specification. Similarly, Set Two (C2) refers to the set of C2 Java Cards.

The performance measurements in this section are only for reference our implementation, as the actual performance will vary depending upon the size of the client and server applications (i.e. hash generation), and the performance of public key operation, symmetric encryption, and random number generation on a given smart card.

7.8 Summary

In this chapter, we discussed popular smart card-based firewall mechanisms and how they work. Then we described the unique requirements of the UCTD and presented a firewall mechanism extended from the Java Card firewall. Based on the proposed firewall architecture, we proposed a protocol that establishes the binding between two applications residing on the same smart card. Furthermore, we extended this firewall mechanism to accommodate cross-device application sharing in which two applications residing on different UCTDs can still share their resources. To support cross-device application sharing, we proposed two protocols, one for platform binding and the second for application binding: PBP and ABPD respectively. We then informally analysed the proposed protocols and this analysis was subsequently extended to mechanical formal analysis by the CasperFDR. Finally, we discussed the test implementation and performance measurements of the proposed protocols.

Chapter 8

Smart Card Runtime Environment

Contents

In this chapter we discuss the User Centric Tamper-Resistant Device (UCTD) execution environment in which the downloaded applications will execute. We begin by describing the Java Card runtime environment associated operations. Later on, we articulate the threat model for the UCTD execution environment, along with how it is aggravated by the openness of the UCTD. Subsequently, we look at counter-measures that can be deployed to provide a secure and reliable execution platform. The discussed counter-measures are then compared in terms of their effectiveness and performance.

8.1 Introduction

After an application is installed on to a smart card, it relies on the Smart Card Runtime Environment (SCRT) for secure and reliable execution. An SCRT provides the platform that facilitates the application execution and it contains a library of Application Programming Interfaces (APIs). These APIs provide a secure and reliable interface between the installed applications and on-card services. An SCRT's responsibility is to: 1) handle communication between applications and external entities, 2) provide a secure and reliable program execution, 3) enforce execution isolation and access to memory locations, and 4) provide an interface to access cryptographic algorithms.

Although they are clearly not the same, the distinction between a smart card operating system and a runtime environment is often blurred. For example, Java Card is considered as a platform that provides a runtime environment (i.e. JCRE, see figure 3.3); whereas, Multos is a smart card operating system that also has a runtime environment (i.e. AAM, see figure 3.2). In this chapter, we will refer to an SCRT that semantically encapsulates both the JCRE and AAM. However, we will focus primarily on the JCRE.

An SCRT has to protect the platform and installed applications from malicious or ill-formed applications. In the ICOM, such issues have limited impact because of the strict controls on application installation [190, 194, 195]. This means that compatibility, security, and reliability of an application with respect to an SCRT are evaluated before installing it. Even if a smart card allows post-issuance installation of applications, centralised control means that it is difficult to introduce a malicious application into a smart card, as the card issuer will vet individual applications along with the associated application providers.

In the early days of smart card technology, an adversary could remove the smart card hardware protection layer and access its various components [196]. However, the smart card industry responded to this attack vector and implemented adequate protection, which simply make such attacks more difficult to mount. On the other hand, it led to the materialisation of the side-channel analysis that provided an avenue to attack the smart card platform, especially the cryptographic algorithms [197]. Nevertheless, most of the modern smart cards employ both hardware and software mechanisms to counter side-channel analysis. During early 2000, fault attacks became the modus operandi of adversaries to subvert the implemented security measures in the smart card industry. Since then the technology has evolved to counter these threats to some extent. There has been a growing interest in combining the software and fault injection [194, 198, 199] attacks to subvert the protection mechanisms on a smart card, and such an attack vector is referred as combined attacks. These attacks have significance in the ICOM; nevertheless, the openness of the UCOM exacerbates their effects. In this chapter, we analyse the attacks that target the SCRT and provide counter-measures. During this chapter, we will constantly refer to the JCRE as

compared to the Multos AAM. The rationale is that the JCRE has an open specification, and new attacks mostly target Java Cards. Furthermore, we consider that Java Card is more closely related to the UCTD proposal.

Structure of the Chapter: We begin the discussion with a brief introduction of the SCRT and the combined attacks (i.e. software and fault attacks). In section 8.3, we provide the motivation behind the runtime protection mechanism, which is a collective term used to refer to the proposed counter-measures. Subsequently, we discuss attacker's capability, and detail the runtime protection mechanism. In section 8.4, we analyse the proposed counter-measures for their security, and performance.

8.2 Smart Card Runtime Environment

In this section, we open the discussion with a brief description of the Java Card Virtual Machine (JCVM) with emphasis on those components that we will refer to in the rest of the chapter. Subsequently, we discuss the threat landscape that targets the SCRTs followed by a brief discussion on related work, and finally, we finish the section with the description of fault attacks.

8.2.1 Java Card Virtual Machine

The concepts regarding the Java Card application development and runtime environment are not exhaustively covered in this section. Nevertheless, the rationale for a brief introduction is to make it easy to follow the subsequent discussion regarding the SCRTs.

The JCRE illustrated in figure 3.3 consists of APIs, system classes, Java Card Virtual Machine (JCVM), and native methods. The architecture of the JCVM is more or less similar between various version of Java Cards including the latest Java Card 3.0.1 Connected Edition [16]. The main difference is the support for various system classes and APIs. As far as the core processes are concerned, JCVM for both Java Card 2.X or 3.0.1 Classic Edition and Java Card 3.0.1 Connected Edition are similar, which is in adherence to the Java virtual machine specification [200] (i.e. JCVM is a subset of the Java virtual machine specification [16, 28]).

Before we delve into the details of the JCVM, we first look at the development process of a Java Card application, as illustrated in figure 8.1. An application is coded in a subset of Java language that is supported by the JCVM, which is represented as a Java file in figure 8.1. The application is then compiled into a class file, and it is packaged along with

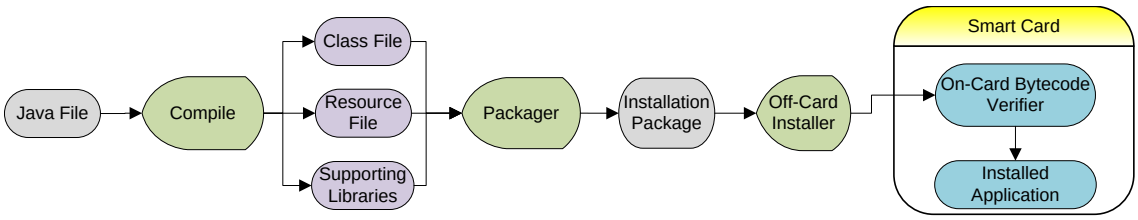

Figure 8.1: Java Card application development process

any resource files and supporting libraries into an installation package (e.g. CAP, or JAR file [16, 28]) that can be downloaded to a Java Card. On the Java Card, the on-card bytecode verifier will analyse the downloaded application and validate that it conforms to the stated Java language semantics.

The class file contains the bytecode representation of the program code, an example is illustrated in figure 8.2. The statement `if_scmplt 22`, at line 08, of bytecode representation is the opcode for if-else statement. The opcode represents that if the statement is true then proceed with next line otherwise jump to line 22. The JCVM for Java Card 3.0.1 has listed approximately 185 opcodes and each opcode (e.g. `if_scmplt`) has an associated byte value. For example, opcode "`if_scmplt`" is represented as byte values 0x6C (in hexadecimal format). The SCRT interprets individual instructions (opcodes) during the application execution.

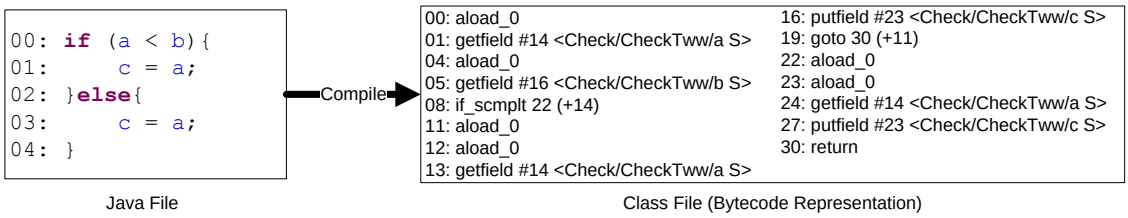

Figure 8.2: Java source file to bytecode conversion

Figure 8.3 illustrates the architecture of a typical JCVM except for the modules in a dotted circle (i.e. runtime security manager) which is part of our proposal discussed in section 8.3. Various components and their functions are described subsequently with emphasis on how they interact during the execution of an application.

The JCVM mainly deals with an abstract storage unit called word that is the smallest storage unit that it can process. The actual size of a word is left to the JCVM implementers and it depends upon the underlying hardware. However, the JCVM specification [16] states that a word should be large enough to hold a value of `byte`, `short`, `reference`, or `returnAddress`.

When an application is initiated, the bytecode representation of an application is loaded to the JCVM memory by a "class loader subsystem". The class loader is responsible for

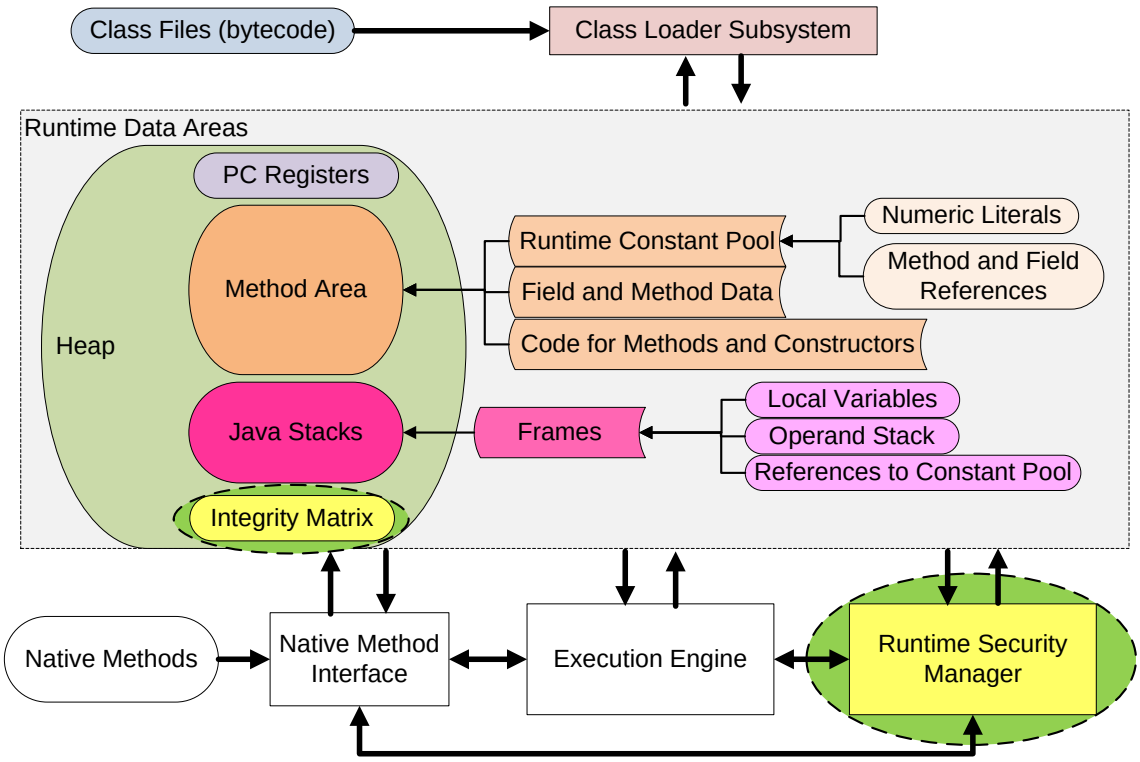

Figure 8.3: Architecture of the Java Card Virtual Machine

locating and loading the class onto the memory areas used by the JCVM. This memory is divided into sub-areas, where each of them contains specific information regarding the application. The JCVM memory area is termed as heap and all data/code related to an application is loaded onto it. The three main storage structures defined on the heap that we are going to discuss here are Program Counter (PC) registers, method area, and Java stacks. These storage structures are briefly discussed here as they are referred to in the remaining chapter, when we discuss our proposed counter-measures.

The PC registers store the memory address of the bytecode instruction currently executing. If the JCVM supports threading then each thread will have its own PC register.

The method area is a shared memory space among executing threads (if the JCVM supports multiple threads) and it consists of structures that include runtime constant pool, field and method data, and code related to methods and constructors. The runtime constant pool stores the constant field values (e.g. numeric literals) and references to the memory address related to methods and fields. The other two structures (e.g. field and method data, and code related to methods and constructors) store the data and code related to fields and methods, etc.

A frame is created by the JCVM each time a method is invoked during the execution of an application. A frame in a JCVM is a construct that stores data, partial results, return

values, and dynamically resolved links, associated with a single method - not the related class. These frames are stored on a last-in first-out (LIFO) stack referred to as Java Stack. For each thread, there will be a different Java Stack. For security reasons, Java Stacks are not directly manipulated by individual applications. The JCVM can only issue the push and pop instructions to Java Stacks. The data structures that reside on a frame include an array of local variables, operand stack, and references to constant pool. The operand stack is a LIFO stack and it is empty when a frame is created. During the execution of a method, JCVM will load data values (of either constant or non-constant variables/fields) onto the operand stack. The JCVM will operate on the values at the top of the operand stack and push the results back on it.

JCVMs provide well-defined interfaces to access native methods; however, contrary to traditional Java virtual machines it does not allow user-defined native methods. Each JCVM has an execution engine that is responsible for execution of the individual instructions (opcodes) in an application code. The design of the execution engine is dependent on the underlying hardware platform and in a simple way, it can be considered as a software interface to the platform's processor.

8.2.2 Related Work

Earlier work on Java Cards was mainly related to the semantic and formal modelling of the JCVM [?, 84, 201, 202], Java Card firewall mechanism [191, 203], and applets [204]–[206]. The assurance for the JCRE reliability against ill-formed applications was based on bytecode verification [128, 161]–[163], which became a compulsory part of the Java Card specification version 3 [16].

In the early 2000s, side channel analysis and fault attacks on smart card platforms were mainly focussed on the cryptographic algorithms [197, 207]–[211]. However, in the second half of the 2000s, logical and fault attacks were combined to target the JCRE [212]–[214].

The discussed attacks in this paragraph are purely logical attacks that use the bugs and inefficient implementation of the JCVM. In 2008, Mostowski and Poll [190] loaded an ill-typed bytecode on various smart cards to test their security and reliability mechanisms. In this work, they showed that on certain smart cards they were able to execute code that should not be possible in the first place (i.e. accessing byte array as short). However, they also noted that smart cards that had an effective on-card bytecode verifier were less susceptible than others.

In 2009, Hogenboom and Mostowski [215] managed to read arbitrary contents of the memory. They performed this attack even in the presence of the Java Card firewall mechanism.

As noted, the reason for the success was the buggy JCVM. Their results were based on eight different smart cards and they only managed to attack one of them, as the other smart cards had effective runtime protection mechanisms. Similar results were also shown by Lanet and Iguchi-Cartigny [195]. Sere et al. [216] use the similar attack of modifying the bytecodes to gain unauthorised access or skip security mechanism on a platform. However, Sere et al. relied on fault attacks to modify the bytecodes rather than modifying them off-card as done by [190, 195, 215]. This way, Sere et al. managed to bypass the on-card bytecode verification. A countermeasure to this attack provided by Sere et al. relied on tagging the bytecode instructions with integrity values (i.e. integrity bits) and during the execution, the JCVM checks these bits and if it fails, the execution terminates.

In 2010, Barbu et al. [194] along with Vétillard and Ferrari [198] used a similar attack methodology to Sere et al. [216] that later came to be known as combined attacks. Later, the combined attack technique was extended to target various components of JCVM in [217]–[220]. These attacks are significant; nevertheless, they require the loading of an application designed specifically to accomplish the attack goals. Therefore, such attacks are not practical to some extent in the ICOM; however, due to the open nature of the UCOM such attacks become a real concern.

In this section we glanced over the attack techniques proposed in the literature that specifically target the SCRT. The discussion is by no means exhaustive but it introduces the challenges faced by the UCTD runtime environment. Before we move to discuss the protection mechanism, we first discuss the fault attacks in some detail in next section.

8.2.2.1 Fault Attacks

The aim of an adversary during a fault attack is to disrupt the correct execution of an application by introducing errors. These errors are usually introduced by physical perturbation of the hardware platform on which the application is executing. By introducing errors at a precise instruction, an adversary can circumvent the security measures implemented by the runtime environment. Possible types of faults an adversary can produce are described as below:

1. Precise bit error: In this scenario, an adversary has total control over the timing and locations of bits that he wants to change.
2. Precise byte error: This scenario is similar to the previous one; however, an adversary only has the ability to change the value of a byte rather than a bit.
3. Unknown byte error: An adversary has no control on the timing and byte that it modifies during the execution of an instruction.

4. Unknown error: In this scenario, an adversary generates a fault but has no location and timing control.

From the above list of fault models, the first model adversary can be considered the most powerful. However, for a smart card environment the second scenario (i.e. precise byte error) is the most realistic one. Due to the advances in the smart card hardware and counter-measures against fault attacks (i.e. especially for cryptographic algorithms) it is difficult to have total control of timing and locations of bits to flip [216]. Furthermore, fault attacks require knowledge of the underlying platform and application execution pattern [191]. This is possible to achieve by side-channel analysis [197].

8.3 Runtime Protection Mechanism

In this section, we provide the motivation behind the runtime protection mechanism, which is followed by the description of the attacker's capability. Subsequently, we discuss the runtime protection mechanism, how it provides a secure and reliable framework for the UCTD runtime environment.

8.3.1 Motivation

During an application's lifetime, it mostly interacts with the runtime environment and the application's security is dependent on the security of the runtime environment. This means that an insecure runtime environment can in fact make a well-designed application insecure. Although, as discussed in section 4.4, a UCTD is required to be certified by a third party evaluation (e.g. CC evaluation); however, we still consider that the runtime environment should not rely only on static security mechanisms including security evaluation and bytecode verifications (both off- and on-card).

As discussed in section 8.2.2, a smart card runtime environment is increasingly facing the convergence of various attack techniques (e.g. fault and logical attacks). Physical protection mechanisms regarding fault attacks are proposed [221]; however, we consider that the necessary software protection for the runtime environment cannot be understated. The software protection can augment the hardware mechanism to protect against the combined attacks, as a similar approach has yielded successful results in the secure design of cryptographic algorithms for smart cards [222]–[224]. In this chapter, we will only focus on the software protection mechanism, without detailing the hardware-based protection.

In literature, several methods are described for software protection mechanism, including application slicing in which an application is partitioned for performance [225, 226] or to protect the intellectual property [227] of an application. Such partitioning can be used to tag individual segments of an application with adequate security requirements. The runtime environment can then take into account the security requirements, tagged with individual segments during the execution; thus providing configurable runtime security architecture. A similar approach is proposed by Java Card 3 [16] and as part of counter-measures to combined attacks proposed by Sere et al. [220] and Bouffard et al. [228]. These proposals are based on using Java annotations to tag segments of an application with required security or reliability levels.

Developers can use Java annotations to provide information regarding an application (or its segment), which is used by either the compiler, or runtime environment (i.e. JCVM). Based on Java annotations, Bouffard et al. [228] and Sere et al. [220] proposed mechanisms to prevent control flow attacks. In addition, Loining et al. [229] used the Java annotations to ensure a secure and reliable development of applications for embedded devices (e.g. smart cards). Furthermore, Java Card 3 Connected Edition also makes provision for Java annotations [16]. The defined annotations by Java Card 3 are integrity, confidentiality, and full (which mean apply both integrity and confidentiality). In addition, the specification also allows proprietary annotations that can be used to invoke specific protection mechanisms implemented by the respective card manufacturer. The Java Card 3 specification does not detail what operations a JCVM should perform when encountering a particular annotation, which are left to the discretion of the card manufacturers.

These proposals are useful, in a closed environment like the ICOM. However, in the UCOM it is difficult to ascertain whether an application has proper (Java) annotations as it is challenging to evaluate their correctness on a smart card. A malicious user can use the annotations to his advantage in order to accomplish his malicious goals. However, if we have an on-card analyser that checks the security and reliability requirements of an application, validate the associated Java annotations (tags) with each segment of the application, and modify the security annotations where adequate. In such a scenario, we may assume that tagging segments of an application with security annotations might be useful in the UCOM. Nevertheless, such an on-card analyser is not available on the smart cards and in this chapter we will not explore its details. In this chapter, we solely focus on adequately hardening of the runtime environment.

In our proposed framework, we tackle the problem from three aspects: application compilation, runtime protection, and trusted component. The Java annotations are used to tag properties of individual segments of an application. Runtime commands (opcodes) that might be subverted to gain unauthorised access are hardened with additional protection (security checks), and finally a trusted component is included to complement the runtime

environment.

8.3.2 Attacker's Capability

Before we delve into the discussion of the proposed runtime protection mechanism, we first define the capabilities of an attacker in the context of a UCOM environment. Due to the advancement in the chip technology and hardware protection mechanisms [230], we have taken a realistic approach in defining the attacker's capability, taking into consideration the current state-of-the-art in attack methodologies for smart cards. The attacker's capabilities taken into consideration for the proposed runtime protection mechanism are listed as below:

1. Has the knowledge of the underlying (hardware and software) architecture.
2. Has the ability to load a customised application onto a given UCTD.
3. Has the capability to induce a fault attack at a precise clock cycle.
4. Has the limited capability of changing a byte value to either 0x00 or 0xFF, or a random value in between.
5. Can change values stored in a non-volatile memory permanently within the limits of the capability four.
6. Has the ability to inject multiple faults; however, only in serial fashion (i.e. after injecting a fault, he waits for the results before injecting the next fault). The adversary cannot inject multiple faults in parallel — injecting two faults simultaneously.

Capability four restricts an adversary to induce a precise byte error rather than the precise bit error (section 8.2.2.1). This restriction is based on the underlying smart card hardware architecture. This is not to say that precise bit errors are not possible in smart cards. On the contrary, they are technically possible but increasing density of packaging (i.e. chip fabrication) makes it challenging to change a value of bit in comparison to changing the value of a byte.

The rationale behind the choice of multiple fault attacks in serial fashion than parallel is to give precise control and reproducibility of the attack. In fault attacks where a malicious user injects multiple faults simultaneously (parallel), it is difficult to assess whether the first fault injection was successful; therefore, injecting the second fault may be less productive. As it may not achieve the desired effect if the first fault did not work.

In our proposed framework, we intend to protect the underlying runtime environment and applications hosted on it. However, if an application is designed with an intention that

it releases sensitive information associated with it or its users, such applications that are designed to self-harm are difficult to protect. For example, if an application is designed in a way that it reveals its user's private key (specific to the application - not related to the platform or other applications); there is a limit to what a protection mechanism can do to prevent such leakages.

8.3.3 Overview of the Runtime Protection Mechanism

The proposed architecture of the runtime protection mechanism is involved at various stages of the application lifecycle - including the application compilation, on-card bytecode verification, and execution as shown in figure 8.4.

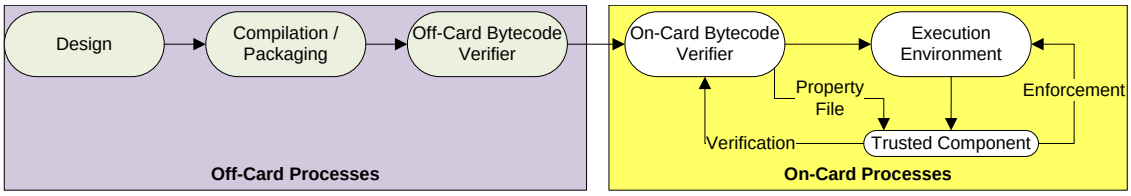

Figure 8.4: Generic Overview of the runtime protection mechanism

During compilation/packaging process additional information regarding individual methods, classes, and objects of an application is generated as part of the property file, discussed in section 8.3.4. The property file assists the runtime environment to provide a security and reliability service during the execution of the application. The off-card bytecode verification checks whether the downloaded application conforms to the (given) language's semantics. The on-card bytecode verifier can also request the TEM to validate the property file. During the application execution, the TEM will actively enforce the security and reliability policy of the platform - taking into account the information included in the property file.

The proposed framework does not require that application developers perform security assessment of their application(s) to adequately tag application segments. The framework only requires that developers compile their applications in a way that it has a property file that stores information related to the respective application. The second requirement of the proposed framework is to adequately harden the UCTD runtime environment discussed in section 8.3.5 along with introducing a trusted component (part of the TEM) that will enforce the platform security policy (section 8.3.6).

In subsequent sections, we will extend the generic architecture discussed in this section and explain how these different components come together to provide a robust and secure runtime environment for UCTD.

8.3.4 Application Compilation

It might be considered adequate to modify the Java virtual machine specification for the smart card environment to provide an effective runtime protection mechanism; for example, reducing the number of opcodes and removing opcode zero from opcode list. However, we avoid it for the sake of simplicity. Instead we use the property files that include meta-data about the respective application. The property file is downloaded to the smart card as part of the application and verified by the respective TEM during the bytecode verification as shown in figure 8.1.

A Java compiler will take a Java file and convert it to a (bytecode) class file. The class file not only has opcodes, but it also includes information about various segments (e.g. methods, and classes) of an application that is necessary for the JCVM to execute the application. However, for our proposal we introduce a property file that includes additional information about an application. If a JCVM knows how to process property files then it will proceed with them; otherwise, it will silently ignore them. In our proposal a property file is stored and used by the TEM during the execution of the associated application. In order to integrate the TEM into the runtime environment, the JCVM is required to be modified so it can communicate with the TEM in order to safeguard the execution environment.

```
1  ApplicationInfo{
2        Application_Identifier ApplicationIdentifier;
3        ClassInformation ClassInfo[class_count];}
4  ClassInfo{
5        Class_Identifier ClassIdentifier;
6        MethodInformation MethodInfo[method_count];}
7  MethodInfo{
8        Method_Identifier MethodIdentifier;
9        MethodIntegrity HashValue;
10       ControlFlowGraph Graph[jumps_count]}
```

Listing 8.1: Structure of the property file of a Java Card application.

The property file contains security and reliability information concerning an application that the runtime environment can utilise to execute an application. The structure of the property file is illustrated in listing 8.1, which includes information regarding the control-flow graph, and integrity matrix (hash values of the non-mutable part of the individual methods in a class).

The `ApplicationInfo` data structure includes the application identifier (e.g. AID) and an array of classes that are part of the respective application. For each class in the application, we have a `ClassInfo` structure that contains the `MethodInformation` array that contains information regarding all methods associated with the given class. Each method is represented by `MethodInfo` structure that includes the control-flow graphs that are gen-

erated for each method. In the control-flow graphs, child nodes represent jumps to other methods whether they are from the same application or from a different application. In a way, combining the method graphs of all classes can give the complete control-flow graph of the respective application. In addition to the control-flow graph, a `MethodInfo` also contains the hash value (of non-mutable code) of the respective method. This hash value can be generated at the compile time and added to the property file, or at the time of the application installation: the TEM calculates the hash value and stores it in the property file.

8.3.5 Execution Environment

The runtime environment of a UCTD platform is adequately modified to support the inclusion of the TEM (i.e. runtime security manager) that is shown in figure 8.3. At the time of application installation, the application bytecode is stored in the respective SP's domain along with the associated property file. The property file is sealed[1] by the TEM so that neither the application nor an off-card entity (e.g. an SP or/and adversary) can modify it. At the time of execution, the TEM will retrieve the file, verify the integrity of the file, and then decrypt it. If an SP wants to update its application on a UCTD then it will proceed with the update command[2] that will notify the TEM of the update. At the completion of the update, the TEM will verify the application security certificate (if available), and update the property file.

8.3.6 Runtime Security Manager

The purpose of the runtime security manager is to enforce the security counter-measures (section 8.3.7) defined by the respective platform. To enforce the security counter-measures, the runtime security manager has the access to the heap area (e.g. method area, Java stacks) and it can be implemented as either a serial or a parallel mode.

A serial runtime security manager will rely on the execution engine of the JCVM (figure 8.3) to perform the required tasks. This means that when an execution engine encounters instructions that require an enforcement of the security policy, it will invoke the runtime security manager that will then perform the checks. If successful the execution engine continues with execution, otherwise, it will terminate. A parallel runtime security manager will have its own dedicated hardware (i.e. processor) support that enables it to perform

[1]Sealed: The data is encrypted by the TEM storage key. The storage key is a symmetric key to encrypt the sensitive data like property file so applications cannot change them

[2]Update Command: We do not propose any update command in this thesis but similar commands are defined as part of the GlobalPlatform card specification. The update command enables an authorise entity (e.g. SP) to modify an application.

checks simultaneously while the execution engine is executing an application. Having multiple processors on a smart card is technically possible [5]. The main question regarding the choice is not the hardware, but the balance between the performance and latency.

Performance, as the name suggests is concerned with the computational speed. Whereas, latency deals with the number of instructions executed between an injected-error to the point it is detected. For example, if during the execution of an application 'A', at instruction A4 a malicious user injects an error, which is detected by the platform security mechanism at instruction A7 of the application, the latency is three (i.e. 4-7=3). A point to note is that the lower the latency value the better the protection mechanism, as it will catch the error quickly. Therefore, theoretically we can assume that a serial runtime security manager will have the low performance but also low latency value, where for a parallel runtime security manager it will have good performance measure but higher latency value. We will return to this discussion later in section 8.4 where we provide test (simulated) implementation results.

It is obvious that implementation of additional components like runtime security manager will also incur additional economic costs (i.e. increase in the price of a UCTD); however, in this thesis we are not concerned with the economic cost of UCTDs.

8.3.7 Runtime Security Counter-Measures

The runtime security manager along with the runtime environment would apply the required security counter-measures (as part of the runtime protection mechanism) that are discussed in subsequent sections.

8.3.7.1 Operand Stack Integrity

As discussed in section 8.2.1, an operand stack is part of the Java stacks and they are associated with individual Java frames (methods). During the execution of an application, the runtime environment pushes and pops local variables, constant fields, and object references to the operand stack. The instructions specified in an application can then process the values at the top of the stack. Barbu et el. [217] showed that a fault injection that changes the values stored on the operand stack could have adverse effect on an application's security. Furthermore, they also provided three different counter-measures to the proposed attack and their second-refined method (countermeasure) is closely related to our protection mechanism.

The proposed countermeasure (second-refined method) of Barbu et al. [217] is based on the

idea of operand stack integrity. They define a variable α, and all values that are pushed on or popped from the operand stack are XORed with the α. Therefore, α is the summation of all the values that are on the operand stack at any point of an application execution. For example, if values o_1, o_2, o_3, and o_4 are on a stack then the α will be $\alpha = o_1 \oplus o_2 \oplus o_3 \oplus o_4$, which can be written as $\alpha = \Sigma_{i=1}^{n} o_i$ where n=4.

According to Barbu et al. [217] on every jump instruction beyond the scope of the current frame (method), the runtime environment XORs all the values stored on the operand and compares the result with α. If they match then the integrity of the operand stack is verified. Their proposal does not measure the integrity of the operand stack on instructions like if-else or loops, which could be the target of the malicious user. In fact, Barbu et al. [217] detail an attack that targets the conditional statement (e.g. if-else) and showed how a malicious user can circumvent the PIN verification in their example application. However, the second-refined method do not protect against such attacks. In their proposed counter-measures they sacrificed security and (to some extent) performance for the sake of memory use, whereas our proposal focuses on security rather than saving the memory. A point to note is that in a traditional smart card (in ICOM) memory is crucial as to keep the cost of the smart card in a reasonable range; however, in the UCOM we focus on the security - sacrificing the (crucial) cost of the final product (e.g. UCTD).

```
1 // Executed by runtime security manager when a value is pushed onto an
      integrity stack.
2 On_Stack_Push(pushedValue){
3  push(InS[top] XOR pushedValue);
4 }
5 // Executed by runtime security manager when a value is popped from an
      operand stack.
6 On_Stack_Pop(poppedValue){
7   if(pop(InS) XOR poppedValue := InS[top]){
8   }else{
9    terminateExecution();
10   }
11 }
```

Listing 8.2: Operand stack integrity operations.

In our proposal, we use a Last In First Out (LIFO) stack referred as integrity stack that can store values of a "word" size, which is the most elementary data structure defined in a JCVM. As already mentioned, the actual size of the word is platform dependent and it is left to the discretion of platform implementers. One thing to note is that JCVM knows the size of the operand stack when it loads a frame (section 8.2.1); therefore, the runtime security manager just creates an integrity stack of the size n where n is the size of the respective operand stack (created by the JCVM). We refer to the integrity stack as "InS" in listing 8.2.

When a frame is loaded, the JCVM and runtime security manager will create an operand

and integrity stack, respectively. Furthermore, the runtime security manager will also generate a random number and stores it as S_r. The rationale for using the random number will become apparent in the subsequent discussion.

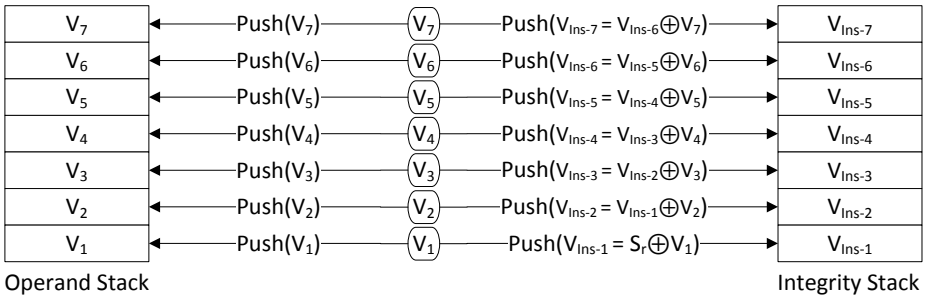

Figure 8.5: Operand and integrity stack push operations

Consider there are seven values (V_1, V_2, V_3 , ... , V_7) that are going to be pushed onto an operand stack. The operations performed at each push operation for these seven values are shown in figure 8.5. When V_1 is pushed onto the operand stack, the integrity stack does not have any value. Therefore, at the beginning integrity stack will XOR V_1 with the generated random value S_r: it is the starting point of the integrity calculation. When an item is pushed on to the operand stack, we XOR the pushed value with the value on the top of the integrity stack. The result is pushed back on to the integrity stack. The push operation can be represented as $V_{Ins\text{-}n}=V_{Ins\text{-}(n\text{-}1)}\oplus V_n$, where n is index to the integrity stack, $V_{Ins\text{-}n}$ is the value stored on the integrity stack. Furthermore, the value on the top of an integrity stack is $V_{Ins\text{-}n}=S_r\oplus\Sigma_{i=1}^{n}V_i$. Therefore, if a card manufacturer wants to implement the α as proposed by the Barbu et al. [217] then it can simply do it by $\alpha = S_r\oplus V_{Ins\text{-}n}$.

The rationale for using a random number is to avoid parallel fault injections that try to change the values on both operand and integrity stack simultaneously. Such a parallel fault injection will become difficult if an adversary cannot predict the values stored on the integrity stack, as each value on the integrity stack will be chained with the generated random number. One point to note is that, although the attacker's capability defined in section 8.3.2 prohibits parallel fault injection but we still try to accommodate it in our proposals; as such attacks might become realistic in future.

When a value is popped out of the operand stack, we also pop the integrity value from the integrity stack, XOR it with the popped value from the operand stack and compare it with the new top value on the integrity stack. If the values match then integrity of the popped value from the operand stack is verified; otherwise, it has been corrupted and the runtime security manager requests the JCVM to terminate the execution as shown in listing 8.2. To explain it further, consider that we pop V_7 from the operand stack in figure 8.5. The runtime security manager will also pop $V_{Ins\text{-}7}$ from the integrity stack, calculate InsValue = $V_{Ins\text{-}7}\oplus V_{Ins\text{-}6}$ and compare the InsValue with the V_7. If InsValue and V_7 match, then the JCVM will proceed with the execution; otherwise, it will abort the execution.

The runtime security manager will continuously monitor the integrity of the operand stack, in comparison to the Barbu's proposal. Furthermore, in this proposal the validation does not require the calculation of integrity value over the entire operand stack. If we take the Barbu's proposal then for an operand stack of length 'n', we have to perform "n-1" XOR operations every time we need to verify the state of the operand stack. However, in our proposal we only need to perform one XOR operation. We sacrifice the memory for the sake of performance in our proposal. We consider that operand stacks are not large data structures so even if we double the memory used by them, it will not have an adverse effect on the overall memory usage.

8.3.7.2 Control Flow Analysis

Control flow analysis, monitors whether the jumps performed in a method are legal or not. In our proposal, we are concerned with jumps that refer to external resources. The term external resources in the context of control flow analysis means any jump that goes beyond the scope of the current Java frame (i.e. method) while it is still on the Java stack. Once a method completes its execution, the JCVM will remove the associated Java frame from the Java stacks (figure 8.3). Examples of such jumps defined in Java virtual machine specification [200] are `invokeinterface`, `invokestatic`, `invokevirtual`, `areturn`, etc.

```
1 byte B(byte inputValue){
2  byte a = 1;
3  if(inputValue != a){
4   C(inputValue);
5  }else{
6   D(inputValue)
7  }
8  return SG(inputValue);
9 }
```

Listing 8.3: Code for an example method B.

To explain the control flow analysis further, we consider an example method `B` that has three jumps before it reaches the return statement that completes the execution of the method. The control flow diagram of method `B` is shown in figure 8.6 and associated code in listing 8.3. Each invocation of a method (e.g. `C`, `D`, and `SG`) shown in the control flow diagram in figure 8.6 is represented by a symbolic method name (i.e. alphanumeric form that is easily readable/recognisable by humans) that has an associated unique byte sequence referred as method identifier in section 8.3.4. For example, unique method identifier of methods `B`, `C`, `D`, and `SG` are 0xF122, 0xF123, 0xF124, and 0xF125, respectively. For explanation we have used method identifiers that consist of two bytes. Along with the method identifier the property file also includes `ControlFlowGraph`, which is a set of legal control flows sanctioned for the given method.

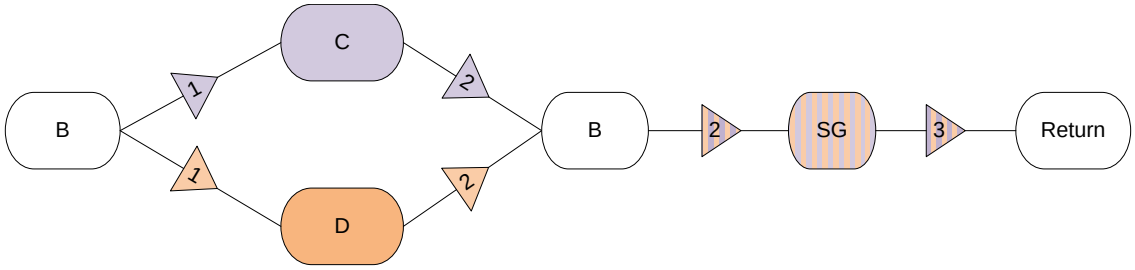

Figure 8.6: Control flow diagram of an example method B

The `ControlFlowGraph` in the property file (listing 8.1) is simply constructed by taking into account every possible (legal) execution flow of a method. Taking the example method `B`, as shown in figure 8.6 the first jump can either be to method `C` or `D` depending upon the input to method `B` (`inputValue` in listing 8.3). The first two possible jumps shown in figure 8.6 are `B`→`C` and `B`→`D`, where "→" represents the direction of the jump. The construction of the `ControlFlowGraph` (set of legal jumps) is constructed by XORing the method identifiers of individual jumps (`B`→`C` and `B`→`D`). The first legal jump in the `ControlFlowGraph` would be either $\text{Jump}_1 = \text{0xF122} \oplus \text{0xF123}$ (i.e. `B`→`C`) and $\text{Jump}_1 = \text{0xF122} \oplus \text{0xF124}$ (i.e. `B`→`D`). The next possible jumps in the method `B` can be either `C`→`SG` or `D`→`SG` that are represented in the `ControlFlowGraph` as $\text{Jump}_3 = \text{Jump}_1 \oplus \text{0xF125}$ and $\text{Jump}_4 = \text{Jump}_2 \oplus \text{0xF125}$, respectively. Finally, for the third jump illustrated in figure 8.6 is `SG`→`Return` that returns the execution back to the method that initiated the method `B`. Therefore, the `ControlFlowGraph` of method `B` would be $\text{B}_{\text{cfa-Set}} = (\text{Jump}_1, \text{Jump}_2, \text{Jump}_3, \text{Jump}_4)$.

The control flow analysis requires that the runtime security manager have a control flow analysis variable "cfa" that stores the path taken by an application as $cfa = \Sigma_{j=1}^{n} C_j$. Where C_j represents the jumps taken during execution of an application. During the execution of a method, when the JCVM encounters a jump to another method the runtime security manager XORs the method identifier with the current value of "cfa" and lookup the `ControlFlowGraph` of the given method in the associated property file. If it finds a matching value, the JCVM will proceed with the execution; if not it will terminate the execution. Taking our example of the method `B`, when the JCVM encounters the first jump `B`→`C` the runtime security manager will calculate the $cfa = \text{0xF122} \oplus \text{0xF123}$ and compare it with the values in the respective `ControlFlowGraph`. As the cfa matches with the value Jump_1, the runtime execution manager assumes the jump `B`→`C` is legal (permitted).

A potential problem with this scheme might be loop instructions that contain jumps to multiple methods depending upon the loop condition. For example, for an odd value of 'i' jump to method `B` and for even values jump to method `C`. The loop iterates through the values of 'i' until it meets the condition that might be based on runtime values (i.e. unpredictable at the time of the compilation of the application). However, we consider that this problem is intrinsically managed by the scheme. Consider a control flow graph of four methods: A, B, C, and D. Methods B and C are part of a loop as dis-

cussed before and illustrated in listing 8.4. The `ControlFlowGraph` set will be $A_{\text{cfa-Set}}$ = {A⊕B, A⊕C, A⊕D, A⊕B⊕D, A⊕C⊕D, A⊕B⊕C⊕D}. Therefore, a potential execution path might be A→B→C→B→C→B→C→D. Therefore, if we compute the "cfa" it would be A⊕B⊕C⊕B⊕C⊕B⊕C⊕D that is effectively A⊕B⊕C⊕D, which is a member of the `ControlFlowGraph` set $A_{\text{cfa-Set}}$.

```
1 byte A(byte inputValue){
2   for(byte i=0; i<inputValue;i++){
3    if(i % 2 == 0){
4     C(inputValue);
5    }else{
6     B(inputValue);
7    }
8   }
9  return D(inputValue);
10 }
```

Listing 8.4: Handling loop statements in the control flow analysis.

8.3.7.3 Bytecode Integrity

The property file associated with an application stores the hash values of individual methods. When the runtime environment fetches an application, the runtime security manager will measure the integrity value of individual methods of the application and compare them with the hash values in the property file. Therefore, any method that is loaded to the heap goes through the integrity validation. This validation protects against the fault attacks on an application stored while it is stored on a non-volatile memory.

8.4 Analysis of the Runtime Protection Mechanism

In this section, we evaluate the proposed counter-measures for their suitability against the attacks discussed in section 8.2.2 under the adversary's capability detailed in section 8.3.2. Furthermore, we provide the latency analysis and performance measurements for both serial and parallel runtime security managers.

8.4.1 Security Analysis

In this section, we discuss how the proposed counter-measures protect against the combined attacks under the attacker's capability detailed in section 8.3.2.

8.4.1.1 Operand Stack Integrity

Barbu et al. [217] proposed an attack in which values stored on the operand stacks were manipulated by fault injections. They also proposed a countermeasure to this attack that was based on calculating the integrity measurement of the whole of the operand stack, every time the state of the stack was required to be verified. We refined their approach and removed the need to perform integrity measurement of the entire operand stack on each validation. In addition, we made the validation process continuous thus checking the integrity of the operand stack on each pop and push operation. If a malicious user changes values on the operand stack, the runtime security manager can not only detect the modification but can also provide error correction service by providing the correct value that was stored on the operand stand. This is possible because the integrity stack stores values pushed on to the operand stack as individual components of the integrity chain (i.e. $V_{\text{Ins-n}}=S_r \oplus \Sigma_{i=1}^{n} V_i$). Furthermore, our proposal also protects against parallel fault injection attacks that could target both operand and integrity stack simultaneously. The reasoning behind this is based on the use of S_r (random number) that makes the values stored on the integrity stack unpredictable over different execution sessions of the same application. Thus making it difficult for an adversary to know the values stored on the integrity stack, even if he has the knowledge of all values on the operand stack.

8.4.1.2 Control Flow Analysis

The control flow analysis performed by the runtime security manager during the execution of an application effectively prevents control flow attacks. If an attacker has the capability of multiple fault injections simultaneously, (which is beyond the stated capability of our attacker in section 8.3.2) then he can in theory affect the runtime security manager execution. Nevertheless, even with simultaneous injection the attacker may be able to skip a node in the execution tree but the runtime security manager calculation on the subsequent nodes will reveal an illegal path of execution. Therefore, even in the parallel injection model the runtime security manager will detect the erroneous execution path, unless the attacker will constantly keep on introducing injections for the whole execution of an application.

8.4.1.3 Bytecode Integrity

This countermeasure is proposed to prevent an adversary to change an application while it is stored on a non-volatile memory (capability four of an adversary discussed in section 8.3.2). To avoid such modifications, the runtime security manager generates a hash of individual methods that are requested by the JCVM. If the hash matches the value

stored (`MethodIntegrity` in listing 8.1) in the respective property file, the JCVM will proceed with execution of the method; otherwise, the runtime security manager will signal the termination of the application (and possibly mark it malicious and up for deletion). Furthermore, this protection mechanism can also safeguard the dynamic loading of applications/classes/routines as part of the web server or other applications, which are stored on off-card storage.

8.4.2 Evaluation Context

For evaluation of proposed counter-measures, we have selected four sample applications. Two of the applications selected are part of the Java Card development kit distribution: Wallet and Java Purse. The other two applications are the implementation of our proposed mechanisms that include the offline attestation algorithm (section 4.5) and STCP_{SP} protocol (section 6.3).

8.4.3 Latency Analysis

As discussed before, latency is the number of instructions executed after an adversary mounts an attack and the system becomes aware of it. Therefore, in this section we analyse the latency of proposed counter-measures under the concept of serial and parallel runtime security managers that are listed in table 8.1 and discussed subsequently.

Table 8.1: Latency measurement of individual countermeasure

counter-measures	**Serial runtime security manager**	**Parallel runtime security manager**
Operand Stack Integrity	0 + i	3 + i
Control Flow Analysis	0	$3(C_n)$
Bytecode Integrity	0	0

In case of the operand stack integrity, the serial runtime security manager finds the occurrence of an error (e.g. fault injection) with latency "0+i", where 'i' is the number of instructions executed before the manipulated value reaches the top of the operand stack. For example, consider an operand stack with values V_1, V_2, V_3, V_4, and V_5, where V_5 is the value on the top. If an adversary changes the value of V_3 by physical perturbation, then the runtime security manager will not find out about his change until the value is popped out of the stack. Therefore, the value of 'i' depends upon the number of instructions that will execute until the V_3 reaches the top of the operand stack and JCVM pops it out. Similarly, the latency value in case of the operand stack integrity for the parallel runtime security manager is "3+i", where '3' is the number of instructions required to perform a

comparison on pop operation (`On_Stack_Pop(poppedValue)` in listing 8.2). The latency value of the parallel runtime security manager is higher than the serial. This has to do with the fact that while parallel runtime security manager is applying the security checks the JCVM does not need to stop the execution of subsequent instructions.

Regarding the control flow analysis, the serial runtime security manager has a latency of zero where the parallel runtime security manager has latency value of "$3(C_n)$", where the value C_n represents the number of legal jumps in the respective `ControlFlowGraph` set. To explain this further, consider the example shown in figure 8.6. The `ControlFlowGraph` of method `B` has four possible values ($B_{\text{cfa-Set}}$ in section 8.3.7.2). Thereby, the latency value for a jump in the method `B` in the worse case is "3(4) = 12". The value '3' represents the number of instructions required to execute individual comparison.

A notable point to mention here is that all latency measurements listed in the table 8.2 are based on the worst-case conditions. Furthermore, it is apparent that it might be difficult to implement a complete parallel runtime security manager. To explain our point, consider two consecutive jump instructions in which the parallel runtime security manager has to perform control flow analysis. In such situation, there might be a possibility that while the runtime security manager is still evaluating the first jump, the JCVM might initiate the second jump instruction. Therefore, this might create a deadlock between the JCVM and parallel runtime security manager - we consider that either JCVM should wait for the runtime security manager to complete the verification, or for the sake of performance the runtime security manager might skip certain verifications. We opt for the parallel runtime security manager that will switch to the serial runtime security manager mode - restricting the JCVM to proceed with next instruction until the runtime security manager can apply the security checks. This situation will be further explained during the discussion on the performance measurements in the next section.

8.4.4 Performance Analysis

To evaluate the performance impact of the proposed counter-measures we developed an abstract virtual machine that takes the bytecode of each Java Card applet and then computes the computational overhead for individual countermeasure. When a Java application is compiled the java compiler (`javac`) produces a class file as discussed in section 8.2.1. The class file is Java bytecode representation, and there are two possible ways to read class files. We can either use a hex editor (an editor that shows a file in hexadecimal format) to read the Java bytecodes or better utilise the `javap` tool that comes with Java Development Kit (JDK). In our practical implementation, we opted for the `javap` as it produces the bytecode representation of a class file in human-readable mnemonics as represented in the JVM specification [200]. We used the `javap` to produce the mnenomic bytecode

representation; the abstract virtual machine takes the mnenomic bytecode representation of an application and searches for push, pop, and jump (e.g. method invokes) opcodes. Subsequently, we calculated the number of extra instructions required to be executed in order to implement the counter-measures discussed in previous sections.

Table 8.2: Performance measurement (percentage increase in computational cost)

Selected Applications	**Serial runtime security manager**	**Parallel runtime security manager**
Wallet	29.43%	22.67%
Java Purse	30.30%	25.82%
Offline Attestation	17.64%	12.93%
$STCP_{SP}$	38.48%	33.23%

To compute the performance overhead, we counted the number of instructions an application has and how long the application takes to execute on our test Java Cards (e.g. C1 and C3). After this measurement, we have associated costs based on additional instructions executed for each JCVM instruction and calculated as an (approximate) increase in the percentage of computational overhead and listed in table 8.2. Furthermore, to measure the cost of the hash generation — we used the hash generation performance measurements for the test Java Cards illustrated in figure 6.2.

For each application, the counter-measures have different computational overhead value because they depend upon how many times certain instructions that invoke the counter-measures are executed. Therefore, the computational overhead measurements in table 8.2 can only give us a measure of how the performance is affected in individual cases - without generalising for other applications.

8.5 Summary

In this chapter we discussed the smart card runtime environment by taking the Java Card as a running example. The JCRE was described with its different data structures that it uses during the execution of an application. Subsequently, we discussed various attacks that target the smart card runtime environment and most of these attacks based on perturbation of the values stored by the runtime environment. These perturbations are called fault injection, which was defined and mapped to an adversary's capability in this chapter. Based on these recent attacks on the smart card runtime environment, we proposed an architecture that includes the provision of a runtime security manager. We also proposed various counter-measures and provided the computational cost imposed by these counter-measures. No doubt, counter-measures that do not change the core architecture the Java virtual machine, will almost always incur extra computational cost. Therefore, we

concluded in this chapter that a better way forward would be to change the architecture of the Java virtual machine. However, in the context of this thesis we showed that current architecture can be hardened at the cost of a computational penalty.

Chapter 9

Backup, Migration, and Decommissioning Mechanisms

Contents

In this chapter we analyse the backup and migration mechanisms that allow a user to securely backup, migrate, and restore her smart card contents. These mechanisms enable a user to retain the same set of applications if she loses her smart card or wants to move to a new smart card. We conclude the chapter with a discussion of the application deletion mechanism that is the final stage in the lifecycle of an application and a UCTD (i.e. decommissioning).

9.1 Introduction

One of the main features of the UCOM is dynamic (wherever, whenever) acquisition of applications by users. Therefore, the UCOM framework enables a user to have most if not all of her applications on a single device. However, this also increases the potential damage if the device is lost. To expedite the recovery process after theft or loss, customers should be able to have their applications restored as quickly as possible to a new devices.

A backup mechanism enables a user to backup her smart card contents. In adverse circumstances, such as losing her smart card, she could retrieve and restore the contents onto a new smart card. Furthermore, a similar mechanism referred to as a migration mechanism can also be used if a user decides to upgrade to a new feature-rich UCTD.

There are some subtle challenges to backup and migration mechanisms in the UCOM, especially in the case of card-bound application leases (section 5.4.3) that restrict applications to their host smart cards. Furthermore, there is a possibility that the remote location (e.g. backup server) might not be tamper-proof and a malicious user could take advantage of it. Therefore, it would be safe to assume that instead of transferring whole applications (i.e. code and data), we should only transfer application download credentials. These credentials can be considered as authorisation tokens that are issued by the respective SPs, so a user could use them to acquire the application in future.

Finally, we discuss the last lifecycle stage of an application — application deletion. As the UCOM allows a user to install and delete any application they desire, this privilege might lead to feature interaction problems discussed as SCR8 and SPR9 in section 3.5. Feature interaction problems arise from environments where two applications share resources and later at some point one of the applications is deleted. The second application's dependencies on the first application might not be resolved, which leads to a situation where the second application tries to access the first application, which no longer exists on the platform. Such scenarios may lead to possible security breaches in the UCTD environment.

We analyse the application deletion process in prominent smart card frameworks: Java Card, GlobalPlatform, and Multos, focusing on how they resolve deadlock conditions. A deadlock condition arises when a user tries to delete an application 'A' which is sharing resources with an application 'B'. Deleting the application 'A' might affect the operations of the application 'B', eventually leading to a feature interaction problem. To avoid such scenarios, we propose a framework that tries to resolve such deadlocks during the application deletion process.

Structure of the Chapter: Section 9.2 begins with a discussion on smart card contents backup and migration mechanisms, followed by the application deletion process in section

9.3. Finally, we conclude the chapter with a discussion on the decommissioning process in section 9.4.

9.2 Backup and Migration Framework

In this section, we describe two mechanisms: backup and migration. In the contents backup process, a user archives her smart card's contents to a backup server and then restores it to the destination smart card. In the migration process, there is no backup server and the smart card contents are transferred between a source and a destination smart card.

9.2.1 Backup Mechanism

In this proposal, instead of backing up the applications (i.e. data and source code) as we traditionally do in desktop computing environments. We only backup the authorisation tokens issued by SPs. The backup package that consists of authorisation tokens should be stored at a secure location, preferably accessible ubiquitously on demand. When a user wants to restore the contents of her old smart card, she has to import the backup package; then the individual applications will be requested from their respective SPs automatically by the smart card using the authorisation tokens.

In our proposal, a secure off-site backup facility is provided by a secure third party referred to as a backup server. We do not consider that a backup server has to be an SP and the only requirement is that users trust the backup server. A backup framework overview is illustrated in the figure 9.1 and described below.

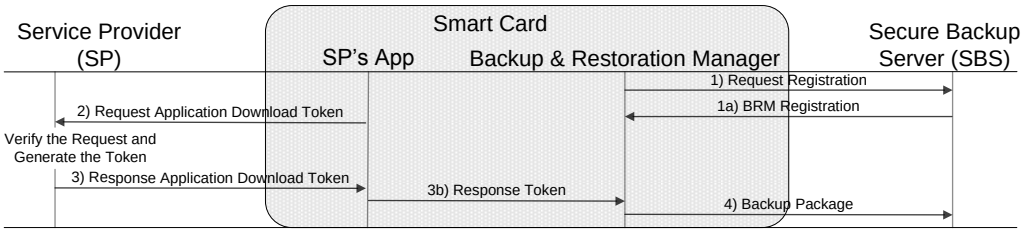

Figure 9.1: Overview of the credential backup mechanism

1. A smart card user registers herself to a backup server using the Secure and Trusted Channel Protocols (STCPs) proposed in chapter 6. After the registration, the backup & restoration manager (figure 4.1) has the user's credentials and details of how to connect with the respective backup server. The backup & restoration manager and backup server will generate a shared secret that they will use in future sessions.

As this shared secret is bound to the specific smart card, it is only used for secure communication and not sealing (encrypting) the backup tokens.

2. After an application is installed on a smart card, the application can initiate the request for an authorisation token only if it is sanctioned by the appropriate SP. We opted for two possible scenarios: restorable and non-restorable applications. These types are inspired by the security policy related to key migration in the TPM specification [18]. For restorable applications, an SP will issue its application with an authorisation token, and the (host) smart card would only migrate this token to the destination smart card or a backup server: for non-restorable, the respective SP will not issue any authorisation token.

3. An SP sends its installed application the authorisation token (if it opts for it) that consists of two sections as shown in figure 9.2. The first section is a public section that is not encrypted and it contains the SP's URL (Universal Resource Locator), authorisation token identifier, and optional section. The URL would instruct a smart card where to establish the connection to download the application. The authorisation token identifier uniquely identifies the token and associated cryptographic keys. The optional segment is made available for the SP/backup-server to include any housekeeping information if necessary. The second section consists of an encrypted message that may contain proprietary information that would ensure that the token is genuine and is generated by the SP. The second section is encrypted by the SP with its token authorisation key and the selection of this key is at the sole discretion of the SP. The contents of this section include an application identifier, a user identifier and a lease identifier. The application identifier refers to the application that was issued to the user indicated by the user identifier. The lease identifier uniquely identifies the smart card to which the application was leased, along with any associated data, including cryptographic keys (if each instance of the application lease has different cryptographic keys). The application will then give the authorisation token to the on-card backup & restoration manager.

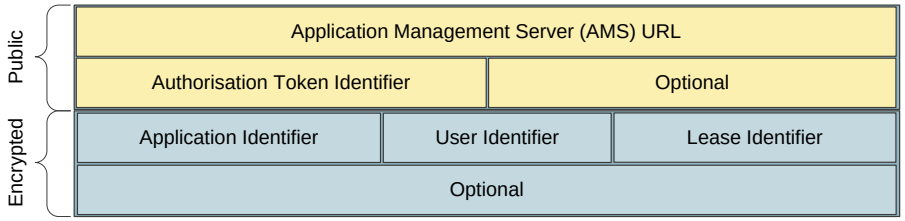

Figure 9.2: Structure of authorisation tokens generated by respective SPs

4. The backup & restoration manager will encrypt the set of authorisation tokens with a package sealing key that is based on some secret that is known to the user. It could be a password, a passphrase or a biometric — something that the user could provide at the time of restoration to prove that she is the genuine user that created the backup package. This key would be generated once, unless the user decided to

change her password or passphrase. The simplest way to generate the package-sealing key is to base it on the user's input. The size of the key and password length is based on the backup server choice; however, we consider that an adequate selection should be made by the backup server to provide a secure service.

At the time of restoration, the user will provide the backup & restoration manager of the new smart card with the credentials for the backup server. The backup & restoration manager and backup server will establish a secure relationship using the proposed STCPs. Subsequently, the backup & restoration manager will download the authorisation tokens from the backup server. These authorisation tokens are sealed by an encryption key based on the user's input. The backup & restoration manager will request the user for the relevant input and decrypt the backup package. After decryption of the package, the backup & restoration manager will retrieve one authorisation token at a time and use its public section to connect with the SP. To establish a secure channel and authenticate the user to the given SP, we modify the STCP_{SP} (discussed in section 6.3). In the fourth message of the STCP_{SP}, we replace the U_{Cre} with the authorisation token issued by the SP.

Before an SP issues a new lease to the user, it terminates the existing lease. This means that although the lost smart card is still usable, a user cannot utilise the downloaded application to access sanctioned services because of the Personal Identification Number (PIN) verification (if implemented) and the SP's back office systems. If an application requires PIN verification before it executes, the usual protection mechanism that disables a smart card (or application) if the user enters the wrong PIN multiple times will suffice. Furthermore, the SP can simply blacklist the application, effectively prohibiting it from accessing the sanctioned services. If the application tries to access these services, the SP can instruct the application to block itself and if possible delete all data related to the particular lease and user. One point to note is that in the UCOM, an SP can only block its application, not the whole of the smart card. Nevertheless, an adversary can still use an application only if it does not require a PIN verification and connection with its SP when it executes.

9.2.2 Migration Mechanism

In the previous section, we discussed the structure of an authorisation token and framework for backup to a remote server (e.g. backup server). In this section, we use the same authorisation tokens but this time for migrating contents from one smart card to another.

Similar to the key migration in the TPM specification [18], two smart cards establish a secure connection with each other and then transfer authorisation tokens. When a user

initiates an application migration process, the TEM of the source smart card establishes a secure channel with the destination smart card using the Platform Binding Protocol discussed in section 7.5. The destination smart card then requests the transfer of the authorisation tokens from the source smart card. The migration process first deletes applications from the source smart card, then transfers the authorisation token to the destination smart card. The applications will be downloaded on the destination smart card in a manner similar to that discussed in the previous section (e.g. application restoration). This process is similar to the TPM key migration, except we use a different protocol to the one specified by the TPM specification [18].

9.2.3 Analysis of the Backup and Migration Mechanism

In the smart card industry, there are not many examples of contents backup or migration mechanisms that we can compare with ours. An example is the backup mechanism for phone-book contacts, but even this mechanism is not like the one discussed in this chapter. The closest we can relate to our proposal to is the TPM key migration architecture [18]. The application migration process is similar to the TPM key migration and the only difference is that instead of migrating keys, we migrate the authorisation tokens to the destination smart cards. In the smart card industry such mechanisms are not required due to the ICOM architecture.

The contents backup mechanism effectively prevents smart card cloning and intellectual property theft. In smart card cloning, a malicious user tries to copy applications from a smart card to another card, without the permission of the respective SPs. To prevent cloning of an application, the relevant SP is given the ability to make its application either restorable or non-restorable. Therefore, the choice of moving the application to a new smart card is not with the user but with the SP. Furthermore, the backup or migration mechanism does not move the application data or/and code. In fact, even when the SP sanctions its application to be restorable, the mechanism still relies on the SP to issue an authorisation token. Without this authorisation token, the application can not be part of the backup or the migration mechanism.

Intellectual property theft refers to the scenario where a malicious user tries to obtain the application code (along with data). To do so, the malicious user has to access the application on a non tamper-resistant device with minimal protection. Such a scenario can arise if we move the entire application (code and data) off-card during the backup or migration mechanisms. Therefore, by using authorisation tokens the backup and migration mechanism effectively prevent intellectual property theft.

In addition, the lease of the application to the destination smart card is at the sole discre-

tion of the SP. Therefore, after evaluating the operational and security capabilities of the destination smart card, the SP can continue and lease its application. Furthermore, the SP could first block the lease of the previous application before leasing to the new smart card. Nevertheless, there are certain concerns in the contents backup mechanism that are related to the key that encrypts/decrypts the backup packages. The framework requires the user to input a secret value that could be a long PIN, password, or passphrase that can be exploited by an adversary. To avoid the use of weak user passwords it is recommended the backup servers should take adequate measures by requiring users to choose strong passwords. Furthermore, before a user can download authorisation tokens from the backup server there should be some offline authorisation (e.g. activation of restoration process on a backup server over the internet or telephone).

The migration mechanism is similar to the backup mechanism, except for one detail. It does not require a backup server, so it avoids the need for user password-based cryptographic keys. We consider that the backup & restoration manager of a given smart card should support both the backup and migration mechanisms.

9.3 Application Deletion

In this section, we discuss the last stage in the lifecycle of an application: deletion of an application.

9.3.1 Existing Framework

In the ICOM, post-issuance application installation or deletion is rare. Nevertheless, application deletion is detailed in all of the major smart card platforms and operating systems.

In Multos, the application deletion process is the same as application loading depicted in figure 5.2. The only differences are that in the deletion process, there is no application load unit generator, and an application provider or a card issuer requests the Multos Certification Authority for the application deletion certificate instead of the application load certificate [97]. The on-card deletion process simply deletes the application data and code. As applications on a Multos card do not have interdependencies, the deletion process does not need to be concerned about the feature interaction problem (section 3.5.3). In the application sharing mechanism of the Multos cards (section 7.2.2), a client application may still make a delegation request. However, because it is just an APDU message, the delegation mechanism can return an error that should be handled by the client application.

9.3 Application Deletion

The application deletion process in the GlobalPlatform card specification can be initiated by any entity that has the privilege to execute a delete command. An application provider or a card issuer (TSM) can issue a delete command that is accompanied by mandatory authorisation parameters to authenticate to the respective smart card(s). The deletion process is handled by the OPEN framework of the GlobalPlatform specification and it performs several checks before proceeding with the deletion of an application. The checks include verifying the deletion request (e.g. delete token [30]), confirming whether the application requested for deletion is referenced by another application, and other optional housekeeping checks. If these verifications fail, the GlobalPlatform specification states that the deletion process should be terminated. We have two concerns with the GlobalPlatform deletion process. Firstly, how it can determine that an application is referenced, which is generally part of the application sharing mechanism. As stated in the GlobalPlatform card specification [30], the specification relies on the underlying platform implementation for the application sharing mechanism. Therefore, this test the deletion process requires the support of the underlying platform's application sharing mechanism (section 7.2). Secondly, if an application is referenced, then why terminate the deletion process? Instead, one could resolve the interdependencies and then proceed with the deletion process. Unfortunately, the GlobalPlatform card specification does not detail the resolution of interdependencies among different applications on a smart card.

The Java Card 2.x and 3.x classic editions have similar schemes, as detailed in the GlobalPlatform card specification. The Java Card specification [28] stipulates that the Java Card Runtime Environment (JCRE) should not attempt to delete an application if it is being referenced from another application. However, the Java Card 3.x connected edition extends the deletion framework and attempts to resolve the interdependencies among different applications. The Java Card 3.x connected edition's application deletion mechanism is based on events and associated listeners. The events mechanism enables an application to register/un-register itself for events generated by other applications, and also enables it to generate similar events. The connected edition defines an event for application deletion as an "application instance deletion request" event (`event:///standard/app/deleted`) [16]. By doing so, a client application can register itself to the deletion events of a server application. Therefore, when the server application is requested to be deleted by an authorised entity, the card manager of a Java Card will instruct the server application regarding the deletion request that in return can signal the deletion event. The client application, on receipt of such event, can perform the tasks needed to remove the dependencies on the server application. The card manager will then proceed with checking whether there are any applications that still have dependencies on the server application. If the dependencies of such applications cannot be removed, the card manager will terminate the deletion process. One thing to note is that it is optional for server and client applications to register, signal and manage any events.

The deletion mechanism for the Java Card 3.x connected edition is a positive step towards provisioning an architecture where application installation and deletion will be more common than before. The deletion mechanism for the UCOM architecture is based on the Java Card 3.x connected edition with compulsory components and includes the provision for a cascade deletion process.

9.3.2 Application Deletion in the UCOM

The deletion process in the UCOM is based on the Java Card 3.x connected edition specification, which is extended by the cascade deletion mechanism. Cascade deletion enables a smart card to proceed with the deletion of any dependent applications if their dependencies cannot be resolved in a satisfactory manner. A smart card can only proceed with cascade deletion if the cardholder explicitly sanctions it. The deletion process for the UCOM is illustrated in figure 9.3 and described below:

In figure 9.3, the hard-bordered rectangles represent operations, the rhombus shapes represent the if-then-else conditional statement that is part of the operation that precedes it. The quadrilaterals with curved vertical sides represent the data structures (e.g. files). The dotted lines represent data read or write operations: if the arrowhead points to the data structure then it is a write operation; otherwise, it is a read operation. During the description of the deletion process, we italicise individual operations (e.g. *operation*) and refer to a data structure with double quotes (e.g. "data structure"). Most of the processes represented in the figure 9.3 are part of application installation & deletion manager illustrated in figure 4.1.

On receipt of the application deletion request from either the user or the SP, the *application deletion* will first check whether the application is installed on the smart card. For illustration, we call the application that is requested to be deleted App_D. If App_D is present on the card, then it will be registered as an installed application in the "registry" maintained by the application installation & deletion manager. In this case, when App_D is present, the request is forwarded to the *application deletion handler*, which will retrieve the "application sharing record" maintained by the smart card firewall and check whether App_D has any dependent applications.

If the application does not have any dependent applications, the *mark application* gathers the application-related information and records it in the file "applications for deletion". Next, it checks that the application is not part of any application-sharing tree — meaning there are no application dependencies to resolve. In this scenario, it might seem a redundant check but this step will become necessary when App_D has dependences. In the next step, the user is notified that the smart card is ready to delete the application App_D. If the user

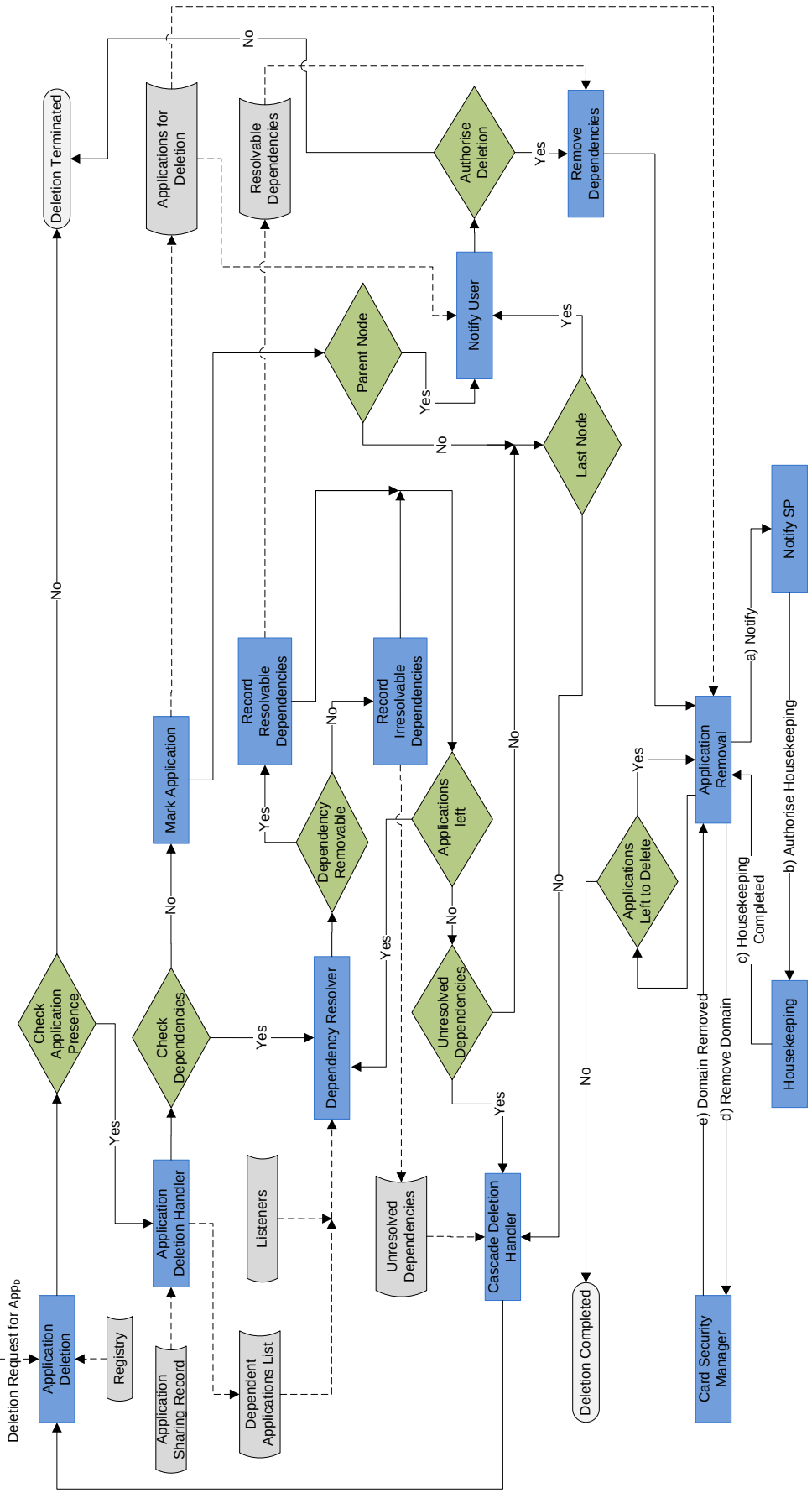

Figure 9.3: Application deletion process in the UCOM

authorises it, first the *remove dependencies* removes any dependencies, which is followed by the *application removal*. The *application removal* will first notify the SP (*notify SP*), which might perform some *housekeeping* tasks like depersonalisation of the application or/and transfer of any log files. Depersonalisation of the application involves removal of all user-related data along with any cryptographic material. Furthermore, transfer of the application log files [231] to the SP that might contain the usage information of the application, which might be necessary for fraud prevention or detection along with evidence of certain activities. After the housekeeping is completed, the *application removal* requests the *card security manager* (section 4.2.2) to delete the domain credentials and reclaim the memory. After a successful outcome, the *application removal* checks whether there are any more applications to delete. If not, then the deletion process will terminate.

In second case, if the *application deletion handler* finds dependencies, then it will generate the "dependent application list". The *dependency resolver* will take the list of applications that are dependent and also ones that registered themselves as listeners to the deletion request for App_D. The *dependency resolver* generates the deletion event for the App_D and notifies all applications that are registered to this event. If the applications can gracefully resolve the dependency then it will record them in the "resolvable dependency". For this step, we rely on the dependent application's response, which might be malicious. If a dependent application App_C might signals that it can remove the dependency, but does not take any action regarding this, its aim might be to use the reference to the App_D for some malicious purpose. However, we protect the platform from such eventualities: the firewall mechanism will also remove the record that the App_C is authorised to access the App_D reference (a memory reference to App_D's application resource manager: figure 7.3). The firewall mechanism can effectively prevent the memory access; however, the main aim of dependency resolution is to avoid any eventualities in which a dependent application might not be able to execute reliably in the future.

The *dependency resolver* will keep on iterating through the "dependent applications list" until it reaches the end of it. It will then check whether there are any unresolved dependencies. If yes, then it moves to the *cascade deletion handler*; otherwise, it will check whether it is at the last application in the "applications for deletion". The *cascade deletion handler* takes the list of unresolved dependencies list and iterates through it, signalling the deletion of the respective applications. For each application, the dependency analysis is performed. This process is iterated until the list of all applications required to be deleted is compiled: "applications for deletion". At this stage, the user is notified by the *notify user* and the "applications for deletion" is communicated. If the user authorises the card to go ahead with the deletion of the applications, the *remove dependencies* will first remove any dependencies. A point to note is that we leave the dependence removal process to the end: if the respective user does not authorise the deletion, at least we will not delete any application sharing instances. Therefore, before the user authorises the deletion, the entire

process only tries to find dependent applications and point out to the user the list of applications that cannot resolve their dependencies on App_D. The *application removal* process will then iterate through the "applications for deletion" and delete them one at a time.

In cases where the deletion request was initiated by the SP of the App_D, and it requires deletion of other applications that do not belong to the SP, the user will still be notified. If the user opts for not deleting it, the SP can then proceed with blocking the App_D. In the block state, an application is not accessible to the user; however, dependent applications can still access it through the application sharing mechanism.

As discussed before, the UCOM deletion process only provides dependent applications with an opportunity to gracefully resolve their dependencies. If an application does not have such a mechanism, the UCOM deletion process marks that application for deletion. Furthermore, during the deletion process the App_D's resource manager that maintains the access to the application via the smart card firewall is removed. Thus, if a dependent application tries to access App_D resources, the firewall mechanism will reject that request. If the application does not gracefully proceed after the firewall rejects its request, the card security manager can either block the application or mark it for deletion. Therefore, any application that affects the reliability of the smart card platform will be removed or at least blocked by the card security manager.

9.4 Decommissioning Process

The decommissioning process in the UCOM involves deletion of all applications from a UCTD and removal of any user-specific data stored by the respective TEM or UCTD platform managers (section 4.2). The decommissioning process is initiated by the user in a manner similar to the ownership acquisition process (section 4.6.3). However, in the decommissioning process the user requests a UCTD to delete all applications in a manner similar to the one discussed in the previous section but this time the UCTD does not check for dependencies. Once all applications are deleted, the card security manager will delete the user-specific cryptographic keys (e.g. user signature key) and associated certificates. It will then request the deletion of ownership credentials that the user has set during the ownership acquisition process. After the decommissioning process is completed, the UCTD reverts to the state it was in when the user acquired it from the card manufacturer (or UCTD suppliers). In other words, it is a blank UCTD.

9.5 Summary

In this chapter, we began by describing the smart card contents backup and migration mechanisms. Both of these mechanisms aim to provide a dynamic architecture to recover from the loss of a UCTD or migration to a new UCTD. These mechanisms are in line with the original statement of purpose for the UCOM environment. In subsequent sections, we discussed the application deletion process in the existing smart card platforms, and how they relate to the proposed deletion mechanism for the UCTD. Discussion on the application deletion and decommissioning completes the lifecycle of both an application and a UCTD.

Chapter 10

Conclusions and Future Research Directions

Contents

In this chapter we conclude the thesis by summarising our contributions and discussing some of the future challenges that need to be addressed.

10.1 Summary and Conclusions

The main goal of this thesis was to explore the viability of user ownership for a security sensitive device whose architecture is based on smart card technology. The introduction of this user ownership affects all stages of the smart card and application lifecycle, which we analysed during the course of this thesis.

We began the discussion by mapping the security and privacy landscape from three different computing fields: smart cards, mobiles and traditional computing environments. These computing devices are used by individual users with an ever growing reliance on them, so there needs to be a unified security and privacy-preserving architecture that can be easily integrated to any of these computing devices. We consider that the User Centric Tamper-Resistant Device (UCTD) has the potential to deliver such a unified (services) architecture. We provided the rationale for the UCTD framework. To explain how we selected an appropriate base architecture for the UCTDs, we provided a comparison between different proposals that included TPM, AEGIS, ARM TrustZone, M-Shield, and GlobalPlatform's TEE and the smart card architecture. This comparison gave us a clear indication that the smart card architecture is the one most suited to be a UCTD that supports unified security, trust, and privacy architecture for different computing devices. However, the issue with smart card technology is its ownership architecture that is stringently under a centralised authority. A possible solution is to delegate smart card ownership from a centralised authority to its users.

Before we delved into the core of the thesis, we provided a detailed coverage of different ownership models that exist in the smart card ecosystem. We began with the centralised control of smart cards provided by the Issuer Centric Smart Card Ownership Model (ICOM), and discussed its advantages and drawbacks. We then briefly examined different proposals that support the ICOM framework, including Java Card, Multos and GlobalPlatform. We referred to these prominent ICOM frameworks throughout the thesis, comparing and contrasting our proposal with them. This short introduction to the ICOM frameworks was provided to set the scenery and to help the reader understand the present characteristics of different frameworks that support the ICOM.

Subsequently, we discussed the frameworks in the smart card industry that come close to providing the user ownership. Unfortunately, the concept of ownership as described in our proposal of the User Centric Smart Card Ownership Model (UCOM) is not close to any of the existing proposals. The concept of ownership in UCOM has to do with *freedom of choice* and not complete control of the smart card device as the card issuers have in the ICOM. Therefore, the concept of *freedom of choice* can be considered a novel idea in the context of the smart card technology. We identified different stakeholders and their security and operational requirements. This discussion served as an introduction to the

concept of the UCOM that enables a smart card to become a UCTD. The structure of the rest of the thesis was closely aligned with the lifecycle stages of a smart card, including manufacture, downloading applications, application execution, and finally the deletion of the applications and decommissioning of the smart card.

As previous frameworks including GlobalPlatform, Java Card, and Multos mainly support the ICOM initiative; therefore, we first analysed whether these can support the user ownership proposal. During the design of the UCOM-based smart card architecture, the strategy that we opted for was to adopt, modify, and introduce new components where required to existing ICOM-based smart card architectures in a way that supports the proposed user ownership. This idea became the root of all of our proposals in this thesis.

We defined a short list of services, based on the GlobalPlatform architecture, which support the user ownership, smart card, and application management operations. A major issue introduced by the UCOM was decentralisation of the trust architecture that has been deployed in the smart card industry. Traditionally, in the ICOM framework, the trust resided in the card issuer and an application provider was only required to trust the card issuer, and vice versa. Whereas, by giving the ownership of the smart cards to their users we removed the card issuers altogether, leaving a vacuum in the traditional trust architecture. We replaced the traditional trust architecture that relied on the card issuer, and moved it to the smart card itself.

We proposed a security assurance and validation mechanism based on third party independent security evaluation and a platform-independent trustworthy component on a smart card. Both of these proposals enable a remote application provider (that we refer to as Service Provider (SP) in the UCOM) to ascertain the security assurance of a smart card. The platform-independent trustworthy component on a smart card is referred to as the Trusted Environment & Execution Manager (TEM), which is similar to the TPM. The TEM provides an attestation mechanism that certifies that the state of the smart card is as it was at the time of evaluation (i.e. in a trustworthy state). To do so, we proposed two attestation mechanisms termed as online and offline attestation mechanisms. To support each type of the attestation mechanism we also proposed two self-test mechanisms based on the Pseudorandom Number Generators (PRNGs) and Physical Unclonable functions (PUFs). Furthermore, as the name suggests the online attestation mechanism requires an entity to vouch for the trustworthiness of a smart card. In our proposal, it is the card's manufacturer. Therefore, to support the online attestation mechanism we proposed a protocol that we referred to as the attestation protocol.

Once a smart card is manufactured, evaluated, and acquired by a user, the framework that comes next is the smart card management architecture. The management architecture is responsible for establishing a relationship with SPs and acquiring their applications by

giving the user authentication credentials for these SPs. We discuss the card management architectures proposed by GlobalPlatform, and Multos. The rationale behind not discussing the Java Card was to do with its support for the GlobalPlatform. We briefly described the shortfalls of both GlobalPlatform and Multos card management architectures. Subsequently, we modified the architecture specified by GlobalPlatform in a way that supported the application installation mechanism of the UCOM. Later, we also proposed the possible attacks that are unique to the UCOM proposal along with how a smart card can adequately implement protection against them.

Based on the card management architecture, we proceeded with the application installation process. The installation process first requires a secure channel to be established between a smart card and an SP. It also requires that an SP is able to ascertain the trustworthiness of the smart card — to enable the SP to verify whether the given smart card supports the SP's security policy for the application lease. For this purpose, we defined the security and operational requirements for a Secure and Trusted Channel Protocol (STCP) for UCOM-based smart cards. We proposed three protocols that satisfy the UCOM requirements and these protocols were subjected to the CasperFDR tool for a mechanical formal analysis. We performed the mechanical formal analysis on the STCPs for the sake of completeness. In addition to this, we provided performance results of test implementations and compared them with existing protocols. Our proposed STCPs not only satisfied the security and operational requirements of the UCOM but also provided an efficient performance. After establishing a secure and trusted channel protocol, an SP may proceed with the application download to the requesting smart card.

A downloaded application on a smart card may establish data and resource sharing with other applications. Both Java Card and Multos support the application sharing mechanism; however, their proposals take two opposite approaches. We discussed both approaches and detailed the reasons why they fail the UCOM's requirements. Subsequently, we proposed a smart card firewall mechanism based on the Java Card application sharing mechanism that supports the UCOM's requirements. To support this proposal, a dynamic mechanism is needed that not only authenticates the applications but also ascertains whether the current states of the applications are secure. For this purpose, we proposed a symmetric key-based protocol that a client and server application can use to authenticate and validate each other's state. Later, we extended the application sharing mechanism that traditionally only supports sharing between the applications on a single smart card, to one that allows applications installed on different smart cards to share their data and resources. We termed this extension as Cross-Device Application Sharing and to support this proposal we detailed two protocols that establish relationships between individual smart cards and applications. All proposed protocols were subjected to mechanical formal analysis by CasperFDR and their test performance measures were provided along with comparisons with other protocols. Once an application is installed and it has established any sharing

with other applications (if required), the next lifecycle stage is the application execution.

The smart card runtime environment provides a secure and reliable platform for the executing applications. From an adversary's point of view, attacking an application while it is executing might yield desirable results, such as skipping any security checks (e.g. PIN verification). To achieve this, the runtime environment is subjected to different types of attacks, including fault injections. Most of the attacks proposed in the literature target an open smart card on which an adversary can install his application, which is difficult in the traditional ICOM framework. However, the open and dynamic nature of the UCOM allows such a facility. This opens up the UCOM proposal to attacks that are specifically designed to target the reliable and safe execution of an application. We described the architecture of the Java Card runtime environment, which was followed by a discussion on how an adversary can affect the execution of an application. To harden the runtime environment from adversarial perturbations, we proposed protection mechanisms. We proposed that the TEM should take a vital role in providing dynamic protection by getting involved during the execution of an application. These mechanisms were then analysed for their latency and performance measurements. This discussion can be considered as a survey of the vulnerabilities of the smart card runtime environment that will have an adverse effect on the UCOM proposal, along with the limitations of the proposed protection mechanisms. We articulated that any protection mechanism built on top of the runtime environment would inadvertently introduce a performance penalty. We concluded from this survey that at the time when the Java Card virtual machine was designed, the focus was on reliability and performance. Their design did not take into account the fault injection and combined attacks. We recommend that a bottom-up approach should be taken — rather than putting extra layers on top of the runtime environment, redesigning it might be a better option.

Finally, we discussed the last lifecycle state of a smart card and an application in the UCOM framework. A user might lose and want to recover contents onto her new smart card, or want to upgrade to a feature-rich smart card. We proposed contents backup and migration mechanisms that support both of these scenarios without compromising the SP's security requirements. Furthermore, we detailed the application deletion mechanism deployed in both Java Card and Multos, illustrating that they may both be unsuitable for the UCOM architecture. The main issue was that if application dependencies are not resolvable, the application will not be deleted. To mitigate this, we proposed a cascade deletion process that supports the deletion of dependent application, if authorised by the user. This discussion completed the lifecycle for both a smart card and an application, supporting a dynamic and open framework that allows a user to have the choice to install or delete applications from their smart cards.

10.2 Recommendations for Future Work

Our intention in this work was to analyse the feasibility of giving the ownership of a security- and reliability-critical device like a smart card to its user. We aimed to explore the possibilities that it would bring, and new application scenarios that might open up for the smart card-based services deployment. We have achieved our goals by providing a pathway for UCTDs and user centric smart cards, which not only provides security and reliability assurance to SPs but also give users "freedom of choice". However, we consider that there is a long journey ahead for the user centric smart card proposal and there are many suggestions for possible improvements and directions for future research.

There can be possible improvements in the hardware protection and remote attestation mechanism for the UCOM framework. An attestation mechanism that not only provides the assurance that the current state of the smart card is secure as stated by the appropriate evaluation authority, but also the uses hardware that will simplify the remote assurance mechanism. Furthermore, we need to provide security and reliability characterisation, classification, and formalisation of smart card / application services. We might employ mechanisms similar to those implemented in service-oriented computing architecture, taking smart cards and applications as two services that need to ascertain whether they can support each other's requirements. This work may lead to devising a language (semantics) to describe the above mentioned features as is done in Web Service Description Language (WSDL). Such a language can be used to create third party evaluation certificates, which in our proposal is the CC authority.

An application tagging mechanism tags segments of an application with security and/or reliability levels, which instruct the runtime environment to apply adequate checks during the execution of the application. To support the application tagging mechanism in the UCOM, we need to have an on-card mechanism that can verify the security and reliability tags. Therefore, an adversary cannot take advantage of such a framework to subvert a smart card's runtime protection. We refer to on-card analysis as on-card application behavioural analysis, which is similar to bytecode analysis but is focused on the nature of an application segment and its associated tag.

One of the major future research directions is the smart card runtime environment, and its security and reliability in the presence of malicious applications, fault, and combined attacks. As discussed above, we need to look into the design of the virtual machine and build the protection from there, rather than implementing them in a piecemeal manner. This requires the study of existing virtual machines and language architectures, to find out a balance between performance, and runtime-protection. This work may reduce the number of opcodes assigned in the Java virtual machine, and/or redefining the execution structure.

Finally, we should be able to gain assurance of feature independence, during the application deletion process. A framework should be designed such that dependency resolutions should be made independent of an application rather than as a voluntary feature of the dependent application.

Appendix A

Description of Protocols Used for Comparison

Contents

In this appendix, we discuss the selected protocols that are used for comparison with the our proposed protocols in this thesis. The protocols are compared on the basis of a pre-defined set of security and operation goals for the UCTD environment. The selection of the protocols was intentionally kept broad to include well-known (studied) Internet protocols, along with protocols designed for mobile and smart card environments. This selection provides a well-balanced comparison with the proposed protocols in terms of pre-defined goals.

A.1 Protocol Notation and Terminology

The notation used to describe protocols in this appendix is as below.

Table A.1: Protocol notation and terminology

Notation	Description
$\mathcal{SC}$	Denotes a smart card (in context of this thesis).
$\mathcal{TTP}$	Denotes the trusted third party.
$\mathcal{SP}$	Denotes an SP (in context of this thesis).
X_i	Indicates the identity of an entity X.
N_X	a random number generated by entity X.
g^X	Diffie-Hellman exponential generated by an entity X.
$h(Z)$	The result of applying a hash algorithm (e.g. SHA-256) on data Z.
k_{X-Y}	Encryption key shared between entities X and Y.
mk_{X-Y}	MAC key for symmetric algorithms shared between entities X and Y.
B_X	Private decryption key associated with an entity X.
V_X	Public encryption key associated with an entity X.
$f_K(Z)$	Result of applying MAC algorithm on data Z with key K.
$zK_X(Z)$	Result of encrypting data Z using public key algorithm (e.g. RSA) with key K_X.
$e_K(Z)$	Result of encrypting data Z using symmetric key algorithm (e.g. AES) with key K.
$Sign_X(Z)$	Is the signature on data Z with the signature key belonging to the entity X using a signature algorithm like DSA or based on the RSA function.
$CertS_X$	Is the certificate for the signature key belonging to the entity X.
$CertE_X$	Certificate for the public key belonging to the entity X.
$X \rightarrow Y : C$	Entity X sends a message to entity Y with contents C.
$X\|\|Y$	Represents the concatenation of data items X and Y.

A.2 Station-to-Station (STS) Protocol

The STS protocol provides a three-pass mutual entity authentication and mutual explicit key authentication to two communicating parties [174]. The protocol described in this section is from the Meneze et al. [146], which includes an encrypted certificate from the smart card to provide privacy preservation.

STS-1. $\mathcal{SC} \rightarrow \mathcal{SP}$: g^{SC}

$\mathcal{SP}$: $k_{SC-SP} = (g^{SC})^{SP}$

The smart card ($\mathcal{SC}$) initiates the STS protocol by generating a Diffie-Hellman exponential and communicating it to the server ($\mathcal{SP}$). The $\mathcal{SP}$ will generate a shared secret by k_{SC-SP} from the shared public key of the $\mathcal{SC}$ (i.e. g^{SC}) with the private key of the $\mathcal{SP}$ (i.e. $N_S P$).

STS-2. $\mathcal{SP} \rightarrow \mathcal{SC}$: $g^{SP}||e_{k_{SC-SP}}(Sign_{SP}(g^{SP}||g^{SC}))||CertS_{SP}$

$\mathcal{SC}$: $k_{SC-SP} = (g^{SP})^{SC}$

In response, the $\mathcal{SP}$ generates a public key (e.g. g^{SP}) along with encrypting the signature

on the public keys generated by both communicating entities. The public key, encrypted signature and the certificate for the $\mathcal{SP}$ is sent to the $\mathcal{SC}$. The certificate is sent in plaintext; therefore, in this protocol there is no privacy protection for the $\mathcal{SP}$ (i.e. which is not necessary to have as most of the servers have public addresses: Internet addresses).

The $\mathcal{SC}$ will generate the shared secret key similar to the $\mathcal{SP}$ (in previous message). The $\mathcal{SC}$ will decrypt the signature and then verify the signature on the public keys generated by the both $\mathcal{SP}$ and $\mathcal{SC}$. This is to avoid man-in-the-middle attack.

STS-3. $\mathcal{SC} \rightarrow \mathcal{SP} \quad : \quad e_{k_{SC-SP}}(Sign_{SC}(g^{SP}||g^{SC})||CertS_{SC})$

The $\mathcal{SC}$ will sign the public keys generated by the $\mathcal{SC}$ and $\mathcal{SP}$ then append the certificate. The entire message is then encrypted by the shared secret key k_{SC-SP}. This message provides mutual entity authentication, and mutual explicit key authentication along with preventing the man-in-the-middle attack.

A.3 Aziz-Diffie (AD) Protocol

The AD protocol was proposed for the wireless local area networks [175] and unlike STS it does not rely on the Diffie-Hellman exponentials to generate the shared secretes.

AD-1. $\mathcal{SC} \rightarrow \mathcal{SP} \quad : \quad CertE_{SC}||CertS_{SC}||N_{SC}$

The AD protocol is started by the $\mathcal{SC}$ that generates a random number N_{SC}, append it with the $\mathcal{SC}$ encryption key pair certificate.

AD-2. $\mathcal{SP} \rightarrow \mathcal{SC} \quad : \quad zV_{SC}(N_{SP})||CertE_{SP}||CertS_{SP}||Sign_{SP}(zV_{SC}(N_{SP})||N_{SC})$

On receiving the first message, the $\mathcal{SP}$ will generate a random number N_{SP} and encrypt it with the $\mathcal{SC}$'s public key. It then appends the signature key pair certificate along with a signed message that includes the encrypted random number of the $\mathcal{SP}$ along with the random number sent by the $\mathcal{SC}$.

AD-3. $\mathcal{SC} \rightarrow \mathcal{SP} \quad : \quad zV_{SP}(r'_{SC})||Sign_{SC}(zV_{SP}(r'_{SC})||zV_{SC}(N_{SP}))$

$\mathcal{SC}, \mathcal{SP} \quad : \quad k_{SC-SP} = r'_{SC} + N_{SP}$

The $\mathcal{SC}$, on receiving the second message will first decrypt the $\mathcal{SP}$'s random number and then verifies the signature. Subsequently, the $\mathcal{SC}$ generates another random number r'_{SC}. Now the $\mathcal{SC}$ can now generate the shared key k_{SC-SP} by adding the r'_{SC} with the $\mathcal{SP}$'s random number.

The $\mathcal{SC}$ will encrypt the r'_{SC} with the public key of the $\mathcal{SP}$ and generate a signature on the encrypted random numbers from $\mathcal{SC}$ and $\mathcal{SP}$. On receipt, the $\mathcal{SP}$ can also generate the shared secret k_{SC-SP} as the $\mathcal{SC}$ has generated it.

A.4 ASPeCT Protocol

The ASPeCT protocol is designed as part of the European Commission ACTS project ASPeCT [232], which focuses on the mobile network environment for value-added transactions. Earlier versions of the ASPeCT protocol is proposed in Martin et al. [176], and Horn and Preneel [177]. However, in this section we describe the protocol as it is detailed in the Horn et al. [168]. Additional notations required for the description of the ASPeCT are as below:

ID_T	identifier of the smart card's certification authority.
$CertSt$	certified (static) public key agreement key (g^{SP}) of the $\mathcal{SP}$.
$h1, h2, h3$	one-way hash functions, that are detailed in Horn and Preneel [177].
cd	details of the charging data.
TS	time stamp.
py	payment confirmation.

ASPeCT-1. $\mathcal{SC} \rightarrow \mathcal{SP}$: $g^{SC}||ID_T$

$\mathcal{SP}$: $k_{SC-SP} = h1(N_{SP}||(g^{SC})^{SP})$

The $\mathcal{SC}$ generates a Diffie-Hellman exponential g^{SC} and append it with the identity of the smart card's certification authority (ID_T). On receipt of this message, the $\mathcal{SP}$ generates the shared secret key by using the g^{SC}, public key agreement key g^{SP} along with a random number generated by the $\mathcal{SP}$.

ASPeCT-2. $\mathcal{SP} \rightarrow \mathcal{SC}$: $N_{SP}||h2(k_{SC-SP}||N_{SP}||S_i)||CertSt$

$\mathcal{SC}$: $k_{SC-SP} = h1(N_{SP}||(g^{SP})^{SC})$

$\mathcal{SC}$: $H = h3(g^{SC}||g^{SP}||N_{SP}||ID_{SP}||cd||TS||py)$

In response, the $\mathcal{SP}$ adds the random number N_{SP} appended with the hash generated by the function $h2$ on the generated key (for key authentication), N_{SP}, and identity of the $\mathcal{SP}$. Finally, appending the certificate of the public key agreement key (e.g. g^{SP}).

On reception of the second message, the $\mathcal{SC}$ retrieves the public key agreement key (g^{SP}) and then follow the similar steps like $\mathcal{SP}$ to generate the shared secret key. After generating the k_{SC-SP}, the $\mathcal{SC}$ will authenticate the shared key by generating the hash with function $h2$ and match with the one received in the message 2. If it matches then $\mathcal{SC}$ got the key authentication from the $\mathcal{SP}$.

The $\mathcal{SC}$ will then generate the transaction that includes the several elements from the first two message along with the charging details associated with the particular $\mathcal{SC}$, which contains the time stamp and payment details (py). All these elements are then hashed using the function $h3$ and the output is referred as H.

ASPeCT-3. $\mathcal{SC} \rightarrow \mathcal{SP}$: $e_{k_{SC-SP}}(Sign_{SC}(H)||CertS_{SC}, py)$

In response, the $\mathcal{SC}$ will sign the H (that is used as non-repudiation of the transaction). It then appends the signature key certificate for the $\mathcal{SC}$ and payment details. The entire message is then encrypted by the shared secret key.

The $\mathcal{SP}$ will decrypt the message and a successful decryption provides the key authentication. It then verifies the signature and process the transaction.

A.5 Just-Fast-Keying (JFK) Protocol

Aiello et al. [178] proposed two variants of JFK protocol, with difference based on who initiate the protocol. In this thesis, we refer to JFKi that provides identity protection for initiator (e.g. smart card) even against active attacks. In the JFKi, the smart card initiates the session that is described below:

JFKi-1. $\mathcal{SC} \rightarrow \mathcal{SP} \quad : \quad h(N_{SC})||g^{SC}||ID_{S'}$

The initiator ($\mathcal{SC}$) generates a random number (N_{SC}) and sends its hash along with Diffie-Hellman exponential (g^{SC}) appended with requirement of the $\mathcal{SC}$ about authentication information that the $\mathcal{SP}$ should use in subsequently messages. The requirement of the $\mathcal{SC}$ is indicated by $ID_{S'}$

JFKi-2.
$$\begin{aligned} \mathcal{SP} &: S_{SP} = Sign_{SP}(g^{SP}||grpinfo_R) \\ \mathcal{SP} &: SID = f_{mk_{SP}}(g^{SP}||N_{SP}||h(N_{SC})||IP_{SC}) \\ \mathcal{SP} \rightarrow \mathcal{SC} &: h(N_{SC})||N_{SP}||g^{SP}||grpinfo_R||ID_{SP}||S_{SP}||SID \end{aligned}$$

In response, the $\mathcal{SP}$ also generates a random number and Diffie-Hellman exponentials. The $\mathcal{SP}$ then sends the $h(N_{SC})$ along with the $grpinfo_R$ and ID_{SP}. The $grpinfo_R$ indicates to the $\mathcal{SC}$ the set of Diffie-Hellman groups supported by the $\mathcal{SP}$. The ID_{SP} provides the authentication information of $\mathcal{SP}$ that was request by the $\mathcal{SC}$ in message one. Furthermore, the $\mathcal{SP}$ generates a signature on the generated g^{SP} and $grpinfo_R$, and finally append the session identifier (SID) to safeguard against possible DoS attacks.

JFKi-3.
$$\begin{aligned} \mathcal{SC} &: K = (g^{SP})^{SC} \\ \mathcal{SC} &: k_{U_S} = f_K(h(N_{SC})||N_{SP}||''1'') \\ \mathcal{SC} &: mk_{U_S} = f_K(h(N_{SC})||N_{SP}||''2'') \\ \mathcal{SC} &: mE = e_{k_{U_S}}(U_i||Sign_{SC}(h(N_{SC})||N_R||g^{SC}||g^{SP}||S_i)) \\ \mathcal{SC} \rightarrow \mathcal{SP} &: N_{SC}||N_{SP}||g^{SC}||g^{SP}||mE||f_{mk_{SC-SP}}(mE)||SID \end{aligned}$$

The $\mathcal{SC}$ generates the session encryption and MAC keys from the shared secret (K). The $\mathcal{SC}$ then generates a message including identities of communicating entities, random numbers generated during the session, and Diffie-Hellman exponentials. The $\mathcal{SC}$ then signs this message and later encrypts it. The encrypted message is then MACed and sent to the $\mathcal{SP}$.

JFKi-4.
$$\begin{aligned} \mathcal{SP} &: mE = e_{k_{U_S}}(Sign_{SP}(h(N_{SC})||N_R||g^{SC}||g^{SP}||U_i)) \\ \mathcal{SP} \rightarrow \mathcal{SC} &: mE||f_{mk_{SC-SP}}(mE) \end{aligned}$$

In response the $\mathcal{SP}$ generates a signature that includes random numbers, Diffie-Hellman exponentials and identity of the $\mathcal{SC}$. The signed message is then encrypted and MACed before sending it to the $\mathcal{SC}$.

A.6 Trusted Transport Layer Protocol (T2LS) Protocol

In this thesis, the T2LS protocol described by Gasmi et al. [165] is used that is described in this section. The messages listed below are on top of the existing TLS protocol that we do not detail in this section.

T2LS-1. $\mathcal{SC} \rightarrow \mathcal{SP} \quad : \quad N_{SC}||CertS_{SC_{bind}}||CertS_{SC_{AIK}}$

The $\mathcal{SC}$ initiates the protocol by sending a random number (N_{SC}) along with associated TPM's certificates (e.g. $CertS_{SC_{bind}}$ and $CertS_{SC_{AIK}}$). The TPM is part of the computing platform that the $\mathcal{SC}$ is using to connect to the $\mathcal{SP}$. This message initiates the TLS protocol, the $\mathcal{SC}$ appends the `ClientHello[ciphersuites,hell_ext_list,nonce]`

In response, the $\mathcal{SP}$ sends the `ServerHello[ciphersuites,hell_ext_list,nonce]`, followed by key exchange message and completion of security parameter negotiations between the $\mathcal{SP}$ and $\mathcal{SC}$.

T2LS-2. $\mathcal{SC} \rightarrow \mathcal{SP} \quad : \quad zV_{SP}(SessionKey_{SC}||CDS_{SC}||N_{SP})$

The $\mathcal{SC}$ will verify the $\mathcal{SP}$'s TPM certificates (e.g. $CertS_{SP_{bind}}$ and $CertS_{SP_{AIK}}$). The $\mathcal{SC}$ encrypts its generated session key along with configuration of TPM and TLS protocol in CDS_{SC}. The public encryption key of the $\mathcal{SP}$ is used to encrypt the message.

The $\mathcal{SP}$ decrypts the message and validates the CDS_{SC} and the received random number

T2LS-3. $\mathcal{SP} \rightarrow \mathcal{SC} \quad : \quad zV_{SC}(SessionKey_{SP}||CDS_{SP}||N_{SC})$

The $\mathcal{SP}$ generates an attestation blob similar to the one generated by the $\mathcal{SC}$ in previous message. On receipt of message three, the $\mathcal{SC}$ will verify the CDS_{SP} and N_{SC}.

The trust in the T2LS comes from the values of CDS and verification of the CDS values by the communicating entities. If the CDS value of a client is satisfactory to the server, then it can trust the state of the client, and vice versa.

Key Generation.

$$\mathcal{SC}\ \&\ \mathcal{SP} \quad : \quad ms = PRF(N_{SC}||N_{SP}||SessionKey_{SC}||SessionKey_{SP})$$
$$\mathcal{SC}\ \&\ \mathcal{SP} \quad : \quad k_{SC-SP} = PRF(ms||CDS_{SC}||SDS_{SP})$$

On receipt of message three, both the $\mathcal{SC}$ and $\mathcal{SP}$ will proceed with generating the session key k_{SC-SP}. The notation of PRF listed above refers to the pseudorandom number generator used by the $\mathcal{SC}$ and $\mathcal{SP}$ to generate the keys.

A.7 Secure Channel Protocol - 81 (SCP81) Protocol

The GlobalPlatform specification for the SCP81 [169] do not change the message structure of the TLS protocol [100]. They provide a structure of how a smart card and a remote administrator authority can use the TLS protocol for remote management of the smart card contents. In this section, we suffice by describing the TLS protocol, which is also useful to the T2LS protocol as the messages discussed in section A.6 are ones that modify

the traditional TLS protocol.

SCP81-1. $\mathcal{SP} \rightarrow \mathcal{SC} \quad : \quad SP_i||N_{SP}||SID||P_{SP}||CertE_{SP \leftarrow TTP}$

The first two messages are referred as the protocol handshake. The $\mathcal{SP}$ initiates the protocol and generates a random number (N_{SP}), append it with the $\mathcal{SP}$ identity. The $\mathcal{SP}$ also includes the session identifier (SID) and preferred parameters for the TLS in P_{SP}. The SID is an arbitrary byte sequence chosen by the $\mathcal{SP}$.

SCP81-2. $\mathcal{SC} \rightarrow \mathcal{SP} \quad : \quad N_{SC}||SID||P_{SC}||CertE_{SC \leftarrow TTP}$

In response, the $\mathcal{SC}$ sends a random number and P_{SC}. On receipt of this message, the $\mathcal{SP}$ generates a premaster-secret that can be Diffie-Hellman exponentials. For the description of the TLS in this thesis, we use the Diffie-Hellman scheme as the session key generation in the TLS.

SCP81-3.
$$\begin{aligned} \mathcal{SP} &: S_{SP} = Sign_{SP}(SP_i||g^{SP}||N'_{SP}||N_{SC}) \\ \mathcal{SP} &: E_{SP} = zV_{SP}(g^{SP}||N'_{SP}) \\ \mathcal{SP} \rightarrow \mathcal{SC} &: E_{SP}||S_{SP}||CertS_{SP} \end{aligned}$$

The $\mathcal{SP}$ generates a Diffie-Hellman exponential, a new random number and append it with a signed message (S_{SP}). The S_{SP} is used to authenticate the $\mathcal{SP}$ to the $\mathcal{SC}$. The $\mathcal{SP}$ also includes the signature key and encryption key certificates, which are verified by the $\mathcal{SC}$ on receipt of message three.

SCP81-4.
$$\begin{aligned} \mathcal{SC} &: S_{SC} = Sign_{SC}(g^{SC}||N'_{SC}||N'_{SP}) \\ \mathcal{SC} &: E_{SC} = zV_{SC}(g^{SC}||N'_{SC}) \\ \mathcal{SC} \rightarrow \mathcal{SP} &: E_{SC}||S_{SC}||CertS_{SC} \end{aligned}$$

The $\mathcal{SC}$ will perform same operations as the $\mathcal{SP}$ has performed in message three. The signed message from the $\mathcal{SC}$ authenticates it to the $\mathcal{SP}$. After message four, both $\mathcal{SC}$ and $\mathcal{SP}$ can generate the master-secret (K_{SC-SP}) from the generated random numbers and premaster-secrets by using a pseudorandom number generator (PRF). Subsequently, session keys and MAC key are generated from the master-secret. Both $\mathcal{SC}$ and $\mathcal{SP}$ use separate keys of encrypting data between them; meaning $\mathcal{SC}$ uses one key to send a message to $\mathcal{SP}$, where in response $\mathcal{SP}$ uses a different key.

Before proceeding with communications between the $\mathcal{SC}$ and $\mathcal{SP}$ for the purpose the TLS session was establish, both entities will first send "finished" message. The "finished" message confirms all the details of agreed during the handshake and verify whether they are being changed.

A.8 Markantonakis-Mayes (MM) Protocol

The MM protocol is based on the GlobalPlatform SCP10 [30]. Before we describe the protocol, we introduces few new notations

$CertS$	certified (static) public key agreement key (g^{SP}) of the $\mathcal{SP}$.
$CertU$	certified (static) public key agreement key (g^{SC}) of the $\mathcal{SC}$.
$\{x, y, z\}$	implies that items in the curly brackets represents an optional message.

MM-1. $\mathcal{SP} \rightarrow \mathcal{SC}$: $CertS||N_{SP}||\{SP_i||Reqt_{DC}(SC)||Reqt_{PC}(SC)||CertE_{SP \leftarrow TTP}\}$

The $\mathcal{SP}$ will send a certified Diffie-Hellman public key agreement key (g^{SP}) along with a random number. As optional part of the message, the $\mathcal{SP}$ can send $\mathcal{SP}$'s identity, and cryptographic certificate for $\mathcal{SP}$'s public encryption key. Furthermore, the optional message also contains requests for smart card's Diffie-Hellman certificate ($Reqt_{DC}(U)$) and public encryption key ($Reqt_{PC}(U)$).

On receipt of the message one, the $\mathcal{SP}$ will verify the certificate and proceed with generating a session key.

MM-2. $\mathcal{SC}$: $K_{SC-SP} = h((g^{SP})^{SC})$

$\mathcal{SC} \rightarrow \mathcal{SP}$: $E_{K_{SC-SP}}(N_{SP}||N_{SC})||z_{V_{SP}}(CertU||CSN||N_{SP})$

The $\mathcal{SC}$ generates a session key and encrypts the generated random numbers by both $\mathcal{SC}$ and $\mathcal{SP}$. Subsequently, the $\mathcal{SC}$ encrypts its Diffie-Hellman certificate with card serial number (CSN) and $\mathcal{SP}$'s random number, using $\mathcal{SP}$'s public encryption key.

On receipt of message two, the $\mathcal{SP}$ decrypts the Diffie-Hellman certificate and proceed with generating the session key similar to the $\mathcal{SC}$.

MM-3. $\mathcal{SP} \rightarrow \mathcal{SC}$: $E_{K_{SC-SP}}(N_{SC}, SK, N_{SP})$

In response the $\mathcal{SP}$ encrypts random numbers generated by the $\mathcal{SP}$ and $\mathcal{SC}$ along with an optional symmetric key (SK). If the $\mathcal{SP}$ sends the SK that this indicates to the $\mathcal{SC}$ that $\mathcal{SP}$ will use SK for to encrypt any future messages.

MM-4. $\mathcal{SC} \rightarrow \mathcal{SP}$: $E_{SK}(N_{SP}, optionalparameters)$

In response the $\mathcal{SC}$ encrypts the $\mathcal{SP}$'s random number using the SK and if required add any optional parameters. This message provide confirmation to the $\mathcal{SP}$ that the $\mathcal{SC}$ has the same SK.

A.9 Sirett-Mayes-Markantonakis (SM) Protocol

The Sirett-Mayes-Markantonakis (SM) protocol is designed to install an applet on a SIM card that is issued by a card issuer and currently deployed in the field. The SM protocol first installs a MIDlet on a mobile phone and then proceeds with installing the application on the smart card. Before we describe the SM protocol, we introduce new notations listed below:

M	represents a mobile phone.
$A \rightarrow B \rightarrow C$	A message send by entity A to entity B, which it relays to the entity C.

SM-1. $\mathcal{SP} \rightarrow \mathcal{M}$: $Cert_{DPRC}$

The $\mathcal{SP}$ sends the root certificate of the mobile phone's J2ME Operator domain. This enables the $\mathcal{SP}$ to install its own MIDlet on the mobile phone that will assist it in the applet installation on the SIM card ($\mathcal{SC}$).

SM-2. $\mathcal{SC} \rightarrow \mathcal{M} \rightarrow \mathcal{SP} \quad : \quad SC_i||N_{SC}$

The $\mathcal{SC}$ will send its identity and a random number back to the $\mathcal{SP}$ via the $\mathcal{M}$ using the Short Message Server (SMS). The SM protocol relies on the SMS to provide security to certain messages.

On receipt, the $\mathcal{SP}$ use the $\mathcal{SC}$ identity to locate the long-term shared secret between the $\mathcal{SC}$ and $\mathcal{SP}$.

SM-3. $\mathcal{SP} \rightarrow \mathcal{M} \quad : \quad MIDlet_{SP}||Sign_{SP}(MIDlet_{SP})$

The $\mathcal{SP}$ then encrypt and MAC the applet that it wants to install on the $\mathcal{SC}$. The encrypted and MACed applet is then embedded in the $MIDlet_{SP}$ that the $\mathcal{SP}$ sends to the $\mathcal{M}$ that installs it in the operator domain of the mobile phone. The $MIDlet_{SP}$ is signed by the $\mathcal{SP}$ signature key that is certified by the root entity to the operator domain (i.e. $Cert_{DRPC}$).

SM-4. $\mathcal{M} \rightarrow \mathcal{SC} \quad : \quad e_{K_{SC-SP}}(Applet)||f_{K_{SC-SP}}(Applet)$

Once the $MIDlet_{SP}$ is installed on the $\mathcal{M}$ it communicates with the $\mathcal{SC}$ and initiate the applet download process which is described by the message 4.

Appendix B

CasperFDR Scripts

Contents

This appendix opens the discussion with a short introduction to the CasperFDR framework. Subsequently, we present the Casper scripts for the protocols that we proposed in this thesis.

B.1 Brief Introduction to the CasperFDR

For the sake of completeness, we subjected the proposed protocols in this thesis to formal mechanical analysis based on the CasperFDR tool. The CasperFDR approach uses the Communicating Sequential Processes (CSP) [143]; a mathematical framework for the description and analysis of systems that consist of processes (sub-systems). The state of a process in the CSP changes by engaging with (pre-defined) events. The CSP language defines how different sub-processes can be constructed along with how to define their interactions. The Failures-Divergence Refinement (FDR) [233] is a model-checking tool for state machines that is rooted in the CSP framework. The FDR model-checking tool defines and analyse a systems as described below:

1. All (honest) agents (entities) taking part in a system are modelled as the CSP (sub) processes, along with the intruder that can interact with other agents in the protocol.
2. The resulting system is tested against the defined (desired) security properties. The FDR searches the state space to investigate whether any insecure traces can be found.
3. If FDR finds an insecure trace, then the system does not satisfy the desired security property and the protocol is considered to be insecure in relation to the given security property.

Using the CSP to define a system is tedious and painstaking, which is remarkably simplified by the Casper framework. In Casper, a user specifies a protocol using abstract notations, similar to the one that are used to describe protocols in academic literature. The Casper takes these notations, convert them to CSP code, which is suitable to be analysed by the FDR model checking tool. Therefore, CasperFDR represents an approach where a protocol is defined in the Casper notations and then FDR tool is used to verify its suitability under given security properties.

A Casper script can be divided into two main sections: protocol and system definition, which are discussed as follow:

B.1.1 Protocol Definition

The protocol definition section of a Casper script defines the generic operations of a protocol. The protocol definition can be sub-divided into four components that are discussed below:

Protocol Description: This section represented by `#Protocol description` in a Casper script defines the message sequence of the protocol. The notations used in this section are similar to the standard method of describing a protocol [142].

Free variables: The variables and functions that are used by the protocol definition are defined in a section that is represented as `#Free variables`. The variables and functions

defined in this section are not instantiated with actual value. The instantiation is done in the system definition of a Casper script.

Processes: Each agent in the system is represented by a CSP process, which is defined in the `#Processes` of a Casper script.

Specifications: The security requirements against which the protocol is analysed by the FDR tool are defined in the `#Specification` section of a Casper script.

B.1.2 System Definition

The system definition describes the actual system that is required to be analysed as part of the protocol analysis by the FDR tool. The system definition contains four sub-components that are discussed below:

Type Definition: The variable types that are going to be used in the actual systems are instantiated in the `#Actual variables` section of a Casper script. The variables defined in the `#Free variables` are instantiated in this section, and the FDR tool will use these variables during the analysis.

Functions: Any functions defined in the `#Free variables` have to be defined under the `#Functions` heading in a Casper script.

System Definition: The agents that would be present during the execution of the protocol as part of the FDR analysis are defined under the heading `System` in a Casper script. The definition of the agents in this section corresponds to the definition of agents under the heading `Processes` of a Casper script.

Intruder: Finally, in the `#Intruder Information` section of a Casper script we define the identity and capability of an intruder in the system againist which the security requirements stipulated in `#Specification` are evaluated by the FDR tool.

B.2 Attestation Protocol

The Casper script in this section corresponds to the attestation protocol described in section 4.7.

```
#Free variables
SC, CM : Agent
ns, nsp, nt, challenge, response : Nonce
SID1, SID2 : Num
VKey: Agent -> PublicKey
SKey: Agent -> SecretKey
InverseKeys = (sKey, sKey), (VKey, SKey)

#Protocol description
0.  -> SC : CM
1.  SC -> CM : SID1,{SC, ns, CM,}{sKey}
2.  CM -> SC : {CM, ns, nm, challenge, SID2}{sKey}
```

```
3. SC -> CM : {ns,nm,nsp,response}{sKey}
4. CM -> SC : {ns,{CM,SC,ns,nsp}{Skey{CM}}}{sKey}


#Actual variables
SmartCard, CardManufacturer, MAppl : Agent
Ns, Nsp, Nt, Nm, Challenge, Response : Nonce
SIDOne, SIDTwo : Num

#Processes
INITIATOR(SC, CM, ns, nsp, response) knows   sKey, VKey
RESPONDER(CM, SC, nm, challenge) knows   sKey, SKey(CM), VKey

#System
INITIATOR(SmartCard, CardManufacturer, Ns, Nsp, Response)
RESPONDER(CardManufacturer, SmartCard, Nm, Challenge)

#Functions
symbolic VKey, SKey

#Intruder Information
Intruder = MAppl
IntruderKnowledge = {SmartCard, CardManufacturer, MAppl, MAppl, Nm, Nsp, SKey(MAppl),
VKey}

#Specification
StrongSecret(SC, sKey, [CM])
StrongSecret(SC, response, [CM])
Aliveness(SC, CM)
Aliveness(CM, SC)
```

B.3 Secure and Trusted Channel Protocol — Service Provider

The Casper script in this section corresponds to the Secure and Trusted Channel Protocol — Service Provider (STCP_{SP}) described in section 6.3.

```
#Free variables
datatype Field = Gen | Exp(Field, Num) unwinding 2
halfkeySP, halfkeyTPM, sessionKey : Field
SP, TPM : Agent
ns, nt, nm, scos, app : Nonce
s, t : Num
VKey: Agent -> PublicKey
SKey: Agent -> SecretKey
EKey: Agent -> PublicKey
DKey: Agent -> SecretKey
InverseKeys = (sessionKey, sessionKey),(VKey, SKey),(EKey, DKey),(Exp, Exp),(Gen, Gen)

#Protocol description
0.   -> SP : TPM
1.  SP -> TPM : SP, VKey(SP)
2.  TPM -> SP : {TPM, SP, nt}{VKey(SP)}
2a. TPM -> SP : {Exp(Gen, t) % halfkeyTPM}{VKey(SP)}
<sessionKey := Exp(halfkeyTPM, s)>
```

```
3. SP -> TPM : {SP, TPM, ns}{EKey(TPM)}
3a.  SP -> TPM : {Exp(Gen, s) % halfkeySP}{EKey(TPM)}
<sessionKey := Exp(halfkeySP, t)>
4. TPM -> SP : {TPM, SP, {scos (+) ns}{SKey(TPM)}}{sessionKey}
5. SP -> TPM : {SP, TPM, nt}{sessionKey}
6. TPM -> SP : {TPM, SP, {app (+) ns}{SKey(TPM)}}{sessionKey}

#Actual variables
SerPro, TruPlaMan, MAppl : Agent
Nsp, Ntpm, Nm : Nonce
SCOS, APP : Nonce
S, T, M : Num
SCOperatingSys, SApplication : Nonce

#Processes
INITIATOR(SP, TPM, ns, s, app, scos) knows SKey(SP), DKey(SP), VKey, EKey
RESPONDER(TPM, SP, nt, t, scos, app) knows SKey(TPM), DKey(TPM), VKey, EKey

#System
INITIATOR(SerPro, TruPlaMan, Nsp, S, APP, SCOS)
RESPONDER(TruPlaMan, SerPro, Ntpm, T, SCOS, APP)

#Functions
symbolic VKey, SKey, EKey, DKey

#Intruder Information
Intruder = MAppl
IntruderKnowledge = {SerPro, TruPlaMan, MAppl, MAppl, Nm, DKey(MAppl), SKey(MAppl),
VKey, EKey, M}

#Specification
StrongSecret(SP, sessionKey, [TPM])
Aliveness(SP, TPM)
Aliveness(TPM, SP)
Agreement(SP, TPM, [sessionKey])
Agreement(TPM, SP, [sessionKey])

#Equivalences
forall x, y : Num . Exp(Exp(Gen, x), y) = Exp(Exp (Gen, y), x)
```

B.4 Secure and Trusted Channel Protocol — Smart Card

The Casper script in this section corresponds to the Secure and Trusted Channel Protocol — Smart Card (STCP_{SC}) described in section6.4.

```
#Free variables
datatype Field = Gen | Exp(Field, Num) unwinding 2
halfkeySP, halfkeyTPM, sessionKey : Field
SP, TPM : Agent
ns, nt, nm, scos, app : Nonce
s, t : Num
VKey: Agent -> PublicKey
SKey: Agent -> SecretKey
```

```
EKey: Agent -> PublicKey
DKey: Agent -> SecretKey
InverseKeys = (sessionKey, sessionKey),(VKey, SKey),(EKey, DKey),(Exp, Exp),\
(Gen, Gen)

#Protocol description
0.  -> SP : TPM
1.  SP -> TPM : SP, VKey(SP)
2.  TPM -> SP : {TPM, SP, nt}{VKey(SP)}
2a. TPM -> SP : {Exp(Gen, t) % halfkeyTPM}  {VKey(SP)}
<sessionKey := Exp(halfkeyTPM, s)>
3. SP -> TPM : {SP, TPM, ns}{EKey(TPM)}
3a.  SP -> TPM : {Exp(Gen, s) % halfkeySP}  {EKey(TPM)}
<sessionKey := Exp(halfkeySP, t)>
4. TPM -> SP : {TPM, SP, {scos (+) ns}{SKey(TPM)}}{sessionKey}
5. SP -> TPM : {SP, TPM, nt}{sessionKey}
6. TPM -> SP : {TPM, SP, {app (+) ns}{SKey(TPM)}}{sessionKey}

#Actual variables
SerPro, TruPlaMan, MAppl : Agent
Nsp, Ntpm, Nm : Nonce
SCOS, APP : Nonce
S, T, M : Num
SCOperatingSys, SApplication : Nonce

#Processes
INITIATOR(SP, TPM, ns, s, app, scos) knows  SKey(SP), DKey(SP), VKey, EKey
RESPONDER(TPM, SP, nt, t, scos, app) knows   SKey(TPM), DKey(TPM), VKey, EKey

#System
INITIATOR(SerPro, TruPlaMan, Nsp, S, APP, SCOS)
RESPONDER(TruPlaMan, SerPro, Ntpm, T, SCOS, APP)

#Functions
symbolic VKey, SKey, EKey, DKey

#Intruder Information
Intruder = MAppl
IntruderKnowledge = {SerPro, TruPlaMan, MAppl, MAppl, Nm, DKey(MAppl),\
SKey(MAppl), VKey, EKey, M}

#Specification
StrongSecret(SP, sessionKey, [TPM])
Aliveness(SP, TPM)
Aliveness(TPM, SP)
Agreement(SP, TPM, [sessionKey])
Agreement(TPM, SP, [sessionKey])

#Equivalences
forall x, y : Num . Exp(Exp(Gen, x), y) = Exp(Exp (Gen, y), x)
```

B.5 Application Acquisition and Contractual Agreement Protocol

The Casper script in this section corresponds to the Application Acquisition and Contractual Agreement Protocol (STCP_{ACA}) described in section 6.5.

```
#Free variables
datatype Field = Gen | Exp(Field, Num) unwinding 2
halfkeySP, halfkeySC, DHKey : Field
datatype ACAPKeys = MAC(Field, Num, Num) unwinding 2
EnKey, MaKey : ACAPKeys
SC, SP, User, TSM: Agent
User: tIdentities
Appi: ApplicationIdentity
CardID, SeudoAppi: SeudoIdentities
gSC, gSP: Num
nSC, nSP, nTSM: Nonce
SCOS: SmartCardOS
App: SPApplication
f: HashFunction
VKey: Agent->PublicKey
SKey: Agent->SecretKey
TEKey, TAKey : SessionKey
InverseKeys = (VKey, SKey), (EnKey, EnKey), (MaKey, MaKey), (TEKey, TEKey),\
 (TAKey, TAKey)


#Protocol description
0.   -> SP : SC
[SC!=SP]
1. SP -> SC : nSP, Exp(Gen,gSP)%halfkeySP
[SC!=SP]
<DHKey:=Exp(halfkeySP,gSC);EnKey:=MAC(DHKey,nSP,nSC);MaKey:=MAC(DHKey, nSP, nSC)>
2. SC -> SP : nSC, Exp(Gen,gSC)%halfkeySC
[SP != SC]
<DHKey:=Exp(halfkeySC,gSP);EnKey:=MAC(DHKey,nSP,nSC);MaKey:=MAC(DHKey,nSP,nSC)>
3. SP -> SC: nSP, nSC
4. SC -> SP: {{{SCi, Useri, nSP, nSC}{SKey(User)}}{EnKey}}{MaKey}
5. SP -> SC : {{{SPi, Appi, nSC, nSP}{SKey(SP)}}{EnKey}}{MaKey}
6. SC -> SP : {{{f(SCOS)%saveHash, SCi, Useri, SPi, nSC, nSP}{SKey(SC)}}\
{EnKey}}{MaKey}
7. SP -> SC : {{App}{EnKey}}{MaKey}
8. SC -> SP : {{{f(App), SPi, Appi, SCi, Useri, nSP, nSC}{SKey(SC)}}\
{EnKey}}{MaKey}
9. SP -> SC : {{{saveHash%f(SCOS), f(App), SCi, Useri, SPi, nSP, nSC}\
{SKey(SP)}}{EnKey}}{MaKey}
10. SC -> TSM : CardID, {{TSMi, SCi, Useri, nSC, SeudoAppi}{TEKey}}{TAKey}
11. TSM -> SC : {{{TSMi, SCi, Useri, SeudoAppi, nTSM, nSC}{SKey(TSM)}}\
{TEKey}}{TAKey}

#Actual variables
SCard, SProvider, USER, TrustedSM, MaliciousEntity: Agent
ISCard, ISProvider, IUSER, ITrustedSM, IMaliciousEntity: AgentIdentities
GSC, GSP, GMalicious: Num
NSC, NSP, NTSM, NMalicious: Nonce
```

```
AppI: ApplicationIdentity
CARDID, SeudoAPPi: SeudoIdentities
SmartCOS: SmartCardOS
APP: SPApplication
TEKEY, TAKEY : SessionKey
InverseKeys = (TEKEY,TEKEY), (TAKEY,TAKEY)

#Processes
INITIATOR(SP, SPi, SC, User, gSP, nSP, App, Appi)knows SKey(SP), VKey
RESPONDER(SC, SCi, SP, User, Useri, TSM, TSMi, SeudoAppi, CardID, SCOS, gSC, nSC,\
 TEKey, TAKey) knows SKey(User), SKey(SC), VKey
SERVER(TSM,TSMi, SC, SCi, User, CardID, nTSM, TEKey, TAKey)knows SKey(TSM), VKey

#System
INITIATOR(SProvider, ISProvider, SCard, USER, GSP, NSP, APP, AppI)
RESPONDER(SCard, ISCard, SProvider, USER, IUSER, TrustedSM, ITrustedSM, SeudoAPPi,
CARDID, SmartCOS, GSC, NSC, TEKEY, TAKEY)
SERVER(TrustedSM,ITrustedSM, SCard, ISCard, USER, CARDID, NTSM, TEKEY, TAKEY)

#Functions
symbolic VKey, SKey

#Intruder Information
Intruder = MaliciousEntity
IntruderKnowledge = {SProvider, SCard, MaliciousEntity, IMaliciousEntity, \
GMalicious, NMalicious, SKey(MaliciousEntity), VKey}

#Specification
Aliveness(SP, SC)
Aliveness(SC, SP)
Aliveness(SC, TSM)
Aliveness(TSM, SC)
Agreement(SP, SC, [DHKey, EnKey, MaKey])
StrongSecret(SP, Appi, [SC])
StrongSecret(SC, Appi, [SP])
StrongSecret(SP, Useri, [SC])
StrongSecret(SC, Useri, [SP])

#Equivalences
forall x, y : Num . Exp(Exp(Gen, x), y) = Exp(Exp(Gen, y), x)
```

B.6 Application Binding Protocol — Local

The Casper script in this section corresponds to the Application Binding Protocol — Local (ABPL) described in section 7.4.

```
#Free variables
S, C, spS, spC : Agent
TPM : Server
nc, ns, nm : Nonce
ksc, abKsc : SessionKey
f : HashFunction
ServerKey : Agent -> ServerKeys
VKey : Agent -> Publickey
```

```
SKey : Agent -> SecretKey
realAgent : Server -> Bool
{math}InverseKeys = (ksc, ksc), (abKsc, abKsc), (ServerKey, ServerKey),(VKey, SKey)
emph{}

#Actual variables
CApp, SApp, MAppl : Agent
TM : Server
Nc, Ns, Nm : Nonce
Ksc, ABKsc : SessionKey
InverseKeys = (Ksc, Ksc), (ABKsc, ABKsc)
emph{}

#Processes
INITIATOR(C, TPM, S, nc) knows f(S), ServerKey(C), SKey(C), VKey
RESPONDER(S, TPM, C, ns, abKsc) knows f(C), ServerKey(S), SKey(S), VKey
SERVER(TPM, ksc) knows ServerKey
emph{}

#System
INITIATOR(CApp,TM, SApp, Nc)
RESPONDER(SApp,TM, CApp, Ns, ABKsc)
SERVER(TM, Ksc)
emph{}

#Protocol description
0.   -> C : S
1.  C -> S : C, S, {C, S, nc, {C, S, nc}{ServerKey(C)} % mTPM}{SKey(C)}
2.  S -> TPM : S, TPM, C, {S, C, ns}{ServerKey(S)},  mTPM % {C,S,nc}{ServerKey(C)}
[realAgent(TPM)]
3.  TPM -> S : TPM, S, {f(S), ksc, nc}{ServerKey(C)}  % tpmC
[realAgent(TPM)]
3a.  TPM -> S : TPM, {f(C), ksc, ns}{ServerKey(S)}
4.  S -> C : S, C, tpmC % {f(S), ksc, nc}{ServerKey(C)}
4a.  S -> C : {abKsc, nc, ns}{ksc},{S, C, nc(+)ns}{abKsc}
5.  C -> S : C, S, {nc(+)ns}{abKsc}
emph{}

#Specification
StrongSecret(TPM, ksc, [S,C])
Aliveness(S, C)
Aliveness(C, S)
StrongSecret(S, abKsc, [C])
Agreement(S, C, [abKsc])
Agreement(C, S, [abKsc])
emph{}

#Inline functions
symbolic ServerKey
symbolic VKey, SKey
realAgent(TM)=true
realAgent(_)=false
emph{}

#Intruder Information
Intruder = MAppl
IntruderKnowledge = {CApp, SApp, MAppl, Nm, ServerKey(MAppl), SKey(MAppl),VKey}
```

B.7 Platform Binding Protocol

The Casper script in this section corresponds to the Platform Binding Protocol (PBP) described in section 7.5.

```
#Free variables
datatype Field = Gen | Exp(Field, Num) unwinding 2
halfkeySP, halfkeyTPM, sessionKey : Field
SP, TPM : Agent
ns, nt, nm, scos, app : Nonce
s, t : Num
VKey: Agent -> PublicKey
SKey: Agent -> SecretKey
EKey: Agent -> PublicKey
DKey: Agent -> SecretKey
InverseKeys = (sessionKey, sessionKey), (VKey, SKey),(EKey, DKey), (Exp, Exp), (Gen,
Gen)

#Protocol description
0.  -> SP : TPM
1.  SP -> TPM : SP, VKey(SP)
2.  TPM -> SP : {TPM, SP, nt}{VKey(SP)}
2a. TPM -> SP : {Exp(Gen, t) % halfkeyTPM}{VKey(SP)}
<sessionKey := Exp(halfkeyTPM, s)>
3. SP -> TPM : {SP, TPM, ns}{EKey(TPM)}
3a.  SP -> TPM : {Exp(Gen, s) % halfkeySP}{EKey(TPM)}
<sessionKey := Exp(halfkeySP, t)>
4. TPM -> SP : {TPM, SP, {scos (+) ns}{SKey(TPM)}}{sessionKey}
5. SP -> TPM : {SP, TPM, nt}{sessionKey}
6. TPM -> SP : {TPM, SP, {app (+) ns}{SKey(TPM)}}{sessionKey}

#Actual variables
SerPro, TruPlaMan, MAppl : Agent
Nsp, Ntpm, Nm : Nonce
SCOS, APP : Nonce
S, T, M : Num
SCOperatingSys, SApplication : Nonce

#Processes
INITIATOR(SP, TPM, ns, s, app, scos) knows SKey(SP), DKey(SP), VKey, EKey
RESPONDER(TPM, SP, nt, t, scos, app) knows SKey(TPM), DKey(TPM), VKey, EKey

#System
INITIATOR(SerPro, TruPlaMan, Nsp, S, APP, SCOS)
RESPONDER(TruPlaMan, SerPro, Ntpm, T, SCOS, APP)

#Functions
symbolic VKey, SKey, EKey, DKey

#Intruder Information
Intruder = MAppl
IntruderKnowledge = {SerPro, TruPlaMan, MAppl, MAppl, Nm, DKey(MAppl),SKey(MAppl),
VKey, EKey, M}

#Specification
StrongSecret(SP, sessionKey, [TPM])
```

```
Aliveness(SP, TPM)
Aliveness(TPM, SP)
Agreement(SP, TPM, [sessionKey])
Agreement(TPM, SP, [sessionKey])

#Equivalences
forall x, y : Num . Exp(Exp(Gen, x), y) = Exp(Exp (Gen, y), x)
```

B.8 Application Binding Protocol — Distributed

The Casper script in this section corresponds to the Application Binding Protocol — Distributed (ABPD) described in section 7.6.

```
#Free variables
datatype Field = Gen | Exp(Field, Num) unwinding 2
halfkeySP, halfkeyTPM, sessionKey : Field
SP, TPM : Agent
ns, nt, nm, scos, app : Nonce
s, t : Num
VKey: Agent -> PublicKey
SKey: Agent -> SecretKey
EKey: Agent -> PublicKey
DKey: Agent -> SecretKey
InverseKeys = (sessionKey, sessionKey), (VKey, SKey),(EKey, DKey), (Exp, Exp), (Gen,
Gen)

#Protocol description
0.  -> SP : TPM
1.  SP -> TPM : SP, VKey(SP)
2.  TPM -> SP : {TPM, SP, nt}{VKey(SP)}
2a. TPM -> SP : {Exp(Gen, t) % halfkeyTPM}{VKey(SP)}
<sessionKey := Exp(halfkeyTPM, s)>
3. SP -> TPM : {SP, TPM, ns}{EKey(TPM)}
3a.  SP -> TPM : {Exp(Gen, s) % halfkeySP}{EKey(TPM)}
<sessionKey := Exp(halfkeySP, t)>
4. TPM -> SP : {TPM, SP, {scos (+) ns}{SKey(TPM)}}{sessionKey}
5. SP -> TPM : {SP, TPM, nt}{sessionKey}
6. TPM -> SP : {TPM, SP, {app (+) ns}{SKey(TPM)}}{sessionKey}

#Actual variables
SerPro, TruPlaMan, MAppl : Agent
Nsp, Ntpm, Nm : Nonce
SCOS, APP : Nonce
S, T, M : Num
SCOperatingSys, SApplication : Nonce

#Processes
INITIATOR(SP, TPM, ns, s, app, scos) knows SKey(SP), DKey(SP), VKey, EKey
RESPONDER(TPM, SP, nt, t, scos, app) knows SKey(TPM), DKey(TPM), VKey, EKey

#System
INITIATOR(SerPro, TruPlaMan, Nsp, S, APP, SCOS)
RESPONDER(TruPlaMan, SerPro, Ntpm, T, SCOS, APP)
```

```
#Functions
symbolic VKey, SKey, EKey, DKey

#Intruder Information
Intruder = MAppl
IntruderKnowledge = {SerPro, TruPlaMan, MAppl, MAppl, Nm, DKey(MAppl) ,SKey(MAppl),
VKey, EKey, M}

#Specification
StrongSecret(SP, sessionKey, [TPM])
Aliveness(SP, TPM)
Aliveness(TPM, SP)
Agreement(SP, TPM, [sessionKey])
Agreement(TPM, SP, [sessionKey])

#Equivalences
forall x, y : Num . Exp(Exp(Gen, x), y) = Exp(Exp (Gen, y), x)
```

Appendix C

Practical Implementation Source Code

Contents

In this appendix, we detail the test implementation of the proposed protocols and frameworks discussed in this thesis.

C.1 Offline Attestation Mechanism

In this section, we detail the Java Card implementation of the offline attestation mechanism based on PRNG and PUF algorithms discussed in section 4.5.2 and 4.5.1, respectively.

C.1.1 Offline PRNG Algorithm

The Java Card implementation of the offline PRNG algorithm discussed in section 4.5.2.

```
1  package selftestOfflinePRNG;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.JCSystem;
8  import javacard.framework.Util;
9  import javacard.security.AESKey;
10 import javacard.security.Key;
11 import javacard.security.KeyBuilder;
12 import javacard.security.KeyPair;
13 import javacard.security.MessageDigest;
14 import javacard.security.RSAPrivateKey;
15 import javacard.security.RSAPublicKey;
16 import javacard.security.RandomData;
17 import javacard.security.Signature;
18 import javacardx.apdu.ExtendedLength;
19 import javacardx.crypto.Cipher;
20 import javacard.security.MessageDigest;
21
22 public class SelftestOffline extends Applet implements ExtendedLength {
23   private static byte[] MemoryContents =  {
24     (byte)0x37, (byte)0x7a, (byte)0xbc, (byte)0xc0, (byte)0xea, (byte)
25      0x07, (byte)0x94, (byte)0x59, (byte)0xd6, (byte)0x37, (byte)0x6b,
26      (byte)0x4c, (byte)0x82, (byte)0xdb, (byte)0x54, (byte)0xb2,
27      (byte)0xe8, (byte)0xea, (byte)0x71, (byte)0xe1, (byte)0xa4,
28      (byte)0x41, (byte)0x06, (byte)0x44, (byte)0xfe, (byte)0x86,
29      (byte)0x8e, (byte)0x4f, (byte)0x39, (byte)0xf5, (byte)0xdb,
30      (byte)0xd1, (byte)0xf1, (byte)0xc5, (byte)0xd8, (byte)0xac,
31      (byte)0xbb, (byte)0x73, (byte)0x51, (byte)0xa1, (byte)0xa3,
32      (byte)0x8a, (byte)0x26, (byte)0x5d, (byte)0xf3, (byte)0x61,
33      (byte)0x55, (byte)0x56, (byte)0x39, (byte)0x3f, (byte)0x4c,
34      (byte)0x2a, (byte)0x43, (byte)0xc4, (byte)0xd7, (byte)0xa1,
35      (byte)0xaa, (byte)0xc1, (byte)0xf2, (byte)0xd6, (byte)0x07,
36      (byte)0xa8, (byte)0x58, (byte)0x9a, (byte)0x70, (byte)0x84,
37      (byte)0x15, (byte)0x19, (byte)0x56, (byte)0x61, (byte)0x3d,
38      (byte)0x88, (byte)0x2a, (byte)0x44, (byte)0x54, (byte)0x29,
39      (byte)0x29, (byte)0x26, (byte)0x36, (byte)0x06, (byte)0xfe,
40      (byte)0xad, (byte)0x27, (byte)0x13, (byte)0x86, (byte)0x0e,
41      (byte)0x85, (byte)0x3c, (byte)0x32, (byte)0xe2, (byte)0x38,
```

```
42      (byte)0xd2, (byte)0x91, (byte)0x82, (byte)0x89, (byte)0xce,
43      (byte)0x79, (byte)0x02, (byte)0x43, (byte)0xfd, (byte)0xaf,
44      (byte)0x18, (byte)0xe8, (byte)0x5b, (byte)0xd4, (byte)0x72,
45      (byte)0x03, (byte)0x63, (byte)0x2b, (byte)0x29, (byte)0x72,
46      (byte)0xe0, (byte)0x92, (byte)0x54, (byte)0x06, (byte)0x1c,
47      (byte)0x7f, (byte)0xc7, (byte)0x37, (byte)0x93, (byte)0x2f,
48      (byte)0x7a, (byte)0x84, (byte)0x95, (byte)0xec, (byte)0x5e,
49      (byte)0xa5, (byte)0xf6, (byte)0x4e, (byte)0x7e, (byte)0x1f,
50      (byte)0xe6, (byte)0xe2, (byte)0x04, (byte)0x2e, (byte)0x25,
51      (byte)0x7f, (byte)0x2f, (byte)0x3c, (byte)0xfe, (byte)0x57,
52      (byte)0x9e, (byte)0x7f, (byte)0xce, (byte)0x72, (byte)0xc0,
53      (byte)0xe9, (byte)0x79, (byte)0x05, (byte)0xc5, (byte)0xfd,
54      (byte)0x6a, (byte)0x46, (byte)0xfe, (byte)0x33, (byte)0x84,
55      (byte)0x3f, (byte)0x09, (byte)0xae, (byte)0x01, (byte)0x18,
56      (byte)0x5a, (byte)0xf6, (byte)0xc6, (byte)0xd3, (byte)0xa1,
57      (byte)0xe2, (byte)0x90, (byte)0x83, (byte)0x79, (byte)0xee,
58      (byte)0xa6, (byte)0xd4, (byte)0xf6, (byte)0xd1, (byte)0x86,
59      (byte)0x91, (byte)0x34, (byte)0x00, (byte)0xd3, (byte)0xe4,
60      (byte)0x8a, (byte)0xfb, (byte)0xaa, (byte)0x6c, (byte)0xe5,
61      (byte)0x46, (byte)0xa7, (byte)0x00, (byte)0x9e, (byte)0xd8,
62      (byte)0x81, (byte)0xbc, (byte)0xd1, (byte)0xb5, (byte)0x60,
63      (byte)0xd5, (byte)0x91, (byte)0x13, (byte)0x06, (byte)0x68,
64      (byte)0x21, (byte)0x8f, (byte)0x7d, (byte)0xc2, (byte)0x3e,
65      (byte)0xd2, (byte)0x75, (byte)0x0f, (byte)0x97, (byte)0x64,
66      (byte)0xb1, (byte)0xdb, (byte)0x74, (byte)0x6e, (byte)0x91,
67      (byte)0x6b, (byte)0xa7, (byte)0x7d, (byte)0xef, (byte)0x8b,
68      (byte)0x37, (byte)0xb7, (byte)0x84, (byte)0x1e, (byte)0xa7,
69      (byte)0x26, (byte)0x26, (byte)0xea, (byte)0xe9, (byte)0xb7,
70      (byte)0x5e, (byte)0x3f, (byte)0xdf, (byte)0xa4, (byte)0xc5,
71      (byte)0x45, (byte)0x4e, (byte)0x34, (byte)0x33, (byte)0xe5,
72      (byte)0x43, (byte)0x46, (byte)0xc0, (byte)0x2b, (byte)0xbd,
73      (byte)0x85, (byte)0x2f, (byte)0xca, (byte)0xf8, (byte)0x9d,
74      (byte)0xb4, (byte)0xbc, (byte)0x67, (byte)0x92, (byte)0xd4,
75      (byte)0x33, (byte)0xfd, (byte)0xbd, (byte)0x82, (byte)0x9d,
76      (byte)0x62, (byte)0xfc, (byte)0xbb, (byte)0xd2, (byte)0xad,
77      (byte)0x05, (byte)0xa2, (byte)0xfc, (byte)0x2d, (byte)0xe3,
78      (byte)0x02, (byte)0xe2, (byte)0x41, (byte)0x9b, (byte)0x1f,
79      (byte)0xf8, (byte)0x87, (byte)0x15, (byte)0x89, (byte)0xfb,
80      (byte)0x53, (byte)0x99, (byte)0xb3, (byte)0xeb, (byte)0xdb,
81      (byte)0x01, (byte)0xaf, (byte)0x71, (byte)0xd2, (byte)0xf2,
82      (byte)0x73, (byte)0xb7, (byte)0x82, (byte)0x30, (byte)0x25,
83      (byte)0x04, (byte)0x29, (byte)0x2b, (byte)0xb9, (byte)0x92,
84      (byte)0x92, (byte)0x35, (byte)0x97, (byte)0x0e, (byte)0xb8,
85      (byte)0xf2, (byte)0xc6, (byte)0x2e, (byte)0xa7, (byte)0x2d,
86      (byte)0x0c, (byte)0x09, (byte)0x5e, (byte)0x07, (byte)0x06,
87      (byte)0x67, (byte)0xa0, (byte)0xdf, (byte)0x55, (byte)0x09,
88      (byte)0xfc, (byte)0xee, (byte)0x2b, (byte)0x13, (byte)0x1a,
89      (byte)0x2e, (byte)0x5d, (byte)0x0a, (byte)0xbb, (byte)0x45,
90      (byte)0x75, (byte)0xf4, (byte)0xd8, (byte)0xdc, (byte)0x2e,
91      (byte)0x99, (byte)0x2a, (byte)0x13, (byte)0xa1, (byte)0x1e,
92      (byte)0x99, (byte)0xfd, (byte)0xdc, (byte)0xcf, (byte)0xcc,
```

```
93      (byte)0x3f, (byte)0x42, (byte)0xf7, (byte)0x3d, (byte)0x73,
94      (byte)0xee, (byte)0xca, (byte)0x76, (byte)0xe4, (byte)0x75,
95      (byte)0xc4, (byte)0x21, (byte)0xd4, (byte)0x14, (byte)0x2e,
96      (byte)0x22, (byte)0x9c, (byte)0xce, (byte)0x10, (byte)0xaf,
97      (byte)0xa6, (byte)0x25, (byte)0xa0, (byte)0x01, (byte)0xb1,
98      (byte)0x82, (byte)0xba, (byte)0x4c, (byte)0xb2, (byte)0x66,
99      (byte)0x89, (byte)0x89, (byte)0x6b, (byte)0x06, (byte)0x15,
100     (byte)0xba, (byte)0x64, (byte)0xa3, (byte)0x73, (byte)0x88,
101     (byte)0x34, (byte)0x99, (byte)0x3e, (byte)0x75, (byte)0x24,
102     (byte)0xf4, (byte)0xba, (byte)0xb0, (byte)0x22, (byte)0x8f,
103     (byte)0xc3, (byte)0x44, (byte)0x74, (byte)0x0b, (byte)0x52,
104     (byte)0x96, (byte)0xc6, (byte)0x97, (byte)0x8b, (byte)0xf2,
105     (byte)0xe3, (byte)0xc1, (byte)0xaf, (byte)0x53, (byte)0x03,
106     (byte)0x51, (byte)0xa7, (byte)0x0d, (byte)0x42, (byte)0x6a,
107     (byte)0x20, (byte)0x03, (byte)0x31, (byte)0xb4, (byte)0xc9,
108     (byte)0xaa, (byte)0x9e, (byte)0xda, (byte)0x6f, (byte)0x7b,
109     (byte)0xb8, (byte)0x6d, (byte)0x54, (byte)0x57, (byte)0xa8,
110     (byte)0xed, (byte)0x51, (byte)0xa4, (byte)0x23, (byte)0x05,
111     (byte)0x0b, (byte)0xb3, (byte)0x90, (byte)0x42, (byte)0x38,
112     (byte)0xa8, (byte)0xbc, (byte)0xd5, (byte)0x2f, (byte)0x87,
113     (byte)0x82, (byte)0x5b, (byte)0xff, (byte)0xdb, (byte)0xba,
114     (byte)0x41, (byte)0x18, (byte)0xe0, (byte)0x4a, (byte)0x07,
115     (byte)0x04, (byte)0xe1, (byte)0x3c, (byte)0xd5, (byte)0xbf,};
116   byte[] tempSeed = {
117     (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
118     (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
119     (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
120     (byte)0x00};
121   private byte[] SignedDataTag =  {
122     (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x02};
123   short copyPointer = (short)0;
124   final static byte CLA = (byte)0xB0;
125   final static byte StartProtocol = (byte)0x40;
126   final static byte selftest = (byte)0xff;
127   final static short SW_CLASSNOTSUPPORTED = 0x6320;
128   final static short SW_ERROR_INS = 0x6300;
129   RandomData randomDataGen;
130   Cipher pkCipher;
131   byte[] receivingBuffer = null;
132   short bytesLeft = 0;
133   short readCount = 0;
134   short rCount = 0;
135   short signlength = 0;
136   MessageDigest SHA256;
137   AESKey cipherKey;
138   Cipher syCipher;
139   byte[] InitialisationVector =  {
140     (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89,
          (byte)0x99,
141     (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)
142     0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52};
```

```
143   Signature phSign;
144   PrngSHA256 myPrngHMAC;
145
146   private SelftestOffline() {
147     phSign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
148     phSCKeyPair = new KeyPair(KeyPair.ALG_RSA, KeyBuilder.LENGTH_RSA_512);
149     cipherKey = (AESKey)KeyBuilder.buildKey
150                     (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
151                     KeyBuilder.LENGTH_AES_128, false);
152     syCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
153                                   false);
154     myPrngHmac = new PrngSHA256();
155     pkCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
156     phSCKeyPair.genKeyPair();
157     SHA256 = MessageDigest.getInstance(MessageDigest.ALG_SHA_256,
158       false);
159   }
160   public static void install(byte bArray[], short bOffset, byte bLength)
161                               throws ISOException {
162     new SelftestOffline().register();
163   }
164
165   public void process(APDU apdu)throws ISOException {
166     byte[] apduBuffer = apdu.getBuffer();
167     if (selectingApplet()) {
168       this.initialise();
169       return ;
170     }
171     if (apduBuffer[ISO7816.OFFSET_CLA] != CLA) {
172       ISOException.throwIt(SW_CLASSNOTSUPPORTED);
173     }
174     if (apduBuffer[ISO7816.OFFSET_INS] == selftest) {
175       receivingBuffer = JCSystem.makeTransientByteArray((short)84,
176         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
177       generateResponse((short)1);
178       apdu.setOutgoing();
179       apdu.setOutgoingLength((short)copyPointer);
180       apdu.sendBytesLong(receivingBuffer, (short)0, (short)copyPointer);
181       return ;
182     }
183     JCSystem.requestObjectDeletion();
184   }
185
186   void selftestProcess(){
187    byte[] memoryWordRead = new byte[4];
188    byte rcount = (byte)0x00;
189    while (rcount <  MemoryContents.length){
190     Util.arrayCopyNonAtomic(MemoryContents,
191          rcount, memoryWordRead, (short)0, memoryWordRead.length);
192          generateSeed(memoryWordRead, tempSeed);
193          rcount += (byte)(rcount+(short)4);
```

```
194     }
195     if(seedZero()){
196           generateMACPrng(tempSeed, cipherKey);
197           }else{
198           ISOException.throwIt((short)0xFA17);
199           }
200   }
201
202   private void generateResponse() {
203     copyPointer = 0;
204     selftestProcess();
205     phDecryption();
206     getSignatureKey();
207   }
208
209   void phDecryption(){
210     syCipher.init(phCipherKey, Cipher.MODE_DECRYPT, InitialisationVector,
211                   (short)0, (short)InitialisationVector.length);
212     syCipher.doFinal(signatureKey, inbuffOffset, inbuffLength,
          signatureKey,
213                      inbuffOffset);
214   }
215   void getSignatureKey(){
216    RSAPrivateKey myPrivate = (RSAPrivateKey)this.phSCKeyPair.getPrivate();
217       short kLen = myPrivate.getExponent(receivingBuffer, (short)
218         (copyPointer + (short)2));
219       this.shortToBytes(receivingBuffer, copyPointer, kLen);
220       copyPointer += (short)(kLen + (short)2);
221       receivingBuffer[6]++;
222       copyPointer = Util.arrayCopyNonAtomic(this.ModulusTag, (short)0,
223                      receivingBuffer, (short)(copyPointer), (short)
224                      this.ModulusTag.length);
225       kLen = myPrivate.getModulus(receivingBuffer, (short)
226                                   (copyPointer + (short)2));
227       this.shortToBytes(receivingBuffer, copyPointer, kLen);
228   }
229
230   boolean seedZero(){
231    for(short i=0; i<tempSeed.length; i++){
232     if(tempSeed[i]!=(byte)0x00){
233      return true;
234     }
235    }
236    return false;
237   }
238
239 }
```

C.1.2 Offline PUF Algorithm

The Java Card implementation that emulates the offline PUF algorithm discussed in section 4.5.1.

```
1  package selftestOfflinePUF;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.JCSystem;
8  import javacard.framework.Util;
9  import javacard.security.AESKey;
10 import javacard.security.Key;
11 import javacard.security.KeyBuilder;
12 import javacard.security.KeyPair;
13 import javacard.security.MessageDigest;
14 import javacard.security.RSAPrivateKey;
15 import javacard.security.RSAPublicKey;
16 import javacard.security.RandomData;
17 import javacard.security.Signature;
18 import javacardx.apdu.ExtendedLength;
19 import javacardx.crypto.Cipher;
20 import javacard.security.MessageDigest;
21
22 public class SelftestOffline extends Applet implements ExtendedLength {
23   private static byte[] MemoryContents =  {
24     (byte)0x37, (byte)0x7a, (byte)0xbc, (byte)0xc0, (byte)0xea, (byte)
25      0x07, (byte)0x94, (byte)0x59, (byte)0xd6, (byte)0x37, (byte)0x6b,
26      (byte)0x4c, (byte)0x82, (byte)0xdb, (byte)0x54, (byte)0xb2,
27      (byte)0xe8, (byte)0xea, (byte)0x71, (byte)0xe1, (byte)0xa4,
28      (byte)0x41, (byte)0x06, (byte)0x44, (byte)0xfe, (byte)0x86,
29      (byte)0x8e, (byte)0x4f, (byte)0x39, (byte)0xf5, (byte)0xdb,
30      (byte)0xd1, (byte)0xf1, (byte)0xc5, (byte)0xd8, (byte)0xac,
31      (byte)0xbb, (byte)0x73, (byte)0x51, (byte)0xa1, (byte)0xa3,
32      (byte)0x8a, (byte)0x26, (byte)0x5d, (byte)0xf3, (byte)0x61,
33      (byte)0x55, (byte)0x56, (byte)0x39, (byte)0x3f, (byte)0x4c,
34      (byte)0x2a, (byte)0x43, (byte)0xc4, (byte)0xd7, (byte)0xa1,
35      (byte)0xaa, (byte)0xc1, (byte)0xf2, (byte)0xd6, (byte)0x07,
36      (byte)0xa8, (byte)0x58, (byte)0x9a, (byte)0x70, (byte)0x84,
37      (byte)0x15, (byte)0x19, (byte)0x56, (byte)0x61, (byte)0x3d,
38      (byte)0x88, (byte)0x2a, (byte)0x44, (byte)0x54, (byte)0x29,
39      (byte)0x29, (byte)0x26, (byte)0x36, (byte)0x06, (byte)0xfe,
40      (byte)0xad, (byte)0x27, (byte)0x13, (byte)0x86, (byte)0x0e,
41      (byte)0x85, (byte)0x3c, (byte)0x32, (byte)0xe2, (byte)0x38,
42      (byte)0xd2, (byte)0x91, (byte)0x82, (byte)0x89, (byte)0xce,
43      (byte)0x79, (byte)0x02, (byte)0x43, (byte)0xfd, (byte)0xaf,
44      (byte)0x18, (byte)0xe8, (byte)0x5b, (byte)0xd4, (byte)0x72,
45      (byte)0x03, (byte)0x63, (byte)0x2b, (byte)0x29, (byte)0x72,
46      (byte)0xe0, (byte)0x92, (byte)0x54, (byte)0x06, (byte)0x1c,
```

```
47      (byte)0x7f, (byte)0xc7, (byte)0x37, (byte)0x93, (byte)0x2f,
48      (byte)0x7a, (byte)0x84, (byte)0x95, (byte)0xec, (byte)0x5e,
49      (byte)0xa5, (byte)0xf6, (byte)0x4e, (byte)0x7e, (byte)0x1f,
50      (byte)0xe6, (byte)0xe2, (byte)0x04, (byte)0x2e, (byte)0x25,
51      (byte)0x7f, (byte)0x2f, (byte)0x3c, (byte)0xfe, (byte)0x57,
52      (byte)0x9e, (byte)0x7f, (byte)0xce, (byte)0x72, (byte)0xc0,
53      (byte)0xe9, (byte)0x79, (byte)0x05, (byte)0xc5, (byte)0xfd,
54      (byte)0x6a, (byte)0x46, (byte)0xfe, (byte)0x33, (byte)0x84,
55      (byte)0x3f, (byte)0x09, (byte)0xae, (byte)0x01, (byte)0x18,
56      (byte)0x5a, (byte)0xf6, (byte)0xc6, (byte)0xd3, (byte)0xa1,
57      (byte)0xe2, (byte)0x90, (byte)0x83, (byte)0x79, (byte)0xee,
58      (byte)0xa6, (byte)0xd4, (byte)0xf6, (byte)0xd1, (byte)0x86,
59      (byte)0x91, (byte)0x34, (byte)0x00, (byte)0xd3, (byte)0xe4,
60      (byte)0x8a, (byte)0xfb, (byte)0xaa, (byte)0x6c, (byte)0xe5,
61      (byte)0x46, (byte)0xa7, (byte)0x00, (byte)0x9e, (byte)0xd8,
62      (byte)0x81, (byte)0xbc, (byte)0xd1, (byte)0xb5, (byte)0x60,
63      (byte)0xd5, (byte)0x91, (byte)0x13, (byte)0x06, (byte)0x68,
64      (byte)0x21, (byte)0x8f, (byte)0x7d, (byte)0xc2, (byte)0x3e,
65      (byte)0xd2, (byte)0x75, (byte)0x0f, (byte)0x97, (byte)0x64,
66      (byte)0xb1, (byte)0xdb, (byte)0x74, (byte)0x6e, (byte)0x91,
67      (byte)0x6b, (byte)0xa7, (byte)0x7d, (byte)0xef, (byte)0x8b,
68      (byte)0x37, (byte)0xb7, (byte)0x84, (byte)0x1e, (byte)0xa7,
69      (byte)0x26, (byte)0x26, (byte)0xea, (byte)0xe9, (byte)0xb7,
70      (byte)0x5e, (byte)0x3f, (byte)0xdf, (byte)0xa4, (byte)0xc5,
71      (byte)0x45, (byte)0x4e, (byte)0x34, (byte)0x33, (byte)0xe5,
72      (byte)0x43, (byte)0x46, (byte)0xc0, (byte)0x2b, (byte)0xbd,
73      (byte)0x85, (byte)0x2f, (byte)0xca, (byte)0xf8, (byte)0x9d,
74      (byte)0xb4, (byte)0xbc, (byte)0x67, (byte)0x92, (byte)0xd4,
75      (byte)0x33, (byte)0xfd, (byte)0xbd, (byte)0x82, (byte)0x9d,
76      (byte)0x62, (byte)0xfc, (byte)0xbb, (byte)0xd2, (byte)0xad,
77      (byte)0x05, (byte)0xa2, (byte)0xfc, (byte)0x2d, (byte)0xe3,
78      (byte)0x02, (byte)0xe2, (byte)0x41, (byte)0x9b, (byte)0x1f,
79      (byte)0xf8, (byte)0x87, (byte)0x15, (byte)0x89, (byte)0xfb,
80      (byte)0x53, (byte)0x99, (byte)0xb3, (byte)0xeb, (byte)0xdb,
81      (byte)0x01, (byte)0xaf, (byte)0x71, (byte)0xd2, (byte)0xf2,
82      (byte)0x73, (byte)0xb7, (byte)0x82, (byte)0x30, (byte)0x25,
83      (byte)0x04, (byte)0x29, (byte)0x2b, (byte)0xb9, (byte)0x92,
84      (byte)0x92, (byte)0x35, (byte)0x97, (byte)0x0e, (byte)0xb8,
85      (byte)0xf2, (byte)0xc6, (byte)0x2e, (byte)0xa7, (byte)0x2d,
86      (byte)0x0c, (byte)0x09, (byte)0x5e, (byte)0x07, (byte)0x06,
87      (byte)0x67, (byte)0xa0, (byte)0xdf, (byte)0x55, (byte)0x09,
88      (byte)0xfc, (byte)0xee, (byte)0x2b, (byte)0x13, (byte)0x1a,
89      (byte)0x2e, (byte)0x5d, (byte)0x0a, (byte)0xbb, (byte)0x45,
90      (byte)0x75, (byte)0xf4, (byte)0xd8, (byte)0xdc, (byte)0x2e,
91      (byte)0x99, (byte)0x2a, (byte)0x13, (byte)0xa1, (byte)0x1e,
92      (byte)0x99, (byte)0xfd, (byte)0xdc, (byte)0xcf, (byte)0xcc,
93      (byte)0x3f, (byte)0x42, (byte)0xf7, (byte)0x3d, (byte)0x73,
94      (byte)0xee, (byte)0xca, (byte)0x76, (byte)0xe4, (byte)0x75,
95      (byte)0xc4, (byte)0x21, (byte)0xd4, (byte)0x14, (byte)0x2e,
96      (byte)0x22, (byte)0x9c, (byte)0xce, (byte)0x10, (byte)0xaf,
97      (byte)0xa6, (byte)0x25, (byte)0xa0, (byte)0x01, (byte)0xb1,
```

```
98      (byte)0x82, (byte)0xba, (byte)0x4c, (byte)0xb2, (byte)0x66,
99      (byte)0x89, (byte)0x89, (byte)0x6b, (byte)0x06, (byte)0x15,
100     (byte)0xba, (byte)0x64, (byte)0xa3, (byte)0x73, (byte)0x88,
101     (byte)0x34, (byte)0x99, (byte)0x3e, (byte)0x75, (byte)0x24,
102     (byte)0xf4, (byte)0xba, (byte)0xb0, (byte)0x22, (byte)0x8f,
103     (byte)0xc3, (byte)0x44, (byte)0x74, (byte)0x0b, (byte)0x52,
104     (byte)0x96, (byte)0xc6, (byte)0x97, (byte)0x8b, (byte)0xf2,
105     (byte)0xe3, (byte)0xc1, (byte)0xaf, (byte)0x53, (byte)0x03,
106     (byte)0x51, (byte)0xa7, (byte)0x0d, (byte)0x42, (byte)0x6a,
107     (byte)0x20, (byte)0x03, (byte)0x31, (byte)0xb4, (byte)0xc9,
108     (byte)0xaa, (byte)0x9e, (byte)0xda, (byte)0x6f, (byte)0x7b,
109     (byte)0xb8, (byte)0x6d, (byte)0x54, (byte)0x57, (byte)0xa8,
110     (byte)0xed, (byte)0x51, (byte)0xa4, (byte)0x23, (byte)0x05,
111     (byte)0x0b, (byte)0xb3, (byte)0x90, (byte)0x42, (byte)0x38,
112     (byte)0xa8, (byte)0xbc, (byte)0xd5, (byte)0x2f, (byte)0x87,
113     (byte)0x82, (byte)0x5b, (byte)0xff, (byte)0xdb, (byte)0xba,
114     (byte)0x41, (byte)0x18, (byte)0xe0, (byte)0x4a, (byte)0x07,
115     (byte)0x04, (byte)0xe1, (byte)0x3c, (byte)0xd5, (byte)0xbf,};
116 byte[] tempSeed = {
117     (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
118     (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
119     (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
120     (byte)0x00};
121 private byte[] SignedDataTag =  {
122   (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x02};
123 short copyPointer = (short)0;
124 final static byte CLA = (byte)0xB0;
125 final static byte StartProtocol = (byte)0x40;
126 final static byte selftest = (byte)0xff;
127 final static short SW_CLASSNOTSUPPORTED = 0x6320;
128 final static short SW_ERROR_INS = 0x6300;
129 RandomData randomDataGen;
130 Cipher pkCipher;
131 byte[] receivingBuffer = null;
132 short bytesLeft = 0;
133 short readCount = 0;
134 short rCount = 0;
135 short signlength = 0;
136 MessageDigest SHA256;
137 AESKey cipherKey;
138 Cipher syCipher;
139 byte[] InitialisationVector =  {
140   (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89,
        (byte)0x99,
141     (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)
142     0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52};
143 Signature phSign;
144 PrngSHA256 myPrngHMAC;
145
146 private SelftestOffline() {
147   phSign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
```

```
148     phSCKeyPair = new KeyPair(KeyPair.ALG_RSA, KeyBuilder.LENGTH_RSA_512);
149     cipherKey = (AESKey)KeyBuilder.buildKey
150                         (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
151                         KeyBuilder.LENGTH_AES_128, false);
152     syCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
153                                   false);
154     myPrngHmac = new PrngSHA256();
155     pkCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
156     phSCKeyPair.genKeyPair();
157     SHA256 = MessageDigest.getInstance(MessageDigest.ALG_SHA_256,
158       false);
159   }
160   public static void install(byte bArray[], short bOffset, byte bLength)
161                               throws ISOException {
162     new SelftestOffline().register();
163   }
164
165   public void process(APDU apdu)throws ISOException {
166     byte[] apduBuffer = apdu.getBuffer();
167     if (selectingApplet()) {
168       this.initialise();
169       return ;
170     }
171     if (apduBuffer[ISO7816.OFFSET_CLA] != CLA) {
172       ISOException.throwIt(SW_CLASSNOTSUPPORTED);
173     }
174     if (apduBuffer[ISO7816.OFFSET_INS] == selftest) {
175       receivingBuffer = JCSystem.makeTransientByteArray((short)84,
176         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
177       generateResponse((short)1);
178       apdu.setOutgoing();
179       apdu.setOutgoingLength((short)copyPointer);
180       apdu.sendBytesLong(receivingBuffer, (short)0, (short)copyPointer);
181       return ;
182     }
183     JCSystem.requestObjectDeletion();
184   }
185
186   void selftestProcess(){
187    byte[] memoryWordRead = new byte[4];
188    byte rcount = (byte)0x00;
189    while (rcount <  MemoryContents.length){
190     Util.arrayCopyNonAtomic(MemoryContents,
191         rcount, memoryWordRead, (short)0, memoryWordRead.length);
192         generateSeed(memoryWordRead, tempSeed);
193         rcount += (byte)(rcount+(short)4);
194    }
195    if(seedZero()){
196          // PUF(tempSeed); // This is emulations
197         }else{
198          ISOException.throwIt((short)0xFA17);
```

```
199         }
200   }
201
202   boolean seedZero(){
203    for(short i=0; i<tempSeed.length; i++){
204     if(tempSeed[i]!=(byte)0x00){
205      return true;
206     }
207    }
208    return false;
209   }
210
211   private void generateResponse() {
212     copyPointer = 0;
213     selftestProcess();
214     phDecryption();
215     getSignatureKey();
216   }
217
218   void phDecryption(){
219     syCipher.init(phCipherKey, Cipher.MODE_DECRYPT, InitialisationVector,
220                   (short)0, (short)InitialisationVector.length);
221     syCipher.doFinal(signatureKey, inbuffOffset, inbuffLength,
          signatureKey,
222                      inbuffOffset);
223   }
224   void getSignatureKey(){
225    RSAPrivateKey myPrivate = (RSAPrivateKey)this.phSCKeyPair.getPrivate();
226       short kLen = myPrivate.getExponent(receivingBuffer, (short)
227         (copyPointer + (short)2));
228       this.shortToBytes(receivingBuffer, copyPointer, kLen);
229       copyPointer += (short)(kLen + (short)2);
230       receivingBuffer[6]++;
231       copyPointer = Util.arrayCopyNonAtomic(this.ModulusTag, (short)0,
232                      receivingBuffer, (short)(copyPointer), (short)
233                      this.ModulusTag.length);
234       kLen = myPrivate.getModulus(receivingBuffer, (short)
235                                   (copyPointer + (short)2));
236       this.shortToBytes(receivingBuffer, copyPointer, kLen);
237   }
238 }
```

C.2 Online Attestation Mechanism

In this section, we detail the Java Card implementation of the online attestation mechanism based on PRNG and PUF algorithms discussed in section 4.5.2 and 4.5.1, respectively.

C.2.1 Online PRNG Algorithm

The Java Card implementation of the offline PRNG algorithm discussed in section 4.5.2.

```
1  package selftestOnlinePRNG;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.JCSystem;
8  import javacard.framework.Util;
9  import javacard.security.AESKey;
10 import javacard.security.Key;
11 import javacard.security.KeyBuilder;
12 import javacard.security.KeyPair;
13 import javacard.security.MessageDigest;
14 import javacard.security.RSAPrivateKey;
15 import javacard.security.RSAPublicKey;
16 import javacard.security.RandomData;
17 import javacard.security.Signature;
18 import javacardx.apdu.ExtendedLength;
19 import javacardx.crypto.Cipher;
20 import javacard.security.MessageDigest;
21
22 public class SelftestOffline extends Applet implements ExtendedLength {
23   private static byte[] MemoryContents =  {
24     (byte)0x37, (byte)0x7a, (byte)0xbc, (byte)0xc0, (byte)0xea, (byte)
25       0x07, (byte)0x94, (byte)0x59, (byte)0xd6, (byte)0x37, (byte)0x6b,
26       (byte)0x4c, (byte)0x82, (byte)0xdb, (byte)0x54, (byte)0xb2,
27       (byte)0xe8, (byte)0xea, (byte)0x71, (byte)0xe1, (byte)0xa4,
28       (byte)0x41, (byte)0x06, (byte)0x44, (byte)0xfe, (byte)0x86,
29       (byte)0x8e, (byte)0x4f, (byte)0x39, (byte)0xf5, (byte)0xdb,
30       (byte)0xd1, (byte)0xf1, (byte)0xc5, (byte)0xd8, (byte)0xac,
31       (byte)0xbb, (byte)0x73, (byte)0x51, (byte)0xa1, (byte)0xa3,
32       (byte)0x8a, (byte)0x26, (byte)0x5d, (byte)0xf3, (byte)0x61,
33       (byte)0x55, (byte)0x56, (byte)0x39, (byte)0x3f, (byte)0x4c,
34       (byte)0x2a, (byte)0x43, (byte)0xc4, (byte)0xd7, (byte)0xa1,
35       (byte)0xaa, (byte)0xc1, (byte)0xf2, (byte)0xd6, (byte)0x07,
36       (byte)0xa8, (byte)0x58, (byte)0x9a, (byte)0x70, (byte)0x84,
37       (byte)0x15, (byte)0x19, (byte)0x56, (byte)0x61, (byte)0x3d,
38       (byte)0x88, (byte)0x2a, (byte)0x44, (byte)0x54, (byte)0x29,
39       (byte)0x29, (byte)0x26, (byte)0x36, (byte)0x06, (byte)0xfe,
40       (byte)0xad, (byte)0x27, (byte)0x13, (byte)0x86, (byte)0x0e,
41       (byte)0x85, (byte)0x3c, (byte)0x32, (byte)0xe2, (byte)0x38,
42       (byte)0xd2, (byte)0x91, (byte)0x82, (byte)0x89, (byte)0xce,
43       (byte)0x79, (byte)0x02, (byte)0x43, (byte)0xfd, (byte)0xaf,
44       (byte)0x18, (byte)0xe8, (byte)0x5b, (byte)0xd4, (byte)0x72,
45       (byte)0x03, (byte)0x63, (byte)0x2b, (byte)0x29, (byte)0x72,
46       (byte)0xe0, (byte)0x92, (byte)0x54, (byte)0x06, (byte)0x1c,
47       (byte)0x7f, (byte)0xc7, (byte)0x37, (byte)0x93, (byte)0x2f,
48       (byte)0x7a, (byte)0x84, (byte)0x95, (byte)0xec, (byte)0x5e,
```

```
49      (byte)0xa5, (byte)0xf6, (byte)0x4e, (byte)0x7e, (byte)0x1f,
50      (byte)0xe6, (byte)0xe2, (byte)0x04, (byte)0x2e, (byte)0x25,
51      (byte)0x7f, (byte)0x2f, (byte)0x3c, (byte)0xfe, (byte)0x57,
52      (byte)0x9e, (byte)0x7f, (byte)0xce, (byte)0x72, (byte)0xc0,
53      (byte)0xe9, (byte)0x79, (byte)0x05, (byte)0xc5, (byte)0xfd,
54      (byte)0x6a, (byte)0x46, (byte)0xfe, (byte)0x33, (byte)0x84,
55      (byte)0x3f, (byte)0x09, (byte)0xae, (byte)0x01, (byte)0x18,
56      (byte)0x5a, (byte)0xf6, (byte)0xc6, (byte)0xd3, (byte)0xa1,
57      (byte)0xe2, (byte)0x90, (byte)0x83, (byte)0x79, (byte)0xee,
58      (byte)0xa6, (byte)0xd4, (byte)0xf6, (byte)0xd1, (byte)0x86,
59      (byte)0x91, (byte)0x34, (byte)0x00, (byte)0xd3, (byte)0xe4,
60      (byte)0x8a, (byte)0xfb, (byte)0xaa, (byte)0x6c, (byte)0xe5,
61      (byte)0x46, (byte)0xa7, (byte)0x00, (byte)0x9e, (byte)0xd8,
62      (byte)0x81, (byte)0xbc, (byte)0xd1, (byte)0xb5, (byte)0x60,
63      (byte)0xd5, (byte)0x91, (byte)0x13, (byte)0x06, (byte)0x68,
64      (byte)0x21, (byte)0x8f, (byte)0x7d, (byte)0xc2, (byte)0x3e,
65      (byte)0xd2, (byte)0x75, (byte)0x0f, (byte)0x97, (byte)0x64,
66      (byte)0xb1, (byte)0xdb, (byte)0x74, (byte)0x6e, (byte)0x91,
67      (byte)0x6b, (byte)0xa7, (byte)0x7d, (byte)0xef, (byte)0x8b,
68      (byte)0x37, (byte)0xb7, (byte)0x84, (byte)0x1e, (byte)0xa7,
69      (byte)0x26, (byte)0x26, (byte)0xea, (byte)0xe9, (byte)0xb7,
70      (byte)0x5e, (byte)0x3f, (byte)0xdf, (byte)0xa4, (byte)0xc5,
71      (byte)0x45, (byte)0x4e, (byte)0x34, (byte)0x33, (byte)0xe5,
72      (byte)0x43, (byte)0x46, (byte)0xc0, (byte)0x2b, (byte)0xbd,
73      (byte)0x85, (byte)0x2f, (byte)0xca, (byte)0xf8, (byte)0x9d,
74      (byte)0xb4, (byte)0xbc, (byte)0x67, (byte)0x92, (byte)0xd4,
75      (byte)0x33, (byte)0xfd, (byte)0xbd, (byte)0x82, (byte)0x9d,
76      (byte)0x62, (byte)0xfc, (byte)0xbb, (byte)0xd2, (byte)0xad,
77      (byte)0x05, (byte)0xa2, (byte)0xfc, (byte)0x2d, (byte)0xe3,
78      (byte)0x02, (byte)0xe2, (byte)0x41, (byte)0x9b, (byte)0x1f,
79      (byte)0xf8, (byte)0x87, (byte)0x15, (byte)0x89, (byte)0xfb,
80      (byte)0x53, (byte)0x99, (byte)0xb3, (byte)0xeb, (byte)0xdb,
81      (byte)0x01, (byte)0xaf, (byte)0x71, (byte)0xd2, (byte)0xf2,
82      (byte)0x73, (byte)0xb7, (byte)0x82, (byte)0x30, (byte)0x25,
83      (byte)0x04, (byte)0x29, (byte)0x2b, (byte)0xb9, (byte)0x92,
84      (byte)0x92, (byte)0x35, (byte)0x97, (byte)0x0e, (byte)0xb8,
85      (byte)0xf2, (byte)0xc6, (byte)0x2e, (byte)0xa7, (byte)0x2d,
86      (byte)0x0c, (byte)0x09, (byte)0x5e, (byte)0x07, (byte)0x06,
87      (byte)0x67, (byte)0xa0, (byte)0xdf, (byte)0x55, (byte)0x09,
88      (byte)0xfc, (byte)0xee, (byte)0x2b, (byte)0x13, (byte)0x1a,
89      (byte)0x2e, (byte)0x5d, (byte)0x0a, (byte)0xbb, (byte)0x45,
90      (byte)0x75, (byte)0xf4, (byte)0xd8, (byte)0xdc, (byte)0x2e,
91      (byte)0x99, (byte)0x2a, (byte)0x13, (byte)0xa1, (byte)0x1e,
92      (byte)0x99, (byte)0xfd, (byte)0xdc, (byte)0xcf, (byte)0xcc,
93      (byte)0x3f, (byte)0x42, (byte)0xf7, (byte)0x3d, (byte)0x73,
94      (byte)0xee, (byte)0xca, (byte)0x76, (byte)0xe4, (byte)0x75,
95      (byte)0xc4, (byte)0x21, (byte)0xd4, (byte)0x14, (byte)0x2e,
96      (byte)0x22, (byte)0x9c, (byte)0xce, (byte)0x10, (byte)0xaf,
97      (byte)0xa6, (byte)0x25, (byte)0xa0, (byte)0x01, (byte)0xb1,
98      (byte)0x82, (byte)0xba, (byte)0x4c, (byte)0xb2, (byte)0x66,
99      (byte)0x89, (byte)0x89, (byte)0x6b, (byte)0x06, (byte)0x15,
```

```
100       (byte)0xba, (byte)0x64, (byte)0xa3, (byte)0x73, (byte)0x88,
101       (byte)0x34, (byte)0x99, (byte)0x3e, (byte)0x75, (byte)0x24,
102       (byte)0xf4, (byte)0xba, (byte)0xb0, (byte)0x22, (byte)0x8f,
103       (byte)0xc3, (byte)0x44, (byte)0x74, (byte)0x0b, (byte)0x52,
104       (byte)0x96, (byte)0xc6, (byte)0x97, (byte)0x8b, (byte)0xf2,
105       (byte)0xe3, (byte)0xc1, (byte)0xaf, (byte)0x53, (byte)0x03,
106       (byte)0x51, (byte)0xa7, (byte)0x0d, (byte)0x42, (byte)0x6a,
107       (byte)0x20, (byte)0x03, (byte)0x31, (byte)0xb4, (byte)0xc9,
108       (byte)0xaa, (byte)0x9e, (byte)0xda, (byte)0x6f, (byte)0x7b,
109       (byte)0xb8, (byte)0x6d, (byte)0x54, (byte)0x57, (byte)0xa8,
110       (byte)0xed, (byte)0x51, (byte)0xa4, (byte)0x23, (byte)0x05,
111       (byte)0x0b, (byte)0xb3, (byte)0x90, (byte)0x42, (byte)0x38,
112       (byte)0xa8, (byte)0xbc, (byte)0xd5, (byte)0x2f, (byte)0x87,
113       (byte)0x82, (byte)0x5b, (byte)0xff, (byte)0xdb, (byte)0xba,
114       (byte)0x41, (byte)0x18, (byte)0xe0, (byte)0x4a, (byte)0x07,
115       (byte)0x04, (byte)0xe1, (byte)0x3c, (byte)0xd5, (byte)0xbf,};
116   byte[] tempSeed = {
117       (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
118       (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
119       (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
120       (byte)0x00};
121   private byte[] SignedDataTag =  {
122     (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x02};
123   short copyPointer = (short)0;
124   final static byte CLA = (byte)0xB0;
125   final static byte StartProtocol = (byte)0x40;
126   final static byte selftest = (byte)0xff;
127   final static short SW_CLASSNOTSUPPORTED = 0x6320;
128   final static short SW_ERROR_INS = 0x6300;
129   RandomData randomDataGen;
130   Cipher pkCipher;
131   byte[] receivingBuffer = null;
132   short bytesLeft = 0;
133   short readCount = 0;
134   short rCount = 0;
135   short signlength = 0;
136   MessageDigest SHA128;
137   AESKey cipherKey;
138   Cipher syCipher;
139   byte[] InitialisationVector =  {
140     (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89,
          (byte)0x99,
141       (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)
142      0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52};
143   Signature phSign;
144   PrngSHA256 myPrngHMAC;
145
146   private SelftestOffline() {
147     phSign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
148     phSCKeyPair = new KeyPair(KeyPair.ALG_RSA, KeyBuilder.LENGTH_RSA_512);
149     cipherKey = (AESKey)KeyBuilder.buildKey
```

```
150                     (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
151                     KeyBuilder.LENGTH_AES_128, false);
152     syCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
153                                   false);
154     myPrngHmac = new PrngSHA256();
155     pkCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
156     phSCKeyPair.genKeyPair();
157     SHA128 = MessageDigest.getInstance(MessageDigest.ALG_SHA_128,
158       false);
159     byte[] responseBuffer = null;
160   }
161   public static void install(byte bArray[], short bOffset, byte bLength)
162                               throws ISOException {
163     new SelftestOffline().register();
164   }
165
166   public void process(APDU apdu)throws ISOException {
167     byte[] apduBuffer = apdu.getBuffer();
168     if (selectingApplet()) {
169       this.initialise();
170       return ;
171     }
172     if (apduBuffer[ISO7816.OFFSET_CLA] != CLA) {
173       ISOException.throwIt(SW_CLASSNOTSUPPORTED);
174     }
175     receivingBuffer = null;
176     bytesLeft = 0;
177     bytesLeft = apdu.getIncomingLength();
178     receivingBuffer = JCSystem.makeTransientByteArray(bytesLeft,
179       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
180     readCount = (short)((short)apdu.setIncomingAndReceive());
181     rCount = 0;
182     if (bytesLeft > 0) {
183       rCount = Util.arrayCopyNonAtomic(apduBuffer,
184         ISO7816.OFFSET_EXT_CDATA, receivingBuffer, rCount, readCount);
185       bytesLeft -= readCount;
186     }
187     while (bytesLeft > 0) {
188       try {
189         readCount = apdu.receiveBytes((short)0);
190         rCount = Util.arrayCopyNonAtomic(apduBuffer, (short)0,
191           receivingBuffer, rCount, readCount);
192         bytesLeft -= readCount;
193       } catch (Exception aE) {
194         ISOException.throwIt((short)0x7AAA);
195       }
196     }
197     responseBuffer = JCSystem.makeTransientByteArray((short)256,
198       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
199       selftestProcess();
200       JCSystem.requestObjectDeletion();
```

```
201         apdu.setOutgoing();
202         apdu.setOutgoingLength((short)responseBuffer.length);
203         apdu.sendBytesLong(responseBuffer, (short)0,
204                                 (short)responseBuffer.length);
205       JCSystem.requestObjectDeletion();
206     }
207
208     void selftestProcess(){
209      byte[] memoryWordRead = new byte[4];
210      byte[] hashValue = new byte[32]
211      byte rcount = (byte)0x00;
212      byte seedRef = (byte)0x00;
213      while (rcount <  receivingBuffer.length){
214       seedRef=(short)(receivingBuffer[rcount] %
215                         (short)(MemoryContents.length-1));
216       seedRef = (byte)(myPrngHMAC.generateRandom(seedRef).[0]
217                           % (MemoryContents.length-16))
218        Util.arrayCopyNonAtomic(MemoryContents,
219           seedRef, hashValue, (short)0, (short)16);
220        Util.arrayCopyNonAtomic(mK,
221           (short)0, hashValue, (short)16, (short)16);
222        SHA128.doFinal(hashValue, (short)0, (short)hashValue.length,
223                         hashValue, (short)0);
224        for(short i=0; i<responseBuffer.length;i++){
225         responseBuffer[i] = responseBuffer[i] ^ hashValue[i];
226        }
227           rcount++;
228      }
229     }
230 }
```

C.2.2 Online PUF Algorithm

The Java Card implementation that emulates the offline PUF algorithm discussed in section 4.5.1.

```
1  package selftestOnlinePUF;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.JCSystem;
8  import javacard.framework.Util;
9  import javacard.security.AESKey;
10 import javacard.security.Key;
11 import javacard.security.KeyBuilder;
12 import javacard.security.KeyPair;
13 import javacard.security.MessageDigest;
14 import javacard.security.RSAPrivateKey;
15 import javacard.security.RSAPublicKey;
```

```
16 import javacard.security.RandomData;
17 import javacard.security.Signature;
18 import javacardx.apdu.ExtendedLength;
19 import javacardx.crypto.Cipher;
20 import javacard.security.MessageDigest;
21
22 public class SelftestOffline extends Applet implements ExtendedLength {
23   private static byte[] MemoryContents =  {
24     (byte)0x37, (byte)0x7a, (byte)0xbc, (byte)0xc0, (byte)0xea, (byte)
25       0x07, (byte)0x94, (byte)0x59, (byte)0xd6, (byte)0x37, (byte)0x6b,
26       (byte)0x4c, (byte)0x82, (byte)0xdb, (byte)0x54, (byte)0xb2,
27       (byte)0xe8, (byte)0xea, (byte)0x71, (byte)0xe1, (byte)0xa4,
28       (byte)0x41, (byte)0x06, (byte)0x44, (byte)0xfe, (byte)0x86,
29       (byte)0x8e, (byte)0x4f, (byte)0x39, (byte)0xf5, (byte)0xdb,
30       (byte)0xd1, (byte)0xf1, (byte)0xc5, (byte)0xd8, (byte)0xac,
31       (byte)0xbb, (byte)0x73, (byte)0x51, (byte)0xa1, (byte)0xa3,
32       (byte)0x8a, (byte)0x26, (byte)0x5d, (byte)0xf3, (byte)0x61,
33       (byte)0x55, (byte)0x56, (byte)0x39, (byte)0x3f, (byte)0x4c,
34       (byte)0x2a, (byte)0x43, (byte)0xc4, (byte)0xd7, (byte)0xa1,
35       (byte)0xaa, (byte)0xc1, (byte)0xf2, (byte)0xd6, (byte)0x07,
36       (byte)0xa8, (byte)0x58, (byte)0x9a, (byte)0x70, (byte)0x84,
37       (byte)0x15, (byte)0x19, (byte)0x56, (byte)0x61, (byte)0x3d,
38       (byte)0x88, (byte)0x2a, (byte)0x44, (byte)0x54, (byte)0x29,
39       (byte)0x29, (byte)0x26, (byte)0x36, (byte)0x06, (byte)0xfe,
40       (byte)0xad, (byte)0x27, (byte)0x13, (byte)0x86, (byte)0x0e,
41       (byte)0x85, (byte)0x3c, (byte)0x32, (byte)0xe2, (byte)0x38,
42       (byte)0xd2, (byte)0x91, (byte)0x82, (byte)0x89, (byte)0xce,
43       (byte)0x79, (byte)0x02, (byte)0x43, (byte)0xfd, (byte)0xaf,
44       (byte)0x18, (byte)0xe8, (byte)0x5b, (byte)0xd4, (byte)0x72,
45       (byte)0x03, (byte)0x63, (byte)0x2b, (byte)0x29, (byte)0x72,
46       (byte)0xe0, (byte)0x92, (byte)0x54, (byte)0x06, (byte)0x1c,
47       (byte)0x7f, (byte)0xc7, (byte)0x37, (byte)0x93, (byte)0x2f,
48       (byte)0x7a, (byte)0x84, (byte)0x95, (byte)0xec, (byte)0x5e,
49       (byte)0xa5, (byte)0xf6, (byte)0x4e, (byte)0x7e, (byte)0x1f,
50       (byte)0xe6, (byte)0xe2, (byte)0x04, (byte)0x2e, (byte)0x25,
51       (byte)0x7f, (byte)0x2f, (byte)0x3c, (byte)0xfe, (byte)0x57,
52       (byte)0x9e, (byte)0x7f, (byte)0xce, (byte)0x72, (byte)0xc0,
53       (byte)0xe9, (byte)0x79, (byte)0x05, (byte)0xc5, (byte)0xfd,
54       (byte)0x6a, (byte)0x46, (byte)0xfe, (byte)0x33, (byte)0x84,
55       (byte)0x3f, (byte)0x09, (byte)0xae, (byte)0x01, (byte)0x18,
56       (byte)0x5a, (byte)0xf6, (byte)0xc6, (byte)0xd3, (byte)0xa1,
57       (byte)0xe2, (byte)0x90, (byte)0x83, (byte)0x79, (byte)0xee,
58       (byte)0xa6, (byte)0xd4, (byte)0xf6, (byte)0xd1, (byte)0x86,
59       (byte)0x91, (byte)0x34, (byte)0x00, (byte)0xd3, (byte)0xe4,
60       (byte)0x8a, (byte)0xfb, (byte)0xaa, (byte)0x6c, (byte)0xe5,
61       (byte)0x46, (byte)0xa7, (byte)0x00, (byte)0x9e, (byte)0xd8,
62       (byte)0x81, (byte)0xbc, (byte)0xd1, (byte)0xb5, (byte)0x60,
63       (byte)0xd5, (byte)0x91, (byte)0x13, (byte)0x06, (byte)0x68,
64       (byte)0x21, (byte)0x8f, (byte)0x7d, (byte)0xc2, (byte)0x3e,
65       (byte)0xd2, (byte)0x75, (byte)0x0f, (byte)0x97, (byte)0x64,
66       (byte)0xb1, (byte)0xdb, (byte)0x74, (byte)0x6e, (byte)0x91,
```

```
67      (byte)0x6b, (byte)0xa7, (byte)0x7d, (byte)0xef, (byte)0x8b,
68      (byte)0x37, (byte)0xb7, (byte)0x84, (byte)0x1e, (byte)0xa7,
69      (byte)0x26, (byte)0x26, (byte)0xea, (byte)0xe9, (byte)0xb7,
70      (byte)0x5e, (byte)0x3f, (byte)0xdf, (byte)0xa4, (byte)0xc5,
71      (byte)0x45, (byte)0x4e, (byte)0x34, (byte)0x33, (byte)0xe5,
72      (byte)0x43, (byte)0x46, (byte)0xc0, (byte)0x2b, (byte)0xbd,
73      (byte)0x85, (byte)0x2f, (byte)0xca, (byte)0xf8, (byte)0x9d,
74      (byte)0xb4, (byte)0xbc, (byte)0x67, (byte)0x92, (byte)0xd4,
75      (byte)0x33, (byte)0xfd, (byte)0xbd, (byte)0x82, (byte)0x9d,
76      (byte)0x62, (byte)0xfc, (byte)0xbb, (byte)0xd2, (byte)0xad,
77      (byte)0x05, (byte)0xa2, (byte)0xfc, (byte)0x2d, (byte)0xe3,
78      (byte)0x02, (byte)0xe2, (byte)0x41, (byte)0x9b, (byte)0x1f,
79      (byte)0xf8, (byte)0x87, (byte)0x15, (byte)0x89, (byte)0xfb,
80      (byte)0x53, (byte)0x99, (byte)0xb3, (byte)0xeb, (byte)0xdb,
81      (byte)0x01, (byte)0xaf, (byte)0x71, (byte)0xd2, (byte)0xf2,
82      (byte)0x73, (byte)0xb7, (byte)0x82, (byte)0x30, (byte)0x25,
83      (byte)0x04, (byte)0x29, (byte)0x2b, (byte)0xb9, (byte)0x92,
84      (byte)0x92, (byte)0x35, (byte)0x97, (byte)0x0e, (byte)0xb8,
85      (byte)0xf2, (byte)0xc6, (byte)0x2e, (byte)0xa7, (byte)0x2d,
86      (byte)0x0c, (byte)0x09, (byte)0x5e, (byte)0x07, (byte)0x06,
87      (byte)0x67, (byte)0xa0, (byte)0xdf, (byte)0x55, (byte)0x09,
88      (byte)0xfc, (byte)0xee, (byte)0x2b, (byte)0x13, (byte)0x1a,
89      (byte)0x2e, (byte)0x5d, (byte)0x0a, (byte)0xbb, (byte)0x45,
90      (byte)0x75, (byte)0xf4, (byte)0xd8, (byte)0xdc, (byte)0x2e,
91      (byte)0x99, (byte)0x2a, (byte)0x13, (byte)0xa1, (byte)0x1e,
92      (byte)0x99, (byte)0xfd, (byte)0xdc, (byte)0xcf, (byte)0xcc,
93      (byte)0x3f, (byte)0x42, (byte)0xf7, (byte)0x3d, (byte)0x73,
94      (byte)0xee, (byte)0xca, (byte)0x76, (byte)0xe4, (byte)0x75,
95      (byte)0xc4, (byte)0x21, (byte)0xd4, (byte)0x14, (byte)0x2e,
96      (byte)0x22, (byte)0x9c, (byte)0xce, (byte)0x10, (byte)0xaf,
97      (byte)0xa6, (byte)0x25, (byte)0xa0, (byte)0x01, (byte)0xb1,
98      (byte)0x82, (byte)0xba, (byte)0x4c, (byte)0xb2, (byte)0x66,
99      (byte)0x89, (byte)0x89, (byte)0x6b, (byte)0x06, (byte)0x15,
100     (byte)0xba, (byte)0x64, (byte)0xa3, (byte)0x73, (byte)0x88,
101     (byte)0x34, (byte)0x99, (byte)0x3e, (byte)0x75, (byte)0x24,
102     (byte)0xf4, (byte)0xba, (byte)0xb0, (byte)0x22, (byte)0x8f,
103     (byte)0xc3, (byte)0x44, (byte)0x74, (byte)0x0b, (byte)0x52,
104     (byte)0x96, (byte)0xc6, (byte)0x97, (byte)0x8b, (byte)0xf2,
105     (byte)0xe3, (byte)0xc1, (byte)0xaf, (byte)0x53, (byte)0x03,
106     (byte)0x51, (byte)0xa7, (byte)0x0d, (byte)0x42, (byte)0x6a,
107     (byte)0x20, (byte)0x03, (byte)0x31, (byte)0xb4, (byte)0xc9,
108     (byte)0xaa, (byte)0x9e, (byte)0xda, (byte)0x6f, (byte)0x7b,
109     (byte)0xb8, (byte)0x6d, (byte)0x54, (byte)0x57, (byte)0xa8,
110     (byte)0xed, (byte)0x51, (byte)0xa4, (byte)0x23, (byte)0x05,
111     (byte)0x0b, (byte)0xb3, (byte)0x90, (byte)0x42, (byte)0x38,
112     (byte)0xa8, (byte)0xbc, (byte)0xd5, (byte)0x2f, (byte)0x87,
113     (byte)0x82, (byte)0x5b, (byte)0xff, (byte)0xdb, (byte)0xba,
114     (byte)0x41, (byte)0x18, (byte)0xe0, (byte)0x4a, (byte)0x07,
115     (byte)0x04, (byte)0xe1, (byte)0x3c, (byte)0xd5, (byte)0xbf,};
116 byte[] tempSeed = {
117     (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
```

```
118       (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
119       (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
120       (byte)0x00};
121   private byte[] SignedDataTag = {
122     (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x02};
123   short copyPointer = (short)0;
124   final static byte CLA = (byte)0xB0;
125   final static byte StartProtocol = (byte)0x40;
126   final static byte selftest = (byte)0xff;
127   final static short SW_CLASSNOTSUPPORTED = 0x6320;
128   final static short SW_ERROR_INS = 0x6300;
129   RandomData randomDataGen;
130   Cipher pkCipher;
131   byte[] receivingBuffer = null;
132   byte[] challenge = null;
133   byte[] randomNumber = null;
134   short bytesLeft = 0;
135   short readCount = 0;
136   short rCount = 0;
137   short signlength = 0;
138   MessageDigest SHA128;
139   AESKey cipherKey;
140   Cipher syCipher;
141   byte[] InitialisationVector = {
142     (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89,
          (byte)0x99,
143      (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)
144      0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52};
145   Signature phSign;
146   PrngSHA256 myPrngHMAC;
147
148   private SelftestOffline() {
149     phSign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
150     phSCKeyPair = new KeyPair(KeyPair.ALG_RSA, KeyBuilder.LENGTH_RSA_512);
151     cipherKey = (AESKey)KeyBuilder.buildKey
152                       (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
153                       KeyBuilder.LENGTH_AES_128, false);
154     syCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
155                                   false);
156     myPrngHmac = new PrngSHA256();
157     pkCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
158     phSCKeyPair.genKeyPair();
159     SHA128 = MessageDigest.getInstance(MessageDigest.ALG_SHA_128,
160       false);
161     byte[] responseBuffer = null;
162   }
163   public static void install(byte bArray[], short bOffset, byte bLength)
164                               throws ISOException {
165     new SelftestOffline().register();
166   }
167
```

```
168   public void process(APDU apdu)throws ISOException {
169     byte[] apduBuffer = apdu.getBuffer();
170     if (selectingApplet()) {
171       this.initialise();
172       return ;
173     }
174     if (apduBuffer[ISO7816.OFFSET_CLA] != CLA) {
175       ISOException.throwIt(SW_CLASSNOTSUPPORTED);
176     }
177     receivingBuffer = null;
178     bytesLeft = 0;
179     bytesLeft = apdu.getIncomingLength();
180     receivingBuffer = JCSystem.makeTransientByteArray(bytesLeft,
181       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
182     readCount = (short)((short)apdu.setIncomingAndReceive());
183     rCount = 0;
184     if (bytesLeft > 0) {
185       rCount = Util.arrayCopyNonAtomic(apduBuffer,
186         ISO7816.OFFSET_EXT_CDATA, receivingBuffer, rCount, readCount);
187       bytesLeft -= readCount;
188     }
189     while (bytesLeft > 0) {
190       try {
191         readCount = apdu.receiveBytes((short)0);
192         rCount = Util.arrayCopyNonAtomic(apduBuffer, (short)0,
193           receivingBuffer, rCount, readCount);
194         bytesLeft -= readCount;
195       } catch (Exception aE) {
196         ISOException.throwIt((short)0x7AAA);
197       }
198     }
199       byte[] challenge = JCSystem.makeTransientByteArray((short)128,
200       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
201       byte[] randomnumber = JCSystem.makeTransientByteArray((short)128,
202       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
203       Util.arrayCopyNonAtomic(receivingBuffer,
204         (short)0, challenge, (short)0, (short)16);
205       Util.arrayCopyNonAtomic(receivingBuffer,
206         (short)16, randomNumber, (short)0, (short)16);
207       responseBuffer = JCSystem.makeTransientByteArray((short)128,
208       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
209       selftestProcess();
210       JCSystem.requestObjectDeletion();
211       apdu.setOutgoing();
212       apdu.setOutgoingLength((short)responseBuffer.length);
213       apdu.sendBytesLong(responseBuffer, (short)0,
214                          (short)responseBuffer.length);
215     JCSystem.requestObjectDeletion();
216   }
217
218   void selftestProcess(){
```

```
219     byte[] memoryWordRead = new byte[4];
220     byte[] hashValue = new byte[32]
221     byte rcount = (byte)0x00;
222     byte seedRef = (byte)0x00;
223     //mK = runPUF(challenge);
224     while (rcount <  randomNumber.length){
225      seedRef=(short)(randomNumber[rcount] %
226                          (short)(MemoryContents.length-1));
227      seedRef = (byte)(myPrngHMAC.generateRandom(seedRef).[0]
228                            % (MemoryContents.length-16))
229       Util.arrayCopyNonAtomic(MemoryContents,
230           seedRef, hashValue, (short)0, (short)16);
231       Util.arrayCopyNonAtomic(responseBuffer,
232           (short)0, hashValue, (short)16, (short)16);
233       Util.arrayCopyNonAtomic(mK,
234           (short)0, hashValue, (short)32, (short)16);
235       SHA128.doFinal(hashValue, (short)0, (short)hashValue.length,
236                          responseBuffer, (short)0);
237       if((short)(randomNumber.length-rcount)==1){
238        // mK = runPUF(responseBuffer);
239       }
240           rcount++;
241     }
242    }
243 }
```

C.3 Attestation Protocol

The Java Card implementation of the attestation protocol discussed in section 4.7 is listed in subsequent sections.

C.3.1 Smart Card Implementation

Following is the smart card implementation of the attestation protocol and this implementation uses the helper function discussed in appendix C.11.3.

```
1  package protocolAttestationSC;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.JCSystem;
8  import javacard.framework.Util;
9  import javacard.security.AESKey;
10 import javacard.security.Key;
11 import javacard.security.KeyBuilder;
12 import javacard.security.KeyPair;
13 import javacard.security.MessageDigest;
14 import javacard.security.RSAPrivateKey;
```

```
15 import javacard.security.RSAPublicKey;
16 import javacard.security.RandomData;
17 import javacard.security.Signature;
18 import javacardx.apdu.ExtendedLength;
19 import javacardx.crypto.Cipher;
20 public class ProtocolHandler extends Applet implements ExtendedLength {
21   private byte[] CMRandomNumberArray;
22   private byte[] CMCookieArray;
23   private byte[] SCRandomNumberArray;
24   private byte[] SCCertificate;
25   private byte[] SCCMDHGeneratedValue= {
26       (byte)0x98, (byte)0xD1, (byte)0x19, (byte)0x52, (byte)0x9A,
27       (byte)0x45, (byte)0xD6, (byte)0xF8, (byte)0x34, (byte)0x56,
28       (byte)0x6E, (byte)0x30, (byte)0x25, (byte)0xE3, (byte)0x16,
29       (byte)0xA3, (byte)0x30, (byte)0xEF, (byte)0xBB, (byte)0x77,
30       (byte)0xA8, (byte)0x6F, (byte)0x0C, (byte)0x1A, (byte)0xB1,
31       (byte)0x5B, (byte)0x05, (byte)0x1A, (byte)0xE3, (byte)0xD4,
32       (byte)0x28, (byte)0xC8, (byte)0xF8, (byte)0xAC, (byte)0xB7,
33       (byte)0x0A, (byte)0x81, (byte)0x37, (byte)0x15, (byte)0x0B,
34       (byte)0x8E, (byte)0xEB, (byte)0x10, (byte)0xE1, (byte)0x83,
35       (byte)0xED, (byte)0xD1, (byte)0x99, (byte)0x63, (byte)0xDD,
36       (byte)0xD9, (byte)0xE2, (byte)0x63, (byte)0xE4, (byte)0x77,
37       (byte)0x05, (byte)0x89, (byte)0xEF, (byte)0x6A, (byte)0xA2,
38       (byte)0x1E, (byte)0x7F, (byte)0x5F, (byte)0x2F, (byte)0xF3,
39       (byte)0x81, (byte)0xB5, (byte)0x39, (byte)0xCC, (byte)0xE3,
40       (byte)0x40, (byte)0x9D, (byte)0x13, (byte)0xCD, (byte)0x56,
41       (byte)0x6A, (byte)0xFB, (byte)0xB4
42   };
43   private byte[] MessageHandlerTagOne =  {
44     (byte)0x1F, (byte)0xC0, (byte)0xAA, (byte)0xAA, (byte)0x00,
         (byte)0x00,
45      (byte)0x00};
46   private byte[] MessageHandlerTagTwo =  {
47     (byte)0x1F, (byte)0xC0, (byte)0xBB, (byte)0xBB, (byte)0x00,
         (byte)0x00,
48      (byte)0x00};
49   private byte[] CMRandomNumberTag =  {
50     (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x01};
51   private byte[] CMCookieTag =  {
52     (byte)0x1F, (byte)0x5F, (byte)0x5B, (byte)0x01};
53   private byte[] EncryptedDataTag =  {
54     (byte)0x1F, (byte)0xC0, (byte)0xFE, (byte)0x01};
55   private byte[] SignedDataTag =  {
56     (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x02};
57   private byte[] MACedDataTag =  {
58     (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x01};
59   private byte[] SCIdentityTag =  {
60     (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x02, (byte)0x00,
         (byte)0x0C,
61      (byte)0x7A, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6, (byte)
62      0xC1, (byte)0x22, (byte)0x07, (byte)0xC9, (byte)0xF9, (byte)0x8D,
```

```
63      (byte)0x11};
64    private byte[] SCRandomNumberTag = {
65      (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x02};
66    private byte[] CMCertificateTag = {
67      (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x01};
68    private byte[] SCCertificateTag = {
69      (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x02};
70    private byte[] SCProtocolInitiatorTag = {
71      (byte)0x1F, (byte)0x5F, (byte)0xA1, (byte)0xB2};
72    short PTLVDataOffset = (short)6;
73    short CTLVDataOffset = (short)7;
74    short TLVLengthOffset = (short)4;
75    short copyPointer = (short)0;
76    final static byte CLA = (byte)0xB0;
77    final static byte StartProtocol = (byte)0x40;
78    final static byte InitiationProtocol = (byte)0xff;
79    final static short SW_CLASSNOTSUPPORTED = 0x6320;
80    final static short SW_ERROR_INS = 0x6300;
81    RandomData randomDataGen;
82    Cipher pkCipher;
83    short messageNumber = 0;
84    byte[] receivingBuffer = null;
85    short bytesLeft = 0;
86    short readCount = 0;
87    short rCount = 0;
88    short signlength = 0;
89    private RSAPublicKey dhKey = (RSAPublicKey)KeyBuilder.buildKey
90                                  (KeyBuilder.TYPE_RSA_PUBLIC,
91                                  KeyBuilder.LENGTH_RSA_2048, false);
92    private byte[] randomExponent;
93    final static byte GEN_KEYCONTRIBUTION = 0x01;
94    final static byte GEN_DHKEY = 0x02;
95    AESKey phCipherKey;
96    Cipher syCipher;
97    byte[] InitialisationVector = {
98      (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89,
          (byte)0x99,
99       (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)
100      0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52};
101   AESKey phMacGeneratorKey;
102   Signature phMacGenerator;
103   Signature phSign;
104   KeyPair phSCKeyPair;
105   KeyPair phUserKeyPair;
106   RSAPublicKey CMVerificationKey = null;
107   private ProtocolHandler() {
108     phMacGeneratorKey = (AESKey)KeyBuilder.buildKey
109                          (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
110                          KeyBuilder.LENGTH_AES_128, false);
111     phMacGenerator =
          Signature.getInstance(Signature.ALG_AES_MAC_128_NOPAD,
```

```
112         false);
113       phSign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
114       phCipherKey = (AESKey)KeyBuilder.buildKey
115                       (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
116                       KeyBuilder.LENGTH_AES_128, false);
117       syCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
118                                     false);
119       randomDataGen = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);
120       pkCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
121       dhInitialisation();
122     }
123     public static void install(byte bArray[], short bOffset, byte bLength)
124                                throws ISOException {
125       new ProtocolHandler().register();
126     }
127     public void initialiseProtocol() {
128       short initialPointer = 0;
129       CMRandomNumberArray = JCSystem.makeTransientByteArray((short)22,
130         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
131       CMCookieArray = JCSystem.makeTransientByteArray((short)22,
132         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
133       SCRandomNumberArray = JCSystem.makeTransientByteArray((short)22,
134         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
135       Util.arrayCopyNonAtomic(this.SCRandomNumberTag, (short)initialPointer,
136                               this.SCRandomNumberArray, (short)
137                               initialPointer, (short)
138                               this.SCRandomNumberTag.length);
139       this.shortToBytes(this.SCRandomNumberArray, (short)4, (short)((short)
140                         this.SCRandomNumberArray.length - (short)
141                         PTLVDataOffset));
142       try {
143         CMVerificationKey = (RSAPublicKey)KeyBuilder.buildKey
144                               (KeyBuilder.TYPE_RSA_PUBLIC,
145                               KeyBuilder.LENGTH_RSA_512, false);
146       } catch (Exception cE) {
147         ISOException.throwIt((short)0x6666);
148       }
149     }
150     public void process(APDU apdu)throws ISOException {
151       byte[] apduBuffer = apdu.getBuffer();
152       if (selectingApplet()) {
153         this.initialiseProtocol();
154         return;
155       }
156       if (apduBuffer[ISO7816.OFFSET_CLA] != CLA) {
157         ISOException.throwIt(SW_CLASSNOTSUPPORTED);
158       }
159       if (apduBuffer[ISO7816.OFFSET_INS] == InitiationProtocol) {
160         receivingBuffer = JCSystem.makeTransientByteArray((short)64,
161           JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
162         generateResponse((short)1);
```

```
163       apdu.setOutgoing();
164       apdu.setOutgoingLength((short)copyPointer);
165       apdu.sendBytesLong(receivingBuffer, (short)0, (short)copyPointer);
166       return ;
167     }
168     receivingBuffer = null;
169     bytesLeft = 0;
170     bytesLeft = apdu.getIncomingLength();
171     receivingBuffer = JCSystem.makeTransientByteArray(bytesLeft,
172       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
173     readCount = (short)((short)apdu.setIncomingAndReceive());
174     rCount = 0;
175     if (bytesLeft > 0) {
176       rCount = Util.arrayCopyNonAtomic(apduBuffer,
177         ISO7816.OFFSET_EXT_CDATA, receivingBuffer, rCount, readCount);
178       bytesLeft -= readCount;
179     }
180     while (bytesLeft > 0) {
181       try {
182         readCount = apdu.receiveBytes((short)0);
183         rCount = Util.arrayCopyNonAtomic(apduBuffer, (short)0,
184           receivingBuffer, rCount, readCount);
185         bytesLeft -= readCount;
186       } catch (Exception aE) {
187         ISOException.throwIt((short)0x7AAA);
188       }
189     }
190     if (this.receivingBuffer[3] == this.MessageHandlerTagOne[3]) {
191       try {
192         parseMessage(receivingBuffer);
193       } catch (Exception cE) {
194         ISOException.throwIt((short)0xA112);
195       }
196       receivingBuffer = JCSystem.makeTransientByteArray((short)600,
197         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
198       generateResponse((short)2);
199       JCSystem.requestObjectDeletion();
200       apdu.setOutgoing();
201       apdu.setOutgoingLength((short)copyPointer);
202       apdu.sendBytesLong(receivingBuffer, (short)0, (short)copyPointer);
203     } else if (this.receivingBuffer[3] == this.MessageHandlerTagTwo[3]) {
204       if (processSecondMsg(receivingBuffer)) {
205         return ;
206       } else {
207         ISOException.throwIt((short)0xFA17);
208       }
209       return ;
210     } else {
211       ISOException.throwIt(ProtocolHandler.SW_ERROR_INS);
212     }
213     JCSystem.requestObjectDeletion();
```

```
214   }
215   private void generateResponse(short msgNumber) {
216     short childPM1 = 0;
217     short childPM2 = 0;
218     copyPointer = 0;
219     if (msgNumber == 1) {
220       copyPointer = Util.arrayCopy(this.SCProtocolInitiatorTag, (short)0,
221                                    this.receivingBuffer, copyPointer,
222                                    (short)
223                                    this.SCProtocolInitiatorTag.length);
224       randomDataGen.generateData(this.SCRandomNumberArray,
225                                  this.PTLVDataOffset, (short)16);
226       childPM1 = copyPointer;
227       copyPointer += 2;
228       phMacGeneratorKey.setKey(this.SCRandomNumberArray,
229                                this.PTLVDataOffset);
230       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
231                           InitialisationVector, (short)0, (short)
232                           InitialisationVector.length);
233       return ;
234     } else if (msgNumber == 2) {
235       keygenerator();
236       childPM1 = (short)6;
237       copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagTwo,
238         (short)0, this.receivingBuffer, copyPointer, (short)
239         this.MessageHandlerTagTwo.length);
240        copyPointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
241         (short)0, this.receivingBuffer, copyPointer, (short)
242         this.SCRandomNumberArray.length);
243       this.receivingBuffer[childPM1]++;
244       copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag, (short)
245         0, this.receivingBuffer, copyPointer, (short)
246         this.EncryptedDataTag.length);
247       copyPointer += 3;
248       childPM2 = (short)(copyPointer - (short)1);
249       this.receivingBuffer[childPM1]++;
250       MessageDigest myHashGen = MessageDigest.getInstance
251         (MessageDigest.ALG_SHA_256, false);
252       short tempLength = (short)myHashGen.doFinal(this.ClassDH.dhModulus,
            (short)0,
253                           (short)this.ClassDH.dhModulus.length,
                                receivingBuffer,
254                           copyPointer);
255       this.receivingBuffer[childPM2]++;
256       this.shortToBytes(this.receivingBuffer, (short)(copyPointer -
            (short)
257                         2), tempLength);
258       copyPointer += tempLength;
259       copyPointer = Util.arrayCopyNonAtomic(this.SCIdentityTag, (short)0,
260         this.receivingBuffer, copyPointer, (short)
261         this.SCIdentityTag.length);
```

```
262       this.receivingBuffer[childPM2]++;
263       copyPointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
264         (short)0, this.receivingBuffer, copyPointer, (short)
265         this.SCRandomNumberArray.length);
266       this.receivingBuffer[childPM2]++;
267       copyPointer = Util.arrayCopyNonAtomic(this.CMRandomNumberArray,
268         (short)0, this.receivingBuffer, copyPointer, (short)
269         this.CMRandomNumberArray.length);
270       this.receivingBuffer[childPM2]++;
271       try {
272         this.signGenerate(this.receivingBuffer, (short)(childPM2 + (short)
273                           1), (short)(copyPointer - (short)(childPM2 +
274                           (short)1)), this.phSCKeyPair.getPrivate(),
275                           Signature.MODE_SIGN);
276       } catch (Exception cE) {
277         ISOException.throwIt((short)0x3141);
278       }
279       this.receivingBuffer[childPM2]++;
280       copyPointer = Util.arrayCopyNonAtomic(this.SCCertificate, (short)0,
281         this.receivingBuffer, copyPointer, (short)
282         this.SCCertificate.length);
283       this.receivingBuffer[childPM2]++;
284       try {
285         this.messageEncryption(this.receivingBuffer, (short)(childPM2 +
286                                (short)1), (short)(copyPointer - (short)
287                                (childPM2 + (short)1)));
288       } catch (Exception ce) {
289         ISOException.throwIt((short)(copyPointer - (short)(childPM2 +
290                              (short)1)));
291       }
292       this.macGenerate(this.receivingBuffer, (short)(childPM2 + (short)1),
293                        (short)(copyPointer - (short)(childPM2 +
                             (short)1)),
294                        Signature.MODE_SIGN);
295       this.receivingBuffer[childPM1]++;
296       copyPointer = Util.arrayCopyNonAtomic(this.CMCookieArray, (short)0,
297                        this.receivingBuffer, (short)copyPointer, (short)
298                        this.CMCookieArray.length);
299       this.receivingBuffer[childPM1]++;
300     }
301   }
302   boolean processSecondMsg(byte[] inArray) {
303     short inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
304     short inLength = (short)(ProtocolHandler.bytesToShort(inArray, (short)
305                      (inOffset - (short)3)));
306     if (this.macGenerate(inArray, inOffset, inLength,
307         Signature.MODE_VERIFY)) {
308       try {
309         this.phDecryption(inArray, inOffset, inLength);
310         inOffset = (short)(this.CTLVDataOffset + this.PTLVDataOffset +
311                    (short)168);
```

```
312         inLength = 3;
313         CMVerificationKey.setExponent(inArray, inOffset, inLength);
314         inOffset += (short)(inLength + this.PTLVDataOffset);
315         inLength = (short)64;
316         CMVerificationKey.setModulus(inArray, inOffset, inLength);
317         inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
318         inLength = (short)84;
319         if (this.signGenerate(inArray, inOffset, inLength,
320             CMVerificationKey, Signature.MODE_VERIFY)) {
321           return true;
322         } else {
323           ISOException.throwIt((short)0x6666);
324         }
325       } catch (Exception ce) {
326         ISOException.throwIt((short)0xAB23);
327       }
328       return true;
329     } else {
330       ISOException.throwIt((short)0xFA18);
331     }
332     return false;
333   }
334   void parseMessage(byte[] inBuffer) {
335     byte childLeft = inBuffer[(short)(this.CTLVDataOffset - (short)1)];
336     short pointer = (short)this.CTLVDataOffset;
337     try {
338       while (childLeft > 0) {
339         if (Util.arrayCompare(CMDHChallengeTag, (short)0, inBuffer,
340             pointer, (short)4) == 0) {
341           Util.arrayCopy(inBuffer, pointer, this.CMDHChanllengerArray,
342                          (short)0,
343                              (short)this.CMDHChanllengerArray.length)
                               ;
344           pointer += (short)this.CMDHChanllengerArray.length;
345         } else if (Util.arrayCompare(this.CMRandomNumberTag, (short)0,
346                    inBuffer, pointer, (short)4) == 0) {
347           Util.arrayCopyNonAtomic(inBuffer, pointer,
348                                   this.CMRandomNumberArray, (short)0,
349                                   (short)(this.CMRandomNumberArray.length))
                                    ;
350           pointer += (short)(this.CMRandomNumberArray.length);
351         } else if (Util.arrayCompare(this.CMCookieTag, (short)0, inBuffer,
352                    pointer, (short)4) == 0) {
353           Util.arrayCopyNonAtomic(inBuffer, pointer, this.CMCookieArray,
354                                   (short)0, (short)
355                                   (this.CMCookieArray.length));
356           pointer += (short)(this.CMCookieArray.length);
357         }
358         childLeft -= (short)1;
359       }
360     } catch (Exception cE) {
361
```

```
362         ISOException.throwIt((short)childLeft);
363       }
364     }
365     void protocolImplementation(){}
366     void dhInitialisation() {
367       dhKey.setModulus(ClassDH.dhModulus, (short)0,
              (short)ClassDH.dhModulus.length);
368     }
369     void keygenerator() {
370       AESKey sessionGenKey = (AESKey)KeyBuilder.buildKey
371                                 (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
372                                 KeyBuilder.LENGTH_AES_128, false);
373       sessionGenKey.setKey(SCCMDHGeneratedValue, (short)0);
374       phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
375                           InitialisationVector, (short)0, (short)
376                           InitialisationVector.length);
377       byte[] keyGenMacData = JCSystem.makeTransientByteArray((short)64,
378         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
379       short pointer = 0;
380       pointer = Util.arrayCopyNonAtomic(this.CMRandomNumberArray,
381         this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
382       pointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
383         this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
384       pointer = Util.arrayCopyNonAtomic(SCCMDHGeneratedValue, (short)16,
385         keyGenMacData, (short)pointer, (short)16);
386       for (short i = 48; i < 64; i++) {
387         keyGenMacData[i] = (byte)0x02;
388       }
389       phMacGenerator.sign(keyGenMacData, (short)0, (short)
390                           keyGenMacData.length, SCCMDHGeneratedValue,
                                (short)
391                           0);
392       this.phCipherKey.setKey(SCCMDHGeneratedValue, (short)0);
393       for (short i = 48; i < 64; i++) {
394         keyGenMacData[i] = (byte)0x03;
395       }
396       phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
397                           InitialisationVector, (short)0, (short)
398                           InitialisationVector.length);
399       phMacGenerator.sign(keyGenMacData, (short)0, (short)
400                           keyGenMacData.length, SCCMDHGeneratedValue,
                                (short)
401                           0);
402       this.phMacGeneratorKey.setKey(SCCMDHGeneratedValue, (short)0);
403       SCCMDHGeneratedValue = null;
404       JCSystem.requestObjectDeletion();
405     }
406     void messageEncryption(byte[] inbuff, short inbuffOffset, short
407                            inbuffLength) {
408       syCipher.init(phCipherKey, Cipher.MODE_ENCRYPT, InitialisationVector,
409                     (short)0, (short)InitialisationVector.length);
```

```
410     this.shortToBytes(inbuff, (short)(inbuffOffset - 3), (short)
411                       syCipher.doFinal(inbuff, inbuffOffset, inbuffLength,
412                       inbuff, inbuffOffset));
413   }
414   void phDecryption(byte[] inbuff, short inbuffOffset, short inbuffLength)
415                     {
416     syCipher.init(phCipherKey, Cipher.MODE_DECRYPT, InitialisationVector,
417                   (short)0, (short)InitialisationVector.length);
418     syCipher.doFinal(inbuff, inbuffOffset, inbuffLength, inbuff,
419                      inbuffOffset);
420   }
421   boolean macGenerate(byte[] inbuff, short inbuffOffset, short
422                       inbuffLength, short macMode) {
423     if (macMode == Signature.MODE_SIGN) {
424       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
425                           InitialisationVector, (short)0, (short)
426                           InitialisationVector.length);
427       try {
428         copyPointer = Util.arrayCopyNonAtomic(this.MACedDataTag, (short)0,
429           this.receivingBuffer, copyPointer, (short)
430           this.MACedDataTag.length);
431         copyPointer += 2;
432       } catch (Exception ce) {
433         ISOException.throwIt((short)0xFA17);
434       }
435       try {
436         short length = (short)phMacGenerator.sign(this.receivingBuffer,
437                         inbuffOffset, inbuffLength, inbuff, copyPointer);
438         this.shortToBytes(inbuff, (short)(copyPointer - (short)2),
                length);
439         copyPointer += length;
440       } catch (Exception ce) {
441         ISOException.throwIt((short)0x0987);
442       }
443       return true;
444     } else if (macMode == Signature.MODE_VERIFY) {
445       try {
446         phMacGenerator.init(phMacGeneratorKey, Signature.MODE_VERIFY,
447                             InitialisationVector, (short)0, (short)
448                             InitialisationVector.length);
449         return phMacGenerator.verify(this.receivingBuffer, inbuffOffset,
450                                      inbuffLength, inbuff, (short)
451                                      (inbuffOffset + inbuffLength +
452                                      this.PTLVDataOffset), (short)16);
453       } catch (Exception cE) {
454         ISOException.throwIt((short)0xC1C2);
455       }
456     }
457     return false;
458   }
459   boolean signGenerate(byte[] inbuff, short inbuffOffset, short
```

```
460                         inbufflength, Key kpSign, short signMode) {
461     if (signMode == Signature.MODE_SIGN) {
462       copyPointer = Util.arrayCopyNonAtomic(this.SignedDataTag, (short)0,
463         this.receivingBuffer, copyPointer, (short)
464         this.SignedDataTag.length);
465       copyPointer += (short)2;
466       phSign.init((RSAPrivateKey)kpSign, Signature.MODE_SIGN);
467       signlength = phSign.sign(inbuff, (short)inbuffOffset, inbufflength,
468                                inbuff, copyPointer);
469       this.shortToBytes(this.receivingBuffer, (short)(copyPointer -
              (short)
470                           2), signlength);
471       copyPointer += signlength;
472       return true;
473     } else if (signMode == Signature.MODE_VERIFY) {
474       phSign.init((RSAPublicKey)kpSign, Signature.MODE_VERIFY);
475       return phSign.verify(inbuff, inbuffOffset, inbufflength, inbuff,
476                            (short)(inbuffOffset + inbufflength +
477                            this.PTLVDataOffset), (short)64);
478     }
479     return false;
480   }
481   public static short bytesToShort(byte[] ArrayBytes) {
482     return (short)(((ArrayBytes[0] << 8)) | ((ArrayBytes[1] & 0xff)));
483   }
484   public static short bytesToShort(byte[] ArrayBytes, short arrayOffset) {
485     return (short)(((ArrayBytes[arrayOffset] << 8)) | ((ArrayBytes[(short)
486              (arrayOffset + (short)1)] & 0xff)));
487   }
488   private void shortToBytes(byte[] Array, short arrayOffset, short
          inShort)
489                             {
490     Array[arrayOffset] = (byte)((short)(inShort & (short)0xFF00) >>
          (short)
491                           0x0008);
492     Array[(short)(arrayOffset + (short)1)] = (byte)(inShort & (short)
493           0x00FF);
494   }
495 }
```

C.3.2 Card Manufacturer Implementation

Following is the card manufacturer's implementation of the attestation protocol and to accomplish its operations it uses helper functions detailed in appendices C.11.1 and C.11.2.

```
1 package javacardterminal;
2
3 import java.util.Arrays;
4 import java.security.interfaces.RSAPublicKey;
5 import java.security.spec.RSAPublicKeySpec;
6 import java.security.*;
```

```
7  import java.math.BigInteger;
8  public class ProtocolHandlerAttestation {
9    private byte[] CMIdentity = {
10     (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)0x7C, (byte)0x62,
           (byte)0x0A,
11     (byte)0x86, (byte)0x52, (byte)0xBE, (byte)0x5E, (byte)0x90, (byte)
12     0x01, (byte)0xA8, (byte)0xD6, (byte)0x6A, (byte)0xD7};
13   private byte[] SCIP = {
14     (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)0x7C};
15   private byte[] PlatformHashPreset = {
16     (byte)0xBF, (byte)0xE5, (byte)0x45, (byte)0x86, (byte)0x2C,
           (byte)0xA1,
17     (byte)0x02, (byte)0xAD, (byte)0x1E, (byte)0xED, (byte)0xDB, (byte)
18     0x5F, (byte)0xBF, (byte)0xA5, (byte)0xBF, (byte)0x85, (byte)0x5A,
19     (byte)0xC4, (byte)0x99, (byte)0x5C, (byte)0x56, (byte)0xA8, (byte)
20     0xB4, (byte)0x08, (byte)0xCE, (byte)0x3F, (byte)0xE0, (byte)0x99,
21     (byte)0xDC, (byte)0xE9, (byte)0x3A, (byte)0x9D};
22   private byte[] SCDHStore = {
23     (byte)0x98, (byte)0xD1, (byte)0x19, (byte)0x52, (byte)0x9A,
24     (byte)0x45, (byte)0xD6, (byte)0xF8, (byte)0x34, (byte)0x56,
25     (byte)0x6E, (byte)0x30, (byte)0x25, (byte)0xE3, (byte)0x16,
26     (byte)0xA3, (byte)0x30, (byte)0xEF, (byte)0xBB, (byte)0x77,
27     (byte)0xA8, (byte)0x6F, (byte)0x0C, (byte)0x1A, (byte)0xB1,
28     (byte)0x5B, (byte)0x05, (byte)0x1A, (byte)0xE3, (byte)0xD4,
29     (byte)0x28, (byte)0xC8, (byte)0xF8, (byte)0xAC, (byte)0xB7,
30     (byte)0x0A, (byte)0x81, (byte)0x37, (byte)0x15, (byte)0x0B,
31     (byte)0x8E, (byte)0xEB, (byte)0x10, (byte)0xE1, (byte)0x83,
32     (byte)0xED, (byte)0xD1, (byte)0x99, (byte)0x63, (byte)0xDD,
33     (byte)0xD9, (byte)0xE2, (byte)0x63, (byte)0xE4, (byte)0x77,
34     (byte)0x05, (byte)0x89, (byte)0xEF, (byte)0x6A, (byte)0xA2,
35     (byte)0x1E, (byte)0x7F, (byte)0x5F, (byte)0x2F, (byte)0xF3,
36     (byte)0x81, (byte)0xB5, (byte)0x39, (byte)0xCC, (byte)0xE3,
37     (byte)0x40, (byte)0x9D, (byte)0x13, (byte)0xCD, (byte)0x56,
38     (byte)0x6A, (byte)0xFB, (byte)0xB4
39   };
40   private byte[] MessageHandlerTagOne = {(byte)0xAA, (byte)0xAA};
41   private byte[] MessageHandlerTagTwo = {(byte)0xBB, (byte)0xBB};
42   private byte[] CMIdentityTag = {(byte)0x5F, (byte)0x01};
43   private byte[] CMSignatureCertTag = {(byte)0xF0, (byte)0xF01};
44   private byte[] CMSigVerificationKeyTag = {(byte)0x51, (byte)0x01};
45   private byte[] CMRandomNumberTag = {(byte)0x5A, (byte)0x01};
46   private byte[] CMCookieTag = {(byte)0x5B, (byte)0x01};
47   private byte[] EncryptedDataTag = {(byte)0xFE, (byte)0x01};
48   private byte[] MACedDataTag = {(byte)0x5D, (byte)0x01};
49   private byte[] SignedDataTag = {(byte)0x5D, (byte)0x02};
50   private byte[] PublicExponentTag = {(byte)0xEE, (byte)0x01};
51   private byte[] PublicModulusTag = {(byte)0xEE, (byte)0x02};
52   private byte[] SCRandomNumberTag = {(byte)0x5A, (byte)0x02};
53   private byte[] SCIdentityTag = {(byte)0x5F, (byte)0x02};
54   private byte[] SCCertificateTag = {(byte)0xF0, (byte)0x02};
55   private byte[] PlatformHashTag = {(byte)0x5E, (byte)0xAF};
```

```
56  private byte[] UserIdentityTag =  {(byte)0x5F, (byte)0x03};
57  private byte[] SCProtocolInitiatorTag =  {(byte)0xA1, (byte)0xB2};
58  public ConstructedTLV MessageHandler = ConstructedTLV.getConstructedTLV
59    (MessageHandlerTagOne);
60  private ConstructedTLV CMSignatureCertificate =
61    ConstructedTLV.getConstructedTLV(CMSignatureCertTag);
62  private PrimitiveTLV CMIdentityTLV = PrimitiveTLV.getPrimitiveTLV
63    (CMIdentityTag, CMIdentity);
64  private PrimitiveTLV CMSigVerificationKey = PrimitiveTLV.getPrimitiveTLV
65    (this.CMSigVerificationKeyTag);
66  private PrimitiveTLV CMRandomNumber = PrimitiveTLV.getPrimitiveTLV
67    (this.CMRandomNumberTag);
68  private PrimitiveTLV CMCookie = PrimitiveTLV.getPrimitiveTLV
69    (this.CMCookieTag);
70  private ConstructedTLV EncryptedData = ConstructedTLV.getConstructedTLV
71    (this.EncryptedDataTag);
72  private PrimitiveTLV MACedData = PrimitiveTLV.getPrimitiveTLV
73    (this.MACedDataTag);
74  private PrimitiveTLV SignedData = PrimitiveTLV.getPrimitiveTLV
75    (this.SignedDataTag);
76  private PrimitiveTLV PublicExponent = PrimitiveTLV.getPrimitiveTLV
77    (this.PublicExponentTag);
78  private PrimitiveTLV PublicModulus = PrimitiveTLV.getPrimitiveTLV
79    (this.PublicModulusTag);
80  private PrimitiveTLV SCRandomNumber = PrimitiveTLV.getPrimitiveTLV
81    (this.SCRandomNumberTag);
82  private PrimitiveTLV SCIdentity = PrimitiveTLV.getPrimitiveTLV
83    (SCIdentityTag);
84  private ConstructedTLV SCUserCertificate =
85    ConstructedTLV.getConstructedTLV(this.SCUserCertificateTag);
86  private ConstructedTLV SCCertificate = ConstructedTLV.getConstructedTLV
87    (this.SCCertificateTag);
88  private PrimitiveTLV PlatformHash = PrimitiveTLV.getPrimitiveTLV
89    (this.PlatformHashTag);
90  private PrimitiveTLV UserIdentity = PrimitiveTLV.getPrimitiveTLV
91    (this.UserIdentityTag);
92  private PrimitiveTLV SCProtocolInitiator = PrimitiveTLV.getPrimitiveTLV
93    (this.SCProtocolInitiatorTag);
94  private ProtocolHelperClass myProtocolHelperObject = new
95    ProtocolHelperClass();
96  private byte[] mySessionEncryptionKey = new byte[16];
97  private byte[] mySessionMacKey = new byte[16];
98  private PublicKey SCUserVerificationKey = null;
99  private PublicKey SCVerificationKey = null;
100 public ProtocolHandlerAttestation() {
101   myProtocolHelperObject.protocolInitialise();
102   RSAPublicKey tempKey = (RSAPublicKey)
103                          myProtocolHelperObject.getPublicKey();
104   byte[] tempExponent = tempKey.getPublicExponent().toByteArray();
105   this.PublicExponent.initialisationPTLV(this.PublicExponentTag,
106     tempExponent.length);
```

```
107     this.PublicExponent.setTlvValues(tempExponent);
108     byte[] tempModulus = tempKey.getModulus().toByteArray();
109     this.PublicModulus.initialisationPTLV(this.PublicModulusTag,
110       (tempModulus.length - 1));
111     this.PublicModulus.setTlvValues(tempModulus, 1, (tempModulus.length -
112                                         1));
113     CMSignatureCertificate.addPTLV(this.PublicExponent);
114     CMSignatureCertificate.addPTLV(this.PublicModulus);
115   }
116   public byte[] outMessageProcessing(int Counter) {
117     if (Counter == 1) {
118       try {
119         this.CMRandomNumber.setTlvValues
120           (this.myProtocolHelperObject.getRandomNumber());
121         this.MessageHandler.addPTLV(this.CMRandomNumber);
122         byte[] temp = new byte[(this.SCProtocolInitiator.getValueBytes()
123                                 .length +
124                                 this.CMRandomNumber.getValueLength())];
125         System.arraycopy(this.CMRandomNumber.getValueBytes(), 0, temp,
126                          0, this.CMRandomNumber.getValueLength());
127         System.arraycopy(this.SCProtocolInitiator.getValueBytes(), 0,
                temp,
128                          temp.length -
129                          this.SCProtocolInitiator.getValueBytes().length,
130                          this.SCProtocolInitiator.getValueBytes().length);
131         byte[] result = new byte[16];
132         this.myProtocolHelperObject.GenerateMac(temp, 0, temp.length,
133           result, 0, this.myProtocolHelperObject.myLongTermMacKey);
134         this.CMCookie.setTlvValues(result);
135         this.MessageHandler.addPTLV(this.CMCookie);
136       } catch (Exception cE) {
137         System.out.println(
138                           "Error ProtocolHandler.inMessageProcessing
                                Option = 1, : " + cE.getClass().getName());
139       }
140     } else if (Counter == 2) {
141       try {
142         this.EncryptedData.initialisationCTLV(this.EncryptedDataTag);
143         this.EncryptedData.addPTLV(this.CMIdentityTLV);
144         this.EncryptedData.addPTLV(this.SCIdentity);
145         this.EncryptedData.addPTLV(this.CMRandomNumber);
146         this.EncryptedData.addPTLV(this.SCRandomNumber);
147         this.myProtocolHelperObject.SignatureMethod
148           (this.EncryptedData.getValueBytes(), 0,
149           this.EncryptedData.getValueBytes().length,
150           this.SignedData.getBytesTlvRepresentation(), 6, null,
151           ProtocolHelperClass.SIGN_MODE_GENERATION);
152         this.EncryptedData.addPTLV(this.SignedData);
153         this.EncryptedData.addCTLV(this.CMSignatureCertificate);
154         this.myProtocolHelperObject.GenerateEncryption
155           (this.EncryptedData.getValueBytes(), 0,
```

```
156             this.EncryptedData.getValueBytes().length,
157             this.EncryptedData.getBytesTlvRepresentation(), 7,
158             this.mySessionEncryptionKey);
159           this.MACedData.initialisationPTLV(this.MACedDataTag, 16);
160           this.myProtocolHelperObject.GenerateMac
161             (this.EncryptedData.getValueBytes(), 0,
162             this.EncryptedData.getTagValueLength(),
163             this.MACedData.getBytesTlvRepresentation(), 6,
164             this.mySessionMacKey);
165           this.MessageHandler.initialisationCTLV(this.MessageHandlerTagTwo);
166           this.MessageHandler.addCTLV(EncryptedData);
167           this.MessageHandler.addPTLV(this.MACedData);
168           this.MessageHandler.addPTLV(this.CMCookie);
169         } catch (Exception cE) {
170           System.out.println(
171                               "Error ProtocolHandler.inMessageProcessing
                                      Option = 1, : " + cE.getClass().getName());
172         }
173       } else {
174         System.out.println(
175                             "Protocol Stoped : Illegal Message Value
                                  (ProtocolHanlder.inMessageProcessing()");
176       }
177       return this.MessageHandler.getBytesTlvRepresentation();
178     }
179     public boolean inMessageProcessing(byte[] inMessage, int Counter) {
180       try {
181         if (Counter == 1) {
182           this.SCProtocolInitiator.setBytesTlvRepresentation(inMessage, 0,
183             22);
184         } else
185         if (Counter == 2) {
186           this.MessageHandler.reset();
187           this.EncryptedData.reset();
188           this.MessageHandler.setBytesTlvRepresentation(inMessage, 0,
189             inMessage.length - 2);
190           this.childExtractionFromCTLV(this.MessageHandler);
191           GenerateKeys(this.SCDHStore.getValueBytes());
192           byte[] temp = new byte[16];
193           this.myProtocolHelperObject.GenerateMac
194             (this.EncryptedData.getValueBytes(), 0,
195             this.EncryptedData.getValueBytes().length, temp, 0,
196             this.mySessionMacKey);
197           if (Arrays.equals(this.MACedData.getValueBytes(), temp)){}
198            else {
199             System.out.println(
200                                 "Integrity Check Failure : ERROR at
                                      ProtocolHandler.inMessageProcessing \n");
201             System.exit(0);
202           }
203           this.myProtocolHelperObject.GenerateDecryption
```

```
204             (this.EncryptedData.getValueBytes(), 0,
205             this.EncryptedData.getValueBytes().length,
206             this.EncryptedData.getBytesTlvRepresentation(), 7,
207             this.mySessionEncryptionKey);
208           this.childExtractionFromCTLV(EncryptedData);
209           if (Arrays.equals(PlatformHashPreset,
210               this.PlatformHash.getValueBytes())){}
211            else {
212             System.out.println("Platform Digest Not Verified");
213           }
214           childExtractionFromCTLV(this.SCCertificate);
215           BigInteger SCpublicExponent = new BigInteger(byteToString
216             (this.PublicExponent.getValueBytes()), 16);
217           BigInteger SCpublicModulus = new BigInteger(byteToString
218             (this.PublicModulus.getValueBytes()), 16);
219           KeyFactory factory = KeyFactory.getInstance("RSA");
220           SCVerificationKey = (PublicKey)factory.generatePublic(new
221                                RSAPublicKeySpec(SCpublicModulus,
222                                SCpublicExponent));
223           temp = new byte[(this.PlatformHash.getTagLength() +
224                            this.SCIdentity.getTagLength() +
225                            this.SCRandomNumber.getTagLength() +
226                            this.CMRandomNumber.getTagLength())];
227           System.arraycopy(this.EncryptedData.getBytesTlvRepresentation(),
                7,
228                            temp, 0, temp.length);
229           if (this.myProtocolHelperObject.SignatureMethod(temp, 0,
230               temp.length, this.SignedData.getValueBytes(), 0,
231               SCVerificationKey,
                      ProtocolHelperClass.SIGN_MODE_VERIFICATION))
232               {}
233            else {
234             System.out.println(
235                                "Signature Verification Failed..... Check
                                     code");
236           }
237         }
238       } catch (Exception cE) {
239         System.out.println("Error in ProtocolHandler.inMessageProcessing : "
240                            + cE.getClass().getName());
241       }
242       return true;
243     }
244     public static String byteToString(byte[] inArray) {
245       byte[] HEX_CHAR_TABLE =  {
246         (byte)'0', (byte)'1', (byte)'2', (byte)'3', (byte)'4', (byte)'5',
247          (byte)'6', (byte)'7', (byte)'8', (byte)'9', (byte)'a', (byte)'b',
248         (byte)'c', (byte)'d', (byte)'e', (byte)'f'
249       };
250       byte[] hex = new byte[2 * inArray.length];
251       int index = 0;
```

```
252     for (byte b: inArray) {
253       int v = b & 0xFF;
254       hex[index++] = HEX_CHAR_TABLE[v >>> 4];
255       hex[index++] = HEX_CHAR_TABLE[v & 0xF];
256     }
257     try {
258       return new String(hex, "ASCII");
259     } catch (Exception cE) {
260       System.out.println("Exception in bytesToString : " +
            cE.getMessage())
261                           ;
262     }
263     return "Error";
264   }
265   void childExtractionFromCTLV(ConstructedTLV inCTLV) {
266     try {
267       int childs = inCTLV.getChildNumbers();
268       PrimitiveTLV pTemp = null;
269       ConstructedTLV cTemp = null;
270       while (childs > 0) {
271         switch (inCTLV.nextType()) {
272           case 1:
273             pTemp = (PrimitiveTLV)inCTLV.getNext();
274             if (Arrays.equals(pTemp.getTagName(),
275                       this.SCRandomNumber.getTagName())) {
276               this.SCRandomNumber = pTemp;
277             }else if (Arrays.equals(pTemp.getTagName(),
278                       this.MACedData.getTagName())) {
279               this.MACedData = pTemp;
280             }else if (Arrays.equals(pTemp.getTagName(),
281                       this.CMCookie.getTagName())) {
282               if (Arrays.equals(pTemp.getBytesTlvRepresentation(),
283                   this.CMCookie.getBytesTlvRepresentation())){}
284             }else if (Arrays.equals(pTemp.getTagName(),
285                       this.SCIdentity.getTagName())) {
286               this.SCIdentity = pTemp;
287             }else if (Arrays.equals(pTemp.getTagName(),
288                       this.SignedData.getTagName())) {
289               this.SignedData = pTemp;
290             }else if (Arrays.equals(pTemp.getTagName(),
291                       this.PublicExponent.getTagName())) {
292               this.PublicExponent = pTemp;
293             }else if (Arrays.equals(pTemp.getTagName(),
294                       this.PublicModulus.getTagName())) {
295               this.PublicModulus = pTemp;
296             }else if (Arrays.equals(pTemp.getTagName(),
297                       this.PlatformHash.getTagName())) {
298               this.PlatformHash = pTemp;
299             }else if (Arrays.equals(pTemp.getTagName(),
300                       this.UserIdentity.getTagName())) {
301               this.UserIdentity = pTemp;
```

```
302             }
303             break;
304           case 0:
305             cTemp = (ConstructedTLV)inCTLV.getNext();
306             if (Arrays.equals(cTemp.getTagName(),
307                 this.EncryptedData.getTagName())) {
308               this.EncryptedData = cTemp;
309             }else if (Arrays.equals(cTemp.getTagName(),
310                       SCUserCertificate.getTagName())) {
311               this.SCUserCertificate = cTemp;
312             }else if (Arrays.equals(cTemp.getTagName(),
313                       SCCertificate.getTagName())) {
314               this.SCCertificate = cTemp;
315             }
316             break;
317           default:
318             System.out.println("Error In Parsing Input Message");
319         }
320         childs--;
321       }
322     } catch (Exception e) {
323       System.out.println(
324                           "Error in ProtocolHanlder.ChildExtractionMethod
                                : " + e.getClass().getName());
325     }
326   }
327   void GenerateKeys(byte[] inbuff) {
328     byte[] DHSecretKey = null;
329     try {
330       DHSecretKey =
331         this.myProtocolHelperObject.GenerateDHSessionKeyMaterial(inbuff,
              0,
332         inbuff.length);
333     } catch (Exception cE) {
334       System.out.println(
335                           "Exception At ProtocolHelperClass.GenerateKeys :
                                " + cE.getClass().getName());
336     }
337     byte[] keyGenKey = new byte[16];
338     System.arraycopy(DHSecretKey, 0, keyGenKey, 0, keyGenKey.length);
339     byte[] macInputValue = new byte[64];
340     System.arraycopy(this.CMRandomNumber.getValueBytes(), 0,
          macInputValue,
341                      0, 16);
342     System.arraycopy(this.SCRandomNumber.getValueBytes(), 0,
          macInputValue,
343                      16, 16);
344     System.arraycopy(DHSecretKey, 16, macInputValue, 32, 16);
345     for (int i = 48; i < 64; i++) {
346       macInputValue[i] = (byte)0x02;
347     }
```

```
348     try {
349       this.myProtocolHelperObject.GenerateMac(macInputValue, 0,
350         macInputValue.length, this.mySessionEncryptionKey, 0, keyGenKey);
351     } catch (Exception cE) {
352       System.out.println("Exception at ProtocolHandler.GenerateKeys : " +
353                          cE.getClass().getName());
354     }
355     for (int i = 48; i < 64; i++) {
356       macInputValue[i] = (byte)0x03;
357     }
358     try {
359       this.myProtocolHelperObject.GenerateMac(macInputValue, 0,
360         macInputValue.length, this.mySessionMacKey, 0, keyGenKey);
361     } catch (Exception cE) {
362       System.out.println("Exception at ProtocolHandler.GenerateKeys : " +
363                          cE.getClass().getName());
364     }
365   }
366 }
```

C.4 Secure and Trusted Channel Protocol — Service Provider

The Java Card implementation of the STCP_{SP} discussed in section 6.3 is listed in subsequent sections.

C.4.1 Smart Card Implementation

Following is the implementation of the smart card protocol handler that supports the STCP_{SP}.

```
1  package protocolSTCPSP;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.JCSystem;
8  import javacard.framework.Util;
9  import javacard.security.AESKey;
10 import javacard.security.Key;
11 import javacard.security.KeyBuilder;
12 import javacard.security.KeyPair;
13 import javacard.security.MessageDigest;
14 import javacard.security.RSAPrivateKey;
15 import javacard.security.RSAPublicKey;
16 import javacard.security.RandomData;
17 import javacard.security.Signature;
18 import javacardx.apdu.ExtendedLength;
19 import javacardx.crypto.Cipher;
20 public class ProtocolHandler extends Applet implements ExtendedLength
```

```
21  {
22  private byte[] SPDHChanllengerArray;
23  private byte[] SPRandomNumberArray;
24  private byte[] SPCookieArray;
25  private byte[] SCSPDHGeneratedValue;
26  private byte[] SCRandomNumberArray;
27  private byte[] SCUserCertificate;
28  private byte[] SCCertificate;
29  private byte[] SPDHChallengeTag = {
30    (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x01};
31  private byte[] MessageHandlerTagOne = {
32    (byte)0x1F, (byte)0xC0, (byte)0xAA, (byte)0xAA, (byte)0x00, (byte)
33     0x00, (byte)0x00};
34  private byte[] MessageHandlerTagTwo = {
35    (byte)0x1F, (byte)0xC0, (byte)0xBB, (byte)0xBB, (byte)0x00, (byte)
36     0x00, (byte)0x00};
37  private byte[] SPIdentity = null;
38  private byte[] SPRandomNumberTag = {
39    (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x01};
40  private byte[] SPCookieTag = {
41    (byte)0x1F, (byte)0x5F, (byte)0x5B, (byte)0x01};
42  private byte[] EncryptedDataTag = {
43    (byte)0x1F, (byte)0xC0, (byte)0xFE, (byte)0x01};
44  private byte[] SignedDataTag = {
45    (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x02};
46  private byte[] MACedDataTag = {
47    (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x01};
48  private byte[] PlatformHash = {
49    (byte)0x1F, (byte)0x5F, (byte)0x5E, (byte)0xAF};
50  private byte[] SCIdentityTag = {
51    (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x02, (byte)0x00, (byte)
52     0x12, (byte)0x7A, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6,
53     (byte)0xC1, (byte)0x22, (byte)0x07, (byte)0xC9, (byte)0xF9,
54     (byte)0x8D, (byte)0x11, (byte)0xED, (byte)0x34, (byte)0xDB,
55     (byte)0xF6, (byte)0x0B, (byte)0x2C};
56  private byte[] UserIdentity = {
57    (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x03, (byte)0x00, (byte)
58     0x14, (byte)0x7A, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6,
59     (byte)0xC1, (byte)0x22, (byte)0x07, (byte)0xC9, (byte)0xC9,
60     (byte)0x8D, (byte)0xD1, (byte)0xED, (byte)0xFC, (byte)0xDB,
61     (byte)0xF6, (byte)0x0B, (byte)0x2C, (byte)0x0B, (byte)0x2C};
62  private byte[] ExponentTag = {
63    (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x01};
64  private byte[] ModulusTag = {
65    (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x02};
66  private byte[] SCDHChalleneTag = {
67    (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x02};
68  private byte[] SCRandomNumberTag = {
69    (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x02};
70  private byte[] SPCertificateTag = {
71    (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x01};
```

```
72  private byte[] SCCertificateTag =  {
73    (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x02};
74  private byte[] SCUserCertificateTag =  {
75    (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x03};
76  short PTLVDataOffset = (short)6;
77  short CTLVDataOffset = (short)7;
78  short TLVLengthOffset = (short)4;
79  short copyPointer = (short)0;
80  byte[] SCDHData;
81  final static byte CLA = (byte)0xB0;
82  final static byte StartProtocol = (byte)0x40;
83  final static byte InitiationProtocol = (byte)0xff;
84  final static short SW_CLASSNOTSUPPORTED = 0x6320;
85  final static short SW_ERROR_INS = 0x6300;
86  RandomData randomDataGen;
87  Cipher pkCipher;
88  short messageNumber = 0;
89  byte[] receivingBuffer = null;
90  short bytesLeft = 0;
91  short readCount = 0;
92  short rCount = 0;
93  short signlength = 0;
94  private RSAPublicKey dhKey = (RSAPublicKey)KeyBuilder.buildKey
95                               (KeyBuilder.TYPE_RSA_PUBLIC,
96                               KeyBuilder.LENGTH_RSA_2048, false);
97  private byte[] randomExponent;
98  final static byte GEN_KEYCONTRIBUTION = 0x01;
99  final static byte GEN_DHKEY = 0x02;
100 AESKey phCipherKey;
101 Cipher syCipher;
102 byte[] InitialisationVector =  {
103   (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89, (byte)
104    0x99, (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1,
105    (byte)0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52};
106 AESKey phMacGeneratorKey;
107 Signature phMacGenerator;
108 Signature phSign;
109 KeyPair phSCKeyPair;
110 KeyPair phUserKeyPair;
111 RSAPublicKey SPVerificationKey = null;
112 private ProtocolHandler() {
113   phMacGeneratorKey = (AESKey)KeyBuilder.buildKey
114                       (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
115                       KeyBuilder.LENGTH_AES_128, false);
116   phMacGenerator = Signature.getInstance
117     (Signature.ALG_AES_MAC_128_NOPAD, false);
118   phSign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false)
119                                 ;
120   phSCKeyPair = new KeyPair(KeyPair.ALG_RSA,
121                             KeyBuilder.LENGTH_RSA_512);
122   phUserKeyPair = new KeyPair(KeyPair.ALG_RSA,
```

```
123                                     KeyBuilder.LENGTH_RSA_512);
124     phCipherKey = (AESKey)KeyBuilder.buildKey
125                       (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
126                       KeyBuilder.LENGTH_AES_128, false);
127     syCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
128                                     false);
129     randomDataGen = RandomData.getInstance
130       (RandomData.ALG_SECURE_RANDOM);
131     pkCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
132     dhInitialisation();
133     phSCKeyPair.genKeyPair();
134     phUserKeyPair.genKeyPair();
135   }
136   public static void install(byte bArray[], short bOffset, byte
137                                 bLength)throws ISOException {
138     new ProtocolHandler().register();
139   }
140   public void initialiseProtocol() {
141     short initialPointer = 0;
142     SCDHData = JCSystem.makeTransientByteArray((short)((short)
143       this.ClassDH.dhModulus.length + PTLVDataOffset),
144       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
145     Util.arrayCopyNonAtomic(this.SCDHChalleneTag, (short)
146                               initialPointer, this.SCDHData, (short)0,
147                               (short)this.SCDHChalleneTag.length);
148     this.shortToBytes(SCDHData, (short)4, (short)((short)
149                         SCDHData.length - (short)PTLVDataOffset));
150     this.dhKeyConGen(this.SCDHData, this.PTLVDataOffset,
151                         ProtocolHandler.GEN_KEYCONTRIBUTION);
152     SPDHChanllengerArray = JCSystem.makeTransientByteArray((short)(
153       (short)this.ClassDH.dhModulus.length + this.PTLVDataOffset),
154       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
155     SPRandomNumberArray = JCSystem.makeTransientByteArray((short)22,
156       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
157     SPCookieArray = JCSystem.makeTransientByteArray((short)22,
158       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
159     SCRandomNumberArray = JCSystem.makeTransientByteArray((short)22,
160       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
161     Util.arrayCopyNonAtomic(this.SCRandomNumberTag, (short)
162                               initialPointer, this.SCRandomNumberArray,
163                               (short)initialPointer, (short)
164                               this.SCRandomNumberTag.length);
165     this.shortToBytes(this.SCRandomNumberArray, (short)4, (short)(
166                         (short)this.SCRandomNumberArray.length - (short)
167                         PTLVDataOffset));
168     try {
169       this.SCUserCertificate = JCSystem.makeTransientByteArray((short)
170         86, JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
171       initialPointer = Util.arrayCopyNonAtomic
172                               (this.SCUserCertificateTag, (short)0,
173                                   this.SCUserCertificate,
```

```
173                                 (short)0,
                                        (short)this.SCUserCertificateTag.length);
174         this.shortToBytes(this.SCUserCertificate, (short)4, (short)
175                             (this.SCUserCertificate.length - (short)7));
176         initialPointer = Util.arrayCopyNonAtomic(this.ExponentTag,
177           (short)0, this.SCUserCertificate, (short)(initialPointer +
178           (short)3), (short)this.ExponentTag.length);
179         RSAPublicKey myPublic = (RSAPublicKey)
180                                     this.phUserKeyPair.getPublic();
181         short kLen = myPublic.getExponent(this.SCUserCertificate,
182           (short)(initialPointer + (short)2));
183         this.shortToBytes(this.SCUserCertificate, initialPointer, kLen);
184         initialPointer += (short)(kLen + (short)2);
185         this.SCUserCertificate[6]++;
186         initialPointer = Util.arrayCopyNonAtomic(this.ModulusTag,
187           (short)0, this.SCUserCertificate, (short)(initialPointer),
188           (short)this.ModulusTag.length);
189         kLen = myPublic.getModulus(this.SCUserCertificate, (short)
190                                       (initialPointer + (short)2));
191         this.shortToBytes(this.SCUserCertificate, initialPointer, kLen);
192         this.SCUserCertificate[6]++;
193         this.SPIdentity = JCSystem.makeTransientByteArray((short)24,
194           JCSystem.MEMORY_TYPE_TRANSIENT_RESET);
195         SPVerificationKey = (RSAPublicKey)KeyBuilder.buildKey
196                               (KeyBuilder.TYPE_RSA_PUBLIC,
197                               KeyBuilder.LENGTH_RSA_512, false);
198       } catch (Exception cE) {
199         ISOException.throwIt((short)0xCCCC);
200       }
201       try {
202         this.SCCertificate = JCSystem.makeTransientByteArray((short)86,
203           JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
204         initialPointer = Util.arrayCopyNonAtomic(this.SCCertificateTag,
205                                     (short)0, this.SCCertificate, (short)0,
                                            (short)
206                                     this.SCCertificateTag.length);
207         this.shortToBytes(this.SCCertificate, (short)4, (short)
208                             (this.SCCertificate.length - (short)7));
209         initialPointer = Util.arrayCopyNonAtomic(this.ExponentTag,
210                                     (short)0, this.SCCertificate,
                                            (short)(initialPointer + (short)
211                                     3), (short)this.ExponentTag.length);
212         RSAPublicKey myPublic = (RSAPublicKey)
213                                     this.phSCKeyPair.getPublic();
214         short kLen = myPublic.getExponent(this.SCCertificate, (short)
215           (initialPointer + (short)2));
216         this.shortToBytes(this.SCCertificate, initialPointer, kLen);
217         initialPointer += (short)(kLen + (short)2);
218         this.SCCertificate[6]++;
219         initialPointer = Util.arrayCopyNonAtomic(this.ModulusTag,
```

```
220                                   (short)0, this.SCCertificate,
                                          (short)(initialPointer), (short)
221                                   this.ModulusTag.length);
222       kLen = myPublic.getModulus(this.SCCertificate, (short)
223                                     (initialPointer + (short)2));
224       this.shortToBytes(this.SCCertificate, initialPointer, kLen);
225       this.SCCertificate[6]++;
226     } catch (Exception cE) {
227       ISOException.throwIt((short)0x6666);
228     }
229   }
230   public void process(APDU apdu)throws ISOException {
231     byte[] apduBuffer = apdu.getBuffer();
232     if (selectingApplet()) {
233       return ;
234     }
235     if (apduBuffer[ISO7816.OFFSET_CLA] != CLA) {
236       ISOException.throwIt(SW_CLASSNOTSUPPORTED);
237     }
238     if (apduBuffer[ISO7816.OFFSET_INS] == InitiationProtocol) {
239       this.initialiseProtocol();
240       return ;
241     }
242     receivingBuffer = null;
243     bytesLeft = 0;
244     bytesLeft = apdu.getIncomingLength();
245     receivingBuffer = JCSystem.makeTransientByteArray(bytesLeft,
246       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
247     readCount = (short)((short)apdu.setIncomingAndReceive());
248     rCount = 0;
249     if (bytesLeft > 0) {
250       rCount = Util.arrayCopyNonAtomic(apduBuffer,
251         ISO7816.OFFSET_EXT_CDATA, receivingBuffer, rCount, readCount);
252       bytesLeft -= readCount;
253     }
254     while (bytesLeft > 0) {
255       try {
256         readCount = apdu.receiveBytes((short)0);
257         rCount = Util.arrayCopyNonAtomic(apduBuffer, (short)0,
258           receivingBuffer, rCount, readCount);
259         bytesLeft -= readCount;
260       } catch (Exception aE) {
261         ISOException.throwIt((short)0x7AAA);
262       }
263     }
264     try {
265       parseMessage(receivingBuffer);
266     } catch (Exception cE) {
267       ISOException.throwIt((short)0xA112);
268     }
269     if (this.receivingBuffer[3] == this.MessageHandlerTagOne[3]) {
```

```
270       receivingBuffer = JCSystem.makeTransientByteArray((short)568,
271         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
272       generateResponse((short)1);
273     } else if (this.receivingBuffer[3] ==
274                 this.MessageHandlerTagTwo[3]) {
275       processSecondMsg(receivingBuffer);
276       receivingBuffer = JCSystem.makeTransientByteArray((short)568,
277         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
278       generateResponse((short)2);
279     } else {
280       ISOException.throwIt(ProtocolHandler.SW_ERROR_INS);
281     }
282     JCSystem.requestObjectDeletion();
283     apdu.setOutgoing();
284     apdu.setOutgoingLength((short)copyPointer);
285     apdu.sendBytesLong(receivingBuffer, (short)0, (short)copyPointer);
286     JCSystem.requestObjectDeletion();
287   }
288   private void generateResponse(short msgNumber) {
289     short childPointerMessage = 6;
290     short encryptionOffset = 0;
291     copyPointer = 0;
292     if (msgNumber == 1) {
293       randomDataGen.generateData(this.SCRandomNumberArray,
294                                  this.PTLVDataOffset, (short)16);
295       this.dhKeyConGen(this.SPDHChanllengerArray, this.PTLVDataOffset,
296                        ProtocolHandler.GEN_DHKEY);
297       copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagOne,
298         (short)0, this.receivingBuffer, copyPointer, (short)
299         this.MessageHandlerTagOne.length);
300       copyPointer = Util.arrayCopyNonAtomic(this.SCDHData, (short)0,
301         this.receivingBuffer, copyPointer, (short)
302         this.SCDHData.length);
303       this.receivingBuffer[childPointerMessage]++;
304       copyPointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
305         (short)0, this.receivingBuffer, copyPointer, (short)
306         this.SCRandomNumberArray.length);
307       this.receivingBuffer[childPointerMessage]++;
308       keygenerator();
309       copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag,
310         (short)0, this.receivingBuffer, copyPointer, (short)
311         this.EncryptedDataTag.length);
312       this.receivingBuffer[childPointerMessage]++;
313       short childEnMessage = (short)(copyPointer + (short)2);
314       copyPointer += (short)3;
315       encryptionOffset = copyPointer;
316       copyPointer = Util.arrayCopyNonAtomic(this.SCIdentityTag,
317         (short)0, this.receivingBuffer, copyPointer, (short)
318         this.SCIdentityTag.length);
319       this.receivingBuffer[childEnMessage]++;
320       copyPointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
```

```
321         (short)0, this.receivingBuffer, copyPointer, (short)
322         this.SCRandomNumberArray.length);
323       this.receivingBuffer[childEnMessage]++;
324       copyPointer = Util.arrayCopyNonAtomic(this.SPRandomNumberArray,
325         (short)0, this.receivingBuffer, copyPointer, (short)
326         this.SPRandomNumberArray.length);
327       this.receivingBuffer[childEnMessage]++;
328       this.signGenerate(this.receivingBuffer, encryptionOffset,
329                         (short)(copyPointer - encryptionOffset),
330                         phUserKeyPair.getPrivate(),
331                         Signature.MODE_SIGN);
332       this.receivingBuffer[childEnMessage]++;
333       copyPointer = Util.arrayCopyNonAtomic(this.SCUserCertificate,
334         (short)0, this.receivingBuffer, copyPointer, (short)
335         this.SCUserCertificate.length);
336       this.receivingBuffer[childEnMessage]++;
337       messageEncryption(this.receivingBuffer, encryptionOffset,
338                         (short)(copyPointer - encryptionOffset));
339       this.shortToBytes(receivingBuffer, (short)(encryptionOffset -
340                         (short)3), (short)(copyPointer -
341                         encryptionOffset));
342       macGenerate(this.receivingBuffer, encryptionOffset, (short)
343                   (copyPointer - encryptionOffset),
344                   Signature.MODE_SIGN);
345       this.receivingBuffer[childPointerMessage]++;
346       copyPointer = Util.arrayCopyNonAtomic(this.SPCookieArray,
347         (short)0, this.receivingBuffer, copyPointer, (short)
348         this.SPCookieArray.length);
349       this.receivingBuffer[childPointerMessage]++;
350       this.shortToBytes(this.receivingBuffer, (short)4, copyPointer);
351     } else if (msgNumber == 2) {
352       copyPointer = (short)0;
353       short tempLength = (short)0;
354       short mainChildPointer = (short)6;
355       short mainLengthPointer = (short)4;
356       short encryptedChildPointer = (short)13;
357       short generalLengthPointer = (short)0;
358       this.receivingBuffer[mainChildPointer] = (short)0;
359       this.receivingBuffer[encryptedChildPointer] = (short)0;
360       copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagTwo,
361         (short)0, this.receivingBuffer, copyPointer, (short)7);
362       this.receivingBuffer[mainChildPointer]++;
363       copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag,
364         (short)0, this.receivingBuffer, copyPointer, (short)4);
365       copyPointer += (short)3;
366       encryptionOffset = copyPointer;
367       copyPointer = Util.arrayCopyNonAtomic(this.PlatformHash, (short)
368         0, receivingBuffer, copyPointer, (short)4);
369       generalLengthPointer = copyPointer;
370       copyPointer += (short)2;
371       MessageDigest myHashGen = MessageDigest.getInstance
```

```
372           (MessageDigest.ALG_SHA_256, false);
373         tempLength = (short)myHashGen.doFinal(this.ClassDH.dhModulus,
              (short)0,
374                       (short)this.ClassDH.dhModulus.length, receivingBuffer,
375                       copyPointer);
376         this.receivingBuffer[encryptedChildPointer]++;
377         this.shortToBytes(this.receivingBuffer, generalLengthPointer,
378                           (short)(tempLength));
379         copyPointer += tempLength;
380         copyPointer = Util.arrayCopyNonAtomic(this.UserIdentity, (short)
381           0, this.receivingBuffer, copyPointer, (short)
382           this.UserIdentity.length);
383         this.receivingBuffer[encryptedChildPointer]++;
384         copyPointer = Util.arrayCopyNonAtomic(this.SPIdentity, (short)0,
385           this.receivingBuffer, copyPointer, (short)
386           this.SPIdentity.length);
387         this.receivingBuffer[encryptedChildPointer]++;
388         copyPointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
389           (short)0, this.receivingBuffer, copyPointer, (short)
390           this.SCRandomNumberArray.length);
391         this.receivingBuffer[encryptedChildPointer]++;
392         copyPointer = Util.arrayCopyNonAtomic(this.SPRandomNumberArray,
393           (short)0, this.receivingBuffer, (short)copyPointer, (short)
394           this.SPRandomNumberArray.length);
395         this.receivingBuffer[encryptedChildPointer]++;
396         try {
397           this.signGenerate(receivingBuffer, (short)(encryptionOffset),
398                             (short)(copyPointer - encryptionOffset),
399                             phSCKeyPair.getPrivate(),
400                             Signature.MODE_SIGN);
401           this.receivingBuffer[encryptedChildPointer]++;
402         } catch (Exception cE) {
403           ISOException.throwIt((short)0xFA17);
404         }
405         copyPointer = Util.arrayCopyNonAtomic(this.SCCertificate,
406           (short)0, this.receivingBuffer, copyPointer, (short)
407           this.SCCertificate.length);
408         this.receivingBuffer[encryptedChildPointer]++;
409         try {
410           this.messageEncryption(receivingBuffer, (short)
411                                  (encryptedChildPointer + (short)1),
412                                  (short)(copyPointer -
413                                  (encryptedChildPointer + (short)1)));
414         } catch (Exception cE) {
415           ISOException.throwIt((short)(copyPointer -
416                                encryptedChildPointer + (short)1));
417         }
418         this.shortToBytes(this.receivingBuffer, (short)
419                           (encryptedChildPointer - (short)2), (short)
420                           (copyPointer - (short)(encryptedChildPointer
421                           + (short)1)));
```

```
422       this.macGenerate(receivingBuffer, (short)(encryptedChildPointer
423                           + (short)1), (short)(copyPointer -
424                           (encryptedChildPointer + (short)1)),
425                           Signature.MODE_SIGN);
426       this.receivingBuffer[mainChildPointer]++;
427       copyPointer = Util.arrayCopyNonAtomic(this.SPCookieArray,
428         (short)0, this.receivingBuffer, copyPointer, (short)
429         this.SPCookieArray.length);
430       this.receivingBuffer[mainChildPointer]++;
431       this.shortToBytes(this.receivingBuffer, mainLengthPointer,
432                           (short)(copyPointer - (short)7));
433     }
434   }
435   void platformHashGeneration(byte[] inArray, short inOffset){}
436   void processSecondMsg(byte[] inArray) {
437     short inOffset = (short)(this.CTLVDataOffset +
438                       this.CTLVDataOffset);
439     short inLength = (short)(ProtocolHandler.bytesToShort(inArray,
440                       (short)(inOffset - (short)3)));
441     if (this.macGenerate(inArray, inOffset, inLength,
442         Signature.MODE_VERIFY)) {
443       this.phDecryption(inArray, inOffset, inLength);
444       Util.arrayCopyNonAtomic(inArray, inOffset, this.SPIdentity,
445                               (short)0, (short)this.SPIdentity.length)
446                               ;
447       inOffset += (short)151;
448       inLength = (short)3;
449       SPVerificationKey.setExponent(inArray, inOffset, inLength);
450       inOffset += (short)(inLength + this.PTLVDataOffset);
451       inLength = (short)64;
452       SPVerificationKey.setModulus(inArray, inOffset, inLength);
453       inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
454       inLength = (short)68;
455       if (this.signGenerate(inArray, inOffset, inLength,
456           SPVerificationKey, Signature.MODE_VERIFY)) {
457         return ;
458       } else {
459         ISOException.throwIt((short)0x6666);
460       }
461     } else {
462       ISOException.throwIt((short)0xFA18);
463     }
464   }
465   void parseMessage(byte[] inBuffer) {
466     byte childLeft = inBuffer[(short)(this.CTLVDataOffset - (short)1)
467                               ];
468     short pointer = (short)this.CTLVDataOffset;
469     try {
470       while (childLeft > 0) {
471         if (Util.arrayCompare(SPDHChallengeTag, (short)0, inBuffer,
472             pointer, (short)4) == 0) {
```

```
473           Util.arrayCopy(inBuffer, pointer, this.SPDHChanllengerArray,
474                          (short)0, (short)
475                          this.SPDHChanllengerArray.length);
476           pointer += (short)this.SPDHChanllengerArray.length;
477         } else if (Util.arrayCompare(this.SPRandomNumberTag, (short)0,
478                    inBuffer, pointer, (short)4) == 0) {
479           Util.arrayCopyNonAtomic(inBuffer, pointer,
480                                   this.SPRandomNumberArray, (short)0,
481                                   (short)
482                                   (this.SPRandomNumberArray.length));
483           pointer += (short)(this.SPRandomNumberArray.length);
484         } else if (Util.arrayCompare(this.SPCookieTag, (short)0,
485                    inBuffer, pointer, (short)4) == 0) {
486           Util.arrayCopyNonAtomic(inBuffer, pointer,
487                                   this.SPCookieArray, (short)0,
488                                   (short)(this.SPCookieArray.length));
489           pointer += (short)(this.SPCookieArray.length);
490         }
491         childLeft -= (short)1;
492       }
493     } catch (Exception cE) {
494       ISOException.throwIt((short)childLeft);
495     }
496   }
497   void protocolImplementation(){}
498   void dhInitialisation() {
499     dhKey.setModulus(ClassDH.dhModulus, (short)0,
          (short)ClassDH.dhModulus.length);
500   }
501   void dhKeyConGen(byte[] inbuff, short inbuffOffset, byte Oper_Mode)
502                    {
503     switch (Oper_Mode) {
504       case GEN_KEYCONTRIBUTION: randomExponent =
505         JCSystem.makeTransientByteArray((short)32,
506         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
507       randomDataGen.generateData(randomExponent, (short)0, (short)
508                                  randomExponent.length);
509       dhKey.setExponent(randomExponent, (short)0, (short)
510                         randomExponent.length);
511       pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
512       pkCipher.doFinal(ClassDH.dhBase, (short)0,
            (short)ClassDH.dhBase.length, inbuff,
513                        inbuffOffset);
514       break;
515       case GEN_DHKEY:
516         try {
517           dhKey.setExponent(randomExponent, (short)0, (short)
518                             randomExponent.length);
519           pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
520           SCSPDHGeneratedValue = JCSystem.makeTransientByteArray(
521             (short)ClassDH.dhModulus.length,
```

```
522             JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
523           pkCipher.doFinal(inbuff, inbuffOffset, (short)((short)
524                           inbuff.length - (short)this.PTLVDataOffset)
525                           , SCSPDHGeneratedValue, (short)0);
526         }
527          catch (Exception cE) {
528           ISOException.throwIt((short)0xD86E);
529         }
530         break;
531       default:
532         ISOException.throwIt((short)0x5FA1);
533     }
534   }
535   void keygenerator() {
536     AESKey sessionGenKey = (AESKey)KeyBuilder.buildKey
537                             (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
538                             KeyBuilder.LENGTH_AES_128, false);
539     sessionGenKey.setKey(SCSPDHGeneratedValue, (short)0);
540     phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
541                         InitialisationVector, (short)0, (short)
542                         InitialisationVector.length);
543     byte[] keyGenMacData = JCSystem.makeTransientByteArray((short)64,
544       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
545     short pointer = 0;
546     pointer = Util.arrayCopyNonAtomic(this.SPRandomNumberArray,
547       this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
548     pointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
549       this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
550     pointer = Util.arrayCopyNonAtomic(SCSPDHGeneratedValue, (short)16,
551       keyGenMacData, (short)pointer, (short)16);
552     for (short i = 48; i < 64; i++) {
553       keyGenMacData[i] = (byte)0x02;
554     }
555     phMacGenerator.sign(keyGenMacData, (short)0, (short)
556                         keyGenMacData.length, SCSPDHGeneratedValue,
557                         (short)0);
558     this.phCipherKey.setKey(SCSPDHGeneratedValue, (short)0);
559     for (short i = 48; i < 64; i++) {
560       keyGenMacData[i] = (byte)0x03;
561     }
562     phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
563                         InitialisationVector, (short)0, (short)
564                         InitialisationVector.length);
565     phMacGenerator.sign(keyGenMacData, (short)0, (short)
566                         keyGenMacData.length, SCSPDHGeneratedValue,
567                         (short)0);
568     this.phMacGeneratorKey.setKey(SCSPDHGeneratedValue, (short)0);
569     SCSPDHGeneratedValue = null;
570     JCSystem.requestObjectDeletion();
571   }
572   void messageEncryption(byte[] inbuff, short inbuffOffset, short
```

```
573                         inbuffLength) {
574     syCipher.init(phCipherKey, Cipher.MODE_ENCRYPT,
575                   InitialisationVector, (short)0, (short)
576                   InitialisationVector.length);
577     short temp;
578     this.shortToBytes(inbuff, (short)(inbuffOffset - 3), temp =
579                       (short)syCipher.doFinal(inbuff, inbuffOffset,
580                       inbuffLength, inbuff, inbuffOffset));
581   }
582   void phDecryption(byte[] inbuff, short inbuffOffset, short
583                     inbuffLength) {
584     syCipher.init(phCipherKey, Cipher.MODE_DECRYPT,
585                   InitialisationVector, (short)0, (short)
586                   InitialisationVector.length);
587     syCipher.doFinal(inbuff, inbuffOffset, inbuffLength, inbuff,
588                      inbuffOffset);
589   }
590   boolean macGenerate(byte[] inbuff, short inbuffOffset, short
591                       inbuffLength, short macMode) {
592     if (macMode == Signature.MODE_SIGN) {
593       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
594                           InitialisationVector, (short)0, (short)
595                           InitialisationVector.length);
596       try {
597         copyPointer = Util.arrayCopyNonAtomic(this.MACedDataTag,
598           (short)0, this.receivingBuffer, copyPointer, (short)
599           this.MACedDataTag.length);
600         copyPointer += 2;
601       } catch (Exception ce) {
602         ISOException.throwIt((short)0xFA17);
603       }
604       try {
605         short length = (short)phMacGenerator.sign
606                         (this.receivingBuffer, inbuffOffset,
607                         inbuffLength, inbuff, copyPointer);
608         this.shortToBytes(inbuff, (short)(copyPointer - (short)2),
609                           length);
610         copyPointer += length;
611       } catch (Exception ce) {
612         ISOException.throwIt((short)0x0987);
613       }
614       return true;
615     } else if (macMode == Signature.MODE_VERIFY) {
616       try {
617         phMacGenerator.init(phMacGeneratorKey, Signature.MODE_VERIFY,
618                             InitialisationVector, (short)0, (short)
619                             InitialisationVector.length);
620         return phMacGenerator.verify(this.receivingBuffer,
621           inbuffOffset, inbuffLength, inbuff, (short)(inbuffOffset +
622           inbuffLength + this.PTLVDataOffset), (short)16);
623       } catch (Exception cE) {
```

```
624           ISOException.throwIt((short)0xC1C2);
625         }
626       }
627       return false;
628     }
629     boolean signGenerate(byte[] inbuff, short inbuffOffset, short
630                          inbufflength, Key kpSign, short signMode) {
631       if (signMode == Signature.MODE_SIGN) {
632         copyPointer = Util.arrayCopyNonAtomic(this.SignedDataTag,
633           (short)0, this.receivingBuffer, copyPointer, (short)
634           this.SignedDataTag.length);
635         copyPointer += (short)2;
636         phSign.init((RSAPrivateKey)kpSign, Signature.MODE_SIGN);
637         signlength = phSign.sign(inbuff, (short)inbuffOffset,
638                                  inbufflength, inbuff, copyPointer);
639         this.shortToBytes(this.receivingBuffer, (short)(copyPointer -
640                           (short)2), signlength);
641         copyPointer += signlength;
642         return true;
643       } else if (signMode == Signature.MODE_VERIFY) {
644         phSign.init((RSAPublicKey)kpSign, Signature.MODE_VERIFY);
645         return phSign.verify(inbuff, inbuffOffset, inbufflength, inbuff,
646                              (short)(inbuffOffset + inbufflength +
647                              this.PTLVDataOffset), (short)64);
648       }
649       return false;
650     }
651     public static short bytesToShort(byte[] ArrayBytes) {
652       return (short)(((ArrayBytes[0] << 8)) | ((ArrayBytes[1] & 0xff)));
653     }
654     public static short bytesToShort(byte[] ArrayBytes, short
655                                      arrayOffset) {
656       return (short)(((ArrayBytes[arrayOffset] << 8)) | ((ArrayBytes[
657               (short)(arrayOffset + (short)1)] & 0xff)));
658     }
659     private void shortToBytes(byte[] Array, short inShort) {
660       Array[0] = (byte)((short)(inShort & (short)0xFF00) >> (short)
661                  0x0008);
662       Array[1] = (byte)(inShort & (short)0x00FF);
663     }
664     private void shortToBytes(byte[] Array, short arrayOffset, short
665                               inShort) {
666       Array[arrayOffset] = (byte)((short)(inShort & (short)0xFF00) >>
667                            (short)0x0008);
668       Array[(short)(arrayOffset + (short)1)] = (byte)(inShort & (short)
669              0x00FF);
670     }
671 }
```

C.4.2 Service Provider Implementation

In this section, we detail the SP's implementation of the STCP_{SP} and the helper functions utlised during the STCP_{SP} are discussed in appendices C.11.1 and C.11.2.

```
1  package javacardterminal;
2
3  import java.math.BigInteger;
4  import java.security.*;
5  import java.security.interfaces.RSAPublicKey;
6  import java.security.spec.RSAPublicKeySpec;
7  import java.util.Arrays;
8  import javax.crypto.*;
9  import javax.crypto.spec.SecretKeySpec;
10 public class ProtocolHandler {
11   private byte[] EncryptedDataTag =  {
12     (byte)0xFE, (byte)0x01};
13   private byte[] MACedDataTag =  {
14     (byte)0x5D, (byte)0x01};
15   private byte[] MessageHandlerTagOne =  {
16     (byte)0xAA, (byte)0xAA};
17   private byte[] MessageHandlerTagTwo =  {
18     (byte)0xBB, (byte)0xBB};
19   private byte[] PlatformHashPreset =  {
20     (byte)0xBF, (byte)0xE5, (byte)0x45, (byte)0x86, (byte)0x2C, (byte)
21       0xA1, (byte)0x02, (byte)0xAD, (byte)0x1E, (byte)0xED, (byte)0xDB,
22       (byte)0x5F, (byte)0xBF, (byte)0xA5, (byte)0xBF, (byte)0x85,
23       (byte)0x5A, (byte)0xC4, (byte)0x99, (byte)0x5C, (byte)0x56,
24       (byte)0xA8, (byte)0xB4, (byte)0x08, (byte)0xCE, (byte)0x3F,
25       (byte)0xE0, (byte)0x99, (byte)0xDC, (byte)0xE9, (byte)0x3A,
26       (byte)0x9D};
27   private byte[] PlatformHashTag =  {
28     (byte)0x5E, (byte)0xAF};
29   private byte[] PublicExponentTag =  {
30     (byte)0xEE, (byte)0x01};
31   private byte[] PublicModulusTag =  {
32     (byte)0xEE, (byte)0x02};
33   private byte[] SCCertificateTag =  {
34     (byte)0xF0, (byte)0x02};
35   private byte[] SCDHChallengeTag =  {
36     (byte)0x5C, (byte)0x02};
37   private byte[] SCIP =  {
38     (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)0x7C};
39   private byte[] SCIdentityTag =  {
40     (byte)0x5F, (byte)0x02};
41   private byte[] SCRandomNumberTag =  {
42     (byte)0x5A, (byte)0x02};
43   private byte[] SCUserCertificateTag =  {
44     (byte)0xF0, (byte)0x03
45   }
46   ;
```

```
47  private PublicKey SCUserVerificationKey = null;
48  private PublicKey SCVerificationKey = null;
49  private byte[] SPCookieTag = {
50    (byte)0x5B, (byte)0x01};
51  private byte[] SPDHChallengeTag = {
52    (byte)0x5C, (byte)0x01};
53  private byte[] SPIdentity = {
54    (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)0x7C, (byte)0x62, (byte)
55     0x0A, (byte)0x86, (byte)0x52, (byte)0xBE, (byte)0x5E, (byte)0x90,
56     (byte)0x01, (byte)0xA8, (byte)0xD6, (byte)0x6A, (byte)0xD7,
57     (byte)0xB1, (byte)0x7C};
58  private byte[] SPIdentityTag = {
59    (byte)0x5F, (byte)0x01};
60  private byte[] SPRandomNumberTag = {
61    (byte)0x5A, (byte)0x01};
62  private byte[] SPSigVerificationKeyTag = {
63    (byte)0x51, (byte)0x01};
64  private byte[] SPSignatureCertTag = {
65    (byte)0xF0, (byte)0xF01};
66  private byte[] SignedDataTag = {
67    (byte)0x5D, (byte)0x02};
68  private byte[] UserIdentityTag = {
69    (byte)0x5F, (byte)0x03};
70  private PrimitiveTLV UserIdentity = PrimitiveTLV.getPrimitiveTLV
71    (this.UserIdentityTag);
72  private PrimitiveTLV SignedData = PrimitiveTLV.getPrimitiveTLV
73    (this.SignedDataTag);
74  private ConstructedTLV SPSignatureCertificate =
75    ConstructedTLV.getConstructedTLV(SPSignatureCertTag);
76  private PrimitiveTLV SPSigVerificationKey =
77    PrimitiveTLV.getPrimitiveTLV(this.SPSigVerificationKeyTag);
78  private PrimitiveTLV SPRandomNumber = PrimitiveTLV.getPrimitiveTLV
79    (this.SPRandomNumberTag);
80  private PrimitiveTLV SPIdentityTLV = PrimitiveTLV.getPrimitiveTLV
81    (SPIdentityTag, SPIdentity);
82  private PrimitiveTLV SPDHChanllenger = PrimitiveTLV.getPrimitiveTLV
83    (this.SPDHChallengeTag);
84  private PrimitiveTLV SPCookie = PrimitiveTLV.getPrimitiveTLV
85    (this.SPCookieTag);
86  private ConstructedTLV SCUserCertificate =
87    ConstructedTLV.getConstructedTLV(this.SCUserCertificateTag);
88  private PrimitiveTLV SCRandomNumber = PrimitiveTLV.getPrimitiveTLV
89    (this.SCRandomNumberTag);
90  private PrimitiveTLV SCIdentity = PrimitiveTLV.getPrimitiveTLV
91    (SCIdentityTag);
92  private PrimitiveTLV SCDHChallenge = PrimitiveTLV.getPrimitiveTLV
93    (this.SCDHChallengeTag);
94  private ConstructedTLV SCCertificate =
95    ConstructedTLV.getConstructedTLV(this.SCCertificateTag);
96  private PrimitiveTLV PublicModulus = PrimitiveTLV.getPrimitiveTLV
97    (this.PublicModulusTag);
```

```
98  private PrimitiveTLV PublicExponent = PrimitiveTLV.getPrimitiveTLV
99    (this.PublicExponentTag);
100 private PrimitiveTLV PlatformHash = PrimitiveTLV.getPrimitiveTLV
101   (this.PlatformHashTag);
102 public ConstructedTLV MessageHandler =
103   ConstructedTLV.getConstructedTLV(MessageHandlerTagOne);
104 private PrimitiveTLV MACedData = PrimitiveTLV.getPrimitiveTLV
105   (this.MACedDataTag);
106 private ConstructedTLV EncryptedData =
107   ConstructedTLV.getConstructedTLV(this.EncryptedDataTag);
108 private ProtocolHelperClass myProtocolHelperObject = new
109   ProtocolHelperClass();
110 private byte[] mySessionEncryptionKey = new byte[16];
111 private byte[] mySessionMacKey = new byte[16];
112 public ProtocolHandler() {
113   myProtocolHelperObject.protocolInitialise();
114   RSAPublicKey tempKey = (RSAPublicKey)
115                          myProtocolHelperObject.getPublicKey();
116   byte[] tempExponent = tempKey.getPublicExponent().toByteArray();
117   this.PublicExponent.initialisationPTLV(this.PublicExponentTag,
118     tempExponent.length);
119   this.PublicExponent.setTlvValues(tempExponent);
120   byte[] tempModulus = tempKey.getModulus().toByteArray();
121   this.PublicModulus.initialisationPTLV(this.PublicModulusTag,
122     (tempModulus.length - 1));
123   this.PublicModulus.setTlvValues(tempModulus, 1,
124     (tempModulus.length - 1));
125   SPSignatureCertificate.addPTLV(this.PublicExponent);
126   SPSignatureCertificate.addPTLV(this.PublicModulus);
127 }
128 public void initialiseProtocol() {
129   try {
130     this.SPDHChanllenger.setTlvValues
131       (this.myProtocolHelperObject.GenerateDHPublicValue());
132     this.MessageHandler.addPTLV(this.SPDHChanllenger);
133   } catch (Exception cE) {
134     System.out.println(
135                       "Error ProtocolHandler.initialiseProtocol Option
                            = 1, : " + cE.getClass().getName());
136   }
137 }
138 public byte[] outMessageProcessing(int Counter) {
139   if (Counter == 1) {
140     try {
141       this.SPRandomNumber.setTlvValues
142         (this.myProtocolHelperObject.getRandomNumber());
143       this.MessageHandler.addPTLV(this.SPRandomNumber);
144       byte[] temp = new byte[(this.SCIP.length +
145                               this.SPDHChanllenger.getValueLength()
146                               + this.SPRandomNumber.getValueLength()
147                               )];
```

```
148           System.arraycopy(this.SPDHChanllenger.getValueBytes(), 0,
149                            temp, 0, this.SPDHChanllenger.getValueLength
150                            ());
151           System.arraycopy(this.SPRandomNumber.getValueBytes(), 0, temp,
152                            this.SPDHChanllenger.getValueLength(),
153                            this.SPRandomNumber.getValueLength());
154           System.arraycopy(this.SCIP, 0, temp, temp.length -
155                            this.SCIP.length, this.SCIP.length);
156           byte[] result = new byte[16];
157           this.myProtocolHelperObject.GenerateMac(temp, 0, temp.length,
158             result, 0, this.myProtocolHelperObject.myLongTermMacKey);
159           this.SPCookie.setTlvValues(result);
160           this.MessageHandler.addPTLV(this.SPCookie);
161         } catch (Exception cE) {
162           System.out.println(
163                            "Error ProtocolHandler.inMessageProcessing
                                 Option = 1, : " + cE.getClass().getName());
164         }
165       } else if (Counter == 2) {
166         try {
167           this.EncryptedData.initialisationCTLV(this.EncryptedDataTag);
168           this.EncryptedData.addPTLV(this.SPIdentityTLV);
169           this.EncryptedData.addPTLV(this.SPRandomNumber);
170           this.EncryptedData.addPTLV(this.SCRandomNumber);
171           this.myProtocolHelperObject.SignatureMethod
172             (this.EncryptedData.getValueBytes(), 0,
173             this.EncryptedData.getValueBytes().length,
174             this.SignedData.getBytesTlvRepresentation(), 6, null,
175             ProtocolHelperClass.SIGN_MODE_GENERATION);
176           this.EncryptedData.addPTLV(this.SignedData);
177           this.EncryptedData.addCTLV(this.SPSignatureCertificate);
178           this.myProtocolHelperObject.GenerateEncryption
179             (this.EncryptedData.getValueBytes(), 0,
180             this.EncryptedData.getValueBytes().length,
181             this.EncryptedData.getBytesTlvRepresentation(), 7,
182             this.mySessionEncryptionKey);
183           this.MACedData.initialisationPTLV(this.MACedDataTag, 16);
184           this.myProtocolHelperObject.GenerateMac
185             (this.EncryptedData.getValueBytes(), 0,
186             this.EncryptedData.getTagValueLength(),
187             this.MACedData.getBytesTlvRepresentation(), 6,
188             this.mySessionMacKey);
189           this.MessageHandler.initialisationCTLV
190             (this.MessageHandlerTagTwo);
191           this.MessageHandler.addCTLV(EncryptedData);
192           this.MessageHandler.addPTLV(this.MACedData);
193           this.MessageHandler.addPTLV(this.SPCookie);
194         } catch (Exception cE) {
195           System.out.println(
196                            "Error ProtocolHandler.inMessageProcessing
                                 Option = 1, : " + cE.getClass().getName());
```

```
197         }
198       } else {
199         System.out.println(
200                             "Protocol Stoped : Illegal Message Value
                                    (ProtocolHanlder.inMessageProcessing()");
201       }
202       return this.MessageHandler.getBytesTlvRepresentation();
203     }
204     public boolean inMessageProcessing(byte[] inMessage, int Counter) {
205       try {
206         if (Counter == 1) {
207           MessageHandler.setBytesTlvRepresentation(inMessage, 0,
208             (inMessage.length - 2));
209           childExtractionFromCTLV(MessageHandler);
210           GenerateKeys(this.SCDHChallenge.getValueBytes());
211           byte[] temp = new byte[16];
212           this.myProtocolHelperObject.GenerateMac
213             (this.EncryptedData.getValueBytes(), 0,
214             this.EncryptedData.getValueBytes().length, temp, 0,
215             this.mySessionMacKey);
216           if (Arrays.equals(this.MACedData.getValueBytes(), temp)){}
217            else {
218             System.out.println(
219                                 "Integrity Check Failure : ERROR at
                                        ProtocolHandler.inMessageProcessing \n");
220             System.exit(0);
221           }
222           this.myProtocolHelperObject.GenerateDecryption
223             (this.EncryptedData.getValueBytes(), 0,
224             this.EncryptedData.getValueBytes().length,
225             this.EncryptedData.getBytesTlvRepresentation(), 7,
226             this.mySessionEncryptionKey);
227           childExtractionFromCTLV(this.EncryptedData);
228           childExtractionFromCTLV(this.SCUserCertificate);
229           BigInteger publicExponent = new BigInteger(byteToString
230             (this.PublicExponent.getValueBytes()), 16);
231           BigInteger publicModulus = new BigInteger(byteToString
232             (this.PublicModulus.getValueBytes()), 16);
233           KeyFactory factory = KeyFactory.getInstance("RSA");
234           SCUserVerificationKey = (PublicKey)factory.generatePublic(new
235                                     RSAPublicKeySpec(publicModulus,
236                                     publicExponent));
237           temp = new byte[(this.SCIdentity.getTagLength() +
238                             this.SCRandomNumber.getTagLength() +
239                             this.SPRandomNumber.getTagLength())];
240           System.arraycopy(this.EncryptedData.getBytesTlvRepresentation
241                             (), 7, temp, 0, temp.length);
242           if (this.myProtocolHelperObject.SignatureMethod(temp, 0,
243               temp.length, this.SignedData.getValueBytes(), 0,
244               SCUserVerificationKey,
245               ProtocolHelperClass.SIGN_MODE_VERIFICATION)){}
```

```
246          else {
247           System.out.println(
248                                "Signature Verification Failed..... Check
                                     code");
249          }
250        } else if (Counter == 2) {
251          this.MessageHandler.reset();
252          this.EncryptedData.reset();
253          this.MessageHandler.setBytesTlvRepresentation(inMessage, 0,
254            inMessage.length - 2);
255          this.childExtractionFromCTLV(this.MessageHandler);
256          byte[] temp = new byte[16];
257          this.myProtocolHelperObject.GenerateMac
258            (this.EncryptedData.getValueBytes(), 0,
259            this.EncryptedData.getValueBytes().length, temp, 0,
260            this.mySessionMacKey);
261          if (Arrays.equals(this.MACedData.getValueBytes(), temp)){}
262           else {
263            System.out.println(
264                                "Integrity Check Failure : ERROR at
                                     ProtocolHandler.inMessageProcessing \n");
265            System.exit(0);
266          }
267          this.myProtocolHelperObject.GenerateDecryption
268            (this.EncryptedData.getValueBytes(), 0,
269            this.EncryptedData.getValueBytes().length,
270            this.EncryptedData.getBytesTlvRepresentation(), 7,
271            this.mySessionEncryptionKey);
272          this.childExtractionFromCTLV(EncryptedData);
273          if (Arrays.equals(PlatformHashPreset,
274              this.PlatformHash.getValueBytes())){}
275           else {
276            System.out.println("Platform Digest Not Verified");
277          }
278          childExtractionFromCTLV(this.SCCertificate);
279          BigInteger SCpublicExponent = new BigInteger(byteToString
280            (this.PublicExponent.getValueBytes()), 16);
281          BigInteger SCpublicModulus = new BigInteger(byteToString
282            (this.PublicModulus.getValueBytes()), 16);
283          KeyFactory factory = KeyFactory.getInstance("RSA");
284          SCVerificationKey = (PublicKey)factory.generatePublic(new
285                               RSAPublicKeySpec(SCpublicModulus,
286                               SCpublicExponent));
287          temp = new byte[(this.PlatformHash.getTagLength() +
288                           this.UserIdentity.getTagLength() +
289                           this.SCIdentity.getTagLength() +
290                           this.SCRandomNumber.getTagLength() +
291                           this.SPRandomNumber.getTagLength())];
292          System.arraycopy(this.EncryptedData.getBytesTlvRepresentation
293                           (), 7, temp, 0, temp.length);
294          if (this.myProtocolHelperObject.SignatureMethod(temp, 0,
```

```
295             temp.length, this.SignedData.getValueBytes(), 0,
296             SCVerificationKey,
297             ProtocolHelperClass.SIGN_MODE_VERIFICATION)){}
298           else {
299            System.out.println(
300                               "Signature Verification Failed..... Check
                                       code");
301         }
302       }
303     } catch (Exception cE) {
304       System.out.println(
305                           "Error in ProtocolHandler.inMessageProcessing :
                                   " + cE.getClass().getName());
306     }
307     return true;
308   }
309   public static String byteToString(byte[] inArray) {
310     byte[] HEX_CHAR_TABLE =  {
311       (byte)'0', (byte)'1', (byte)'2', (byte)'3', (byte)'4', (byte)
312        '5', (byte)'6', (byte)'7', (byte)'8', (byte)'9', (byte)'a',
313        (byte)'b', (byte)'c', (byte)'d', (byte)'e', (byte)'f'
314     };
315     byte[] hex = new byte[2 * inArray.length];
316     int index = 0;
317     for (byte b: inArray) {
318       int v = b & 0xFF;
319       hex[index++] = HEX_CHAR_TABLE[v >>> 4];
320       hex[index++] = HEX_CHAR_TABLE[v & 0xF];
321     }
322     try {
323       return new String(hex, "ASCII");
324     } catch (Exception cE) {
325       System.out.println("Exception in bytesToString : " +
326                           cE.getMessage());
327     }
328     return "Error";
329   }
330   void childExtractionFromCTLV(ConstructedTLV inCTLV) {
331     try {
332       int childs = inCTLV.getChildNumbers();
333       PrimitiveTLV pTemp = null;
334       ConstructedTLV cTemp = null;
335       while (childs > 0) {
336         switch (inCTLV.nextType()) {
337           case 1:
338             pTemp = (PrimitiveTLV)inCTLV.getNext();
339             if (Arrays.equals(pTemp.getTagName(),
340                 this.SCDHChallenge.getTagName())) {
341               this.SCDHChallenge = pTemp;
342             }else if (Arrays.equals(pTemp.getTagName(),
343                       this.SCRandomNumber.getTagName())) {
```

```
344                 this.SCRandomNumber = pTemp;
345               }else if (Arrays.equals(pTemp.getTagName(),
346                         this.MACedData.getTagName())) {
347                 this.MACedData = pTemp;
348               }else if (Arrays.equals(pTemp.getTagName(),
349                         this.SPCookie.getTagName())) {
350                 if (Arrays.equals(pTemp.getBytesTlvRepresentation(),
351                     this.SPCookie.getBytesTlvRepresentation())){}
352               }else if (Arrays.equals(pTemp.getTagName(),
353                         this.SCIdentity.getTagName())) {
354                 this.SCIdentity = pTemp;
355               }else if (Arrays.equals(pTemp.getTagName(),
356                         this.SignedData.getTagName())) {
357                 this.SignedData = pTemp;
358               }else if (Arrays.equals(pTemp.getTagName(),
359                         this.PublicExponent.getTagName())) {
360                 this.PublicExponent = pTemp;
361               }else if (Arrays.equals(pTemp.getTagName(),
362                         this.PublicModulus.getTagName())) {
363                 this.PublicModulus = pTemp;
364               }else if (Arrays.equals(pTemp.getTagName(),
365                         this.PlatformHash.getTagName())) {
366                 this.PlatformHash = pTemp;
367               }else if (Arrays.equals(pTemp.getTagName(),
368                         this.UserIdentity.getTagName())) {
369                 this.UserIdentity = pTemp;
370               }
371               break;
372               case 0: cTemp = (ConstructedTLV)inCTLV.getNext();
373               if (Arrays.equals(cTemp.getTagName(),
374                   this.EncryptedData.getTagName())) {
375                 this.EncryptedData = cTemp;
376               }else if (Arrays.equals(cTemp.getTagName(),
377                         SCUserCertificate.getTagName())) {
378                 this.SCUserCertificate = cTemp;
379               }
380                else
381               if (Arrays.equals(cTemp.getTagName(),
382                   SCCertificate.getTagName())) {
383                 this.SCCertificate = cTemp;
384               }
385               break;
386             default:
387               System.out.println("Error In Parsing Input Message");
388           }
389           childs--;
390         }
391       } catch (Exception e) {
392         System.out.println(
393                             "Error in ProtocolHanlder.ChildExtractionMethod
                                : " + e.getClass().getName());
```

```
394       }
395     }
396     void GenerateKeys(byte[] inbuff) {
397       byte[] DHSecretKey = null;
398       try {
399         DHSecretKey =
400           this.myProtocolHelperObject.GenerateDHSessionKeyMaterial
401           (inbuff, 0, inbuff.length);
402       } catch (Exception cE) {
403         System.out.println(
404                             "Exception At ProtocolHelperClass.GenerateKeys :
                                " + cE.getClass().getName());
405       }
406       byte[] keyGenKey = new byte[16];
407       System.arraycopy(DHSecretKey, 0, keyGenKey, 0, keyGenKey.length);
408       byte[] macInputValue = new byte[64];
409       System.arraycopy(this.SPRandomNumber.getValueBytes(), 0,
410                        macInputValue, 0, 16);
411       System.arraycopy(this.SCRandomNumber.getValueBytes(), 0,
412                        macInputValue, 16, 16);
413       System.arraycopy(DHSecretKey, 16, macInputValue, 32, 16);
414       for (int i = 48; i < 64; i++) {
415         macInputValue[i] = (byte)0x02;
416       }
417       try {
418         this.myProtocolHelperObject.GenerateMac(macInputValue, 0,
419           macInputValue.length, this.mySessionEncryptionKey, 0,
420           keyGenKey);
421       } catch (Exception cE) {
422         System.out.println(
423                             "Exception at ProtocolHandler.GenerateKeys : " +
                                cE.getClass().getName());
424       }
425       for (int i = 48; i < 64; i++) {
426         macInputValue[i] = (byte)0x03;
427       }
428       try {
429         this.myProtocolHelperObject.GenerateMac(macInputValue, 0,
430           macInputValue.length, this.mySessionMacKey, 0, keyGenKey);
431       } catch (Exception cE) {
432         System.out.println(
433                             "Exception at ProtocolHandler.GenerateKeys : " +
                                cE.getClass().getName());
434       }
435     }
436 }
```

C.5 Secure and Trusted Channel Protocol — Smart Card

The Java Card implementation of the STCP_{SC} discussed in section 6.4 is listed in subsequent sections.

C.5.1 Smart Card Implementation

In this section, we list the smart card implementation of the STCP_{SC}, and the implementation is similar to the on discussed in section C.9.

```
1  package protocolSTCPSC;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.JCSystem;
8  import javacard.framework.Util;
9  import javacard.security.AESKey;
10 import javacard.security.Key;
11 import javacard.security.KeyBuilder;
12 import javacard.security.KeyPair;
13 import javacard.security.MessageDigest;
14 import javacard.security.RSAPrivateKey;
15 import javacard.security.RSAPublicKey;
16 import javacard.security.RandomData;
17 import javacard.security.Signature;
18 import javacardx.apdu.ExtendedLength;
19 import javacardx.crypto.Cipher;
20 public class ProtocolHandler extends Applet implements ExtendedLength {
21   private byte[] SPRandomNumberArray;
22   private byte[] SPCookieArray;
23   private byte[] SCSPDHGeneratedValue;
24   private byte[] SCRandomNumberArray;
25   private byte[] SCCertificate;
26   private byte[] SPDHChallengeTag = {
27     (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x01};
28   private byte[] MessageHandlerTagOne = {
29     (byte)0x1F, (byte)0xC0, (byte)0xAA, (byte)0xAA, (byte)0x00,
         (byte)0x00,
30     (byte)0x00};
31   private byte[] MessageHandlerTagTwo = {
32     (byte)0x1F, (byte)0xC0, (byte)0xBB, (byte)0xBB, (byte)0x00,
         (byte)0x00,
33     (byte)0x00};
34   private byte[] SPIdentity = null;
35   private byte[] SPRandomNumberTag = {
36     (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x01};
37   private byte[] SPCookieTag = {
38     (byte)0x1F, (byte)0x5F, (byte)0x5B, (byte)0x01};
```

```
39  private byte[] EncryptedDataTag = {
40    (byte)0x1F, (byte)0xC0, (byte)0xFE, (byte)0x01};
41  private byte[] SignedDataTag = {
42    (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x02};
43  private byte[] MACedDataTag = {
44    (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x01};
45  private byte[] PlatformHash = {
46    (byte)0x1F, (byte)0x5F, (byte)0x5E, (byte)0xAF};
47  private byte[] SCIdentityTag = {
48    (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x02, (byte)0x00,
        (byte)0x0C,
49     (byte)0x7A, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6, (byte)
50     0xC1, (byte)0x22, (byte)0x07, (byte)0xC9, (byte)0xF9, (byte)0x8D,
51     (byte)0x11};
52  private byte[] ExponentTag = {
53    (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x01};
54  private byte[] ModulusTag = {
55    (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x02};
56  private byte[] SCDHChalleneTag = {
57    (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x02};
58  private byte[] SCRandomNumberTag = {
59    (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x02};
60  private byte[] SPCertificateTag = {
61    (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x01};
62  private byte[] SCCertificateTag = {
63    (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x02};
64  private byte[] SCProtocolInitiatorTag = {
65    (byte)0x1F, (byte)0x5F, (byte)0xA1, (byte)0xB2};
66  short PTLVDataOffset = (short)6;
67  short CTLVDataOffset = (short)7;
68  short TLVLengthOffset = (short)4;
69  short copyPointer = (short)0;
70  byte[] SCDHData;
71  final static byte CLA = (byte)0xB0;
72  final static byte StartProtocol = (byte)0x40;
73  final static byte InitiationProtocol = (byte)0xff;
74  final static short SW_CLASSNOTSUPPORTED = 0x6320;
75  final static short SW_ERROR_INS = 0x6300;
76  RandomData randomDataGen;
77  Cipher pkCipher;
78  short messageNumber = 0;
79  byte[] receivingBuffer = null;
80  short bytesLeft = 0;
81  short readCount = 0;
82  short rCount = 0;
83  short signlength = 0;
84  private RSAPublicKey dhKey = (RSAPublicKey)KeyBuilder.buildKey
85                               (KeyBuilder.TYPE_RSA_PUBLIC,
86                               KeyBuilder.LENGTH_RSA_2048, false);
87  private byte[] randomExponent;
88  final static byte GEN_KEYCONTRIBUTION = 0x01;
```

```
89  final static byte GEN_DHKEY = 0x02;
90  AESKey phCipherKey;
91  Cipher syCipher;
92  byte[] InitialisationVector = {
93    (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89,
        (byte)0x99,
94     (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)
95     0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52};
96  AESKey phMacGeneratorKey;
97  Signature phMacGenerator;
98  Signature phSign;
99  KeyPair phSCKeyPair;
100 KeyPair phUserKeyPair;
101 RSAPublicKey SPVerificationKey = null;
102 private ProtocolHandler() {
103   phMacGeneratorKey = (AESKey)KeyBuilder.buildKey
104                       (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
105                       KeyBuilder.LENGTH_AES_128, false);
106   phMacGenerator =
        Signature.getInstance(Signature.ALG_AES_MAC_128_NOPAD,
107     false);
108   phSign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
109   phSCKeyPair = new KeyPair(KeyPair.ALG_RSA, KeyBuilder.LENGTH_RSA_512);
110   phUserKeyPair = new KeyPair(KeyPair.ALG_RSA,
        KeyBuilder.LENGTH_RSA_512)
111                                 ;
112   phCipherKey = (AESKey)KeyBuilder.buildKey
113                 (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
114                 KeyBuilder.LENGTH_AES_128, false);
115   syCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
116                                 false);
117   randomDataGen = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);
118   pkCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
119   dhInitialisation();
120   phSCKeyPair.genKeyPair();
121   phUserKeyPair.genKeyPair();
122 }
123 public static void install(byte bArray[], short bOffset, byte bLength)
124                            throws ISOException {
125   new ProtocolHandler().register();
126 }
127 public void initialiseProtocol() {
128   short initialPointer = 0;
129   SCDHData = JCSystem.makeTransientByteArray((short)((short)
130     this.ClassDH.dhModulus.length + PTLVDataOffset),
131     JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
132   Util.arrayCopyNonAtomic(this.SCDHChalleneTag, (short)initialPointer,
133                           this.SCDHData, (short)0, (short)
134                           this.SCDHChalleneTag.length);
135   this.shortToBytes(SCDHData, (short)4, (short)((short)SCDHData.length -
136                     (short)PTLVDataOffset));
```

```
137     this.dhKeyConGen(this.SCDHData, this.PTLVDataOffset,
138                      ProtocolHandler.GEN_KEYCONTRIBUTION);
139     SPDHChanllengerArray = JCSystem.makeTransientByteArray((short)((short)
140       this.ClassDH.dhModulus.length + this.PTLVDataOffset),
141       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
142     SPRandomNumberArray = JCSystem.makeTransientByteArray((short)22,
143       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
144     SPCookieArray = JCSystem.makeTransientByteArray((short)22,
145       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
146     SCRandomNumberArray = JCSystem.makeTransientByteArray((short)22,
147       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
148     Util.arrayCopyNonAtomic(this.SCRandomNumberTag, (short)initialPointer,
149                             this.SCRandomNumberArray, (short)
150                             initialPointer, (short)
151                             this.SCRandomNumberTag.length);
152     this.shortToBytes(this.SCRandomNumberArray, (short)4, (short)((short)
153                       this.SCRandomNumberArray.length - (short)
154                       PTLVDataOffset));
155     try {
156       this.SCCertificate = JCSystem.makeTransientByteArray((short)86,
157         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
158       initialPointer = Util.arrayCopyNonAtomic(this.SCCertificateTag,
159                                (short)0, this.SCCertificate, (short)0,
                                     (short)
160                                this.SCCertificateTag.length);
161       this.shortToBytes(this.SCCertificate, (short)4, (short)
162                         (this.SCCertificate.length - (short)7));
163       initialPointer = Util.arrayCopyNonAtomic(this.ExponentTag, (short)0,
164                        this.SCCertificate, (short)(initialPointer +
                             (short)3), (short)
165                        this.ExponentTag.length);
166       RSAPublicKey myPublic = (RSAPublicKey)this.phSCKeyPair.getPublic();
167       short kLen = myPublic.getExponent(this.SCCertificate, (short)
168         (initialPointer + (short)2));
169       this.shortToBytes(this.SCCertificate, initialPointer, kLen);
170       initialPointer += (short)(kLen + (short)2);
171       this.SCCertificate[6]++;
172       initialPointer = Util.arrayCopyNonAtomic(this.ModulusTag, (short)0,
173                        this.SCCertificate, (short)(initialPointer), (short)
174                        this.ModulusTag.length);
175       kLen = myPublic.getModulus(this.SCCertificate, (short)
176                                  (initialPointer + (short)2));
177       this.shortToBytes(this.SCCertificate, initialPointer, kLen);
178       this.SCCertificate[6]++;
179       SPVerificationKey = (RSAPublicKey)KeyBuilder.buildKey
180                           (KeyBuilder.TYPE_RSA_PUBLIC,
181                           KeyBuilder.LENGTH_RSA_512, false);
182     } catch (Exception cE) {
183       ISOException.throwIt((short)0x6666);
184     }
185   }
```

```
186   public void process(APDU apdu)throws ISOException {
187     byte[] apduBuffer = apdu.getBuffer();
188     if (selectingApplet()) {
189       this.initialiseProtocol();
190       return ;
191     }
192     if (apduBuffer[ISO7816.OFFSET_CLA] != CLA) {
193       ISOException.throwIt(SW_CLASSNOTSUPPORTED);
194     }
195     if (apduBuffer[ISO7816.OFFSET_INS] == InitiationProtocol) {
196       receivingBuffer = JCSystem.makeTransientByteArray((short)64,
197         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
198       generateResponse((short)1);
199       apdu.setOutgoing();
200       apdu.setOutgoingLength((short)copyPointer);
201       apdu.sendBytesLong(receivingBuffer, (short)0, (short)copyPointer);
202       return ;
203     }
204     receivingBuffer = null;
205     bytesLeft = 0;
206     bytesLeft = apdu.getIncomingLength();
207     receivingBuffer = JCSystem.makeTransientByteArray(bytesLeft,
208       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
209     readCount = (short)((short)apdu.setIncomingAndReceive());
210     rCount = 0;
211     if (bytesLeft > 0) {
212       rCount = Util.arrayCopyNonAtomic(apduBuffer,
213         ISO7816.OFFSET_EXT_CDATA, receivingBuffer, rCount, readCount);
214       bytesLeft -= readCount;
215     }
216     while (bytesLeft > 0) {
217       try {
218         readCount = apdu.receiveBytes((short)0);
219         rCount = Util.arrayCopyNonAtomic(apduBuffer, (short)0,
220           receivingBuffer, rCount, readCount);
221         bytesLeft -= readCount;
222       } catch (Exception aE) {
223         ISOException.throwIt((short)0x7AAA);
224       }
225     }
226     if (this.receivingBuffer[3] == this.MessageHandlerTagOne[3]) {
227       try {
228         parseMessage(receivingBuffer);
229       } catch (Exception cE) {
230         ISOException.throwIt((short)0xA112);
231       }
232       receivingBuffer = JCSystem.makeTransientByteArray((short)600,
233         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
234       generateResponse((short)2);
235       JCSystem.requestObjectDeletion();
236       apdu.setOutgoing();
```

```
237       apdu.setOutgoingLength((short)copyPointer);
238       apdu.sendBytesLong(receivingBuffer, (short)0, (short)copyPointer);
239     } else if (this.receivingBuffer[3] == this.MessageHandlerTagTwo[3]) {
240       if (processSecondMsg(receivingBuffer)) {
241         return;
242       } else {
243         ISOException.throwIt((short)0xFA17);
244       }
245       return;
246     } else {
247       ISOException.throwIt(ProtocolHandler.SW_ERROR_INS);
248     }
249     JCSystem.requestObjectDeletion();
250   }
251   private void generateResponse(short msgNumber) {
252     short childPM1 = 0;
253     short childPM2 = 0;
254     copyPointer = 0;
255     if (msgNumber == 1) {
256       copyPointer = Util.arrayCopy(this.SCProtocolInitiatorTag, (short)0,
257                                    this.receivingBuffer, copyPointer,
258                                    (short)
259                                    this.SCProtocolInitiatorTag.length);
260       randomDataGen.generateData(this.SCRandomNumberArray,
261                                  this.PTLVDataOffset, (short)16);
262       childPM1 = copyPointer;
263       copyPointer += 2;
264       phMacGeneratorKey.setKey(this.SCRandomNumberArray,
265                                this.PTLVDataOffset);
266       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
267                           InitialisationVector, (short)0, (short)
268                           InitialisationVector.length);
269       short length = 0;
270       length = phMacGenerator.sign(SCDHData, (short)this.PTLVDataOffset,
271                                    (short)(SCDHData.length -
272                                    this.PTLVDataOffset),
273                                    this.receivingBuffer, copyPointer);
274       copyPointer += length;
275       this.shortToBytes(this.receivingBuffer, childPM1, length);
276       return;
277     } else if (msgNumber == 2) {
278       this.dhKeyConGen(this.SPDHChanllengerArray, this.PTLVDataOffset,
279                        ProtocolHandler.GEN_DHKEY);
280       keygenerator();
281       childPM1 = (short)6;
282       copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagTwo,
283         (short)0, this.receivingBuffer, copyPointer, (short)
284         this.MessageHandlerTagTwo.length);
285       copyPointer = Util.arrayCopyNonAtomic(this.SCDHData, (short)0,
286                        this.receivingBuffer, (short)copyPointer, (short)
287                        this.SCDHData.length);
```

```
288         this.receivingBuffer[childPM1]++;
289         copyPointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
290           (short)0, this.receivingBuffer, copyPointer, (short)
291           this.SCRandomNumberArray.length);
292         this.receivingBuffer[childPM1]++;
293         copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag, (short)
294           0, this.receivingBuffer, copyPointer, (short)
295           this.EncryptedDataTag.length);
296         copyPointer += 3;
297         childPM2 = (short)(copyPointer - (short)1);
298         this.receivingBuffer[childPM1]++;
299         copyPointer = Util.arrayCopyNonAtomic(this.PlatformHash, (short)0,
300           this.receivingBuffer, copyPointer,
                (short)this.PlatformHash.length)
301           ;
302         copyPointer += 2;
303         MessageDigest myHashGen = MessageDigest.getInstance
304           (MessageDigest.ALG_SHA_256, false);
305         short tempLength = (short)myHashGen.doFinal(this.ClassDH.dhModulus,
              (short)0,
306                           (short)this.ClassDH.dhModulus.length,
                                receivingBuffer,
307                           copyPointer);
308         this.receivingBuffer[childPM2]++;
309         this.shortToBytes(this.receivingBuffer, (short)(copyPointer -
              (short)
310                         2), tempLength);
311         copyPointer += tempLength;
312         copyPointer = Util.arrayCopyNonAtomic(this.SCIdentityTag, (short)0,
313           this.receivingBuffer, copyPointer, (short)
314           this.SCIdentityTag.length);
315         this.receivingBuffer[childPM2]++;
316         copyPointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
317           (short)0, this.receivingBuffer, copyPointer, (short)
318           this.SCRandomNumberArray.length);
319         this.receivingBuffer[childPM2]++;
320         copyPointer = Util.arrayCopyNonAtomic(this.SPRandomNumberArray,
321           (short)0, this.receivingBuffer, copyPointer, (short)
322           this.SPRandomNumberArray.length);
323         this.receivingBuffer[childPM2]++;
324         try {
325           this.signGenerate(this.receivingBuffer, (short)(childPM2 + (short)
326                           1), (short)(copyPointer - (short)(childPM2 +
327                           (short)1)), this.phSCKeyPair.getPrivate(),
328                           Signature.MODE_SIGN);
329         } catch (Exception cE) {
330           ISOException.throwIt((short)0x3141);
331         }
332         this.receivingBuffer[childPM2]++;
333         copyPointer = Util.arrayCopyNonAtomic(this.SCCertificate, (short)0,
334           this.receivingBuffer, copyPointer, (short)
```

```
335         this.SCCertificate.length);
336       this.receivingBuffer[childPM2]++;
337       try {
338         this.messageEncryption(this.receivingBuffer, (short)(childPM2 +
339                                (short)1), (short)(copyPointer - (short)
340                                (childPM2 + (short)1)));
341       } catch (Exception ce) {
342         ISOException.throwIt((short)(copyPointer - (short)(childPM2 +
343                              (short)1)));
344       }
345       this.shortToBytes(this.receivingBuffer, (short)(childPM2 -
              (short)2),
346                         (short)(copyPointer - childPM2 - (short)1));
347       this.macGenerate(this.receivingBuffer, (short)(childPM2 + (short)1),
348                        (short)(copyPointer - (short)(childPM2 +
                             (short)1)),
349                        Signature.MODE_SIGN);
350       this.receivingBuffer[childPM1]++;
351       copyPointer = Util.arrayCopyNonAtomic(this.SPCookieArray, (short)0,
352                        this.receivingBuffer, (short)copyPointer, (short)
353                        this.SPCookieArray.length);
354       this.receivingBuffer[childPM1]++;
355       this.shortToBytes(this.receivingBuffer, (short)(childPM1 -
              (short)2),
356                         (short)(copyPointer - (short)7));
357     }
358   }
359   boolean processSecondMsg(byte[] inArray) {
360     short inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
361     short inLength = (short)(ProtocolHandler.bytesToShort(inArray, (short)
362                      (inOffset - (short)3)));
363     if (this.macGenerate(inArray, inOffset, inLength,
364         Signature.MODE_VERIFY)) {
365       try {
366         this.phDecryption(inArray, inOffset, inLength);
367         inOffset = (short)(this.CTLVDataOffset + this.PTLVDataOffset +
368                    (short)168);
369         inLength = 3;
370         SPVerificationKey.setExponent(inArray, inOffset, inLength);
371         inOffset += (short)(inLength + this.PTLVDataOffset);
372         inLength = (short)64;
373         SPVerificationKey.setModulus(inArray, inOffset, inLength);
374         inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
375         inLength = (short)84;
376         if (this.signGenerate(inArray, inOffset, inLength,
377             SPVerificationKey, Signature.MODE_VERIFY)) {
378           return true;
379         } else {
380           ISOException.throwIt((short)0x6666);
381         }
382       } catch (Exception ce) {
```

```
383         ISOException.throwIt((short)0xAB23);
384       }
385       return true;
386     } else {
387       ISOException.throwIt((short)0xFA18);
388     }
389     return false;
390   }
391   void parseMessage(byte[] inBuffer) {
392     byte childLeft = inBuffer[(short)(this.CTLVDataOffset - (short)1)];
393     short pointer = (short)this.CTLVDataOffset;
394     try {
395       while (childLeft > 0) {
396         if (Util.arrayCompare(SPDHChallengeTag, (short)0, inBuffer,
397             pointer, (short)4) == 0) {
398           Util.arrayCopy(inBuffer, pointer, this.SPDHChanllengerArray,
399                          (short)0,
                                (short)this.SPDHChanllengerArray.length)
400                          ;
401           pointer += (short)this.SPDHChanllengerArray.length;
402         } else if (Util.arrayCompare(this.SPRandomNumberTag, (short)0,
403                    inBuffer, pointer, (short)4) == 0) {
404           Util.arrayCopyNonAtomic(inBuffer, pointer,
405                                   this.SPRandomNumberArray, (short)0,
406                                   (short)(this.SPRandomNumberArray.length))
407                                   ;
408           pointer += (short)(this.SPRandomNumberArray.length);
409         } else if (Util.arrayCompare(this.SPCookieTag, (short)0, inBuffer,
410                    pointer, (short)4) == 0) {
411           Util.arrayCopyNonAtomic(inBuffer, pointer, this.SPCookieArray,
412                                   (short)0, (short)
413                                   (this.SPCookieArray.length));
414           pointer += (short)(this.SPCookieArray.length);
415         }
416         childLeft -= (short)1;
417       }
418     } catch (Exception cE) {
419       ISOException.throwIt((short)childLeft);
420     }
421   }
422   void protocolImplementation(){}
423   void dhInitialisation() {
424     dhKey.setModulus(ClassDH.dhModulus, (short)0,
          (short)ClassDH.dhModulus.length);
425   }
426   void dhKeyConGen(byte[] inbuff, short inbuffOffset, byte Oper_Mode) {
427     switch (Oper_Mode) {
428       case GEN_KEYCONTRIBUTION:
429         randomExponent = JCSystem.makeTransientByteArray((short)32,
430           JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
431         randomDataGen.generateData(randomExponent, (short)0, (short)
```

```
                                        randomExponent.length);
        dhKey.setExponent(randomExponent, (short)0, (short)
                                randomExponent.length);
        pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
        pkCipher.doFinal(ClassDH.dhBase, (short)0,
            (short)ClassDH.dhBase.length, inbuff,
                                inbuffOffset);
        break;
      case GEN_DHKEY:
        try {
          dhKey.setExponent(randomExponent, (short)0, (short)
                                  randomExponent.length);
          pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
          SCSPDHGeneratedValue = JCSystem.makeTransientByteArray((short)
            ClassDH.dhModulus.length,
                JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
          pkCipher.doFinal(inbuff, inbuffOffset, (short)((short)
                                inbuff.length - (short)this.PTLVDataOffset),
                                SCSPDHGeneratedValue, (short)0);
        }
         catch (Exception cE) {
          ISOException.throwIt((short)0xD86E);
        }
        break;
      default:
        ISOException.throwIt((short)0x5FA1);
    }
  }
  void keygenerator() {
    AESKey sessionGenKey = (AESKey)KeyBuilder.buildKey
                              (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
                              KeyBuilder.LENGTH_AES_128, false);
    sessionGenKey.setKey(SCSPDHGeneratedValue, (short)0);
    phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
                          InitialisationVector, (short)0, (short)
                          InitialisationVector.length);
    byte[] keyGenMacData = JCSystem.makeTransientByteArray((short)64,
      JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
    short pointer = 0;
    pointer = Util.arrayCopyNonAtomic(this.SPRandomNumberArray,
      this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
    pointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
      this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
    pointer = Util.arrayCopyNonAtomic(SCSPDHGeneratedValue, (short)16,
      keyGenMacData, (short)pointer, (short)16);
    for (short i = 48; i < 64; i++) {
      keyGenMacData[i] = (byte)0x02;
    }
    phMacGenerator.sign(keyGenMacData, (short)0, (short)
                          keyGenMacData.length, SCSPDHGeneratedValue,
                              (short)
```

```
480                         0);
481     this.phCipherKey.setKey(SCSPDHGeneratedValue, (short)0);
482     for (short i = 48; i < 64; i++) {
483       keyGenMacData[i] = (byte)0x03;
484     }
485     phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
486                         InitialisationVector, (short)0, (short)
487                         InitialisationVector.length);
488     phMacGenerator.sign(keyGenMacData, (short)0, (short)
489                         keyGenMacData.length, SCSPDHGeneratedValue,
                                (short)
490                         0);
491     this.phMacGeneratorKey.setKey(SCSPDHGeneratedValue, (short)0);
492     SCSPDHGeneratedValue = null;
493     JCSystem.requestObjectDeletion();
494   }
495   void messageEncryption(byte[] inbuff, short inbuffOffset, short
496                          inbuffLength) {
497     syCipher.init(phCipherKey, Cipher.MODE_ENCRYPT, InitialisationVector,
498                   (short)0, (short)InitialisationVector.length);
499     this.shortToBytes(inbuff, (short)(inbuffOffset - 3), (short)
500                       syCipher.doFinal(inbuff, inbuffOffset, inbuffLength,
501                       inbuff, inbuffOffset));
502   }
503   void phDecryption(byte[] inbuff, short inbuffOffset, short inbuffLength)
504                     {
505     syCipher.init(phCipherKey, Cipher.MODE_DECRYPT, InitialisationVector,
506                   (short)0, (short)InitialisationVector.length);
507     syCipher.doFinal(inbuff, inbuffOffset, inbuffLength, inbuff,
508                      inbuffOffset);
509   }
510   boolean macGenerate(byte[] inbuff, short inbuffOffset, short
511                       inbuffLength, short macMode) {
512     if (macMode == Signature.MODE_SIGN) {
513       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
514                           InitialisationVector, (short)0, (short)
515                           InitialisationVector.length);
516       try {
517         copyPointer = Util.arrayCopyNonAtomic(this.MACedDataTag, (short)0,
518           this.receivingBuffer, copyPointer, (short)
519           this.MACedDataTag.length);
520         copyPointer += 2;
521       } catch (Exception ce) {
522         ISOException.throwIt((short)0xFA17);
523       }
524       try {
525         short length = (short)phMacGenerator.sign(this.receivingBuffer,
526                           inbuffOffset, inbuffLength, inbuff, copyPointer);
527         this.shortToBytes(inbuff, (short)(copyPointer - (short)2),
                 length);
528         copyPointer += length;
```

```
529       } catch (Exception ce) {
530         ISOException.throwIt((short)0x0987);
531       }
532       return true;
533     } else if (macMode == Signature.MODE_VERIFY) {
534       try {
535         phMacGenerator.init(phMacGeneratorKey, Signature.MODE_VERIFY,
536                             InitialisationVector, (short)0, (short)
537                             InitialisationVector.length);
538         return phMacGenerator.verify(this.receivingBuffer, inbuffOffset,
539                                      inbuffLength, inbuff, (short)
540                                      (inbuffOffset + inbuffLength +
541                                      this.PTLVDataOffset), (short)16);
542       } catch (Exception cE) {
543         ISOException.throwIt((short)0xC1C2);
544       }
545     }
546     return false;
547   }
548   boolean signGenerate(byte[] inbuff, short inbuffOffset, short
549                        inbufflength, Key kpSign, short signMode) {
550     if (signMode == Signature.MODE_SIGN) {
551       copyPointer = Util.arrayCopyNonAtomic(this.SignedDataTag, (short)0,
552         this.receivingBuffer, copyPointer, (short)
553         this.SignedDataTag.length);
554       copyPointer += (short)2;
555       phSign.init((RSAPrivateKey)kpSign, Signature.MODE_SIGN);
556       signlength = phSign.sign(inbuff, (short)inbuffOffset, inbufflength,
557                                inbuff, copyPointer);
558       this.shortToBytes(this.receivingBuffer, (short)(copyPointer -
              (short)
559                         2), signlength);
560       copyPointer += signlength;
561       return true;
562     } else if (signMode == Signature.MODE_VERIFY) {
563       phSign.init((RSAPublicKey)kpSign, Signature.MODE_VERIFY);
564       return phSign.verify(inbuff, inbuffOffset, inbufflength, inbuff,
565                            (short)(inbuffOffset + inbufflength +
566                            this.PTLVDataOffset), (short)64);
567     }
568     return false;
569   }
570   public static short bytesToShort(byte[] ArrayBytes) {
571     return (short)(((ArrayBytes[0] << 8)) | ((ArrayBytes[1] & 0xff)));
572   }
573   public static short bytesToShort(byte[] ArrayBytes, short arrayOffset) {
574     return (short)(((ArrayBytes[arrayOffset] << 8)) | ((ArrayBytes[(short)
575             (arrayOffset + (short)1)] & 0xff)));
576   }
577   private void shortToBytes(byte[] Array, short arrayOffset, short
        inShort)
```

```
578                                 {
579     Array[arrayOffset] = (byte)((short)(inShort & (short)0xFF00) >>
            (short)
580                                 0x0008);
581     Array[(short)(arrayOffset + (short)1)] = (byte)(inShort & (short)
582             0x00FF);
583   }
584 }
```

C.5.2 Service Provider Implementation

Following is the implementation code for the protocol hander used by the SP during the STCP_{SC}.

```
1  package javacardterminal;
2
3  import java.util.Arrays;
4  import java.security.interfaces.RSAPublicKey;
5  import java.security.spec.RSAPublicKeySpec;
6  import java.security.*;
7  import java.math.BigInteger;
8  public class ProtocolHandlerSCIn {
9    private byte[] SPIdentity = {
10     (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)0x7C, (byte)0x62,
           (byte)0x0A,
11     (byte)0x86, (byte)0x52, (byte)0xBE, (byte)0x5E, (byte)0x90, (byte)
12     0x01, (byte)0xA8, (byte)0xD6, (byte)0x6A, (byte)0xD7};
13   private byte[] SCIP = {
14     (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)0x7C};
15   private byte[] PlatformHashPreset = {
16     (byte)0xBF, (byte)0xE5, (byte)0x45, (byte)0x86, (byte)0x2C,
           (byte)0xA1,
17     (byte)0x02, (byte)0xAD, (byte)0x1E, (byte)0xED, (byte)0xDB, (byte)
18     0x5F, (byte)0xBF, (byte)0xA5, (byte)0xBF, (byte)0x85, (byte)0x5A,
19     (byte)0xC4, (byte)0x99, (byte)0x5C, (byte)0x56, (byte)0xA8, (byte)
20     0xB4, (byte)0x08, (byte)0xCE, (byte)0x3F, (byte)0xE0, (byte)0x99,
21     (byte)0xDC, (byte)0xE9, (byte)0x3A, (byte)0x9D};
22   private byte[] MessageHandlerTagOne = {(byte)0xAA, (byte)0xAA};
23   private byte[] MessageHandlerTagTwo = {(byte)0xBB, (byte)0xBB};
24   private byte[] SPIdentityTag = {(byte)0x5F, (byte)0x01};
25   private byte[] SPDHChallengeTag = {(byte)0x5C, (byte)0x01};
26   private byte[] SPSignatureCertTag = {(byte)0xF0, (byte)0xF01};
27   private byte[] SPSigVerificationKeyTag = {(byte)0x51, (byte)0x01};
28   private byte[] SPRandomNumberTag = {(byte)0x5A, (byte)0x01};
29   private byte[] SPCookieTag = {(byte)0x5B, (byte)0x01};
30   private byte[] EncryptedDataTag = {(byte)0xFE, (byte)0x01};
31   private byte[] MACedDataTag = {(byte)0x5D, (byte)0x01};
32   private byte[] SignedDataTag = {(byte)0x5D, (byte)0x02};
33   private byte[] PublicExponentTag = {(byte)0xEE, (byte)0x01};
34   private byte[] PublicModulusTag = {(byte)0xEE, (byte)0x02};
35   private byte[] SCDHChallengeTag = {(byte)0x5C, (byte)0x02};
```

```
36  private byte[] SCRandomNumberTag = {(byte)0x5A, (byte)0x02};
37  private byte[] SCIdentityTag = {(byte)0x5F, (byte)0x02};
38  private byte[] SCUserCertificateTag = {(byte)0xF0, (byte)0x03};
39  private byte[] SCCertificateTag = {(byte)0xF0, (byte)0x02};
40  private byte[] PlatformHashTag = {(byte)0x5E, (byte)0xAF};
41  private byte[] UserIdentityTag = {(byte)0x5F, (byte)0x03};
42  private byte[] SCProtocolInitiatorTag = {(byte)0xA1, (byte)0xB2};
43  public ConstructedTLV MessageHandler = ConstructedTLV.getConstructedTLV
44    (MessageHandlerTagOne);
45  private ConstructedTLV SPSignatureCertificate =
46    ConstructedTLV.getConstructedTLV(SPSignatureCertTag);
47  private PrimitiveTLV SPIdentityTLV = PrimitiveTLV.getPrimitiveTLV
48    (SPIdentityTag, SPIdentity);
49  private PrimitiveTLV SPSigVerificationKey = PrimitiveTLV.getPrimitiveTLV
50    (this.SPSigVerificationKeyTag);
51  private PrimitiveTLV SPDHChanllenger = PrimitiveTLV.getPrimitiveTLV
52    (this.SPDHChallengeTag);
53  private PrimitiveTLV SPRandomNumber = PrimitiveTLV.getPrimitiveTLV
54    (this.SPRandomNumberTag);
55  private PrimitiveTLV SPCookie = PrimitiveTLV.getPrimitiveTLV
56    (this.SPCookieTag);
57  private ConstructedTLV EncryptedData = ConstructedTLV.getConstructedTLV
58    (this.EncryptedDataTag);
59  private PrimitiveTLV MACedData = PrimitiveTLV.getPrimitiveTLV
60    (this.MACedDataTag);
61  private PrimitiveTLV SignedData = PrimitiveTLV.getPrimitiveTLV
62    (this.SignedDataTag);
63  private PrimitiveTLV PublicExponent = PrimitiveTLV.getPrimitiveTLV
64    (this.PublicExponentTag);
65  private PrimitiveTLV PublicModulus = PrimitiveTLV.getPrimitiveTLV
66    (this.PublicModulusTag);
67  private PrimitiveTLV SCDHChallenge = PrimitiveTLV.getPrimitiveTLV
68    (this.SCDHChallengeTag);
69  private PrimitiveTLV SCRandomNumber = PrimitiveTLV.getPrimitiveTLV
70    (this.SCRandomNumberTag);
71  private PrimitiveTLV SCIdentity = PrimitiveTLV.getPrimitiveTLV
72    (SCIdentityTag);
73  private ConstructedTLV SCUserCertificate =
74    ConstructedTLV.getConstructedTLV(this.SCUserCertificateTag);
75  private ConstructedTLV SCCertificate = ConstructedTLV.getConstructedTLV
76    (this.SCCertificateTag);
77  private PrimitiveTLV PlatformHash = PrimitiveTLV.getPrimitiveTLV
78    (this.PlatformHashTag);
79  private PrimitiveTLV UserIdentity = PrimitiveTLV.getPrimitiveTLV
80    (this.UserIdentityTag);
81  private PrimitiveTLV SCProtocolInitiator = PrimitiveTLV.getPrimitiveTLV
82    (this.SCProtocolInitiatorTag);
83  private ProtocolHelperClass myProtocolHelperObject = new
84    ProtocolHelperClass();
85  private byte[] mySessionEncryptionKey = new byte[16];
86  private byte[] mySessionMacKey = new byte[16];
```

```
87  private PublicKey SCUserVerificationKey = null;
88  private PublicKey SCVerificationKey = null;
89  public ProtocolHandlerSCIn() {
90    myProtocolHelperObject.protocolInitialise();
91    RSAPublicKey tempKey = (RSAPublicKey)
92                              myProtocolHelperObject.getPublicKey();
93    byte[] tempExponent = tempKey.getPublicExponent().toByteArray();
94    this.PublicExponent.initialisationPTLV(this.PublicExponentTag,
95      tempExponent.length);
96    this.PublicExponent.setTlvValues(tempExponent);
97    byte[] tempModulus = tempKey.getModulus().toByteArray();
98    this.PublicModulus.initialisationPTLV(this.PublicModulusTag,
99      (tempModulus.length - 1));
100   this.PublicModulus.setTlvValues(tempModulus, 1, (tempModulus.length -
101                                    1));
102   SPSignatureCertificate.addPTLV(this.PublicExponent);
103   SPSignatureCertificate.addPTLV(this.PublicModulus);
104 }
105 public void initialiseProtocol() {
106   try {
107     this.SPDHChanllenger.setTlvValues
108       (this.myProtocolHelperObject.GenerateDHPublicValue());
109     this.MessageHandler.addPTLV(this.SPDHChanllenger);
110   } catch (Exception cE) {
111     System.out.println(
112                         "Error ProtocolHandler.initialiseProtocol Option
                                = 1, : " + cE.getClass().getName());
113   }
114 }
115 public byte[] outMessageProcessing(int Counter) {
116   if (Counter == 1) {
117     try {
118       this.SPRandomNumber.setTlvValues
119         (this.myProtocolHelperObject.getRandomNumber());
120       this.MessageHandler.addPTLV(this.SPRandomNumber);
121       byte[] temp = new byte[(this.SCProtocolInitiator.getValueBytes()
122                               .length +
123                               this.SPDHChanllenger.getValueLength() +
124                               this.SPRandomNumber.getValueLength())];
125       System.arraycopy(this.SPDHChanllenger.getValueBytes(), 0, temp, 0,
126                        this.SPDHChanllenger.getValueLength());
127       System.arraycopy(this.SPRandomNumber.getValueBytes(), 0, temp,
128                        this.SPDHChanllenger.getValueLength(),
129                        this.SPRandomNumber.getValueLength());
130       System.arraycopy(this.SCProtocolInitiator.getValueBytes(), 0,
              temp,
131                        temp.length -
132                        this.SCProtocolInitiator.getValueBytes().length,
133                        this.SCProtocolInitiator.getValueBytes().length);
134       byte[] result = new byte[16];
135       this.myProtocolHelperObject.GenerateMac(temp, 0, temp.length,
```

```
136           result, 0, this.myProtocolHelperObject.myLongTermMacKey);
137         this.SPCookie.setTlvValues(result);
138         this.MessageHandler.addPTLV(this.SPCookie);
139       } catch (Exception cE) {
140         System.out.println(
141                             "Error ProtocolHandler.inMessageProcessing
                                    Option = 1, : " + cE.getClass().getName());
142       }
143     } else if (Counter == 2) {
144       try {
145         this.EncryptedData.initialisationCTLV(this.EncryptedDataTag);
146         this.EncryptedData.addPTLV(this.SPIdentityTLV);
147         this.EncryptedData.addPTLV(this.SCIdentity);
148         this.EncryptedData.addPTLV(this.SPRandomNumber);
149         this.EncryptedData.addPTLV(this.SCRandomNumber);
150         this.myProtocolHelperObject.SignatureMethod
151           (this.EncryptedData.getValueBytes(), 0,
152           this.EncryptedData.getValueBytes().length,
153           this.SignedData.getBytesTlvRepresentation(), 6, null,
154           ProtocolHelperClass.SIGN_MODE_GENERATION);
155         this.EncryptedData.addPTLV(this.SignedData);
156         this.EncryptedData.addCTLV(this.SPSignatureCertificate);
157         this.myProtocolHelperObject.GenerateEncryption
158           (this.EncryptedData.getValueBytes(), 0,
159           this.EncryptedData.getValueBytes().length,
160           this.EncryptedData.getBytesTlvRepresentation(), 7,
161           this.mySessionEncryptionKey);
162         this.MACedData.initialisationPTLV(this.MACedDataTag, 16);
163         this.myProtocolHelperObject.GenerateMac
164           (this.EncryptedData.getValueBytes(), 0,
165           this.EncryptedData.getTagValueLength(),
166           this.MACedData.getBytesTlvRepresentation(), 6,
167           this.mySessionMacKey);
168         this.MessageHandler.initialisationCTLV(this.MessageHandlerTagTwo);
169         this.MessageHandler.addCTLV(EncryptedData);
170         this.MessageHandler.addPTLV(this.MACedData);
171         this.MessageHandler.addPTLV(this.SPCookie);
172       } catch (Exception cE) {
173         System.out.println(
174                             "Error ProtocolHandler.inMessageProcessing
                                    Option = 1, : " + cE.getClass().getName());
175       }
176     } else {
177       System.out.println(
178                           "Protocol Stoped : Illegal Message Value
                                (ProtocolHanlder.inMessageProcessing()");
179     }
180     return this.MessageHandler.getBytesTlvRepresentation();
181   }
182   public boolean inMessageProcessing(byte[] inMessage, int Counter) {
183     try {
```

```
184         if (Counter == 1) {
185           this.SCProtocolInitiator.setBytesTlvRepresentation(inMessage, 0,
186             22);
187         } else
188         if (Counter == 2) {
189           this.MessageHandler.reset();
190           this.EncryptedData.reset();
191           this.MessageHandler.setBytesTlvRepresentation(inMessage, 0,
192             inMessage.length - 2);
193           this.childExtractionFromCTLV(this.MessageHandler);
194           GenerateKeys(this.SCDHChallenge.getValueBytes());
195           byte[] temp = new byte[16];
196           this.myProtocolHelperObject.GenerateMac
197             (this.EncryptedData.getValueBytes(), 0,
198             this.EncryptedData.getValueBytes().length, temp, 0,
199             this.mySessionMacKey);
200           if (Arrays.equals(this.MACedData.getValueBytes(), temp)){}
201            else {
202             System.out.println(
203                                 "Integrity Check Failure : ERROR at
                                        ProtocolHandler.inMessageProcessing \n");
204             System.exit(0);
205           }
206           this.myProtocolHelperObject.GenerateDecryption
207             (this.EncryptedData.getValueBytes(), 0,
208             this.EncryptedData.getValueBytes().length,
209             this.EncryptedData.getBytesTlvRepresentation(), 7,
210             this.mySessionEncryptionKey);
211           this.childExtractionFromCTLV(EncryptedData);
212           if (Arrays.equals(PlatformHashPreset,
213               this.PlatformHash.getValueBytes())){}
214            else {
215             System.out.println("Platform Digest Not Verified");
216           }
217           childExtractionFromCTLV(this.SCCertificate);
218           BigInteger SCpublicExponent = new BigInteger(byteToString
219             (this.PublicExponent.getValueBytes()), 16);
220           BigInteger SCpublicModulus = new BigInteger(byteToString
221             (this.PublicModulus.getValueBytes()), 16);
222           KeyFactory factory = KeyFactory.getInstance("RSA");
223           SCVerificationKey = (PublicKey)factory.generatePublic(new
224                                 RSAPublicKeySpec(SCpublicModulus,
225                                 SCpublicExponent));
226           temp = new byte[(this.PlatformHash.getTagLength() +
227                            this.SCIdentity.getTagLength() +
228                            this.SCRandomNumber.getTagLength() +
229                            this.SPRandomNumber.getTagLength())];
230           System.arraycopy(this.EncryptedData.getBytesTlvRepresentation(),
                7,
231                            temp, 0, temp.length);
232           if (this.myProtocolHelperObject.SignatureMethod(temp, 0,
```

```
233             temp.length, this.SignedData.getValueBytes(), 0,
234             SCVerificationKey,
                    ProtocolHelperClass.SIGN_MODE_VERIFICATION))
235             {}
236           else {
237            System.out.println(
238                              "Signature Verification Failed..... Check
                                     code");
239         }
240       }
241     } catch (Exception cE) {
242       System.out.println("Error in ProtocolHandler.inMessageProcessing : "
243                          + cE.getClass().getName());
244     }
245     return true;
246   }
247   public static String byteToString(byte[] inArray) {
248     byte[] HEX_CHAR_TABLE =  {
249       (byte)'0', (byte)'1', (byte)'2', (byte)'3', (byte)'4', (byte)'5',
250        (byte)'6', (byte)'7', (byte)'8', (byte)'9', (byte)'a', (byte)'b',
251        (byte)'c', (byte)'d', (byte)'e', (byte)'f'
252     };
253     byte[] hex = new byte[2 * inArray.length];
254     int index = 0;
255     for (byte b: inArray) {
256       int v = b & 0xFF;
257       hex[index++] = HEX_CHAR_TABLE[v >>> 4];
258       hex[index++] = HEX_CHAR_TABLE[v & 0xF];
259     }
260     try {
261       return new String(hex, "ASCII");
262     } catch (Exception cE) {
263       System.out.println("Exception in bytesToString : " +
              cE.getMessage())
264                          ;
265     }
266     return "Error";
267   }
268   void childExtractionFromCTLV(ConstructedTLV inCTLV) {
269     try {
270       int childs = inCTLV.getChildNumbers();
271       PrimitiveTLV pTemp = null;
272       ConstructedTLV cTemp = null;
273       while (childs > 0) {
274         switch (inCTLV.nextType()) {
275           case 1:
276             pTemp = (PrimitiveTLV)inCTLV.getNext();
277             if (Arrays.equals(pTemp.getTagName(),
278                  this.SCDHChallenge.getTagName())) {
279               this.SCDHChallenge = pTemp;
280             }else if (Arrays.equals(pTemp.getTagName(),
```

```
281                         this.SCRandomNumber.getTagName())) {
282               this.SCRandomNumber = pTemp;
283             }else if (Arrays.equals(pTemp.getTagName(),
284                         this.MACedData.getTagName())) {
285               this.MACedData = pTemp;
286             }else if (Arrays.equals(pTemp.getTagName(),
287                         this.SPCookie.getTagName())) {
288               if (Arrays.equals(pTemp.getBytesTlvRepresentation(),
289                   this.SPCookie.getBytesTlvRepresentation())){}
290             }else if (Arrays.equals(pTemp.getTagName(),
291                         this.SCIdentity.getTagName())) {
292               this.SCIdentity = pTemp;
293             }else if (Arrays.equals(pTemp.getTagName(),
294                         this.SignedData.getTagName())) {
295               this.SignedData = pTemp;
296             }else if (Arrays.equals(pTemp.getTagName(),
297                         this.PublicExponent.getTagName())) {
298               this.PublicExponent = pTemp;
299             }else if (Arrays.equals(pTemp.getTagName(),
300                         this.PublicModulus.getTagName())) {
301               this.PublicModulus = pTemp;
302             }else if (Arrays.equals(pTemp.getTagName(),
303                         this.PlatformHash.getTagName())) {
304               this.PlatformHash = pTemp;
305             }else if (Arrays.equals(pTemp.getTagName(),
306                         this.UserIdentity.getTagName())) {
307               this.UserIdentity = pTemp;
308             }
309             break;
310           case 0:
311             cTemp = (ConstructedTLV)inCTLV.getNext();
312             if (Arrays.equals(cTemp.getTagName(),
313                 this.EncryptedData.getTagName())) {
314               this.EncryptedData = cTemp;
315             }else if (Arrays.equals(cTemp.getTagName(),
316                         SCUserCertificate.getTagName())) {
317               this.SCUserCertificate = cTemp;
318             }else if (Arrays.equals(cTemp.getTagName(),
319                         SCCertificate.getTagName())) {
320               this.SCCertificate = cTemp;
321             }
322             break;
323           default:
324             System.out.println("Error In Parsing Input Message");
325         }
326         childs--;
327       }
328     } catch (Exception e) {
329       System.out.println(
330                           "Error in ProtocolHanlder.ChildExtractionMethod
                                : " + e.getClass().getName());
```

```
331     }
332   }
333   void GenerateKeys(byte[] inbuff) {
334     byte[] DHSecretKey = null;
335     try {
336       DHSecretKey =
337         this.myProtocolHelperObject.GenerateDHSessionKeyMaterial(inbuff,
              0,
338         inbuff.length);
339     } catch (Exception cE) {
340       System.out.println(
341                           "Exception At ProtocolHelperClass.GenerateKeys :
                                " + cE.getClass().getName());
342     }
343     byte[] keyGenKey = new byte[16];
344     System.arraycopy(DHSecretKey, 0, keyGenKey, 0, keyGenKey.length);
345     byte[] macInputValue = new byte[64];
346     System.arraycopy(this.SPRandomNumber.getValueBytes(), 0,
          macInputValue,
347                           0, 16);
348     System.arraycopy(this.SCRandomNumber.getValueBytes(), 0,
          macInputValue,
349                           16, 16);
350     System.arraycopy(DHSecretKey, 16, macInputValue, 32, 16);
351     for (int i = 48; i < 64; i++) {
352       macInputValue[i] = (byte)0x02;
353     }
354     try {
355       this.myProtocolHelperObject.GenerateMac(macInputValue, 0,
356         macInputValue.length, this.mySessionEncryptionKey, 0, keyGenKey);
357     } catch (Exception cE) {
358       System.out.println("Exception at ProtocolHandler.GenerateKeys : " +
359                           cE.getClass().getName());
360     }
361     for (int i = 48; i < 64; i++) {
362       macInputValue[i] = (byte)0x03;
363     }
364     try {
365       this.myProtocolHelperObject.GenerateMac(macInputValue, 0,
366         macInputValue.length, this.mySessionMacKey, 0, keyGenKey);
367     } catch (Exception cE) {
368       System.out.println("Exception at ProtocolHandler.GenerateKeys : " +
369                           cE.getClass().getName());
370     }
371   }
372 }
```

C.6 Application Acquisition and Contractual Agreement Protocol

The Java Card implementation of the STCP_{ACA} discussed in section 6.5 is listed in subsequent sections.

C.6.1 Smart Card Implementation

Following is the implementation of the smart card protocol handler that supports the STCP_{ACA}.

```
1  package protocolACAP;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.JCSystem;
8  import javacard.framework.Util;
9  import javacard.security.AESKey;
10 import javacard.security.Key;
11 import javacard.security.KeyBuilder;
12 import javacard.security.KeyPair;
13 import javacard.security.MessageDigest;
14 import javacard.security.RSAPrivateKey;
15 import javacard.security.RSAPublicKey;
16 import javacard.security.RandomData;
17 import javacard.security.Signature;
18 import javacardx.apdu.ExtendedLength;
19 import javacardx.crypto.Cipher;
20 public class ProtocolHandler extends Applet implements ExtendedLength {
21   private byte[] SPDHChanllengerArray;
22   private byte[] SPRandomNumberArray;
23   private byte[] SPCookieArray;
24   private byte[] SCSPDHGeneratedValue;
25   private byte[] SCRandomNumberArray;
26   private byte[] SCUserCertificate;
27   private byte[] SCCertificate;
28   private byte[] SID;
29   private byte[] SPDHChallengeTag = {
30     (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x01};
31   private byte[] MessageHandlerTagOne = {
32     (byte)0x1F, (byte)0xC0, (byte)0xAA, (byte)0xAA, (byte)0x00,
           (byte)0x00,
33      (byte)0x00};
34   private byte[] MessageHandlerTagTwo = {
35     (byte)0x1F, (byte)0xC0, (byte)0xBB, (byte)0xBB, (byte)0x00,
           (byte)0x00,
36      (byte)0x00};
```

```
37  private byte[] MessageHandlerTagThree = {
38    (byte)0x1F, (byte)0xC0, (byte)0xCC, (byte)0xCC, (byte)0x00,
          (byte)0x00,
39     (byte)0x00};
40  private byte[] MessageHandlerTagSCTSM = {
41    (byte)0x1F, (byte)0xC0, (byte)0xFF, (byte)0xFF, (byte)0x00,
          (byte)0x00,
42     (byte)0x00};
43  private byte[] SPIdentityTag = {
44    (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x01};
45  private byte[] SPSignatureCertTag = {
46    (byte)0xF0, (byte)0xF01};
47  private byte[] SPSigVerificationKeyTag = {
48    (byte)0x1F, (byte)0x5F, (byte)0x51, (byte)0x01};
49  private byte[] SPIdentity = null;
50  private byte[] AppIdentity = null;
51  private byte[] SPSignatureCert = null;
52  private byte[] SPRandomNumberTag = {
53    (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x01};
54  private byte[] SPCookieTag = {
55    (byte)0x1F, (byte)0x5F, (byte)0x5B, (byte)0x01};
56  private byte[] EncryptedDataTag = {
57    (byte)0x1F, (byte)0xC0, (byte)0xFE, (byte)0x01};
58  private byte[] SignedDataTag = {
59    (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x02};
60  private byte[] MACedDataTag = {
61    (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x01};
62  private byte[] PlatformHash = {
63    (byte)0x1F, (byte)0x5F, (byte)0x5E, (byte)0xAF};
64  private byte[] SCIdentityTag = {
65    (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x02, (byte)0x00,
          (byte)0x12,
66     (byte)0x7A, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6, (byte)
67     0xC1, (byte)0x22, (byte)0x07, (byte)0xC9, (byte)0xF9, (byte)0x8D,
68     (byte)0x11, (byte)0xED, (byte)0x34, (byte)0xDB, (byte)0xF6, (byte)
69     0x0B, (byte)0x2C};
70  private byte[] UserIdentity = {
71    (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x03, (byte)0x00,
          (byte)0x14,
72     (byte)0x7A, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6, (byte)
73     0xC1, (byte)0x22, (byte)0x07, (byte)0xC9, (byte)0xC9, (byte)0x8D,
74     (byte)0xD1, (byte)0xED, (byte)0xFC, (byte)0xDB, (byte)0xF6, (byte)
75     0x0B, (byte)0x2C, (byte)0x0B, (byte)0x2C};
76  private byte[] TSMIdentity = {
77    (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x04, (byte)0x00,
          (byte)0x12,
78     (byte)0x7d, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6, (byte)
79     0xC1, (byte)0x2e, (byte)0x07, (byte)0xe9, (byte)0x69, (byte)0x8D,
80     (byte)0x11, (byte)0xEf, (byte)0x34, (byte)0xfB, (byte)0xFe, (byte)
81     0x0B, (byte)0x2C};
82  private byte[] CardID = {
```

```
83      (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x05, (byte)0x00,
            (byte)0x12,
84       (byte)0x7d, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6, (byte)
85       0xC1, (byte)0x2e, (byte)0x06, (byte)0xe9, (byte)0xe9, (byte)0x8D,
86       (byte)0x11, (byte)0xEf, (byte)0x37, (byte)0xfB, (byte)0xFe, (byte)
87       0x0B, (byte)0x2C};
88    private byte[] ExponentTag = {
89      (byte)0x14, (byte)0x5F, (byte)0x5E, (byte)0x01};
90    private byte[] ModulusTag = {
91      (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x02};
92    private byte[] SCDHChalleneTag = {
93      (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x02};
94    private byte[] SCRandomNumberTag = {
95      (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x02};
96    private byte[] SPCertificateTag = {
97      (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x01};
98    private byte[] SCCertificateTag = {
99      (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x02};
100   private byte[] SCUserCertificateTag = {
101     (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x03};
102   short PTLVDataOffset = (short)6;
103   short CTLVDataOffset = (short)7;
104   short TLVLengthOffset = (short)4;
105   short copyPointer = (short)0;
106   byte[] SCDHData;
107   final static byte CLA = (byte)0xB0;
108   final static byte StartProtocol = (byte)0x40;
109   final static byte InitiationProtocol = (byte)0xF1;
110   private static final byte PhaseTwo = (byte)0xF2;
111   private static final byte PhaseThree = (byte)0xF3;
112   final static short SW_CLASSNOTSUPPORTED = 0x6320;
113   final static short SW_ERROR_INS = 0x6300;
114   RandomData randomDataGen;
115   Cipher pkCipher;
116   short messageNumber = 0;
117   byte[] receivingBuffer = null;
118   short bytesLeft = 0;
119   short readCount = 0;
120   short rCount = 0;
121   short signlength = 0;
122   private RSAPublicKey dhKey = (RSAPublicKey)KeyBuilder .buildKey
123                                (KeyBuilder.TYPE_RSA_PUBLIC,
124                                KeyBuilder.LENGTH_RSA_2048, false);
125   private byte[] randomExponent;
126   final static byte GEN_KEYCONTRIBUTION = 0x01;
127   final static byte GEN_DHKEY = 0x02;
128   private byte[] myLongTermEncryptionKey = {
129     (byte)0x9D, (byte)0xF3, (byte)0x0B, (byte)0x5C, (byte)0x8F,
            (byte)0xFD,
130      (byte)0xAC, (byte)0x50, (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)
131      0x7B, (byte)0x89, (byte)0x99, (byte)0x8C, (byte)0xAF};
```

```
132   private byte[] myLongTermMacKey =  {
133     (byte)0x74, (byte)0x86, (byte)0x6A, (byte)0x08, (byte)0xCF,
          (byte)0xE4,
134      (byte)0xFF, (byte)0xE3, (byte)0xA6, (byte)0x82, (byte)0x4A, (byte)
135      0x4E, (byte)0x10, (byte)0xB9, (byte)0xA6, (byte)0xF0};
136   AESKey phCipherKey;
137   Cipher syCipher;
138   byte[] InitialisationVector =  {
139     (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89,
          (byte)0x99,
140      (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)
141      0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52};
142   AESKey phMacGeneratorKey;
143   Signature phMacGenerator;
144   Signature phSign;
145   KeyPair phSCKeyPair;
146   KeyPair phUserKeyPair;
147   RSAPublicKey SPVerificationKey = null;
148   RSAPublicKey TSMVerificationKey = null;
149   private ProtocolHandler() {
150     phMacGeneratorKey = (AESKey)KeyBuilder.buildKey
151                         (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
152                         KeyBuilder.LENGTH_AES_128, false);
153     phMacGenerator =
          Signature.getInstance(Signature.ALG_AES_MAC_128_NOPAD,
154       false);
155     phSign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
156     phSCKeyPair = new KeyPair(KeyPair.ALG_RSA, KeyBuilder.LENGTH_RSA_512);
157     phUserKeyPair = new KeyPair(KeyPair.ALG_RSA,
          KeyBuilder.LENGTH_RSA_512)
158                                     ;
159     phCipherKey = (AESKey)KeyBuilder.buildKey
160                   (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
161                   KeyBuilder.LENGTH_AES_128, false);
162     syCipher = Cipher .getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
163                                    false);
164     randomDataGen = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);
165     pkCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
166     dhInitialisation();
167     phSCKeyPair.genKeyPair();
168     phUserKeyPair.genKeyPair();
169   }
170   public static void install(byte bArray[], short bOffset, byte bLength)
171                              throws ISOException {
172     new ProtocolHandler().register();
173   }
174   public void initialiseProtocol() {
175     SID = JCSystem.makeTransientByteArray((short)16,
176       JCSystem.CLEAR_ON_RESET);
177     short initialPointer = 0;
178     SCDHData = JCSystem.makeTransientByteArray((short)((short)
```

```
179         this.ClassDH.dhModulus.length + PTLVDataOffset),
180         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
181       Util.arrayCopyNonAtomic(this.SCDHChalleneTag, (short)initialPointer,
182                               this.SCDHData, (short)0, (short)
183                               this.SCDHChalleneTag.length);
184       this.shortToBytes(SCDHData, (short)4, (short)((short)SCDHData.length -
185                         (short)PTLVDataOffset));
186       this.dhKeyConGen(this.SCDHData, this.PTLVDataOffset,
187                        ProtocolHandler.GEN_KEYCONTRIBUTION);
188       SPDHChanllengerArray = JCSystem.makeTransientByteArray((short)((short)
189         this.ClassDH.dhModulus.length + this.PTLVDataOffset),
190         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
191       SPRandomNumberArray = JCSystem.makeTransientByteArray((short)22,
192         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
193       SPCookieArray = JCSystem.makeTransientByteArray((short)22,
194         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
195       SCRandomNumberArray = JCSystem.makeTransientByteArray((short)22,
196         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
197       Util.arrayCopyNonAtomic(this.SCRandomNumberTag, (short)initialPointer,
198                               this.SCRandomNumberArray, (short)
199                               initialPointer, (short)
200                               this.SCRandomNumberTag.length);
201       this .shortToBytes(this.SCRandomNumberArray, (short)4, (short)((short)
202                          this.SCRandomNumberArray.length - (short)
203                          PTLVDataOffset));
204       try {
205         this.SCUserCertificate = JCSystem.makeTransientByteArray((short)86,
206           JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
207         initialPointer = Util.arrayCopyNonAtomic(this.SCUserCertificateTag,
208           (short)0, this.SCUserCertificate, (short)0, (short)
209           this.SCUserCertificateTag.length);
210         this.shortToBytes(this.SCUserCertificate, (short)4, (short)
211                           (this.SCUserCertificate.length - (short)7));
212         initialPointer = Util.arrayCopyNonAtomic(this.ExponentTag, (short)0,
213                             this.SCUserCertificate, (short)(initialPointer +
                                  (short)3), (short)
214                             this.ExponentTag.length);
215         RSAPublicKey myPublic = (RSAPublicKey)this.phUserKeyPair
              .getPublic()
216                                 ;
217         short kLen = myPublic.getExponent(this.SCUserCertificate, (short)
218           (initialPointer + (short)2));
219         this.shortToBytes(this.SCUserCertificate, initialPointer, kLen);
220         initialPointer += (short)(kLen + (short)2);
221         this.SCUserCertificate[6]++;
222         initialPointer = Util.arrayCopyNonAtomic(this.ModulusTag, (short)0,
223                             this.SCUserCertificate, (short)(initialPointer),
                                  (short)
224                             this.ModulusTag.length);
225         kLen = myPublic.getModulus(this.SCUserCertificate, (short)
226                                    (initialPointer + (short)2));
```

```
227       this.shortToBytes(this.SCUserCertificate, initialPointer, kLen);
228       this.SCUserCertificate[6]++;
229       this.SPIdentity = JCSystem.makeTransientByteArray((short)24,
230         JCSystem.MEMORY_TYPE_TRANSIENT_RESET);
231       this.AppIdentity = JCSystem.makeTransientByteArray((short)28,
232         JCSystem.MEMORY_TYPE_TRANSIENT_RESET);
233       SPVerificationKey = (RSAPublicKey)KeyBuilder.buildKey
234                           (KeyBuilder.TYPE_RSA_PUBLIC,
235                           KeyBuilder.LENGTH_RSA_512, false);
236       TSMVerificationKey = (RSAPublicKey)KeyBuilder.buildKey
237                            (KeyBuilder.TYPE_RSA_PUBLIC,
238                            KeyBuilder.LENGTH_RSA_512, false);
239     } catch (Exception cE) {
240       ISOException.throwIt((short)0xCCCC);
241     }
242     try {
243       this.SCCertificate = JCSystem.makeTransientByteArray((short)86,
244         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
245       initialPointer = Util.arrayCopyNonAtomic(this.SCCertificateTag,
246                                (short)0, this.SCCertificate, (short)0,
                                     (short)
247                                this.SCCertificateTag.length);
248       this.shortToBytes(this.SCCertificate, (short)4, (short)
249                         (this.SCCertificate.length - (short)7));
250       initialPointer = Util.arrayCopyNonAtomic(this.ExponentTag, (short)0,
251                        this.SCCertificate, (short)(initialPointer +
                             (short)3), (short)
252                        this.ExponentTag.length);
253       RSAPublicKey myPublic = (RSAPublicKey)this.phSCKeyPair.getPublic();
254       short kLen = myPublic.getExponent(this.SCCertificate, (short)
255         (initialPointer + (short)2));
256       this.shortToBytes(this.SCCertificate, initialPointer, kLen);
257       initialPointer += (short)(kLen + (short)2);
258       this.SCCertificate[6]++;
259       initialPointer = Util.arrayCopyNonAtomic(this.ModulusTag, (short)0,
260                        this.SCCertificate, (short)(initialPointer), (short)
261                        this.ModulusTag.length);
262       kLen = myPublic.getModulus(this.SCCertificate, (short)
263                                  (initialPointer + (short)2));
264       this.shortToBytes(this.SCCertificate, initialPointer, kLen);
265       this.SCCertificate[6]++;
266     } catch (Exception cE) {
267       ISOException.throwIt((short)0x6666);
268     }
269   }
270   public void process(APDU apdu)throws ISOException {
271     byte[] apduBuffer = apdu.getBuffer();
272     if (selectingApplet()) {
273       return ;
274     }
275     if (apduBuffer[ISO7816.OFFSET_CLA] != CLA) {
```

```
276         ISOException.throwIt(SW_CLASSNOTSUPPORTED);
277       }
278       if (apduBuffer[ISO7816.OFFSET_INS] == InitiationProtocol) {
279         this.initialiseProtocol();
280         return ;
281       }
282       if (apduBuffer[ISO7816.OFFSET_INS] == PhaseTwo) {
283         this.AppDownloadCompleted(apdu);
284       }
285       if (apduBuffer[ISO7816.OFFSET_INS] == PhaseThree) {
286         receivingBuffer = JCSystem.makeTransientByteArray((short)568,
287           JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
288         this.SCTSMChargeRequest(apdu);
289         JCSystem.requestObjectDeletion();
290         apdu.setOutgoing();
291         apdu.setOutgoingLength((short)copyPointer);
292         apdu.sendBytesLong(receivingBuffer, (short)0, (short)copyPointer);
293         JCSystem.requestObjectDeletion();
294         return ;
295       }
296       bytesLeft = apdu.getIncomingLength();
297       if (bytesLeft > 255) {
298         receivingBuffer = JCSystem.makeTransientByteArray(bytesLeft,
299           JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
300         readCount = (short)((short)apdu.setIncomingAndReceive());
301         rCount = 0;
302         short bytesRead = 0;
303         while (bytesLeft > 0) {
304           try {
305             rCount = Util.arrayCopyNonAtomic(apduBuffer, (short)
306               ISO7816.OFFSET_EXT_CDATA, receivingBuffer, rCount, readCount);
307             bytesLeft -= readCount;
308             if (bytesLeft != 0) {
309               readCount = apdu
                      .receiveBytes((short)ISO7816.OFFSET_EXT_CDATA)
310                 ;
311             }
312           } catch (Exception aE) {
313             ISOException.throwIt((short)bytesRead);
314           }
315         }
316       } else {
317         try {
318           receivingBuffer = JCSystem.makeTransientByteArray(bytesLeft,
319             JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
320           Util.arrayCopyNonAtomic(apduBuffer, ISO7816.OFFSET_CDATA,
321                                   this.receivingBuffer, (short)0, (short)
322                                   this.receivingBuffer.length);
323         } catch (Exception cE) {
324           ISOException.throwIt((short)apduBuffer.length);
325         }
```

```
326     }
327     if (this.receivingBuffer[3] == this.MessageHandlerTagOne[3]) {
328       try {
329         parseMessage(receivingBuffer);
330       } catch (Exception cE) {
331         ISOException.throwIt((short)0xA112);
332       }
333       receivingBuffer = JCSystem.makeTransientByteArray((short)568,
334         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
335       generateResponse((short)1);
336     } else
337     if (this.receivingBuffer[3] == this.MessageHandlerTagTwo[3]) {
338       processSecondMsg(receivingBuffer);
339       receivingBuffer = JCSystem.makeTransientByteArray((short)568,
340         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
341       generateResponse((short)2);
342     } else if (this.receivingBuffer[3] == this.MessageHandlerTagThree[3])
          {
343       processSPsThirdMsg(receivingBuffer);
344       JCSystem.requestObjectDeletion();
345       return ;
346     } else if (this.receivingBuffer[3] == (byte)0xF1) {
347       if (processTSMActAppMessage()) {
348         JCSystem.requestObjectDeletion();
349         return ;
350       } else {
351         JCSystem.requestObjectDeletion();
352         apdu.setOutgoing();
353         apdu.setOutgoingLength((short)receivingBuffer.length);
354         apdu.sendBytesLong(receivingBuffer, (short)0, (short)
355                            receivingBuffer.length);
356       }
357     } else {
358       ISOException.throwIt(ProtocolHandler.SW_ERROR_INS);
359     }
360     JCSystem.requestObjectDeletion();
361     apdu.setOutgoing();
362     apdu.setOutgoingLength((short)copyPointer);
363     apdu.sendBytesLong(receivingBuffer, (short)0, (short)copyPointer);
364     JCSystem.requestObjectDeletion();
365   }
366   private void SCTSMChargeRequest(APDU apdu) {
367     short childPointerMessage = 6;
368     short encryptionOffset = 0;
369     short encryptedDataChild = 0;
370     short encryptionLength = 0;
371     copyPointer = 0;
372     try {
373       copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagSCTSM,
374         (short)0, this.receivingBuffer, copyPointer, (short)
375         this.MessageHandlerTagSCTSM.length)copyPointer =
```

```
376                             Util.arrayCopyNonAtomic(this.CardID, (short)0,
377                             this.receivingBuffer, copyPointer,
                                    (short)this.CardID.length);
378         this.receivingBuffer[childPointerMessage]++;
379         copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag, (short)
380           0, this.receivingBuffer, copyPointer, (short)
381           this.EncryptedDataTag.length);
382         encryptedDataChild = (short)(copyPointer + (short)2);
383         copyPointer += (short)3;
384         encryptionOffset = copyPointer;
385         this.receivingBuffer[childPointerMessage]++;
386         copyPointer = Util.arrayCopyNonAtomic(this.SCIdentityTag, (short)0,
387           this.receivingBuffer, copyPointer, (short)
388           this.SCIdentityTag.length);
389         this.receivingBuffer[encryptedDataChild]++;
390         copyPointer = Util.arrayCopyNonAtomic(this.UserIdentity, (short)0,
391           this.receivingBuffer, copyPointer,
                (short)this.UserIdentity.length)
392           ;
393         this.receivingBuffer[encryptedDataChild]++;
394         copyPointer = Util.arrayCopyNonAtomic(this.TSMIdentity, (short)0,
395           this.receivingBuffer, copyPointer,
                (short)this.TSMIdentity.length);
396         this.receivingBuffer[encryptedDataChild]++;
397         randomDataGen.generateData(this.SCRandomNumberArray,
398                                    this.PTLVDataOffset, (short)16);
399         copyPointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
400           (short)0, this.receivingBuffer, copyPointer, (short)
401           this.SCRandomNumberArray.length);
402         this.receivingBuffer[encryptedDataChild]++;
403         encryptionLength = (short)(copyPointer - encryptionOffset);
404         this.phCipherKey.setKey(this.myLongTermEncryptionKey, (short)0);
405         messageEncryption(this.receivingBuffer, encryptionOffset, (short)
406                             (copyPointer - encryptionOffset));
407         this.shortToBytes(receivingBuffer, (short)(encryptionOffset -
              (short)
408                             3), (short)(copyPointer - encryptionOffset));
409         this.phMacGeneratorKey.setKey(this.myLongTermMacKey, (short)0);
410         macGenerate(this.receivingBuffer, encryptionOffset, (short)
411                       (copyPointer - encryptionOffset), Signature.MODE_SIGN);
412         this.receivingBuffer[childPointerMessage]++;
413         copyPointer = Util.arrayCopyNonAtomic(this.SPCookieArray, (short)0,
414           this.receivingBuffer, copyPointer, (short)
415           this.SPCookieArray.length);
416         this.receivingBuffer[childPointerMessage]++;
417         this.shortToBytes(this.receivingBuffer, (short)4, copyPointer);
418       } catch (Exception cE) {
419         ISOException.throwIt((short)encryptionLength);
420       }
421     }
422     private void AppDownloadCompleted(APDU apdu) {
```

```
423     receivingBuffer = JCSystem.makeTransientByteArray((short)568,
424       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
425     generateResponse((short)2);
426     JCSystem.requestObjectDeletion();
427     apdu.setOutgoing();
428     apdu.setOutgoingLength((short)copyPointer);
429     apdu.sendBytesLong(receivingBuffer, (short)0, (short)copyPointer);
430     JCSystem.requestObjectDeletion();
431   }
432   private void generateResponse(short msgNumber) {
433     short childPointerMessage = 6;
434     short encryptionOffset = 0;
435     copyPointer = 0;
436     if (msgNumber == 1) {
437       randomDataGen.generateData(this.SCRandomNumberArray,
438                                  this.PTLVDataOffset, (short)16);
439       this.dhKeyConGen(this.SPDHChanllengerArray, this.PTLVDataOffset,
440                        ProtocolHandler.GEN_DHKEY);
441       copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagOne,
442         (short)0, this.receivingBuffer, copyPointer, (short)
443         this.MessageHandlerTagOne.length);
444       copyPointer = Util.arrayCopyNonAtomic(this.SCDHData, (short)0,
445         this.receivingBuffer, copyPointer, (short)this.SCDHData.length);
446       this.receivingBuffer[childPointerMessage]++;
447       copyPointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
448         (short)0, this.receivingBuffer, copyPointer, (short)
449         this.SCRandomNumberArray.length);
450       this.receivingBuffer[childPointerMessage]++;
451       keygenerator();
452       copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag, (short)
453         0, this.receivingBuffer, copyPointer, (short)
454         this.EncryptedDataTag.length);
455       this.receivingBuffer[childPointerMessage]++;
456       short childEnMessage = (short)(copyPointer + (short)2);
457       copyPointer += (short)3;
458       encryptionOffset = copyPointer;
459       copyPointer = Util.arrayCopyNonAtomic(this.SCIdentityTag, (short)0,
460         this.receivingBuffer, copyPointer, (short)
461         this.SCIdentityTag.length);
462       this.receivingBuffer[childEnMessage]++;
463       copyPointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
464         (short)0, this.receivingBuffer, copyPointer, (short)
465         this.SCRandomNumberArray.length);
466       this.receivingBuffer[childEnMessage]++;
467       copyPointer = Util.arrayCopyNonAtomic(this.SPRandomNumberArray,
468         (short)0, this.receivingBuffer, copyPointer, (short)
469         this.SPRandomNumberArray.length);
470       this.receivingBuffer[childEnMessage]++;
471       this.signGenerate(this.receivingBuffer, encryptionOffset, (short)
472                         (copyPointer - encryptionOffset), phUserKeyPair
473                         .getPrivate(), Signature.MODE_SIGN);
```

```
474         this.receivingBuffer[childEnMessage]++;
475         copyPointer = Util.arrayCopyNonAtomic(this.SCUserCertificate,
                (short)
476           0, this.receivingBuffer, copyPointer, (short)
477           this.SCUserCertificate.length);
478         this.receivingBuffer[childEnMessage]++;
479         messageEncryption(this.receivingBuffer, encryptionOffset, (short)
480                           (copyPointer - encryptionOffset));
481         this.shortToBytes(receivingBuffer, (short)(encryptionOffset -
                (short)
482                           3), (short)(copyPointer - encryptionOffset));
483         macGenerate(this.receivingBuffer, encryptionOffset, (short)
484                     (copyPointer - encryptionOffset), Signature.MODE_SIGN);
485         this.receivingBuffer[childPointerMessage]++;
486         copyPointer = Util.arrayCopyNonAtomic(this.SPCookieArray, (short)0,
487           this.receivingBuffer, copyPointer, (short)
488           this.SPCookieArray.length);
489         this.receivingBuffer[childPointerMessage]++;
490         this.shortToBytes(this.receivingBuffer, (short)4, copyPointer);
491       } else if (msgNumber == 2) {
492         copyPointer = (short)0;
493         short tempLength = (short)0;
494         short mainChildPointer = (short)6;
495         short mainLengthPointer = (short)4;
496         short encryptedChildPointer = (short)13;
497         short generalLengthPointer = (short)0;
498         this.receivingBuffer[mainChildPointer] = (short)0;
499         this.receivingBuffer[encryptedChildPointer] = (short)0;
500         copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagTwo,
501           (short)0, this.receivingBuffer, copyPointer, (short)7);
502         this.receivingBuffer[mainChildPointer]++;
503         copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag, (short)
504           0, this.receivingBuffer, copyPointer, (short)4);
505         copyPointer += (short)3;
506         encryptionOffset = copyPointer;
507         copyPointer = Util.arrayCopyNonAtomic(this.PlatformHash, (short)0,
508           receivingBuffer, copyPointer, (short)4);
509         generalLengthPointer = copyPointer;
510         copyPointer += (short)2;
511         MessageDigest myHashGen = MessageDigest.getInstance
512           (MessageDigest.ALG_SHA_256, false);
513         tempLength = (short)myHashGen .doFinal(this.ClassDH.dhModulus,
                (short)0,
514                           (short)this.ClassDH.dhModulus.length, receivingBuffer,
515                           copyPointer);
516         this.receivingBuffer[encryptedChildPointer]++;
517         this.shortToBytes(this.receivingBuffer, generalLengthPointer,
                (short)
518                           (tempLength));
519         copyPointer += tempLength;
520         copyPointer = Util.arrayCopyNonAtomic(this.UserIdentity, (short)0,
```

```
521           this.receivingBuffer, copyPointer,
                  (short)this.UserIdentity.length)
522           ;
523         this.receivingBuffer[encryptedChildPointer]++;
524         copyPointer = Util.arrayCopyNonAtomic(this.SPIdentity, (short)0,
525           this.receivingBuffer, copyPointer, (short)this.SPIdentity.length);
526         this.receivingBuffer[encryptedChildPointer]++;
527         copyPointer = Util.arrayCopyNonAtomic(this.SCRandomNumberArray,
528           (short)0, this.receivingBuffer, copyPointer, (short)
529           this.SCRandomNumberArray.length);
530         this.receivingBuffer[encryptedChildPointer]++;
531         copyPointer = Util.arrayCopyNonAtomic(this.SPRandomNumberArray,
532                                       (short)0, this.receivingBuffer,
                                              (short)copyPointer, (short)
533                                       this.SPRandomNumberArray.length);
534         this.receivingBuffer[encryptedChildPointer]++;
535         try {
536           this.signGenerate(receivingBuffer, (short)(encryptionOffset),
537                             (short)(copyPointer - encryptionOffset),
538                             phSCKeyPair.getPrivate(), Signature.MODE_SIGN);
539           this.receivingBuffer[encryptedChildPointer]++;
540         } catch (Exception cE) {
541           ISOException.throwIt((short)0xFA17);
542         }
543         copyPointer = Util.arrayCopyNonAtomic(this.SCCertificate, (short)0,
544           this.receivingBuffer, copyPointer, (short)
545           this.SCCertificate.length);
546         this.receivingBuffer[encryptedChildPointer]++;
547         try {
548           this.messageEncryption(receivingBuffer, (short)
549                                  (encryptedChildPointer + (short)1),
                                       (short)
550                                  (copyPointer - (encryptedChildPointer +
551                                  (short)1)));
552         } catch (Exception cE) {
553           ISOException.throwIt((short)(copyPointer - encryptedChildPointer +
554                                (short)1));
555         }
556         this.shortToBytes(this.receivingBuffer, (short)
557                           (encryptedChildPointer - (short)2), (short)
558                           (copyPointer - (short)(encryptedChildPointer +
559                           (short)1)));
560         this.macGenerate(receivingBuffer, (short)(encryptedChildPointer +
561                          (short)1), (short)(copyPointer -
562                          (encryptedChildPointer + (short)1)),
563                          Signature.MODE_SIGN);
564         this.receivingBuffer[mainChildPointer]++;
565         copyPointer = Util.arrayCopyNonAtomic(this.SPCookieArray, (short)0,
566           this.receivingBuffer, copyPointer, (short)
567           this.SPCookieArray.length);
568         this.receivingBuffer[mainChildPointer]++;
```

```
569       this.shortToBytes(this.receivingBuffer, mainLengthPointer, (short)
570                         (copyPointer - (short)7));
571     }
572   }
573   void platformHashGeneration(byte[] inArray, short inOffset){}
574   void processSecondMsg(byte[] inArray) {
575     short inOffset = (short)14;
576     short inLength = (short)(ProtocolHandler.bytesToShort(inArray, (short)
577                         11));
578     if (this.macGenerate(inArray, inOffset, inLength,
579         Signature.MODE_VERIFY)) {
580       this.phDecryption(inArray, inOffset, inLength);
581       Util.arrayCopyNonAtomic(inArray, inOffset, this.SPIdentity,
            (short)0,
582                               (short)this.SPIdentity.length);
583       Util.arrayCopyNonAtomic(inArray, (short)(inOffset + (short)
584                               this.SPIdentity.length), this.AppIdentity,
585                               (short)0, (short)this.AppIdentity.length);
586     } else {
587       ISOException.throwIt((short)0xFA18);
588     }
589   }
590   boolean processTSMActAppMessage() {
591     short inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
592     short inLength = (short)(ProtocolHandler.bytesToShort(receivingBuffer,
593                         (short)(inOffset - (short)3)));
594     if (this.macGenerate(receivingBuffer, inOffset, inLength,
595         Signature.MODE_VERIFY)) {
596       this.phDecryption(receivingBuffer, inOffset, inLength);
597       inOffset += (short)225;
598       inLength = (short)3;
599       TSMVerificationKey.setExponent(receivingBuffer, inOffset, inLength);
600       inOffset += (short)(inLength + this.PTLVDataOffset);
601       inLength = (short)64;
602       TSMVerificationKey.setModulus(receivingBuffer, inOffset, inLength);
603       inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
604       inLength = (short)142;
605       if (this.signGenerate(receivingBuffer, (short)inOffset, (short)
606           inLength, TSMVerificationKey, Signature.MODE_VERIFY)) {
607         Util.arrayCopyNonAtomic(receivingBuffer, (short)224,
608                                 this.SPCookieArray, (short)0, (short)
609                                 this.SPCookieArray.length);
610         this.phMacGeneratorKey.setKey(this.myLongTermMacKey, (short)0);
611         phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
612                             InitialisationVector, (short)0, (short)
613                             InitialisationVector.length);
614         phMacGenerator.sign(this.receivingBuffer, (short)14, (short)96,
615                             this.SID, (short)0);
616         return true;
617       } else {
618         return false;
```

```
619       }
620     } else {
621       return false;
622     }
623   }
624   void processSPsThirdMsg(byte[] inArray) {
625     short inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
626     short inLength = (short)(ProtocolHandler.bytesToShort(inArray, (short)
627                       (inOffset - (short)3)));
628     if (this .macGenerate(inArray, inOffset, inLength,
629         Signature.MODE_VERIFY)) {
630       this.phDecryption(inArray, inOffset, inLength);
631       Util.arrayCopyNonAtomic(inArray, inOffset, this.SPIdentity,
            (short)0,
632                               (short)this.SPIdentity.length);
633       inOffset += (short)151;
634       inLength = (short)3;
635       SPVerificationKey.setExponent(inArray, inOffset, inLength);
636       inOffset += (short)(inLength + this.PTLVDataOffset);
637       inLength = (short)64;
638       SPVerificationKey.setModulus(inArray, inOffset, inLength);
639       inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
640       inLength = (short)68;
641       if (this.signGenerate(inArray, inOffset, inLength,
            SPVerificationKey,
642           Signature.MODE_VERIFY)) {
643         return ;
644       } else {
645         ISOException.throwIt((short)0x6666);
646       }
647     } else {
648       ISOException.throwIt((short)0xFA18);
649     }
650   }
651   void parseMessage(byte[] inBuffer) {
652     byte childLeft = inBuffer[(short)(this.CTLVDataOffset - (short)1)];
653     short pointer = (short)this.CTLVDataOffset;
654     while (childLeft > 0) {
655       if (Util.arrayCompare(SPDHChallengeTag, (short)0, inBuffer, pointer,
656           (short)4) == 0) {
657         Util.arrayCopy(inBuffer, pointer, this.SPDHChanllengerArray,
658                        (short)0, (short)this.SPDHChanllengerArray.length);
659         pointer += (short)this.SPDHChanllengerArray.length;
660       } else if (Util.arrayCompare(this.SPRandomNumberTag, (short)0,
661                  inBuffer, pointer, (short)4) == 0) {
662         Util.arrayCopyNonAtomic(inBuffer, pointer,
663                                 this.SPRandomNumberArray, (short)0,
                                    (short)
664                                 (this.SPRandomNumberArray.length));
665         pointer += (short)(this.SPRandomNumberArray.length);
666       } else if (Util.arrayCompare(this.SPCookieTag, (short)0, inBuffer,
```

```
667                     pointer, (short)4) == 0) {
668           Util.arrayCopyNonAtomic(inBuffer, pointer, this.SPCookieArray,
669                                   (short)0, (short)
670                                   (this.SPCookieArray.length));
671           pointer += (short)(this.SPCookieArray.length);
672         }
673         childLeft -= (short)1;
674       }
675     }
676     void dhInitialisation() {
677       dhKey.setModulus(ClassDH.dhModulus, (short)0,
            (short)ClassDH.dhModulus.length);
678     }
679     void dhKeyConGen(byte[] inbuff, short inbuffOffset, byte Oper_Mode) {
680       switch (Oper_Mode) {
681         case GEN_KEYCONTRIBUTION:
682           randomExponent = JCSystem.makeTransientByteArray((short)32,
683             JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
684           randomDataGen.generateData(randomExponent, (short)0, (short)
685                                      randomExponent.length);
686           dhKey.setExponent(randomExponent, (short)0, (short)
687                             randomExponent.length);
688           pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
689           pkCipher.doFinal(ClassDH.dhBase, (short)0,
                (short)ClassDH.dhBase.length, inbuff,
690                            inbuffOffset);
691           break;
692         case GEN_DHKEY:
693           dhKey.setExponent(randomExponent, (short)0, (short)
694                             randomExponent.length);
695           pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
696           SCSPDHGeneratedValue = JCSystem.makeTransientByteArray((short)
697             ClassDH.dhModulus.length,
                  JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
698           pkCipher .doFinal(inbuff, inbuffOffset, (short)((short)
699                             inbuff.length - (short)this.PTLVDataOffset),
700                             SCSPDHGeneratedValue, (short)0);
701           break;
702         default:
703           ISOException.throwIt((short)0x5FA1);
704       }
705     }
706     void keygenerator() {
707       AESKey sessionGenKey = (AESKey)KeyBuilder.buildKey
708                               (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
709                               KeyBuilder.LENGTH_AES_128, false);
710       sessionGenKey.setKey(SCSPDHGeneratedValue, (short)0);
711       phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
712                           InitialisationVector, (short)0, (short)
713                           InitialisationVector.length);
714       byte[] keyGenMacData = JCSystem.makeTransientByteArray((short)64,
```

```
715       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
716     short pointer = 0;
717     pointer = Util .arrayCopyNonAtomic(this.SPRandomNumberArray,
718       this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
719     pointer = Util .arrayCopyNonAtomic(this.SCRandomNumberArray,
720       this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
721     pointer = Util.arrayCopyNonAtomic(SCSPDHGeneratedValue, (short)16,
722       keyGenMacData, (short)pointer, (short)16);
723     for (short i = 48; i < 64; i++) {
724       keyGenMacData[i] = (byte)0x02;
725     }
726     phMacGenerator.sign(keyGenMacData, (short)0, (short)
727                         keyGenMacData.length, SCSPDHGeneratedValue,
                                (short)
728                         0);
729     this.phCipherKey.setKey(SCSPDHGeneratedValue, (short)0);
730     for (short i = 48; i < 64; i++) {
731       keyGenMacData[i] = (byte)0x03;
732     }
733     phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
734                         InitialisationVector, (short)0, (short)
735                         InitialisationVector.length);
736     phMacGenerator.sign(keyGenMacData, (short)0, (short)
737                         keyGenMacData.length, SCSPDHGeneratedValue,
                                (short)
738                         0);
739     this.phMacGeneratorKey.setKey(SCSPDHGeneratedValue, (short)0);
740     SCSPDHGeneratedValue = null;
741     JCSystem.requestObjectDeletion();
742   }
743   void messageEncryption(byte[] inbuff, short inbuffOffset, short
744                          inbuffLength) {
745     syCipher.init(phCipherKey, Cipher.MODE_ENCRYPT, InitialisationVector,
746                   (short)0, (short)InitialisationVector.length);
747     this.shortToBytes(inbuff, (short)(inbuffOffset - 3), (short)syCipher
748                       .doFinal(inbuff, inbuffOffset, inbuffLength, inbuff,
749                       inbuffOffset));
750   }
751   void phDecryption(byte[] inbuff, short inbuffOffset, short inbuffLength)
752                     {
753     syCipher.init(phCipherKey, Cipher.MODE_DECRYPT, InitialisationVector,
754                   (short)0, (short)InitialisationVector.length);
755     syCipher.doFinal(inbuff, inbuffOffset, inbuffLength, inbuff,
756                      inbuffOffset);
757   }
758   boolean macGenerate(byte[] inbuff, short inbuffOffset, short
759                       inbuffLength, short macMode) {
760     if (macMode == Signature.MODE_SIGN) {
761       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
762                           InitialisationVector, (short)0, (short)
763                           InitialisationVector.length);
```

```
764       copyPointer = Util.arrayCopyNonAtomic(this.MACedDataTag, (short)0,
765         this.receivingBuffer, copyPointer,
              (short)this.MACedDataTag.length)
766         ;
767       copyPointer += 2;
768       short length = (short)phMacGenerator.sign(this.receivingBuffer,
769                       inbuffOffset, inbuffLength, inbuff, copyPointer);
770       this.shortToBytes(inbuff, (short)(copyPointer - (short)2), length);
771       copyPointer += length;
772       return true;
773     } else if (macMode == Signature.MODE_VERIFY) {
774       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_VERIFY,
775                           InitialisationVector, (short)0, (short)
776                           InitialisationVector.length);
777       return phMacGenerator.verify(this.receivingBuffer, inbuffOffset,
778                                    inbuffLength, inbuff, (short)
779                                    (inbuffOffset + inbuffLength +
780                                    this.PTLVDataOffset), (short)16);
781     }
782     return false;
783   }
784   boolean signGenerate(byte[] inbuff, short inbuffOffset, short
785                        inbufflength, Key kpSign, short signMode) {
786     if (signMode == Signature.MODE_SIGN) {
787       copyPointer = Util.arrayCopyNonAtomic(this.SignedDataTag, (short)0,
788         this.receivingBuffer, copyPointer, (short)
789         this.SignedDataTag.length);
790       copyPointer += (short)2;
791       phSign.init((RSAPrivateKey)kpSign, Signature.MODE_SIGN);
792       signlength = phSign.sign(inbuff, (short)inbuffOffset, inbufflength,
793                                inbuff, copyPointer);
794       this.shortToBytes(this.receivingBuffer, (short)(copyPointer -
            (short)
795                         2), signlength);
796       copyPointer += signlength;
797       return true;
798     } else if (signMode == Signature.MODE_VERIFY) {
799       phSign.init((RSAPublicKey)kpSign, Signature.MODE_VERIFY);
800       return phSign.verify(inbuff, inbuffOffset, inbufflength, inbuff,
801                            (short)(inbuffOffset + inbufflength +
802                            this.PTLVDataOffset), (short)64);
803     }
804     return false;
805   }
806   public static short bytesToShort(byte[] ArrayBytes) {
807     return (short)(((ArrayBytes[0] << 8)) | ((ArrayBytes[1] & 0xff)));
808   }
809   public static short bytesToShort(byte[] ArrayBytes, short arrayOffset) {
810     return (short)(((ArrayBytes[arrayOffset] << 8)) | ((ArrayBytes[(short)
811             (arrayOffset + (short)1)] & 0xff)));
812   }
```

```
813   private void shortToBytes(byte[] Array, short arrayOffset, short
          inShort)
814                                   {
815     Array[arrayOffset] = (byte)((short)(inShort & (short)0xFF00) >>
            (short)
816                                     0x0008);
817     Array[(short)(arrayOffset + (short)1)] = (byte)(inShort & (short)
818             0x00FF);
819   }
820 }
```

C.6.2 Service Provider Implementation

In this section, we detail the SP's implementation of the STCP_{ACA} and the helper functions utlised during the STCP_{SP} are discussed in appendices C.11.1 and C.11.2.

```
1  package ACAPTerminal;
2
3  import java.util.Arrays;
4  import java.security.interfaces.RSAPublicKey;
5  import javax.crypto.spec.SecretKeySpec;
6  import java.security.spec.RSAPublicKeySpec;
7  import javax.crypto.*;
8  import java.security.*;
9  import java.math.BigInteger;
10 public class ServiceProviderProtocolHandler {
11   private byte[] SPIdentity = {
12     (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)0x7C, (byte)0x62,
13     (byte)0x0A, (byte)0x86, (byte)0x52, (byte)0xBE, (byte)0x5E,
14     (byte)0x90, (byte)0x01, (byte)0xA8, (byte)0xD6, (byte)0x6A,
15     (byte)0xD7, (byte)0xB1, (byte)0x7C};
16   private byte[] AppIdentity = {
17     (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)0x7C, (byte)0x62,
           (byte)0x0A,
18      (byte)0x86, (byte)0x52, (byte)0xBE, (byte)0x5E, (byte)0x90, (byte)
19      0x01, (byte)0xA8, (byte)0xD6, (byte)0x6A, (byte)0xD7, (byte)0xB1,
20      (byte)0x7C, (byte)0xA8, (byte)0xD6, (byte)0x6A, (byte)0xD7};
21   private byte[] SCIP = {
22     (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)0x7C};
23   private byte[] PlatformHashPreset = {
24     (byte)0xBF, (byte)0xE5, (byte)0x45, (byte)0x86, (byte)0x2C,
           (byte)0xA1,
25      (byte)0x02, (byte)0xAD, (byte)0x1E, (byte)0xED, (byte)0xDB, (byte)
26      0x5F, (byte)0xBF, (byte)0xA5, (byte)0xBF, (byte)0x85, (byte)0x5A,
27      (byte)0xC4, (byte)0x99, (byte)0x5C, (byte)0x56, (byte)0xA8, (byte)
28      0xB4, (byte)0x08, (byte)0xCE, (byte)0x3F, (byte)0xE0, (byte)0x99,
29      (byte)0xDC, (byte)0xE9, (byte)0x3A, (byte)0x9D};
30   private byte[] MessageHandlerTagOne = {(byte)0xAA, (byte)0xAA};
31   private byte[] MessageHandlerTagTwo = {(byte)0xBB, (byte)0xBB};
32   private byte[] MessageHandlerTagThree = {(byte)0xCC, (byte)0xCC};
33   private byte[] SPIdentityTag = {(byte)0x5F, (byte)0x01};
```

```
34  private byte[] AppIdentityTag = {(byte)0x5F, (byte)0x0E};
35  private byte[] SPDHChallengeTag = {(byte)0x5C, (byte)0x01};
36  private byte[] SPSignatureCertTag = {(byte)0xF0, (byte)0xF01};
37  private byte[] SPSigVerificationKeyTag = {(byte)0x51, (byte)0x01};
38  private byte[] SPRandomNumberTag = {(byte)0x5A, (byte)0x01};
39  private byte[] SPCookieTag = {(byte)0x5B, (byte)0x01};
40  private byte[] EncryptedDataTag = {(byte)0xFE, (byte)0x01};
41  private byte[] MACedDataTag = {(byte)0x5D, (byte)0x01};
42  private byte[] SignedDataTag = {(byte)0x5D, (byte)0x02};
43  private byte[] PublicExponentTag = {(byte)0xEE, (byte)0x01};
44  private byte[] PublicModulusTag = {(byte)0xEE, (byte)0x02};
45  private byte[] SCDHChallengeTag = {(byte)0x5C, (byte)0x02};
46  private byte[] SCRandomNumberTag = {(byte)0x5A, (byte)0x02};
47  private byte[] SCIdentityTag = {(byte)0x5F, (byte)0x02};
48  private byte[] SCUserCertificateTag = {(byte)0xF0, (byte)0x03};
49  private byte[] SCCertificateTag = {(byte)0xF0, (byte)0x02};
50  private byte[] PlatformHashTag = {(byte)0x5E, (byte)0xAF};
51  private byte[] UserIdentityTag = {(byte)0x5F, (byte)0x03};
52  public ConstructedTLV MessageHandler = ConstructedTLV.getConstructedTLV
53    (MessageHandlerTagOne);
54  private ConstructedTLV SPSignatureCertificate =
55    ConstructedTLV.getConstructedTLV(SPSignatureCertTag);
56  private PrimitiveTLV SPIdentityTLV = PrimitiveTLV.getPrimitiveTLV
57    (SPIdentityTag, SPIdentity);
58  private PrimitiveTLV AppIdentityTLV = PrimitiveTLV.getPrimitiveTLV
59    (AppIdentityTag, AppIdentity);
60  private PrimitiveTLV SPSigVerificationKey = PrimitiveTLV.getPrimitiveTLV
61    (this.SPSigVerificationKeyTag);
62  private PrimitiveTLV SPDHChanllenger = PrimitiveTLV.getPrimitiveTLV
63    (this.SPDHChallengeTag);
64  private PrimitiveTLV SPRandomNumber = PrimitiveTLV.getPrimitiveTLV
65    (this.SPRandomNumberTag);
66  private PrimitiveTLV SPCookie = PrimitiveTLV.getPrimitiveTLV
67    (this.SPCookieTag);
68  private ConstructedTLV EncryptedData = ConstructedTLV.getConstructedTLV
69    (this.EncryptedDataTag);
70  private PrimitiveTLV MACedData = PrimitiveTLV.getPrimitiveTLV
71    (this.MACedDataTag);
72  private PrimitiveTLV SignedData = PrimitiveTLV.getPrimitiveTLV
73    (this.SignedDataTag);
74  private PrimitiveTLV PublicExponent = PrimitiveTLV.getPrimitiveTLV
75    (this.PublicExponentTag);
76  private PrimitiveTLV PublicModulus = PrimitiveTLV.getPrimitiveTLV
77    (this.PublicModulusTag);
78  private PrimitiveTLV SCDHChallenge = PrimitiveTLV.getPrimitiveTLV
79    (this.SCDHChallengeTag);
80  private PrimitiveTLV SCRandomNumber = PrimitiveTLV.getPrimitiveTLV
81    (this.SCRandomNumberTag);
82  private PrimitiveTLV SCIdentity = PrimitiveTLV.getPrimitiveTLV
83    (SCIdentityTag);
84  private ConstructedTLV SCUserCertificate =
```

```
85      ConstructedTLV.getConstructedTLV(this.SCUserCertificateTag);
86    private ConstructedTLV SCCertificate = ConstructedTLV.getConstructedTLV
87      (this.SCCertificateTag);
88    private PrimitiveTLV PlatformHash = PrimitiveTLV.getPrimitiveTLV
89      (this.PlatformHashTag);
90    private PrimitiveTLV UserIdentity = PrimitiveTLV.getPrimitiveTLV
91      (this.UserIdentityTag);
92    private ProtocolHelperClass myProtocolHelperObject = new
93      ProtocolHelperClass();
94    private byte[] mySessionEncryptionKey = new byte[16];
95    private byte[] mySessionMacKey = new byte[16];
96    private PublicKey SCUserVerificationKey = null;
97    private PublicKey SCVerificationKey = null;
98    public ServiceProviderProtocolHandler() {
99      myProtocolHelperObject.protocolInitialise();
100     RSAPublicKey tempKey = (RSAPublicKey)
101                               myProtocolHelperObject.getPublicKey();
102     byte[] tempExponent = tempKey.getPublicExponent().toByteArray();
103     this.PublicExponent.initialisationPTLV(this.PublicExponentTag,
104       tempExponent.length);
105     this.PublicExponent.setTlvValues(tempExponent);
106     byte[] tempModulus = tempKey.getModulus().toByteArray();
107     this.PublicModulus.initialisationPTLV(this.PublicModulusTag,
108       (tempModulus.length - 1));
109     this.PublicModulus.setTlvValues(tempModulus, 1, (tempModulus.length -
110                                       1));
111     SPSignatureCertificate.addPTLV(this.PublicExponent);
112     SPSignatureCertificate.addPTLV(this.PublicModulus);
113   }
114   public void initialiseProtocol() {
115     try {
116       this.SPDHChanllenger.setTlvValues
117         (this.myProtocolHelperObject.GenerateDHPublicValue());
118       this.MessageHandler.addPTLV(this.SPDHChanllenger);
119     } catch (Exception cE) {
120       System.out.println(
121                           "Error ProtocolHandler.initialiseProtocol Option
                                = 1, : " + cE.getClass().getName());
122     }
123   }
124   public byte[] outMessageProcessing(int Counter) {
125     if (Counter == 1) {
126       try {
127         this.SPRandomNumber.setTlvValues
128           (this.myProtocolHelperObject.getRandomNumber());
129         this.MessageHandler.addPTLV(this.SPRandomNumber);
130         byte[] temp = new byte[(this.SCIP.length +
131                                 this.SPDHChanllenger.getValueLength() +
132                                 this.SPRandomNumber.getValueLength())];
133         System.arraycopy(this.SPDHChanllenger.getValueBytes(), 0, temp, 0,
134                           this.SPDHChanllenger.getValueLength());
```

```
135           System.arraycopy(this.SPRandomNumber.getValueBytes(), 0, temp,
136                            this.SPDHChanllenger.getValueLength(),
137                            this.SPRandomNumber.getValueLength());
138           System.arraycopy(this.SCIP, 0, temp, temp.length -
139                            this.SCIP.length, this.SCIP.length);
140           byte[] result = new byte[16];
141           this.myProtocolHelperObject.GenerateMac(temp, 0, temp.length,
142             result, 0, this.myProtocolHelperObject.myLongTermMacKey);
143           this.SPCookie.setTlvValues(result);
144           this.MessageHandler.addPTLV(this.SPCookie);
145         } catch (Exception cE) {
146           System.out.println(
147                            "Error ProtocolHandler.inMessageProcessing
                                 Option = 2, : " + cE.getClass().getName());
148         }
149       } else if (Counter == 2) {
150         try {
151           this.EncryptedData.initialisationCTLV(this.EncryptedDataTag);
152           this.EncryptedData.addPTLV(this.SPIdentityTLV);
153           this.EncryptedData.addPTLV(this.AppIdentityTLV);
154           this.EncryptedData.addPTLV(this.SPRandomNumber);
155           this.EncryptedData.addPTLV(this.SCRandomNumber);
156           this.myProtocolHelperObject.GenerateEncryption
157             (this.EncryptedData.getValueBytes(), 0,
158             this.EncryptedData.getValueBytes().length,
159             this.EncryptedData.getBytesTlvRepresentation(), 7,
160             this.mySessionEncryptionKey);
161           this.MACedData.initialisationPTLV(this.MACedDataTag, 16);
162           this.myProtocolHelperObject.GenerateMac
163             (this.EncryptedData.getValueBytes(), 0,
164             this.EncryptedData.getTagValueLength(),
165             this.MACedData.getBytesTlvRepresentation(), 6,
166             this.mySessionMacKey);
167           this.MessageHandler.initialisationCTLV(this.MessageHandlerTagTwo);
168           this.MessageHandler.addCTLV(EncryptedData);
169           this.MessageHandler.addPTLV(this.MACedData);
170         } catch (Exception cE) {
171           System.out.println(
172                            "Error ProtocolHandler.inMessageProcessing
                                 Option = 3, : " + cE.getClass().getName());
173         }
174       } else if (Counter == 3) {
175         try {
176           this.EncryptedData.initialisationCTLV(this.EncryptedDataTag);
177           this.EncryptedData.addPTLV(this.SPIdentityTLV);
178           this.EncryptedData.addPTLV(this.SPRandomNumber);
179           this.EncryptedData.addPTLV(this.SCRandomNumber);
180           this.myProtocolHelperObject.SignatureMethod
181             (this.EncryptedData.getValueBytes(), 0,
182             this.EncryptedData.getValueBytes().length,
183             this.SignedData.getBytesTlvRepresentation(), 6, null,
```

```
184             ProtocolHelperClass.SIGN_MODE_GENERATION);
185           this.EncryptedData.addPTLV(this.SignedData);
186           this.EncryptedData.addCTLV(this.SPSignatureCertificate);
187           this.myProtocolHelperObject.GenerateEncryption
188             (this.EncryptedData.getValueBytes(), 0,
189             this.EncryptedData.getValueBytes().length,
190             this.EncryptedData.getBytesTlvRepresentation(), 7,
191             this.mySessionEncryptionKey);
192           this.MACedData.initialisationPTLV(this.MACedDataTag, 16);
193           this.myProtocolHelperObject.GenerateMac
194             (this.EncryptedData.getValueBytes(), 0,
195             this.EncryptedData.getTagValueLength(),
196             this.MACedData.getBytesTlvRepresentation(), 6,
197             this.mySessionMacKey);
198           this.MessageHandler.initialisationCTLV(this.MessageHandlerTagThree)
199             ;
200           this.MessageHandler.addCTLV(EncryptedData);
201           this.MessageHandler.addPTLV(this.MACedData);
202           this.MessageHandler.addPTLV(this.SPCookie);
203         } catch (Exception cE) {
204           System.out.println(
205                               "Error ProtocolHandler.inMessageProcessing
                                        Option = 1, : " + cE.getClass().getName());
206         }
207       } else {
208         System.out.println(
209                             "Protocol Stoped : Illegal Message Value
                                    (ProtocolHanlder.inMessageProcessing()");
210       }
211       return this.MessageHandler.getBytesTlvRepresentation();
212     }
213     public boolean inMessageProcessing(byte[] inMessage, int Counter) {
214       try {
215         if (Counter == 1) {
216           MessageHandler.setBytesTlvRepresentation(inMessage, 0,
217             (inMessage.length - 2));
218           childExtractionFromCTLV(MessageHandler);
219           GenerateKeys(this.SCDHChallenge.getValueBytes());
220           byte[] temp = new byte[16];
221           this.myProtocolHelperObject.GenerateMac
222             (this.EncryptedData.getValueBytes(), 0,
223             this.EncryptedData.getValueBytes().length, temp, 0,
224             this.mySessionMacKey);
225           if (Arrays.equals(this.MACedData.getValueBytes(), temp)){}
226            else {
227             System.out.println(
228                                 "Integrity Check Failure : ERROR at
                                        ProtocolHandler.inMessageProcessing \n");
229             System.exit(0);
230           }
231           this.myProtocolHelperObject.GenerateDecryption
```

```
232           (this.EncryptedData.getValueBytes(), 0,
233           this.EncryptedData.getValueBytes().length,
234           this.EncryptedData.getBytesTlvRepresentation(), 7,
235           this.mySessionEncryptionKey);
236         childExtractionFromCTLV(this.EncryptedData);
237         childExtractionFromCTLV(this.SCUserCertificate);
238         BigInteger publicExponent = new BigInteger(byteToString
239           (this.PublicExponent.getValueBytes()), 16);
240         BigInteger publicModulus = new BigInteger(byteToString
241           (this.PublicModulus.getValueBytes()), 16);
242         KeyFactory factory = KeyFactory.getInstance("RSA");
243         SCUserVerificationKey = (PublicKey)factory.generatePublic(new
244                                 RSAPublicKeySpec(publicModulus,
245                                 publicExponent));
246         temp = new byte[(this.SCIdentity.getTagLength() +
247                         this.SCRandomNumber.getTagLength() +
248                         this.SPRandomNumber.getTagLength())];
249         System.arraycopy(this.EncryptedData.getBytesTlvRepresentation(),
                7,
250                         temp, 0, temp.length);
251         if (this.myProtocolHelperObject.SignatureMethod(temp, 0,
252             temp.length, this.SignedData.getValueBytes(), 0,
253             SCUserVerificationKey,
254             ProtocolHelperClass.SIGN_MODE_VERIFICATION)){}
255          else {
256           System.out.println(
257                           "Signature Verification Failed..... Check
                                code");
258         }
259       } else if (Counter == 2) {
260         this.MessageHandler.reset();
261         this.EncryptedData.reset();
262         this.MessageHandler.setBytesTlvRepresentation(inMessage, 0,
263           inMessage.length - 2);
264         this.childExtractionFromCTLV(this.MessageHandler);
265         byte[] temp = new byte[16];
266         this.myProtocolHelperObject.GenerateMac
267           (this.EncryptedData.getValueBytes(), 0,
268           this.EncryptedData.getValueBytes().length, temp, 0,
269           this.mySessionMacKey);
270         if (Arrays.equals(this.MACedData.getValueBytes(), temp)){}
271          else {
272           System.out.println(
273                           "Integrity Check Failure : ERROR at
                                ProtocolHandler.inMessageProcessing \n");
274           System.exit(0);
275         }
276         this.myProtocolHelperObject.GenerateDecryption
277           (this.EncryptedData.getValueBytes(), 0,
278           this.EncryptedData.getValueBytes().length,
279           this.EncryptedData.getBytesTlvRepresentation(), 7,
```

```
280             this.mySessionEncryptionKey);
281           this.childExtractionFromCTLV(EncryptedData);
282           if (Arrays.equals(PlatformHashPreset,
283               this.PlatformHash.getValueBytes())){}
284            else {
285             System.out.println("Platform Digest Not Verified");
286           }
287           childExtractionFromCTLV(this.SCCertificate);
288           BigInteger SCpublicExponent = new BigInteger(byteToString
289             (this.PublicExponent.getValueBytes()), 16);
290           BigInteger SCpublicModulus = new BigInteger(byteToString
291             (this.PublicModulus.getValueBytes()), 16);
292           KeyFactory factory = KeyFactory.getInstance("RSA");
293           SCVerificationKey = (PublicKey)factory.generatePublic(new
294                                RSAPublicKeySpec(SCpublicModulus,
295                                SCpublicExponent));
296           temp = new byte[(this.PlatformHash.getTagLength() +
297                            this.UserIdentity.getTagLength() +
298                            this.SCIdentity.getTagLength() +
299                            this.SCRandomNumber.getTagLength() +
300                            this.SPRandomNumber.getTagLength())];
301           System.arraycopy(this.EncryptedData.getBytesTlvRepresentation(),
                7,
302                            temp, 0, temp.length);
303           if (this.myProtocolHelperObject.SignatureMethod(temp, 0,
304               temp.length, this.SignedData.getValueBytes(), 0,
305               SCVerificationKey,
                    ProtocolHelperClass.SIGN_MODE_VERIFICATION))
306               {}
307            else {
308             System.out.println(
309                            "Signature Verification Failed..... Check
                                code");
310           }
311         }
312       } catch (Exception cE) {
313         System.out.println("Error in ProtocolHandler.inMessageProcessing : "
314                            + cE.getClass().getName());
315       }
316       return true;
317     }
318     public static String byteToString(byte[] inArray) {
319       byte[] HEX_CHAR_TABLE =  {
320         (byte)'0', (byte)'1', (byte)'2', (byte)'3', (byte)'4', (byte)'5',
321          (byte)'6', (byte)'7', (byte)'8', (byte)'9', (byte)'a', (byte)'b',
322          (byte)'c', (byte)'d', (byte)'e', (byte)'f'
323       };
324       byte[] hex = new byte[2 * inArray.length];
325       int index = 0;
326       for (byte b: inArray) {
327         int v = b & 0xFF;
```

```
328         hex[index++] = HEX_CHAR_TABLE[v  >>> 4];
329         hex[index++] = HEX_CHAR_TABLE[v & 0xF];
330       }
331       try {
332         return new String(hex, "ASCII");
333       } catch (Exception cE) {
334         System.out.println("Exception in bytesToString : " +
              cE.getMessage())
335                            ;
336       }
337       return "Error";
338     }
339     void childExtractionFromCTLV(ConstructedTLV inCTLV) {
340       try {
341         int childs = inCTLV.getChildNumbers();
342         PrimitiveTLV pTemp = null;
343         ConstructedTLV cTemp = null;
344         while (childs > 0) {
345           switch (inCTLV.nextType()) {
346             case 1:
347               pTemp = (PrimitiveTLV)inCTLV.getNext();
348               if (Arrays.equals(pTemp.getTagName(),
349                   this.SCDHChallenge.getTagName())) {
350                 this.SCDHChallenge = pTemp;
351               }else if (Arrays.equals(pTemp.getTagName(),
352                         this.SCRandomNumber.getTagName())) {
353                 this.SCRandomNumber = pTemp;
354               }else if (Arrays.equals(pTemp.getTagName(),
355                         this.MACedData.getTagName())) {
356                 this.MACedData = pTemp;
357               }else if (Arrays.equals(pTemp.getTagName(),
358                         this.SPCookie.getTagName())) {
359                 if (Arrays.equals(pTemp.getBytesTlvRepresentation(),
360                     this.SPCookie.getBytesTlvRepresentation())){}
361               }else if (Arrays.equals(pTemp.getTagName(),
362                         this.SCIdentity.getTagName())) {
363                 this.SCIdentity = pTemp;
364               }else if (Arrays.equals(pTemp.getTagName(),
365                         this.SignedData.getTagName())) {
366                 this.SignedData = pTemp;
367               }else if (Arrays.equals(pTemp.getTagName(),
368                         this.PublicExponent.getTagName())) {
369                 this.PublicExponent = pTemp;
370               }else if (Arrays.equals(pTemp.getTagName(),
371                         this.PublicModulus.getTagName())) {
372                 this.PublicModulus = pTemp;
373               }else if (Arrays.equals(pTemp.getTagName(),
374                         this.PlatformHash.getTagName())) {
375                 this.PlatformHash = pTemp;
376               }else if (Arrays.equals(pTemp.getTagName(),
377                         this.UserIdentity.getTagName())) {
```

```
378                 this.UserIdentity = pTemp;
379               }
380               break;
381             case 0:
382               cTemp = (ConstructedTLV)inCTLV.getNext();
383               if (Arrays.equals(cTemp.getTagName(),
384                   this.EncryptedData.getTagName())) {
385                 this.EncryptedData = cTemp;
386               }else if (Arrays.equals(cTemp.getTagName(),
387                         SCUserCertificate.getTagName())) {
388                 this.SCUserCertificate = cTemp;
389               }else if (Arrays.equals(cTemp.getTagName(),
390                         SCCertificate.getTagName())) {
391                 this.SCCertificate = cTemp;
392               }
393               break;
394             default:
395               System.out.println("Error In Parsing Input Message");
396           }
397           childs--;
398         }
399       } catch (Exception e) {
400         System.out.println(
401                             "Error in ProtocolHanlder.ChildExtractionMethod
                                : " + e.getClass().getName());
402       }
403     }
404     void GenerateKeys(byte[] inbuff) {
405       byte[] DHSecretKey = null;
406       try {
407         DHSecretKey =
408           this.myProtocolHelperObject.GenerateDHSessionKeyMaterial(inbuff,
                0,
409           inbuff.length);
410       } catch (Exception cE) {
411         System.out.println(
412                             "Exception At ProtocolHelperClass.GenerateKeys :
                                " + cE.getClass().getName());
413       }
414       byte[] keyGenKey = new byte[16];
415       System.arraycopy(DHSecretKey, 0, keyGenKey, 0, keyGenKey.length);
416       byte[] macInputValue = new byte[64];
417       System.arraycopy(this.SPRandomNumber.getValueBytes(), 0,
            macInputValue,
418                        0, 16);
419       System.arraycopy(this.SCRandomNumber.getValueBytes(), 0,
            macInputValue,
420                        16, 16);
421       System.arraycopy(DHSecretKey, 16, macInputValue, 32, 16);
422       for (int i = 48; i < 64; i++) {
423         macInputValue[i] = (byte)0x02;
```

```
424     }
425     try {
426       this.myProtocolHelperObject.GenerateMac(macInputValue, 0,
427         macInputValue.length, this.mySessionEncryptionKey, 0, keyGenKey);
428     } catch (Exception cE) {
429       System.out.println("Exception at ProtocolHandler.GenerateKeys : " +
430                          cE.getClass().getName());
431     }
432     for (int i = 48; i < 64; i++) {
433       macInputValue[i] = (byte)0x03;
434     }
435     try {
436       this.myProtocolHelperObject.GenerateMac(macInputValue, 0,
437         macInputValue.length, this.mySessionMacKey, 0, keyGenKey);
438     } catch (Exception cE) {
439       System.out.println("Exception at ProtocolHandler.GenerateKeys : " +
440                          cE.getClass().getName());
441     }
442   }
443 }
```

C.6.3 Administrative Authority Implementation

Below is the code related to the administrative authority's implementation for the STCP_{ACA}.

```
1  package ACAPTerminal;
2
3  import java.math.BigInteger;
4  import java.security.*;
5  import java.security.interfaces.RSAPublicKey;
6  import java.security.spec.RSAPublicKeySpec;
7  import java.util.Arrays;
8  public class TSMProtocolHandler {
9    private byte[] AppAct = {
10     (byte)0x7d, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6, (byte)
11      0xC1, (byte)0x2e, (byte)0x07, (byte)0xe9, (byte)0x69, (byte)0x8D,
12      (byte)0x11, (byte)0xB6, (byte)0xC1, (byte)0x2e, (byte)0x07,
13      (byte)0xe9, (byte)0x69, };
14   private byte[] AppActTag = {
15     (byte)0x9A, (byte)0x9B};
16   private byte[] CardIDTag = {
17     (byte)0x5F, (byte)0x05};
18   private byte[] EncryptedDataTag = {
19     (byte)0xFE, (byte)0x01};
20   private byte[] LongTermEncryptionKey = new byte[16];
21   private byte[] LongTermMacKey = new byte[16];
22   private byte[] MACedDataTag = {
23     (byte)0x5D, (byte)0x01};
24   private byte[] MessageHandlerTSMSC = {
25     (byte)0xF1, (byte)0xF1};
26   private byte[] PublicExponentTag = {
27     (byte)0xEE, (byte)0x01};
```

```
28  private byte[] PublicModulusTag = {
29    (byte)0xEE, (byte)0x02};
30  private byte[] SCIdentityTag = {
31    (byte)0x5F, (byte)0x02};
32  private byte[] SCRandomNumberTag = {
33    (byte)0x5A, (byte)0x02};
34  private byte[] SIDTag = {
35    (byte)0x9B, (byte)0x9D};
36  private byte[] SignedDataTag = {
37    (byte)0x5D, (byte)0x02};
38  private byte[] TSMIDTag = {
39    (byte)0x5F, (byte)0x04};
40  private byte[] TSMIdentity = {
41    (byte)0x7d, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6, (byte)
42     0xC1, (byte)0x2e, (byte)0x07, (byte)0xe9, (byte)0x69, (byte)0x8D,
43     (byte)0x11, (byte)0xEf, (byte)0x34, (byte)0xfB, (byte)0xFe,
44     (byte)0x0B, (byte)0x2C};
45  private byte[] TSMRandomNumberTag = {
46    (byte)0x5A, (byte)0x04};
47  private byte[] TSMSignatureCertTag = {
48    (byte)0xF9, (byte)0xF9};
49  private byte[] TempTag = {
50    (byte)0x00, (byte)0x00};
51  private byte[] UserIdentityTag = {
52    (byte)0x5F, (byte)0x03};
53  private byte[] myLongTermEncryptionKey = {
54    (byte)0x9D, (byte)0xF3, (byte)0x0B, (byte)0x5C, (byte)0x8F, (byte)
55     0xFD, (byte)0xAC, (byte)0x50, (byte)0x6C, (byte)0xDE, (byte)0xBE,
56     (byte)0x7B, (byte)0x89, (byte)0x99, (byte)0x8C, (byte)0xAF};
57  private byte[] myLongTermMacKey = {
58    (byte)0x74, (byte)0x86, (byte)0x6A, (byte)0x08, (byte)0xCF, (byte)
59     0xE4, (byte)0xFF, (byte)0xE3, (byte)0xA6, (byte)0x82, (byte)0x4A,
60     (byte)0x4E, (byte)0x10, (byte)0xB9, (byte)0xA6, (byte)0xF0};
61  private PrimitiveTLV UserIdentity = PrimitiveTLV.getPrimitiveTLV
62    (this.UserIdentityTag);
63  private ConstructedTLV TSMSignatureCertificate =
64    ConstructedTLV.getConstructedTLV(TSMSignatureCertTag);
65  private PrimitiveTLV TSMRandomNumber = PrimitiveTLV.getPrimitiveTLV
66    (this.TSMRandomNumberTag, 16);
67  private PrimitiveTLV TSMID = PrimitiveTLV.getPrimitiveTLV(TSMIDTag,
68    TSMIdentity);
69  private PrimitiveTLV SignedData = PrimitiveTLV.getPrimitiveTLV
70    (this.SignedDataTag, 64);
71  private PrimitiveTLV SID = PrimitiveTLV.getPrimitiveTLV(SIDTag, 16);
72  private PrimitiveTLV SCRandomNumber = PrimitiveTLV.getPrimitiveTLV
73    (this.SCRandomNumberTag);
74  private PrimitiveTLV SCIdentity = PrimitiveTLV.getPrimitiveTLV
75    (SCIdentityTag);
76  private PrimitiveTLV PublicModulus = PrimitiveTLV.getPrimitiveTLV
77    (this.PublicModulusTag);
78  private PrimitiveTLV PublicExponent = PrimitiveTLV.getPrimitiveTLV
```

```
79      (this.PublicExponentTag);
80    private ConstructedTLV MessageHandler =
81      ConstructedTLV.getConstructedTLV(TempTag);
82    private PrimitiveTLV MACedData = PrimitiveTLV.getPrimitiveTLV
83      (this.MACedDataTag);
84    private ConstructedTLV EncryptedData =
85      ConstructedTLV.getConstructedTLV(this.EncryptedDataTag);
86    private PrimitiveTLV CardID = PrimitiveTLV.getPrimitiveTLV
87      (CardIDTag);
88    private PrimitiveTLV AppActTLV = PrimitiveTLV.getPrimitiveTLV
89      (this.AppActTag, this.AppAct);
90    private ProtocolHelperClass myProtocolHelperObject = new
91      ProtocolHelperClass();
92    public TSMProtocolHandler() {
93      myProtocolHelperObject.protocolInitialise();
94      RSAPublicKey tempKey = (RSAPublicKey)
95                               myProtocolHelperObject.getPublicKey();
96      byte[] tempExponent = tempKey.getPublicExponent().toByteArray();
97      this.PublicExponent.initialisationPTLV(this.PublicExponentTag,
98        tempExponent.length);
99      this.PublicExponent.setTlvValues(tempExponent);
100     byte[] tempModulus = tempKey.getModulus().toByteArray();
101     this.PublicModulus.initialisationPTLV(this.PublicModulusTag,
102       (tempModulus.length - 1));
103     this.PublicModulus.setTlvValues(tempModulus, 1,
104       (tempModulus.length - 1));
105     TSMSignatureCertificate.addPTLV(this.PublicExponent);
106     TSMSignatureCertificate.addPTLV(this.PublicModulus);
107   }
108   public void initialiseProtocol() {
109     try{}
110      catch (Exception cE) {
111       System.out.println(
112                          "Error ProtocolHandler.initialiseProtocol Option
                               = 1, : " + cE.getClass().getName());
113     }
114   }
115   public byte[] outMessageProcessing() {
116     try {
117       this.EncryptedData.reset();
118       this.EncryptedData.initialisationCTLV(this.EncryptedDataTag);
119       this.EncryptedData.addPTLV(this.TSMID);
120       this.EncryptedData.addPTLV(this.SCIdentity);
121       this.TSMRandomNumber.setTlvValues
122         (this.myProtocolHelperObject.getRandomNumber());
123       this.EncryptedData.addPTLV(this.TSMRandomNumber);
124       this.EncryptedData.addPTLV(this.SCRandomNumber);
125       this.EncryptedData.addPTLV(this.UserIdentity);
126       this.EncryptedData.addPTLV(this.AppActTLV);
127       this.myProtocolHelperObject.SignatureMethod
128         (this.EncryptedData.getValueBytes(), 0,
```

```
129           this.EncryptedData.getValueBytes().length,
130           this.SignedData.getBytesTlvRepresentation(), 6, null,
131           ProtocolHelperClass.SIGN_MODE_GENERATION);
132         this.EncryptedData.addPTLV(this.SignedData);
133         this.EncryptedData.addCTLV(this.TSMSignatureCertificate);
134         ConstructedTLV Temp = ConstructedTLV.getConstructedTLV(TempTag);
135         Temp.addPTLV(this.TSMID);
136         Temp.addPTLV(this.SCIdentity);
137         Temp.addPTLV(this.SCRandomNumber);
138         Temp.addPTLV(this.TSMRandomNumber);
139         this.myProtocolHelperObject.GenerateMac(Temp.getValueBytes(), 0,
140                             Temp.getTagValueLength(),
141                             this.CardID.getBytesTlvRepresentation(), 6,
142                             this.myLongTermMacKey);
143         Temp.addPTLV(this.CardID);
144         this.myProtocolHelperObject.GenerateMac(Temp.getValueBytes(), 0,
145                             Temp.getTagValueLength(),
                                  this.SID.getBytesTlvRepresentation()
146                             , 6, this.myLongTermMacKey);
147         this.EncryptedData.addPTLV(SID);
148         this.myProtocolHelperObject.GenerateEncryption
149           (this.EncryptedData.getValueBytes(), 0,
150           this.EncryptedData.getValueBytes().length,
151           this.EncryptedData.getBytesTlvRepresentation(), 7,
152           this.myLongTermEncryptionKey);
153         this.MACedData.initialisationPTLV(this.MACedDataTag, 16);
154         this.myProtocolHelperObject.GenerateMac
155           (this.EncryptedData.getValueBytes(), 0,
156           this.EncryptedData.getTagValueLength(),
157           this.MACedData.getBytesTlvRepresentation(), 6,
158           this.myLongTermMacKey);
159         this.MessageHandler.reset();
160         this.MessageHandler.initialisationCTLV(this.MessageHandlerTSMSC)
161           ;
162         this.MessageHandler.addCTLV(EncryptedData);
163         this.MessageHandler.addPTLV(this.MACedData);
164       } catch (Exception cE) {
165         System.out.println(
166                             "Error ProtocolHandler.inMessageProcessing
                                  Option = 1, : " + cE.getClass().getName());
167       }
168       return this.MessageHandler.getBytesTlvRepresentation();
169     }
170     public boolean inMessageProcessing(byte[] inMessage) {
171       try {
172         this.MessageHandler.reset();
173         this.EncryptedData.reset();
174         MessageHandler.setBytesTlvRepresentation(inMessage, 0,
175           (inMessage.length - 2));
176         childExtractionFromCTLV(MessageHandler);
177         byte[] temp = new byte[16];
```

```
178       this.myProtocolHelperObject.GenerateMac
179         (this.EncryptedData.getValueBytes(), 0,
180         this.EncryptedData.getValueBytes().length, temp, 0,
181         this.myLongTermMacKey);
182       if (Arrays.equals(this.MACedData.getValueBytes(), temp)){}
183        else {
184         System.out.println(
185                              "Integrity Check Failure : ERROR at
                                    ProtocolHandler.inMessageProcessing \n");
186         System.exit(0);
187       }
188       this.myProtocolHelperObject.GenerateDecryption
189         (this.EncryptedData.getValueBytes(), 0,
190         this.EncryptedData.getValueBytes().length,
191         this.EncryptedData.getBytesTlvRepresentation(), 7,
192         this.myLongTermEncryptionKey);
193       this.childExtractionFromCTLV(EncryptedData);
194     } catch (Exception cE) {
195       System.out.println(
196                            "Error in ProtocolHandler.inMessageProcessing :
                                 " + cE.getClass().getName());
197     }
198     return true;
199   }
200   public static String byteToString(byte[] inArray) {
201     byte[] HEX_CHAR_TABLE =  {
202       (byte)'0', (byte)'1', (byte)'2', (byte)'3', (byte)'4', (byte)
203        '5', (byte)'6', (byte)'7', (byte)'8', (byte)'9', (byte)'a',
204       (byte)'b', (byte)'c', (byte)'d', (byte)'e', (byte)'f'
205     };
206     byte[] hex = new byte[2 * inArray.length];
207     int index = 0;
208     for (byte b: inArray) {
209       int v = b & 0xFF;
210       hex[index++] = HEX_CHAR_TABLE[v  >>> 4];
211       hex[index++] = HEX_CHAR_TABLE[v & 0xF];
212     }
213     try {
214       return new String(hex, "ASCII");
215     } catch (Exception cE) {
216       System.out.println("Exception in bytesToString : " +
217                            cE.getMessage() + "\n" + cE.getStackTrace()
218                            .toString());
219     }
220     return "Error";
221   }
222   void childExtractionFromCTLV(ConstructedTLV inCTLV) {
223     try {
224       int childs = inCTLV.getChildNumbers();
225       PrimitiveTLV pTemp = null;
226       ConstructedTLV cTemp = null;
```

```
227         while (childs > 0) {
228           switch (inCTLV.nextType()) {
229             case 1:
230               pTemp = (PrimitiveTLV)inCTLV.getNext();
231               if (Arrays.equals(pTemp.getTagName(),
232                   this.CardID.getTagName())) {
233                 this.CardID = pTemp;
234               }
235                else if (Arrays.equals(pTemp.getTagName(),
236                         this.SCIdentity.getTagName())) {
237                 this.SCIdentity = pTemp;
238               }
239                else if (Arrays.equals(pTemp.getTagName(),
240                         this.SCRandomNumber.getTagName())) {
241                 this.SCRandomNumber = pTemp;
242               }
243                else if (Arrays.equals(pTemp.getTagName(),
244                         this.MACedData.getTagName())) {
245                 this.MACedData = pTemp;
246               }
247               if (Arrays.equals(pTemp.getTagName(),
248                   this.UserIdentity.getTagName())) {
249                 this.UserIdentity = pTemp;
250               }
251               break;
252               case 0: cTemp = (ConstructedTLV)inCTLV.getNext();
253               if (Arrays.equals(cTemp.getTagName(),
254                   this.EncryptedData.getTagName())) {
255                 this.EncryptedData = cTemp;
256               }
257               break;
258             default:
259               System.out.println("Error In Parsing Input Message");
260           }
261           childs--;
262         }
263       } catch (Exception e) {
264         System.out.println(
265                           "Error in ProtocolHanlder.ChildExtractionMethod
                                : " + e.getClass().getName());
266       }
267     }
268 }
```

C.7 Application Binding Protocol - Local

The Java Card implementation of the *ABPL* discussed in section 7.4 is listed in subsequent sections.

C.7.1 Client Application

Implementation of a client application that request for the application binding in the UCOM firewall mechanism is listed as below:

```
1  package AppBindingProt;
2
3  import javacard.framework.*;
4  import javacard.security.*;
5  import javacardx.crypto.*;
6  public class ClientApp {
7    byte[] ClientIdentity = {
8      (byte)0xbc, (byte)0xc0, (byte)0xea, (byte)0x07, (byte)0x94};
9    byte[] ServerDigest = new byte[32];
10   byte[] ServerIdentity = {
11     (byte)0x4f, (byte)0x39, (byte)0xf5, (byte)0xdb, (byte)0xd1};
12   byte[] TokenValue = {
13     (byte)0x4f, (byte)0x39, (byte)0xf5, (byte)0xdb};
14   byte[] clientR = {
15     (byte)0x4D, (byte)0xAB, (byte)0xC0, (byte)0x70, (byte)0x8B, (byte)
16      0x11, (byte)0x45, (byte)0xA9, (byte)0xCC, (byte)0xD7, (byte)0x4F,
17      (byte)0x3A, (byte)0xD8, (byte)0xBB, (byte)0xF1, (byte)0x61};
18   Cipher AESCipher;
19   RandomData clientPRNG;
20   AESKey clientTPMKey;
21   private KeyPair client_SignKeyPair;
22   short encryptionLength;
23   AESKey myClientAppServerKey;
24   Cipher myClientAppSignature;
25   ScTPM mySCTPMRef;
26   ServerApp myServerAppRef;
27   byte[] pMessage;
28   PublicKey serverVerificationKey;
29   protected ClientApp() {
30     clientPRNG = RandomData.getInstance(RandomData.ALG_PSEUDO_RANDOM);
31     client_SignKeyPair = new KeyPair(KeyPair.ALG_RSA_CRT,
32       KeyBuilder.LENGTH_RSA_512);
33     myClientAppServerKey = (AESKey)KeyBuilder.buildKey
34                              (KeyBuilder.TYPE_AES,
35                              KeyBuilder.LENGTH_AES_128, false);
36     clientTPMKey = (AESKey)KeyBuilder.buildKey(KeyBuilder.TYPE_AES,
37                       KeyBuilder.LENGTH_AES_128, false);
38     AESCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
39                                    false);
40     myClientAppSignature = Cipher.getInstance(Cipher.ALG_RSA_NOPAD,
41       false);
42   }
43   public static ClientApp objectGenerator() {
44     return new ClientApp();
45   }
46   public void objectInstantiation() {
```

```
47      client_SignKeyPair.genKeyPair();
48      clientPRNG.generateData(clientR, (short)0, (short)clientR.length);
49      clientTPMKey.setKey(clientR, (short)0);
50    }
51    public void clientUpdate(ScTPM obSCTPM, ServerApp obServerApp) {
52      mySCTPMRef = obSCTPM;
53      myServerAppRef = obServerApp;
54      myServerAppRef.clientSignVerificationUpdate
55        (client_SignKeyPair.getPublic());
56      obSCTPM.clientTPMKeyAgreement(clientTPMKey);
57    }
58    public void serverSignVerificationUpdate(PublicKey signVerification)
59      {
60      serverVerificationKey = signVerification;
61    }
62    public void digestUpdate(byte[] spServerDigest) {
63      ServerDigest = spServerDigest;
64    }
65    public byte[] startProtocol() {
66      pMessage = JCSystem.makeTransientByteArray((short)256,
67        JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
68      short initialTab = 4;
69      pMessage[0] = (byte)initialTab;
70      pMessage[0] = (byte)Util.arrayCopyNonAtomic(ClientIdentity,
71                    (short)0, pMessage, initialTab, (short)
72                    ClientIdentity.length);
73      pMessage[0] = (byte)Util.arrayCopyNonAtomic(ServerIdentity,
74                    (short)0, pMessage, pMessage[0], (short)
75                    ServerIdentity.length);
76      clientPRNG.generateData(clientR, (short)0, (short)clientR.length);
77      pMessage[0] = (byte)Util.arrayCopyNonAtomic(clientR, (short)0,
78                    pMessage, pMessage[0], (short)clientR.length);
79      generateEncryptionData(clientTPMKey);
80      for (short i = (short)((short)58+(short)initialTab); i < (short)
81           (64+initialTab); i++) {
82        pMessage[i] = (byte)0xCC;
83      }
84      generateSignatureData();
85      try {
86        myServerAppRef.protocolManager((short)1, pMessage);
87      } catch (Exception e) {
88        ISOException.throwIt((short)0xB001);
89      }
90      AESKey sessionKey = (AESKey)KeyBuilder.buildKey
91                          (KeyBuilder.TYPE_AES,
92                          KeyBuilder.LENGTH_AES_128, false);
93      generatedDecryptedData((short)4, (short)64, clientTPMKey);
94      sessionKey.setKey(pMessage, (short)36);
95      try {
96        generatedDecryptedData((short)68, (short)48, sessionKey);
97        myClientAppServerKey.setKey(pMessage, (short)68);
```

```
98      } catch (Exception e) {
99        ISOException.throwIt((short)0x0001);
100     }
101     try {
102       verifySignedData((short)116, (short)64);
103     } catch (Exception e) {
104       ISOException.throwIt((short)0x0002);
105     }
106     try {
107       generatedDecryptedData((short)116, (short)32,
108                              myClientAppServerKey);
109     } catch (Exception e) {
110       ISOException.throwIt((short)0x0003);
111     }
112     Util.arrayCopyNonAtomic(pMessage, (short)116, TokenValue, (short)
113                             0, (short)TokenValue.length);
114     pMessage[0] = (byte)Util.arrayCopyNonAtomic(ClientIdentity,
115                   (short)0, pMessage, initialTab, (short)
116                   ClientIdentity.length);
117     pMessage[0] = (byte)Util.arrayCopyNonAtomic(ServerIdentity,
118                   (short)0, pMessage, pMessage[0], (short)
119                   ServerIdentity.length);
120     pMessage[2] = pMessage[0];
121     pMessage[0] = (byte)Util.arrayCopyNonAtomic(TokenValue, (short)0,
122                   pMessage, pMessage[0], (short)TokenValue.length);
123     pMessage[0] = (byte)Util.arrayCopyNonAtomic(clientR, (short)0,
124                   pMessage, pMessage[0], (short)clientR.length);
125     pMessage[0] = (byte)(pMessage[0] - pMessage[2]);
126     try {
127       encryptData((short)4, (short)30, myClientAppServerKey);
128     } catch (Exception e) {
129       ISOException.throwIt((short)0x0004);
130     }
131     try {
132       myServerAppRef.protocolManager((short)2, pMessage);
133     } catch (Exception e) {
134       ISOException.throwIt((short)0x00A5);
135     }
136     return TokenValue;
137   }
138   public void protocolManager(byte[] pMessage){}
139   protected void generateEncryptionData(AESKey Key) {
140     pMessage[3] += (short)(pMessage[0] - 4);
141     AESCipher.init(Key, Cipher.MODE_ENCRYPT);
142     short paddingbytes = (short)(16-((pMessage[0] % 16) - (short)4));
143     if (paddingbytes != 0) {
144       for (short i = 0; i < paddingbytes; i++) {
145         pMessage[(short)(pMessage[0] + i)] = (byte)0xFF;
146       }
147     }
148     pMessage[0] += (byte)paddingbytes;
```

```
149     byte[] temp = new byte[pMessage[0]];
150     pMessage[1] = (byte)AESCipher.doFinal(pMessage, (short)4, (short)
151                     (pMessage[0] - 4), temp, (short)0);
152     pMessage[3] += pMessage[1];
153     pMessage[0] -= (byte)paddingbytes;
154     Util.arrayCopyNonAtomic(temp, (short)0, pMessage, pMessage[0],
155                             (short)pMessage[1]);
156   }
157   protected void encryptData(short start, short length, AESKey Key) {
158     short paddingbytes = 0;
159     if ((short)(length % 16) != 0) {
160       paddingbytes = (short)(16-(length % 16));
161     }
162     byte[] temp = JCSystem.makeTransientByteArray((short)(length +
163       paddingbytes), JCSystem.CLEAR_ON_DESELECT);
164     AESCipher.init(Key, Cipher.MODE_ENCRYPT);
165     Util.arrayCopyNonAtomic(pMessage, (short)start, temp, (short)0,
166                             (short)length);
167     if (paddingbytes != 0) {
168       for (short i = 0; i < paddingbytes; i++, length++) {
169         temp[(short)(length)] = (byte)0xFF;
170       }
171     }
172     AESCipher.doFinal(temp, (short)0, (short)length, pMessage, (short)
173                       start);
174   }
175   protected void generateSignatureData() {
176     byte[] sigBuff = JCSystem.makeTransientByteArray((short)256,
177       JCSystem.CLEAR_ON_DESELECT);
178     short sigLen = 0;
179     myClientAppSignature.init(client_SignKeyPair.getPrivate(),
180                               Cipher.MODE_ENCRYPT);
181     sigLen = myClientAppSignature.doFinal(pMessage, (short)4, (short)
182       64, sigBuff, (short)0);
183     Util.arrayCopyNonAtomic(sigBuff, (short)0, pMessage, (short)4,
184                             sigLen);
185     pMessage[2] = (byte)sigLen;
186   }
187   protected void generatedDecryptedData(short start, short length,
188     AESKey Key) {
189     byte[] tempBuff = JCSystem.makeTransientByteArray(length,
190       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
191     AESCipher.init(Key, Cipher.MODE_DECRYPT);
192     Util.arrayCopyNonAtomic(pMessage, start, tempBuff, (short)0,
193                             (short)length);
194     AESCipher.doFinal(tempBuff, (short)0, (short)length, pMessage,
195                       (short)start);
196   }
197   protected boolean verifySignedData(short start, short length) {
198     myClientAppSignature.init(serverVerificationKey,
199                               Cipher.MODE_DECRYPT);
```

```
200     byte[] tempBuff = JCSystem.makeTransientByteArray((short)256,
201       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
202     Util.arrayCopyNonAtomic(pMessage, (short)start, tempBuff, (short)
203                                 0, (short)length);
204     myClientAppSignature.doFinal(tempBuff, (short)0, (short)length,
205                                     pMessage, (short)start);
206     return true;
207   }
208 }
```

C.7.2 Server Application

Implementation of a server application that responds to the application binding request in the UCOM firewall mechanism is listed as below:

```
1  package AppBindingProt;
2
3  import javacard.framework.*;
4  import javacard.security.*;
5  import javacardx.crypto.*;
6  public class ServerApp {
7    byte[] ClientDigest = new byte[32];
8    byte[] ClientIdentity = {
9      (byte)0xbc, (byte)0xc0, (byte)0xea, (byte)0x07, (byte)0x94};
10   byte[] RandomNumberClient = new byte[16];
11   byte[] RandomNumberServer = {
12     (byte)0x04, (byte)0x95, (byte)0x5E, (byte)0x4F, (byte)0x13, (byte)
13      0x9A, (byte)0x06, (byte)0x89, (byte)0x2C, (byte)0x3D, (byte)0x79,
14      (byte)0xFA, (byte)0xD1, (byte)0xAB, (byte)0x2D, (byte)0x5F};
15   byte[] ServerIdentity = {
16     (byte)0x4f, (byte)0x39, (byte)0xf5, (byte)0xdb, (byte)0xd1};
17   byte[] TokenValue = {
18     (byte)0xbc, (byte)0xbc, (byte)0xbc, (byte)0xbc};
19   RandomData myServerAppRandomData = RandomData.getInstance
20     (RandomData.ALG_PSEUDO_RANDOM);
21   KeyPair server_SignKeyPair = new KeyPair(KeyPair.ALG_RSA_CRT,
22     KeyBuilder.LENGTH_RSA_512);
23   AESKey SerTpmKey = (AESKey)KeyBuilder.buildKey(KeyBuilder.TYPE_AES,
24                         KeyBuilder.LENGTH_AES_128, false);
25   Cipher AESCipher = Cipher.getInstance
26     (Cipher.ALG_AES_BLOCK_128_CBC_NOPAD, false);
27   ClientApp myClientAppRef;
28   PublicKey myClientVerificationKey;
29   Cipher myServerAppSignature;
30   AESKey myServerClientAppKey;
31   ScTPM myTPMRef;
32   byte[] pMessage;
33   AESKey sessionKey;
34   protected ServerApp() {
35     myServerAppSignature = Cipher.getInstance(Cipher.ALG_RSA_NOPAD,
36       false);
```

```
37  }
38  public static ServerApp objectGenerator() {
39    return new ServerApp();
40  }
41  public void objectInstantiation() {
42    server_SignKeyPair.genKeyPair();
43    SerTpmKey.setKey(RandomNumberServer, (short)0);
44  }
45  public void serverUpdate(ScTPM obScTPM, ClientApp obClientApp) {
46    myTPMRef = obScTPM;
47    myClientAppRef = obClientApp;
48    myClientAppRef.serverSignVerificationUpdate
49      (server_SignKeyPair.getPublic());
50    obScTPM.serverTPMKeyAgreement(SerTpmKey);
51  }
52  public void clientSignVerificationUpdate(PublicKey signVerification)
53    {
54    myClientVerificationKey = signVerification;
55  }
56  public void digestUpdate(byte[] spClientDigest) {
57    ClientDigest = spClientDigest;
58  }
59  public void protocolManager(short stage, byte[] p_Message) {
60    this.pMessage = p_Message;
61    if (stage == 1) {
62      pMessage[3] = (byte)64;
63      verifySignedData();
64      Util.arrayCopyNonAtomic(pMessage, (short)(4
65                              +ClientIdentity.length +
66                              ServerIdentity.length),
67                              RandomNumberClient, (short)0, (short)
68                              RandomNumberClient.length);
69      pMessage[3] = (byte)68;
70      pMessage[2] = pMessage[3];
71      pMessage[3] = (byte)Util.arrayCopyNonAtomic(ServerIdentity,
72                      (short)0, pMessage, (short)pMessage[3], (short)
73                      ServerIdentity.length);
74      pMessage[3] = (byte)Util.arrayCopyNonAtomic(ClientIdentity,
75                      (short)0, pMessage, (short)pMessage[3], (short)
76                      ClientIdentity.length);
77      myServerAppRandomData.generateData(RandomNumberServer, (short)0,
78        (short)RandomNumberServer.length);
79      pMessage[3] = (byte)Util.arrayCopyNonAtomic(RandomNumberServer,
80                      (short)0, pMessage, (short)pMessage[3], (short)
81                      RandomNumberServer.length);
82      pMessage[0] = (byte)(ClientIdentity.length +
83                      ServerIdentity.length +
84                      RandomNumberServer.length);
85      for (short i = 0; i < 6; i++) {
86        pMessage[(short)(pMessage[3] + i)] = (byte)0xCA;
87      }
```

```
88        pMessage[2] = (byte)32;
89        pMessage[0] = (byte)68;
90        generateEncryptedData((short)pMessage[0], (short)pMessage[2],
91                              SerTpmKey);
92        myTPMRef.validateApplications(pMessage);
93        generatedDecryptedData((short)68, (short)64, SerTpmKey);
94        sessionKey = (AESKey)KeyBuilder.buildKey(KeyBuilder.TYPE_AES,
95                     KeyBuilder.LENGTH_AES_128, false);
96        sessionKey.setKey(pMessage, (short)(100));
97        myServerClientAppKey = (AESKey)KeyBuilder.buildKey
98                               (KeyBuilder.TYPE_AES,
99                               KeyBuilder.LENGTH_AES_128, false);
100       byte[] keyGenerationArray = JCSystem.makeTransientByteArray(
101         (short)16, JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
102       myServerAppRandomData.generateData(keyGenerationArray, (short)0,
103         (short)keyGenerationArray.length);
104       myServerClientAppKey.setKey(keyGenerationArray, (short)0);
105       for (short i = 0; i < 16; i++) {
106         RandomNumberClient[i] = (byte)0xFF;
107       }
108       myServerClientAppKey.getKey(pMessage, (short)68);
109       Util.arrayCopyNonAtomic(RandomNumberClient, (short)0, p_Message,
110                               (short)84, (short)
111                               RandomNumberClient.length);
112       Util.arrayCopyNonAtomic(RandomNumberServer, (short)0, p_Message,
113                               (short)100, (short)
114                               RandomNumberServer.length);
115       generateEncryptedData((short)68, (short)48, sessionKey);
116       Util.arrayCopyNonAtomic(TokenValue, (short)0, pMessage, (short)
117                               116, (short)TokenValue.length);
118       xorRandomNumberCS((short)120);
119       generateEncryptedData((short)116, (short)20,
120                             myServerClientAppKey);
121       generateSignatureData((short)116, (short)32);
122       return ;
123     }
124     if (stage == 2) {
125       generatedDecryptedData((short)4, (short)32,
126                              myServerClientAppKey);
127       if ((byte)Util.arrayCompare(TokenValue, (short)0, pMessage,
128           (short)14, (short)4) == (byte)0) {
129         return ;
130       } else {
131         ISOException.throwIt((short)0xFFFF);
132       }
133     } else {
134       ISOException.throwIt((short)0x6300);
135     }
136   }
137   protected void xorRandomNumberCS(short start) {
138     for (short i = 0; i < (short)16; i++, start++) {
```

```
139       pMessage[start] = (byte)(RandomNumberServer[i] |
140                                RandomNumberClient[i]);
141     }
142   }
143   protected void generateEncryptedData(short start, short length,
144     AESKey Key) {
145     short paddingbytes = 0;
146     if ((short)(length % 16) != 0) {
147       paddingbytes = (short)(16-(length % 16));
148     }
149     byte[] temp = JCSystem.makeTransientByteArray((short)(length +
150       paddingbytes), JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
151     AESCipher.init(Key, Cipher.MODE_ENCRYPT);
152     Util.arrayCopyNonAtomic(pMessage, (short)start, temp, (short)0,
153                             (short)length);
154     if (paddingbytes != 0) {
155       for (short i = 0; i < paddingbytes; i++, length++) {
156         temp[(short)(length)] = (byte)0xFF;
157       }
158     }
159     AESCipher.doFinal(temp, (short)0, (short)length, pMessage, (short)
160                       start);
161   }
162   protected void generatedDecryptedData(short start, short length,
163     AESKey Key) {
164     byte[] tempBuff = JCSystem.makeTransientByteArray(length,
165       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
166     AESCipher.init(Key, Cipher.MODE_DECRYPT);
167     Util.arrayCopyNonAtomic(pMessage, start, tempBuff, (short)0,
168                             (short)length);
169     AESCipher.doFinal(tempBuff, (short)0, (short)length, pMessage,
170                       (short)start);
171   }
172   protected void generateSignatureData(short start, short length) {
173     byte[] sigBuff = JCSystem.makeTransientByteArray((short)(64+2),
174       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
175     myServerAppSignature.init(server_SignKeyPair.getPrivate(),
176                               Cipher.MODE_ENCRYPT);
177     if (length < 64) {
178       for (short i = 0; i < (short)32; i++) {
179         pMessage[(short)(start + length + i)] = (byte)0x5A;
180       }
181     }
182     myServerAppSignature.doFinal(pMessage, (short)start, (short)64,
183                                  sigBuff, (short)0);
184     Util.arrayCopyNonAtomic(sigBuff, (short)0, pMessage, (short)start,
185                             (short)64);
186   }
187   protected boolean verifySignedData() {
188     myServerAppSignature.init(myClientVerificationKey,
189                               Cipher.MODE_DECRYPT);
```

```
190       byte[] tempBuff = JCSystem.makeTransientByteArray((short)256,
191         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
192       Util.arrayCopyNonAtomic(pMessage, (short)4, tempBuff, (short)0,
193                                 (short)64);
194       myServerAppSignature.doFinal(tempBuff, (short)0, (short)64,
195                                       pMessage, (short)4);
196       return true;
197     }
198 }
```

C.7.3 TEM Handler

Implementation of TEM handler that generates the state proof of individual applications in the UCOM firewall mechanism is listed as below:

```
1  package AppBindingProt;
2
3  import javacard.framework.*;
4  import javacard.security.*;
5  import javacardx.crypto.*;
6  public class ScTPM {
7    private static byte[] AppDataFile =  {
8      (byte)0x37, (byte)0x7a, (byte)0xbc, (byte)0xc0, (byte)0xea, (byte)
9       0x07, (byte)0x94, (byte)0x59, (byte)0xd6, (byte)0x37, (byte)0x6b,
10      (byte)0x4c, (byte)0x82, (byte)0xdb, (byte)0x54, (byte)0xb2,
11      (byte)0xe8, (byte)0xea, (byte)0x71, (byte)0xe1, (byte)0xa4,
12      (byte)0x41, (byte)0x06, (byte)0x44, (byte)0xfe, (byte)0x86,
13      (byte)0x8e, (byte)0x4f, (byte)0x39, (byte)0xf5, (byte)0xdb,
14      (byte)0xd1, (byte)0xf1, (byte)0xc5, (byte)0xd8, (byte)0xac,
15      (byte)0xbb, (byte)0x73, (byte)0x51, (byte)0xa1, (byte)0xa3,
16      (byte)0x8a, (byte)0x26, (byte)0x5d, (byte)0xf3, (byte)0x61,
17      (byte)0x55, (byte)0x56, (byte)0x39, (byte)0x3f, (byte)0x4c,
18      (byte)0x2a, (byte)0x43, (byte)0xc4, (byte)0xd7, (byte)0xa1,
19      (byte)0xaa, (byte)0xc1, (byte)0xf2, (byte)0xd6, (byte)0x07,
20      (byte)0xa8, (byte)0x58, (byte)0x9a, (byte)0x70, (byte)0x84,
21      (byte)0x15, (byte)0x19, (byte)0x56, (byte)0x61, (byte)0x3d,
22      (byte)0x88, (byte)0x2a, (byte)0x44, (byte)0x54, (byte)0x29,
23      (byte)0x29, (byte)0x26, (byte)0x36, (byte)0x06, (byte)0xfe,
24      (byte)0xad, (byte)0x27, (byte)0x13, (byte)0x86, (byte)0x0e,
25      (byte)0x85, (byte)0x3c, (byte)0x32, (byte)0xe2, (byte)0x38,
26      (byte)0xd2, (byte)0x91, (byte)0x82, (byte)0x89, (byte)0xce,
27      (byte)0x79, (byte)0x02, (byte)0x43, (byte)0xfd, (byte)0xaf,
28      (byte)0x18, (byte)0xe8, (byte)0x5b, (byte)0xd4, (byte)0x72,
29      (byte)0x03, (byte)0x63, (byte)0x2b, (byte)0x29, (byte)0x72,
30      (byte)0xe0, (byte)0x92, (byte)0x54, (byte)0x06, (byte)0x1c,
31      (byte)0x7f, (byte)0xc7, (byte)0x37, (byte)0x93, (byte)0x2f,
32      (byte)0x7a, (byte)0x84, (byte)0x95, (byte)0xec, (byte)0x5e,
33      (byte)0xa5, (byte)0xf6, (byte)0x4e, (byte)0x7e, (byte)0x1f,
34      (byte)0xe6, (byte)0xe2, (byte)0x04, (byte)0x2e, (byte)0x25,
35      (byte)0x7f, (byte)0x2f, (byte)0x3c, (byte)0xfe, (byte)0x57,
36      (byte)0x9e, (byte)0x7f, (byte)0xce, (byte)0x72, (byte)0xc0,
```

```
37      (byte)0xe9, (byte)0x79, (byte)0x05, (byte)0xc5, (byte)0xfd,
38      (byte)0x6a, (byte)0x46, (byte)0xfe, (byte)0x33, (byte)0x84,
39      (byte)0x3f, (byte)0x09, (byte)0xae, (byte)0x01, (byte)0x18,
40      (byte)0x5a, (byte)0xf6, (byte)0xc6, (byte)0xd3, (byte)0xa1,
41      (byte)0xe2, (byte)0x90, (byte)0x83, (byte)0x79, (byte)0xee,
42      (byte)0xa6, (byte)0xd4, (byte)0xf6, (byte)0xd1, (byte)0x86,
43      (byte)0x91, (byte)0x34, (byte)0x00, (byte)0xd3, (byte)0xe4,
44      (byte)0x8a, (byte)0xfb, (byte)0xaa, (byte)0x6c, (byte)0xe5,
45      (byte)0x46, (byte)0xa7, (byte)0x00, (byte)0x9e, (byte)0xd8,
46      (byte)0x81, (byte)0xbc, (byte)0xd1, (byte)0xb5, (byte)0x60,
47      (byte)0xd5, (byte)0x91, (byte)0x13, (byte)0x06, (byte)0x68,
48      (byte)0x21, (byte)0x8f, (byte)0x7d, (byte)0xc2, (byte)0x3e,
49      (byte)0xd2, (byte)0x75, (byte)0x0f, (byte)0x97, (byte)0x64,
50      (byte)0xb1, (byte)0xdb, (byte)0x74, (byte)0x6e, (byte)0x91,
51      (byte)0x6b, (byte)0xa7, (byte)0x7d, (byte)0xef, (byte)0x8b,
52      (byte)0x37, (byte)0xb7, (byte)0x84, (byte)0x1e, (byte)0xa7,
53      (byte)0x26, (byte)0x26, (byte)0xea, (byte)0xe9, (byte)0xb7,
54      (byte)0x5e, (byte)0x3f, (byte)0xdf, (byte)0xa4, (byte)0xc5,
55      (byte)0x45, (byte)0x4e, (byte)0x34, (byte)0x33, (byte)0xe5,
56      (byte)0x43, (byte)0x46, (byte)0xc0, (byte)0x2b, (byte)0xbd,
57      (byte)0x85, (byte)0x2f, (byte)0xca, (byte)0xf8, (byte)0x9d,
58      (byte)0xb4, (byte)0xbc, (byte)0x67, (byte)0x92, (byte)0xd4,
59      (byte)0x33, (byte)0xfd, (byte)0xbd, (byte)0x82, (byte)0x9d,
60      (byte)0x62, (byte)0xfc, (byte)0xbb, (byte)0xd2, (byte)0xad,
61      (byte)0x05, (byte)0xa2, (byte)0xfc, (byte)0x2d, (byte)0xe3,
62      (byte)0x02, (byte)0xe2, (byte)0x41, (byte)0x9b, (byte)0x1f,
63      (byte)0xf8, (byte)0x87, (byte)0x15, (byte)0x89, (byte)0xfb,
64      (byte)0x53, (byte)0x99, (byte)0xb3, (byte)0xeb, (byte)0xdb,
65      (byte)0x01, (byte)0xaf, (byte)0x71, (byte)0xd2, (byte)0xf2,
66      (byte)0x73, (byte)0xb7, (byte)0x82, (byte)0x30, (byte)0x25,
67      (byte)0x04, (byte)0x29, (byte)0x2b, (byte)0xb9, (byte)0x92,
68      (byte)0x92, (byte)0x35, (byte)0x97, (byte)0x0e, (byte)0xb8,
69      (byte)0xf2, (byte)0xc6, (byte)0x2e, (byte)0xa7, (byte)0x2d,
70      (byte)0x0c, (byte)0x09, (byte)0x5e, (byte)0x07, (byte)0x06,
71      (byte)0x67, (byte)0xa0, (byte)0xdf, (byte)0x55, (byte)0x09,
72      (byte)0xfc, (byte)0xee, (byte)0x2b, (byte)0x13, (byte)0x1a,
73      (byte)0x2e, (byte)0x5d, (byte)0x0a, (byte)0xbb, (byte)0x45,
74      (byte)0x75, (byte)0xf4, (byte)0xd8, (byte)0xdc, (byte)0x2e,
75      (byte)0x99, (byte)0x2a, (byte)0x13, (byte)0xa1, (byte)0x1e,
76      (byte)0x99, (byte)0xfd, (byte)0xdc, (byte)0xcf, (byte)0xcc,
77      (byte)0x3f, (byte)0x42, (byte)0xf7, (byte)0x3d, (byte)0x73,
78      (byte)0xee, (byte)0xca, (byte)0x76, (byte)0xe4, (byte)0x75,
79      (byte)0xc4, (byte)0x21, (byte)0xd4, (byte)0x14, (byte)0x2e,
80      (byte)0x22, (byte)0x9c, (byte)0xce, (byte)0x10, (byte)0xaf,
81      (byte)0xa6, (byte)0x25, (byte)0xa0, (byte)0x01, (byte)0xb1,
82      (byte)0x82, (byte)0xba, (byte)0x4c, (byte)0xb2, (byte)0x66,
83      (byte)0x89, (byte)0x89, (byte)0x6b, (byte)0x06, (byte)0x15,
84      (byte)0xba, (byte)0x64, (byte)0xa3, (byte)0x73, (byte)0x88,
85      (byte)0x34, (byte)0x99, (byte)0x3e, (byte)0x75, (byte)0x24,
86      (byte)0xf4, (byte)0xba, (byte)0xb0, (byte)0x22, (byte)0x8f,
87      (byte)0xc3, (byte)0x44, (byte)0x74, (byte)0x0b, (byte)0x52,
```

```
88      (byte)0x96, (byte)0xc6, (byte)0x97, (byte)0x8b, (byte)0xf2,
89      (byte)0xe3, (byte)0xc1, (byte)0xaf, (byte)0x53, (byte)0x03,
90      (byte)0x51, (byte)0xa7, (byte)0x0d, (byte)0x42, (byte)0x6a,
91      (byte)0x20, (byte)0x03, (byte)0x31, (byte)0xb4, (byte)0xc9,
92      (byte)0xaa, (byte)0x9e, (byte)0xda, (byte)0x6f, (byte)0x7b,
93      (byte)0xb8, (byte)0x6d, (byte)0x54, (byte)0x57, (byte)0xa8,
94      (byte)0xed, (byte)0x51, (byte)0xa4, (byte)0x23, (byte)0x05,
95      (byte)0x0b, (byte)0xb3, (byte)0x90, (byte)0x42, (byte)0x38,
96      (byte)0xa8, (byte)0xbc, (byte)0xd5, (byte)0x2f, (byte)0x87,
97      (byte)0x82, (byte)0x5b, (byte)0xff, (byte)0xdb, (byte)0xba,
98      (byte)0x41, (byte)0x18, (byte)0xe0, (byte)0x4a, (byte)0x07,
99      (byte)0x04, (byte)0xe1, (byte)0x3c, (byte)0xd5, (byte)0xbf,
100     (byte)0x3e, (byte)0x40, (byte)0x7a, (byte)0x33, (byte)0x2a,
101     (byte)0x61, (byte)0x33, (byte)0xa5, (byte)0xb4, (byte)0x06,
102     (byte)0x96, (byte)0xc0, (byte)0xaa, (byte)0xdb, (byte)0x79,
103     (byte)0x46, (byte)0xe7, (byte)0xe5, (byte)0x6d, (byte)0xae,
104     (byte)0x16, (byte)0x6d, (byte)0xa9, (byte)0x4a, (byte)0x39,
105     (byte)0x0e, (byte)0x5b, (byte)0x99, (byte)0x35, (byte)0x42,
106     (byte)0xc3, (byte)0xac, (byte)0xc1, (byte)0x1b, (byte)0x6c,
107     (byte)0x3b, (byte)0xdc, (byte)0x74, (byte)0x3b, (byte)0x52,
108     (byte)0x5d, (byte)0x74, (byte)0x30, (byte)0x77, (byte)0x3f,
109     (byte)0x95, (byte)0xf3, (byte)0x92, (byte)0xfb, (byte)0xf6,
110     (byte)0xd7, (byte)0x94, (byte)0x49, (byte)0x63, (byte)0x56,
111     (byte)0xd4, (byte)0x8c, (byte)0x7d, (byte)0x56, (byte)0x84,
112     (byte)0xca, (byte)0x54, (byte)0x77, (byte)0x29, (byte)0x3b,
113     (byte)0xa7, (byte)0x60};
114   public byte[] ClientIdentity = {
115     (byte)0xbc, (byte)0xc0, (byte)0xea, (byte)0x07, (byte)0x94};
116   public byte[] ServerIdentity = {
117     (byte)0x4f, (byte)0x39, (byte)0xf5, (byte)0xdb, (byte)0xd1};
118   byte[] scTPMDigestBuffer = new byte[(short)32];
119   MessageDigest tpmDigestGen = MessageDigest.getInstance
120     (MessageDigest.ALG_SHA_256, false);
121   byte[] ServerRandomNumber = JCSystem.makeTransientByteArray((short)
122     16, JCSystem.CLEAR_ON_RESET);
123   byte[] ClientRandomNumber = JCSystem.makeTransientByteArray((short)
124     16, JCSystem.CLEAR_ON_RESET);
125   Cipher AESCipher;
126   AESKey TpmClientApp, TpmServerApp;
127   ClientApp myClientAppRef;
128   ServerApp myServerAppRef;
129   byte[] pMessage;
130   protected ScTPM(){}
131   public void instantiateObject() {
132     tpmDigestGen.doFinal(AppDataFile, (short)0, (short)
133                          AppDataFile.length, scTPMDigestBuffer,
134                          (short)0);
135     AESCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
136                                    false);
137   }
138   public static ScTPM objectGenerator() {
```

```
139     return new ScTPM();
140   }
141   public void ScTPMUpdate(ServerApp obServerApp, ClientApp
142                           obClientApp) {
143     myServerAppRef = obServerApp;
144     myClientAppRef = obClientApp;
145     myServerAppRef.digestUpdate(scTPMDigestBuffer);
146     myClientAppRef.digestUpdate(scTPMDigestBuffer);
147   }
148   public void clientTPMKeyAgreement(AESKey TPMClient) {
149     TpmClientApp = TPMClient;
150   }
151   public void serverTPMKeyAgreement(AESKey TPMServer) {
152     TpmServerApp = TPMServer;
153   }
154   public void validateApplications(byte[] p_Message) {
155     pMessage = p_Message;
156     generateDecrytion((short)30, (short)32, TpmClientApp);
157     Util.arrayCopyNonAtomic(pMessage, (short)(pMessage[0] +
158                             ClientIdentity.length +
159                             ServerIdentity.length),
160                             ClientRandomNumber, (short)0, (short)
161                             ClientRandomNumber.length);
162     generateDecrytion((short)68, (short)32, TpmServerApp);
163     Util.arrayCopyNonAtomic(pMessage, (short)(pMessage[0] +
164                             ClientIdentity.length +
165                             ServerIdentity.length),
166                             ServerRandomNumber, (short)0, (short)
167                             ServerRandomNumber.length);
168     tpmDigestGen.doFinal(AppDataFile, (short)0, (short)
169                          AppDataFile.length, scTPMDigestBuffer,
170                          (short)0);
171     tpmDigestGen.doFinal(AppDataFile, (short)0, (short)
172                          AppDataFile.length, scTPMDigestBuffer,
173                          (short)0);
174     AESKey sessionKey;
175     byte[] tempDebugSessionKey = JCSystem.makeTransientByteArray(
176       (short)16, JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
177     sessionKey = (AESKey)KeyBuilder.buildKey(KeyBuilder.TYPE_AES,
178                  KeyBuilder.LENGTH_AES_128, false);
179     RandomData tpmKeyGenPRNG = RandomData.getInstance
180       (RandomData.ALG_PSEUDO_RANDOM);
181     tpmKeyGenPRNG.generateData(tempDebugSessionKey, (short)0, (short)
182                                tempDebugSessionKey.length);
183     sessionKey.setKey(tempDebugSessionKey, (short)0);
184     pMessage[0] = (byte)Util.arrayCopyNonAtomic(scTPMDigestBuffer,
185                   (short)0, pMessage, (short)4, (short)
186                   scTPMDigestBuffer.length);
187     pMessage[0] += (byte)sessionKey.getKey(pMessage, (short)
188                    (pMessage[0]));
189     ClientRandomNumber[15] = (byte)(ClientRandomNumber[15] | (byte)
```

```
190                                     0x01);
191     pMessage[0] = (byte)Util.arrayCopyNonAtomic(ClientRandomNumber,
192                         (short)0, pMessage, (short)pMessage[0], (short)
193                         ClientRandomNumber.length);
194     generateEncryption((short)4, (short)64, TpmClientApp);
195     pMessage[0] = (byte)Util.arrayCopyNonAtomic(scTPMDigestBuffer,
196                         (short)0, pMessage, (short)68, (short)
197                         scTPMDigestBuffer.length);
198     pMessage[0] += (byte)sessionKey.getKey(pMessage, (short)
199                          pMessage[0]);
200     ServerRandomNumber[15] = (byte)(ServerRandomNumber[15] + (byte)1);
201     pMessage[0] = (byte)Util.arrayCopyNonAtomic(ServerRandomNumber,
202                         (short)0, pMessage, (short)pMessage[0], (short)
203                         ServerRandomNumber.length);
204     generateEncryption((short)68, (short)64, TpmServerApp);
205   }
206   public void generateDecrytion(short start, short length, AESKey Key)
207                                       {
208     byte[] tempBuff = JCSystem.makeTransientByteArray(length,
209       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
210     AESCipher.init(Key, Cipher.MODE_DECRYPT);
211     Util.arrayCopyNonAtomic(pMessage, (short)start, tempBuff, (short)
212                                   0, (short)length);
213     AESCipher.doFinal(tempBuff, (short)0, (short)length, pMessage,
214                           (short)start);
215   }
216   public void generateEncryption(short start, short length, AESKey
217                                        Key) {
218     AESCipher.init(Key, Cipher.MODE_ENCRYPT);
219     short paddingbytes = (short)(length % 16);
220     byte[] temp = JCSystem.makeTransientByteArray((short)(length +
221       paddingbytes), JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
222     Util.arrayCopyNonAtomic(pMessage, start, temp, (short)0, length);
223     if (paddingbytes != 0) {
224       for (short i = 0; i < paddingbytes; i++, length++) {
225         temp[length] = (byte)0xFF;
226       }
227     }
228     pMessage[1] = (byte)AESCipher.doFinal(temp, (short)0, length,
229                         pMessage, start);
230   }
231 }
```

C.8 Application Binding Protocol - Distributed

The Java Card implementation of the *ABPD* discussed in section 7.6 is listed in subsequent sections.

C.8.1 Client Application

Implementation of a client application that request for the application binding in the CDAM firewall mechanism is listed as below:

```
1  package protocolABPDClient;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.JCSystem;
8  import javacard.framework.Util;
9  import javacard.security.AESKey;
10 import javacard.security.Key;
11 import javacard.security.KeyBuilder;
12 import javacard.security.KeyPair;
13 import javacard.security.MessageDigest;
14 import javacard.security.RSAPrivateKey;
15 import javacard.security.RSAPublicKey;
16 import javacard.security.RandomData;
17 import javacard.security.Signature;
18 import javacardx.apdu.ExtendedLength;
19 import javacardx.crypto.Cipher;
20 public class ProtocolHandler extends Applet implements ExtendedLength
21   {
22   private byte[] ServerAppDHChanllengerArray;
23   private byte[] ServerAppRandomNumberArray;
24   private byte[] ServerAppCookieArray;
25   private byte[] ClientAppServerAppDHGeneratedValue;
26   private byte[] ClientAppRandomNumberArray;
27   private byte[] ClientAppUserCertificate;
28   private byte[] ClientAppCertificate;
29   private byte[] ServerAppDHChallengeTag = {
30     (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x01};
31   private byte[] MessageHandlerTagOne = {
32     (byte)0x1F, (byte)0xC0, (byte)0xAA, (byte)0xAA, (byte)0x00, (byte)
33      0x00, (byte)0x00};
34   private byte[] MessageHandlerTagTwo = {
35     (byte)0x1F, (byte)0xC0, (byte)0xBB, (byte)0xBB, (byte)0x00, (byte)
36      0x00, (byte)0x00};
37   private byte[] ServerAppIdentity = null;
38   private byte[] ServerAppRandomNumberTag = {
39     (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x01};
40   private byte[] ServerAppCookieTag = {
41     (byte)0x1F, (byte)0x5F, (byte)0x5B, (byte)0x01};
42   private byte[] EncryptedDataTag = {
43     (byte)0x1F, (byte)0xC0, (byte)0xFE, (byte)0x01};
44   private byte[] SignedDataTag = {
45     (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x02};
46   private byte[] MACedDataTag = {
```

```
47      (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x01};
48    private byte[] PlatformHash = {
49      (byte)0x1F, (byte)0x5F, (byte)0x5E, (byte)0xAF};
50    private byte[] ClientAppIdentityTag = {
51      (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x02, (byte)0x00, (byte)
52       0x12, (byte)0x7A, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6,
53       (byte)0xC1, (byte)0x22, (byte)0x07, (byte)0xC9, (byte)0xF9,
54       (byte)0x8D, (byte)0x11, (byte)0xED, (byte)0x34, (byte)0xDB,
55       (byte)0xF6, (byte)0x0B, (byte)0x2C};
56    private byte[] UserIdentity = {
57      (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x03, (byte)0x00, (byte)
58       0x14, (byte)0x7A, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6,
59       (byte)0xC1, (byte)0x22, (byte)0x07, (byte)0xC9, (byte)0xC9,
60       (byte)0x8D, (byte)0xD1, (byte)0xED, (byte)0xFC, (byte)0xDB,
61       (byte)0xF6, (byte)0x0B, (byte)0x2C, (byte)0x0B, (byte)0x2C};
62    private byte[] ExponentTag = {
63      (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x01};
64    private byte[] ModulusTag = {
65      (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x02};
66    private byte[] ClientAppDHChalleneTag = {
67      (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x02};
68    private byte[] ClientAppRandomNumberTag = {
69      (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x02};
70    private byte[] ServerAppCertificateTag = {
71      (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x01};
72    private byte[] ClientAppCertificateTag = {
73      (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x02};
74    private byte[] ClientAppUserCertificateTag = {
75      (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x03};
76    short PTLVDataOffset = (short)6;
77    short CTLVDataOffset = (short)7;
78    short TLVLengthOffset = (short)4;
79    byte[] ClientAppDHData;
80    final static byte CLA = (byte)0xB0;
81    final static byte StartProtocol = (byte)0x40;
82    final static byte InitiationProtocol = (byte)0xff;
83    final static short SW_CLASSNOTSUPPORTED = 0x6320;
84    final static short SW_ERROR_INS = 0x6300;
85    RandomData randomDataGen;
86    Cipher pkCipher;
87    short messageNumber = 0;
88    byte[] receivingBuffer = null;
89    short bytesLeft = 0;
90    short readCount = 0;
91    short rCount = 0;
92    short signlength = 0;
93    private RSAPublicKey dhKey = (RSAPublicKey)KeyBuilder.buildKey
94                                  (KeyBuilder.TYPE_RSA_PUBLIC,
95                                  KeyBuilder.LENGTH_RSA_2048, false);
96    private byte[] randomExponent;
97    final static byte GEN_KEYCONTRIBUTION = 0x01;
```

```
98  final static byte GEN_DHKEY = 0x02;
99  AESKey phCipherKey;
100 Cipher syCipher;
101 byte[] InitialisationVector = {
102   (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89, (byte)
103    0x99, (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1,
104    (byte)0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52};
105 AESKey phMacGeneratorKey;
106 Signature phMacGenerator;
107 Signature phSign;
108 KeyPair phClientAppKeyPair;
109 KeyPair phUserKeyPair;
110 RSAPublicKey ServerAppVerificationKey = null;
111 private ProtocolHandler() {
112   phMacGeneratorKey = (AESKey)KeyBuilder.buildKey
113                       (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
114                       KeyBuilder.LENGTH_AES_128, false);
115   phMacGenerator = Signature.getInstance
116     (Signature.ALG_AES_MAC_128_NOPAD, false);
117   phSign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false)
118                                  ;
119   phClientAppKeyPair = new KeyPair(KeyPair.ALG_RSA,
120                                    KeyBuilder.LENGTH_RSA_512);
121   phUserKeyPair = new KeyPair(KeyPair.ALG_RSA,
122                                    KeyBuilder.LENGTH_RSA_512);
123   phCipherKey = (AESKey)KeyBuilder.buildKey
124                  (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
125                  KeyBuilder.LENGTH_AES_128, false);
126   syCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
127                                 false);
128   randomDataGen = RandomData.getInstance
129     (RandomData.ALG_SECURE_RANDOM);
130   pkCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
131   dhInitialisation();
132   phClientAppKeyPair.genKeyPair();
133   phUserKeyPair.genKeyPair();
134 }
135 public static void install(byte bArray[], short bOffset, byte
136                            bLength)throws ISOException {
137   new ProtocolHandler().register();
138 }
139 public void initialiseProtocol() {
140   short initialPointer = 0;
141   ClientAppDHData = JCSystem.makeTransientByteArray((short)((short)
142     this.ClassDH.dhModulus.length + PTLVDataOffset),
143     JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
144   Util.arrayCopyNonAtomic(this.ClientAppDHChalleneTag, (short)
145                           initialPointer, this.ClientAppDHData,
                                (short)0,
146                           (short)this.ClientAppDHChalleneTag.length);
147   this.shortToBytes(ClientAppDHData, (short)4, (short)((short)
```

```
148                         ClientAppDHData.length - (short)PTLVDataOffset));
149     this.dhKeyConGen(this.ClientAppDHData, this.PTLVDataOffset,
150                         ProtocolHandler.GEN_KEYCONTRIBUTION);
151     ServerAppDHChanllengerArray = JCSystem.makeTransientByteArray((short)(
152       (short)this.ClassDH.dhModulus.length + this.PTLVDataOffset),
153       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
154     ServerAppRandomNumberArray =
            JCSystem.makeTransientByteArray((short)22,
155       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
156     ServerAppCookieArray = JCSystem.makeTransientByteArray((short)22,
157       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
158     ClientAppRandomNumberArray =
            JCSystem.makeTransientByteArray((short)22,
159       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
160     Util.arrayCopyNonAtomic(this.ClientAppRandomNumberTag, (short)
161                               initialPointer,
                                      this.ClientAppRandomNumberArray,
162                               (short)initialPointer, (short)
163                               this.ClientAppRandomNumberTag.length);
164     this.shortToBytes(this.ClientAppRandomNumberArray, (short)4, (short)(
165                         (short)this.ClientAppRandomNumberArray.length -
166                         (short)PTLVDataOffset));
167     try {
168       this.ClientAppUserCertificate = JCSystem.makeTransientByteArray(
169         (short)86, JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
170       initialPointer = Util.arrayCopyNonAtomic
171                               (this.ClientAppUserCertificateTag, (short)0,
172                               this.ClientAppUserCertificate, (short)0,
                                      (short)
173                               this.ClientAppUserCertificateTag.length);
174       this.shortToBytes(this.ClientAppUserCertificate, (short)4, (short)
175                           (this.ClientAppUserCertificate.length -
                                (short)7));
176       initialPointer = Util.arrayCopyNonAtomic(this.ExponentTag,
177         (short)0, this.ClientAppUserCertificate, (short)(initialPointer +
178         (short)3), (short)this.ExponentTag.length);
179       RSAPublicKey myPublic = (RSAPublicKey)
180                               this.phUserKeyPair.getPublic();
181       short kLen = myPublic.getExponent(this.ClientAppUserCertificate,
182         (short)(initialPointer + (short)2));
183       this.shortToBytes(this.ClientAppUserCertificate, initialPointer,
              kLen)
184                         ;
185       initialPointer += (short)(kLen + (short)2);
186       this.ClientAppUserCertificate[6]++;
187       initialPointer = Util.arrayCopyNonAtomic(this.ModulusTag,
188         (short)0, this.ClientAppUserCertificate, (short)(initialPointer),
189         (short)this.ModulusTag.length);
190       kLen = myPublic.getModulus(this.ClientAppUserCertificate, (short)
191                               (initialPointer + (short)2));
```

```
192         this.shortToBytes(this.ClientAppUserCertificate, initialPointer,
                kLen)
193                                 ;
194         this.ClientAppUserCertificate[6]++;
195         this.ServerAppIdentity = JCSystem.makeTransientByteArray((short)24,
196           JCSystem.MEMORY_TYPE_TRANSIENT_RESET);
197         ServerAppVerificationKey = (RSAPublicKey)KeyBuilder.buildKey
198                                     (KeyBuilder.TYPE_RSA_PUBLIC,
199                                     KeyBuilder.LENGTH_RSA_512, false);
200       } catch (Exception cE) {
201         ISOException.throwIt((short)0xCCCC);
202       }
203       try {
204         this.ClientAppCertificate =
                JCSystem.makeTransientByteArray((short)86,
205           JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
206         initialPointer =
                Util.arrayCopyNonAtomic(this.ClientAppCertificateTag,
207           (short)0, this.ClientAppCertificate, (short)0, (short)
208           this.ClientAppCertificateTag.length);
209         this.shortToBytes(this.ClientAppCertificate, (short)4, (short)
210                             (this.ClientAppCertificate.length - (short)7));
211         initialPointer = Util.arrayCopyNonAtomic(this.ExponentTag,
212           (short)0, this.ClientAppCertificate, (short)(initialPointer +
213           (short)3), (short)this.ExponentTag.length);
214         RSAPublicKey myPublic = (RSAPublicKey)
215                                     this.phClientAppKeyPair.getPublic();
216         short kLen = myPublic.getExponent(this.ClientAppCertificate, (short)
217           (initialPointer + (short)2));
218         this.shortToBytes(this.ClientAppCertificate, initialPointer, kLen);
219         initialPointer += (short)(kLen + (short)2);
220         this.ClientAppCertificate[6]++;
221         initialPointer = Util.arrayCopyNonAtomic(this.ModulusTag,
222           (short)0, this.ClientAppCertificate, (short)(initialPointer),
223           (short)this.ModulusTag.length);
224         kLen = myPublic.getModulus(this.ClientAppCertificate, (short)
225                                     (initialPointer + (short)2));
226         this.shortToBytes(this.ClientAppCertificate, initialPointer, kLen);
227         this.ClientAppCertificate[6]++;
228       } catch (Exception cE) {
229         ISOException.throwIt((short)0x6666);
230       }
231     }
232     public void process(APDU apdu)throws ISOException {
233       byte[] apduBuffer = apdu.getBuffer();
234       if (selectingApplet()) {
235         return ;
236       }
237       if (apduBuffer[ISO7816.OFFSET_CLA] != CLA) {
238         ISOException.throwIt(SW_CLASSNOTSUPPORTED);
239       }
```

```
240     if (apduBuffer[ISO7816.OFFSET_INS] == InitiationProtocol) {
241       this.initialiseProtocol();
242       return ;
243     }
244     receivingBuffer = null;
245     bytesLeft = 0;
246     bytesLeft = apdu.getIncomingLength();
247     receivingBuffer = JCSystem.makeTransientByteArray(bytesLeft,
248       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
249     readCount = (short)((short)apdu.setIncomingAndReceive());
250     rCount = 0;
251     if (bytesLeft > 0) {
252       rCount = Util.arrayCopyNonAtomic(apduBuffer,
253         ISO7816.OFFSET_EXT_CDATA, receivingBuffer, rCount, readCount);
254       bytesLeft -= readCount;
255     }
256     while (bytesLeft > 0) {
257       try {
258         readCount = apdu.receiveBytes((short)0);
259         rCount = Util.arrayCopyNonAtomic(apduBuffer, (short)0,
260           receivingBuffer, rCount, readCount);
261         bytesLeft -= readCount;
262       } catch (Exception aE) {
263         ISOException.throwIt((short)0x7AAA);
264       }
265     }
266     try {
267       parseMessage(receivingBuffer);
268     } catch (Exception cE) {
269       ISOException.throwIt((short)0xA112);
270     }
271     if (this.receivingBuffer[3] == this.MessageHandlerTagOne[3]) {
272       receivingBuffer = JCSystem.makeTransientByteArray((short)568,
273         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
274       generateResponse((short)1);
275     } else if (this.receivingBuffer[3] ==
276               this.MessageHandlerTagTwo[3]) {
277       processSecondMsg(receivingBuffer);
278       receivingBuffer = JCSystem.makeTransientByteArray((short)568,
279         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
280       generateResponse((short)2);
281     } else {
282       ISOException.throwIt(ProtocolHandler.SW_ERROR_INS);
283     }
284     JCSystem.requestObjectDeletion();
285     apdu.setOutgoing();
286     apdu.setOutgoingLength((short)copyPointer);
287     apdu.sendBytesLong(receivingBuffer, (short)0, (short)copyPointer);
288     JCSystem.requestObjectDeletion();
289   }
290   private void generateResponse(short msgNumber) {
```

```
291     short childPointerMessage = 6;
292     short encryptionOffset = 0;
293     copyPointer = 0;
294     if (msgNumber == 1) {
295       randomDataGen.generateData(this.ClientAppRandomNumberArray,
296                                  this.PTLVDataOffset, (short)16);
297       this.dhKeyConGen(this.ServerAppDHChanllengerArray,
298                        this.PTLVDataOffset, ProtocolHandler.GEN_DHKEY)
299                        ;
300       copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagOne,
301         (short)0, this.receivingBuffer, copyPointer, (short)
302         this.MessageHandlerTagOne.length);
303       copyPointer = Util.arrayCopyNonAtomic(this.ClientAppDHData,
            (short)0,
304         this.receivingBuffer, copyPointer, (short)
305         this.ClientAppDHData.length);
306       this.receivingBuffer[childPointerMessage]++;
307       copyPointer =
              Util.arrayCopyNonAtomic(this.ClientAppRandomNumberArray,
308         (short)0, this.receivingBuffer, copyPointer, (short)
309         this.ClientAppRandomNumberArray.length);
310       this.receivingBuffer[childPointerMessage]++;
311       keygenerator();
312       copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag,
313         (short)0, this.receivingBuffer, copyPointer, (short)
314         this.EncryptedDataTag.length);
315       this.receivingBuffer[childPointerMessage]++;
316       short childEnMessage = (short)(copyPointer + (short)2);
317       copyPointer += (short)3;
318       encryptionOffset = copyPointer;
319       copyPointer = Util.arrayCopyNonAtomic(this.ClientAppIdentityTag,
320         (short)0, this.receivingBuffer, copyPointer, (short)
321         this.ClientAppIdentityTag.length);
322       this.receivingBuffer[childEnMessage]++;
323       copyPointer =
              Util.arrayCopyNonAtomic(this.ClientAppRandomNumberArray,
324         (short)0, this.receivingBuffer, copyPointer, (short)
325         this.ClientAppRandomNumberArray.length);
326       this.receivingBuffer[childEnMessage]++;
327       copyPointer =
              Util.arrayCopyNonAtomic(this.ServerAppRandomNumberArray,
328         (short)0, this.receivingBuffer, copyPointer, (short)
329         this.ServerAppRandomNumberArray.length);
330       this.receivingBuffer[childEnMessage]++;
331       this.signGenerate(this.receivingBuffer, encryptionOffset,
332                         (short)(copyPointer - encryptionOffset),
333                         phUserKeyPair.getPrivate(),
334                         Signature.MODE_SIGN);
335       this.receivingBuffer[childEnMessage]++;
336       copyPointer = Util.arrayCopyNonAtomic(this.ClientAppUserCertificate,
337         (short)0, this.receivingBuffer, copyPointer, (short)
```

```
338           this.ClientAppUserCertificate.length);
339         this.receivingBuffer[childEnMessage]++;
340         messageEncryption(this.receivingBuffer, encryptionOffset,
341                           (short)(copyPointer - encryptionOffset));
342         this.shortToBytes(receivingBuffer, (short)(encryptionOffset -
343                           (short)3), (short)(copyPointer -
344                           encryptionOffset));
345         macGenerate(this.receivingBuffer, encryptionOffset, (short)
346                     (copyPointer - encryptionOffset),
347                     Signature.MODE_SIGN);
348         this.receivingBuffer[childPointerMessage]++;
349         copyPointer = Util.arrayCopyNonAtomic(this.ServerAppCookieArray,
350           (short)0, this.receivingBuffer, copyPointer, (short)
351           this.ServerAppCookieArray.length);
352         this.receivingBuffer[childPointerMessage]++;
353         this.shortToBytes(this.receivingBuffer, (short)4, copyPointer);
354       } else if (msgNumber == 2) {
355         copyPointer = (short)0;
356         short tempLength = (short)0;
357         short mainChildPointer = (short)6;
358         short mainLengthPointer = (short)4;
359         short encryptedChildPointer = (short)13;
360         short generalLengthPointer = (short)0;
361         this.receivingBuffer[mainChildPointer] = (short)0;
362         this.receivingBuffer[encryptedChildPointer] = (short)0;
363         copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagTwo,
364           (short)0, this.receivingBuffer, copyPointer, (short)7);
365         this.receivingBuffer[mainChildPointer]++;
366         copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag,
367           (short)0, this.receivingBuffer, copyPointer, (short)4);
368         copyPointer += (short)3;
369         encryptionOffset = copyPointer;
370         copyPointer = Util.arrayCopyNonAtomic(this.PlatformHash, (short)
371           0, receivingBuffer, copyPointer, (short)4);
372         generalLengthPointer = copyPointer;
373         copyPointer += (short)2;
374         MessageDigest myHashGen = MessageDigest.getInstance
375           (MessageDigest.ALG_SHA_256, false);
376         tempLength = (short)myHashGen.doFinal(this.ClassDH.dhModulus,
              (short)0,
377                       (short)this.ClassDH.dhModulus.length, receivingBuffer,
378                       copyPointer);
379         this.receivingBuffer[encryptedChildPointer]++;
380         this.shortToBytes(this.receivingBuffer, generalLengthPointer,
381                           (short)(tempLength));
382         copyPointer += tempLength;
383         copyPointer = Util.arrayCopyNonAtomic(this.UserIdentity, (short)
384           0, this.receivingBuffer, copyPointer, (short)
385           this.UserIdentity.length);
386         this.receivingBuffer[encryptedChildPointer]++;
```

```
387         copyPointer = Util.arrayCopyNonAtomic(this.ServerAppIdentity,
                (short)
388           0, this.receivingBuffer, copyPointer, (short)
389           this.ServerAppIdentity.length);
390         this.receivingBuffer[encryptedChildPointer]++;
391         copyPointer =
                Util.arrayCopyNonAtomic(this.ClientAppRandomNumberArray,
392           (short)0, this.receivingBuffer, copyPointer, (short)
393           this.ClientAppRandomNumberArray.length);
394         this.receivingBuffer[encryptedChildPointer]++;
395         copyPointer =
                Util.arrayCopyNonAtomic(this.ServerAppRandomNumberArray,
396           (short)0, this.receivingBuffer, (short)copyPointer, (short)
397           this.ServerAppRandomNumberArray.length);
398         this.receivingBuffer[encryptedChildPointer]++;
399         try {
400           this.signGenerate(receivingBuffer, (short)(encryptionOffset),
401                             (short)(copyPointer - encryptionOffset),
402                             phClientAppKeyPair.getPrivate(),
403                             Signature.MODE_SIGN);
404           this.receivingBuffer[encryptedChildPointer]++;
405         } catch (Exception cE) {
406           ISOException.throwIt((short)0xFA17);
407         }
408         copyPointer = Util.arrayCopyNonAtomic(this.ClientAppCertificate,
409           (short)0, this.receivingBuffer, copyPointer, (short)
410           this.ClientAppCertificate.length);
411         this.receivingBuffer[encryptedChildPointer]++;
412         try {
413           this.messageEncryption(receivingBuffer, (short)
414                                  (encryptedChildPointer + (short)1),
415                                  (short)(copyPointer -
416                                  (encryptedChildPointer + (short)1)));
417         } catch (Exception cE) {
418           ISOException.throwIt((short)(copyPointer -
419                                encryptedChildPointer + (short)1));
420         }
421         this.shortToBytes(this.receivingBuffer, (short)
422                           (encryptedChildPointer - (short)2), (short)
423                           (copyPointer - (short)(encryptedChildPointer
424                           + (short)1)));
425         this.macGenerate(receivingBuffer, (short)(encryptedChildPointer
426                          + (short)1), (short)(copyPointer -
427                          (encryptedChildPointer + (short)1)),
428                          Signature.MODE_SIGN);
429         this.receivingBuffer[mainChildPointer]++;
430         copyPointer = Util.arrayCopyNonAtomic(this.ServerAppCookieArray,
431           (short)0, this.receivingBuffer, copyPointer, (short)
432           this.ServerAppCookieArray.length);
433         this.receivingBuffer[mainChildPointer]++;
434         this.shortToBytes(this.receivingBuffer, mainLengthPointer,
```

```
435                         (short)(copyPointer - (short)7));
436     }
437   }
438   void platformHashGeneration(byte[] inArray, short inOffset){}
439   void processSecondMsg(byte[] inArray) {
440     short inOffset = (short)(this.CTLVDataOffset +
441                         this.CTLVDataOffset);
442     short inLength = (short)(ProtocolHandler.bytesToShort(inArray,
443                         (short)(inOffset - (short)3)));
444     if (this.macGenerate(inArray, inOffset, inLength,
445         Signature.MODE_VERIFY)) {
446       this.phDecryption(inArray, inOffset, inLength);
447       Util.arrayCopyNonAtomic(inArray, inOffset, this.ServerAppIdentity,
448                               (short)0, (short)
449                               this.ServerAppIdentity.length);
450       inOffset += (short)151;
451       inLength = (short)3;
452       ServerAppVerificationKey.setExponent(inArray, inOffset, inLength);
453       inOffset += (short)(inLength + this.PTLVDataOffset);
454       inLength = (short)64;
455       ServerAppVerificationKey.setModulus(inArray, inOffset, inLength);
456       inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
457       inLength = (short)68;
458       if (this.signGenerate(inArray, inOffset, inLength,
459           ServerAppVerificationKey, Signature.MODE_VERIFY)) {
460         return;
461       } else {
462         ISOException.throwIt((short)0x6666);
463       }
464     } else {
465       ISOException.throwIt((short)0xFA18);
466     }
467   }
468   void parseMessage(byte[] inBuffer) {
469     byte childLeft = inBuffer[(short)(this.CTLVDataOffset - (short)1)
470                               ];
471     short pointer = (short)this.CTLVDataOffset;
472     try {
473       while (childLeft > 0) {
474         if (Util.arrayCompare(ServerAppDHChallengeTag, (short)0, inBuffer,
475             pointer, (short)4) == 0) {
476           Util.arrayCopy(inBuffer, pointer,
477                          this.ServerAppDHChanllengerArray, (short)0,
                               (short)
478                          this.ServerAppDHChanllengerArray.length);
479           pointer += (short)this.ServerAppDHChanllengerArray.length;
480         } else if (Util.arrayCompare(this.ServerAppRandomNumberTag,
              (short)
481                 0, inBuffer, pointer, (short)4) == 0) {
482           Util.arrayCopyNonAtomic(inBuffer, pointer,
```

```
483                                     this.ServerAppRandomNumberArray,
                                            (short)0,
484                                     (short)
485                                     (this.ServerAppRandomNumberArray.length));
486           pointer += (short)(this.ServerAppRandomNumberArray.length);
487         } else if (Util.arrayCompare(this.ServerAppCookieTag, (short)0,
488                     inBuffer, pointer, (short)4) == 0) {
489           Util.arrayCopyNonAtomic(inBuffer, pointer,
490                                     this.ServerAppCookieArray, (short)0,
491                                     (short)(this.ServerAppCookieArray.length))
492                                     ;
493           pointer += (short)(this.ServerAppCookieArray.length);
494         }
495         childLeft -= (short)1;
496       }
497     } catch (Exception cE) {
498       ISOException.throwIt((short)childLeft);
499     }
500   }
501   void protocolImplementation(){}
502   void dhInitialisation() {
503     dhKey.setModulus(ClassDH.dhModulus, (short)0,
          (short)ClassDH.dhModulus.length);
504   }
505   void dhKeyConGen(byte[] inbuff, short inbuffOffset, byte Oper_Mode)
506                     {
507     switch (Oper_Mode) {
508       case GEN_KEYCONTRIBUTION: randomExponent =
509         JCSystem.makeTransientByteArray((short)32,
510         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
511       randomDataGen.generateData(randomExponent, (short)0, (short)
512                                       randomExponent.length);
513       dhKey.setExponent(randomExponent, (short)0, (short)
514                           randomExponent.length);
515       pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
516       pkCipher.doFinal(ClassDH.dhBase, (short)0,
              (short)ClassDH.dhBase.length, inbuff,
517                           inbuffOffset);
518       break;
519       case GEN_DHKEY:
520         try {
521           dhKey.setExponent(randomExponent, (short)0, (short)
522                               randomExponent.length);
523           pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
524           ClientAppServerAppDHGeneratedValue =
                  JCSystem.makeTransientByteArray(
525             (short)ClassDH.dhModulus.length,
526             JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
527           pkCipher.doFinal(inbuff, inbuffOffset, (short)((short)
528                               inbuff.length - (short)this.PTLVDataOffset)
```

```
529                                 , ClientAppServerAppDHGeneratedValue,
                                      (short)0);
530           }
531            catch (Exception cE) {
532             ISOException.throwIt((short)0xD86E);
533           }
534           break;
535         default:
536           ISOException.throwIt((short)0x5FA1);
537       }
538     }
539     void keygenerator() {
540       AESKey sessionGenKey = (AESKey)KeyBuilder.buildKey
541                                 (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
542                                 KeyBuilder.LENGTH_AES_128, false);
543       sessionGenKey.setKey(ClientAppServerAppDHGeneratedValue, (short)0);
544       phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
545                           InitialisationVector, (short)0, (short)
546                           InitialisationVector.length);
547       byte[] keyGenMacData = JCSystem.makeTransientByteArray((short)64,
548         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
549       short pointer = 0;
550       pointer = Util.arrayCopyNonAtomic(this.ServerAppRandomNumberArray,
551         this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
552       pointer = Util.arrayCopyNonAtomic(this.ClientAppRandomNumberArray,
553         this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
554       pointer = Util.arrayCopyNonAtomic(ClientAppServerAppDHGeneratedValue,
            (short)
555         16, keyGenMacData, (short)pointer, (short)16);
556       for (short i = 48; i < 64; i++) {
557         keyGenMacData[i] = (byte)0x02;
558       }
559       phMacGenerator.sign(keyGenMacData, (short)0, (short)
560                           keyGenMacData.length,
                                ClientAppServerAppDHGeneratedValue,
561                           (short)0);
562       this.phCipherKey.setKey(ClientAppServerAppDHGeneratedValue, (short)0);
563       for (short i = 48; i < 64; i++) {
564         keyGenMacData[i] = (byte)0x03;
565       }
566       phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
567                           InitialisationVector, (short)0, (short)
568                           InitialisationVector.length);
569       phMacGenerator.sign(keyGenMacData, (short)0, (short)
570                           keyGenMacData.length,
                                ClientAppServerAppDHGeneratedValue,
571                           (short)0);
572       this.phMacGeneratorKey.setKey(ClientAppServerAppDHGeneratedValue,
            (short)0);
573       ClientAppServerAppDHGeneratedValue = null;
574       JCSystem.requestObjectDeletion();
```

```
575   }
576   void messageEncryption(byte[] inbuff, short inbuffOffset, short
577                          inbuffLength) {
578     syCipher.init(phCipherKey, Cipher.MODE_ENCRYPT,
579                   InitialisationVector, (short)0, (short)
580                   InitialisationVector.length);
581     short temp;
582     this.shortToBytes(inbuff, (short)(inbuffOffset - 3), temp =
583                       (short)syCipher.doFinal(inbuff, inbuffOffset,
584                       inbuffLength, inbuff, inbuffOffset));
585   }
586   void phDecryption(byte[] inbuff, short inbuffOffset, short
587                     inbuffLength) {
588     syCipher.init(phCipherKey, Cipher.MODE_DECRYPT,
589                   InitialisationVector, (short)0, (short)
590                   InitialisationVector.length);
591     syCipher.doFinal(inbuff, inbuffOffset, inbuffLength, inbuff,
592                      inbuffOffset);
593   }
594   boolean macGenerate(byte[] inbuff, short inbuffOffset, short
595                       inbuffLength, short macMode) {
596     if (macMode == Signature.MODE_SIGN) {
597       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
598                           InitialisationVector, (short)0, (short)
599                           InitialisationVector.length);
600       try {
601         copyPointer = Util.arrayCopyNonAtomic(this.MACedDataTag,
602           (short)0, this.receivingBuffer, copyPointer, (short)
603           this.MACedDataTag.length);
604         copyPointer += 2;
605       } catch (Exception ce) {
606         ISOException.throwIt((short)0xFA17);
607       }
608       try {
609         short length = (short)phMacGenerator.sign
610                        (this.receivingBuffer, inbuffOffset,
611                        inbuffLength, inbuff, copyPointer);
612         this.shortToBytes(inbuff, (short)(copyPointer - (short)2),
613                           length);
614         copyPointer += length;
615       } catch (Exception ce) {
616         ISOException.throwIt((short)0x0987);
617       }
618       return true;
619     } else if (macMode == Signature.MODE_VERIFY) {
620       try {
621         phMacGenerator.init(phMacGeneratorKey, Signature.MODE_VERIFY,
622                             InitialisationVector, (short)0, (short)
623                             InitialisationVector.length);
624         return phMacGenerator.verify(this.receivingBuffer,
625           inbuffOffset, inbuffLength, inbuff, (short)(inbuffOffset +
```

```
626             inbuffLength + this.PTLVDataOffset), (short)16);
627         } catch (Exception cE) {
628           ISOException.throwIt((short)0xC1C2);
629         }
630       }
631       return false;
632     }
633     boolean signGenerate(byte[] inbuff, short inbuffOffset, short
634                          inbufflength, Key kpSign, short signMode) {
635       if (signMode == Signature.MODE_SIGN) {
636         copyPointer = Util.arrayCopyNonAtomic(this.SignedDataTag,
637           (short)0, this.receivingBuffer, copyPointer, (short)
638           this.SignedDataTag.length);
639         copyPointer += (short)2;
640         phSign.init((RSAPrivateKey)kpSign, Signature.MODE_SIGN);
641         signlength = phSign.sign(inbuff, (short)inbuffOffset,
642                                  inbufflength, inbuff, copyPointer);
643         this.shortToBytes(this.receivingBuffer, (short)(copyPointer -
644                           (short)2), signlength);
645         copyPointer += signlength;
646         return true;
647       } else if (signMode == Signature.MODE_VERIFY) {
648         phSign.init((RSAPublicKey)kpSign, Signature.MODE_VERIFY);
649         return phSign.verify(inbuff, inbuffOffset, inbufflength, inbuff,
650                              (short)(inbuffOffset + inbufflength +
651                              this.PTLVDataOffset), (short)64);
652       }
653       return false;
654     }
655     public static short bytesToShort(byte[] ArrayBytes) {
656       return (short)(((ArrayBytes[0] << 8)) | ((ArrayBytes[1] & 0xff)));
657     }
658     public static short bytesToShort(byte[] ArrayBytes, short
659                                      arrayOffset) {
660       return (short)(((ArrayBytes[arrayOffset] << 8)) | ((ArrayBytes[
661               (short)(arrayOffset + (short)1)] & 0xff)));
662     }
663     private void shortToBytes(byte[] Array, short inShort) {
664       Array[0] = (byte)((short)(inShort & (short)0xFF00) >> (short)
665                   0x0008);
666       Array[1] = (byte)(inShort & (short)0x00FF);
667     }
668     private void shortToBytes(byte[] Array, short arrayOffset, short
669                               inShort) {
670       Array[arrayOffset] = (byte)((short)(inShort & (short)0xFF00) >>
671                           (short)0x0008);
672       Array[(short)(arrayOffset + (short)1)] = (byte)(inShort & (short)
673             0x00FF);
674     }
675 }
```

C.8.2 Server Application

Implementation of a server application that responds to the application binding request in the CDAM firewall mechanism is listed as below:

```
1  package protocolABPDServerApp;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.JCSystem;
8  import javacard.framework.Util;
9  import javacard.security.AESKey;
10 import javacard.security.Key;
11 import javacard.security.KeyBuilder;
12 import javacard.security.KeyPair;
13 import javacard.security.MessageDigest;
14 import javacard.security.RSAPrivateKey;
15 import javacard.security.RSAPublicKey;
16 import javacard.security.RandomData;
17 import javacard.security.Signature;
18 import javacardx.apdu.ExtendedLength;
19 import javacardx.crypto.Cipher;
20 public class ProtocolHandler extends Applet implements ExtendedLength
21   {
22   private byte[] ClientAppRandomNumberArray;
23   private byte[] ClientAppCookieArray;
24   private byte[] ServerAppClientAppDHGeneratedValue;
25   private byte[] ServerAppRandomNumberArray;
26   private byte[] ServerAppCertificate;
27   private byte[] ClientAppDHChallengeTag = {
28     (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x01};
29   private byte[] MessageHandlerTagOne = {
30     (byte)0x1F, (byte)0xC0, (byte)0xAA, (byte)0xAA, (byte)0x00, (byte)
31      0x00, (byte)0x00};
32   private byte[] MessageHandlerTagTwo = {
33     (byte)0x1F, (byte)0xC0, (byte)0xBB, (byte)0xBB, (byte)0x00, (byte)
34      0x00, (byte)0x00};
35   private byte[] ClientAppIdentity = null;
36   private byte[] ClientAppRandomNumberTag = {
37     (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x01};
38   private byte[] ClientAppCookieTag = {
39     (byte)0x1F, (byte)0x5F, (byte)0x5B, (byte)0x01};
40   private byte[] EncryptedDataTag = {
41     (byte)0x1F, (byte)0xC0, (byte)0xFE, (byte)0x01};
42   private byte[] SignedDataTag = {
43     (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x02};
44   private byte[] MACedDataTag = {
45     (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x01};
46   private byte[] PlatformHash = {
```

```
47      (byte)0x1F, (byte)0x5F, (byte)0x5E, (byte)0xAF};
48    private byte[] ServerAppIdentityTag = {
49      (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x02, (byte)0x00, (byte)
50       0x0C, (byte)0x7A, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6,
51       (byte)0xC1, (byte)0x22, (byte)0x07, (byte)0xC9, (byte)0xF9,
52       (byte)0x8D, (byte)0x11};
53    private byte[] ExponentTag = {
54      (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x01};
55    private byte[] ModulusTag = {
56      (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x02};
57    private byte[] ServerAppDHChalleneTag = {
58      (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x02};
59    private byte[] ServerAppRandomNumberTag = {
60      (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x02};
61    private byte[] ClientAppCertificateTag = {
62      (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x01};
63    private byte[] ServerAppCertificateTag = {
64      (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x02};
65    private byte[] ServerAppProtocolInitiatorTag = {
66      (byte)0x1F, (byte)0x5F, (byte)0xA1, (byte)0xB2};
67    short PTLVDataOffset = (short)6;
68    short CTLVDataOffset = (short)7;
69    short TLVLengthOffset = (short)4;
70    short copyPointer = (short)0;
71    byte[] ServerAppDHData;
72    final static byte CLA = (byte)0xB0;
73    final static byte StartProtocol = (byte)0x40;
74    final static byte InitiationProtocol = (byte)0xff;
75    final static short SW_CLASSNOTSUPPORTED = 0x6320;
76    final static short SW_ERROR_INS = 0x6300;
77    RandomData randomDataGen;
78    Cipher pkCipher;
79    short messageNumber = 0;
80    byte[] receivingBuffer = null;
81    short bytesLeft = 0;
82    short readCount = 0;
83    short rCount = 0;
84    short signlength = 0;
85    private RSAPublicKey dhKey = (RSAPublicKey)KeyBuilder.buildKey
86                                  (KeyBuilder.TYPE_RSA_PUBLIC,
87                                  KeyBuilder.LENGTH_RSA_2048, false);
88    private byte[] randomExponent;
89    final static byte GEN_KEYCONTRIBUTION = 0x01;
90    final static byte GEN_DHKEY = 0x02;
91    AESKey phCipherKey;
92    Cipher syCipher;
93    byte[] InitialisationVector = {
94      (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89, (byte)
95       0x99, (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1,
96       (byte)0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52};
97    AESKey phMacGeneratorKey;
```

```
98  Signature phMacGenerator;
99  Signature phSign;
100 KeyPair phServerAppKeyPair;
101 KeyPair phUserKeyPair;
102 RSAPublicKey ClientAppVerificationKey = null;
103 private ProtocolHandler() {
104   phMacGeneratorKey = (AESKey)KeyBuilder.buildKey
105                       (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
106                       KeyBuilder.LENGTH_AES_128, false);
107   phMacGenerator = Signature.getInstance
108     (Signature.ALG_AES_MAC_128_NOPAD, false);
109   phSign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false)
110                                         ;
111   phServerAppKeyPair = new KeyPair(KeyPair.ALG_RSA,
112                                    KeyBuilder.LENGTH_RSA_512);
113   phUserKeyPair = new KeyPair(KeyPair.ALG_RSA,
114                                  KeyBuilder.LENGTH_RSA_512);
115   phCipherKey = (AESKey)KeyBuilder.buildKey
116                 (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
117                 KeyBuilder.LENGTH_AES_128, false);
118   syCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
119                                    false);
120   randomDataGen = RandomData.getInstance
121     (RandomData.ALG_SECURE_RANDOM);
122   pkCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
123   dhInitialisation();
124   phServerAppKeyPair.genKeyPair();
125   phUserKeyPair.genKeyPair();
126 }
127 public static void install(byte bArray[], short bOffset, byte
128                            bLength)throws ISOException {
129   new ProtocolHandler().register();
130 }
131 public void initialiseProtocol() {
132   short initialPointer = 0;
133   ServerAppDHData = JCSystem.makeTransientByteArray((short)((short)
134     this.ClassDH.dhModulus.length + PTLVDataOffset),
135     JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
136   Util.arrayCopyNonAtomic(this.ServerAppDHChalleneTag, (short)
137                           initialPointer, this.ServerAppDHData,
                                (short)0,
138                           (short)this.ServerAppDHChalleneTag.length);
139   this.shortToBytes(ServerAppDHData, (short)4, (short)((short)
140                     ServerAppDHData.length - (short)PTLVDataOffset));
141   this.dhKeyConGen(this.ServerAppDHData, this.PTLVDataOffset,
142                    ProtocolHandler.GEN_KEYCONTRIBUTION);
143   ClientAppDHChanllengerArray = JCSystem.makeTransientByteArray((short)(
144     (short)this.ClassDH.dhModulus.length + this.PTLVDataOffset),
145     JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
146   ClientAppRandomNumberArray =
        JCSystem.makeTransientByteArray((short)22,
```

```
147         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
148       ClientAppCookieArray = JCSystem.makeTransientByteArray((short)22,
149         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
150       ServerAppRandomNumberArray =
              JCSystem.makeTransientByteArray((short)22,
151         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
152       Util.arrayCopyNonAtomic(this.ServerAppRandomNumberTag, (short)
153                               initialPointer,
                                    this.ServerAppRandomNumberArray,
154                               (short)initialPointer, (short)
155                               this.ServerAppRandomNumberTag.length);
156       this.shortToBytes(this.ServerAppRandomNumberArray, (short)4, (short)(
157                         (short)this.ServerAppRandomNumberArray.length -
                              (short)
158                         PTLVDataOffset));
159       try {
160         this.ServerAppCertificate =
                JCSystem.makeTransientByteArray((short)86,
161           JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
162         initialPointer =
              Util.arrayCopyNonAtomic(this.ServerAppCertificateTag,
163                                 (short)0, this.ServerAppCertificate,
                                      (short)0, (short)
164                                 this.ServerAppCertificateTag.length);
165         this.shortToBytes(this.ServerAppCertificate, (short)4, (short)
166                           (this.ServerAppCertificate.length - (short)7));
167         initialPointer = Util.arrayCopyNonAtomic(this.ExponentTag,
168                                 (short)0, this.ServerAppCertificate,
                                      (short)(initialPointer + (short)
169                                 3), (short)this.ExponentTag.length);
170         RSAPublicKey myPublic = (RSAPublicKey)
171                                 this.phServerAppKeyPair.getPublic();
172         short kLen = myPublic.getExponent(this.ServerAppCertificate, (short)
173           (initialPointer + (short)2));
174         this.shortToBytes(this.ServerAppCertificate, initialPointer, kLen);
175         initialPointer += (short)(kLen + (short)2);
176         this.ServerAppCertificate[6]++;
177         initialPointer = Util.arrayCopyNonAtomic(this.ModulusTag,
178                                 (short)0, this.ServerAppCertificate,
                                      (short)(initialPointer), (short)
179                                 this.ModulusTag.length);
180         kLen = myPublic.getModulus(this.ServerAppCertificate, (short)
181                                    (initialPointer + (short)2));
182         this.shortToBytes(this.ServerAppCertificate, initialPointer, kLen);
183         this.ServerAppCertificate[6]++;
184         ClientAppVerificationKey = (RSAPublicKey)KeyBuilder.buildKey
185                               (KeyBuilder.TYPE_RSA_PUBLIC,
186                               KeyBuilder.LENGTH_RSA_512, false);
187       } catch (Exception cE) {
188         ISOException.throwIt((short)0x6666);
189       }
```

```
190   }
191   public void process(APDU apdu)throws ISOException {
192     byte[] apduBuffer = apdu.getBuffer();
193     if (selectingApplet()) {
194       this.initialiseProtocol();
195       return ;
196     }
197     if (apduBuffer[ISO7816.OFFSET_CLA] != CLA) {
198       ISOException.throwIt(SW_CLASSNOTSUPPORTED);
199     }
200     if (apduBuffer[ISO7816.OFFSET_INS] == InitiationProtocol) {
201       receivingBuffer = JCSystem.makeTransientByteArray((short)64,
202         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
203       generateResponse((short)1);
204       apdu.setOutgoing();
205       apdu.setOutgoingLength((short)copyPointer);
206       apdu.sendBytesLong(receivingBuffer, (short)0, (short)
207                          copyPointer);
208       return ;
209     }
210     receivingBuffer = null;
211     bytesLeft = 0;
212     bytesLeft = apdu.getIncomingLength();
213     receivingBuffer = JCSystem.makeTransientByteArray(bytesLeft,
214       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
215     readCount = (short)((short)apdu.setIncomingAndReceive());
216     rCount = 0;
217     if (bytesLeft > 0) {
218       rCount = Util.arrayCopyNonAtomic(apduBuffer,
219         ISO7816.OFFSET_EXT_CDATA, receivingBuffer, rCount, readCount);
220       bytesLeft -= readCount;
221     }
222     while (bytesLeft > 0) {
223       try {
224         readCount = apdu.receiveBytes((short)0);
225         rCount = Util.arrayCopyNonAtomic(apduBuffer, (short)0,
226           receivingBuffer, rCount, readCount);
227         bytesLeft -= readCount;
228       } catch (Exception aE) {
229         ISOException.throwIt((short)0x7AAA);
230       }
231     }
232     if (this.receivingBuffer[3] == this.MessageHandlerTagOne[3]) {
233       try {
234         parseMessage(receivingBuffer);
235       } catch (Exception cE) {
236         ISOException.throwIt((short)0xA112);
237       }
238       receivingBuffer = JCSystem.makeTransientByteArray((short)600,
239         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
240       generateResponse((short)2);
```

```
241       JCSystem.requestObjectDeletion();
242       apdu.setOutgoing();
243       apdu.setOutgoingLength((short)copyPointer);
244       apdu.sendBytesLong(receivingBuffer, (short)0, (short)
245                          copyPointer);
246     } else if (this.receivingBuffer[3] ==
247                this.MessageHandlerTagTwo[3]) {
248       if (processSecondMsg(receivingBuffer)) {
249         return;
250       } else {
251         ISOException.throwIt((short)0xFA17);
252       }
253       return;
254     } else {
255       ISOException.throwIt(ProtocolHandler.SW_ERROR_INS);
256     }
257     JCSystem.requestObjectDeletion();
258   }
259   private void generateResponse(short msgNumber) {
260     short childPM1 = 0;
261     short childPM2 = 0;
262     copyPointer = 0;
263     if (msgNumber == 1) {
264       copyPointer = Util.arrayCopy(this.ServerAppProtocolInitiatorTag,
265                                    (short)0, this.receivingBuffer,
266                                    copyPointer, (short)
267                                    this.ServerAppProtocolInitiatorTag.length)
268                                    ;
269       randomDataGen.generateData(this.ServerAppRandomNumberArray,
270                                  this.PTLVDataOffset, (short)16);
271       childPM1 = copyPointer;
272       copyPointer += 2;
273       phMacGeneratorKey.setKey(this.ServerAppRandomNumberArray,
274                                this.PTLVDataOffset);
275       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
276                           InitialisationVector, (short)0, (short)
277                           InitialisationVector.length);
278       short length = 0;
279       length = phMacGenerator.sign(ServerAppDHData, (short)
280                                    this.PTLVDataOffset, (short)
281                                    (ServerAppDHData.length -
282                                    this.PTLVDataOffset),
283                                    this.receivingBuffer, copyPointer);
284       copyPointer += length;
285       this.shortToBytes(this.receivingBuffer, childPM1, length);
286       return;
287     } else if (msgNumber == 2) {
288       this.dhKeyConGen(this.ClientAppDHChanllengerArray,
289           this.PTLVDataOffset,
290                        ProtocolHandler.GEN_DHKEY);
291       keygenerator();
```

```
291         childPM1 = (short)6;
292         copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagTwo,
293           (short)0, this.receivingBuffer, copyPointer, (short)
294           this.MessageHandlerTagTwo.length);
295         copyPointer = Util.arrayCopyNonAtomic(this.ServerAppDHData,
                (short)0,
296                             this.receivingBuffer, (short)copyPointer, (short)
297                             this.ServerAppDHData.length);
298         this.receivingBuffer[childPM1]++;
299         copyPointer =
                Util.arrayCopyNonAtomic(this.ServerAppRandomNumberArray,
300           (short)0, this.receivingBuffer, copyPointer, (short)
301           this.ServerAppRandomNumberArray.length);
302         this.receivingBuffer[childPM1]++;
303         copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag,
304           (short)0, this.receivingBuffer, copyPointer, (short)
305           this.EncryptedDataTag.length);
306         copyPointer += 3;
307         childPM2 = (short)(copyPointer - (short)1);
308         this.receivingBuffer[childPM1]++;
309         copyPointer = Util.arrayCopyNonAtomic(this.PlatformHash, (short)
310           0, this.receivingBuffer, copyPointer, (short)
311           this.PlatformHash.length);
312         copyPointer += 2;
313         MessageDigest myHashGen = MessageDigest.getInstance
314           (MessageDigest.ALG_SHA_256, false);
315         short tempLength = (short)myHashGen.doFinal(this.ClassDH.dhModulus,
316                             (short)0, (short)this.ClassDH.dhModulus.length,
317                             receivingBuffer, copyPointer);
318         this.receivingBuffer[childPM2]++;
319         this.shortToBytes(this.receivingBuffer, (short)(copyPointer -
320                           (short)2), tempLength);
321         copyPointer += tempLength;
322         copyPointer = Util.arrayCopyNonAtomic(this.ServerAppIdentityTag,
323           (short)0, this.receivingBuffer, copyPointer, (short)
324           this.ServerAppIdentityTag.length);
325         this.receivingBuffer[childPM2]++;
326         copyPointer =
                Util.arrayCopyNonAtomic(this.ServerAppRandomNumberArray,
327           (short)0, this.receivingBuffer, copyPointer, (short)
328           this.ServerAppRandomNumberArray.length);
329         this.receivingBuffer[childPM2]++;
330         copyPointer =
                Util.arrayCopyNonAtomic(this.ClientAppRandomNumberArray,
331           (short)0, this.receivingBuffer, copyPointer, (short)
332           this.ClientAppRandomNumberArray.length);
333         this.receivingBuffer[childPM2]++;
334         try {
335           this.signGenerate(this.receivingBuffer, (short)(childPM2 +
336                             (short)1), (short)(copyPointer - (short)
337                             (childPM2 + (short)1)),
```

```
338                             this.phServerAppKeyPair.getPrivate(),
339                             Signature.MODE_SIGN);
340       } catch (Exception cE) {
341         ISOException.throwIt((short)0x3141);
342       }
343       this.receivingBuffer[childPM2]++;
344       copyPointer = Util.arrayCopyNonAtomic(this.ServerAppCertificate,
345         (short)0, this.receivingBuffer, copyPointer, (short)
346         this.ServerAppCertificate.length);
347       this.receivingBuffer[childPM2]++;
348       try {
349         this.messageEncryption(this.receivingBuffer, (short)(childPM2
350                                 + (short)1), (short)(copyPointer -
351                                 (short)(childPM2 + (short)1)));
352       } catch (Exception ce) {
353         ISOException.throwIt((short)(copyPointer - (short)(childPM2 +
354                               (short)1)));
355       }
356       this.shortToBytes(this.receivingBuffer, (short)(childPM2 -
357                         (short)2), (short)(copyPointer - childPM2 -
358                         (short)1));
359       this.macGenerate(this.receivingBuffer, (short)(childPM2 +
360                        (short)1), (short)(copyPointer - (short)
361                        (childPM2 + (short)1)), Signature.MODE_SIGN);
362       this.receivingBuffer[childPM1]++;
363       copyPointer = Util.arrayCopyNonAtomic(this.ClientAppCookieArray,
364         (short)0, this.receivingBuffer, (short)copyPointer, (short)
365         this.ClientAppCookieArray.length);
366       this.receivingBuffer[childPM1]++;
367       this.shortToBytes(this.receivingBuffer, (short)(childPM1 -
368                         (short)2), (short)(copyPointer - (short)7));
369     }
370   }
371   boolean processSecondMsg(byte[] inArray) {
372     short inOffset = (short)(this.CTLVDataOffset +
373                      this.CTLVDataOffset);
374     short inLength = (short)(ProtocolHandler.bytesToShort(inArray,
375                      (short)(inOffset - (short)3)));
376     if (this.macGenerate(inArray, inOffset, inLength,
377         Signature.MODE_VERIFY)) {
378       try {
379         this.phDecryption(inArray, inOffset, inLength);
380         inOffset = (short)(this.CTLVDataOffset + this.PTLVDataOffset
381                    + (short)168);
382         inLength = 3;
383         ClientAppVerificationKey.setExponent(inArray, inOffset, inLength);
384         inOffset += (short)(inLength + this.PTLVDataOffset);
385         inLength = (short)64;
386         ClientAppVerificationKey.setModulus(inArray, inOffset, inLength);
387         inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
388         inLength = (short)84;
```

```
389         if (this.signGenerate(inArray, inOffset, inLength,
390             ClientAppVerificationKey, Signature.MODE_VERIFY)) {
391           return true;
392         } else {
393           ISOException.throwIt((short)0x6666);
394         }
395       } catch (Exception ce) {
396         ISOException.throwIt((short)0xAB23);
397       }
398       return true;
399     } else {
400       ISOException.throwIt((short)0xFA18);
401     }
402     return false;
403   }
404   void parseMessage(byte[] inBuffer) {
405     byte childLeft = inBuffer[(short)(this.CTLVDataOffset - (short)1)
406                                ];
407     short pointer = (short)this.CTLVDataOffset;
408     try {
409       while (childLeft > 0) {
410         if (Util.arrayCompare(ClientAppDHChallengeTag, (short)0, inBuffer,
411             pointer, (short)4) == 0) {
412           Util.arrayCopy(inBuffer, pointer,
                 this.ClientAppDHChanllengerArray,
413                          (short)0, (short)
414                          this.ClientAppDHChanllengerArray.length);
415           pointer += (short)this.ClientAppDHChanllengerArray.length;
416         } else if (Util.arrayCompare(this.ClientAppRandomNumberTag,
              (short)0,
417                    inBuffer, pointer, (short)4) == 0) {
418           Util.arrayCopyNonAtomic(inBuffer, pointer,
419                                   this.ClientAppRandomNumberArray,
                                        (short)0,
420                                   (short)
421                                   (this.ClientAppRandomNumberArray.length));
422           pointer += (short)(this.ClientAppRandomNumberArray.length);
423         } else if (Util.arrayCompare(this.ClientAppCookieTag, (short)0,
424                    inBuffer, pointer, (short)4) == 0) {
425           Util.arrayCopyNonAtomic(inBuffer, pointer,
426                                   this.ClientAppCookieArray, (short)0,
427                                   (short)(this.ClientAppCookieArray.length));
428           pointer += (short)(this.ClientAppCookieArray.length);
429         }
430         childLeft -= (short)1;
431       }
432     } catch (Exception cE) {
433       ISOException.throwIt((short)childLeft);
434     }
435   }
436   void protocolImplementation(){}
```

```
437   void dhInitialisation() {
438     dhKey.setModulus(ClassDH.dhModulus, (short)0,
          (short)ClassDH.dhModulus.length);
439   }
440   void dhKeyConGen(byte[] inbuff, short inbuffOffset, byte Oper_Mode)
441                       {
442     switch (Oper_Mode) {
443       case GEN_KEYCONTRIBUTION: randomExponent =
444         JCSystem.makeTransientByteArray((short)32,
445         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
446       randomDataGen.generateData(randomExponent, (short)0, (short)
447                                       randomExponent.length);
448       dhKey.setExponent(randomExponent, (short)0, (short)
449                           randomExponent.length);
450       pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
451       pkCipher.doFinal(ClassDH.dhBase, (short)0,
            (short)ClassDH.dhBase.length, inbuff,
452                          inbuffOffset);
453       break;
454       case GEN_DHKEY:
455         try {
456           dhKey.setExponent(randomExponent, (short)0, (short)
457                                 randomExponent.length);
458           pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
459           ServerAppClientAppDHGeneratedValue =
                JCSystem.makeTransientByteArray(
460             (short)ClassDH.dhModulus.length,
461             JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
462           pkCipher.doFinal(inbuff, inbuffOffset, (short)((short)
463                               inbuff.length - (short)this.PTLVDataOffset)
464                               , ServerAppClientAppDHGeneratedValue,
                                    (short)0);
465         }
466          catch (Exception cE) {
467           ISOException.throwIt((short)0xD86E);
468         }
469         break;
470       default:
471         ISOException.throwIt((short)0x5FA1);
472     }
473   }
474   void keygenerator() {
475     AESKey sessionGenKey = (AESKey)KeyBuilder.buildKey
476                               (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
477                               KeyBuilder.LENGTH_AES_128, false);
478     sessionGenKey.setKey(ServerAppClientAppDHGeneratedValue, (short)0);
479     phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
480                           InitialisationVector, (short)0, (short)
481                           InitialisationVector.length);
482     byte[] keyGenMacData = JCSystem.makeTransientByteArray((short)64,
483       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
```

```
484     short pointer = 0;
485     pointer = Util.arrayCopyNonAtomic(this.ClientAppRandomNumberArray,
486       this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
487     pointer = Util.arrayCopyNonAtomic(this.ServerAppRandomNumberArray,
488       this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
489     pointer = Util.arrayCopyNonAtomic(ServerAppClientAppDHGeneratedValue,
          (short)16,
490       keyGenMacData, (short)pointer, (short)16);
491     for (short i = 48; i < 64; i++) {
492       keyGenMacData[i] = (byte)0x02;
493     }
494     phMacGenerator.sign(keyGenMacData, (short)0, (short)
495                         keyGenMacData.length,
                                ServerAppClientAppDHGeneratedValue,
496                         (short)0);
497     this.phCipherKey.setKey(ServerAppClientAppDHGeneratedValue, (short)0);
498     for (short i = 48; i < 64; i++) {
499       keyGenMacData[i] = (byte)0x03;
500     }
501     phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
502                         InitialisationVector, (short)0, (short)
503                         InitialisationVector.length);
504     phMacGenerator.sign(keyGenMacData, (short)0, (short)
505                         keyGenMacData.length,
                                ServerAppClientAppDHGeneratedValue,
506                         (short)0);
507     this.phMacGeneratorKey.setKey(ServerAppClientAppDHGeneratedValue,
          (short)0);
508     ServerAppClientAppDHGeneratedValue = null;
509     JCSystem.requestObjectDeletion();
510   }
511   void messageEncryption(byte[] inbuff, short inbuffOffset, short
512                          inbuffLength) {
513     syCipher.init(phCipherKey, Cipher.MODE_ENCRYPT,
514                   InitialisationVector, (short)0, (short)
515                   InitialisationVector.length);
516     this.shortToBytes(inbuff, (short)(inbuffOffset - 3), (short)
517                       syCipher.doFinal(inbuff, inbuffOffset,
518                       inbuffLength, inbuff, inbuffOffset));
519   }
520   void phDecryption(byte[] inbuff, short inbuffOffset, short
521                     inbuffLength) {
522     syCipher.init(phCipherKey, Cipher.MODE_DECRYPT,
523                   InitialisationVector, (short)0, (short)
524                   InitialisationVector.length);
525     syCipher.doFinal(inbuff, inbuffOffset, inbuffLength, inbuff,
526                      inbuffOffset);
527   }
528   boolean macGenerate(byte[] inbuff, short inbuffOffset, short
529                       inbuffLength, short macMode) {
530     if (macMode == Signature.MODE_SIGN) {
```

```
531       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
532                           InitialisationVector, (short)0, (short)
533                           InitialisationVector.length);
534       try {
535         copyPointer = Util.arrayCopyNonAtomic(this.MACedDataTag,
536           (short)0, this.receivingBuffer, copyPointer, (short)
537           this.MACedDataTag.length);
538         copyPointer += 2;
539       } catch (Exception ce) {
540         ISOException.throwIt((short)0xFA17);
541       }
542       try {
543         short length = (short)phMacGenerator.sign
544                         (this.receivingBuffer, inbuffOffset,
545                         inbuffLength, inbuff, copyPointer);
546         this.shortToBytes(inbuff, (short)(copyPointer - (short)2),
547                           length);
548         copyPointer += length;
549       } catch (Exception ce) {
550         ISOException.throwIt((short)0x0987);
551       }
552       return true;
553     } else if (macMode == Signature.MODE_VERIFY) {
554       try {
555         phMacGenerator.init(phMacGeneratorKey, Signature.MODE_VERIFY,
556                             InitialisationVector, (short)0, (short)
557                             InitialisationVector.length);
558         return phMacGenerator.verify(this.receivingBuffer,
559           inbuffOffset, inbuffLength, inbuff, (short)(inbuffOffset +
560           inbuffLength + this.PTLVDataOffset), (short)16);
561       } catch (Exception cE) {
562         ISOException.throwIt((short)0xC1C2);
563       }
564     }
565     return false;
566   }
567   boolean signGenerate(byte[] inbuff, short inbuffOffset, short
568                        inbufflength, Key kpSign, short signMode) {
569     if (signMode == Signature.MODE_SIGN) {
570       copyPointer = Util.arrayCopyNonAtomic(this.SignedDataTag,
571         (short)0, this.receivingBuffer, copyPointer, (short)
572         this.SignedDataTag.length);
573       copyPointer += (short)2;
574       phSign.init((RSAPrivateKey)kpSign, Signature.MODE_SIGN);
575       signlength = phSign.sign(inbuff, (short)inbuffOffset,
576                                inbufflength, inbuff, copyPointer);
577       this.shortToBytes(this.receivingBuffer, (short)(copyPointer -
578                         (short)2), signlength);
579       copyPointer += signlength;
580       return true;
581     } else if (signMode == Signature.MODE_VERIFY) {
```

```
582         phSign.init((RSAPublicKey)kpSign, Signature.MODE_VERIFY);
583         return phSign.verify(inbuff, inbuffOffset, inbufflength, inbuff,
584                              (short)(inbuffOffset + inbufflength +
585                              this.PTLVDataOffset), (short)64);
586       }
587       return false;
588     }
589     public static short bytesToShort(byte[] ArrayBytes) {
590       return (short)(((ArrayBytes[0] << 8)) | ((ArrayBytes[1] & 0xff)));
591     }
592     public static short bytesToShort(byte[] ArrayBytes, short
593                                      arrayOffset) {
594       return (short)(((ArrayBytes[arrayOffset] << 8)) | ((ArrayBytes[
595               (short)(arrayOffset + (short)1)] & 0xff)));
596     }
597     private void shortToBytes(byte[] Array, short arrayOffset, short
598                               inShort) {
599       Array[arrayOffset] = (byte)((short)(inShort & (short)0xFF00) >>
600                           (short)0x0008);
601       Array[(short)(arrayOffset + (short)1)] = (byte)(inShort & (short)
602               0x00FF);
603     }
604 }
```

C.9 Platform Binding Protocol

The Java Card implementation of the *PBP* discussed in section 7.5 is listed in subsequent sections.

C.9.1 Initiator Smart Card Implementation

Implementation of a initiator smart card that request for the platform binding in the CDAM firewall mechanism is listed as below:

```
1  package protocolSCA;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.JCSystem;
8  import javacard.framework.Util;
9  import javacard.security.AESKey;
10 import javacard.security.Key;
11 import javacard.security.KeyBuilder;
12 import javacard.security.KeyPair;
13 import javacard.security.MessageDigest;
14 import javacard.security.RSAPrivateKey;
15 import javacard.security.RSAPublicKey;
16 import javacard.security.RandomData;
```

```
17 import javacard.security.Signature;
18 import javacardx.apdu.ExtendedLength;
19 import javacardx.crypto.Cipher;
20 public class ProtocolHandler extends Applet implements ExtendedLength
21   {
22   private byte[] SCBDHChanllengerArray;
23   private byte[] SCBRandomNumberArray;
24   private byte[] SCBCookieArray;
25   private byte[] SCASCBDHGeneratedValue;
26   private byte[] SCARandomNumberArray;
27   private byte[] SCAUserCertificate;
28   private byte[] SCACertificate;
29   private byte[] SCBDHChallengeTag = {
30     (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x01};
31   private byte[] MessageHandlerTagOne = {
32     (byte)0x1F, (byte)0xC0, (byte)0xAA, (byte)0xAA, (byte)0x00, (byte)
33      0x00, (byte)0x00};
34   private byte[] MessageHandlerTagTwo = {
35     (byte)0x1F, (byte)0xC0, (byte)0xBB, (byte)0xBB, (byte)0x00, (byte)
36      0x00, (byte)0x00};
37   private byte[] SCBIdentity = null;
38   private byte[] SCBRandomNumberTag = {
39     (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x01};
40   private byte[] SCBCookieTag = {
41     (byte)0x1F, (byte)0x5F, (byte)0x5B, (byte)0x01};
42   private byte[] EncryptedDataTag = {
43     (byte)0x1F, (byte)0xC0, (byte)0xFE, (byte)0x01};
44   private byte[] SignedDataTag = {
45     (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x02};
46   private byte[] MACedDataTag = {
47     (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x01};
48   private byte[] PlatformHash = {
49     (byte)0x1F, (byte)0x5F, (byte)0x5E, (byte)0xAF};
50   private byte[] SCAIdentityTag = {
51     (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x02, (byte)0x00, (byte)
52      0x12, (byte)0x7A, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6,
53      (byte)0xC1, (byte)0x22, (byte)0x07, (byte)0xC9, (byte)0xF9,
54      (byte)0x8D, (byte)0x11, (byte)0xED, (byte)0x34, (byte)0xDB,
55      (byte)0xF6, (byte)0x0B, (byte)0x2C};
56   private byte[] UserIdentity = {
57     (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x03, (byte)0x00, (byte)
58      0x14, (byte)0x7A, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6,
59      (byte)0xC1, (byte)0x22, (byte)0x07, (byte)0xC9, (byte)0xC9,
60      (byte)0x8D, (byte)0xD1, (byte)0xED, (byte)0xFC, (byte)0xDB,
61      (byte)0xF6, (byte)0x0B, (byte)0x2C, (byte)0x0B, (byte)0x2C};
62   private byte[] ExponentTag = {
63     (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x01};
64   private byte[] ModulusTag = {
65     (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x02};
66   private byte[] SCADHChalleneTag = {
67     (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x02};
```

```
68  private byte[] SCARandomNumberTag = {
69    (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x02};
70  private byte[] SCBCertificateTag = {
71    (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x01};
72  private byte[] SCACertificateTag = {
73    (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x02};
74  private byte[] SCAUserCertificateTag = {
75    (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x03};
76  short PTLVDataOffset = (short)6;
77  short CTLVDataOffset = (short)7;
78  short TLVLengthOffset = (short)4;
79  byte[] SCADHData;
80  final static byte CLA = (byte)0xB0;
81  final static byte StartProtocol = (byte)0x40;
82  final static byte InitiationProtocol = (byte)0xff;
83  final static short SW_CLASSNOTSUPPORTED = 0x6320;
84  final static short SW_ERROR_INS = 0x6300;
85  RandomData randomDataGen;
86  Cipher pkCipher;
87  short messageNumber = 0;
88  byte[] receivingBuffer = null;
89  short bytesLeft = 0;
90  short readCount = 0;
91  short rCount = 0;
92  short signlength = 0;
93  private RSAPublicKey dhKey = (RSAPublicKey)KeyBuilder.buildKey
94                                (KeyBuilder.TYPE_RSA_PUBLIC,
95                                KeyBuilder.LENGTH_RSA_2048, false);
96  private byte[] randomExponent;
97  final static byte GEN_KEYCONTRIBUTION = 0x01;
98  final static byte GEN_DHKEY = 0x02;
99  AESKey phCipherKey;
100 Cipher syCipher;
101 byte[] InitialisationVector = {
102   (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89, (byte)
103    0x99, (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1,
104    (byte)0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52};
105 AESKey phMacGeneratorKey;
106 Signature phMacGenerator;
107 Signature phSign;
108 KeyPair phSCAKeyPair;
109 KeyPair phUserKeyPair;
110 RSAPublicKey SCBVerificationKey = null;
111 private ProtocolHandler() {
112   phMacGeneratorKey = (AESKey)KeyBuilder.buildKey
113                        (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
114                        KeyBuilder.LENGTH_AES_128, false);
115   phMacGenerator = Signature.getInstance
116     (Signature.ALG_AES_MAC_128_NOPAD, false);
117   phSign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false)
118                                  ;
```

```
119     phSCAKeyPair = new KeyPair(KeyPair.ALG_RSA,
120                                KeyBuilder.LENGTH_RSA_512);
121     phUserKeyPair = new KeyPair(KeyPair.ALG_RSA,
122                                 KeyBuilder.LENGTH_RSA_512);
123     phCipherKey = (AESKey)KeyBuilder.buildKey
124                       (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
125                       KeyBuilder.LENGTH_AES_128, false);
126     syCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
127                                   false);
128     randomDataGen = RandomData.getInstance
129       (RandomData.ALG_SECURE_RANDOM);
130     pkCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
131     dhInitialisation();
132     phSCAKeyPair.genKeyPair();
133     phUserKeyPair.genKeyPair();
134   }
135   public static void install(byte bArray[], short bOffset, byte
136                              bLength)throws ISOException {
137     new ProtocolHandler().register();
138   }
139   public void initialiseProtocol() {
140     short initialPointer = 0;
141     SCADHData = JCSystem.makeTransientByteArray((short)((short)
142       this.ClassDH.dhModulus.length + PTLVDataOffset),
143       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
144     Util.arrayCopyNonAtomic(this.SCADHChalleneTag, (short)
145                             initialPointer, this.SCADHData, (short)0,
146                             (short)this.SCADHChalleneTag.length);
147     this.shortToBytes(SCADHData, (short)4, (short)((short)
148                       SCADHData.length - (short)PTLVDataOffset));
149     this.dhKeyConGen(this.SCADHData, this.PTLVDataOffset,
150                      ProtocolHandler.GEN_KEYCONTRIBUTION);
151     SCBDHChanllengerArray = JCSystem.makeTransientByteArray((short)(
152       (short)this.ClassDH.dhModulus.length + this.PTLVDataOffset),
153       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
154     SCBRandomNumberArray = JCSystem.makeTransientByteArray((short)22,
155       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
156     SCBCookieArray = JCSystem.makeTransientByteArray((short)22,
157       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
158     SCARandomNumberArray = JCSystem.makeTransientByteArray((short)22,
159       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
160     Util.arrayCopyNonAtomic(this.SCARandomNumberTag, (short)
161                             initialPointer, this.SCARandomNumberArray,
162                             (short)initialPointer, (short)
163                             this.SCARandomNumberTag.length);
164     this.shortToBytes(this.SCARandomNumberArray, (short)4, (short)(
165                       (short)this.SCARandomNumberArray.length -
166                       (short)PTLVDataOffset));
167     try {
168       this.SCAUserCertificate = JCSystem.makeTransientByteArray(
169         (short)86, JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
```

```
170       initialPointer = Util.arrayCopyNonAtomic
171                                (this.SCAUserCertificateTag, (short)0,
172                                this.SCAUserCertificate, (short)0, (short)
173                                this.SCAUserCertificateTag.length);
174       this.shortToBytes(this.SCAUserCertificate, (short)4, (short)
175                           (this.SCAUserCertificate.length - (short)7));
176       initialPointer = Util.arrayCopyNonAtomic(this.ExponentTag,
177         (short)0, this.SCAUserCertificate, (short)(initialPointer +
178         (short)3), (short)this.ExponentTag.length);
179       RSAPublicKey myPublic = (RSAPublicKey)
180                                 this.phUserKeyPair.getPublic();
181       short kLen = myPublic.getExponent(this.SCAUserCertificate,
182         (short)(initialPointer + (short)2));
183       this.shortToBytes(this.SCAUserCertificate, initialPointer, kLen)
184                           ;
185       initialPointer += (short)(kLen + (short)2);
186       this.SCAUserCertificate[6]++;
187       initialPointer = Util.arrayCopyNonAtomic(this.ModulusTag,
188         (short)0, this.SCAUserCertificate, (short)(initialPointer),
189         (short)this.ModulusTag.length);
190       kLen = myPublic.getModulus(this.SCAUserCertificate, (short)
191                                    (initialPointer + (short)2));
192       this.shortToBytes(this.SCAUserCertificate, initialPointer, kLen)
193                           ;
194       this.SCAUserCertificate[6]++;
195       this.SCBIdentity = JCSystem.makeTransientByteArray((short)24,
196         JCSystem.MEMORY_TYPE_TRANSIENT_RESET);
197       SCBVerificationKey = (RSAPublicKey)KeyBuilder.buildKey
198                             (KeyBuilder.TYPE_RSA_PUBLIC,
199                             KeyBuilder.LENGTH_RSA_512, false);
200     } catch (Exception cE) {
201       ISOException.throwIt((short)0xCCCC);
202     }
203     try {
204       this.SCACertificate = JCSystem.makeTransientByteArray((short)86,
205         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
206       initialPointer = Util.arrayCopyNonAtomic(this.SCACertificateTag,
207         (short)0, this.SCACertificate, (short)0, (short)
208         this.SCACertificateTag.length);
209       this.shortToBytes(this.SCACertificate, (short)4, (short)
210                           (this.SCACertificate.length - (short)7));
211       initialPointer = Util.arrayCopyNonAtomic(this.ExponentTag,
212         (short)0, this.SCACertificate, (short)(initialPointer +
213         (short)3), (short)this.ExponentTag.length);
214       RSAPublicKey myPublic = (RSAPublicKey)
215                                 this.phSCAKeyPair.getPublic();
216       short kLen = myPublic.getExponent(this.SCACertificate, (short)
217         (initialPointer + (short)2));
218       this.shortToBytes(this.SCACertificate, initialPointer, kLen);
219       initialPointer += (short)(kLen + (short)2);
220       this.SCACertificate[6]++;
```

```
221       initialPointer = Util.arrayCopyNonAtomic(this.ModulusTag,
222         (short)0, this.SCACertificate, (short)(initialPointer),
223         (short)this.ModulusTag.length);
224       kLen = myPublic.getModulus(this.SCACertificate, (short)
225                                  (initialPointer + (short)2));
226       this.shortToBytes(this.SCACertificate, initialPointer, kLen);
227       this.SCACertificate[6]++;
228     } catch (Exception cE) {
229       ISOException.throwIt((short)0x6666);
230     }
231   }
232   public void process(APDU apdu)throws ISOException {
233     byte[] apduBuffer = apdu.getBuffer();
234     if (selectingApplet()) {
235       return ;
236     }
237     if (apduBuffer[ISO7816.OFFSET_CLA] != CLA) {
238       ISOException.throwIt(SW_CLASSNOTSUPPORTED);
239     }
240     if (apduBuffer[ISO7816.OFFSET_INS] == InitiationProtocol) {
241       this.initialiseProtocol();
242       return ;
243     }
244     receivingBuffer = null;
245     bytesLeft = 0;
246     bytesLeft = apdu.getIncomingLength();
247     receivingBuffer = JCSystem.makeTransientByteArray(bytesLeft,
248       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
249     readCount = (short)((short)apdu.setIncomingAndReceive());
250     rCount = 0;
251     if (bytesLeft > 0) {
252       rCount = Util.arrayCopyNonAtomic(apduBuffer,
253         ISO7816.OFFSET_EXT_CDATA, receivingBuffer, rCount, readCount);
254       bytesLeft -= readCount;
255     }
256     while (bytesLeft > 0) {
257       try {
258         readCount = apdu.receiveBytes((short)0);
259         rCount = Util.arrayCopyNonAtomic(apduBuffer, (short)0,
260           receivingBuffer, rCount, readCount);
261         bytesLeft -= readCount;
262       } catch (Exception aE) {
263         ISOException.throwIt((short)0x7AAA);
264       }
265     }
266     try {
267       parseMessage(receivingBuffer);
268     } catch (Exception cE) {
269       ISOException.throwIt((short)0xA112);
270     }
271     if (this.receivingBuffer[3] == this.MessageHandlerTagOne[3]) {
```

```
272       receivingBuffer = JCSystem.makeTransientByteArray((short)568,
273         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
274       generateResponse((short)1);
275     } else if (this.receivingBuffer[3] ==
276                 this.MessageHandlerTagTwo[3]) {
277       processSecondMsg(receivingBuffer);
278       receivingBuffer = JCSystem.makeTransientByteArray((short)568,
279         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
280       generateResponse((short)2);
281     } else {
282       ISOException.throwIt(ProtocolHandler.SW_ERROR_INS);
283     }
284     JCSystem.requestObjectDeletion();
285     apdu.setOutgoing();
286     apdu.setOutgoingLength((short)copyPointer);
287     apdu.sendBytesLong(receivingBuffer, (short)0, (short)copyPointer);
288     JCSystem.requestObjectDeletion();
289   }
290   private void generateResponse(short msgNumber) {
291     short childPointerMessage = 6;
292     short encryptionOffset = 0;
293     copyPointer = 0;
294     if (msgNumber == 1) {
295       randomDataGen.generateData(this.SCARandomNumberArray,
296                                  this.PTLVDataOffset, (short)16);
297       this.dhKeyConGen(this.SCBDHChanllengerArray,
298                        this.PTLVDataOffset, ProtocolHandler.GEN_DHKEY)
299                        ;
300       copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagOne,
301         (short)0, this.receivingBuffer, copyPointer, (short)
302         this.MessageHandlerTagOne.length);
303       copyPointer = Util.arrayCopyNonAtomic(this.SCADHData, (short)0,
304         this.receivingBuffer, copyPointer, (short)
305         this.SCADHData.length);
306       this.receivingBuffer[childPointerMessage]++;
307       copyPointer = Util.arrayCopyNonAtomic(this.SCARandomNumberArray,
308         (short)0, this.receivingBuffer, copyPointer, (short)
309         this.SCARandomNumberArray.length);
310       this.receivingBuffer[childPointerMessage]++;
311       keygenerator();
312       copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag,
313         (short)0, this.receivingBuffer, copyPointer, (short)
314         this.EncryptedDataTag.length);
315       this.receivingBuffer[childPointerMessage]++;
316       short childEnMessage = (short)(copyPointer + (short)2);
317       copyPointer += (short)3;
318       encryptionOffset = copyPointer;
319       copyPointer = Util.arrayCopyNonAtomic(this.SCAIdentityTag,
320         (short)0, this.receivingBuffer, copyPointer, (short)
321         this.SCAIdentityTag.length);
322       this.receivingBuffer[childEnMessage]++;
```

```
323         copyPointer = Util.arrayCopyNonAtomic(this.SCARandomNumberArray,
324           (short)0, this.receivingBuffer, copyPointer, (short)
325           this.SCARandomNumberArray.length);
326         this.receivingBuffer[childEnMessage]++;
327         copyPointer = Util.arrayCopyNonAtomic(this.SCBRandomNumberArray,
328           (short)0, this.receivingBuffer, copyPointer, (short)
329           this.SCBRandomNumberArray.length);
330         this.receivingBuffer[childEnMessage]++;
331         this.signGenerate(this.receivingBuffer, encryptionOffset,
332                           (short)(copyPointer - encryptionOffset),
333                           phUserKeyPair.getPrivate(),
334                           Signature.MODE_SIGN);
335         this.receivingBuffer[childEnMessage]++;
336         copyPointer = Util.arrayCopyNonAtomic(this.SCAUserCertificate,
337           (short)0, this.receivingBuffer, copyPointer, (short)
338           this.SCAUserCertificate.length);
339         this.receivingBuffer[childEnMessage]++;
340         messageEncryption(this.receivingBuffer, encryptionOffset,
341                           (short)(copyPointer - encryptionOffset));
342         this.shortToBytes(receivingBuffer, (short)(encryptionOffset -
343                           (short)3), (short)(copyPointer -
344                           encryptionOffset));
345         macGenerate(this.receivingBuffer, encryptionOffset, (short)
346                     (copyPointer - encryptionOffset),
347                     Signature.MODE_SIGN);
348         this.receivingBuffer[childPointerMessage]++;
349         copyPointer = Util.arrayCopyNonAtomic(this.SCBCookieArray,
350           (short)0, this.receivingBuffer, copyPointer, (short)
351           this.SCBCookieArray.length);
352         this.receivingBuffer[childPointerMessage]++;
353         this.shortToBytes(this.receivingBuffer, (short)4, copyPointer);
354       } else if (msgNumber == 2) {
355         copyPointer = (short)0;
356         short tempLength = (short)0;
357         short mainChildPointer = (short)6;
358         short mainLengthPointer = (short)4;
359         short encryptedChildPointer = (short)13;
360         short generalLengthPointer = (short)0;
361         this.receivingBuffer[mainChildPointer] = (short)0;
362         this.receivingBuffer[encryptedChildPointer] = (short)0;
363         copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagTwo,
364           (short)0, this.receivingBuffer, copyPointer, (short)7);
365         this.receivingBuffer[mainChildPointer]++;
366         copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag,
367           (short)0, this.receivingBuffer, copyPointer, (short)4);
368         copyPointer += (short)3;
369         encryptionOffset = copyPointer;
370         copyPointer = Util.arrayCopyNonAtomic(this.PlatformHash, (short)
371           0, receivingBuffer, copyPointer, (short)4);
372         generalLengthPointer = copyPointer;
373         copyPointer += (short)2;
```

```
374         MessageDigest myHashGen = MessageDigest.getInstance
375           (MessageDigest.ALG_SHA_256, false);
376         tempLength = (short)myHashGen.doFinal(this.ClassDH.dhModulus,
                (short)0,
377                       (short)this.ClassDH.dhModulus.length, receivingBuffer,
378                       copyPointer);
379         this.receivingBuffer[encryptedChildPointer]++;
380         this.shortToBytes(this.receivingBuffer, generalLengthPointer,
381                           (short)(tempLength));
382         copyPointer += tempLength;
383         copyPointer = Util.arrayCopyNonAtomic(this.UserIdentity, (short)
384           0, this.receivingBuffer, copyPointer, (short)
385           this.UserIdentity.length);
386         this.receivingBuffer[encryptedChildPointer]++;
387         copyPointer = Util.arrayCopyNonAtomic(this.SCBIdentity, (short)
388           0, this.receivingBuffer, copyPointer, (short)
389           this.SCBIdentity.length);
390         this.receivingBuffer[encryptedChildPointer]++;
391         copyPointer = Util.arrayCopyNonAtomic(this.SCARandomNumberArray,
392           (short)0, this.receivingBuffer, copyPointer, (short)
393           this.SCARandomNumberArray.length);
394         this.receivingBuffer[encryptedChildPointer]++;
395         copyPointer = Util.arrayCopyNonAtomic(this.SCBRandomNumberArray,
396           (short)0, this.receivingBuffer, (short)copyPointer, (short)
397           this.SCBRandomNumberArray.length);
398         this.receivingBuffer[encryptedChildPointer]++;
399         try {
400           this.signGenerate(receivingBuffer, (short)(encryptionOffset),
401                             (short)(copyPointer - encryptionOffset),
402                             phSCAKeyPair.getPrivate(),
403                             Signature.MODE_SIGN);
404           this.receivingBuffer[encryptedChildPointer]++;
405         } catch (Exception cE) {
406           ISOException.throwIt((short)0xFA17);
407         }
408         copyPointer = Util.arrayCopyNonAtomic(this.SCACertificate,
409           (short)0, this.receivingBuffer, copyPointer, (short)
410           this.SCACertificate.length);
411         this.receivingBuffer[encryptedChildPointer]++;
412         try {
413           this.messageEncryption(receivingBuffer, (short)
414                                  (encryptedChildPointer + (short)1),
415                                  (short)(copyPointer -
416                                  (encryptedChildPointer + (short)1)));
417         } catch (Exception cE) {
418           ISOException.throwIt((short)(copyPointer -
419                                encryptedChildPointer + (short)1));
420         }
421         this.shortToBytes(this.receivingBuffer, (short)
422                           (encryptedChildPointer - (short)2), (short)
423                           (copyPointer - (short)(encryptedChildPointer
```

```
424                         + (short)1)));
425       this.macGenerate(receivingBuffer, (short)(encryptedChildPointer
426                         + (short)1), (short)(copyPointer -
427                         (encryptedChildPointer + (short)1)),
428                         Signature.MODE_SIGN);
429       this.receivingBuffer[mainChildPointer]++;
430       copyPointer = Util.arrayCopyNonAtomic(this.SCBCookieArray,
431         (short)0, this.receivingBuffer, copyPointer, (short)
432         this.SCBCookieArray.length);
433       this.receivingBuffer[mainChildPointer]++;
434       this.shortToBytes(this.receivingBuffer, mainLengthPointer,
435                         (short)(copyPointer - (short)7));
436     }
437   }
438   void platformHashGeneration(byte[] inArray, short inOffset){}
439   void processSecondMsg(byte[] inArray) {
440     short inOffset = (short)(this.CTLVDataOffset +
441                       this.CTLVDataOffset);
442     short inLength = (short)(ProtocolHandler.bytesToShort(inArray,
443                       (short)(inOffset - (short)3)));
444     if (this.macGenerate(inArray, inOffset, inLength,
445         Signature.MODE_VERIFY)) {
446       this.phDecryption(inArray, inOffset, inLength);
447       Util.arrayCopyNonAtomic(inArray, inOffset, this.SCBIdentity,
448                               (short)0, (short)
449                               this.SCBIdentity.length);
450       inOffset += (short)151;
451       inLength = (short)3;
452       SCBVerificationKey.setExponent(inArray, inOffset, inLength);
453       inOffset += (short)(inLength + this.PTLVDataOffset);
454       inLength = (short)64;
455       SCBVerificationKey.setModulus(inArray, inOffset, inLength);
456       inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
457       inLength = (short)68;
458       if (this.signGenerate(inArray, inOffset, inLength,
459           SCBVerificationKey, Signature.MODE_VERIFY)) {
460         return;
461       } else {
462         ISOException.throwIt((short)0x6666);
463       }
464     } else {
465       ISOException.throwIt((short)0xFA18);
466     }
467   }
468   void parseMessage(byte[] inBuffer) {
469     byte childLeft = inBuffer[(short)(this.CTLVDataOffset - (short)1)
470                               ];
471     short pointer = (short)this.CTLVDataOffset;
472     try {
473       while (childLeft > 0) {
474         if (Util.arrayCompare(SCBDHChallengeTag, (short)0, inBuffer,
```

```
475             pointer, (short)4) == 0) {
476           Util.arrayCopy(inBuffer, pointer,
477                          this.SCBDHChanllengerArray, (short)0, (short)
478                          this.SCBDHChanllengerArray.length);
479           pointer += (short)this.SCBDHChanllengerArray.length;
480         } else if (Util.arrayCompare(this.SCBRandomNumberTag, (short)
481                    0, inBuffer, pointer, (short)4) == 0) {
482           Util.arrayCopyNonAtomic(inBuffer, pointer,
483                                   this.SCBRandomNumberArray, (short)0,
484                                   (short)
485                                   (this.SCBRandomNumberArray.length));
486           pointer += (short)(this.SCBRandomNumberArray.length);
487         } else if (Util.arrayCompare(this.SCBCookieTag, (short)0,
488                    inBuffer, pointer, (short)4) == 0) {
489           Util.arrayCopyNonAtomic(inBuffer, pointer,
490                                   this.SCBCookieArray, (short)0,
491                                   (short)(this.SCBCookieArray.length))
492                                   ;
493           pointer += (short)(this.SCBCookieArray.length);
494         }
495         childLeft -= (short)1;
496       }
497     } catch (Exception cE) {
498       ISOException.throwIt((short)childLeft);
499     }
500   }
501   void protocolImplementation(){}
502   void dhInitialisation() {
503     dhKey.setModulus(ClassDH.dhModulus, (short)0,
          (short)ClassDH.dhModulus.length);
504   }
505   void dhKeyConGen(byte[] inbuff, short inbuffOffset, byte Oper_Mode)
506                    {
507     switch (Oper_Mode) {
508       case GEN_KEYCONTRIBUTION: randomExponent =
509         JCSystem.makeTransientByteArray((short)32,
510         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
511       randomDataGen.generateData(randomExponent, (short)0, (short)
512                                  randomExponent.length);
513       dhKey.setExponent(randomExponent, (short)0, (short)
514                         randomExponent.length);
515       pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
516       pkCipher.doFinal(ClassDH.dhBase, (short)0,
            (short)ClassDH.dhBase.length, inbuff,
517                        inbuffOffset);
518       break;
519       case GEN_DHKEY:
520         try {
521           dhKey.setExponent(randomExponent, (short)0, (short)
522                             randomExponent.length);
523           pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
```

```
524             SCASCBDHGeneratedValue = JCSystem.makeTransientByteArray(
525               (short)ClassDH.dhModulus.length,
526               JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
527             pkCipher.doFinal(inbuff, inbuffOffset, (short)((short)
528                               inbuff.length - (short)this.PTLVDataOffset)
529                               , SCASCBDHGeneratedValue, (short)0);
530           }
531            catch (Exception cE) {
532             ISOException.throwIt((short)0xD86E);
533           }
534           break;
535         default:
536           ISOException.throwIt((short)0x5FA1);
537       }
538     }
539     void keygenerator() {
540       AESKey sessionGenKey = (AESKey)KeyBuilder.buildKey
541                               (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
542                               KeyBuilder.LENGTH_AES_128, false);
543       sessionGenKey.setKey(SCASCBDHGeneratedValue, (short)0);
544       phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
545                           InitialisationVector, (short)0, (short)
546                           InitialisationVector.length);
547       byte[] keyGenMacData = JCSystem.makeTransientByteArray((short)64,
548         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
549       short pointer = 0;
550       pointer = Util.arrayCopyNonAtomic(this.SCBRandomNumberArray,
551         this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
552       pointer = Util.arrayCopyNonAtomic(this.SCARandomNumberArray,
553         this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
554       pointer = Util.arrayCopyNonAtomic(SCASCBDHGeneratedValue, (short)
555         16, keyGenMacData, (short)pointer, (short)16);
556       for (short i = 48; i < 64; i++) {
557         keyGenMacData[i] = (byte)0x02;
558       }
559       phMacGenerator.sign(keyGenMacData, (short)0, (short)
560                           keyGenMacData.length, SCASCBDHGeneratedValue,
561                           (short)0);
562       this.phCipherKey.setKey(SCASCBDHGeneratedValue, (short)0);
563       for (short i = 48; i < 64; i++) {
564         keyGenMacData[i] = (byte)0x03;
565       }
566       phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
567                           InitialisationVector, (short)0, (short)
568                           InitialisationVector.length);
569       phMacGenerator.sign(keyGenMacData, (short)0, (short)
570                           keyGenMacData.length, SCASCBDHGeneratedValue,
571                           (short)0);
572       this.phMacGeneratorKey.setKey(SCASCBDHGeneratedValue, (short)0);
573       SCASCBDHGeneratedValue = null;
574       JCSystem.requestObjectDeletion();
```

```
575   }
576   void messageEncryption(byte[] inbuff, short inbuffOffset, short
577                          inbuffLength) {
578     syCipher.init(phCipherKey, Cipher.MODE_ENCRYPT,
579                   InitialisationVector, (short)0, (short)
580                   InitialisationVector.length);
581     short temp;
582     this.shortToBytes(inbuff, (short)(inbuffOffset - 3), temp =
583                       (short)syCipher.doFinal(inbuff, inbuffOffset,
584                       inbuffLength, inbuff, inbuffOffset));
585   }
586   void phDecryption(byte[] inbuff, short inbuffOffset, short
587                     inbuffLength) {
588     syCipher.init(phCipherKey, Cipher.MODE_DECRYPT,
589                   InitialisationVector, (short)0, (short)
590                   InitialisationVector.length);
591     syCipher.doFinal(inbuff, inbuffOffset, inbuffLength, inbuff,
592                      inbuffOffset);
593   }
594   boolean macGenerate(byte[] inbuff, short inbuffOffset, short
595                       inbuffLength, short macMode) {
596     if (macMode == Signature.MODE_SIGN) {
597       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
598                           InitialisationVector, (short)0, (short)
599                           InitialisationVector.length);
600       try {
601         copyPointer = Util.arrayCopyNonAtomic(this.MACedDataTag,
602           (short)0, this.receivingBuffer, copyPointer, (short)
603           this.MACedDataTag.length);
604         copyPointer += 2;
605       } catch (Exception ce) {
606         ISOException.throwIt((short)0xFA17);
607       }
608       try {
609         short length = (short)phMacGenerator.sign
610                        (this.receivingBuffer, inbuffOffset,
611                        inbuffLength, inbuff, copyPointer);
612         this.shortToBytes(inbuff, (short)(copyPointer - (short)2),
613                           length);
614         copyPointer += length;
615       } catch (Exception ce) {
616         ISOException.throwIt((short)0x0987);
617       }
618       return true;
619     } else if (macMode == Signature.MODE_VERIFY) {
620       try {
621         phMacGenerator.init(phMacGeneratorKey, Signature.MODE_VERIFY,
622                             InitialisationVector, (short)0, (short)
623                             InitialisationVector.length);
624         return phMacGenerator.verify(this.receivingBuffer,
625           inbuffOffset, inbuffLength, inbuff, (short)(inbuffOffset +
```

```
626             inbuffLength + this.PTLVDataOffset), (short)16);
627         } catch (Exception cE) {
628           ISOException.throwIt((short)0xC1C2);
629         }
630       }
631       return false;
632     }
633     boolean signGenerate(byte[] inbuff, short inbuffOffset, short
634                          inbufflength, Key kpSign, short signMode) {
635       if (signMode == Signature.MODE_SIGN) {
636         copyPointer = Util.arrayCopyNonAtomic(this.SignedDataTag,
637           (short)0, this.receivingBuffer, copyPointer, (short)
638           this.SignedDataTag.length);
639         copyPointer += (short)2;
640         phSign.init((RSAPrivateKey)kpSign, Signature.MODE_SIGN);
641         signlength = phSign.sign(inbuff, (short)inbuffOffset,
642                                  inbufflength, inbuff, copyPointer);
643         this.shortToBytes(this.receivingBuffer, (short)(copyPointer -
644                           (short)2), signlength);
645         copyPointer += signlength;
646         return true;
647       } else if (signMode == Signature.MODE_VERIFY) {
648         phSign.init((RSAPublicKey)kpSign, Signature.MODE_VERIFY);
649         return phSign.verify(inbuff, inbuffOffset, inbufflength, inbuff,
650                              (short)(inbuffOffset + inbufflength +
651                              this.PTLVDataOffset), (short)64);
652       }
653       return false;
654     }
655     public static short bytesToShort(byte[] ArrayBytes) {
656       return (short)(((ArrayBytes[0] << 8)) | ((ArrayBytes[1] & 0xff)));
657     }
658     public static short bytesToShort(byte[] ArrayBytes, short
659                                      arrayOffset) {
660       return (short)(((ArrayBytes[arrayOffset] << 8)) | ((ArrayBytes[
661               (short)(arrayOffset + (short)1)] & 0xff)));
662     }
663     private void shortToBytes(byte[] Array, short inShort) {
664       Array[0] = (byte)((short)(inShort & (short)0xFF00) >> (short)
665                   0x0008);
666       Array[1] = (byte)(inShort & (short)0x00FF);
667     }
668     private void shortToBytes(byte[] Array, short arrayOffset, short
669                               inShort) {
670       Array[arrayOffset] = (byte)((short)(inShort & (short)0xFF00) >>
671                           (short)0x0008);
672       Array[(short)(arrayOffset + (short)1)] = (byte)(inShort & (short)
673             0x00FF);
674     }
675 }
```

C.9.2 Responder Smart Card Implementation

Implementation of a responder smart card that request for the platform binding in the CDAM firewall mechanism is listed as below:

```
1  package protocolSCB;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.JCSystem;
8  import javacard.framework.Util;
9  import javacard.security.AESKey;
10 import javacard.security.Key;
11 import javacard.security.KeyBuilder;
12 import javacard.security.KeyPair;
13 import javacard.security.MessageDigest;
14 import javacard.security.RSAPrivateKey;
15 import javacard.security.RSAPublicKey;
16 import javacard.security.RandomData;
17 import javacard.security.Signature;
18 import javacardx.apdu.ExtendedLength;
19 import javacardx.crypto.Cipher;
20 public class ProtocolHandler extends Applet implements ExtendedLength
21   {
22   private byte[] SCARandomNumberArray;
23   private byte[] SCACookieArray;
24   private byte[] SCBSCADHGeneratedValue;
25   private byte[] SCBRandomNumberArray;
26   private byte[] SCBCertificate;
27   private byte[] SCADHChallengeTag = {
28     (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x01};
29   private byte[] MessageHandlerTagOne = {
30     (byte)0x1F, (byte)0xC0, (byte)0xAA, (byte)0xAA, (byte)0x00, (byte)
31      0x00, (byte)0x00};
32   private byte[] MessageHandlerTagTwo = {
33     (byte)0x1F, (byte)0xC0, (byte)0xBB, (byte)0xBB, (byte)0x00, (byte)
34      0x00, (byte)0x00};
35   private byte[] SCAIdentity = null;
36   private byte[] SCARandomNumberTag = {
37     (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x01};
38   private byte[] SCACookieTag = {
39     (byte)0x1F, (byte)0x5F, (byte)0x5B, (byte)0x01};
40   private byte[] EncryptedDataTag = {
41     (byte)0x1F, (byte)0xC0, (byte)0xFE, (byte)0x01};
42   private byte[] SignedDataTag = {
43     (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x02};
44   private byte[] MACedDataTag = {
45     (byte)0x1F, (byte)0x5F, (byte)0x5D, (byte)0x01};
46   private byte[] PlatformHash = {
```

```
47      (byte)0x1F, (byte)0x5F, (byte)0x5E, (byte)0xAF};
48    private byte[] SCBIdentityTag = {
49      (byte)0x1F, (byte)0x5F, (byte)0x5F, (byte)0x02, (byte)0x00, (byte)
50       0x0C, (byte)0x7A, (byte)0xD5, (byte)0xB7, (byte)0xD0, (byte)0xB6,
51       (byte)0xC1, (byte)0x22, (byte)0x07, (byte)0xC9, (byte)0xF9,
52       (byte)0x8D, (byte)0x11};
53    private byte[] ExponentTag = {
54      (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x01};
55    private byte[] ModulusTag = {
56      (byte)0x1F, (byte)0x5F, (byte)0xEE, (byte)0x02};
57    private byte[] SCBDHChalleneTag = {
58      (byte)0x1F, (byte)0x5F, (byte)0x5C, (byte)0x02};
59    private byte[] SCBRandomNumberTag = {
60      (byte)0x1F, (byte)0x5F, (byte)0x5A, (byte)0x02};
61    private byte[] SCACertificateTag = {
62      (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x01};
63    private byte[] SCBCertificateTag = {
64      (byte)0x1F, (byte)0xC0, (byte)0xF0, (byte)0x02};
65    private byte[] SCBProtocolInitiatorTag = {
66      (byte)0x1F, (byte)0x5F, (byte)0xA1, (byte)0xB2};
67    short PTLVDataOffset = (short)6;
68    short CTLVDataOffset = (short)7;
69    short TLVLengthOffset = (short)4;
70    short copyPointer = (short)0;
71    byte[] SCBDHData;
72    final static byte CLA = (byte)0xB0;
73    final static byte StartProtocol = (byte)0x40;
74    final static byte InitiationProtocol = (byte)0xff;
75    final static short SW_CLASSNOTSUPPORTED = 0x6320;
76    final static short SW_ERROR_INS = 0x6300;
77    RandomData randomDataGen;
78    Cipher pkCipher;
79    short messageNumber = 0;
80    byte[] receivingBuffer = null;
81    short bytesLeft = 0;
82    short readCount = 0;
83    short rCount = 0;
84    short signlength = 0;
85    private RSAPublicKey dhKey = (RSAPublicKey)KeyBuilder.buildKey
86                                 (KeyBuilder.TYPE_RSA_PUBLIC,
87                                 KeyBuilder.LENGTH_RSA_2048, false);
88    private byte[] randomExponent;
89    final static byte GEN_KEYCONTRIBUTION = 0x01;
90    final static byte GEN_DHKEY = 0x02;
91    AESKey phCipherKey;
92    Cipher syCipher;
93    byte[] InitialisationVector = {
94      (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89, (byte)
95       0x99, (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1,
96       (byte)0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52};
97    AESKey phMacGeneratorKey;
```

```
98  Signature phMacGenerator;
99  Signature phSign;
100 KeyPair phSCBKeyPair;
101 KeyPair phUserKeyPair;
102 RSAPublicKey SCAVerificationKey = null;
103 private ProtocolHandler() {
104   phMacGeneratorKey = (AESKey)KeyBuilder.buildKey
105                       (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
106                       KeyBuilder.LENGTH_AES_128, false);
107   phMacGenerator = Signature.getInstance
108     (Signature.ALG_AES_MAC_128_NOPAD, false);
109   phSign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false)
110                                     ;
111   phSCBKeyPair = new KeyPair(KeyPair.ALG_RSA,
112                              KeyBuilder.LENGTH_RSA_512);
113   phUserKeyPair = new KeyPair(KeyPair.ALG_RSA,
114                               KeyBuilder.LENGTH_RSA_512);
115   phCipherKey = (AESKey)KeyBuilder.buildKey
116                 (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
117                 KeyBuilder.LENGTH_AES_128, false);
118   syCipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
119                                 false);
120   randomDataGen = RandomData.getInstance
121     (RandomData.ALG_SECURE_RANDOM);
122   pkCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
123   dhInitialisation();
124   phSCBKeyPair.genKeyPair();
125   phUserKeyPair.genKeyPair();
126 }
127 public static void install(byte bArray[], short bOffset, byte
128                            bLength)throws ISOException {
129   new ProtocolHandler().register();
130 }
131 public void initialiseProtocol() {
132   short initialPointer = 0;
133   SCBDHData = JCSystem.makeTransientByteArray((short)((short)
134     this.ClassDH.dhModulus.length + PTLVDataOffset),
135     JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
136   Util.arrayCopyNonAtomic(this.SCBDHChalleneTag, (short)
137                           initialPointer, this.SCBDHData, (short)0,
138                           (short)this.SCBDHChalleneTag.length);
139   this.shortToBytes(SCBDHData, (short)4, (short)((short)
140                     SCBDHData.length - (short)PTLVDataOffset));
141   this.dhKeyConGen(this.SCBDHData, this.PTLVDataOffset,
142                    ProtocolHandler.GEN_KEYCONTRIBUTION);
143   SCADHChanllengerArray = JCSystem.makeTransientByteArray((short)(
144     (short)this.ClassDH.dhModulus.length + this.PTLVDataOffset),
145     JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
146   SCARandomNumberArray = JCSystem.makeTransientByteArray((short)22,
147     JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
148   SCACookieArray = JCSystem.makeTransientByteArray((short)22,
```

```
149       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
150     SCBRandomNumberArray = JCSystem.makeTransientByteArray((short)22,
151       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
152     Util.arrayCopyNonAtomic(this.SCBRandomNumberTag, (short)
153                             initialPointer, this.SCBRandomNumberArray,
154                             (short)initialPointer, (short)
155                             this.SCBRandomNumberTag.length);
156     this.shortToBytes(this.SCBRandomNumberArray, (short)4, (short)(
157                       (short)this.SCBRandomNumberArray.length - (short)
158                       PTLVDataOffset));
159     try {
160       this.SCBCertificate = JCSystem.makeTransientByteArray((short)86,
161         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
162       initialPointer = Util.arrayCopyNonAtomic(this.SCBCertificateTag,
163                                (short)0, this.SCBCertificate, (short)0,
                                     (short)
164                                this.SCBCertificateTag.length);
165       this.shortToBytes(this.SCBCertificate, (short)4, (short)
166                         (this.SCBCertificate.length - (short)7));
167       initialPointer = Util.arrayCopyNonAtomic(this.ExponentTag,
168                                (short)0, this.SCBCertificate,
                                     (short)(initialPointer + (short)
169                                3), (short)this.ExponentTag.length);
170       RSAPublicKey myPublic = (RSAPublicKey)
171                                this.phSCBKeyPair.getPublic();
172       short kLen = myPublic.getExponent(this.SCBCertificate, (short)
173         (initialPointer + (short)2));
174       this.shortToBytes(this.SCBCertificate, initialPointer, kLen);
175       initialPointer += (short)(kLen + (short)2);
176       this.SCBCertificate[6]++;
177       initialPointer = Util.arrayCopyNonAtomic(this.ModulusTag,
178                                (short)0, this.SCBCertificate,
                                     (short)(initialPointer), (short)
179                                this.ModulusTag.length);
180       kLen = myPublic.getModulus(this.SCBCertificate, (short)
181                                  (initialPointer + (short)2));
182       this.shortToBytes(this.SCBCertificate, initialPointer, kLen);
183       this.SCBCertificate[6]++;
184       SCAVerificationKey = (RSAPublicKey)KeyBuilder.buildKey
185                            (KeyBuilder.TYPE_RSA_PUBLIC,
186                            KeyBuilder.LENGTH_RSA_512, false);
187     } catch (Exception cE) {
188       ISOException.throwIt((short)0x6666);
189     }
190   }
191   public void process(APDU apdu)throws ISOException {
192     byte[] apduBuffer = apdu.getBuffer();
193     if (selectingApplet()) {
194       this.initialiseProtocol();
195       return ;
196     }
```

```
197     if (apduBuffer[ISO7816.OFFSET_CLA] != CLA) {
198       ISOException.throwIt(SW_CLASSNOTSUPPORTED);
199     }
200     if (apduBuffer[ISO7816.OFFSET_INS] == InitiationProtocol) {
201       receivingBuffer = JCSystem.makeTransientByteArray((short)64,
202         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
203       generateResponse((short)1);
204       apdu.setOutgoing();
205       apdu.setOutgoingLength((short)copyPointer);
206       apdu.sendBytesLong(receivingBuffer, (short)0, (short)
207                          copyPointer);
208       return ;
209     }
210     receivingBuffer = null;
211     bytesLeft = 0;
212     bytesLeft = apdu.getIncomingLength();
213     receivingBuffer = JCSystem.makeTransientByteArray(bytesLeft,
214       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
215     readCount = (short)((short)apdu.setIncomingAndReceive());
216     rCount = 0;
217     if (bytesLeft > 0) {
218       rCount = Util.arrayCopyNonAtomic(apduBuffer,
219         ISO7816.OFFSET_EXT_CDATA, receivingBuffer, rCount, readCount);
220       bytesLeft -= readCount;
221     }
222     while (bytesLeft > 0) {
223       try {
224         readCount = apdu.receiveBytes((short)0);
225         rCount = Util.arrayCopyNonAtomic(apduBuffer, (short)0,
226           receivingBuffer, rCount, readCount);
227         bytesLeft -= readCount;
228       } catch (Exception aE) {
229         ISOException.throwIt((short)0x7AAA);
230       }
231     }
232     if (this.receivingBuffer[3] == this.MessageHandlerTagOne[3]) {
233       try {
234         parseMessage(receivingBuffer);
235       } catch (Exception cE) {
236         ISOException.throwIt((short)0xA112);
237       }
238       receivingBuffer = JCSystem.makeTransientByteArray((short)600,
239         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
240       generateResponse((short)2);
241       JCSystem.requestObjectDeletion();
242       apdu.setOutgoing();
243       apdu.setOutgoingLength((short)copyPointer);
244       apdu.sendBytesLong(receivingBuffer, (short)0, (short)
245                          copyPointer);
246     } else if (this.receivingBuffer[3] ==
247                this.MessageHandlerTagTwo[3]) {
```

```
248       if (processSecondMsg(receivingBuffer)) {
249         return ;
250       } else {
251         ISOException.throwIt((short)0xFA17);
252       }
253       return ;
254     } else {
255       ISOException.throwIt(ProtocolHandler.SW_ERROR_INS);
256     }
257     JCSystem.requestObjectDeletion();
258   }
259   private void generateResponse(short msgNumber) {
260     short childPM1 = 0;
261     short childPM2 = 0;
262     copyPointer = 0;
263     if (msgNumber == 1) {
264       copyPointer = Util.arrayCopy(this.SCBProtocolInitiatorTag,
265                                    (short)0, this.receivingBuffer,
266                                    copyPointer, (short)
267                                    this.SCBProtocolInitiatorTag.length)
268                                    ;
269       randomDataGen.generateData(this.SCBRandomNumberArray,
270                                  this.PTLVDataOffset, (short)16);
271       childPM1 = copyPointer;
272       copyPointer += 2;
273       phMacGeneratorKey.setKey(this.SCBRandomNumberArray,
274                                this.PTLVDataOffset);
275       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
276                           InitialisationVector, (short)0, (short)
277                           InitialisationVector.length);
278       short length = 0;
279       length = phMacGenerator.sign(SCBDHData, (short)
280                                    this.PTLVDataOffset, (short)
281                                    (SCBDHData.length -
282                                    this.PTLVDataOffset),
283                                    this.receivingBuffer, copyPointer);
284       copyPointer += length;
285       this.shortToBytes(this.receivingBuffer, childPM1, length);
286       return ;
287     } else if (msgNumber == 2) {
288       this.dhKeyConGen(this.SCADHChanllengerArray, this.PTLVDataOffset,
289                        ProtocolHandler.GEN_DHKEY);
290       keygenerator();
291       childPM1 = (short)6;
292       copyPointer = Util.arrayCopyNonAtomic(this.MessageHandlerTagTwo,
293         (short)0, this.receivingBuffer, copyPointer, (short)
294         this.MessageHandlerTagTwo.length);
295       copyPointer = Util.arrayCopyNonAtomic(this.SCBDHData, (short)0,
296                        this.receivingBuffer, (short)copyPointer, (short)
297                        this.SCBDHData.length);
298       this.receivingBuffer[childPM1]++;
```

```
299         copyPointer = Util.arrayCopyNonAtomic(this.SCBRandomNumberArray,
300           (short)0, this.receivingBuffer, copyPointer, (short)
301           this.SCBRandomNumberArray.length);
302         this.receivingBuffer[childPM1]++;
303         copyPointer = Util.arrayCopyNonAtomic(this.EncryptedDataTag,
304           (short)0, this.receivingBuffer, copyPointer, (short)
305           this.EncryptedDataTag.length);
306         copyPointer += 3;
307         childPM2 = (short)(copyPointer - (short)1);
308         this.receivingBuffer[childPM1]++;
309         copyPointer = Util.arrayCopyNonAtomic(this.PlatformHash, (short)
310           0, this.receivingBuffer, copyPointer, (short)
311           this.PlatformHash.length);
312         copyPointer += 2;
313         MessageDigest myHashGen = MessageDigest.getInstance
314           (MessageDigest.ALG_SHA_256, false);
315         short tempLength = (short)myHashGen.doFinal(this.ClassDH.dhModulus,
316                             (short)0, (short)this.ClassDH.dhModulus.length,
317                             receivingBuffer, copyPointer);
318         this.receivingBuffer[childPM2]++;
319         this.shortToBytes(this.receivingBuffer, (short)(copyPointer -
320                           (short)2), tempLength);
321         copyPointer += tempLength;
322         copyPointer = Util.arrayCopyNonAtomic(this.SCBIdentityTag,
323           (short)0, this.receivingBuffer, copyPointer, (short)
324           this.SCBIdentityTag.length);
325         this.receivingBuffer[childPM2]++;
326         copyPointer = Util.arrayCopyNonAtomic(this.SCBRandomNumberArray,
327           (short)0, this.receivingBuffer, copyPointer, (short)
328           this.SCBRandomNumberArray.length);
329         this.receivingBuffer[childPM2]++;
330         copyPointer = Util.arrayCopyNonAtomic(this.SCARandomNumberArray,
331           (short)0, this.receivingBuffer, copyPointer, (short)
332           this.SCARandomNumberArray.length);
333         this.receivingBuffer[childPM2]++;
334         try {
335           this.signGenerate(this.receivingBuffer, (short)(childPM2 +
336                             (short)1), (short)(copyPointer - (short)
337                             (childPM2 + (short)1)),
338                             this.phSCBKeyPair.getPrivate(),
339                             Signature.MODE_SIGN);
340         } catch (Exception cE) {
341           ISOException.throwIt((short)0x3141);
342         }
343         this.receivingBuffer[childPM2]++;
344         copyPointer = Util.arrayCopyNonAtomic(this.SCBCertificate,
345           (short)0, this.receivingBuffer, copyPointer, (short)
346           this.SCBCertificate.length);
347         this.receivingBuffer[childPM2]++;
348         try {
349           this.messageEncryption(this.receivingBuffer, (short)(childPM2
```

```
350                                 + (short)1), (short)(copyPointer -
351                                 (short)(childPM2 + (short)1)));
352       } catch (Exception ce) {
353         ISOException.throwIt((short)(copyPointer - (short)(childPM2 +
354                               (short)1)));
355       }
356       this.shortToBytes(this.receivingBuffer, (short)(childPM2 -
357                         (short)2), (short)(copyPointer - childPM2 -
358                         (short)1));
359       this.macGenerate(this.receivingBuffer, (short)(childPM2 +
360                        (short)1), (short)(copyPointer - (short)
361                        (childPM2 + (short)1)), Signature.MODE_SIGN);
362       this.receivingBuffer[childPM1]++;
363       copyPointer = Util.arrayCopyNonAtomic(this.SCACookieArray,
364         (short)0, this.receivingBuffer, (short)copyPointer, (short)
365         this.SCACookieArray.length);
366       this.receivingBuffer[childPM1]++;
367       this.shortToBytes(this.receivingBuffer, (short)(childPM1 -
368                         (short)2), (short)(copyPointer - (short)7));
369     }
370   }
371   boolean processSecondMsg(byte[] inArray) {
372     short inOffset = (short)(this.CTLVDataOffset +
373                      this.CTLVDataOffset);
374     short inLength = (short)(ProtocolHandler.bytesToShort(inArray,
375                      (short)(inOffset - (short)3)));
376     if (this.macGenerate(inArray, inOffset, inLength,
377         Signature.MODE_VERIFY)) {
378       try {
379         this.phDecryption(inArray, inOffset, inLength);
380         inOffset = (short)(this.CTLVDataOffset + this.PTLVDataOffset
381                    + (short)168);
382         inLength = 3;
383         SCAVerificationKey.setExponent(inArray, inOffset, inLength);
384         inOffset += (short)(inLength + this.PTLVDataOffset);
385         inLength = (short)64;
386         SCAVerificationKey.setModulus(inArray, inOffset, inLength);
387         inOffset = (short)(this.CTLVDataOffset + this.CTLVDataOffset);
388         inLength = (short)84;
389         if (this.signGenerate(inArray, inOffset, inLength,
390             SCAVerificationKey, Signature.MODE_VERIFY)) {
391           return true;
392         } else {
393           ISOException.throwIt((short)0x6666);
394         }
395       } catch (Exception ce) {
396         ISOException.throwIt((short)0xAB23);
397       }
398       return true;
399     } else {
400       ISOException.throwIt((short)0xFA18);
```

```
401     }
402     return false;
403   }
404   void parseMessage(byte[] inBuffer) {
405     byte childLeft = inBuffer[(short)(this.CTLVDataOffset - (short)1)
406                                   ];
407     short pointer = (short)this.CTLVDataOffset;
408     try {
409       while (childLeft > 0) {
410         if (Util.arrayCompare(SCADHChallengeTag, (short)0, inBuffer,
411             pointer, (short)4) == 0) {
412           Util.arrayCopy(inBuffer, pointer, this.SCADHChanllengerArray,
413                          (short)0, (short)
414                          this.SCADHChanllengerArray.length);
415           pointer += (short)this.SCADHChanllengerArray.length;
416         } else if (Util.arrayCompare(this.SCARandomNumberTag, (short)0,
417                    inBuffer, pointer, (short)4) == 0) {
418           Util.arrayCopyNonAtomic(inBuffer, pointer,
419                                   this.SCARandomNumberArray, (short)0,
420                                   (short)
421                                   (this.SCARandomNumberArray.length));
422           pointer += (short)(this.SCARandomNumberArray.length);
423         } else if (Util.arrayCompare(this.SCACookieTag, (short)0,
424                    inBuffer, pointer, (short)4) == 0) {
425           Util.arrayCopyNonAtomic(inBuffer, pointer,
426                                   this.SCACookieArray, (short)0,
427                                   (short)(this.SCACookieArray.length));
428           pointer += (short)(this.SCACookieArray.length);
429         }
430         childLeft -= (short)1;
431       }
432     } catch (Exception cE) {
433       ISOException.throwIt((short)childLeft);
434     }
435   }
436   void protocolImplementation(){}
437   void dhInitialisation() {
438     dhKey.setModulus(ClassDH.dhModulus, (short)0,
          (short)ClassDH.dhModulus.length);
439   }
440   void dhKeyConGen(byte[] inbuff, short inbuffOffset, byte Oper_Mode)
                    {
442     switch (Oper_Mode) {
443       case GEN_KEYCONTRIBUTION: randomExponent =
444         JCSystem.makeTransientByteArray((short)32,
445         JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
446       randomDataGen.generateData(randomExponent, (short)0, (short)
447                                  randomExponent.length);
448       dhKey.setExponent(randomExponent, (short)0, (short)
449                         randomExponent.length);
450       pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
```

```
451       pkCipher.doFinal(ClassDH.dhBase, (short)0,
              (short)ClassDH.dhBase.length, inbuff,
452                           inbuffOffset);
453       break;
454       case GEN_DHKEY:
455         try {
456           dhKey.setExponent(randomExponent, (short)0, (short)
457                                 randomExponent.length);
458           pkCipher.init(dhKey, Cipher.MODE_ENCRYPT);
459           SCBSCADHGeneratedValue = JCSystem.makeTransientByteArray(
460             (short)ClassDH.dhModulus.length,
461             JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
462           pkCipher.doFinal(inbuff, inbuffOffset, (short)((short)
463                               inbuff.length - (short)this.PTLVDataOffset)
464                               , SCBSCADHGeneratedValue, (short)0);
465         }
466          catch (Exception cE) {
467           ISOException.throwIt((short)0xD86E);
468         }
469         break;
470       default:
471         ISOException.throwIt((short)0x5FA1);
472     }
473   }
474   void keygenerator() {
475     AESKey sessionGenKey = (AESKey)KeyBuilder.buildKey
476                               (KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
477                               KeyBuilder.LENGTH_AES_128, false);
478     sessionGenKey.setKey(SCBSCADHGeneratedValue, (short)0);
479     phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
480                           InitialisationVector, (short)0, (short)
481                           InitialisationVector.length);
482     byte[] keyGenMacData = JCSystem.makeTransientByteArray((short)64,
483       JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT);
484     short pointer = 0;
485     pointer = Util.arrayCopyNonAtomic(this.SCARandomNumberArray,
486       this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
487     pointer = Util.arrayCopyNonAtomic(this.SCBRandomNumberArray,
488       this.PTLVDataOffset, keyGenMacData, (short)pointer, (short)16);
489     pointer = Util.arrayCopyNonAtomic(SCBSCADHGeneratedValue, (short)16,
490       keyGenMacData, (short)pointer, (short)16);
491     for (short i = 48; i < 64; i++) {
492       keyGenMacData[i] = (byte)0x02;
493     }
494     phMacGenerator.sign(keyGenMacData, (short)0, (short)
495                           keyGenMacData.length, SCBSCADHGeneratedValue,
496                           (short)0);
497     this.phCipherKey.setKey(SCBSCADHGeneratedValue, (short)0);
498     for (short i = 48; i < 64; i++) {
499       keyGenMacData[i] = (byte)0x03;
500     }
```

```
501     phMacGenerator.init(sessionGenKey, Signature.MODE_SIGN,
502                         InitialisationVector, (short)0, (short)
503                         InitialisationVector.length);
504     phMacGenerator.sign(keyGenMacData, (short)0, (short)
505                         keyGenMacData.length, SCBSCADHGeneratedValue,
506                         (short)0);
507     this.phMacGeneratorKey.setKey(SCBSCADHGeneratedValue, (short)0);
508     SCBSCADHGeneratedValue = null;
509     JCSystem.requestObjectDeletion();
510   }
511   void messageEncryption(byte[] inbuff, short inbuffOffset, short
512                          inbuffLength) {
513     syCipher.init(phCipherKey, Cipher.MODE_ENCRYPT,
514                   InitialisationVector, (short)0, (short)
515                   InitialisationVector.length);
516     this.shortToBytes(inbuff, (short)(inbuffOffset - 3), (short)
517                       syCipher.doFinal(inbuff, inbuffOffset,
518                       inbuffLength, inbuff, inbuffOffset));
519   }
520   void phDecryption(byte[] inbuff, short inbuffOffset, short
521                     inbuffLength) {
522     syCipher.init(phCipherKey, Cipher.MODE_DECRYPT,
523                   InitialisationVector, (short)0, (short)
524                   InitialisationVector.length);
525     syCipher.doFinal(inbuff, inbuffOffset, inbuffLength, inbuff,
526                      inbuffOffset);
527   }
528   boolean macGenerate(byte[] inbuff, short inbuffOffset, short
529                       inbuffLength, short macMode) {
530     if (macMode == Signature.MODE_SIGN) {
531       phMacGenerator.init(phMacGeneratorKey, Signature.MODE_SIGN,
532                           InitialisationVector, (short)0, (short)
533                           InitialisationVector.length);
534       try {
535         copyPointer = Util.arrayCopyNonAtomic(this.MACedDataTag,
536           (short)0, this.receivingBuffer, copyPointer, (short)
537           this.MACedDataTag.length);
538         copyPointer += 2;
539       } catch (Exception ce) {
540         ISOException.throwIt((short)0xFA17);
541       }
542       try {
543         short length = (short)phMacGenerator.sign
544                           (this.receivingBuffer, inbuffOffset,
545                           inbuffLength, inbuff, copyPointer);
546         this.shortToBytes(inbuff, (short)(copyPointer - (short)2),
547                           length);
548         copyPointer += length;
549       } catch (Exception ce) {
550         ISOException.throwIt((short)0x0987);
551       }
```

```
552       return true;
553     } else if (macMode == Signature.MODE_VERIFY) {
554       try {
555         phMacGenerator.init(phMacGeneratorKey, Signature.MODE_VERIFY,
556                             InitialisationVector, (short)0, (short)
557                             InitialisationVector.length);
558         return phMacGenerator.verify(this.receivingBuffer,
559           inbuffOffset, inbuffLength, inbuff, (short)(inbuffOffset +
560           inbuffLength + this.PTLVDataOffset), (short)16);
561       } catch (Exception cE) {
562         ISOException.throwIt((short)0xC1C2);
563       }
564     }
565     return false;
566   }
567   boolean signGenerate(byte[] inbuff, short inbuffOffset, short
568                        inbufflength, Key kpSign, short signMode) {
569     if (signMode == Signature.MODE_SIGN) {
570       copyPointer = Util.arrayCopyNonAtomic(this.SignedDataTag,
571         (short)0, this.receivingBuffer, copyPointer, (short)
572         this.SignedDataTag.length);
573       copyPointer += (short)2;
574       phSign.init((RSAPrivateKey)kpSign, Signature.MODE_SIGN);
575       signlength = phSign.sign(inbuff, (short)inbuffOffset,
576                                inbufflength, inbuff, copyPointer);
577       this.shortToBytes(this.receivingBuffer, (short)(copyPointer -
578                         (short)2), signlength);
579       copyPointer += signlength;
580       return true;
581     } else if (signMode == Signature.MODE_VERIFY) {
582       phSign.init((RSAPublicKey)kpSign, Signature.MODE_VERIFY);
583       return phSign.verify(inbuff, inbuffOffset, inbufflength, inbuff,
584                            (short)(inbuffOffset + inbufflength +
585                            this.PTLVDataOffset), (short)64);
586     }
587     return false;
588   }
589   public static short bytesToShort(byte[] ArrayBytes) {
590     return (short)(((ArrayBytes[0] << 8)) | ((ArrayBytes[1] & 0xff)));
591   }
592   public static short bytesToShort(byte[] ArrayBytes, short
593                                    arrayOffset) {
594     return (short)(((ArrayBytes[arrayOffset] << 8)) | ((ArrayBytes[
595             (short)(arrayOffset + (short)1)] & 0xff)));
596   }
597   private void shortToBytes(byte[] Array, short arrayOffset, short
598                             inShort) {
599     Array[arrayOffset] = (byte)((short)(inShort & (short)0xFF00) >>
600                          (short)0x0008);
601     Array[(short)(arrayOffset + (short)1)] = (byte)(inShort & (short)
602           0x00FF);
```

```
603   }
604 }
```

C.10 Abstract Virtual Machine

In this section, we illustrate the implementation of the abstract virtual machine that counts the number of selected opcodes a Java Card application has and calculate the associated cost for individual security mechanism.

```
1  package abstractVM;
2
3  import java.io.*;
4  import java.util.Iterator;
5  import org.apache.lucene.analysis.Analyzer;
6  import org.apache.lucene.analysis.standard.StandardAnalyzer;
7  import org.apache.lucene.document.Document;
8  import org.apache.lucene.document.Field;
9  import org.apache.lucene.index.CorruptIndexException;
10 import org.apache.lucene.index.IndexReader;
11 import org.apache.lucene.index.IndexWriter;
12 import org.apache.lucene.queryParser.ParseException;
13 import org.apache.lucene.queryParser.QueryParser;
14 import org.apache.lucene.search.Hit;
15 import org.apache.lucene.search.Hits;
16 import org.apache.lucene.search.IndexSearcher;
17 import org.apache.lucene.search.Query;
18 import org.apache.lucene.store.Directory;
19 import org.apache.lucene.store.FSDirectory;
20 import org.apache.lucene.store.LockObtainFailedException;
21 public class abstractVirtualMachine {
22   private String inputClassFileName = "evaluationFile";
23   private String mnemonicOutputFileName = inputClassFileName + ".txt";
24   public static final String FILES_TO_INDEX_DIRECTORY =
25     "D:\\evaluationFolder";
26   public static final String INDEX_DIRECTORY =
27     "D:\\evaluationFolder\\indexFolder";
28   public static final String FIELD_PATH = "path";
29   public static final String FIELD_CONTENTS = "contents";
30   public static final String[] keywordList =  {
31     "aaload", "iand", "aastore", "iastore", "aconst_null", "icmp",
32       "aload", "iconst_0", "aload_0", "iconst_1", "aload_1",
33       "iconst_2", "aload_2", "iconst_3", "aload_3", "iconst_4",
34       "anewarray", "iconst_5", "areturn", "iconst_m1", "arraylength",
35       "idiv", "astore", "if_acmpeq", "astore_0", "if_acmpeq_w",
36       "astore_1", "if_acmpne", "astore_2", "if_acmpne_w", "astore_3",
37       "if_scmpeq", "athrow", "if_scmpeq_w", "baload", "if_scmpge",
38       "bastore", "if_scmpge_w", "bipush", "if_scmpgt", "bspush",
39       "if_scmpgt_w", "checkcast", "if_scmple", "dup", "if_scmple_w",
40       "dup_x", "if_scmplt", "dup2", "if_scmplt_w", "getfield_a",
41       "if_scmpne", "getfield_a_this", "if_scmpne_w", "getfield_a_w",
```

```
42        "ifeq", "getfield_b", "ifeq_w", "getfield_b_this", "ifge",
43        "getfield_b_w", "ifge_w", "getfield_i", "ifgt",
44        "getfield_i_this", "ifgt_w", "getfield_i_w", "ifle",
45        "getfield_s", "ifle_w", "getfield_s_this", "iflt",
46        "getfield_s_w", "iflt_w", "getstatic_a", "ifne", "getstatic_b",
47        "ifne_w", "getstatic_i", "ifnonnull", "getstatic_s",
48        "ifnonnull_w", "goto", "ifnull", "goto_w", "ifnull_w", "i2b",
49        "iinc", "i2s", "iinc_w", "iadd", "iipush", "iaload", "iload",
50        "iload_0", "putstatic_s", "iload_1", "ret", "iload_2", "return",
51        "iload_3", "s2b", "ilookupswitch", "s2i", "imul", "sadd",
52        "ineg", "saload", "instanceof", "sand", "invokeinterface",
53        "sastore", "invokespecial", "sconst_0", "invokestatic",
54        "sconst_1", "invokevirtual", "sconst_2", "ior", "sconst_3",
55        "irem", "sconst_4", "ireturn", "sconst_5", "ishl", "sconst_m1",
56        "ishr", "sdiv", "istore", "sinc", "istore_0", "sinc_w",
57        "istore_1", "sipush", "istore_2", "sload", "istore_3",
58        "sload_0", "isub", "sload_1", "itableswitch", "sload_2",
59        "iushr", "sload_3", "ixor", "slookupswitch", "jsr", "smul",
60        "new", "sneg", "newarray", "sor", "nop", "srem", "pop",
61        "sreturn", "pop2", "sshl", "putfield_a", "sshr",
62        "putfield_a_this", "sspush", "putfield_a_w", "sstore",
63        "putfield_b", "sstore_0", "putfield_b_this", "sstore_1",
64        "putfield_b_w", "sstore_2", "putfield_i", "sstore_3",
65        "putfield_i_this", "ssub", "putfield_i_w", "stableswitch",
66        "putfield_s", "sushr", "putfield_s_this", "swap_x",
67        "putfield_s_w", "sxor", "putstatic_a", "putstatic_b",
68        "putstatic_i",
69    };
70    public static void main(String[] args) {
71      abstractVirtualMachine virtualMachine = new
72        abstractVirtualMachine();
73      String command = "javap -c " + virtualMachine.inputClassFileName;
74      virtualMachine.runCommand(command.split(" "));
75      createIndex();
76      String myFile = "D:\\evaluationFolder\\EvaluationResults.eva";
77      FileOutputStream outStream = new FileOutputStream(myFile);
78      PrintWriter out = new PrintWriter(outStream);
79      int numberPresent = 0;
80      for (int i = 0; i < keywordList.length; i++) {
81        numberPresent = searchIndex(keywordList[i]);
82        out.println(keywordList[i] + " : " + numberPresent + );
83      }
84      out.close();
85      System.out.println("========== END ==========");
86    }
87    public void runCommand(String[] inputCommandString) {
88      int number = inputCommandString.length;
89      try {
90        String[] commands = new String[inputCommandString.length + 2];
91        commands[0] = "cmd.exe";
92        commands[1] = "/c";
```

```
93        for (int i = 0; i < number; i++) {
94          commands[i + 2] = inputCommandString[i];
95        }
96        System.out.print("Executing: ");
97        for (int i = 0; i < commands.length; i++) {
98          System.out.print(commands[i] + " ");
99        }
100       Runtime runtime = Runtime.getRuntime();
101       Process process = runtime.exec(commands);
102       CheckStream cmdProcessInputStream = new CheckStream
103         (process.getInputStream());
104       CheckStream cmdProcessErrorStream = new CheckStream
105         (process.getErrorStream());
106       cmdProcessInputStream.start();
107       cmdProcessErrorStream.start();
108       System.out.print("Waiting.......");
109       int done = process.waitFor();
110       process.destroy();
111       System.out.println("Conversion Completed.");
112     } catch (InterruptedException ie) {
113       System.out.println("Error Execution: " + ie.getMessage());
114     } catch (IOException ioe) {
115       System.out.println("Error IO Operations: " + ioe.getMessage());
116     }
117   }
118   public static void createIndex()throws CorruptIndexException,
119                                     LockObtainFailedException,
120                                     IOException {
121     File fileDir = new File(FILES_TO_INDEX_DIRECTORY);
122     File indexDir = new File(INDEX_DIRECTORY);
123     Analyzer luceneAnalyzer = new StandardAnalyzer();
124     IndexWriter indexWriter = new IndexWriter(indexDir,
125       luceneAnalyzer, true);
126     File[] textFiles = fileDir.listFiles();
127     long startTime = new Date().getTime();
128     for (int i = 0; i < textFiles.length; i++) {
129       if (textFiles[i].isFile() >> textFiles[i].getName().endsWith(
130           ".txt")) {
131         System.out.println("File " + textFiles[i].getCanonicalPath()
132                            + " is being indexed");
133         Reader textReader = new FileReader(textFiles[i]);
134         Document document = new Document();
135         document.add(Field.Text(FIELD_CONTENTS, textReader));
136         document.add(Field.Text(FIELD_PATH, textFiles[i].getPath()));
137         indexWriter.addDocument(document);
138       }
139     }
140     indexWriter.optimize();
141     indexWriter.close();
142     long endTime = new Date().getTime();
143     System.out.println("It took " + (endTime - startTime) +
```

```
144                         " milliseconds to create an index for the files in
                               the directory " + fileDir.getPath());
145   }
146   public static int searchIndex(String searchString)throws
147                                     IOException, ParseException {
148     System.out.println("Searching for '" + searchString + "'");
149     Directory directory = FSDirectory.getDirectory(INDEX_DIRECTORY);
150     IndexReader indexReader = IndexReader.open(directory);
151     IndexSearcher indexSearcher = new IndexSearcher(indexReader);
152     Analyzer analyzer = new StandardAnalyzer();
153     QueryParser queryParser = new QueryParser(FIELD_CONTENTS,
154       analyzer);
155     Query query = queryParser.parse(searchString);
156     Hits hits = indexSearcher.search(query);
157     System.out.println("Number of hits: " + hits.length());
158     return hits.length();
159   }
160   class CheckStream extends Thread {
161     BufferedReader bufferedReader;
162     String lineread = "";
163     CheckStream(InputStream inputStream) {
164       bufferedReader = new BufferedReader(new InputStreamReader
165         (inputStream));
166     }
167     public void run() {
168       try {
169         FileWriter fileWriter = new FileWriter(mnemonicOutputFileName)
170           ;
171         while ((lineread = bufferedReader.readLine()) != null) {
172           System.out.println(lineread);
173           fileWriter.write(lineread + "\n");
174         }
175         fileWriter.close();
176       } catch (IOException ioe) {
177         System.out.println("IOException: " + ioe.getMessage());
178       }
179     }
180   }
181 }
```

C.11 Implementation Helper Classes

In this section, we detail the helper classes that we implemented to overcome the limited capability of our test bed.

C.11.1 Protocol Cryptographic Support

The helper function this section implements the support of cryptographic algorihtms that an SP, card manufacturer or administrative authority uses during the respective protocol

execution.

```
1  package javacardterminal;
2
3  import java.math.BigInteger;
4  import java.nio.ByteBuffer;
5  import java.security.spec.RSAPublicKeySpec;
6  import java.security.spec.InvalidKeySpecException;
7  import java.security.interfaces.RSAPublicKey;
8  import javax.crypto.spec.SecretKeySpec;
9  import javax.crypto.spec.IvParameterSpec;
10 import java.security.*;
11 import javax.crypto.*;
12 import org.bouncycastle.crypto.macs.CBCBlockCipherMac;
13 import org.bouncycastle.crypto.engines.AESEngine;
14 import org.bouncycastle.crypto.params.KeyParameter;
15 import org.bouncycastle.crypto.params.ParametersWithIV;
16 public class ProtocolHelperClass {
17   byte[] ServiceProviderRandom = new byte[16];
18   byte[] SmartCardRandom = new byte[16];
19   Cipher rsaCipher = null;
20   SecureRandom myRNG = null;
21   PrivateKey mySignatureGenerationKey = null;
22   PublicKey mySignatureVerificationKey = null;
23   Signature mySignature = null;
24   public static final short SIGN_MODE_GENERATION = 1;
25   public static final short SIGN_MODE_VERIFICATION = 2;
26   byte[] mySessionAESEnKey = new byte[16];
27   SecretKeySpec myAESKey = null;
28   Cipher myAESCipher = null;
29   byte[] InitialisationVector = {
30     (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89, (byte)
31      0x99, (byte)0x8C, (byte)0xAF, (byte)0xC5, (byte)0x7D, (byte)0xB1,
32      (byte)0x7C, (byte)0x62, (byte)0x0A, (byte)0x86, (byte)0x52 };
33   byte[] myLongTermMacKey = {
34     (byte)0xAC, (byte)0x40, (byte)0x32, (byte)0xEF, (byte)0x4F, (byte)
35      0x2D, (byte)0x9A, (byte)0xE3, (byte)0x9D, (byte)0xF3, (byte)0x0B,
36      (byte)0x5C, (byte)0x8F, (byte)0xFD, (byte)0xAC, (byte)0x50 };
37   byte[] mySessionMacKey = new byte[16];
38   SecretKeySpec myMacKey = null;
39   private RSAPublicKey myRSAPublicKey;
40   private byte[] dhBase = {
41     (byte)0xAC, (byte)0x40, (byte)0x32, (byte)0xEF, (byte)0x4F, (byte)
42      0x2D, (byte)0x9A, (byte)0xE3, (byte)0x9D, (byte)0xF3, (byte)0x0B,
43      (byte)0x5C, (byte)0x8F, (byte)0xFD, (byte)0xAC, (byte)0x50,
44      (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89,
45      (byte)0x99, (byte)0x8C, (byte)0xAF, (byte)0x74, (byte)0x86,
46      (byte)0x6A, (byte)0x08, (byte)0xCF, (byte)0xE4, (byte)0xFF,
47      (byte)0xE3, (byte)0xA6, (byte)0x82, (byte)0x4A, (byte)0x4E,
48      (byte)0x10, (byte)0xB9, (byte)0xA6, (byte)0xF0, (byte)0xDD,
49      (byte)0x92, (byte)0x1F, (byte)0x01, (byte)0xA7, (byte)0x0C,
50      (byte)0x4A, (byte)0xFA, (byte)0xAB, (byte)0x73, (byte)0x9D,
```

```
51        (byte)0x77, (byte)0x00, (byte)0xC2, (byte)0x9F, (byte)0x52,
52        (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)0x7C, (byte)0x62,
53        (byte)0x0A, (byte)0x86, (byte)0x52, (byte)0xBE, (byte)0x5E,
54        (byte)0x90, (byte)0x01, (byte)0xA8, (byte)0xD6, (byte)0x6A,
55        (byte)0xD7, (byte)0xC1, (byte)0x76, (byte)0x69, (byte)0x10,
56        (byte)0x19, (byte)0x99, (byte)0x02, (byte)0x4A, (byte)0xF4,
57        (byte)0xD0, (byte)0x27, (byte)0x27, (byte)0x5A, (byte)0xC1,
58        (byte)0x34, (byte)0x8B, (byte)0xB8, (byte)0xA7, (byte)0x62,
59        (byte)0xD0, (byte)0x52, (byte)0x1B, (byte)0xC9, (byte)0x8A,
60        (byte)0xE2, (byte)0x47, (byte)0x15, (byte)0x04, (byte)0x22,
61        (byte)0xEA, (byte)0x1E, (byte)0xD4, (byte)0x09, (byte)0x93,
62        (byte)0x9D, (byte)0x54, (byte)0xDA, (byte)0x74, (byte)0x60,
63        (byte)0xCD, (byte)0xB5, (byte)0xF6, (byte)0xC6, (byte)0xB2,
64        (byte)0x50, (byte)0x71, (byte)0x7C, (byte)0xBE, (byte)0xF1,
65        (byte)0x80, (byte)0xEB, (byte)0x34, (byte)0x11, (byte)0x8E,
66        (byte)0x98, (byte)0xD1, (byte)0x19, (byte)0x52, (byte)0x9A,
67        (byte)0x45, (byte)0xD6, (byte)0xF8, (byte)0x34, (byte)0x56,
68        (byte)0x6E, (byte)0x30, (byte)0x25, (byte)0xE3, (byte)0x16,
69        (byte)0xA3, (byte)0x30, (byte)0xEF, (byte)0xBB, (byte)0x77,
70        (byte)0xA8, (byte)0x6F, (byte)0x0C, (byte)0x1A, (byte)0xB1,
71        (byte)0x5B, (byte)0x05, (byte)0x1A, (byte)0xE3, (byte)0xD4,
72        (byte)0x28, (byte)0xC8, (byte)0xF8, (byte)0xAC, (byte)0xB7,
73        (byte)0x0A, (byte)0x81, (byte)0x37, (byte)0x15, (byte)0x0B,
74        (byte)0x8E, (byte)0xEB, (byte)0x10, (byte)0xE1, (byte)0x83,
75        (byte)0xED, (byte)0xD1, (byte)0x99, (byte)0x63, (byte)0xDD,
76        (byte)0xD9, (byte)0xE2, (byte)0x63, (byte)0xE4, (byte)0x77,
77        (byte)0x05, (byte)0x89, (byte)0xEF, (byte)0x6A, (byte)0xA2,
78        (byte)0x1E, (byte)0x7F, (byte)0x5F, (byte)0x2F, (byte)0xF3,
79        (byte)0x81, (byte)0xB5, (byte)0x39, (byte)0xCC, (byte)0xE3,
80        (byte)0x40, (byte)0x9D, (byte)0x13, (byte)0xCD, (byte)0x56,
81        (byte)0x6A, (byte)0xFB, (byte)0xB4, (byte)0x8D, (byte)0x6C,
82        (byte)0x01, (byte)0x91, (byte)0x81, (byte)0xE1, (byte)0xBC,
83        (byte)0xFE, (byte)0x94, (byte)0xB3, (byte)0x02, (byte)0x69,
84        (byte)0xED, (byte)0xFE, (byte)0x72, (byte)0xFE, (byte)0x9B,
85        (byte)0x6A, (byte)0xA4, (byte)0xBD, (byte)0x7B, (byte)0x5A,
86        (byte)0x0F, (byte)0x1C, (byte)0x71, (byte)0xCF, (byte)0xFF,
87        (byte)0x4C, (byte)0x19, (byte)0xC4, (byte)0x18, (byte)0xE1,
88        (byte)0xF6, (byte)0xEC, (byte)0x01, (byte)0x79, (byte)0x81,
89        (byte)0xBC, (byte)0x08, (byte)0x7F, (byte)0x2A, (byte)0x70,
90        (byte)0x65, (byte)0xB3, (byte)0x84, (byte)0xB8, (byte)0x90,
91        (byte)0xD3, (byte)0x19, (byte)0x1F, (byte)0x2B, (byte)0xFA};
92    private static final String dhModulus =
93 "AD107E1E9123A9D0D660FAA79559C51FA20D64E5683B9FD1B54B1597B61D0A75E6FA1"
94 + "41DF95A56DBAF9A3C407BA1DF15EB3D688A309C180E1DE6B85A1274A0A66D3F8152AD"
95 + "6AC2129037C9EDEFDA4DF8D91E8FEF55B7394B7AD5B7D0B6C12207C9F98D11ED34DBF"
96 + "6C6BA0B2C8BBC27BE6A00E0A0B9C49708B3BF8A317091883681286130BC8985DB1602"
97 + "E714415D9330278273C7DE31EFDC7310F7121FD5A07415987D9ADC0A486DCDF93ACC4"
98 + "4328387315D75E198C641A480CD86A1B9E587E8BE60E69CC928B2B9C52172E413042E"
99 + "9B23F10B0E16E79763C9B53DCF4BA80A29E3FB73C16B8E75B97EF363E2FFA31F71CF9"
100 + "DE5384E71B81C0AC4DFFE0C10E64F";
101    private byte[] randomExponent = new byte[32];
```

```
102   private byte[] DHContribution = new byte[512];
103   void protocolInitialise() {
104     try {
105       KeyPairGenerator myKeyGenerator = KeyPairGenerator.getInstance(
106         "RSA");
107       myKeyGenerator.initialize(512);
108       KeyPair myKeyPair = myKeyGenerator.genKeyPair();
109       mySignatureGenerationKey = myKeyPair.getPrivate();
110       mySignatureVerificationKey = myKeyPair.getPublic();
111       mySignature = Signature.getInstance("SHA1withRSA");
112     } catch (Exception cE) {
113       System.out.println(
114                           "Protocol Helper Class Initialisation Failed : "
                                + cE.getMessage());
115     }
116   }
117   byte[] GenerateDHPublicValue()throws NoSuchAlgorithmException,
118                                       InvalidKeyException,
119                                       IllegalBlockSizeException,
120                                       NoSuchProviderException,
121                                       BadPaddingException,
122                                       NoSuchPaddingException,
123                                       InvalidKeySpecException {
124     rsaCipher = Cipher.getInstance("RSA/None/NoPadding", "BC");
125     KeyFactory myKeyFactory = KeyFactory.getInstance("RSA", "BC");
126     myRNG = SecureRandom.getInstance("SHA1PRNG");
127     myRNG.nextBytes(randomExponent);
128     RSAPublicKeySpec myPublicKeySpec = new RSAPublicKeySpec(new
129             BigInteger(dhModulus, 16), new BigInteger(byteToString
130             (randomExponent), 16));
131     myRSAPublicKey = (RSAPublicKey)myKeyFactory.generatePublic
132                           (myPublicKeySpec);
133     rsaCipher.init(Cipher.ENCRYPT_MODE, myRSAPublicKey);
134     DHContribution = rsaCipher.doFinal(dhBase);
135     return DHContribution;
136   }
137   byte[] GenerateDHSessionKeyMaterial(byte[] inbuff, int offset, int
138     length)throws NoSuchAlgorithmException, InvalidKeyException,
139     IllegalBlockSizeException, NoSuchProviderException,
140     BadPaddingException, NoSuchPaddingException,
141     InvalidKeySpecException {
142     rsaCipher.init(Cipher.ENCRYPT_MODE, myRSAPublicKey);
143     return rsaCipher.doFinal(inbuff, offset, length);
144   }
145   void GenerateMac(byte[] inbuff, int inbuffOffset, int inbuffLength,
146                    byte[] outbuff, int outbuffOffset, byte[] MacKey)
147                    throws NoSuchAlgorithmException,
148                    InvalidKeyException, IllegalBlockSizeException,
149                    NoSuchProviderException, BadPaddingException,
150                    NoSuchPaddingException, InvalidKeySpecException {
151     AESEngine AESMacEngine = new AESEngine();
```

```
152     KeyParameter myMacKey = new KeyParameter(MacKey);
153     CBCBlockCipherMac myAESMac = new CBCBlockCipherMac(AESMacEngine,
154       128);
155     ParametersWithIV ivparam = new ParametersWithIV(myMacKey,
156       InitialisationVector);
157     myAESMac.init(ivparam);
158     myAESMac.update(inbuff, inbuffOffset, inbuffLength);
159     myAESMac.doFinal(outbuff, outbuffOffset);
160   }
161   boolean SignatureMethod(byte[] inBuff, int inBuffOffset, int
162                           inBuffLength, byte[] outBuff, int
163                           outBuffOffset, Key inKey, short SIGN_MODE)
164                           throws InvalidKeyException,
165                           SignatureException {
166     ByteBuffer myTempByteBuffer = ByteBuffer.wrap(inBuff,
167       inBuffOffset, inBuffLength);
168     if (inKey == null && SIGN_MODE == this.SIGN_MODE_GENERATION) {
169       inKey = (Key)this.mySignatureGenerationKey;
170     } else if (inKey == null && SIGN_MODE ==
171                 this.SIGN_MODE_VERIFICATION) {
172       inKey = (Key)this.mySignatureVerificationKey;
173     }
174     switch (SIGN_MODE) {
175       case SIGN_MODE_GENERATION: mySignature.initSign((PrivateKey)
176         inKey);
177       mySignature.update(myTempByteBuffer);
178       mySignature.sign(outBuff, outBuffOffset, 64);
179       return true;
180       case SIGN_MODE_VERIFICATION:
181         mySignature.initVerify((PublicKey)inKey);
182         mySignature.update(myTempByteBuffer);
183         return mySignature.verify(outBuff, outBuffOffset, 64);
184       default:
185         System.out.println("ERROR----------WRONG MODE SELECTION");
186     }
187     return false;
188   }
189   void GenerateDecryption(byte[] inbuff, int inbuffOffset, int
190                           inbuffLength, byte[] outbuff, int
191                           outbuffOffset, byte[] EnKey)throws
192                           NoSuchAlgorithmException,
193                           InvalidKeyException,
194                           IllegalBlockSizeException,
195                           NoSuchProviderException,
196                           BadPaddingException, NoSuchPaddingException,
197                           InvalidKeySpecException,
198                           ShortBufferException,
199                           InvalidAlgorithmParameterException {
200     try {
201       IvParameterSpec ivS = new IvParameterSpec(InitialisationVector);
202       myAESKey = new SecretKeySpec(EnKey, "AES");
```

```
203       myAESCipher = Cipher.getInstance("AES/CBC/NoPadding");
204       myAESCipher.init(Cipher.DECRYPT_MODE, myAESKey, ivS);
205       myAESCipher.doFinal(inbuff, inbuffOffset, inbuffLength, outbuff,
206                           outbuffOffset);
207     } catch (IllegalBlockSizeException ce) {
208       System.out.println("Error at liken 140 : " + ce.getMessage() +
209                          "\nInput Length " + inbuffLength);
210     }
211   }
212   void GenerateEncryption(byte[] inbuff, int inbuffOffset, int
213                           inbuffLength, byte[] outbuff, int
214                           outbuffOffset, byte[] EnKey)throws
215                           NoSuchAlgorithmException,
216                           InvalidKeyException,
217                           IllegalBlockSizeException,
218                           NoSuchProviderException,
219                           BadPaddingException, NoSuchPaddingException,
220                           InvalidKeySpecException,
221                           ShortBufferException,
222                           InvalidAlgorithmParameterException {
223     IvParameterSpec ivS = new IvParameterSpec(InitialisationVector);
224     myAESKey = new SecretKeySpec(EnKey, "AES");
225     myAESCipher = Cipher.getInstance("AES/CBC/NoPadding");
226     myAESCipher.init(Cipher.ENCRYPT_MODE, myAESKey, ivS);
227     myAESCipher.doFinal(inbuff, inbuffOffset, inbuffLength, outbuff,
228                         outbuffOffset);
229   }
230   public ProtocolHelperClass() {
231     Security.addProvider(new
232                          org.bouncycastle.jce.provider.BouncyCastleProvider());
233   }
234   public PublicKey getPublicKey() {
235     return this.mySignatureVerificationKey;
236   }
237   public byte[] getRandomNumber() {
238     try {
239       myRNG = SecureRandom.getInstance("SHA1PRNG");
240       myRNG.nextBytes(this.ServiceProviderRandom);
241     } catch (Exception cE) {
242       System.out.println(
243                          "Error ? : ProtocolHelperClass.getRandomNumber :
                               " + cE.getMessage());
244     }
245     return this.ServiceProviderRandom;
246   }
247   public static String byteToString(byte[] inArray) {
248     byte[] HEX_CHAR_TABLE =  {
249       (byte)'0', (byte)'1', (byte)'2', (byte)'3', (byte)'4', (byte)
250         '5', (byte)'6', (byte)'7', (byte)'8', (byte)'9', (byte)'a',
251         (byte)'b', (byte)'c', (byte)'d', (byte)'e', (byte)'f'
252     };
```

```
253     byte[] hex = new byte[2 * inArray.length];
254     int index = 0;
255     for (byte b: inArray) {
256       int v = b & 0xFF;
257       hex[index++] = HEX_CHAR_TABLE[v >>> 4];
258       hex[index++] = HEX_CHAR_TABLE[v & 0xF];
259     }
260     try {
261       return new String(hex, "ASCII");
262     } catch (Exception cE) {
263       System.out.println("Exception in bytesToString : " +
264                          cE.getMessage());
265     }
266     return "Error";
267   }
268 }
```

C.11.2 CAMS Implementation

The implementation of the helper function discussed in this section provides the functionality of a Card Application Management System (CAMS) that provides an interface to the smart card.

```
1  package javacardterminal;
2  import java.util.*;
3  import javax.smartcardio.*;
4  import java.math.BigInteger;
5  import java.io.FileOutputStream;
6  import java.io.PrintWriter;
7  public class Terminal {
8    long protocolStartTime = 0;
9    long protocolEndTime = 0;
10   TerminalFactory myTerminal = TerminalFactory.getDefault();
11   CardTerminals myCardTerminals = myTerminal.terminals();
12   List < CardTerminal > listTerminal = null;
13   Card myCard = null;
14   CardChannel myCardChannel = null;
15   CardTerminal myCardTerminal = null;
16   ProtocolHandler myProtocolHandler = new ProtocolHandler();
17   ProtocolHandlerSCIn myProtocolHanlderSCIn = new ProtocolHandlerSCIn();
18   private static final int TimeOut = 10;
19   private static final int MAX_APDU_SIZE = 1028;
20   private ConstructedTLV messageIncoming =
         ConstructedTLV.getConstructedTLV();
21   private static final byte[] CMD_APPLICATION_SELECT = {
22     (byte)0x00, (byte)0xA4, (byte)0x04, (byte)0x00, (byte)0x09,
           (byte)0xD0,
23     (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x62, (byte)0x02,
           (byte)0x01,
24     (byte)0x0C, (byte)0x08};
25   private static final byte[] CMD_APPLICATION_INITIALISATION = {
```

```
26      (byte)0xB0, (byte)0xFF, (byte)0x00, (byte)0x00, (byte)0x01,
            (byte)0xAA};
27    private static CommandAPDU SELECT_APDU = new CommandAPDU
28      (CMD_APPLICATION_SELECT);
29    private static CommandAPDU Application_INITIALISATION = new CommandAPDU
30      (CMD_APPLICATION_INITIALISATION);
31    byte[] response;
32    public Terminal(){}
33    public void TerminalConnection() {
34      try {
35        listTerminal = myCardTerminals.list();
36      } catch (Exception e) {
37        System.out.println("Error Listing Attached Terminals: " +
              e.toString());
38      }
39      ListIterator terminalIterator = listTerminal.listIterator();
40      while (terminalIterator.hasNext()) {
41        terminalIterator.next();
42      }
43      myCardTerminal = myCardTerminals.getTerminal("OMNIKEY CardMan 3x21
            0");
44      try {
45        try {
46          myCard = myCardTerminal.connect("T=1");
47        } catch (Exception e) {
48          System.out.println("Terminal Disconnected");
49        }
50        myCardChannel = myCard.getBasicChannel();
51        if (ResponseTest(myCardChannel.transmit(SELECT_APDU))){}
52         else {
53          System.out.println("Application Not Selected");
54        }
55      } catch (Exception eX) {
56        System.out.println("Error" + eX.getClass() + "\n" +
              eX.getMessage());
57      }
58    }
59    private boolean ResponseTest(ResponseAPDU resAPDU) {
60      byte[] testBytes = resAPDU.getBytes();
61      return (testBytes[testBytes.length - 2] == (byte)0x90 &&
62              testBytes[testBytes.length - 1] == (byte)0x00);
63    }
64    public static String byteToString(byte[] inArray) {
65      byte[] HEX_CHAR_TABLE =  {
66        (byte)'0', (byte)'1', (byte)'2', (byte)'3', (byte)'4', (byte)'5',
              (byte)
67          '6', (byte)'7', (byte)'8', (byte)'9', (byte)'a', (byte)'b',
                (byte)'c',
68        (byte)'d', (byte)'e', (byte)'f'
69      };
70      byte[] hex = new byte[2 * inArray.length];
```

```
71      int index = 0;
72      for (byte b: inArray) {
73        int v = b & 0xFF;
74        hex[index++] = HEX_CHAR_TABLE[v  >>> 4];
75        hex[index++] = HEX_CHAR_TABLE[v & 0xF];
76      }
77      try {
78        return new String(hex, "ASCII").replaceAll(".(?!$).(?!$)", "$0 ");
79      } catch (Exception cE) {
80        System.out.println("Exception in bytesToString : " +
              cE.getMessage());
81      }
82      return "Error";
83    }
84    public static void main(String[] args) {
85      try {
86        String myFile = "C:\\SCTP-SP_Data\\PerformanceSPInitiator.txt";
87        FileOutputStream outStream = new FileOutputStream(myFile);
88        PrintWriter out = new PrintWriter(outStream);
89        int iterator = 1000;
90        int counter = 1;
91        while (iterator > 0) {
92          Terminal myTerminal = new Terminal();
93          myTerminal.TerminalConnection();
94          myTerminal.startProtocolSPInitiaror();
95          System.out.println("ITERATION NUMBER : " + counter + " SPEED : " +
96                             (double)((myTerminal.protocolEndTime -
97                             myTerminal.protocolStartTime)));
98          iterator--;
99          counter++;
100         out.println((double)((myTerminal.protocolEndTime -
101                    myTerminal.protocolStartTime)));
102         myTerminal = null;
103       }
104       out.close();
105     } catch (Exception cE) {
106       System.out.println("Error : Error " + cE.getMessage());
107     }
108   }
109   public void cardTerminalCommunicator(CommandAPDU commandApduMsg) {
110     try {
111       response = myCardChannel.transmit(commandApduMsg).getBytes();
112     } catch (CardException cE) {
113       System.out.println(cE.getMessage());
114     }
115   }
116   public void startProtocolSCInitiator() {
117     CommandAPDU commandApduMsg;
118     this.myProtocolHanlderSCIn.initialiseProtocol();
119     try {
```

```
120       response =
              myCardChannel.transmit(Application_INITIALISATION).getBytes();
121       protocolStartTime = System.currentTimeMillis();
122       this.myProtocolHanlderSCIn.inMessageProcessing(response, 1);
123       commandApduMsg = new CommandAPDU(0xB0, 0x44, 0x00, 0x00,
124                           this.myProtocolHanlderSCIn.outMessageProcessing(1));
125       cardTerminalCommunicator(commandApduMsg);
126       if (this.myProtocolHanlderSCIn.inMessageProcessing(response, 2)) {
127         commandApduMsg = new CommandAPDU(0xB0, 0x44, 0x00, 0x00,
128           this.myProtocolHanlderSCIn.outMessageProcessing(2));
129         if (ResponseTest(myCardChannel.transmit(commandApduMsg))){}
130          else {
131           System.out.println("Error in Protocol");
132           System.exit(0);
133         }
134       }
135       protocolEndTime = System.currentTimeMillis();
136     } catch (Exception ce) {
137       System.out.println("Error in Terminal.startProtocolSCInitiator : " +
138                           ce.getMessage());
139     }
140   }
141   public void startProtocolSPInitiaror() {
142     CommandAPDU commandApduMsg;
143     myProtocolHandler.initialiseProtocol();
144     try {
145       myCardChannel.transmit(Application_INITIALISATION);
146       protocolStartTime = System.currentTimeMillis();
147       commandApduMsg = new CommandAPDU(0xB0, 0x44, 0x00, 0x00,
148                                         myProtocolHandler.outMessageProcessing(
149                                         (short)1));
150       cardTerminalCommunicator(commandApduMsg);
151       if (this.response[3] == (byte)0xAA) {
152         if (myProtocolHandler.inMessageProcessing(response, (short)1)) {
153           commandApduMsg = new CommandAPDU(0xB0, 0x44, 0x00, 0x00,
154             myProtocolHandler.outMessageProcessing((short)2));
155           cardTerminalCommunicator(commandApduMsg);
156         }
157       }
158       if (this.response[3] == (byte)0xBB) {
159         if (myProtocolHandler.inMessageProcessing(response, (short)2)){}
160       }
161       protocolEndTime = System.currentTimeMillis();
162     } catch (Exception cE) {
163       System.out.println("Exception in Terminal.startProtocol : " +
              cE.getClass
164                           ().getName());
165     }
166     try {
167       myCard.disconnect(true);
168     } catch (CardException cE) {
```

```
169         System.out.println(cE.getMessage());
170       }
171     }
172 }
```

C.11.3 Diffie-Hellman Group

The Diffie-Hellman group used by the SPs, and SCs in this thesis is listed as below:

```
1  package {Which ever protocol is using this DH group};
2
3  public class ClassDH
4    {
5    public byte[] dhBase =  {(byte)0xAC,
6        (byte)0x40, (byte)0x32, (byte)0xEF, (byte)0x4F, (byte)0x2D,
7        (byte)0x9A, (byte)0xE3, (byte)0x9D, (byte)0xF3, (byte)0x0B,
8        (byte)0x5C, (byte)0x8F, (byte)0xFD, (byte)0xAC, (byte)0x50,
9        (byte)0x6C, (byte)0xDE, (byte)0xBE, (byte)0x7B, (byte)0x89,
10       (byte)0x99, (byte)0x8C, (byte)0xAF, (byte)0x74, (byte)0x86,
11       (byte)0x6A, (byte)0x08, (byte)0xCF, (byte)0xE4, (byte)0xFF,
12       (byte)0xE3, (byte)0xA6, (byte)0x82, (byte)0x4A, (byte)0x4E,
13       (byte)0x10, (byte)0xB9, (byte)0xA6, (byte)0xF0, (byte)0xDD,
14       (byte)0x92, (byte)0x1F, (byte)0x01, (byte)0xA7, (byte)0x0C,
15       (byte)0x4A, (byte)0xFA, (byte)0xAB, (byte)0x73, (byte)0x9D,
16       (byte)0x77, (byte)0x00, (byte)0xC2, (byte)0x9F, (byte)0x52,
17       (byte)0xC5, (byte)0x7D, (byte)0xB1, (byte)0x7C, (byte)0x62,
18       (byte)0x0A, (byte)0x86, (byte)0x52, (byte)0xBE, (byte)0x5E,
19       (byte)0x90, (byte)0x01, (byte)0xA8, (byte)0xD6, (byte)0x6A,
20       (byte)0xD7, (byte)0xC1, (byte)0x76, (byte)0x69, (byte)0x10,
21       (byte)0x19, (byte)0x99, (byte)0x02, (byte)0x4A, (byte)0xF4,
22       (byte)0xD0, (byte)0x27, (byte)0x27, (byte)0x5A, (byte)0xC1,
23       (byte)0x34, (byte)0x8B, (byte)0xB8, (byte)0xA7, (byte)0x62,
24       (byte)0xD0, (byte)0x52, (byte)0x1B, (byte)0xC9, (byte)0x8A,
25       (byte)0xE2, (byte)0x47, (byte)0x15, (byte)0x04, (byte)0x22,
26       (byte)0xEA, (byte)0x1E, (byte)0xD4, (byte)0x09, (byte)0x93,
27       (byte)0x9D, (byte)0x54, (byte)0xDA, (byte)0x74, (byte)0x60,
28       (byte)0xCD, (byte)0xB5, (byte)0xF6, (byte)0xC6, (byte)0xB2,
29       (byte)0x50, (byte)0x71, (byte)0x7C, (byte)0xBE, (byte)0xF1,
30       (byte)0x80, (byte)0xEB, (byte)0x34, (byte)0x11, (byte)0x8E,
31       (byte)0x98, (byte)0xD1, (byte)0x19, (byte)0x52, (byte)0x9A,
32       (byte)0x45, (byte)0xD6, (byte)0xF8, (byte)0x34, (byte)0x56,
33       (byte)0x6E, (byte)0x30, (byte)0x25, (byte)0xE3, (byte)0x16,
34       (byte)0xA3, (byte)0x30, (byte)0xEF, (byte)0xBB, (byte)0x77,
35       (byte)0xA8, (byte)0x6F, (byte)0x0C, (byte)0x1A, (byte)0xB1,
36       (byte)0x5B, (byte)0x05, (byte)0x1A, (byte)0xE3, (byte)0xD4,
37       (byte)0x28, (byte)0xC8, (byte)0xF8, (byte)0xAC, (byte)0xB7,
38       (byte)0x0A, (byte)0x81, (byte)0x37, (byte)0x15, (byte)0x0B,
39       (byte)0x8E, (byte)0xEB, (byte)0x10, (byte)0xE1, (byte)0x83,
40       (byte)0xED, (byte)0xD1, (byte)0x99, (byte)0x63, (byte)0xDD,
41       (byte)0xD9, (byte)0xE2, (byte)0x63, (byte)0xE4, (byte)0x77,
42       (byte)0x05, (byte)0x89, (byte)0xEF, (byte)0x6A, (byte)0xA2,
43       (byte)0x1E, (byte)0x7F, (byte)0x5F, (byte)0x2F, (byte)0xF3,
```

```
44      (byte)0x81, (byte)0xB5, (byte)0x39, (byte)0xCC, (byte)0xE3,
45      (byte)0x40, (byte)0x9D, (byte)0x13, (byte)0xCD, (byte)0x56,
46      (byte)0x6A, (byte)0xFB, (byte)0xB4, (byte)0x8D, (byte)0x6C,
47      (byte)0x01, (byte)0x91, (byte)0x81, (byte)0xE1, (byte)0xBC,
48      (byte)0xFE, (byte)0x94, (byte)0xB3, (byte)0x02, (byte)0x69,
49      (byte)0xED, (byte)0xFE, (byte)0x72, (byte)0xFE, (byte)0x9B,
50      (byte)0x6A, (byte)0xA4, (byte)0xBD, (byte)0x7B, (byte)0x5A,
51      (byte)0x0F, (byte)0x1C, (byte)0x71, (byte)0xCF, (byte)0xFF,
52      (byte)0x4C, (byte)0x19, (byte)0xC4, (byte)0x18, (byte)0xE1,
53      (byte)0xF6, (byte)0xEC, (byte)0x01, (byte)0x79, (byte)0x81,
54      (byte)0xBC, (byte)0x08, (byte)0x7F, (byte)0x2A, (byte)0x70,
55      (byte)0x65, (byte)0xB3, (byte)0x84, (byte)0xB8, (byte)0x90,
56      (byte)0xD3, (byte)0x19, (byte)0x1F, (byte)0x2B, (byte)0xFA};
57  public byte[] dhModulus =  {(byte)0xAD,
58      (byte)0x10, (byte)0x7E, (byte)0x1E, (byte)0x91, (byte)0x23,
59      (byte)0xA9, (byte)0xD0, (byte)0xD6, (byte)0x60, (byte)0xFA,
60      (byte)0xA7, (byte)0x95, (byte)0x59, (byte)0xC5, (byte)0x1F,
61      (byte)0xA2, (byte)0x0D, (byte)0x64, (byte)0xE5, (byte)0x68,
62      (byte)0x3B, (byte)0x9F, (byte)0xD1, (byte)0xB5, (byte)0x4B,
63      (byte)0x15, (byte)0x97, (byte)0xB6, (byte)0x1D, (byte)0x0A,
64      (byte)0x75, (byte)0xE6, (byte)0xFA, (byte)0x14, (byte)0x1D,
65      (byte)0xF9, (byte)0x5A, (byte)0x56, (byte)0xDB, (byte)0xAF,
66      (byte)0x9A, (byte)0x3C, (byte)0x40, (byte)0x7B, (byte)0xA1,
67      (byte)0xDF, (byte)0x15, (byte)0xEB, (byte)0x3D, (byte)0x68,
68      (byte)0x8A, (byte)0x30, (byte)0x9C, (byte)0x18, (byte)0x0E,
69      (byte)0x1D, (byte)0xE6, (byte)0xB8, (byte)0x5A, (byte)0x12,
70      (byte)0x74, (byte)0xA0, (byte)0xA6, (byte)0x6D, (byte)0x3F,
71      (byte)0x81, (byte)0x52, (byte)0xAD, (byte)0x6A, (byte)0xC2,
72      (byte)0x12, (byte)0x90, (byte)0x37, (byte)0xC9, (byte)0xED,
73      (byte)0xEF, (byte)0xDA, (byte)0x4D, (byte)0xF8, (byte)0xD9,
74      (byte)0x1E, (byte)0x8F, (byte)0xEF, (byte)0x55, (byte)0xB7,
75      (byte)0x39, (byte)0x4B, (byte)0x7A, (byte)0xD5, (byte)0xB7,
76      (byte)0xD0, (byte)0xB6, (byte)0xC1, (byte)0x22, (byte)0x07,
77      (byte)0xC9, (byte)0xF9, (byte)0x8D, (byte)0x11, (byte)0xED,
78      (byte)0x34, (byte)0xDB, (byte)0xF6, (byte)0xC6, (byte)0xBA,
79      (byte)0x0B, (byte)0x2C, (byte)0x8B, (byte)0xBC, (byte)0x27,
80      (byte)0xBE, (byte)0x6A, (byte)0x00, (byte)0xE0, (byte)0xA0,
81      (byte)0xB9, (byte)0xC4, (byte)0x97, (byte)0x08, (byte)0xB3,
82      (byte)0xBF, (byte)0x8A, (byte)0x31, (byte)0x70, (byte)0x91,
83      (byte)0x88, (byte)0x36, (byte)0x81, (byte)0x28, (byte)0x61,
84      (byte)0x30, (byte)0xBC, (byte)0x89, (byte)0x85, (byte)0xDB,
85      (byte)0x16, (byte)0x02, (byte)0xE7, (byte)0x14, (byte)0x41,
86      (byte)0x5D, (byte)0x93, (byte)0x30, (byte)0x27, (byte)0x82,
87      (byte)0x73, (byte)0xC7, (byte)0xDE, (byte)0x31, (byte)0xEF,
88      (byte)0xDC, (byte)0x73, (byte)0x10, (byte)0xF7, (byte)0x12,
89      (byte)0x1F, (byte)0xD5, (byte)0xA0, (byte)0x74, (byte)0x15,
90      (byte)0x98, (byte)0x7D, (byte)0x9A, (byte)0xDC, (byte)0x0A,
91      (byte)0x48, (byte)0x6D, (byte)0xCD, (byte)0xF9, (byte)0x3A,
92      (byte)0xCC, (byte)0x44, (byte)0x32, (byte)0x83, (byte)0x87,
93      (byte)0x31, (byte)0x5D, (byte)0x75, (byte)0xE1, (byte)0x98,
94      (byte)0xC6, (byte)0x41, (byte)0xA4, (byte)0x80, (byte)0xCD,
```

```
95        (byte)0x86, (byte)0xA1, (byte)0xB9, (byte)0xE5, (byte)0x87,
96        (byte)0xE8, (byte)0xBE, (byte)0x60, (byte)0xE6, (byte)0x9C,
97        (byte)0xC9, (byte)0x28, (byte)0xB2, (byte)0xB9, (byte)0xC5,
98        (byte)0x21, (byte)0x72, (byte)0xE4, (byte)0x13, (byte)0x04,
99        (byte)0x2E, (byte)0x9B, (byte)0x23, (byte)0xF1, (byte)0x0B,
100       (byte)0x0E, (byte)0x16, (byte)0xE7, (byte)0x97, (byte)0x63,
101       (byte)0xC9, (byte)0xB5, (byte)0x3D, (byte)0xCF, (byte)0x4B,
102       (byte)0xA8, (byte)0x0A, (byte)0x29, (byte)0xE3, (byte)0xFB,
103       (byte)0x73, (byte)0xC1, (byte)0x6B, (byte)0x8E, (byte)0x75,
104       (byte)0xB9, (byte)0x7E, (byte)0xF3, (byte)0x63, (byte)0xE2,
105       (byte)0xFF, (byte)0xA3, (byte)0x1F, (byte)0x71, (byte)0xCF,
106       (byte)0x9D, (byte)0xE5, (byte)0x38, (byte)0x4E, (byte)0x71,
107       (byte)0xB8, (byte)0x1C, (byte)0x0A, (byte)0xC4, (byte)0xDF,
108       (byte)0xFE, (byte)0x0C, (byte)0x10, (byte)0xE6, (byte)0x4F};
109 }
```

C.11.4 SHA256 Pseudorandom Number Generator

The Pseudorandom Number Generator (PRNG) algorithm used during the attestation mechanism is based on the SHA 256 and its implementation is listed as below:

```
1  package prngSHA256;
2
3  import javacard.framework.Applet;
4  import javacard.framework.ISO7816;
5  import javacard.framework.ISOException;
6  import javacard.framework.JCSystem;
7  import javacard.framework.Util;
8  import javacard.security.MessageDigest;
9  public class PrngSHA256 extends Applet {
10   private static byte[] CyclicSeedFile = {
11     (byte)0x49, (byte)0x29, (byte)0x8e, (byte)0x5f, (byte)0xd3, (byte)
12       0x61, (byte)0xc9, (byte)0xd2, (byte)0x88, (byte)0x4c, (byte)0xfa,
13       (byte)0xd5, (byte)0xcb, (byte)0x9f, (byte)0x93, (byte)0x91,
14       (byte)0x26, (byte)0xba, (byte)0x65, (byte)0xd0, (byte)0x0c,
15       (byte)0x7c, (byte)0x5e, (byte)0x74, (byte)0x92, (byte)0x00,
16       (byte)0x47, (byte)0xa5, (byte)0x74, (byte)0x44, (byte)0xe1,
17       (byte)0xc2, (byte)0x14, (byte)0x9e, (byte)0xff, (byte)0xe8,
18       (byte)0x77, (byte)0x62, (byte)0x95, (byte)0x0b, (byte)0x10,
19       (byte)0x5d, (byte)0xf8, (byte)0x66, (byte)0x12, (byte)0x12,
20       (byte)0x79, (byte)0x9d, (byte)0x83, (byte)0xbf, (byte)0x74,
21       (byte)0xae, (byte)0xd2, (byte)0x45, (byte)0xf9, (byte)0x01,
22       (byte)0x54, (byte)0x22, (byte)0xbb, (byte)0x39, (byte)0xf8,
23       (byte)0xf0, (byte)0xe2, (byte)0x4e, (byte)0xec, (byte)0x2f,
24       (byte)0x26, (byte)0x38, (byte)0x95, (byte)0x06, (byte)0x79,
25       (byte)0x0d, (byte)0x4c, (byte)0xdb, (byte)0x58, (byte)0x12,
26       (byte)0xeb, (byte)0xf2, (byte)0xee, (byte)0x92, (byte)0xde,
27       (byte)0x9f, (byte)0x51, (byte)0x8a, (byte)0xb4, (byte)0x1a,
28       (byte)0xcb, (byte)0x46, (byte)0x84, (byte)0x8a, (byte)0x28,
29       (byte)0xa4, (byte)0x15, (byte)0xdf, (byte)0x21, (byte)0xa6,
30       (byte)0xcd, (byte)0x88, (byte)0xdb, (byte)0x01, (byte)0x07,
```

```
31      (byte)0xb3, (byte)0xd8, (byte)0x04, (byte)0x9c, (byte)0xdd,
32      (byte)0x55, (byte)0x3e, (byte)0x4a, (byte)0xf0, (byte)0x00,
33      (byte)0xb9, (byte)0x8e, (byte)0x85, (byte)0x4d, (byte)0x36,
34      (byte)0x7d, (byte)0xef, (byte)0x40, (byte)0xa0, (byte)0x66,
35      (byte)0x18, (byte)0xcb, (byte)0x43, (byte)0x59, (byte)0xfa,
36      (byte)0x64, (byte)0x01, (byte)0xda, (byte)0x34, (byte)0x7d,
37      (byte)0xcd, (byte)0x40, (byte)0x14, (byte)0xc4, (byte)0xd6,
38      (byte)0x50, (byte)0x05, (byte)0x52, (byte)0x5e, (byte)0x67,
39      (byte)0xec, (byte)0xa6, (byte)0xef, (byte)0x34, (byte)0x71,
40      (byte)0xb3, (byte)0x9a, (byte)0x87, (byte)0xc3, (byte)0xa9,
41      (byte)0xe9, (byte)0xc7, (byte)0x0b, (byte)0xb6, (byte)0xfd,
42      (byte)0xbc, (byte)0xb5, (byte)0x8d, (byte)0x21, (byte)0xde,
43      (byte)0x44, (byte)0x27, (byte)0xf7, (byte)0xd0, (byte)0xd2,
44      (byte)0x67, (byte)0xac, (byte)0x00, (byte)0xbb, (byte)0x2b,
45      (byte)0xa4, (byte)0x1a, (byte)0x7f, (byte)0x82, (byte)0x85,
46      (byte)0x23, (byte)0x5f, (byte)0x13, (byte)0x27, (byte)0x0d,
47      (byte)0x78, (byte)0x59, (byte)0xab, (byte)0xa5, (byte)0xd0,
48      (byte)0x96, (byte)0x1a, (byte)0x11, (byte)0x8f, (byte)0x6e,
49      (byte)0x87, (byte)0x33, (byte)0x0f, (byte)0x20, (byte)0xec,
50      (byte)0x61, (byte)0x31, (byte)0x79, (byte)0xd9, (byte)0x36,
51      (byte)0x1c, (byte)0xa6, (byte)0xd7, (byte)0x2a, (byte)0xdc,
52      (byte)0x3a, (byte)0x9d, (byte)0xdb, (byte)0xf5, (byte)0x77,
53      (byte)0x95, (byte)0x79, (byte)0xdf, (byte)0xe4, (byte)0x0b,
54      (byte)0x7d, (byte)0xbc, (byte)0xd0, (byte)0xc5, (byte)0xe8,
55      (byte)0x29, (byte)0x22, (byte)0x8a, (byte)0x52, (byte)0xf1,
56      (byte)0x02, (byte)0x9e, (byte)0x06, (byte)0x3b, (byte)0x73,
57      (byte)0x28, (byte)0xdd, (byte)0xbc, (byte)0xe7, (byte)0x7b,
58      (byte)0xd3, (byte)0xb6, (byte)0xc2, (byte)0x25, (byte)0x33,
59      (byte)0x14, (byte)0xdb, (byte)0x49, (byte)0x06, (byte)0xbe,
60      (byte)0xd8, (byte)0x38, (byte)0xff, (byte)0x59, (byte)0xfe,
61      (byte)0x7e, (byte)0x5b, (byte)0x9f, (byte)0x87, (byte)0x0b,
62      (byte)0x05, (byte)0x0a, (byte)0xcd, (byte)0x21, (byte)0xfb,
63      (byte)0x58, (byte)0xf6, (byte)0x57, (byte)0xb0, (byte)0x12,
64      (byte)0xc2, (byte)0xe8, (byte)0x8b, (byte)0x87, (byte)0x42,
65      (byte)0xf6, (byte)0x03, (byte)0x43, (byte)0x4c, (byte)0x96,
66      (byte)0x3a, (byte)0x37, (byte)0xac, (byte)0x06, (byte)0x3a,
67      (byte)0x6a, (byte)0xf0, (byte)0x92, (byte)0xf2, (byte)0x48,
68      (byte)0x77, (byte)0x0c, (byte)0xe4, (byte)0x1f, (byte)0x8c,
69      (byte)0xff, (byte)0x58, (byte)0x70, (byte)0x00, (byte)0x1b,
70      (byte)0xb6, (byte)0x0d, (byte)0x65, (byte)0x2f, (byte)0x53,
71      (byte)0xcd, (byte)0xb6, (byte)0xc4, (byte)0x2f, (byte)0x63,
72      (byte)0x3f, (byte)0x5f, (byte)0x47, (byte)0x63, (byte)0x92,
73      (byte)0xce, (byte)0x7b, (byte)0x59, (byte)0x01, (byte)0x8b,
74      (byte)0x9a, (byte)0xe8, (byte)0xfd, (byte)0xe6, (byte)0x61,
75      (byte)0xb2, (byte)0x88, (byte)0x9c, (byte)0x4e, (byte)0x18,
76      (byte)0xd4, (byte)0xca, (byte)0xbd, (byte)0x02, (byte)0x3e,
77      (byte)0x06, (byte)0xd4, (byte)0xa7, (byte)0x81, (byte)0xae,
78      (byte)0x11, (byte)0x9c, (byte)0x6c, (byte)0xae, (byte)0x97,
79      (byte)0xd5, (byte)0x55, (byte)0x1c, (byte)0x16, (byte)0x74,
80      (byte)0x67, (byte)0x44, (byte)0xf9, (byte)0xfd, (byte)0xd6,
81      (byte)0xad, (byte)0xff, (byte)0x35, (byte)0xcc, (byte)0x69,
```

```
82      (byte)0x14, (byte)0xc9, (byte)0xe6, (byte)0x44, (byte)0x4b,
83      (byte)0x10, (byte)0xff, (byte)0x98, (byte)0x8f, (byte)0x60,
84      (byte)0x02, (byte)0x4a, (byte)0x44, (byte)0x60, (byte)0x5c,
85      (byte)0x2b, (byte)0xe5, (byte)0x2e, (byte)0x0a, (byte)0x49,
86      (byte)0x98, (byte)0x5e, (byte)0x75, (byte)0x6a, (byte)0xde,
87      (byte)0xd9, (byte)0x42, (byte)0xda, (byte)0x2f, (byte)0xbd,
88      (byte)0xcd, (byte)0xfb, (byte)0xbd, (byte)0x03, (byte)0x00,
89      (byte)0x4b, (byte)0xa9, (byte)0x40, (byte)0x4a, (byte)0x5a,
90      (byte)0xa7, (byte)0x98, (byte)0x77, (byte)0xbb, (byte)0x0a,
91      (byte)0x28, (byte)0xec, (byte)0x14, (byte)0x5c, (byte)0xa6,
92      (byte)0x47, (byte)0xd0, (byte)0xf4, (byte)0x42, (byte)0xb5,
93      (byte)0x81, (byte)0x20, (byte)0x79, (byte)0xff, (byte)0x2b,
94      (byte)0xe7, (byte)0xc6, (byte)0x95, (byte)0x96, (byte)0xe4,
95      (byte)0x45, (byte)0xf1, (byte)0x10, (byte)0xe0, (byte)0x12,
96      (byte)0xcf, (byte)0xb5, (byte)0x3a, (byte)0x99, (byte)0x66,
97      (byte)0x8c, (byte)0x6b, (byte)0xb6, (byte)0x7c, (byte)0xab,
98      (byte)0x38, (byte)0x63, (byte)0x72, (byte)0x22, (byte)0x14,
99      (byte)0x2d, (byte)0x4c, (byte)0x87, (byte)0x86, (byte)0x89,
100     (byte)0x8d, (byte)0xf5, (byte)0x53, (byte)0xa1, (byte)0x02,
101     (byte)0x6b, (byte)0xd4, (byte)0xa3, (byte)0xce, (byte)0x7b,
102     (byte)0x56, (byte)0x06, (byte)0x19, (byte)0x0b, (byte)0x4f,
103     (byte)0x74, (byte)0x03, (byte)0x8d, (byte)0x51, (byte)0x7a,
104     (byte)0xb8, (byte)0xe0, (byte)0xdc, (byte)0x2a, (byte)0x26,
105     (byte)0xdd, (byte)0xff, (byte)0x3e, (byte)0x23, (byte)0xe5,
106     (byte)0x9b, (byte)0x2f, (byte)0xc8, (byte)0x6c, (byte)0x25,
107     (byte)0x60, (byte)0xd7, (byte)0x33, (byte)0x95, (byte)0xca,
108     (byte)0xaf, (byte)0x0c, (byte)0x7f, (byte)0x3f, (byte)0x95,
109     (byte)0x09, (byte)0xe8, (byte)0xd5, (byte)0x64, (byte)0x8c,
110     (byte)0x82, (byte)0x12, (byte)0x7e, (byte)0x68, (byte)0x0e,
111     (byte)0xb5, (byte)0xd0, (byte)0x15, (byte)0x85, (byte)0x72,
112     (byte)0x6b, (byte)0x3c, (byte)0xc6, (byte)0x17, (byte)0x7a,
113     (byte)0x3c, (byte)0x4a, (byte)0xba, (byte)0x71, (byte)0xa4,
114     (byte)0x30, (byte)0x26, (byte)0xfe, (byte)0x1c, (byte)0x21,
115     (byte)0x44, (byte)0x46, (byte)0xbc, (byte)0x90, (byte)0x15,
116     (byte)0x77, (byte)0x22, (byte)0x54, (byte)0x60, (byte)0x02,
117     (byte)0x81, (byte)0xe2, (byte)0x5b, (byte)0x98, (byte)0x9f,
118     (byte)0xfe, (byte)0x18, (byte)0xcd, (byte)0x3d, (byte)0x72,
119     (byte)0xc0, (byte)0x67, (byte)0x7b, (byte)0x7c, (byte)0x26,
120     (byte)0x09, (byte)0x45, (byte)0xa5, (byte)0x2c, (byte)0x5f,
121     (byte)0x63, (byte)0x9f, (byte)0x2f, (byte)0xc3, (byte)0x05,
122     (byte)0x07, (byte)0xbe
123 };
124 private final short FILE_SIZE = (short)540;
125 private final byte RECORD_SIZE = (byte)54;
126 private static short CyclicRecordReadPointer = 54;
127 private static short CyclicRecordWritePointer = 0;
128 final static byte CLA = (byte)0xB0;
129 final static byte GETRND = (byte)0x40;
130 final static short SW_CLASSNOTSUPPORTED = 0x6320;
131 final static short SW_ERROR_INS = 0x6300;
132 byte[] Buffer = JCSystem.makeTransientByteArray((short)32,
```

```
133     JCSystem.CLEAR_ON_DESELECT);
134   MessageDigest SHA256;
135   void Xor(byte[] inputBuffer, byte condition) {
136     if (condition == 1) {
137       for (short i = 1, j = 0; i <= RECORD_SIZE + 1; i++) {
138         inputBuffer[i] = (byte)(inputBuffer[i] ^ Buffer[j]);
139         if (++j >= (short)32) {
140           j = 0;
141         }
142       }
143     } else {
144       for (short i = 1; i <= RECORD_SIZE; i++) {
145         inputBuffer[(short)i] = (byte)(inputBuffer[(short)i] ^
146                       inputBuffer[(byte)(RECORD_SIZE)]);
147       }
148     }
149   }
150   private void CyclicSeedFileRead(byte[] readBuffer) {
151     if (CyclicRecordReadPointer >= FILE_SIZE) {
152       CyclicRecordReadPointer = 0;
153     } Util.arrayCopyNonAtomic(CyclicSeedFile, CyclicRecordReadPointer,
154                               readBuffer, (short)1, (short)
155                               (RECORD_SIZE + 1));
156     CyclicRecordReadPointer = (short)((short)CyclicRecordReadPointer
157                                + RECORD_SIZE);
158   }
159   private void CyclicSeedFileWrite(byte[] writeBuffer) {
160     if (CyclicRecordWritePointer >= FILE_SIZE) {
161       CyclicRecordWritePointer = 0;
162     } Util.arrayCopyNonAtomic(CyclicSeedFile,
163                               CyclicRecordWritePointer, writeBuffer,
164                               (short)(RECORD_SIZE + 1), RECORD_SIZE);
165     Xor(writeBuffer, (byte)0);
166     Util.arrayCopyNonAtomic(writeBuffer, (byte)1, CyclicSeedFile,
167                             CyclicRecordWritePointer, RECORD_SIZE);
168     CyclicRecordWritePointer = (short)((short)
169                                  CyclicRecordWritePointer +
170                                  RECORD_SIZE);
171   }
172   private void Adjuster(inputBuffer) {
173     for (short i = 0; i < 16; i++) {
174       inputBuffer[(short)i] = (byte)(Buffer[(short)i] ^ Buffer[(short)(31
175               -i)]);
176     }
177   }
178
179
180   private PrngSHA256(byte bArray[], short bOffset, byte bLength) {
181     SHA256 = MessageDigest.getInstance(MessageDigest.ALG_SHA_256,
182       false);
183     register();
```

```
184   }
185   public static void install(byte bArray[], short bOffset, byte
186                              bLength) {
187     new PrngSHA256(bArray, bOffset, bLength);
188   }
189
190   byte[] generateMACPrng(){
191     apduBuffer[0] = (byte)0x02;
192     CyclicSeedFileRead(apduBuffer);
193     SHA256.doFinal(apduBuffer, (short)0, (short)(RECORD_SIZE + 1),
194                    Buffer, (short)0);
195     Xor(apduBuffer, (byte)1);
196     apduBuffer[0] = (byte)0x03;
197     SHA256.doFinal(apduBuffer, (short)0, (short)(RECORD_SIZE + 1),
198                    Buffer, (short)0);
199     Xor(apduBuffer, (byte)1);
200     CyclicSeedFileWrite(apduBuffer);
201     Adjuster();
202   }
203   byte[] generateMACPrng(byte[] InputBuffer){
204     CyclicSeedFileRead(InputBuffer);
205     SHA256.doFinal(inputBuffer, (short)0, (short)(RECORD_SIZE + 1),
206                    Buffer, (short)0);
207       Adjuster(inputBuffer, Buffer);
208   }
209
210     byte[] generateMACPrng(byte[] InputBuffer, byte[] OutputBuffer){
211     CyclicSeedFileRead(InputBuffer);
212     SHA256.doFinal(inputBuffer, (short)0, (short)(RECORD_SIZE + 1),
213                    Buffer, (short)0);
214       Adjuster(OutputBuffer);
215   }
216 }
```

Bibliography

[1] "GlobalPlatform Card Security Requirement Specification 1.0," Online, Redwood City, USA, Specification, May 2003. Online Available: http://www.globalplatform.org/specificationscard.asp

[2] (Visited December, 2009) History of Plastic Cards in the UK. Online. The UK Cards Association. London, United Kingdom. Online Available: http://www.theukcardsassociation.org.uk/history_of_cards/

[3] (Visited March, 2011) History of the Card Payments System. Online. Master Card Worldwide. Online Available: http://www.mastercard.com/us/company/en/docs/history%20of%20payments.pdf

[4] J. Wonglimpiyarat, *Strategies of Competition in the Bank Card Business: Innovation Management in a Complex Economic Environment.* Brighton, United Kingdom: Sussex Academic Press, 2005.

[5] W. Rankl and W. Effing, *Smart Card Handbook*, 3rd ed. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[6] K. Mayes and K. Markantonakis, Eds., *Smart Cards, Tokens, Security and Applications.* Springer, 2008.

[7] *ISO/IEC 7813: Information Technology - Identification Cards - Financial Transaction Cards*, Online, International Organization for Standardization (ISO) Std., 2006. Online Available: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=43317

[8] W. C. Barker, D. Howard, T. Grance, and L. Eyuboglu, "NIST IR 7056: Card Technology Developments and Gap Analysis Interagency Report," Electronic, Gaithersburg, Maryland, United Kingdom, Recommedation, March 2004. Online Available: http://csrc.nist.gov/publications/nistir/nistir-7056.pdf

[9] *EMV 4.2 : Book 1 - Application Independent ICC to Terminal Interface Requirements, Book 2 - Security and Key Management, Book 3 - Application Specification, Book 4 - Cardholder, Attendant, and Acquirer Interface Requirements*, Online, EMVCo Specification 4.2, May 2008. Online Available: http://www.emvco.com/specifications.aspx?id=155

[10] R. N. Akram, K. Markantonakis, and K. Mayes, "Application Management Framework in User Centric Smart Card Ownership Model," in *The 10th International Workshop on Information Security Applications (WISA09)*, H. Y. YOUM and M. Yung, Eds., vol. 5932/2009. Busan, Korea: Springer,

August 2009, pp. 20–35. Online Available: http://www.springerlink.com/content/f7027021h1067261/fulltext.pdf

[11] D. Sauveron, "Multiapplication Smart Card: Towards an Open Smart Card?" *Inf. Secur. Tech. Rep.*, vol. 14, no. 2, pp. 70–78, 2009.

[12] W. Atkins, *The Smart Card Report*, 8th ed. Elsevier, January 2004.

[13] E. Brack, "Déterminants du Développement du Porte-Monnaie Électronique: Analyse Théorique et Empirique: Exemple Moneo, English Title: Electronic Wallet Development Determinants: Theoretical and Empirical Analysis: Moneo," University Library of Munich, Germany, MPRA Paper 23453, 2003. Online Available: http://ideas.repec.org/p/pra/mprapa/23453.html

[14] D. Deville, A. Galland, G. Grimaud, and S. Jean, "Smart Card Operating Systems: Past, Present and Future," in *In Proceedings of the 5 th NORDU/USENIX Conference*, 2003. Online Available: http://www.gemplus.com/smart/rd/publications/pdf/DGGJ03os.pdf

[15] K. Markantonakis, "The Case for a Secure Multi-Application Smart Card Operating System," in *ISW '97: Proceedings of the First International Workshop on Information Security*. London, UK: Springer-Verlag, 1998, pp. 188–197. Online Available: http://www.springerlink.com/content/w62286334532m71n/fulltext.pdf

[16] *Java Card Platform Specification: Classic Edition; Application Programming Interface, Runtime Environment Specification, Virtual Machine Specification, Connected Edition; Runtime Environment Specification, Java Servlet Specification, Application Programming Interface, Virtual Machine Specification, Sample Structure of Application Modules*, Oracle Std. Version 3.0.1, May 2009. Online Available: http://java.sun.com/javacard/3.0.1/specs.jsp

[17] *ISO/IEC 18092: Near Field Communication - Interface and Protocol (NFCIP-1)*, International Organization for Standardization (ISO) Std., April 2004.

[18] *Trusted Module Specification 1.2: Part 1- Design Principles, Part 2- Structures of the TPM, Part 3- Commands*, Trusted Computing Group Std., Rev. 103, July 2007. Online Available: http://www.trustedcomputinggroup.org/resources/tpm_specification_version_12_revision_103_part_1__3

[19] "TCG Mobile Trusted Module Specification," Trusted Computing Group (TCG), Specification Ver 1.0, June 2008.

[20] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," in *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2003, pp. 160–171.

[21] "ARM Security Technology: Building a Secure System using TrustZone Technology," ARM, White Paper PRD29-GENC-009492C, 2009.

[22] "M-Shield Mobile Security Technology: Making Wireless Secure," Texas Instruments, Whilte Paper, February 2008. Online Available: http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf

[23] "GlobalPlatform Device Technology: TEE System Architecture," GlobalPlatform, Specification Version 0.4, October 2011.

[24] *ISO/IEC 7816-5, "Information Technology - Identification cards - Integrated Circuit(s) cards with contacts - Part 5: Numbering systems and registration procedure for application identifiers, International Organization for Standardization*, International Organization for Standardization (ISO) Std., 2004. Online Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=34259

[25] *ISO/IEC 14443-1: Identification Cards - Contactless Integrated Circuit(s) Cards - Proximity Cards, Part1: Physical Characteristics, Part 2: Radio Frequency Power and Signal Interface, Part3: Initialization and Anticollision, Part 4: Transmission Protocol*, International Organization for Standardization (ISO) Std., Rev. 2nd Edition, June 2008. Online Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=28728

[26] *ETSI. Digital Cellular Telecommunications System*, European Telecommunications Standards Institution (ETSI) Std. Online Available: http://www.etsi.org/website/Technologies/gsm.aspx

[27] *ITSO Technical Specification 1000: Interoperable Public Transport Ticketing Using Contactless Smart Customer Media*, Online, Integrated Transport Smartcard Organisation Std., Rev. V2.1.4, February 2010. Online Available: http://www.itso.org.uk/page49/ITSO%20Specification

[28] *Java Card Platform Specification; Application Programming Interface, Runtime Environment Specification, Virtual Machine Specification*, Sun Microsystem Inc Std. Version 2.2.2, March 2006. Online Available: http://java.sun.com/javacard/specs.html

[29] *Multos: The Multos Specification,*, Online, Std. Online Available: http://www.multos.com/

[30] *GlobalPlatform: GlobalPlatform Card Specification, Version 2.2,*, GlobalPlatform Std., March 2006. Online Available: http://www.globalplatform.org/specificationscard.asp

[31] R. N. Akram, K. Markantonakis, and K. Mayes, "User Centric Security Model for Tamper-Resistant Devices," in *the 8th IEEE International Conference on e-Business Engineering (ICEBE 2011)*, J. Li and J.-Y. Chung, Eds. Beijing, China: IEEE Computer Science, October 2011.

[32] R. N. Akram, K. Markantonakis, and K. Mayes, "A Paradigm Shift in Smart Card Ownership Model," in *Proceedings of the 2010 International Conference on Computational Science and Its Applications (ICCSA 2010)*, B. O. Apduhan, O. Gervasi, A. Iglesias, D. Taniar, and M. Gavrilova, Eds. Fukuoka, Japan: IEEE Computer Society, March 2010, pp. 191–200.

[33] A. M. Brandenburger and B. J. Nalebuff, *Co-Opetition : A Revolution Mindset That Combines Competition and Cooperation : The Game Theory Strategy That's Changing the Game of Business*, 1st ed. Doubleday Business, Dec. 1997. Online

Available: http://www.worldcat.org/isbn/0385479506

[34] M' Chirgui, Zouhaier, "The Economics of the Smart Card Industry: Towards Coopetitive Strategies," *Economics of Innovation and New Technology*, vol. 14, no. 6, pp. 455–477, 2005. Online Available: http://www.informaworld.com/openurl?genre=article&doi=10.1080/10438590420003040 70&magic=crossref

[35] *Coopetition An Introduction to the Subject and an Agenda for Research*, vol. 37, no. 2, August 2007.

[36] "Trusted Computing Group, TCG Specification Architecture Overview," The Trusted Computing Group (TCG), Beaverton, Oregon, USA, revision 1.4, August 2007. Online Available: http://www.trustedcomputinggroup.org/files/resource_files/AC652DE1-1D09-3519-ADA026A0C05CFAC2/TCG_1_4_Architecture_Overview.pdf

[37] H. Kopetz, "Internet of Things," in *Real-Time Systems*, ser. Real-Time Systems Series. Springer US, 2011, pp. 307–323.

[38] M. E. Porter, "How Competitive Forces Shape Strategy," *Harvard Business Review*, vol. 57, no. 2, 1979. Online Available: http://search.ebscohost.com.ezproxy.libraries.claremont.edu/login.aspx?direct=true&db=buh&AN=3867673&site=ehost-live

[39] N. Mallat, "Exploring Consumer Adoption of Mobile Payments - A Qualitative Study," *J. Strateg. Inf. Syst.*, vol. 16, pp. 413–432, December 2007. Online Available: http://portal.acm.org/citation.cfm?id=1321790.1322013

[40] J. Laugesen and Y. Yuan, "What Factors Contributed to the Success of Apple's iPhone?" in *Proceedings of the 2010 Ninth International Conference on Mobile Business / 2010 Ninth Global Mobility Roundtable*, ser. ICMB-GMR '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 91–99.

[41] (Visited January, 2011) NFC Trials, Pilots, Tests and Live Services around the World. Online. NFC World. Online Available: http://www.nearfieldcommunicationsworld.com/list-of-nfc-trials-pilots-tests-and-commercial-services-around-the-world/

[42] "Pay-Buy-Mobile: Business Opportunity Analysis," GSM Association, White Paper 1.0, November 2007. Online Available: http://www.gsmworld.com/documents/gsma_nfc_tech_guide_vs1.pdf

[43] "EPC-GSMA Mobile Contactless Payments Service Management Roles Requirements and Specifications," European Payments Council (EPC) and GSM Association, Tech. Rep. EPC 220-08, October 2010.

[44] "The Role and Scope of EMVCo in Standardising the Mobile Payments Infrastructure," Online, EMVCo., California, USA, Tech. Rep., October 2007. Online Available: http://www.emvco.com/download_agreement.aspx?id=385

[45] "Framework for Smart card Use in Government," Foundation for Information Policy Research, Consultation Response, 1999. Online Available: http://www.cl.cam.ac.uk/~rja14/Papers/smartcards-fipr.pdf

[46] P. Girard, "Which Security Policy for Multiplication Smart Cards?" in *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on*

Smartcard Technology. Berkeley, CA, USA: USENIX Association, 1999, pp. 3–3. Online Available: http://portal.acm.org/citation.cfm?id=1267115.1267118

[47] C. K. Prahalad and G. Hamel, "The Core Competence of the Corporation," *Harvard Business Review*, 1990. Online Available: http://tle-inc.com/PDFS/FILES/resources/The%20Core%20Competencies%20of%20the%20Corp.pdf

[48] J. Vincent, "Affiliations, Emotion and the Mobile Phone," in *Cross-Modal Analysis of Speech, Gestures, Gaze and Facial Expressions*, ser. Lecture Notes in Computer Science, A. Esposito and R. Vích, Eds. Springer, 2009, vol. 5641, pp. 28–41.

[49] R. Ling, *New Tech, New Ties: How Mobile Communication Is Reshaping Social Cohesion*. The MIT Press, 2008.

[50] "GlobalPlatform Device: Secure Element Remote Application Management," Online, GlobalPlatform, Specification, February 2011.

[51] N. Seriot, "iPhone Privacy," in *Black Hat DC*, 2010.

[52] J. Winter, "Trusted computing building blocks for embedded linux-based ARM trustzone platforms," in *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, ser. STC '08. New York, NY, USA: ACM, 2008, pp. 21–30.

[53] S. R. White, "ABYSS: A Trusted Architecture for Software Protection," *Security and Privacy, IEEE Symposium on*, vol. 0, p. 38, 1987.

[54] *GlobalPlatform Device Technology: Device Application Security Management - Concepts and Description Document Specification*, Online, GlobalPlatform Specification, April 2008. Online Available: http://www.globalplatform.org/specificationsdevice.asp

[55] V. Costan, L. F. Sarmenta, M. Dijk, and S. Devadas, "The Trusted Execution Module: Commodity General-Purpose Trusted Computing," in *8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*. London, United Kingdom: Springer, 2008, pp. 133–148.

[56] R. N. Akram, K. Markantonakis, and K. Mayes, "A Dynamic and Ubiquitous Smart Card Security Assurance and Validation Mechanism," in *25th IFIP International Information Security Conference (SEC 2010)*, ser. IFIP AICT Series, K. Rannenberg and V. Varadharajan, Eds. Brisbane, Australia: Springer, September 2010, pp. 161–172.

[57] S. Peng and Z. Han, "Design and Implementation of Portable TPM Device Driver Based on Extensible Firmware Interface," *International Conference on Multimedia Information Networking and Security*, vol. 2, pp. 342–345, November 2009.

[58] K. Dietrich and J. Winter, "Implementation Aspects of Mobile and Embedded Trusted Computing," in *Trust '09: Proceedings of the 2nd International Conference on Trusted Computing*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 29–44.

[59] P. England and T. Tariq, "Towards a Programmable TPM," in *Trusted Computing*, ser. LNCS, L. Chen, C. Mitchell, and A. Martin, Eds. Springer Berlin / Heidelberg, 2009, vol. 5471, pp. 1–13.

[60] R. S. Pappu, "Physical One-way Functions," Ph.D. dissertation, Massachusetts

Institute of Technology, March 2001. Online Available: http://pubs.media.mit.edu/pubs/papers/01.03.pappuphd.powf.pdf

[61] G. Suh, C. O'Donnell, and S. Devadas, "Aegis: A Single-Chip Secure Processor," *Design Test of Computers*, vol. 24, no. 6, pp. 570 –580, December 2007.

[62] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves, "Implementing Embedded Security on Dual-Virtual-CPU Systems," *IEEE Design and Test of Computers*, vol. 24, pp. 582–591, 2007.

[63] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala, "On-board credentials with open provisioning," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ser. ASIACCS '09. New York, NY, USA: ACM, 2009, pp. 104–115.

[64] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold boot attacks on encryption keys," in *Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 45–60.

[65] "GlobalPlatform Device: GPD/STIP Specification Overview," GlobalPlatform, Specification Version 2.3, August 2007.

[66] F. C. Bormann, L. Manteau, A. Linke, J. C. Pailles, and J. D. van, "Concept for Trusted Personal Devices in a Mobile and Networked Environment," in *15th IST Mobile & Wireless Communications Summit*, June 2006. Online Available: http://doc.utwente.nl/59784/1/Bormann06concept.pdf

[67] S. Drimer, S. J. Murdoch, and R. Anderson, "Thinking Inside the Box: System-Level Failures of Tamper Proofing," in *IEEE Symposium on Security and Privacy*. USA: IEEE CS, 2008, pp. 281–295.

[68] R. N. Akram, K. Markantonakis, and K. Mayes, "Location Based Application Availability," in *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*, R. M. P. Herrero and T. Dillon, Eds., vol. 5872/2009. Vilamoura, Portugal: Springer, November 2009, pp. 128 – 138.

[69] *Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and General Model, Part 2: Security Functional Requirements, Part 3: Security Assurance Requirements,*, Common Criteria Std. Version 3.1, August 2006. Online Available: http://www.commoncriteriaportal.org/thecc.html

[70] S. Chaumette and D. Sauveron, "New Security Problems Raised by Open Multiapplication Smart Cards." *LaBRI, Université Bordeaux 1.*, pp. 1332–04, 2004.

[71] B. Parkinson and J. J. Spiker, *Global Positioning System: Theory and Applications*. AIAA, January 1996, vol. 1.

[72] (Visited June, 2010) London Underground: Oyster Card. London Underground. United Kingdom. Online Available: https://oyster.tfl.gov.uk/oyster/entry.do

[73] (Visited December, 2010) Octopus. Octopus Holdings Ltd. Hong Kong, China. Online Available: http://www.octopus.com.hk/home/en/index.html

[74] "The GlobalPlatform Proposition for NFC Mobile: Secure Element Management

and Messaging," GlobalPlatform, White Paper, April 2009. Online Available: http://www.globalplatform.org/documents/GlobalPlatform_NFC_Mobile_White_Paper.pdf

[75] "3D-Secure: Verified by Visa System Overview," Visa International Service Association, External Version 1.0.2, December 2006.

[76] S. J. Murdoch and R. J. Anderson, "Verified by Visa and MasterCard SecureCode: Or, How Not to Design Authentication," in *Financial Cryptography and Data Security, 14th International Conference, FC 2010*, ser. LNCS, R. Sion, Ed., vol. 6052. Springer, January 2010, pp. 336–342.

[77] R. Anderson, "Can We Fix the Security Economics of Federated Authentication?" in *SPW 2011, 19th International Workshop on Security Protocols*, J. A. Malcolm, Ed. London, UK: Springer, March 2011.

[78] S. Drimer, S. J. Murdoch, and R. J. Anderson, "Optimised to Fail: Card Readers for Online Banking," in *Financial Cryptography and Data Security, 13th International Conference, FC 2009*, R. Dingledine and P. Golle, Eds., vol. 5628. Barbados: Springer, February 2009, pp. 184–200.

[79] (Visited May, 2011) Sony's PlayStation Network Hack: When Did They Know? Online. PCMagazine. Online Available: http://www.pcmag.com/article2/0,2817,2384366,00.asp

[80] T. Moore, R. Clayton, and R. Anderson, "The Economics of Online Crime," *Journal of Economic Perspectives*, vol. 23, no. 3, pp. 3–20, Summer 2009.

[81] (Visited June, 2010) Barclaycard OnePulse. BarclayCard, Barclay Bank PLC. United Kingdom. Online Available: http://www.barclaycard-onepulse.co.uk

[82] Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[83] K. Markantonakis and K. Mayes, "A Secure Channel Protocol for Multi-application Smart Cards based on Public Key Cryptography," in *CMS 2004 - Eight IFIP TC-6-11 Conference on Communications and Multimedia Security*, D. Chadwick and B. Prennel, Eds. Springer, September 2004, pp. 79–96. Online Available: http://www.scc.rhul.ac.uk/public/A%20Secure%20Channel%20Protocol%20for%20Multi%20application%20smart%20cards%20based%20on%20PK%20Crytpography.pdf

[84] G. Barthe, G. Dufay, L. Jakubiec, and a. M. d. Sousa, Sim "A Formal Correspondence between Offensive and Defensive JavaCard Virtual Machines," in *VMCAI '02: Revised Papers from the Third International Workshop on Verification, Model Checking, and Abstract Interpretation*. London, UK: Springer-Verlag, 2002, pp. 32–45.

[85] R. N. Akram, K. Markantonakis, and K. Mayes, "Simulator Problem in User Centric Smart Card Ownership Model," in *6th IEEE/IFIP International Symposium on Trusted Computing and Communications (TrustCom-10)*, H. Y. Tang and X. Fu, Eds. HongKong, China: IEEE Computer Society, December 2010.

[86] D. Sauveron and P. Dusart, "Which Trust Can Be Expected of the Common Criteria Certification at End-User Level?" *Future Generation Communication and Network-*

ing, vol. 2, pp. 423–428, 2007.

[87] *Smartcard-Web-Server, Smartcard Web Server Enabler Architecture, Smartcard Web Server Requirements*, Open Mobile Alliance (OMA) Std., 2008. Online Available: http://www.openmobilealliance.org/technical/release_program/SCWS_v1_0.aspx

[88] *ISO/IEC 7816-3:2006, "Identification Cards – Integrated Circuit Cards – Part 3: Cards with Contacts – Electrical Interface and Transmission Protocols*, International Organization for Standardization (ISO) Std., 2006. Online Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38770

[89] *ISO/IEC 28361: Near Field Communication Wired Interface (NFC-WI)*, International Organization for Standardization (ISO) Std., October 2007. Online Available: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=44659&ics1=35&ics2=100&ics3=10

[90] Kevin Foster and Erik Meijer and Scott Schuh and Micahael A. Zabek, "The 2008 Survey of Consumer Payment Choice," Public Policy Discussion Papers No. 09-10, Federal Reserve Bank of Boston, USA, Tech. Rep., April 2010. Online Available: http://www.bos.frb.org/economic/ppdp/2009/ppdp0910.pdf

[91] Y. A. Au and R. J. Kauffman, "The Economics of Mobile Payments: Understanding Stakeholder Issues for an Emerging Financial Technology Application," *Electron. Commer. Rec. Appl.*, vol. 7, pp. 141–164, July 2008.

[92] "Information Technology Security Evaluation Criteria (ITSEC) - Provisional Harmonised Criteria," Office for Official Publications of the European Communities, Luxembourg, Brussels, Tech. Rep. COM(90) 314, June 1991. Online Available: http://www.ssi.gouv.fr/site_documents/ITSEC/ITSEC-uk.pdf

[93] "Multos: Version 4 on Hitachi AE45C Integrated Circuit Card," Uk IT Security Evaluation and Certification Scheme, Cheltenham, United Kingdom, Certification Report NO. P167, June 2002. Online Available: http://www.cesg.gov.uk/products_services/iacs/cc_and_itsec/media/certreps/CRP167.pdf

[94] T. Frane-Massey, "Multos - the High Security Smart Card OS," MAOSCO, Tech. Rep., September 2005. Online Available: http://www.multos.com/downloads/marketing/Whitepaper_MULTOS_Security.pdf

[95] "StepNexus: Multos International Division for Key Managnement Authority (KMA)," Visited July 2010. Online Available: http://www.multosinternational.com/services/stepnexus.aspx

[96] "Multos SmartDeck: The Software Development Kit for Multos," StepNexus Developer Tools, Visited July 2010. Online Available: http://www.stepdeveloper.com/

[97] "Multos: Guide to Loading and Deleting Applications," MAOSCO, Tech. Rep. MAO-DOC-TEC-008 v2.21, 2006. Online Available: http://www.multos.com/downloads/technical/glda.pdf

[98] B. du Castel. (Visted March, 2010) Personal History of the Java Card. Internet. Online Available: http://knol.google.com/k/bertrand-du-castel/

personal-history-of-the-java-card/3cjtq1rfm2r15/8

[99] "RFC 1122 - Requirements for Internet Hosts - Communication Layers," United States, Tech. Rep., 1989.

[100] T. Dierks and E. Rescorla, "RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2," Tech. Rep., August 2008. Online Available: http://tools.ietf.org/html/rfc5246

[101] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1," United States, Tech. Rep., 1999.

[102] E. Rescorla and A. Schiffman, "RFC 2660 - The Secure HyperText Transfer Protocol," United States, Tech. Rep., 1999.

[103] K. Markantonakis and K. Mayes, "An Overview of the GlobalPlatform Smart Card Specification," *Information Security Technical Report*, vol. 8, no. 1, pp. 17 – 29, 2003. Online Available: http://www.sciencedirect.com/science/article/B6VJC-48GF0G2-3/2/84de64208e223dea3f18a3e887c524ed

[104] "Security of Proximity Mobile Payments," Smart Card Alliance, 191 Clarksville Rd. Princeton Junction, NJ 08550, White Paper, May 2009. Online Available: http://www.smartcardalliance.org/resources/pdf/Security_of_Proximity_Mobile_Payments.pdf

[105] T. M. Jurgensen and S. Guthery, *The Smart Cards: A Developer's Toolkit*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[106] ECMA, *ECMA-107: Volume and File Structure of Disk Cartridges for Information Interchange*, 2nd ed. Geneva, Switzerland: European Association for Standardizing Information and Communication Systems (ECMA), June 1995. Online Available: http://www.ecma.ch/ecma1/STAND/ECMA-107.HTM

[107] *ISO/IEC 9293: Information Technology - Volume and File Structure of Disk Cartridges for Information Interchange*, Online, International Organization for Standardization (ISO) Standard, November 1994. Online Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=21273

[108] (Visited June, 2010) The Smart Card Deployment Cookbook . Microsoft. USA. Online Available: http://technet.microsoft.com/en-gb/library/dd277386.aspx

[109] (Visited April, 2011) Gemalto .Net Smart Card Solutions. Online. Gemalto. Online Available: http://www.protiva.gemalto.com/download/Gemalto.net.pdf

[110] "ZeitControl Card Systems GmbH." Online Available: http://www.basiccard.com

[111] W. Stallings, *Data and Computer Communications (8th Edition)*, 8th ed. Prentice Hall, August 2006. Online Available: http://www.worldcat.org/isbn/0132433109

[112] R. M. Cohen, "Defensive Java Virtual Machine Version 0.5 alpha," Online, May 1997. Online Available: http://www.computationallogic.com/software/djvm/

[113] *FIPS 140-2: Security Requirements for Cryptographic Modules*, Online, National Institute of Standards and Technology (NIST) Federal Information Processing Standards Publication, Rev. Supercedes FIPS PUB 140-1, May 2005. Online

Available: http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf

[114] (Visited September, 2011) Trusted Computing Group: Embedded Systems Work Group. Online. Trusted Compouting Group. Oregon, USA. Online Available: http://www.trustedcomputinggroup.org/developers/embedded_systems

[115] K. Eagles, K. Markantonakis, and K. Mayes, "A comparative analysis of common threats, vulnerabilities, attacks and countermeasures within smart card and wireless sensor network node technologies," in *Proceedings of the 1st IFIP TC6 /WG8.8 /WG11.2 international conference on Information security theory and practices: smart cards, mobile and ubiquitous computing systems*, ser. WISTP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 161–174. Online Available: http://ubiquinet.org/Files/Smart_Card_&_WSN_Node_Threat_Comparisons.pdf

[116] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon Physical Random Functions," in *Proceedings of the 9th ACM conference on Computer and communications security*, ser. CCS '02. New York, NY, USA: ACM, 2002, pp. 148–160.

[117] H. Busch, M. Sotáková, S. Katzenbeisser, and R. Sion, "The PUF promise," in *Proceedings of the 3rd international conference on Trust and trustworthy computing*, ser. TRUST'10. Berlin, Heidelberg: Springer-Verlag, June 2010, pp. 290–297. Online Available: http://portal.acm.org/citation.cfm?id=1875652.1875675

[118] D. Kirovski, "Anti-Counterfeiting: Mixing the Physical and the Digital World," in *Foundations for Forgery-Resilient Cryptographic Hardware*, ser. Dagstuhl Seminar Proceedings, J. Guajardo, B. Preneel, A.-R. Sadeghi, and P. Tuyls, Eds., no. 09282. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. Online Available: http://drops.dagstuhl.de/opus/volltexte/2010/2406

[119] S. S. Kumar, J. Guajardo, R. Maes, G.-J. Schrijen, and P. Tuyls, "Extended Abstract: The Butterfly PUF Protecting IP on every FPGA," in *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 67–70. Online Available: http://www.cosic.esat.kuleuven.be/publications/article-1154.pdf

[120] J. H. Anderson, "A PUF Design for Secure FPGA-based Embedded Systems," in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ser. ASPDAC '10. Piscataway, NJ, USA: IEEE Press, 2010, pp. 1–6.

[121] J. Guajardo, S. S. Kumar, G. J. Schrijen, and P. Tuyls, "Physical Unclonable Functions and Public-Key Crypto for FPGA IP Protection," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 2007, pp. 189–195.

[122] P. Tuyls, G.-J. Schrijen, B. Škorić, J. van Geloven, N. Verhaegh, and R. Wolters, "Read-proof Hardware from Protective Coatings," in *Cryptographic Hardware and Embedded Systems Workshop*, ser. LNCS, vol. 4249. Springer, October 2006, pp. 369–383. Online Available: http://www.springerlink.com/content/8454587207415662/fulltext.pdf

[123] G. E. Suh and S. Devadas, "Physical Unclonable Functions for Device Authentication and Secret Key Generation," in *Proceedings of the 44th annual Design Automation*

Conference, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 9–14.

[124] L. Bolotnyy and G. Robins, "Physically Unclonable Function-Based Security and Privacy in RFID Systems," in *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 211–220. Online Available: http://portal.acm.org/citation.cfm?id=1263542.1263714

[125] D. E. Lazich and M. Wuensche, "Protection of Sensitive Security Parameters in Integrated Circuits," in *Mathematical Methods in Computer Science*, J. Calmet, W. Geiselmann, and J. Müller-Quade, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 157–178.

[126] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Function," *SIGARCH Comput. Archit. News*, vol. 33, pp. 25–36, May 2005.

[127] D. Merli, D. Schuster, F. Stumpf, and G. Sigl, "Side-Channel Analysis of PUFs and Fuzzy Extractors," in *Trust and Trustworthy Computing*, ser. Lecture Notes in Computer Science, J. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, and Y. Beres, Eds. Springer Berlin / Heidelberg, 2011, vol. 6740, pp. 33–47, 10.1007/978-3-642-21599-5_3. Online Available: http://www.springerlink.com/content/h77526861527tg06/fulltext.pdf

[128] X. Leroy, "Bytecode verification on Java smart cards," *Softw. Pract. Exper.*, vol. 32, no. 4, pp. 319–340, 2002.

[129] *ISO/IEC 15408: Common Criteria for Information Technology Security Evaluation*, Std. Version 2.2, Rev. 256, 2004.

[130] "Common Methodology for Information Technology Security Evaluation; Evaluation Methodology," Tech. Rep. Version 3.1, July 2009. Online Available: http://www.commoncriteriaportal.org/thecc.html

[131] E. Roback, "Exploring Common Criteria: Can it Ensure that the Federal Government Gets Needed Security in Software?" *US govt. publication*, Sept. 2003.

[132] B. Schneier, *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. New York, NY, USA: John Wiley & Sons, Inc., 1995.

[133] A. Maiti, R. Nagesh, A. Reddy, and P. Schaumont, "Physical unclonable function and true random number generator: a compact and scalable implementation," in *Proceedings of the 19th ACM Great Lakes symposium on VLSI*, ser. GLSVLSI '09. New York, NY, USA: ACM, 2009, pp. 425–428.

[134] D. E. Holcomb, W. P. Burleson, and K. Fu, "Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers," *IEEE Transactions on Computers*, vol. 58, pp. 1198–1210, 2009.

[135] S. Schulz, C. Wachsmann, and A.-R. Sadeghis, "Lightweight Remote Attestation using Physical Functions," Technische Universitat Darmstadt, Darmstadt, Germany, Technical Report TR-2001-06-11, July 2011. Online Available: http://www.informatik.tu-darmstadt.de/fileadmin/user_upload/

Group_TRUST/PubsPDF/CASED-TR-2011-06-01.pdf

[136] S. Chari, V. V. Diluoffo, P. A. Karger, E. R. Palmer, T. Rabin, J. R. Rao, P. Rohatgi, H. Scherzer, M. Steiner, and D. C. Toll, "Designing a Side Channel Resistant Random Number Generator," in *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010*, D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, Eds. Springer, April 2010, pp. 49–64.

[137] J. Bringer, H. Chabanne, T. Kevenaar, and B. Kindarji, "Extending Match-On-Card to Local Biometric Identification," in *Biometric ID Management and Multimodal Communication*, ser. Lecture Notes in Computer Science, J. Fierrez, J. Ortega-Garcia, A. Esposito, A. Drygajlo, and M. Faundez-Zanuy, Eds. Springer Berlin / Heidelberg, 2009, vol. 5707, pp. 178–186, 10.1007/978-3-642-04391-8_23. Online Available: http://www.springerlink.com/content/b16016708315549v/fulltext.pdf

[138] T. Bourlai, J. Kittler, and K. Messer, "On design and optimization of face verification systems that are smart-card based," *Machine Vision and Applications*, vol. 21, pp. 695–711, 2010, 10.1007/s00138-009-0187-x. Online Available: http://www.springerlink.com/content/e73334305v740016/fulltext.pdf

[139] L. Beaugé and A. Drygajlo, "Fully featured secure biometric smart card device for fingerprint-based authentication and identification," in *Proceedings of the 12th ACM workshop on Multimedia and security*, ser. MM&Sec '10. New York, NY, USA: ACM, 2010, pp. 181–186.

[140] *ISO/IEC 14888: Information Technology – Security Techiques – Digital Signature with Appendix*, International Organization for Standardization (ISO) Std., April 2008.

[141] *ISO/IEC 9796: Information Technology – Security Techniques – Digital Signature Schemes Giving Message Recovery* , International Organization for Standardization (ISO) Std., December 2010.

[142] G. Lowe, "Casper: a compiler for the analysis of security protocols," *J. Comput. Secur.*, vol. 6, pp. 53–84, January 1998. Online Available: http://dl.acm.org/citation.cfm?id=353677.353680

[143] C. A. R. Hoare, *Communicating sequential processes.* New York, NY, USA: ACM, 1978, vol. 21, no. 8.

[144] P. Ryan and S. Schneider, *The Modelling and Analysis of Security Protocols: the CSP Approach.* Addison-Wesley Professional, 2000.

[145] Joan Daemen and Vincent Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard.* Berlin, Heidelberg, New York: Springer Verlag, 2002.

[146] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography.* CRC, October 1996.

[147] *FIPS 180-2: Secure Hash Standard (SHS)*, National Institute of Standards and Technology (NIST) Std., 2002. Online Available: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[148] M. Lepinski and S. Kent, "RFC 5114 - Additional Diffie-Hellman Groups

for Use with IETF Standards," Tech. Rep., January 2008. Online Available: http://tools.ietf.org/html/rfc5114

[149] R. N. Akram, "Pseudorandom Number Generation/Attacks in Smart Cards," Master's Thesis, Smart Card Centre, Information Security Group, Royal Holloway, University of London, Egham, United Kingdom, September 2007.

[150] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "SCUBA: Secure Code Update By Attestation in sensor networks," in *Proceedings of the 5th ACM workshop on Wireless security*, ser. WiSe '06. New York, NY, USA: ACM, 2006, pp. 85–94.

[151] Y. Li, J. M. McCune, and A. Perrig, "SBAP: software-based attestation for peripherals," in *Proceedings of the 3rd international conference on Trust and trustworthy computing*, ser. TRUST'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 16–29. Online Available: http://portal.acm.org/citation.cfm?id=1875652.1875655

[152] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: SoftWare-based ATTestation for Embedded Devices," *Security and Privacy, IEEE Symposium on*, vol. 0, p. 272, 2004.

[153] D. Schellekens, B. Wyseur, and B. Preneel, "Remote attestation on legacy operating systems with trusted platform modules," *Sci. Comput. Program.*, vol. 74, pp. 13–22, December 2008. Online Available: http://portal.acm.org/citation.cfm?id=1464515.1464789

[154] H. Busch, S. Katzenbeisser, and P. Baecher, "PUF-Based Authentication Protocols - Revisited," in *Information Security Applications*, ser. Lecture Notes in Computer Science, H. Youm and M. Yung, Eds. Springer Berlin / Heidelberg, 2009, vol. 5932, pp. 296–308, 10.1007/978-3-642-10838-9_22.

[155] "GlobalPlatform's Proposition for NFC Mobile: Secure Element Managment and Messaging," Online, GlobalPlatform, Specification, April 2009.

[156] (Visited August, 2010) Global Systems for Mobile Communication (GSM). GSM Association. Online Available: http://www.gsm.org

[157] "GlobalPlatform Guide to Common Personalization," Online, Redwood City, USA, Specification 1.0, May 2003.

[158] "GlobalPlatform Card: Confidential Card Content Management. Card Specification v2.2 - Amendment A," Online, Redwood City, USA, Specification 1.0.1, January 2011. Online Available: http://www.globalplatform.org/specificationscard.asp

[159] "Multos: Guide to Generating Application Load Units," MAOSCO, Tech. Rep. MAO-DOC-TEC-009 v2.52, 2006. Online Available: http://www.multos.com/downloads/technical/galu.pdf

[160] "Future Networks and the Internet: Early Challenges Regarding the Internet of Things," Commission of the European Communities, Brussels, Commisision Staff Working Document SEC(2008) 2516, September 2008. Online Available: http://ec.europa.eu/information_society/eeurope/i2010/docs/future_internet/swp_internet_things.pdf

[161] D. A. Basin, S. Friedrich, J. Posegga, and H. Vogt, "Java Bytecode Verification by Model Checking," in *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1999, pp. 491–494.

[162] D. A. Basin, S. Friedrich, and M. Gawkowski, "Verified Bytecode Model Checkers," in *TPHOLs '02: Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*. London, UK: Springer-Verlag, 2002, pp. 47–66.

[163] X. Leroy, "On-Card Bytecode Verification for Java Card," in *E-SMART '01: Proceedings of the International Conference on Research in Smart Cards*. London, UK: Springer-Verlag, 2001, pp. 150–164.

[164] "Smart Cards; Smart Card Platform Requirements Stage 1(Release 9)," European Telecommunications Standards Institute (ETSI), France, Technical Specification ETSI TS 102 412 (V9.1.0), June 2009. Online Available: http://www.etsi.org/deliver/etsi_ts/102400_102499/102412/09.01.00_60/ts_102412v090100p.pdf

[165] Y. Gasmi, A.-R. Sadeghi, P. Stewin, M. Unger, and N. Asokan, "Beyond Secure Channels," in *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*. New York, NY, USA: ACM, 2007, pp. 30–40.

[166] L. Zhou and Z. Zhang, "Trusted Channels with Password-Based Authentication and TPM-Based Attestation," *Communications and Mobile Computing, International Conference on*, vol. 1, pp. 223–227, 2010.

[167] F. Armknecht, Y. Gasmi, A.-R. Sadeghi, P. Stewin, M. Unger, G. Ramunno, and D. Vernizzi, "An efficient implementation of trusted channels based on openssl," in *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, ser. STC '08. New York, NY, USA: ACM, 2008, pp. 41–50.

[168] G. Horn, K. M. Martin, and C. J. Mitchell, "Authentication Protocols for Mobile Network Environment Value-Added Services," in *IEEE Transactions on Vehicular Technology*, vol. 51. IEEE, March 2002, pp. 383–392. Online Available: http://www.isg.rhul.ac.uk/cjm/apfmne2.pdf

[169] *Remote Application Management over HTTP, Card Specification v 2.2 - Amendment B*, Online, GlobalPlatform Specification, September 2006. Online Available: http://www.globalplatform.org/specificationscard.asp

[170] *GlobalPlatform Card Technology: Secure Channel Protocol 03, Card Specification v 2.2 - Amendment D,*, Online, GlobalPlatform Public Release GPC SPE 014, September 2009. Online Available: http://www.globalplatform.org/specificationscard.asp

[171] "Smart Cards; Secured Packet Structure for UICC based Applications (Release 6)," European Telecommunications Standards Institute (ETSI), France, Technical Specification ETSI TS 102 225 (V6.8.0), April 2006.

[172] Y. S. T. Tin, C. Boyd, and J. M. G. Nieto, "Provably Secure Mobile Key Exchange: Applying the Canetti-Krawczyk Approach," in *Proceedings of the 8th Australasian conference on Information security and privacy*, ser. ACISP'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 166–179. Online Available: http://portal.acm.org/citation.cfm?id=1760479.1760499

[173] W. G. Sirett, J. A. MacDonald, K. Mayes, and C. Markantonakis, "Design, Installation and Execution of a Security Agent for Mobile Stations," in *Smart Card Research and Advanced Applications, 7th IFIP WG 8.8/11.2 International Conference, CARDIS*, ser. LNCS, J. Domingo-Ferrer, J. Posegga, and D. Schreckling, Eds., vol. 3928. Tarragona, Spain: Springer, April 2006, pp. 1–15.

[174] W. Diffie, P. C. van Oorschot, and M. J. Wiener, "Authentication and Authenticated Key Exchanges," *Designs, Codes and Cryptography*, vol. 2, no. 2, pp. 107–125, 1992.

[175] A. Aziz and W. Diffie, "Privacy And Authentication For Wireless Local Area Networks," *IEEE Personal Communications*, vol. 1, pp. 25–31, First Quarter 1994.

[176] K. Martin, B. Preneel, C. Mitchell, H. Hitz, G. Horn, A. Poliakova, and P. Howard, "Secure billing for mobile information services in UMTS," in *Intelligence in Services and Networks: Technology for Ubiquitous Telecom Services*, ser. Lecture Notes in Computer Science, S. Trigila, A. Mullery, M. Campolargo, H. Vanderstraeten, and M. Mampaey, Eds. Springer Berlin / Heidelberg, 1998, vol. 1430, pp. 535–548, 10.1007/BFb0056997.

[177] G. Horn and B. Preneel, "Authentication and Payment in Future Mobile Systems," in *Computer Security — ESORICS 98*, ser. Lecture Notes in Computer Science, J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, Eds. Springer Berlin / Heidelberg, 1998, vol. 1485, pp. 277–293, 10.1007/BFb0055870.

[178] W. Aiello, S. M. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, A. D. Keromytis, and O. Reingold, "Just fast keying: Key agreement in a hostile internet," *ACM Trans. Inf. Syst. Secur.*, vol. 7, pp. 242–273, May 2004.

[179] S. Blake-Wilson, D. Johnson, and A. Menezes, "Key Agreement Protocols and Their Security Analysis," in *Proceedings of the 6th IMA International Conference on Cryptography and Coding*. London, UK: Springer-Verlag, 1997, pp. 30–45. Online Available: http://portal.acm.org/citation.cfm?id=647993.742138

[180] C. Mitchell, M. Ward, and P. Wilson, "Key Control in Key Agreement Protocols," *Electronics Letters*, vol. 34, no. 10, pp. 980 –981, May 1998.

[181] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, no. 6, pp. 644–654, 1976.

[182] P. Urien, "Collaboration of SSL Smart Cards within the WEB2 Landscape," *Collaborative Technologies and Systems, International Symposium on*, vol. 0, pp. 187–194, 2009.

[183] P. Urien and S. Elrharbi, "Tandem Smart Cards: Enforcing Trust for TLS-Based Network Services," *Applications and Services in Wireless Networks, International Workshop on*, vol. 0, pp. 96–104, 2008.

[184] A. Harbitter and D. A. Menascé, "The Performance of Public Key-Enabled Kerberos Authentication in Mobile Computing Applications," pp. 78–85, 2001.

[185] M. Montgomery and K. Krishna, "Secure Object Sharing in Java Card," in *WOST'99: Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*. Berkeley, CA, USA: USENIX Association, 1999,

pp. 14–14.

[186] R. N. Akram, K. Markantonakis, and K. Mayes, "Firewall Mechanism in a User Centric Smart Card Ownership Model," in *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010*, D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, Eds., vol. 6035/2010. Passau, Germany: Springer, April 2010, pp. 118–132.

[187] M. Éluard, T. P. Jensen, and E. Denney, "An Operational Semantics of the Java Card Firewall," in *E-SMART '01: Proceedings of the International Conference on Research in Smart Cards*. London, UK: Springer-Verlag, 2001, pp. 95–110.

[188] C. Bernardeschi and L. Martini, "Enforcement of Applet Boundaries in Java Card Systems," in *IASTED Conf. on Software Engineering and Applications*, 2004, pp. 96–101.

[189] M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov, "Checking Absence of Illicit Applet Interactions: A Case Study," in *Fundamental Approaches to Software Engineering, FASE 2004*, ser. Lecture Notes in Computer Science, no. 2984. Springer, 2004.

[190] W. Mostowski and E. Poll, "Malicious Code on Java Card Smartcards: Attacks and Countermeasures," in *CARDIS '08: Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 1–16.

[191] M. Éluard and T. Jensen, "Secure Object Flow Analysis for Java Card," in *CARDIS'02: Proceedings of the 5th conference on Smart Card Research and Advanced Application Conference*. Berkeley, CA, USA: USENIX Association, 2002, pp. 11–11.

[192] P. Bieber, J. Cazin, A. E. Marouani, P. Girard, J. L. Lanet, V. Wiels, and G. Zanon, "The PACAP Prototype: A Tool for Detecting Java Card Illegal Flow," in *JavaCard '00: Revised Papers from the First International Workshop on Java on Smart Cards: Programming and Security*. London, UK: Springer-Verlag, 2001, pp. 25–37.

[193] "National Strategy for Trusted Identities in Cyberspace," Department of Homeland Security, USA, Draft Proposal, June 2010. Online Available: http://www.dhs.gov/xlibrary/assets/ns_tic.pdf

[194] G. Barbu, H. Thiebeauld, and V. Guerin, "Attacks on Java Card 3.0 Combining Fault and Logical Attacks," in *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010*, ser. LNCS, D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, Eds., vol. 6035/2010, 2010, pp. 148–163.

[195] J.-L. Lanet and J. Iguchi-Cartigny, "Developing a Trojan applet in a Smart Card ," *Journal in Computer Virology*, vol. 6, no. 1, 2009.

[196] R. Anderson and M. Kuhn, "Low Cost Attacks on Tamper Resistant Devices," in *Security Protocols*, B. Christianson, B. Crispo, M. Lomas, and M. Roe, Eds. Springer, 1998, vol. 1361, pp. 125–136.

[197] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *CRYPTO '99:*

Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology. London, UK: Springer-Verlag, 1999, pp. 388–397.

[198] E. Vétillard and A. Ferrari, "Combined Attacks and Countermeasures," in *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010*, ser. LNCS, D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, Eds., vol. 6035/2010, 2010, pp. 133–147.

[199] S. Chaumette and D. Sauveron, "An Efficient and Simple Way to Test the Security of Java Cards," in *Security in Information Systems, Proceedings of the 3rd International Workshop on Security in Information Systems, WOSIS 2005, In conjunction with ICEIS2005*, E. Fernández-Medina, J. C. H. Castro, and L. J. G. Castro, Eds. INSTICC Press, 2005, pp. 331–341.

[200] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley Longman, Amsterdam, April 1999.

[201] G. Barthe, G. Dufay, L. Jakubiec, B. P. Serpette, and S. a. M. d. Sousa, "A Formal Executable Semantics of the JavaCard Platform," in *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ser. ESOP '01. London, UK: Springer-Verlag, 2001, pp. 302–319. Online Available: http://dl.acm.org/citation.cfm?id=645395.757559

[202] G. Barthe and S. Stratulat, "Validation of the JavaCard Platform with Implicit Induction Techniques," in *RTA*, 2003, pp. 337–351.

[203] M. Éluard, T. P. Jensen, and E. Denney, "An Operational Semantics of the Java Card Firewall," in *Proceedings of the International Conference on Research in Smart Cards: Smart Card Programming and Security*, ser. E-SMART '01. London, UK, UK: Springer-Verlag, 2001, pp. 95–110. Online Available: http://dl.acm.org/citation.cfm?id=646803.706114

[204] J. L. Lanet and A. Requet, "Formal Proof of Smart Card Applets Correctness," in *Proceedings of the The International Conference on Smart Card Research and Applications*. London, UK: Springer-Verlag, 2000, pp. 85–97. Online Available: http://dl.acm.org/citation.cfm?id=646692.703437

[205] H. Meijer and E. Poll, "Towards a Full Formal Specification of the JavaCard API," in *Smart Card Programming and Security*, ser. Lecture Notes in Computer Science, I. Attali and T. Jensen, Eds. Springer Berlin / Heidelberg, 2001, vol. 2140, pp. 165–178, 10.1007/3-540-45418-7_14.

[206] V. Almaliotis, A. Loizidis, P. Katsaros, P. Louridas, and D. Spinellis, "Static Program Analysis for Java Card Applets," in *CARDIS '08: Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 17–31.

[207] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," in *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*. London, UK: Springer-Verlag, 1997, pp. 513–525. Online Available: http://dl.acm.org/citation.cfm?id=646762.706179

[208] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Investigations of power analysis attacks on smartcards," in *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*. Berkeley, CA, USA: USENIX Association, 1999, pp. 17–17. Online Available: http://dl.acm.org/citation.cfm?id=1267115.1267132

[209] S. P. Skorobogatov and R. J. Anderson, "Optical Fault Induction Attacks," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '02. London, UK, UK: Springer-Verlag, 2003, pp. 2–12. Online Available: http://dl.acm.org/citation.cfm?id=648255.752727

[210] J.-J. Quisquater and D. Samyde, *Eddy current for Magnetic Analysis with Active Sensor*. Springer, 2002.

[211] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, ser. Lecture Notes in Computer Science. Springer, 2003, vol. 2523, pp. 81–95, 10.1007/3-540-36400-5_20.

[212] "Joint Interpretation Library - Application of Attack Potential to Smartcards," Online, Tech. Rep., Apirl 2006. Online Available: http://www.ssi.gouv.fr/site_documents/JIL/JIL-The_application_o_f_attack_potential_to_smartcards_V2-1.pdf

[213] O. Vertanen, "Java Type Confusion and Fault Attacks," in *Fault Diagnosis and Tolerance in Cryptography*, ser. Lecture Notes in Computer Science, L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, Eds. Springer Berlin / Heidelberg, 2006, vol. 4236, pp. 237–251, 10.1007/11889700_21.

[214] A. Lemarechal, "Introduction to fault attacks on smartcard," in *On-Line Testing Symposium, 2005. IOLTS 2005. 11th IEEE International*, july 2005, p. 116.

[215] J. Hogenboom and W. Mostowski, "Full Memory Read Attack on a Java Card," in *4th Benelux Workshop on Information and System Security*, O. Pereira, J.-J. Quisquater, and F.-X. Standaert, Eds. Belgium: Springer, November 2009.

[216] A. A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet, "Automatic Detection of Fault Attack and Countermeasures," in *Proceedings of the 4th Workshop on Embedded Systems Security*, ser. WESS '09. New York, NY, USA: ACM, 2009, pp. 71–77.

[217] G. Barbu, G. Duc, and P. Hoogvorst, "Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures," in *The tenth Smart Card Research and Advanced Application IFIP Conference (CARDIS2011)*, ser. LNCS, E. Prouff, Ed. Belgium: Sp, September 2011.

[218] G. Barbu and H. Thiebeauld, "Synchronized Attacks on Multithreaded Systems - Application to Java Card 3.0-," in *The tenth Smart Card Research and Advanced Application IFIP Conference (CARDIS2011)*, ser. LNCS, E. Prouff, Ed. Springer, September 2011.

[219] G. Bouffard, J. Iguchi-Cartigny, and J.-L. Lanet, "Combined Software and Hardware Attacks on the Java Card Control Flow," in *The tenth Smart Card Research and*

Advanced Application IFIP Conference (CARDIS2011), ser. LNCS, E. Prouff, Ed. Belgium: Springer, September 2011.

[220] A. A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet, "Evaluation of Countermeasures Against Fault Attacks on Smart Cards," in *International Journal of Security and its Applications*, vol. 5, no. 2, April 2011.

[221] O. Derouet. (2007, September) Secure Smartcard Design Againist Laser Fault In. (Invited Speaker) 4th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDRC 2007). IEEE-CS. Vienna, Austria. Online Available: http://conferenze.dei.polimi.it/FDTC07/Derouet_remaster.pdf

[222] S.-K. Kim, T. H. Kim, D.-G. Han, and S. Hong, "An efficient CRT-RSA algorithm secure against power and fault attacks," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1660–1669, 2011. Online Available: http://linkinghub.elsevier.com/retrieve/pii/S0164121211001014

[223] S. Liu, B. King, and W. Wang, "A CRT-RSA Algorithm Secure against Hardware Fault Attacks," *2006 2nd IEEE International Symposium on Dependable Autonomic and Secure Computing*, pp. 51–60, 2006. Online Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4030866

[224] E. Trichina and R. Korkikyan, "Multi Fault Laser Attacks on Protected CRT-RSA," *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 75–86, 2010. Online Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5577278

[225] S. Zhou, B. R. Childers, and N. Kumar, "Profile Guided Management of Code Partitions for Embedded Systems," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2004, p. 21396.

[226] T. Zhang, S. Pande, and A. Valverde, "Tamper-resistant Whole Program Partitioning," in *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems.* New York, NY, USA: ACM, 2003, pp. 209–219.

[227] X. Zhuang, T. Zhang, H.-H. S. Lee, and S. Pande, "Hardware Assisted Control Flow Obfuscation for Embedded Processors," in *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems.* New York, NY, USA: ACM, 2004, pp. 292–302.

[228] G. Bouffard, J.-L. Lanet, J.-B. Machemie, J.-Y. Poichotte, and J.-P. Wary, "Evaluation of the Ability to Transform SIM Application into Hostile Application," in *the Tenth Smart Card Research and Advanced Application Conference (CARDIS 2011*, ser. LNCS, E. Prouff, Ed. Leuven, Belgium: Springer, September 2011.

[229] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, "Identification and Verification of Security Relevant Functions in Embedded Systems Based on Source Code Annotations and Assertions," in *Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices*, ser. Lecture Notes in Computer

Science, P. Samarati, M. Tunstall, J. Posegga, K. Markantonakis, and D. Sauveron, Eds. Springer, 2010, vol. 6033, pp. 316–323, 10.1007/978-3-642-12368-9_25.

[230] A.-A.-K. Séré, J. Iguchi-Cartigny, and J.-L. Lanet, "Checking the Paths to Identify Mutant Application on Embedded Systems," in *Future Generation Information Technology - Second International Conference (FGIT 2010)*, ser. Lecture Notes in Computer Science, T.-H. Kim, Y.-H. Lee, B. H. Kang, and D. Slezak, Eds., vol. 6485. Jeju Island, Korea,: Springer, December 2010, pp. 459–468.

[231] K. Markantonakis, "Secure Logging Mechanisms for Smart Cards," Ph.D. dissertation, Royal Holloway, University of London, Egham, United Kingdom, December 1999.

[232] "Advanced Security for Personal Communications Technology (ASPeCT)." Online Available: http://www.esat.kuleuven.be/cosic/aspect/

[233] *Failures-Divergence Refinement*, Formal Systems (Europe) Ltd, June 2005. Online Available: www.fsel.com