Authorisation Issues for Mobile Code in Mobile Systems

Eimear Gallery

Technical Report
RHUL–MA–2007–3
18 May 2007

Department of Mathematics
Royal Holloway, University of London
Egham, Surrey TW20 0EX, England
http://www.rhul.ac.uk/mathematics/techreports

Authorisation Issues for Mobile Code in Mobile Systems

by

Eimear Gallery

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

Department of Mathematics
Royal Holloway, University of London
2006

Abstract

This thesis is concerned with authorisation issues for mobile code in mobile systems. It is divided into three main parts. Part I covers the development of a policy-based framework for the authorisation of mobile code and agents by host systems. Part II addresses the secure download, storage and execution of a conditional access application, used in the secure distribution of digital video broadcast content. Part III explores the way in which trusted computing technology may be used in the robust implementation of OMA DRM version 2.

In part I of this thesis, we construct a policy-based mobile code and agent authorisation framework, with the objective of providing both mobile devices and service providers with the ability to assign appropriate privileges to incoming executables. Whilst mobile code and agent authorisation mechanisms have previously been considered in a general context, this thesis focuses on the special requirements resulting from mobile code and agent authorisation in a mobile environment, which restrict the types of solutions that may be viable. Following the description and analysis of a number of architectural models upon which a policy-based framework for mobile code and agent authorisation may be constructed, we outline a list of features desirable in the definitive underlying architecture. Specific implementation requirements for the capabilities of the policy and attribute certificate specification languages and the associated policy engine are then extracted. Candidate policy specification languages, namely KeyNote (and Nereus), Ponder (and (D)TPL) and SAML are then examined, and conclusions drawn regarding their suitability for framework expression. Finally, the definitive policy based framework for mobile code and agent authorisation is described.

In the second part of this thesis, a flexible approach that allows consumer products to support a wide range of proprietary content protection systems, or more specifically digital video broadcast conditional access systems, is proposed. Two protocols for the secure download of content protection software to mobile devices are described. The protocols apply concepts from trusted computing to demonstrate that a platform is in a sufficiently trustworthy state before any application or associated keys are securely downloaded. The protocols are designed to allow mobile devices to receive broadcast content protected by proprietary conditional access applications. Generic protocols are first described, followed by an analysis of how well the downloaded code is protected in transmission. How the generic protocols may be implemented using specific trusted computing technologies is then investigated. For each of the selected trusted computing technologies, an analysis of how the conditional access application is protected while in storage and while executing on the mobile host is also presented. We then examine two previously proposed download protocols, which assume a mo-

bile receiver compliant with the XOM and AEGIS system architectures. Both protocols are then analysed against the security requirements defined for secure application download, storage and execution. We subsequently give a series of proposed enhancements to the protocols which are designed to address the identified shortcomings.

In the final section of this thesis, we examine OMA DRM version 2, which defines the messages, protocols and mechanisms necessary in order to control the use of digital content in a mobile environment. However, an organisation, such as the CMLA, must specify how robust implementations of the OMA DRM version 2 specification should be, so that content providers can be confident that their content will be safe on OMA DRM version 2 devices. We take the requirements extracted for the robust implementation of the OMA DRM version 2 specification and propose an implementation which meets these requirements using the TCG architecture and TPM/TSS version 1.2 commands.

Acknowledgements

I would like to thank my supervisor Chris Mitchell, who has been a constant source of knowledge, advice and encouragement throughout the course of my study.

My thanks to Mobile VCE (www.mobilevce.com), which funded the work completed in part I of this thesis. This work was completed as part of the Core 2 research program.

I would like to thank Vodafone (in particular Tim Wright and Nick Bone) for many valuable comments and suggestions, and for originally suggesting the work completed in part III of this thesis.

I would also like to thank the staff and the other post-graduate students in the Information Security Group for making my time spent at Royal Holloway both enjoyable and challenging.

Finally, I would like to thank my family and friends who have been so supportive throughout this entire process.

Declaration

These doctoral studies were conducted under the supervision of Chris Mitchell and Kenny Paterson. 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.

Eimear Gallery

Contents

1 Introduction **26**
- 1.1 Motivation and challenges . . . 28
  - 1.1.1 Part I: Mobile host protection . . . 28
  - 1.1.2 Part II: Mobile code protection . . . 29
  - 1.1.3 Part III: Remote code protection . . . 30
- 1.2 Structure of thesis . . . 31
- 1.3 Contribution of thesis . . . 35
- 1.4 List of publications . . . 39
- 1.5 Cryptographic primitives . . . 40
  - 1.5.1 Hash functions . . . 40
  - 1.5.2 Message authentication codes . . . 41
  - 1.5.3 Symmetric encryption . . . 42
  - 1.5.4 Asymmetric cryptography . . . 42
  - 1.5.5 Asymmetric encryption . . . 43
  - 1.5.6 Digital signatures . . . 43
  - 1.5.7 Public key infrastructure . . . 44
  - 1.5.8 Web of trust . . . 45
  - 1.5.9 Privilege management infrastructure . . . 45
  - 1.5.10 Authentication protocols . . . 46
  - 1.5.11 Freshness mechanisms . . . 46
- 1.6 Trusted computing primitives . . . 47
  - 1.6.1 Trust . . . 47
  - 1.6.2 Roots of trust . . . 48
  - 1.6.3 Integrity measurement . . . 50
  - 1.6.4 Authenticated boot . . . 50
  - 1.6.5 Secure boot . . . 51
  - 1.6.6 Attestation . . . 51
  - 1.6.7 Sealing . . . 51
  - 1.6.8 Process isolation . . . 52
  - 1.6.9 Secure I/O . . . 53
- 1.7 Definitions . . . 53

I Mobile host protection **57**

2 Mobile code and agent authorisation **58**
- 2.1 Introduction . . . 59
- 2.2 Agents . . . 62

2.3 Mobile agents . . . 64
2.4 Mobile agent security . . . 64
2.5 Mobile agent authorisation techniques . . . 66
2.5.1 Code and agent behaviour . . . 66
2.5.2 Code and agent origin . . . 72
2.5.3 Code and agent integrity . . . 76
2.6 Conclusions . . . 80

3 Architectural models for mobile code and agent authorisation 83
3.1 Introduction . . . 84
3.2 Entities involved . . . 86
3.3 Scenario 1 . . . 87
3.4 Scenario 2 . . . 91
3.5 Scenario 3 . . . 93
3.6 Scenario 4 . . . 98
3.7 Scenario 5 . . . 101
3.8 Scenario 6 . . . 104
3.9 Conclusions . . . 107

4 A policy engine for mobile code and agent authorisation 110
4.1 Introduction . . . 112
4.2 The policy engine . . . 113
4.2.1 Policy statements . . . 113
4.2.2 Attribute certificates . . . 114
4.2.3 Authentication evidence . . . 115
4.2.4 Compliance values . . . 117
4.2.5 The PAP . . . 117
4.2.6 The PIP . . . 117
4.2.7 The AP . . . 118
4.2.8 The TEM . . . 119
4.2.9 The PDP . . . 119
4.2.10 The PEP . . . 119
4.3 Approaches to policy specification . . . 119
4.4 KeyNote . . . 124
4.4.1 Scenario 1 . . . 128
4.4.2 Scenario 2 . . . 134
4.4.3 Scenario 3 . . . 139
4.4.4 Scenario 4 . . . 147
4.4.5 Scenarios 5 and 6 . . . 149
4.4.6 Conclusions . . . 151
4.5 Ponder . . . 159
4.5.1 Prior art . . . 163
4.5.2 Scenario 1 . . . 169
4.5.3 Scenarios 2 – 6 . . . 175
4.5.4 Conclusions . . . 176
4.6 SAML . . . 177
4.6.1 Scenario 1 . . . 184
4.6.2 Scenario 2 . . . 189
4.6.3 Scenarios 3 and 4 . . . 191
4.6.4 Scenarios 5 and 6 . . . 195

4.6.5 Conclusions . . . 196
4.7 Conclusions . . . 196

5 A policy-based authorisation framework 201
5.1 Introduction . . . 202
5.2 Requirements . . . 203
5.3 The framework — A high level view . . . 203
5.4 Design decisions . . . 207
5.5 Notation . . . 208
5.6 Assumptions . . . 209
5.7 Trusted domain server activity . . . 211
5.7.1 Evaluating a mobile agent . . . 211
5.7.2 Attribute certificates . . . 216
5.7.3 Authentication evidence . . . 219
5.8 End host activity . . . 219
5.8.1 The PIP . . . 221
5.8.2 The AP . . . 221
5.8.3 The TEM . . . 223
5.8.4 Policy statements . . . 227
5.8.5 The PDP and PEP . . . 227
5.8.6 The PAP . . . 228
5.9 Conclusions . . . 229

II Mobile code protection 231

6 Conditional access in mobile systems 232
6.1 Introduction . . . 234
6.2 Conditional access systems . . . 236
6.2.1 DVB standards . . . 238
6.2.2 Simulcrypt . . . 238
6.2.3 Common interface . . . 239
6.2.4 Limitations of existing mechanisms . . . 240
6.2.5 Modifications required for mobile receivers . . . 241
6.3 Security issues . . . 242
6.3.1 Security threats . . . 242
6.3.2 Security services and mechanisms . . . 243
6.4 Conclusions . . . 245

7 Protocols for secure application download 247
7.1 Introduction . . . 248
7.2 Model . . . 248
7.3 Prior art . . . 250
7.4 Notation . . . 252
7.5 Assumptions . . . 253
7.6 Protocol initiation . . . 257
7.7 Key exchange protocol . . . 258
7.7.1 Protocol specification . . . 258
7.7.2 Security analysis of the key exchange protocol . . . 261
7.8 Key agreement protocol . . . 264

7.8.1 Protocol specification . . . 264
7.8.2 Security analysis of the key agreement protocol . . . 267
7.9 Conclusions . . . 269

8 Protocol implementation using trusted computing frameworks 270
8.1 Introduction . . . 272
8.2 Notation . . . 272
8.3 Implementing the protocols using the TCG specifications . . . 273
8.3.1 Key exchange protocol . . . 276
8.3.2 Key agreement protocol . . . 286
8.3.3 Implementation specific security analysis . . . 291
8.4 Implementing the protocols using the TCG specification set and an integrated isolation kernel . . . 296
8.4.1 Key exchange protocol . . . 297
8.4.2 Key agreement protocol . . . 298
8.4.3 Implementation specific security analysis . . . 298
8.5 Implementing the protocols using NGSCB . . . 302
8.5.1 Key exchange protocol . . . 303
8.5.2 Key agreement protocol . . . 304
8.5.3 Implementation specific security analysis . . . 304
8.6 Conclusions . . . 307

9 Secure application download using XOM and AEGIS architectures 309
9.1 Introduction . . . 311
9.2 Model . . . 312
9.3 Notation . . . 313
9.4 Assumptions . . . 313
9.5 Protocol initiation . . . 315
9.6 The XOM application download protocol . . . 316
9.6.1 The XOM system architecture . . . 316
9.6.2 The XOM download protocol . . . 317
9.6.3 Security analysis . . . 319
9.6.4 Proposed security enhancements/clarifications . . . 325
9.7 The AEGIS application download protocol . . . 327
9.7.1 The AEGIS system architecture . . . 327
9.7.2 The AEGIS download protocol . . . 328
9.7.3 Security analysis . . . 331
9.7.4 Proposed security enhancements/clarifications . . . 337
9.8 Conclusions . . . 339

III Remote code protection 341

10 OMA DRM 342
10.1 Introduction . . . 343
10.1.1 The MPWG . . . 343
10.1.2 Digital rights management . . . 344
10.1.3 Scope of part III . . . 345
10.2 DRM . . . 346

10.3 The OMA . . . 347
10.4 Model . . . 348
10.4.1 Functional entities . . . 348
10.4.2 Functional components . . . 349
10.4.3 Functional architecture . . . 350
10.5 OMA DRM v1 . . . 350
10.6 OMA DRM v2 . . . 351
10.7 Conclusions . . . 354

11 Requirements for a robust implementation of OMA DRM v2 355
11.1 Introduction . . . 356
11.2 OMA DRM v2 agent installation . . . 357
11.3 The rights object acquisition protocol (ROAP) suite . . . 360
11.3.1 Notation . . . 360
11.3.2 The 4-pass registration protocol . . . 363
11.3.3 The rights acquisition protocols . . . 368
11.3.4 The 2-pass join domain protocol . . . 373
11.3.5 The 2-pass leave domain protocol . . . 376
11.4 Conclusions . . . 378

12 A robust implementation of OMA DRM v2 379
12.1 Introduction . . . 381
12.2 Requirements analysis . . . 382
12.2.1 Requirement 1 . . . 382
12.2.2 Requirement 2 – 5 and 8 – 21 . . . 384
12.2.3 Requirement 6 . . . 385
12.2.4 Requirement 7 . . . 386
12.2.5 Meeting the requirements using a TMP . . . 386
12.3 Model . . . 387
12.4 Assumptions . . . 388
12.5 The trusted mobile platform architecture . . . 390
12.6 Authenticated boot . . . 392
12.7 Secure boot . . . 394
12.7.1 Prior art . . . 394
12.7.2 Secure boot using a version 1.1 compliant TPM . . . 398
12.7.3 Secure boot using a version 1.2 compliant TPM . . . 399
12.8 Platform run-time integrity . . . 403
12.9 Fundamental TSS and TPM command sequences . . . 405
12.9.1 TPM permanent flags . . . 405
12.9.2 TPM initialisation . . . 406
12.9.3 TPM startup . . . 406
12.9.4 Context management . . . 407
12.9.5 Endorsement key pair generation . . . 410
12.9.6 Accessing the public endorsement key . . . 410
12.9.7 TPM self testing . . . 412
12.9.8 Enabling the TPM . . . 412
12.9.9 The ownership flag . . . 413
12.9.10 Taking ownership of the TPM . . . 413
12.9.11 TPM activation . . . 415
12.10 Secure storage . . . 415

12.10.1 Key hierarchy . . . 415
12.10.2 Installing integrity and confidentiality sensitive OMA DRM v2 data on the device . . . 416
12.10.3 Secure storage of and access control to OMA DRM v2 data 418
12.10.4 Security of the OMA DRM v2 data while in use on the device . . . 426
12.11 Platform attestation . . . 426
12.12 Demonstrating privilege . . . 429
12.13 Random number generation . . . 434
12.14 Trusted time source . . . 435
12.15 Conclusions . . . 436

13 Conclusions **444**
13.1 Summary and conclusions . . . 445
13.1.1 Part I: Mobile host protection . . . 445
13.1.2 Part II: Mobile code protection . . . 454
13.1.3 Part III: Remote code protection . . . 458
13.2 Future work . . . 460

Bibliography **463**

A The TCG specification set **488**
A.1 Introduction . . . 489
A.2 Notation . . . 490
A.3 The TCG . . . 490
A.4 A trusted platform . . . 492
A.5 Entities involved . . . 493
A.6 The trusted platform subsystem . . . 495
A.6.1 Roots of trust . . . 495
A.6.2 The TSS . . . 496
A.7 Properties of a TPM . . . 498
A.7.1 Protected capabilities and shielded locations . . . 499
A.7.2 TPM functional components . . . 499
A.7.3 PCRs . . . 502
A.7.4 The endorsement key . . . 503
A.8 Initialising the TPM . . . 503
A.9 Enabling, activating and taking ownership of the TPM . . . 505
A.9.1 Enabling the TPM . . . 505
A.9.2 Enabling ownership of the TPM . . . 506
A.9.3 Activating the TPM . . . 507
A.9.4 Taking ownership . . . 508
A.9.5 Clearing the TPM . . . 509
A.10 Platform identification and certification . . . 510
A.10.1 An endorsement credential . . . 510
A.10.2 A conformance credential . . . 512
A.10.3 A platform credential . . . 513
A.10.4 Attestation identities . . . 513
A.10.5 DAA . . . 517
A.11 Integrity measuring, recording and reporting . . . 518
A.11.1 Platform configuration registers . . . 518

A.11.2 Data integrity register 520
A.11.3 Integrity measurement 521
A.11.4 Assessing the software state of a platform 522
A.12 Locality 524
A.13 Protected storage 525
A.13.1 Object hierarchy 526
A.13.2 Sealing 528
A.13.3 Binding 529
A.13.4 Wrapping 530
A.14 Transport security 530
A.14.1 Session establishment 531
A.14.2 Transport encryption and authorisation 532
A.14.3 Transport logging 535
A.14.4 Error handling and exclusive transport sessions 535
A.15 Monotonic counter 536
A.16 Demonstrating privilege 536
A.16.1 Physical presence 537
A.16.2 Cryptographic authorisation 537
A.16.3 The OIAP 538
A.16.4 The OSAP 540
A.16.5 Changing authorisation data 543
A.16.6 The ADIP 545
A.17 Context manager 546
A.18 Delegation 547
A.18.1 Family and delegation tables 547
A.18.2 The delegate-specific authorisation protocol (DSAP) . . . 551
A.19 Time-stamping 553
A.20 Migration mechanisms 554
A.21 Maintenance mechanisms 556
A.22 Audit 557
A.23 NGSCB 559
A.23.1 The relationship between TCG and NGSCB 559
A.23.2 The NGSCB architecture 560
A.23.3 The tamper resistant crypto chip 560
A.23.4 The isolation kernel 562
A.24 LaGrande 564
A.24.1 The architecture 565
A.24.2 Hardware enhancements and extensions 565
A.25 Conclusions 567

B Technologies related to trusted computing 569
B.1 Introduction 570
B.2 Secure co-processors 570
B.3 Hardened processors 571
B.4 XOM 573
B.4.1 The abstract XOM machine 573
B.4.2 XOM machine implementation 574
B.4.3 Compartments 578
B.5 AEGIS 583
B.5.1 Secure computing model: Assumptions 584

B.5.2 The AEGIS architecture 584
B.5.3 Tamper evident processing 586
B.5.4 Private tamper resistant processing 592
B.5.5 Conclusions 595

List of Figures

3.1 Scenario 1 . . . 89
3.2 Scenario 2 . . . 93
3.3 Scenario 3 . . . 95
3.4 Scenario 3 . . . 96
3.5 Scenario 4 . . . 100
3.6 Scenario 5 — Domain server activity . . . 103
3.7 Scenario 5 — Mobile device activity . . . 103
3.8 Scenario 6 — Domain server activity . . . 105

4.1 The KeyNote trust management system . . . 127
4.2 Scenario 1 — Policy assertion . . . 128
4.3 Scenario 1 — Signed assertion/credential . . . 130
4.4 Scenario 1 — Signed assertion/credential . . . 130
4.5 Scenario 1 — Ordered compliance value set . . . 131
4.6 Scenario 1 — Action attribute set . . . 132
4.7 Scenario 1 — Action attribute set . . . 133
4.8 Scenario 1 — Action attribute set . . . 133
4.9 Scenario 2 — Policy assertion . . . 134
4.10 Scenario 2 — Policy assertion . . . 135
4.11 Scenario 2 — Policy assertion . . . 135
4.12 Scenario 2 — Signed assertion/credential . . . 136
4.13 Scenario 2 — Ordered compliance value set . . . 137
4.14 Scenario 2 — Action attribute set . . . 138
4.15 Scenario 2 — Action attribute set . . . 138
4.16 Scenario 2 — Action attribute set . . . 138
4.17 Scenario 3 — Policy assertion . . . 139
4.18 Scenario 3 — Policy assertion . . . 140
4.19 Scenario 3 — Policy assertion . . . 141
4.20 Scenario 3 — Policy assertion . . . 141
4.21 Scenario 3 — Policy assertion . . . 142
4.22 Scenario 3 — Policy assertion . . . 143
4.23 Scenario 3 — Ordered compliance value set . . . 144
4.24 Scenario 3 — Action attribute set . . . 145
4.25 Scenario 3 — Action attribute set . . . 145
4.26 Scenario 3 — Action attribute set . . . 145
4.27 Scenario 3 — Action attribute set . . . 146
4.28 Scenario 4 — Policy assertion . . . 147
4.29 Scenario 4 — Action attribute set . . . 148

4.30 Scenario 5 and 6 — Policy assertion 149
4.31 Scenario 5 and 6 — Action attribute set 150
4.32 Scenario 1 (Alternative) — Policy assertion 152
4.33 Scenario 1 (Alternative) — Policy assertion 153
4.34 Scenario 1 (Alternative) — Signed assertion/credential 155
4.35 Scenario 1 (Alternative) — Signed assertion/credential 155
4.36 Scenario 3 (Alternative) — Policy assertion 157
4.37 Scenario 3 (Alternative) — Policy assertion 157
4.38 Scenario 3 (Alternative) — Signed assertion 158
4.39 Scenario 3 (Alternative) — Action attribute set 158
4.40 The Ponder policy specification framework 163
4.41 The Extended April Agent Platform — Sample policy statement 166
4.42 The Extended April Agent Platform — Sample policy statement 166
4.43 SOMA — Sample policy statement 168
4.44 Scenario 1 — Policy statement 169
4.45 Scenario 1 — Policy statement 170
4.46 Scenario 1 — Policy statement 170
4.47 Scenario 1 — Policy statement 171
4.48 Scenario 1 — TE policy assertion 172
4.49 Scenario 1 — Signed assertion/credential 172
4.50 Scenario 1 — Action attribute set 173
4.51 SAML 183
4.52 Scenario 1 — Attribute assertion 187
4.53 Scenario 1 — Attribute assertion request 188
4.54 Scenario 1 — Attribute assertion response 188
4.55 Scenario 2 — Attribute assertion 190
4.56 Scenario 3 — Attribute assertion 192
4.57 Scenario 4 — Attribute assertion 193
4.58 Scenarios 3 and 4 — Attribute assertion request 194
4.59 Scenarios 3 and 4 — Attribute assertion response 195

5.1 Architecture model 204
5.2 Sample SAML attribute assertion generated by TDS 218
5.3 Sample SAML attribute assertion request 221
5.4 Sample SAML attribute assertion response 222
5.5 Sample DTPL policy statement 226
5.6 Sample Ponder composite policy statement 227

6.1 Scrambling broadcast services using DVB standards 238
6.2 Simulcrypt 239
6.3 Common interface module 240

7.1 Architecture model 250
7.2 Key exchange protocol 258
7.3 Key agreement protocol 264

8.1 Key hierarchy for the download of A_C 276
8.2 Key hierarchy for the download of A_C 286

9.1 Architecture model 313

10.1 Architecture model 348

12.1 Revised architecture model 387
12.2 OMA DRM v2 agent installer key hierarchy 416

A.1 The protected storage object hierarchy 526

List of Tables

4.1 A summary of KeyNote's applicability to the scenarios 198
4.2 A summary of Ponder's applicability to the scenarios 199
4.3 A summary of SAML's applicability to the scenarios 200

6.1 Conditional access system vendors 237

11.1 OMA DRM v2 agent installation 357
11.2 The 4-pass registration protocol 364
11.3 The 2-pass rights acquisition protocol 369
11.4 The 1-pass rights acquisition protocol 369
11.5 The 2-pass join domain protocol 374
11.6 The 2-pass leave domain protocol 376

12.1 The *VALIDATION_DATA* structure 402
12.2 The *TSS_EVENT_CERT* structure 402
12.3 TPM permanent flags 406
12.4 TPM initialisation 406
12.5 TPM start-up 407
12.6 Creating a context 407
12.7 Creating a TPM object 408
12.8 Connecting to a context 408
12.9 Closing a context 408
12.10 Freeing memory allocated to the context 408
12.11 The default policy object (created on TPM initialisation) 409
12.12 TPM device driver communications 410
12.13 Creating an endorsement key pair 411
12.14 Accessing the public endorsement key 411
12.15 Self testing 412
12.16 Physically enabling the TPM 412
12.17 Physically disabling the TPM 413
12.18 Enabling/Disabling the TPM 413
12.19 Setting the state of the *TPM_PF_OWNERSHIP* flag 413
12.20 Taking ownership of the TPM 414
12.21 Activating the TPM 415
12.22 A transport session 417
12.23 Creating a wrap key 420
12.24 Loading a key 422
12.25 Sealing data using a storage key 423

12.26 Wrapping a key to a PCR(s) 425
12.27 Creating a platform attestation identity key 427
12.28 Platform attestation 428
12.29 TPM owner reading of the public endorsement key 431
12.30 Authorising a load key and an object seal 433
12.31 Random number retrieval 435
12.32 TPM commands required for a robust implementation of OMA DRM v2 437
12.33 TPM commands required in a MTPM 441

13.1 A summary of scenario description and analysis 447
13.2 Policy engine requirements 450
13.3 Policy engine component analysis 452
13.4 A summary of the security threats, services and mechanisms pertaining to secure software download and execution 455
13.5 TPM commands required for a robust implementation of OMA DRM v2 458

A.1 TPM functional components 500
A.2 The delegation process 548
A.3 Tick session values 554
A.4 The NGSCB tamper resistant crypto chip 561
A.5 Security primitives supported by the NGSCB crypto chip 561

B.1 XOM enter_xom and exit_xom instructions 581
B.2 XOM secure_load and secure_store instructions 581
B.3 XOM mv_to_shared and mv_from_shared instructions 582
B.4 XOM save_register and restore_register instructions 582
B.5 AEGIS TE processing instructions 586
B.6 AEGIS TE processing — protection of initial state 588
B.7 AEGIS TE processing — protection of state on interrupts 589
B.8 AEGIS TE processing — on-chip cache integrity 590
B.9 AEGIS TE processing — sign_msg operation 592
B.10 AEGIS PTR processing instructions 592
B.11 AEGIS PTR processing — protection of state on interrupts . . . 593
B.12 AEGIS PTR processing — on-chip cache privacy 594

Acronyms

AC	Attribute Certificate
ACL	Access Control List
ADCP	Authorisation Data Change Protocol
ADIP	Authorisation Data Insertion Protocol
ADS	Agent Directory Service
AIK	Attestation Identity Key
AMS	Agent Management System
AP	Authentication Point
AR	Authorisation Requestor
ASL	Authorisation Specification Language
BIOS	Basic Input Output System
BBB	BIOS Boot Block
CA	Certification Authority or Conditional Access (depending on context)
CAPP	Controlled Access Protection Profile
CDC	Connected Device Configuration
CE	Conformance Entity
CEK	Content Encrypting Key
CI	Common Interface or Content Issuer or Cryptographic Interface (depending on context)

CLDC	Connect Limited Device Connection
CMK	Certifiable Migratable Key
CMLA	Content Management Licensing Administration
CPU	Central Processing Unit
CRTM	Core Root of Trust for Measurement
CRTV	Core Root of Trust for Verification
CSA	Common Scrambling Algorithm
CW	Control Word
DCF	DRM Container Format
DF	Directory Facilitator
DIR	Data Integrity Register
DMA	Direct Memory Access
DRM	Digital Rights Management
DSAP	Delegate-Specific Authorisation Protocol
DTD	Document Type Definition
DTPL	Definitive Trust Policy Language
DVB	Digital Video Broadcast
EAL	Evaluation Assurance Level
ECM	Entitlement Control Message
EMM	Entitlement Management Message
FIPA	Foundation for Intelligent Physical Agents
FIPS PUBS	Federal Information Processing Standards Publications
GSM	Global System for Mobile Communication
HMAC	Hash Message Authentication Code
IA-32	Intel Architecture, 32-bit
IANA	Internet Assigned Numbers Authority
ID	Identifier

IEC	International Electrotechnical Commission
IETF	Internet Engineering Task Force
I/O	Input/Output
IPSec	Internet Protocol Security
ISO	International Organization for Standardization
J2ME	Java 2 Platform Micro Edition
KEK	Key Encrypting Key
LDAP	Lightweight Directory Access Protocol
LT	LaGrande Technology
OMA	Open Mobile Alliance
OpenPGP	Open Pretty Good Privacy
OS	Operating System
MAC	Message Authentication Code
MAS	Multi-Agent System
MCC	Model Carrying Code
MExE	Mobile Station Application Execution Environment
MGF1	Mask Generation Function 1
MIDP	Mobile Information Device Profile
MPWG	Mobile Phone Working Group
MS	Mobile Station
MSB	Most Significant Bit
MTPM	Mobile Trusted Platform Module
MVCE	Mobile Virtual Centre of Excellence
NGSCB	Next Generation Secure Computing Base
NIST	National Institute of Standards and Technology
NV	Non-Volatile
OAEP	Optimal Asymmetric Encryption Padding
OCSP	Online Certificate Status Protocol

OEM	Original Equipment Manufacturer
OIAP	Object Independent Authorisation Protocol
OSAP	Object Specific Authorisation Protocol
OSGi	Open Service Gateway Initiative
PAP	Policy Administration Point
PC	Personal Computer
P-CA	Privacy Certification Authority
PCC	Proof Carrying Code
PCMCIA	Personal Computer Memory Card International Association
PCR	Platform Configuration Register
PDA	Personal Digital Assistant
PDAP	Personal Digital Assistant Profile
PDP	Policy Decision Point
PE	Platform Entity
PEP	Policy Enforcement Point
PGP	Pretty Good Privacy
PIP	Policy Information Point
PKI	Public Key Infrastructure
PKIX	Public-Key Infrastructure (X.509)
PMI	Privilege Management Infrastructure
PPC	Place Permission Certificate
PPL	Place Permission List
PROM	Programmable Read-Only Memory
PRNG	Pseudo-Random Number Generator
PRS	Policy Repository Service
PSPL	Portfolio and Service Protection Language
PTEC	Page Table Edit Control

PTR	Private Tamper Resistant
REK	Rights Encrypting Key
RFC	Request For Comments
RI	Rights Issuer
RIM	Reference Integrity Metric
RO	Rights Object
ROAP	Rights Object Acquisition Protocol
ROM	Read Only Memory
RNG	Random Number Generator
RSA	Rivest, Shamir, and Adleman – a public key cryptosystem named after its inventors
RTM	Root of Trust for Measurement
RTR	Root of Trust for Reporting
RTS	Root of Trust for Storage
SAML	Security Assertion Markup Language
SCM	Secure Context Manager
SDL	Standard Deontic Logic
SDR	Software Defined Radio
SDRF	Software Defined Radio Forum
SHA-1	Secure Hash Algorithm revision 1
SIM	Subscriber Identity Module
SPC	Sender Permission Certificate
SPID	Secure Processor Identity
SPL	Security Policy Language or Sender Permission List (depending on context)
SSL	Secure Socket Layer
TBB	Trusted Building Block
TC	Trusted Computing

TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCP	Transmission Control Protocol
TCPA	Trusted Computing Platform Alliance
TCS	TSS Core Services
TCSi	TSS Core Services Interface
TCV	Tick Count Value
TDD	TPM Device Driver
TDDI	TPM Device Driver Interface
TDDL	TPM Device Driver Library
TDDLi	TPM Device Driver Library Interface
TDS	Trusted Domain Server
TE	Tamper Evident
TEM	Trusted Establishment Module
TIR	Tick Increment Value
TLB	Translation Lookaside Buffer
TLS	Transport Layer Security
TMP	Trusted Mobile Platform
TPL	Trust Policy Language
TPM	Trusted Platform Module
TPME	Trusted Platform Module Entity
TRBAC	Temporal Role Based Access Control
TSN	Tick Session Nonce
TSP	TSS Service Provider
TSPi	TSS Service Provider Interface
TSS	TCG Software Stack
TTP	Trusted Third Party
UDP	User Datagram Protocol

URI	Uniform Resource Indicator
URL	Uniform Resource Locator
URN	Uniform Resource Number
UTC	Coordinated Universal Time
VCGen	Verification Condition Generator
VE	Validation Entity
VMM	Virtual Machine Monitor
WAP	Wireless Access Protocol
WTLS	Wireless Transport Layer Security
XML	Extensible Markup Language
XOM	Execute Only Memory
XVMM	XOM Virtual Machine Monitor

Chapter 1

Introduction

Contents

1.1 Motivation and challenges 28
1.1.1 Part I: Mobile host protection 28
1.1.2 Part II: Mobile code protection 29
1.1.3 Part III: Remote code protection 30
1.2 Structure of thesis 31
1.3 Contribution of thesis 35
1.4 List of publications 39
1.5 Cryptographic primitives 40
1.5.1 Hash functions 40
1.5.2 Message authentication codes 41
1.5.3 Symmetric encryption 42
1.5.4 Asymmetric cryptography 42
1.5.5 Asymmetric encryption 43
1.5.6 Digital signatures 43
1.5.7 Public key infrastructure 44
1.5.8 Web of trust 45
1.5.9 Privilege management infrastructure 45
1.5.10 Authentication protocols 46
1.5.11 Freshness mechanisms 46
1.6 Trusted computing primitives 47
1.6.1 Trust 47
1.6.2 Roots of trust 48
1.6.3 Integrity measurement 50
1.6.4 Authenticated boot 50
1.6.5 Secure boot 51
1.6.6 Attestation 51
1.6.7 Sealing 51
1.6.8 Process isolation 52
1.6.9 Secure I/O 53
1.7 Definitions 53

This chapter describes the context of this research, and its contribution to the field of authorisation for mobile code in mobile systems. The structure of the thesis is presented, together with a summary of the main contributions. Definitions and notation used throughout the thesis are also given.

1.1 Motivation and challenges

This thesis is comprised of three parts, each of which tackles an aspect of the problem of authorisation for mobile executables in a mobile environment. In this section we highlight the motivation and challenges behind each of the three aspects.

1.1.1 Part I: Mobile host protection

Mobile agents appear to be a potentially important software paradigm for the mobile domain. If, however, these persistent, autonomous programs are permitted to roam freely in networks, interacting with systems and other agents to fulfil their predefined goals, the risk of a mobile agent with malicious intent damaging any system on which it is executed becomes a very real danger. Thus, determining whether or not a mobile agent should be executed on a particular platform, and with what privileges, is a very serious issue. This becomes especially critical in a mobile environment, where limited bandwidth and processing power may restrict the ability of a mobile node to perform detailed checking on the agent. In part I of this thesis, we examine the authorisation of incoming executables in a mobile environment.

While numerous schemes have been developed to support mobile executable authorisation, and, more specifically, mobile agent authorisation, the solutions proposed often place a substantial burden on the host machine. It is our aim in part I of this thesis to develop a solution which makes minimal use of the end host CPU and storage resources for authorisation procedures. It is moreover required that the solution be flexible enough to be deployed on a range of different mobile platform types, and in a wide variety of operating environments. The proposed solution should support mechanisms which provide assurance re-

garding the origin of the executable, executable code quality and/or the state of an agent. The solution should be scalable, i.e. it should have the ability to either handle growing amounts of work in a graceful manner, or to be readily enlarged [18].

1.1.2 Part II: Mobile code protection

Recent developments in communications systems enable the delivery of complex content to mobile devices. It is expected that the next generation of mobile communications systems will be able to collaborate with broadcast networks to provide wireless access to video content via a wide range of mobile devices [160]. For a service like this to achieve its full commercial potential, the owners of the content will require assurance that their material is not illegally accessed. Current broadcast systems accomplish this by using conditional access systems to ensure that only bona fide subscribers have access to the content.

The Digital Video Broadcast (DVB) organisation has developed several standards defining a common interface to conditional access systems at both the transmission site and at the receiver, while allowing the systems themselves to remain proprietary [22, 44, 45]. Services broadcast today, therefore, are protected by a range of proprietary access control systems. While receivers remain static, and consumers subscribe to one or two service providers, the DVB standards provide a practical solution. However, if a mobile subscriber requires access to services protected by a range of conditional access systems, then the current solutions become impractical. Part II of this thesis proposes a flexible approach that allows consumer products to support a wide range of proprietary content protection systems through the re-configuration of the mobile device to be compatible with the appropriate conditional access system. This proposal is dependent on the proprietary conditional access application being implemented

entirely in software.

We aim to develop a secure method by which a conditional access application can be downloaded to and executed on a mobile device, thereby countering the security threats introduced by re-configurable receivers. These threats include unauthorised reading of the application code and data; unauthorised modification of the application code and data; unknowingly communicating with an unknown and potentially malicious entity; the inability to corroborate the source of the conditional access application; replay of communications; unauthorised reading or modification of any cryptographic keys used in the provision of confidentiality and integrity protection to the conditional access application code and data; and/or unauthorised reading or modification of the application code and data while it executes on the mobile host. The protocols we propose are not intended to supersede or replace the DVB standards or existing conditional access systems. Instead, they are intended to co-exist with existing mechanisms, so that the receipt of digital video broadcast may be achieved more efficiently in a mobile environment.

1.1.3 Part III: Remote code protection

The Trusted Computing Group's Mobile Phone Working Group (TCG MPWG), is currently developing a specification set for a mobile trusted platform module (MTPM). In order to identify the capabilities required of a trusted mobile phone, a number of use cases, whose secure implementation may be aided by the application of trusted platform functionality, have been identified by the MPWG. Among these use cases are SIMLock, device authentication, mobile ticketing, mobile payment and robust digital right management (DRM) implementation [159]. As stated by the MPWG [159], the use cases lay a foundation for the ways in which:

- The MPWG will derive requirements that address situations described in the use cases.
- The MPWG will specify an architecture based on the TCG architecture that will meet these requirements.
- The MPWG will specify the functions and interfaces that will meet the requirements in the specified architecture.

In part III of this thesis we describe the Open Mobile Alliance (OMA) DRM v2 use case and extract the security threats that may impact upon devices, and protected content received by devices, on which an OMA DRM v2 agent is not robustly implemented. This threat analysis enables the derivation of requirements for a robust implementation of OMA DRM v2. Following this, a description is given of the architectural components, based on the TCG architecture, and the functions and interfaces, as specified in the current trusted platform module (TPM) and TCG software stack (TSS) specifications, which meet these requirements. This has enabled the identification of those architecture components and functionality not currently defined within the TCG specification set, but required for the implementation of a robust and secure DRM solution on a trusted mobile platform.

1.2 Structure of thesis

In the remainder of this chapter, the main contributions of this thesis are first described. Following this, the notation, cryptographic primitives and trusted computing primitives used throughout the thesis are specified. Finally, we define the security terminology used in this thesis.

As stated above, the remainder of this thesis is comprised of three distinct

parts. Chapters 2 to 5 (Part I) describe a policy-based framework for the authorisation of mobile executables in a mobile environment. Some of the work presented in part I has been previously published in [53–55, 135]. Chapters 6 to 9 (Part II) describe two protocols for the secure download and execution of a conditional access application to a mobile device. Some of the work presented in part II has been previously published in [56, 58–60]. Chapters 10 to 12 (Part III) describe the functionality required of a trusted mobile platform in order to enable the robust implementation of OMA DRM v2. Some of the work presented in part III has been previously published in [159] and has been used to formulate contributions to the TCG MPWG. Two appendices describe a selection of trusted computing and related technologies.

Chapter 2 introduces the agent paradigm, focusing specifically on the concept of the mobile agent. The security issues surrounding the deployment of mobile agents in a mobile environment are then examined. Finally, the state of the art in technologies for mobile agent authorisation, and more generally mobile code authorisation, are described.

Chapter 3 describes six possible architectural models upon which a policy-based framework for the authorisation of incoming executables and, more specifically, mobile agents, may be constructed. Each model is analysed with respect to the level of security it can support, and with regard to its suitability for implementation in a mobile environment.

In chapter 4, we examine three selected policy statement and attribute certificate specification languages, namely KeyNote, Ponder and SAML, and explore the functionality of their supporting policy engine components. In doing so, we conclude whether these three languages can express the policy statements and attribute certificates required in order to implement the architectural mod-

els described in chapter 3, and support the necessary policy engine component functionality.

Chapter 5 describes a policy-based framework for the authorisation of executables and, more specifically, mobile agents, in a mobile environment.

Chapter 6 examines the DVB standards, which specify two mechanisms designed to provide some flexibility in the application of proprietary conditional access systems to broadcast services [33], and describes certain limitations which arise when they are applied in a mobile environment. In order to overcome these limitations, the mobile platform could be re-configured to be compatible with the appropriate conditional access system, if the proprietary conditional access application is implemented entirely in software. Such a software application could be delivered to the mobile device on demand. The remainder of this chapter explores the threats resulting from the introduction of reconfigurable receivers in a mobile environment, and identifies the security services and security mechanisms required for the protected download of a conditional access application to a mobile receiver.

Chapter 7 describes two protocols designed to meet the security requirements described in chapter 6.

Chapter 8 explores three possible implementations of the generic key exchange and key agreement protocols described in chapter 7. The first implementation assumes the presence of a mobile device into which components described in the TCG version 1.2 specification set are integrated. Following this, we examine the implementation of the protocols given a mobile device architecture into which a version 1.2 compliant TPM and a core root of trust for measurement (CRTM) are integrated and an isolation layer deployed. Finally, protocol implementation given a Next Generation Secure Computing Base (NGSCB)

compliant platform, as described by Microsoft, is explored. Each implementation description is accompanied by an analysis examining how well the security of the downloaded application is protected while in storage and while executing on the mobile device.

In chapter 9, we examine two previously proposed download protocols, which assume a mobile receiver compliant with the execute-only memory (XOM) and AEGIS system architectures, respectively. Both protocols are then analysed against the security requirements given in chapter 6. As a result of these analyses, recommendations are made regarding possible protocol modifications designed to address identified security issues.

Chapter 10 provides an overview of DRM, with particular focus on the OMA DRM standards. The model to which the OMA DRM architecture applies is introduced. A high level critique of OMA DRM version 1 is given, followed by an examination of the OMA DRM version 2 specification set.

In chapter 11 the lifecycle of an OMA DRM v2 agent is considered. Each lifecycle stage is analysed in order to derive a list of security threats that may impact on devices, and protected content received by devices, on which an OMA DRM v2 agent is not robustly implemented. The functionality required of a trusted mobile platform on which an OMA DRM v2 agent is to be robustly implemented, thereby thwarting any threats to the DRM agent and its associated data, is also defined.

In chapter 12 the requirements extracted in chapter 11 are utilised in order to examine which architectural components and functionality described within the TCG version 1.2 specification set may be used to facilitate a robust implementation of OMA DRM v2. This examination also allows us to identify any architecture components and functionality not currently defined within the

TCG specification set but required for the implementation of a robust and secure DRM solution on a trusted mobile platform.

Appendix A examines the specifications for trusted platform functionality produced by the TCG. Microsoft's NGSCB and Intel's LaGrande Technology are also briefly examined.

In appendix B three architectures are examined, each of which have been developed with the goal of providing more secure and trustworthy computing platforms, namely the IBM 4758 and the XOM and AEGIS architectures.

1.3 Contribution of thesis

On examination of state of the art in mobile agent and, more generally mobile code authorisation, we conclude that, while numerous solutions enabling mobile code and agent authorisation have been proposed, the majority of these solutions are ill-suited to application in a mobile environment. Based on this analysis we present a novel policy-based authorisation framework for mobile code and agents. In order to construct this framework, we analyse a variety of architectures upon which a policy-based framework for mobile agent, and more generally mobile code, authorisation could be based. This analysis considers the level of security each could support, and each architecture's suitability for implementation in a mobile environment, and enables us to extract requirements for an optimal architecture model, which include the following.

- Minimal use of the end host's CPU processing power and the end host's storage for authorisation data structures.
- Support for mechanisms which provide assurances regarding the origin of the executable, executable code quality and the state of an agent.

- Incorporation of a policy engine, comprised of a policy administration point, a policy information point, an authentication point, a trust establishment module, a policy decision point, and a policy enforcement point, into each end host.
- Specification, storage and/or processing of policy statements and signed attribute certificates by an end host.

Following this, a critical analysis of a selection of policy specification languages is completed, in order that the most appropriate language can be chosen for use in our policy-based framework for mobile executable authorisation. The analysis of KeyNote, Ponder and SAML gives us new insights into each of the expression languages and their accompanying policy engine components. While KeyNote is a simple and expressive trust management framework, which enables the expression of both policy statements and attribute certificates, issues arose in relation to limiting delegation and the expression of fine-grained access control policies. It also became clear that there is no way to create an attribute certificate in which there is no inherent notion of delegation of authority. While it is claimed by Damianou [37] that Ponder can support similar functionality to that of TPL, this does not appear to be the case since policies cannot be specified in terms of subject or target attributes. While the expression of a large set of different policy types is enabled, and the syntax and semantics of Ponder are reasonably easy to understand, for our particular use case it can only be deployed in conjunction with a trust management or trust establishment mechanism and an attribute certificate expression mechanism. SAML is a simple, expressive language, which meets our requirements for attribute certificate expression. Finally, we present a policy-based framework for mobile agent and, more generally, mobile code authorisation. A modified version of this framework, designed to enable the authorisation of incoming mobile code, has been

integrated into the Software Defined Radio Forum (SDRF) technical specifications [135]. At the time of writing, we are unaware of any other published work with the same scope as the policy-based framework outlined in part I of this thesis.

In part II we propose the deployment of downloadable conditional access systems, so that mobile devices can support an unlimited number of proprietary schemes. This would enable consumers to efficiently access a wide range of video broadcast services within a mobile environment. In order to ensure that this proposal is viable, we identify the threats resulting from the introduction of reconfigurable receivers in a mobile environment, and identify the security services and security mechanisms required for the protected download of a conditional access application to a mobile receiver. We then define two protocols which support the secure download and execution of a conditional access application to a mobile device. Following this, we examine how the two protocols may be implemented using a range of trusted computing technologies, namely the TCG specification set, the TCG specification set in conjunction with an isolation layer, or an NGSCB system. Both of the protocols and each of the possible protocol implementations are analysed in terms of the security services they were designed to meet. While both protocols meet the required security services for secure application transmission, meeting the security requirements for secure application storage and execution is dependent on the trusted computing architecture upon which the protocols are implemented. Completion of this analysis allowed us to conclude that, while TCG defined components enable the implementation of a more trusted platform, in order to take full advantage of the functionality provided, additional software and hardware elements must be introduced, for example the integration of an isolation layer, and/or the modification or extension of the platform's CPU and chipset. Finally, we examine

the download protocols proposed by the designers of the XOM and AEGIS architectures, Lie et al. and Suh et al. We then analyse both of these protocols against our pre-defined set of security requirements for the secure download and execution of a conditional access system. As a result of this analyses, we uncover security shortcomings in both protocols, which appear to arise for two main reasons. Firstly, the sets of requirements used to develop the protocols appear to be incomplete. Secondly, both sets of authors focus on ensuring that their architectures and download protocols support the copy and tamper resistant execution of software rather than the copy and tamper resistant download and execution of software. We subsequently propose a series of enhancements to the protocols designed to address the identified shortcomings. At the time of writing, the key exchange secure download protocol described in part II of this thesis is being considered for integration into the security architecture for software defined radio.

In part III we describe the OMA DRM v2 use case, a shortened version of which is included in the TCG MPWG Use Case Scenarios document [159]. We then describe the threats that may impact on devices, and protected content received by devices, on which an OMA DRM v2 agent is not robustly implemented. The functionality required of a trusted mobile platform in order to thwart these threats is also defined. These requirements have been included in an internal TCG MPWG Requirements document. Finally, we examine the architectural components and functionality within the TCG version 1.2 specifications that can be used to facilitate a robust implementation of OMA DRM v2. This examination has allowed us to identify architecture components and functionality not currently defined within the TCG specification set but required in order to support a robust implementation of OMA DRM v2, namely, secure boot and run-time integrity protection mechanisms. This result will enable the

designers of future trusted mobile platforms to produce systems capable of robustly supporting DRM. This work has contributed to the MPWG MTPM specifications, which are not yet published.

1.4 List of publications

- Eimear Gallery. Towards a Policy-Based Framework for Mobile Agent Authorisation in Mobile Systems. In *Proceedings of the 4th International Conference on 3G Mobile Communication Technologies (3G 2003)*, number 494 in IEE Conference Publication, pages 13–18, Savoy Place, London, UK, 25–27 June 2003. The Institute of Electrical Engineers (IEE), London, UK [1].

- Eimear Gallery. Mobile Agent and Mobile Code Authorisation in Mobile Systems: A Policy-Based Authorisation Framework. In *Proceedings of the 10th Wireless World Research Forum Meeting (WWRF 10)*, New York, USA, 27–28 October 2003. Wireless World Research Forum (WWRF).

- Eimear Gallery. A Policy-Based Authorisation Framework for Software Download. In *Proceedings of the 2nd Software Defined Radio Forum Technical Conference (SDR 2003)*, Orlando, Florida, USA, 17–19 November 2003. Software Defined Radio Forum.

- Eimear Gallery and Allan Tomlinson. Conditional Access in Mobile Systems: Securing the Application. In *Proceedings of the 1st International Conference on Distributed Frameworks for Multimedia Applications (DFMA 2005)*, pages 190–197, Besancon, France, 6–9 February 2005. IEEE Computer Society.

[1] This paper received the Nokia "best paper prize"

- Eimear Gallery and Allan Tomlinson. Protection of Downloadable Software on SDR Devices. In *Proceedings of the Software Defined Radio Forum Technical Conference (SDR 2005)*, Orange County, California, USA, 14–18 November 2005. Software Defined Radio Forum.

- Eimear Gallery and Allan Tomlinson. Secure Delivery of Conditional Access Applications to Mobile Receivers. In Chris J. Mitchell, editor, *Trusted Computing*. IEE Professional Applications of Computing Series 6, chapter 3, pages 29–114, The Institute of Electrical Engineers (IEE), London, UK, April 2005.

- Eimear Gallery. An Overview of Trusted Computing Technology. In Chris J. Mitchell, editor, *Trusted Computing*. IEE Professional Applications of Computing Series 6, chapter 7, pages 197–328, The Institute of Electrical Engineers (IEE), London, UK, April 2005.

1.5 Cryptographic primitives

In this section we briefly describe the fundamental cryptographic primitives and notation used throughout this thesis.

1.5.1 Hash functions

A hash function is a computationally efficient one-way function mapping binary strings of arbitrary length to binary strings of some fixed length, called hash-values [101]. A hash value, h, is generated by a hash function H, so that we write

$$h = H(Z)$$

where Z is an arbitrary length binary string and h is a fixed-length binary string. The purpose of a hash function is to produce a 'fingerprint' of a file, message or

block of data. There are a variety of well-established hash functions, (see, for example, [38, 101, 128, 142]).

A hash function may have the following properties [101].

- Pre-image resistance — given any hash value, h, for which a corresponding input is not known, it is computationally infeasible to find an input, Z, such that $H(Z) = h$.
- 2nd pre-image resistance — given an input, Z, it is computationally infeasible to find a second input, Z', where $Z \neq Z'$, such that $h(Z) = h(Z')$.
- Collision resistance — it is infeasible to find two distinct inputs, Z and Z', such that $H(Z) = H(Z')$.

Throughout this thesis we assume that hash functions fulfil the three security requirements defined above.

Merkle hash trees are often used to protect the integrity of dynamic data in untrusted storage [102, 144]. To construct a Merkle hash tree, each binary string to be recorded in the Merkle tree is initially hashed. The resulting hash values are divided into a number of groups. The hash values in each group are then concatenated and hashed to output a parent hash value. Once these hash values have been calculated, the resulting hash values are iteratively divided into groups, concatenated and rehashed in a tree-like fashion until a single 'root hash value' is created [102]. This 'root hash value' must be integrity-protected.

1.5.2 Message authentication codes

Message authentication codes (MACs) are designed to guarantee the source and integrity of a message. A MAC is sent together with the message it is protecting.

For the purposes of this thesis

$$\mathrm{MAC}_K(Z)$$

denotes a MAC computed on data Z using the secret key K.

There are a variety of well-established means for computing MACs, typically either based on the use of a block cipher or a cryptographic hash function (see, for example, [38, 101, 128, 142]). There are also standards for such schemes, including, most notably, ISO/IEC 9797 parts 1 and 2 [82, 83]. Standards for MACs are discussed in [38].

1.5.3 Symmetric encryption

Symmetric or secret key encryption uses a secret key and an algorithm to transform a plaintext message into ciphertext, i.e. to encrypt the plaintext. The same key is used to decrypt the ciphertext into the original plaintext. For the purposes of this thesis

$$E_K(Z)$$

denotes the symmetric encryption of data string Z using secret key K. We use $K_{X,Y}$ to denote a secret key shared between X and Y, e.g. to be used to compute a MAC or encrypt transferred data.

Many symmetric encryption algorithms have been proposed (see, for example, [38, 101, 128, 142]). Standards for symmetric ciphers are discussed in [38].

1.5.4 Asymmetric cryptography

Asymmetric cryptography, or public-key cryptography, involves the assignment of two distinct keys, one public and one private, to each entity. For the purposes of this thesis P_X denotes the public key of X and S_X denotes the private key of X.

The private key is kept secret by its owner, while the public key can be freely distributed. It must be ensured that the correct public key can be associated with a user. This may, for example, be accomplished using mechanisms described in sections 1.5.7 and 1.5.8. Asymmetric cryptographic schemes include encryption schemes and digital signatures.

1.5.5 Asymmetric encryption

In an asymmetric encryption scheme, the public key is used for encryption and the private key for decryption. For the purposes of this thesis

$$E_{P_X}(Z)$$

denotes the asymmetric encryption of data string Z using the public key, P_X, of entity X. Algorithms for public-key encryption may be found in [38, 101] and include RSA (PKCS #1 [93]). Standards describing how to use asymmetric encryption include [71] and are discussed in greater detail in [38].

1.5.6 Digital signatures

Digital signatures are used to guarantee the origin and integrity of a message. In a digital signature scheme, the private key is used for signing and the public key is used for digital signature verification. For the purposes of this thesis

$$S_X(Z)$$

denotes the digital signature of X on the data string Z.

There are two main types of digital signature scheme. Digital signature schemes with appendix require the original message as input to the verification algorithm, which is used to verify that the signature is authentic [101]. Digital signature schemes with message recovery do not require the original message as

input to the verification algorithm [101]. Instead, the original message is recovered from the signature. We assume the use of a digital signature schemes with appendix throughout this thesis. There are many signature schemes available (see for example [101]), including a number of techniques which are international standards; see, for example, the Digital Signature Standard [48], which contains a scheme called the Digital Signature Algorithm (DSA), RSA [93], and ISO/IEC 14888 [77–79], which describes digital signatures with appendix.

1.5.7 Public key infrastructure

In a public key infrastructure (PKI), certification authorities (CAs) issue digitally signed certificates which bind a public key to an identity and possibly other information (e.g. the certificate expiry date). X.509 [89] is a widely adopted standard specifying the format of public key certificates. The structure of an X.509 v3 public key certificate is as follows:

```
Certificate
    Version
    Serial Number
    Algorithm ID
    Issuer
    Validity
      Not Before
      Not After
    Subject
    Subject Public Key Info
        Public Key Algorithm
        Subject Public Key
    Issuer Unique Identifier (Optional)
    Subject Unique Identifier (Optional)
    Extensions (Optional)
Certificate Signature Algorithm
Certificate Signature
```

A self signed certificate is one in which the issuer and the subject are the same. In order to trust a self-signed certificate its origin and integrity must be

guaranteed by external means. For the purposes of this thesis

$$\mathrm{Cert}_X$$

denotes the public key certificate for entity X. Apart from X.509, standards also exist for PKIs, see for example IETF PKIX[2]. For further details of PKI and PKI related standards, see [38].

1.5.8 Web of trust

A web of trust is an alternative method by which a public key can be bound to an identity. In a web of trust, any user can sign an identity certificate (which includes a public key, an identity and, potentially, other key owner information such as an expiry time) and in doing so endorses the association between the identity and the public key contained within the certificate. A key owner may acquire an unlimited number of identity certificates endorsing the link between the key owner's identity and public key. These certificates are then used by entities within the system in order to determine whether they trust that a particular public key belongs to a specific identity. For further information see [142]. The web of trust concept is deployed in PGP [169], OpenPGP [19] and GnuPG[3] compatible systems.

1.5.9 Privilege management infrastructure

In a privilege management infrastructure (PMI), attribute authorities issue digitally signed attribute certificates. An attribute certificate (AC) may contain attributes that specify group membership, role, security clearance, or other authorisation information associated with the AC holder [46]. Standardised formats for attribute certificates include X.509 [74] and SAML attribute asser-

[2] http://www.ietf.org/html.charters/pkix-charter.html
[3] http://www.gnupg.org/

tions [112–118].

1.5.10 Authentication protocols

A unilateral authentication protocol provides one entity with assurance of the other's identity but not vice versa. A mutual authentication protocol provides both entities with assurance of each other's identities. For the purposes of this thesis Id_X denotes the identity of entity X.

International standards which describe authentication protocols include ISO/IEC 9798-2 [83], ISO/IEC 9798-3 [85], ISO/IEC 9798-4 [86] and ISO/IEC 9798-5 [87]. For further information on authentication protocols, see [38] and [101].

1.5.11 Freshness mechanisms

There are two main methods for freshness checking [38];

- the use of time-stamps; and
- the use of nonces.

If a timestamp is included in a message which is cryptographically protected, the recipient can check if it is fresh [38]. In order to enable the use of this freshness method, both the sender and the recipient must have securely synchronised clocks [38]. Alternatively, each pair of communicating entities may maintain a pair of counters, C_{AB} and C_{BA}. Each time A sends a message to B the value of C_{AB} is included in the message and the counter value is incremented by A. When B receives the message, the sequence number contained in the message is compared to the value of C_{AB} stored by B. If the sequence number contained in the message received from A is less than or equal to the value of C_{AB} stored by B, the message is then rejected as old. If the sequence number contained in

the message received from A is greater than the value of C_{AB} stored by B, then the message is accepted as fresh, and the value of C_{AB} stored by B is set to the value of the received value.

Alternatively, nonces may be used to provide freshness. A nonce is a random number that is included in a challenge message sent to B by A. The nonce is then included in the response message sent to A by B in order to demonstrate that the response could only have been generated after the nonce had been received by B. For the purposes of this thesis R_X denotes a nonce generated by by entity X. For further information see [38].

1.6 Trusted computing primitives

In this section we briefly describe the fundamental trusted computing primitives and notation used in this thesis. A more extensive description of trusted computing technology is given in Appendices A and B.

1.6.1 Trust

In the context of trusted computing, a platform is trusted if it "behaves in an expected manner for an intended purpose" [149]. This does not necessarily imply, however, that a trusted platform (TP) is a secure platform. For example, if an entity can determine that a platform is infected with a virus, whose effects are known, the platform can be trusted by that entity to behave in an expected but malicious manner [65]. In order to implement a platform of this nature, a trusted component, which is usually in the form of built-in hardware, is integrated into a computing platform [5]. This trusted component is then used to create a foundation of trust for software processes running on the platform [5].

The concept of trust has been explored in greater depth by Balacheff et

al. [5], who classify trusted platform components into two groups, namely those that satisfy the behavioural definition of trust and those that fulfil the social definition of trust.

1. Social trust is static:
    - it provides a means of knowing whether platform components should be trusted; and also
    - provides evidence as to whether platform components are capable of behaving properly.
2. Conversely, behavioural trust is dynamic:
    - it provides a means of knowing whether platform components can be trusted; and
    - it results from the dynamic collection of behavioural evidence.

1.6.2 Roots of trust

A root of trust is defined as a component that must be trusted for a platform to be trusted [149]. Within the TCG, three roots of trust are defined upon which a trusted platform can be built, the root of trust for measurement (RTM), the root of trust for storage (RTS) and the root of trust for reporting (RTR). A description of these roots of trust can be found in [5]. Standards describing the RTM and RTS include [149, 156–158].

1.6.2.1 RTM

The RTM is an engine capable of measuring at least one platform component, and hence providing an integrity measurement, as described in section 1.6.3. The RTM is typically implemented as the normal platform engine controlled by

a particular instruction set (the so-called 'core root of trust for measurement' (CRTM)). On a PC, the CRTM may be contained within the BIOS or the BIOS boot block (BBB), and is executed by the platform when it is acting as the RTM.

1.6.2.2 RTS and RTR

The RTS is a collection of capabilities which must be trusted if storage of data inside a platform is to be trusted [5]. The RTS is capable of maintaining an accurate summary of integrity measurements made by the RTM, i.e. condensing integrity measurements and storing the resulting integrity metrics, as described in section 1.6.4. The RTS also provides integrity and confidentiality protection to data and enables sealing, which is described in section 1.6.7. In conjunction with the RTM and RTS, an additional root of trust is necessary for the implementation of platform attestation, which is described in section 1.6.6, namely the RTR. The RTR is a collection of capabilities that must be trusted if reports of integrity metrics are to be trusted [5]. The RTR and the RTS constitute the minimum functionality that should be provided by a trusted platform module. A TPM is typically implemented as a tamper-evident chip which must be uniquely bound to a platform. In order to support RTS and RTR functionality, a TPM incorporates various functional components such as: I/O; non-volatile storage; a minimum of 16 platform configuration registers (PCRs), which are used by the RTS to store the platform's integrity metrics; a random number generator; a hash engine; key generation capabilities; an asymmetric encryption and digital signature engine; an execution engine; and an opt-in component.

If the roots of trust, essentially the TPM and the CRTM, once integrated in a platform, are to be implicitly trusted, then there is an obvious need for validation that they are working as expected. A set of credentials, including an endorsement credential, conformance credentials, a platform credential and

attestation identity credential(s) must be generated for each TCG-conformant trusted platform. This credential set describes the properties of the trusted platform, as endorsed by a set of trusted third parties, thereby providing social trust in the certified trusted platform components and properties.

1.6.3 Integrity measurement

An integrity measurement is defined in [104] as the cryptographic digest or hash of a platform component. For example, an integrity measurement of a program can be calculated by computing the cryptographic digest or hash of its instruction sequence, its initial state (i.e. the executable file) and its input.

1.6.4 Authenticated boot

An authenticated boot process represents the process by which a platform's configuration or state is reliably measured, and the resulting measurement is reliably stored. During this process, the integrity of a pre-defined set of platform components is measured, as defined in section 1.6.3, in a particular order. These measurements are condensed to form a set of integrity metrics which are stored in a tamper resistant log. A record of the platform components which have been measured is also stored on the platform. The TCG define two fundamental roots of trust necessary for the implementation of an authenticated boot process, namely the RTM and the RTS. For the purposes of this thesis, I_2 denotes the summary of the set of measurements stored to the tamper resistant log, i.e. the integrity metrics of the platform. These integrity metrics represent the current state of the platform.

1.6.5 Secure boot

A secure boot is the process by which the integrity of a pre-defined set of system components is measured, as described in section 1.6.3, and these measurements then compared against a set of expected measurements which must be securely stored and accessed by the platform during the boot process. A secure boot process is not defined by the TCG. The concept of a secure boot has been widely discussed in the literature, however, most notably by Tygar and Yee [161], Clark [27], Arbaugh, Farber and Smith [4] and Itoi et al. [88].

1.6.6 Attestation

Attestation is the process by which a platform can reliably export evidence of its identity and its current state (i.e. the integrity metrics which have been stored to the tamper resistant log, and the record of the platform components which have been measured, as described in section 1.6.4) [7]. In conjunction with the RTM and RTS, as described in section 1.6.4, the TCG define an additional root of trust which is necessary for the implementation of platform attestation, namely, the RTR.

1.6.7 Sealing

Sealing represents the action of associating stored data with a set of integrity metrics representing a particular platform configuration, and encrypting it. The data can only be decrypted and released when the state of the platform is the same as that indicated by the integrity metrics sealed with the data. For the purposes of this thesis $\text{Seal}_I(Z)$ denotes the result of the encryption of data Z concatenated with integrity metrics, I, such that Z can only be deciphered and accessed if the platform is in a specified software state, where I includes I_1 the state that the platform must be in if subsequent use of the protected object is

to be permitted, and I_2, the state of the execution environment at the time that the command causing the data to be sealed was issued.

1.6.8 Process isolation

Process isolation provides assured memory space separation between processes [7]. Both software and hardware mechanisms have been proposed in order to enable process isolation.

An isolation layer has been defined as "providing separate execution environments for operating systems, applications and applets" [104]. Implementations include those based on virtual machine monitors [61] and para-virtualisation techniques [6, 127, 132]. Microsoft has also proposed an isolation layer, as described in [24, 43, 126]. This latter isolation layer has been designed as part of a complete platform architecture called the NGSCB. Within this architecture, CPU and chipset extensions are required so that the isolation layer may be physically protected in a separated environment. The isolation layer, in turn, provides separate execution environments for operating systems, applications and/or applets [104]. The CPU and chipset extensions required in order to implement Microsoft's NGSCB have been implemented by Intel as part of their LaGrande Technology (LT) project [72]. These components support protected execution (including prevention of DMA attacks) in an IA-32 platform.

While the execute-only memory (XOM) architecture and the architecture for tamper evident and tamper resistant processing (AEGIS) are not strictly examples of trusted computing platforms, they provide strong process isolation through the development of hardened processors. The XOM architecture proposed by Lie et al. [98, 99] attempts to fulfil two fundamental objectives: the prevention of the unauthorised execution of software; and the prevention of any software consumer from examining protected executable code. This is achieved

through the provision of on-chip protection of caches and registers, protection of cache and register values during context switching and on interrupts, and confidentiality and integrity protection of application data when transferred to external memory.

The architecture for a single chip AEGIS processor bears a strong resemblance to the XOM architecture described above. "AEGIS provides users with tamper evident (TE) authenticated environments in which any physical or software tampering by an adversary is guaranteed to be detected, and private and authenticated tamper resistant (PTR) environments, where, additionally, the adversary is unable to obtain any information about software and data by tampering with, or otherwise observing, system operation" [143].

1.6.9 Secure I/O

Secure I/O allows applications to be assured regarding the end-points of input and output operations [7]. Intel have developed keyboard, mouse and graphic subsystem enhancements as part of the LaGrande Technology project [72] in order to facilitate this.

1.7 Definitions

In this section, security terminology used in the reminder of this thesis is defined. This is by no means intended to be a complete list of security terminology.

Access control: The means of enforcing authorisation [51].

Authorisation: The granting of rights by the owner or controller of a resource, for others to access that resource [51].

Authorisation data: The data that must be presented for access to be allowed [5].

Availability: The property that resources, including data and processing, are accessible to authorised entities [128].

Attribute authority: An authority trusted to create and assign attribute certificates [74].

Certification authority: An authority trusted to create and assign public key certificates. Optionally, the certification authority may create and assign keys to the entities [75].

Confidentiality: The property that data is not made available or disclosed to unauthorised individuals, entities, or processes [80].

Data integrity: The property that data has not been altered or destroyed in an unauthorised manner [80].

Data origin authentication: The corroboration that the source of data received is as claimed [80].

Digital certificate: A digitally signed data structure containing an identifier for an entity and certain information associated with that entity, e.g. a public key or an access control attribute [47].

Entity authentication: The corroboration that an entity is the one claimed [84].

Integrity measurements: Measurements about the state of the platform, used in the process of deciding whether a platform can be trusted, condensed to make integrity metrics, usually associated with explanatory history information [5].

Integrity metrics: Data that is a condensed value of integrity measurements, which indicate the history of the platform that relates to the platform's trustworthiness [5].

Non-repudiation of origin: Protects against the originator's false denial of having approved the content of a message and of having sent a message [76].

Non-repudiation of receipt: Protects against a recipient's false denial of having received a message [76].

Roots of trust: In TCG systems, roots of trust are components that must be trusted because their misbehaviour might not be detected [149]. A complete set of roots of trust has at least the minimum functionality necessary to describe the platform characteristics that affect the trustworthiness of the platform [149].

Root of trust for measurement: A collection of capabilities that must be trustworthy if integrity measurements of the environment inside a platform are to be trusted; the component that makes the first integrity measurement of a platform [5].

Root of trust for storage: A collection of capabilities that must be trustworthy if storage of data inside a platform is to be trusted; part of the functionality of the TPM [5].

Root of trust for reporting: A collection of capabilities that must be trustworthy if reports of integrity measurements of the environment inside a platform are to be trusted; part of the functionality of the TPM [5].

Trusted system or component: A system or component whose failure can break the security policy [3].

Trustworthy system or component: A system or component that will not fail [3].

Trusted third party (TTP): A security authority, or its representative, trusted by other entities with respect to security related activities [78].

Security policy: A set of rules that apply to all security-relevant activities

in a domain [51].

Part I

Mobile host protection

Chapter 2

Mobile code and agent authorisation

Contents

2.1 Introduction 59
2.2 Agents 62
2.3 Mobile agents 64
2.4 Mobile agent security 64
2.5 Mobile agent authorisation techniques 66
2.5.1 Code and agent behaviour 66
2.5.2 Code and agent origin 72
2.5.3 Code and agent integrity 76
2.6 Conclusions 80

This chapter introduces the agent paradigm, focusing specifically on the concept of the mobile agent. The security issues surrounding the deployment of mobile agents in a mobile environment are then examined. Finally, the state of the art in technologies for mobile code and agent authorisation is described.

2.1 Introduction

As the benefits of mobile computing become more apparent, the deployment and use of wireless technologies increases. Despite the fact that mobile computing has become progressively more prevalent, Gray et al. [66] have identified several remaining obstacles to the development of distributed applications that make effective use of networked resources from mobile platforms. The fact that mobile computers do not have a permanent connection and are often disconnected for long periods of time is an obvious obstacle, as is the fact that connections are often characterised by low bandwidth, high latency and may be error prone.

A mobile agent is defined as "an autonomous, reactive, goal-oriented, adaptive, persistent, socially aware software entity, which can actively migrate from host to host" [131], and their deployment leads to considerable potential benefit for both distributed applications and mobile computing. The deployment of mobile agents can decrease the bandwidth required in order to complete a transaction between a client and a server. In general, a transaction between a client and a server may require many round trips to ensure its completion. If many transactions are being completed concurrently, the communications requirements may exceed the available bandwidth, leading to poor performance. When an agent is used, the only transfer made is that of the agent from the client to the server. The transfer of all intermediate results is eliminated, thereby reducing the overall bandwidth requirement.

In the traditional client/server architecture, the roles of the client and the server are often pre-defined, and decisions are often made as to where particular functionality will lie during design. If incorrect or inaccurate decisions are made at design time, an inefficient system will result. By contrast, however, agent systems require very few decisions to be made at design time, making the system

more flexible in operation.

Agents also provide a solution to the problem of unreliable network connections. In the majority of systems in use today it is necessary that a network connection remains for the duration of a transaction. Should the network connection be lost, the client is required to re-establish the session and begin the transaction again. If an agent is deployed, a connection is only required for agent transmission and for the receipt of results. If the connection is lost at any point after the agent is transmitted, the agent will continue to do its work and return with the results when the connection is restored. This shows that the use of a mobile agent can also reduce the amount of online time required in order to complete a transaction [111].

If, however, these persistent, autonomous programs are permitted to roam freely in networks, interacting as they please with systems and other agents in order to fulfil their predefined goals, the risk of a mobile agent with malicious intent damaging any system on which it is executed, becomes a real danger. It therefore becomes apparent that the problem of determining whether or not a mobile agent should be authorised to execute on a particular platform is a serious one. This problem becomes especially critical in a mobile environment, where bandwidth and processing power may often be limited; this, in turn, restricts the capabilities of a mobile node either to contact the originator of the agent or to perform detailed checking of the mobile agent code.

In part I of this thesis, we develop a policy-based framework for mobile agent authorisation, and more generally, mobile code authorisation, for implementation within a mobile environment. Part I is structured as follows.

- In this chapter we examine the agent paradigm, with particular focus on the mobile agent. The security issues surrounding the deployment of

mobile agents in a mobile environment are then explored. Following this, state of the art technologies in mobile code and agent authorisation are described.

- Chapter 3 describes six possible architectural models upon which a policy-based framework for the authorisation of incoming executables and, more specifically, mobile agents, may be constructed. Each model is then analysed with respect to the level of security it can support and with regard to its suitability for implementation in a mobile environment. From this analysis, we outline a list of features desirable in the underlying architectural model of a policy-based framework for mobile code and agent authorisation.

- In chapter 4 the six possible architectural models, upon which a policy-based framework may be constructed, are re-examined. As a result we deduce the functional requirements for the language chosen for policy statement expression, the language chosen for attribute certificate expression, and the supporting policy engine, in each of the six scenarios. Following this, a selection of policy and attribute certificate specification languages are examined using these requirements, namely KeyNote, Ponder and SAML, and conclusions drawn regarding their suitability for use in our framework.

- Finally, in chapter 5, our policy-based framework for mobile code and mobile agent authorisation is described.

In section 2.2 the properties that constitute an agent are specified. Following this, a variety of well defined agent types are described. Section 2.3 focuses specifically on the mobile agent paradigm.

Section 2.4 outlines the security issues surrounding the deployment of mobile

agents in a mobile environment. The threats specific to host security, which arise from the circulation of malicious mobile agents, are then explored.

Section 2.5 examines the state of the art technologies for mobile code and mobile agent authorisation. The authorisation techniques identified are grouped into three categories: authorisation techniques based on code and agent behaviour; authorisation techniques based on code and agent origin; and, finally, authorisation techniques based on code and agent integrity.

2.2 Agents

By examining the various agent definitions gathered by Franklin and Graesser [52], we can extract certain generic properties that characterise an agent. Ideally, agents are:

- Autonomous in their execution. They require no human or machine intervention as regards execution. They have control over their actions.
- Perceptive with respect to their environment.
- Reactive, and thereby responsive to changes in their environment.
- Able to realise a set of predefined goals.
- Persistent software entities, unlike those that run and then come to an end.
- Interactive with respect to other agents.
- (Sometimes) adaptive, where their behaviour changes based on previous experience.

When compared to conventional software, there are two main differences [52]:

- The output of a program does not normally affect what it senses later; and
- Most programs run once and then terminate until called on again, whereas agents have temporal continuity.

Agents are programs, but not all programs are agents.

In conjunction with the properties highlighted above, an agent may possess other attributes which allow it to be placed into one or more of the following groups [111].

- Collaborative agents are capable of cooperating with other agents in order to achieve their goals. They can negotiate between themselves in order to reach mutually acceptable agreements.
- Interface agents are capable of learning. Over time they adapt to their user's preferences.
- Mobile agents are capable of roaming.
- Information agents are capable of managing and collating information from many different sources.
- Reactive agents react to the environment into which they are put.
- Smart agents are defined as agents capable of both cooperation and learning.
- Hybrid agents are agents that contain two or more of the above characteristics.

2.3 Mobile agents

In this thesis, we focus on the mobile agent paradigm. In order to understand more deeply the focus of our study, we adopt a widely used definition of a mobile agent as "a process that can actively migrate from one host to another host, and based on locally computed decisions, can actively migrate to a third host" [131].

2.4 Mobile agent security

In practice, neither the agents nor the machines on which they execute are necessarily considered trustworthy, see [28,91]. A malicious host can pose a serious threat to an agent. If appropriate measures are not taken, a malevolent machine on which an agent is executing may attempt to gain access to confidential information stored by the agent, alter the information held by the agent, or force the agent to carry out unauthorised actions.

On the other hand, a malicious agent may pose numerous threats to a host machine. It may use up an unauthorised amount of resources leading to a denial of service attack, or it may attempt to access, destroy or disseminate secret information stored on the host machine, if the appropriate access control mechanisms are not put in place.

In part I of this thesis, we focus primarily on the latter problem, i.e. malicious agents and host security. The confidentiality, integrity and availability of data held on a host system must be ensured. Should malicious or corrupt agents come into circulation, however, the successful provision of these security services by the host may be threatened. The threat of downloading and executing malicious, corrupted or incorrect program code from anywhere on the Internet has been widely documented and discussed. The evolution of agent technologies has, however, given rise to some new security challenges. In the

case of the mobile agent there is the added concern that agent state information — program counters, registers, local environment, control stack, store, etc. — may possibly have been changed in ways that adversely impact the agent's functionality [11]. Indeed, an initially benign agent may become malicious in its behaviour.

Attacks that may be mounted by malicious mobile code include:

- Unauthorised access to resources, which may lead to the deletion of user files or to the leakage of sensitive information.
- Flooding attacks caused by the replication of programs, which potentially wreak havoc on networks and may crash distributed systems.
- Unauthorised monitoring of the execution environment.
- Modification or deletion of the host configuration.
- Insertion of back doors into a system, leading to possible future security violations.

We must also consider the possibility of a malicious host that manipulates the state information of the agent in order to attack hosts that the agent subsequently visits, or, simply, to change the agent behaviour in ways advantageous to the host. Berkovits, Guttman and Swarup [11] give an example involving two independent airline servers, a travel agency server and a consumer server. The travel agency is responsible for programming an agent, which is then distributed by the customer. The agent is designed to query airline databases to find details of seat availability and cost. When queries on one server have been completed and the results have been stored in the agent state, the agent moves on to query the next server. When all servers have been visited, the agent compares flight and fare information, chooses the most appropriate option, and returns to the

chosen airline server to reserve the flights. Finally, the agent will return to the customer with the results.

If we assume that the number of reservations requested by the consumer is stored as part of the state information of the agent, the first airline can mount a simple attack on the second airline by increasing the number of reservations requested by the customer (from 2 to 100 for example) causing the agent to reserve 100 seats on the second (perhaps cheaper) airline. This will have the effect that subsequent customers will book their tickets on the first airline, as the second believes it is full and that it cannot sell any further tickets. This illustrates the ease with which state information can be corrupted in order to inflict economic damage. What is also highlighted here is the importance of protecting any immutable state information — in this case the number of seats specified by the user — as well as the source code of the agent.

2.5 Mobile agent authorisation techniques

In order to prevent a mobile agent, and more generally mobile code, from performing unauthorised actions on a host machine, the development of an authorisation framework is required. Within this framework, we need to devise a method of deciding whether or not an incoming executable should be authorised to execute, and what privileges (if any) should be assigned to an agent authorised to execute. We will begin this process with an examination of existing work on mobile code and mobile agent authorisation.

2.5.1 Code and agent behaviour

Recent approaches to mobile code and mobile agent authorisation analyse the executable's behaviour to determine the appropriate level of access to be at-

tributed to it. In this section, we will consider approaches of this type, including proof-carrying code and model-carrying code, as well as some of the security features built into certain languages.

2.5.1.1 Proof-carrying code

The objective of proof-carrying code (PCC) is to enable a computer system to "conclude, automatically and with certainty, that program code, provided by another system, is safe to install and execute" [107]. The mechanism involves the code producer providing an encoding of a proof that the code adheres to the security policy of the receiving entity/host. The proof is such that it can be easily transmitted with the code, and checked via a simple proof-checking process. At first glance, this approach seems very promising, offering the following advantages: the properties that can be proved as safe are not limited to a specific set; it is low risk and automatic; it is efficient and flexible; and it does not require the existence of trust relationships.

This particular mechanism may be implemented in the following way [107]. The code producer initially adds annotations to code it produces, either manually or automatically with a tool called a certifying compiler. These annotations help the host understand the security relevant behaviour of the code.

The host, which has specified a security/safety policy, receives the code. The host machine then uses a tool called a verification condition generator (VCGen). This tool first checks all untrusted code for simple security properties. Secondly VCGen watches for instructions that may violate the security policy laid down by the host. From this, a predicate is extracted that highlights the conditions under which the execution of the code is safe. This security predicate (set of verification conditions) is then sent to the 'proof producer' which proves it and sends it back to the host. If the proof is deemed valid by the host, when put

through a program known as the proof checker, the untrusted code is installed and run.

On closer examination, however, it becomes clear that there are some fundamental hurdles accompanying the use of this mechanism, including how proofs are constructed, in what formalism, and how it can be guaranteed that proofs can be verified efficiently and simply. As of yet, it is also true that not all properties can be captured in proofs [138]. Currently, only low level security properties such as memory safety can be dealt with [107]. Also, in the case of large executables, the size of the proof can sometimes be an order of magnitude larger than the code.

2.5.1.2 Model-carrying code

A more recent approach to mobile code authorisation is a method called model-carrying code (MCC) [138]. This technique involves a behavioural model being sent in conjunction with the executable. The model is far less complex than the program code it accompanies, and can therefore be checked with relative ease by the host platform to see if it satisfies the relevant host security policies. As opposed to proof-carrying code, where policies are rigidly set, this model offers the host platform the option of making the relevant changes to the security policy based on the requirements of the code and the behavioural guarantees presented by the model accompanying the code.

There is, however, one remaining issue: how can the model be trusted to be a true reflection of the executable? To address this issue, the deployment of either run-time checking on the code as it is executing, model signing or the use of PCC has been recommended [138]. Model signing may appear to be almost identical to code signing, but in this case the signature attests that the model reflects the behaviour of the code. Therefore, should anything go wrong, it is

easily proved whether or not the signer of the model acted fraudulently. In contrast, should signed mobile code act in a destructive fashion, the signer may be able to claim ignorance that the code could do any harm.

Alternatively, runtime checking may be employed. In this instance, an enforcement model is passed to the runtime monitor, which intercepts security relevant events and compares them to the expected model behaviour. Should the program deviate from the model while executing, it is immediately terminated. It is the aim of this runtime enforcement to thwart the threat of inconsistent mobile code and model production.

In lieu of model signing or runtime checking, a formal proof may be produced in order to prove that the model is a safe approximation of the program's behaviour [138].

2.5.1.3 Language security

A number of recent programming languages incorporate security features such as type safety. Notable examples include Java [64], Safe Tcl [125] and C# [2]. These languages have made a considerable contribution to the security of mobile code in recent years. In this section we will briefly examine some of the security features of Java and Safe-Tcl in more detail, and comment on the contribution each can make to mobile code security.

Firstly, the design of the language in which the executable is written may have a major impact upon host security. With respect to the prevention of integrity violations on the host, issues such as type safety are important. This involves the compiler ensuring that programs do not access memory in ways that are inappropriate or dangerous. As regards a confidentiality violation of the host, there are many ways in which mobile code can leak secrets; for exam-

ple, information channels can arise from contention among processes for shared resources. For more information on these language issues see, for example, Volpano and Smith [163].

Secondly we examine the native security features of various languages, such as Java and Safe-Tcl. In order to address security concerns, Java incorporates a security model referred to as the sandbox. It is the job of the sandbox to ensure that untrusted Java code cannot access sensitive system resources. The sandbox consists of three main components [62].

- The byte code verifier performs static checks on the incoming code before it is executed. These checks are concerned with the structure of the code, such as whether or not all class files have the proper class file format, rather than code behaviour.
- An applet class loader is responsible for downloading the classes which do not already exist on the client machine, but are necessary for a particular Java executable to run. The Java virtual machine then tags each of the classes so that, when a class attempts access to a resource, the security manager can use the class loader tag to determine whether the class has come from the local machine or has been downloaded, and to determine whether the access is authorised.
- The security manager performs dynamic checks on the code while it is executing. It can enforce boundaries between classes to prevent one class from accessing private variables or classes from outside its class, and it is also consulted when any potentially dangerous methods are invoked.

Originally, the only differentiation made in Java was between trusted code originating from the local file system, which was run on the system, and untrusted code, downloaded from the network and executed in the sandbox. In

later editions of the Java developer's kit, signed code was introduced. In this instance, if code could be authenticated via a digital signature as originating from a particular source trusted by the host, then the code could be given free access to the system; otherwise it was confined to the sandbox. The most recent developments introduce multiple sandboxes/zones enforcing differing degrees of restriction. The signature accompanying the code is used as the deciding factor in determining the appropriate sandbox for code execution.

The scripting language Safe-Tcl [125] works on the 'padded cell' approach. In order to execute code, two interpreters are used, a master interpreter and a safe interpreter. Each executable is isolated in a safe interpreter, a restricted virtual machine for the executing code, in which executables have access to a minimum set of commands known as a 'safe base'. All unsafe commands, which may lead to a host security violation, are made inaccessible within the safe interpreter. The execution environment of a safe interpreter is, however, controlled by trusted scripts running in a master interpreter, which retains full functionality.

In order to allow executables limited access to restricted commands, an alias mechanism is used. An alias is used by executable code to request 'unsafe' services or access to protected resources from the safe interpreter, which in turn invokes the commands in the master interpreter.

A different policy, which consists of the 'safe base' and a set of aliases, can then be associated with each safe interpreter. This particular security model has two main strengths. Firstly, trusted code is separated from untrusted code, as was the case in Java. Secondly, Safe-Tcl does not prescribe any particular security policy. Different security policies can be set by the particular organisation by providing different sets of aliases in a safe interpreter. For example, highly

restrictive policies may be set for incoming code produced by unknown or untrusted code producers, and more liberal policies set for code accompanied by a signature that shows it has originated from a trusted source. More information on this particular model is given by Ousterhout, Levy and Welsh [125].

2.5.1.4 Authorisation and attribute certificates

Johnston, Mudumbai and Thompson [92] use two kinds of certificates in their scheme for widely distributed access control.

- An 'authorisation certificate' contains a list of 'use-conditions' that need to be satisfied before access to a resource on a host machine can be authorised.
- An 'attribute certificate' can be associated with a piece of executable code or agent produced and signed by a third party that is trusted by the owner of the host resources.

When an incoming executable requests authorisation to access a particular resource, an access control gateway initially verifies the identity of the requesting subject. This request is then passed to the policy engine, which collects the authorisation certificates of the host machine and the attribute certificates of the particular executable or agent. If the use conditions are satisfied by the attribute certificates, a capability for use of a specified resource by a particular executable is generated.

2.5.2 Code and agent origin

All of the systems we describe in this section rely on cryptographic methods to establish the identity and trustworthiness of the code author, agent creator or owner. In particular, the receiving host will check the validity of a digital signature and/or digital certificate that is associated with the incoming executable.

2.5.2.1 D'Agents

The D'Agents system [67] supports agents written in three different languages, namely Tcl, Java and Scheme. In a D'Agents system, each agent may be classified by a D'Agents host as either owned or anonymous. An owned agent is defined as one whose owner can be authenticated and is included in the D'Agents host's list of authorised users. An anonymous agent is defined as one whose owner cannot be authenticated, or, alternatively, as one whose owner is not included in the D'Agents host's list of authorised users.

An agent owner can be authenticated if the agent has been signed using the owner's private key. Alternatively, an agent owner can be authenticated if the agent has been signed with the agent sender's private key, where the sending machine was able to authenticate the agent owner and is trusted by the receiving host. On arrival, the digital signature accompanying the incoming agent is verified, and the agent classified accordingly as owned or anonymous. If the agent is labeled anonymous, it may be rejected, depending on the host policy.

A D'Agents host machine will also contain a set of security policies. One category of policy that can be enforced on a D'Agents host are absolute policies, for example "an agent can terminate another agent only if its owner is the system administrator or if it has the same owner as the other agent" [67]. These policies must be adhered to by all agents, irrespective of the language in which they are written.

All other policy statements are defined and enforced using language-specific security/enforcement modules and a set of language-independent resource manager agents. When an agent requests access to a resource, the request is forwarded by the language-specific security module to the correct resource man-

ager. The resource managers determine whether access should be allowed or denied, based on a specified security policy. This decision is then enforced by the security/enforcement module. In the version 2.0 implementation of D'Agents [67], there are seven resource managers: a consumable resources manager, a file system manager, a library manager, a program manager, a network manager, a screen manager and a manager for other generic resources. There are also three security/enforcement modules, one for each of the languages listed above. Security policies typically specify access rights and limits for a particular agent owner. Anonymous agents are given very limited (if any) access to resources.

The D'Agents system is based on a PGP-style web of trust, see section 1.5.8, rather than a PKI, see section 1.5.7. Note, also, that once an agent leaves its trusted domain it will be marked as anonymous by any host it visits, and will therefore have very limited capabilities. These characteristics of the D'agents system are likely to limit its use in an open environment — precisely the environment where mobile agents have the most potential.

2.5.2.2 MExE

The mobile station application execution environment (MExE) [140] provides a standardised execution environment for advanced mobile services and applications in mobile equipment. The resources available to an executable are determined by the identity of the executable's owner and the type of the device upon which it executes. Three MExE device types are defined; classmark 1 wireless application protocol (WAP) devices, which have simple man-machine interfaces and limited processing power; classmark 2 Java devices, which have greater power and storage; and, finally, classmark 3 devices, which are based on connect limited device connection (CLDC) and mobile information device pro-

file (MIDP) reference implementations for application development on devices, such as mobile phones and PDAs, that have tight resource constraints. The authorisation mechanisms differ depending on the classmark of the device.

The authorisation framework for classmark 1 and 3 devices is a simple one. Such devices have tight resource constraints, as mentioned above, and thus will not support signed content. All executables downloaded are treated as untrusted and are run in a sandbox with limited privileges.

Classmark 2 devices have a more complex authorisation mechanism. Every device incorporates the root public keys of the device manufacturer, the network operator, the administrator, and any other relevant third parties. There are also four MExE domains available on each device, including a third party domain, an operator domain, a manufacturer domain and one general execution area, which is referred to as the untrusted execution domain. These domains are used for executables signed by the manufacturer of the mobile station, the network operator or an independent third party, with executables signed by the network operator receiving the most flexible set of access rights. In conjunction with domain restrictions, user permission is also required before any action may be performed.

2.5.2.3 Telescript

Telescript [145], the first commercial mobile agent system, also uses the identity of the executable's owner in order to determine the level of access to be attributed to an incoming agent by the host platform.

In conjunction with the inherent runtime object safety inbuilt into Telescript engines, a number of explicit security mechanisms are also incorporated into the architecture. Each agent carries cryptographic credentials which allow the

identity of the agent's owner to be verified. In conjunction with this, a set of permissions, which specify a set of access rights to utilise Telescript instructions and host machine resources [67], are allotted to the agent.

The host machine then gives the incoming agent the permitted access rights. Should an agent attempt to violate its restricted set of permissions, its execution is immediately terminated. This system assumes, however, that all agents are created and utilised within a trusted environment, and therefore does not scale well.

2.5.3 Code and agent integrity

Agents consist of code, data and execution state, and it is the static code and data that is protected when an executable is digitally signed by its author, as described in 2.5.2. Some aspects of an agent's state, however, cannot be signed by the agent code author because of their transient nature. A malicious host may, therefore, change the state of an agent so that the behaviour of the agent is modified in ways advantageous to the host. A number of techniques have been proposed to counter this threat. In this section we examine several such techniques.

2.5.3.1 SPLs, PPLs and state appraisal functions

A state appraisal function is used to evaluate the state of an agent before its execution is authorised. This process involves the host machine evaluating a function to verify that the incoming agent is not in a harmful state. This function is necessarily application-specific, so the most obvious entity to define it would be the agent code author. The author then signs the state appraisal function and the code.

Berkovits et al. [11] highlight a more specific application of the state appraisal

function in their authorisation framework. They place authorisation controls at each point in the life cycle of an agent, from creation of the underlying program, through the creation of the agent itself, to the migration of the agent. The author prepares the source code of the agent and a state appraisal function (denoted *max*), which is used to calculate the maximum set of permissions that should be given to the agent, based on the state of the agent. If this state appraisal function at a later time discovers that the state has been corrupted in some way, the set of permissions the agent was originally to be given is diminished or possibly made null.

A sender permission list (SPL) may also be defined in order to specify which users are permitted to send the resulting agent. All this information — the code, the state appraisal function, the SPL, and the identity of the code author — is hashed to yield a message digest, known as the program name, which the author signs with its private key. If it is not possible to identify all potential senders at the beginning of this process, a sender permission certificate (SPC) can be used in order to add senders to the SPL at a later date. This involves the author of the program signing an attribute certificate containing the identity of the sender in question and the name of the program.

The agent is then constructed by combining the agent code and its execution state. At this stage, the agent creator prepares a second state appraisal function (*req*), which calculates the set of permissions a sender wishes the agent to have while executing. These permissions are calculated as a function of the agent's current state. The output of the function *req* may define a set of permissions that are a subset of the permissions output by the function *max*, as the agent creator may wish to limit its responsibility for any malicious behaviour that the program may attempt.

The creator may also construct a place permission list (PPL), which specifies the places/hosts that can run the agent on the creator's behalf. The creator then takes the agent, the name of the program (calculated by the agent code author), the function *req*, the PPL and the identity of the creator, and creates a message digest from the components. This digest is known as the agent name. The agent creator then signs this digest with its private key. Places that have not been included in the PPL can be authorised by the creator through the generation of an attribute certificate, called a place permissions certificate (PPC), after the agent has been launched.

When the agent arrives at a destination host, the digital signatures accompanying the agent are verified. Once the host has been assured of the identities of the agent code author and agent creator; that the agent's creator's identity is listed in the SPL; that its own identity is listed in the PPL; and that none of the signed elements have been tampered with; the functions *req* and *max* are executed. Should the output from the *req* function be neither equal to nor a subset of the permissions output from the *max* function, no permissions are granted to the agent. Equally, if either of the state appraisal functions detects that the state has been altered in a malicious way, the function will return an empty permission set. Otherwise, permissions requested by the agent are granted.

While this system has a number of desirable features, certain technical issues must be addressed if this mechanism is to be deployed within an authorisation framework. It is not clear how well the theory will hold up in practice, since the state space for an agent could be quite large. While appraisal functions for obvious attacks may be easily formulated, more subtle attacks may be significantly harder to foresee and detect. It may not always be possible to distinguish normal results from deceptive alternatives [91]. In conjunction with this, a practical implementation of such a mechanism has not been documented, so tangible

performance results are not available.

2.5.3.2 Tracing

Tracing [162] was originally developed with agent security in mind, but may also be used to counteract the threat posed by malicious hosts tampering with an agent in an attempt to damage hosts that the agent subsequently visits. Rather than preventing a host maliciously altering an agent in any way, and thereby causing damage to a future host on which the agent may execute, cryptographic tracing acts as a log of all the operations performed by the agent over its lifetime.

Jansen and Karygiannis [91] describe the technique as follows. Each platform on which an agent executes is required to create and retain an undeniable trace of the operations performed by the agent while resident there, and to submit to the agent a cryptographic hash of the trace, known as a trace summary or fingerprint. A trace is composed of a sequence of statement identifiers and signature information from all platforms the agent has visited. The signature of the platform is required only on those instructions that depend on interactions with the computational environment maintained by the platform. For instructions that rely only on the values of internal variables, a signature is not required and is therefore omitted.

The technique also defines a secure protocol to transmit agents and associated security related information between the various parties involved. Therefore, assuming that the creator of the agent is trustworthy, and the original agent was in no way malicious, traces and trace summaries can be used to detect whether a host machine has altered the agent in a detrimental way, and can enable a third party to pinpoint exactly which host is responsible for the damage, thereby acting as a deterrent to tampering.

2.5.3.3 Code obfuscation

Another mechanism devised with agent security in mind, but which by implication has an impact on host security, is code obfuscation [69]. Code obfuscation helps prevent original 'safe' code from being tampered with and thereby affecting and possibly damaging future host machines it may visit. In this case, a series of one or more transformations are applied to the code before it is sent on to another site in such a way that makes re-engineering the code extremely difficult, while still preserving the executable's original functionality [69]. While code obfuscation offers protection against both static and dynamic analysis attacks, and is efficient in terms of the transformation process, which is automatic, the memory and computational time required to execute obfuscated code increases with each transformation applied to the executable [20].

2.6 Conclusions

In this chapter, the mobile agent paradigm has been introduced and the security issues associated with the deployment of mobile agents in a mobile environment have been described. Finally, a review of the state of the art in mobile code and agent authorisation has been presented. While it is clear from this chapter that the problem of mobile code and agent authorisation has been widely discussed, many of the solutions described above are limited in terms of their application in a mobile environment.

While the mechanisms described in section 2.5.1 focus on the behaviour of the code rather than the entity which generated it, which is advantageous in an open environment, PCC, MCC and language security are not without their problems. While PCC is a promising area of research, there are questions regarding how proofs of code should be constructed, in what formalism, and how it can be

guaranteed that proofs can be verified efficiently and simply. Currently, not all properties can be captured in proofs [138], only low level security properties such as memory safety [107]. In the case of large executables, the size of the proof can sometimes be an order of magnitude larger than the code. While the use of languages which incorporate security functionality is clearly helpful, controls then become dependent on all incoming executables being written in one of these 'safe' languages. MCC remains partially dependent on proofs of code or identity-based means in order to ensure model soundness [138], which may introduce another class of problem, as described below.

In the identity-based authorisation mechanisms described in section 2.5.2, the authorisation of an incoming executable is dependent on the identity of the its originator(s). If mobile agents, and indeed mobile code, are to be deployed in an open mobile environment, where a host device may potentially need to authorise an unbounded number of entities, the security value of an identity-based authorisation mechanism quickly diminishes. In conjunction with this, no consideration is given to the threat of a known entity generating a malicious executable, or buggy code which results in an end host system security vulnerability.

Of the mechanisms described in section 2.5.3, tracing facilitates the detection of malicious activity rather than its prevention, which is not ideal, and, while the concept behind code obfuscation is a simple one, there are currently no known algorithms which enable this approach [91]. At the time of writing, state appraisal functions represent the only mechanism by which the security threats, introduced by potentially malicious agent state information, can be thwarted; however, the implementation of such an approach requires a complex verification process to be completed by the end host, which is disadvantageous in a mobile environment, where devices may be limited in terms of processing power.

In the remainder of this part of the thesis we therefore work towards the development of a policy-based framework for the authorisation of mobile executables in a mobile environment, where a minimal set of checks need to be completed by the end device, and both the origin and/or the behaviour of the executable can be considered. In the next chapter, we start this process by describing and analysing a number of architectural models upon which a policy-based framework for the authorisation of incoming executables and, more specifically, mobile agents, may be constructed.

Chapter 3

Architectural models for mobile code and agent authorisation

Contents

3.1 Introduction 84
3.2 Entities involved 86
3.3 Scenario 1 87
3.4 Scenario 2 91
3.5 Scenario 3 93
3.6 Scenario 4 98
3.7 Scenario 5 101
3.8 Scenario 6 104
3.9 Conclusions 107

This chapter describes six possible architectural models upon which a policy-based framework for the authorisation of incoming executables and, more specifically, mobile agents, could be constructed. Each model is analysed with respect to the level of security it can support, and with regard to its suitability for implementation in a mobile environment.

3.1 Introduction

In chapter 2, the state of the art in technologies for mobile code and agent authorisation was examined. As we are concerned with mobile code and agent authorisation in a mobile environment, for the remainder of part I of this thesis we will focus on the development of a policy-based authorisation framework, which provides both mobile devices and service providers with the ability to assign the appropriate privileges to incoming executables and, more specifically, mobile agents. This framework is designed to avoid the use of computationally intensive procedures and to operate in a non-overly restrictive manner.

This chapter documents the preliminary analytical work necessary in the construction of this policy-based authorisation framework for a mobile environment. The framework's underlying architectural model must first be developed. This includes identification of the parties involved; assignment of roles and responsibilities to each of the identified participants; definition of the protocols associated with the architecture; and selection of the authorisation techniques to be deployed within the framework based on state of the art mechanisms.

We depict six scenarios, each of which describes an architecture that supports policy-based authorisation of mobile code and agents. Each architecture is analysed with respect to the level of security it can support and with regard to its suitability for implementation in a mobile environment. This analysis is used to compile a list of fundamental requirements for the underlying architecture of a policy-based framework which enables mobile code and agent authorisation in a mobile environment.

Section 3.2 introduces the participant roles involved in the six scenarios, which are then described in sections 3.3 to 3.8.

- In the first scenario, described in section 3.3, executables are signed, and authorisation of an executable by the mobile host is based solely on the identity of the party signing the code.

- In the second scenario, described in section 3.4, an executable is permitted to execute in one of four execution environments (each of which has a predefined set of executable permissions associated with it) depending on the role of the entity which signed the executable.

- In scenario three, described in section 3.5, each executable is tested by a TTP, and an attribute certificate is generated describing the security relevant properties of the executable as verified via analysis. Authorisation of an executable by the mobile host is based on the security relevant properties of the executable, as described by the TTP.

- In scenario four, described in section 3.6, the security controls associated with an incoming executable are verified by a domain server, with which the destination mobile host is affiliated. An attribute certificate is then generated by the domain server, which describes the security relevant properties of the executable. Authorisation of an executable by the mobile host is based on the security relevant properties of the executable, as specified in the executable's attribute certificate.

- In the fifth scenario, described in section 3.7, the incoming executable is executed on an emulator on a domain server with which the destination mobile host is affiliated. The executable is then signed and forwarded to the destination host by the domain server if no malicious executable behaviour was detected. Authorisation of an executable by the mobile host is based on the domain server signature on the executable.

- In scenario six, described in section 3.8, the state appraisal functions and

signatures appended to an incoming agent are verified by a domain server to which the destination mobile host is affiliated. The executable is only signed and forwarded to the destination host by the domain server if all controls can be validated. Authorisation of an agent by the mobile host is based on the domain server signature on the agent.

These six architectural models were chosen because they incorporate the mechanisms for mobile code and agent authorisation that are most widely discussed in the literature. Based on the analyses of these six models, in section 3.9 we list the minimum requirements for an architectural model supporting a policy-based framework enabling mobile code and agent authorisation in a mobile environment.

3.2 Entities involved

We begin by introducing the participant roles involved in the scenarios described in sections 3.3 to 3.8.

- An *agent code author* is responsible for the production of agent code. More generically, we define a *code author* as the entity responsible for the generation of executable code.
- An *agent creator* is responsible for agent creation, i.e. the combination of the program code from the agent code author with data and initial state information. The agent creator may also be responsible for agent distribution.
- A *mobile device* represents a mobile host on which an executable may execute.

- A *service provider* represents a host machine/server on which code may execute, or an agent may execute and interact with other agents.
- A *certification authority* is responsible for the generation of public key certificates.
- A *device manufacturer* is responsible for the manufacture of mobile devices.
- A *network operator* is responsible for the provision of cellular communications functionality to a platform.
- A *trusted third party* is, in the broadest context, a third party trusted to complete a specified task.
- A *device user* is an end user of a device.

3.3 Scenario 1

The first scenario involves four roles: the CA, the device manufacturer, the (agent) code author, and the mobile device. The mobile device contains the public key of its manufacturer and a public key store containing trusted root CA public keys, so that the signatures on incoming executables can be verified. Each (agent) code author and device manufacturer possesses a public/private key pair and an X.509 certificate for its public key issued by their chosen CA.

When an (agent) code author wishes to distribute executables, he must first apply to a device manufacturer for an attribute certificate which indicates that the specified (agent) code author's executables may be executed on mobile devices manufactured by that particular device manufacturer. An (agent) code author may repeat this process with one or more device manufacturers.

Individual device manufacturers may then respond with a signed attribute certificate consisting of:

- an identifier element representing the requesting (agent) code author;
- an identifier element representing the signing device manufacturer;
- an attribute element, which indicates that the (agent) code author is trusted to generate executables which will be executed on mobile devices which that particular device manufacturer has manufactured; and
- the signature of the device manufacturer generated on the aforementioned elements.

The decision of the device manufacturer to certify an (agent) code author may be based on a variety of factors, including the quality of the code generated by the author, compliance of the author with accepted industry standards for code generation and testing, contracts and/or liability agreements, and/or performance and reputation. If the device manufacturer does not deem the (agent) code author to be trustworthy, no attribute certificate is generated.

The author may then create, sign and distribute an executable, together with its attribute certificate(s). When the executable is received by a mobile device, the signature of the author on the code is verified. The relevant device manufacturer signed attribute certificate is retrieved (if it exists). The signature of the device manufacturer on the appropriate attribute certificate is verified and the presence of the code author's identifier in the attribute certificate is confirmed. Finally, the attribute element, which indicates that the author is trusted to generate executables which will be executed on devices which that particular device manufacturer has manufactured, is checked. If all checks are positive, executable execution is authorised.

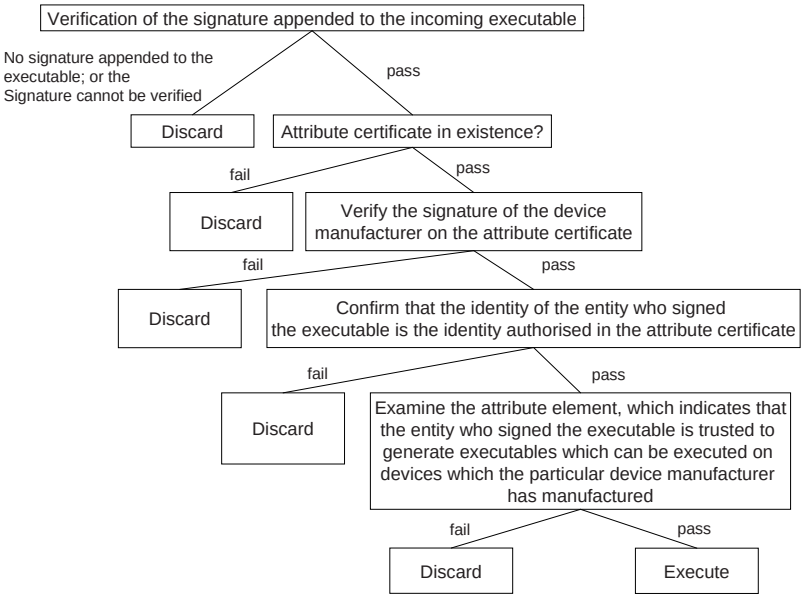

Figure 3.1: Scenario 1

If the 'executable' is traditional mobile code, the above mechanism provides a simple method of protecting mobile devices from incoming malicious code. If, however, the 'executable' is a mobile agent, the protection of agent state information must also be considered.

In order to protect static agent state information, agent creators may also possess a public/private key pair and an X.509 public key certificate issued by their chosen CA. They may then request an attribute certificate from a device manufacturer, which indicates whether or not the specified agent creator's mobile agents may be executed on mobile devices manufactured by that particular device manufacturer.

When agents are created, agent creators can then sign agent code in conjunction with any associated static data and state information. On receipt of a mobile agent, a mobile device will complete the verification process illustrated in figure 3.1 not only for the agent code author, but also for the agent creator.

As the number of attribute certificates associated with either an (agent) code author or, indeed, an agent creator grows, it may become more efficient for attribute certificates to be stored in directories and accessed by a mobile device when required, using a protocol similar to the lightweight directory access protocol (LDAP). In this way, an executable would not have to be transported with all the relevant certificates, which could prove cumbersome and use a significant amount of bandwidth.

This particular identity-based authorisation architecture is rather coarse-grained. An (agent) code author or, indeed, an agent creator is labeled either safe or unsafe, with no allowance made for the possibility that not all executables coming from a particular source are of a similar standard. In reality, it may be the case that certain executables pose a threat to the mobile device on which they are executed because of careless mistakes made by the (agent) code author and/or agent creator. There is also the possibility that an employee of an approved software provider company could circulate and have executed malicious executables, by distributing them signed using a key/keys associated with the company and which has/have been approved by one or more device manufacturers. In conjunction with this, since code authors or agent creators must be approved by the manufacturer of a device on which they require their executables to be authorised to execute, the number of executables that a user may be permitted to download and execute on their platform may be severely restricted.

In this scenario, much responsibility is placed on the device manufacturer — it is not clear, however, whether, or why, the device manufacturer should be made the sole point of trust. Finally, while this architecture provides security mechanisms to protect a mobile device from incoming malicious mobile (agent) code and static agent data and state information, no consideration is given to

the protection of the mobile device from dynamic state information which may cause an incoming agent to behave maliciously.

3.4 Scenario 2

The fundamental concept described in scenario two mirrors that implemented in MExE [140]. Five participant roles are defined: the CA, the device manufacturer, the network operator, the TTP, and the mobile device. The mobile device contains the public keys of its manufacturer, network operator and a selection of TTPs, and a public key store containing trusted root CA public keys so that the signatures on incoming executables can be verified. Each device manufacturer, network operator and TTP possesses a public/private key pair and an X.509 public key certificate issued by their chosen CA. The device manufacturer, network operator, and TTPs each have an associated execution environment on the mobile device. Executables contained within the device manufacturer execution environment are permitted the broadest set of access rights. Executables contained within the network operator domain receive a more restrictive set of access rights, and executables contained within the TTP domain receive fewer rights than executables in the device manufacturer or network operator domains. There is also a fourth, restrictive, execution environment within which executables signed by unknown third parties may be contained. If the mobile device on which an executable wishes to execute is not a mobile phone, it is possible that only three execution domains are defined, i.e. the device manufacturer domain, the TTP domain and the restrictive domain.

When an executable is received by a mobile device, the process illustrated in figure 3.2 is completed. The signature of the (agent) code author on the executable is verified. If the (agent) code author cannot be authenticated, either

because the incoming executable has not been signed or, alternatively, because the signature cannot be verified, the executable is immediately discarded. If the signature is verified, the executable is then permitted to execute in the domain corresponding to the role of the executable's signer. If the incoming executable has been signed by an unknown entity, it will be permitted to execute only in the restrictive domain with minimal privileges.

Alternatively, if the identity of the signer cannot be associated with one of the roles listed above, an unknown entity who has signed the incoming executable may present an attribute certificate signed, for example, by the device manufacturer such that a chain of trust may be constructed between the mobile device and the unknown entity. If this chain of trust can be verified, the incoming executable may be executed in a less restrictive domain.

In this instance, the signed attribute certificate would typically contain:

- an identifier element representing the requesting entity;
- an identifier element representing the trusted signing entity (e.g. the device manufacturer);
- an attribute field indicating to what level the entity is trusted by the trusted signing entity (i.e. the domain in which their executables should be executed); and
- the signature of the trusted signing entity, e.g. the device manufacturer, generated on the aforementioned elements.

As was the case with the architecture described in figure 3.1, this scenario is rather coarse-grained. Authorisation is based solely on the identity of the entity which signed the executable. This particular scenario also leads one to query what intrinsic qualities make the device manufacturer more trustworthy

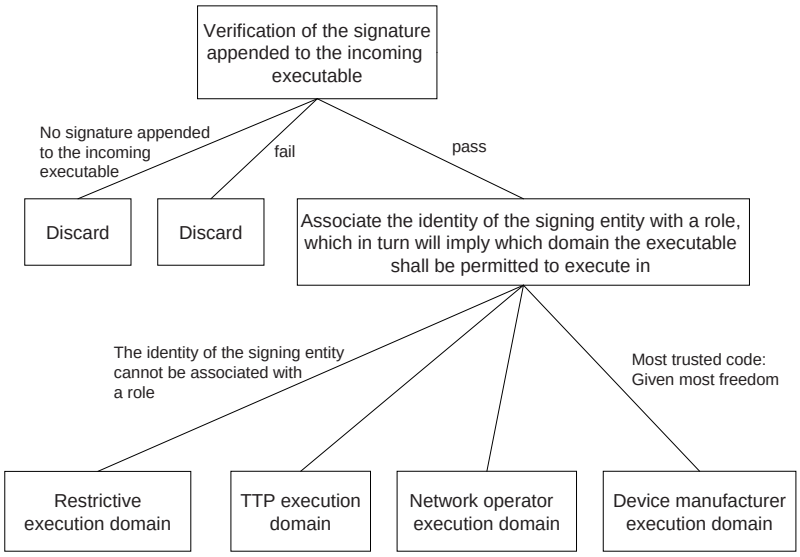

Figure 3.2: Scenario 2

than, say, a network operator? Finally, while this architecture provides security mechanisms to protect a mobile device from incoming malicious mobile code or, indeed, malicious mobile agent code, no consideration is given to the protection of the mobile device from malicious agent state information, either static or dynamic.

3.5 Scenario 3

Participant roles in this scenario include the CA, the TTP, and, as always, the mobile device. In this case, the mobile device contains a selection of TTP public keys and a public key store containing trusted root CA public keys. Each TTP possesses a public/private key pair and an X.509 public key certificate issued by their chosen CA. In this scenario, the (agent) code author submits an executable to a chosen TTP for analysis following its production. The type or level of analysis which the executable undergoes may vary considerably. For the purposes of this chapter, we will examine two possible approaches to executable

analysis.

A TTP may analyse the behaviour of an executable through the formulation and verification of proofs of code, see section 2.5.1.1, or by capturing a behavioural model of the executable and the validating the behaviour described in this model, as described in section 2.5.1.2. For the results of this type of analysis to be meaningful to a mobile host (i.e. useable in making an authorisation decision regarding executable execution) there must be a pre-defined set of security relevant attributes which are used to describe the executable following analysis. Security relevant attributes might define, for example, a pre-set bandwidth limit on network packets transmitted by the executable, or the number of CPU cycles which will be used during execution. Both of these properties may be established using proofs of code [107].

Once the analysis has been completed, the TTP assembles a signed attribute certificate consisting of:

- an executable identifier (for example, a hash of the static executable code, data and, potentially, any static state information);
- an identifier element representing the trusted signing entity, i.e. the TTP;
- the security relevant properties of the executable, as verified via analysis; and
- the signature of the TTP generated on the aforementioned elements.

When the executable is received by a mobile device, the signature of the TTP on the attribute assertion is verified and the executable identifier is validated. Finally, the executable is authorised to execute if the security relevant properties of the executable do not violate the security policy of the mobile device. This architectural model is illustrated in figure 3.3. As described by Sekar [138], the

device security policy may be changed in order to facilitate the needs of the incoming executable.

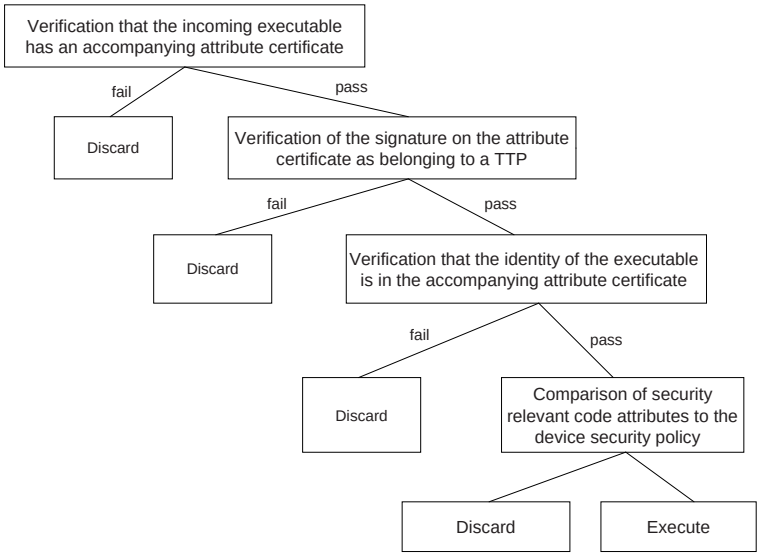

Figure 3.3: Scenario 3

Alternatively, a TTP may analyse the executable using a set of pre-defined tools developed in order to test the security of code. "Static analysis tools try to prevent attacks by finding the security vulnerabilities in the source code" [168]. Tools deployed by a TTP may include the likes of BOON [165], which aims to detect buffer overflow vulnerabilities; MOPS [23], which checks ordering constraints; Mjolnir [167], which makes use of dependence graphs and constraint solving to find buffer overflows in C code; or IPSSA [100], a tool which Livshits and Lam have developed for finding buffer overflow and format string bugs.

The test set completed on the executable may be dependent on the request made by the (agent) code author. If the (agent) code fails any of the tests completed by the TTP, it will notify the (agent) code author rather than generate an attribute certificate for the executable.

Once the executable has achieved success in the requested test set, the TTP

assembles a signed attribute certificate consisting of:

- an executable identifier;
- an identifier element representing the trusted signing entity, i.e. the TTP;
- the tests successfully completed on the executable; and
- the signature of the TTP over the aforementioned elements.

When the executable is received by a mobile device, the signature of the TTP on the attribute assertion is verified and the executable identifier is validated. Finally, the executable is authorised to execute in one of a range of execution domains, as shown in figure 3.4, depending on the nature of the tests completed on the executable. If the signature on the attribute assertion is that of an unknown third party, the executable is discarded.

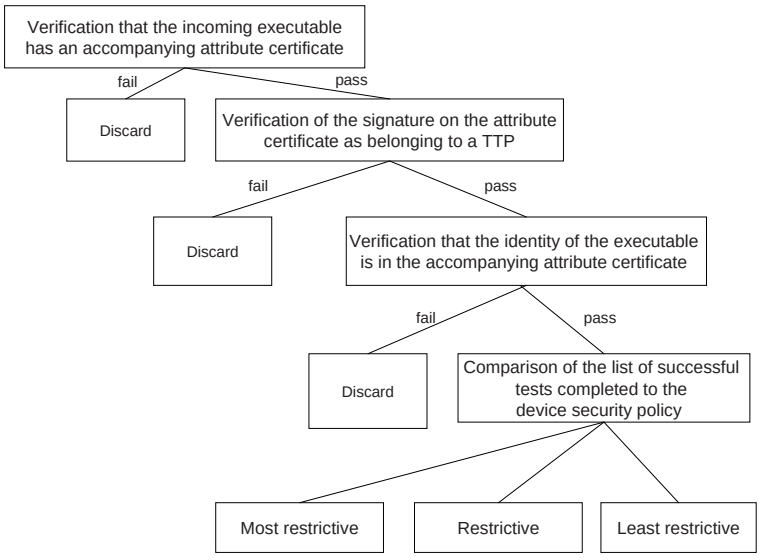

Figure 3.4: Scenario 3

In this scenario, a TTP is trusted to attest to the results of the analysis completed on a particular executable, as opposed to the previous scenarios where all executables signed by a particular entity are approved for execution. If a

malicious executable is detected, the TTP which signed its associated attribute certificate may be held accountable, if the analysis of the executable was not accurately or correctly completed as specified in the attribute certificate.

There are, however, a number of obstacles associated with the executable-centric authorisation approaches described above. Using the first style of analysis it might prove difficult to compile a comprehensive list of the security relevant attributes under which an executable is described. In conjunction with this, currently, not all security relevant attributes can be verified using methods such as proofs of code [138].

In the alternative approach, it may be difficult to decide on the permissions that should be allotted to an incoming executable based on the tests successfully passed by the executable. In conjunction with this, many of the test tools are only applicable to executables written in a particular language; for example all the tools listed above can only be used to analyse C code. There may also be a problem with some tests in relation to false positives or, indeed, false negatives, which mainly arise from the use of poor vulnerability databases [168].

As was the case with the previous scenario, while this architecture uses security mechanisms to protect a mobile device from incoming malicious mobile code or, indeed, malicious mobile agent code, no consideration is given to the protection of the mobile device from malicious agent state information.

By integrating access control lists (ACLs) into mobile devices, the scenario described above can be made either more secure or, alternatively, more efficient. The integrated ACLs will contain the identifiers for trusted (agent) code authors and, in the case of mobile agents, trusted agent creators. In this instance, each (agent) code author and agent creator possesses a public/private key pair and an X.509 public key certificate issued by their chosen CA.

One option allows executables signed by trusted code authors and, potentially, trusted agent creators to bypass the security checks imposed on unsigned executables. If a received executable has been signed by an entity/entities whose identifier(s) are contained in the device ACL(s), it may immediately be authorised to execute. If not, then the executable will undergo the checks outlined above. This modification for the sake of efficiency could, however, jeopardise the security of the device.

Alternatively, ACLs could be used to increase the security of the system. In this case, an incoming executable must be accompanied by an attribute statement, signed by a TTP, which not only describes the type and results of the analysis completed on the executable, but also the identity of the (agent) code author and, potentially, the identity of the agent creator. The identity of the (agent) code author, and, potentially, the identity of the agent creator, must be verified as present in the mobile device ACLs, before any further checks are completed on the incoming executable. This set-up, however, may be too restrictive for the free flow of executables and, more specifically, mobile agents. It may also necessitate the storage of large ACLs in resource-restricted mobile devices.

3.6 Scenario 4

In this scenario, we assume that each mobile device belongs to a domain, and that each domain has an associated domain server. Before any executable reaches its destination host, the domain server responsible for that device will process any security controls associated with the incoming executable on behalf of the destination mobile device. Participant roles include the CA, the (agent) code author, the agent creator and the domain server. The domain server con-

tains a public key store containing trusted root CA public keys, so that the signatures on incoming executables can be verified. The mobile device contains the public key of its associated domain server. Each (agent) code author, agent creator and domain server possesses a public/private key pair and an X.509 public key certificate issued by their chosen CA. After (agent) code generation and agent creation, the (agent) code author and agent creator formulate the required security controls for distribution with the executable, e.g. digital signatures, proofs of code and/or state appraisal functions. When a domain server receives an incoming executable destined for a mobile device within its domain, any signatures appended to the executable are verified, and any proofs, models and/or state appraisal functions that accompany the executable are processed.

An attribute certificate is then generated and signed by the domain server, containing the executable's identity and the security relevant attributes of the executable. This is sent to the destination mobile device. If the 'executable' is an agent, then the agent's identity may be represented by the hash of the agent code, data and both static and dynamic state information. The mobile device then verifies the signature of the domain server on the attribute certificate, and, depending on the security relevant attribute values and the mobile device policy statements, either authorises execution or not. This mobile device policy model is illustrated in figure 3.5.

The deployment of a domain server in the validation of security relevant attributes means that computationally intensive checks can be completed on the executable without overburdening the mobile device CPU. In this particular model, if the incoming executable is a mobile agent, checks can not only be completed to ensure that the agent code is trustworthy, but also to ensure that the agent's state information is not malicious. For every mobile device on which the agent wishes to execute, its corresponding domain server will validate any

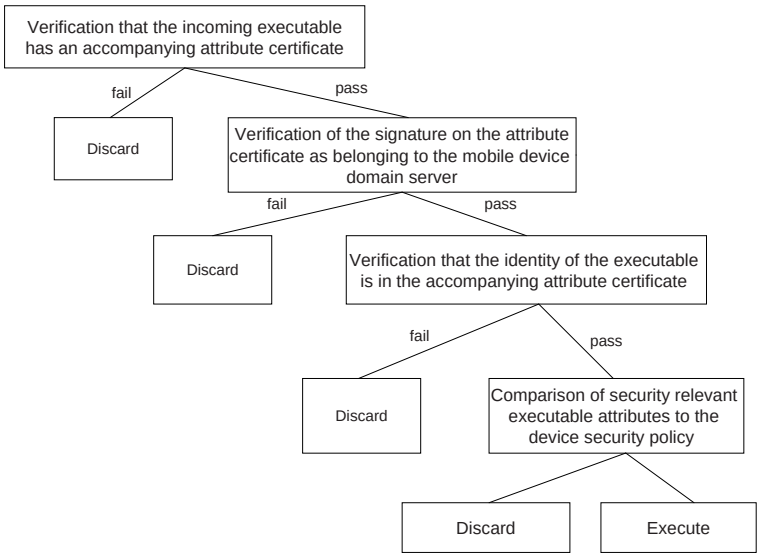

Figure 3.5: Scenario 4

agent controls before forwarding the executable and, potentially, its associated attribute certificate, directly to the device. If the validated controls include state appraisal functions, the domain server can verify that the state of the agent has not become malicious, and then forward the agent and its attribute certificate to the device. The agent identity defined within the attribute certificate ensures that the verified agent state cannot be modified before it reaches its destination mobile device.

Efficiency and throughput/bottleneck issues may arise, however, as for every destination host an agent wishes to visit, a domain server may have to process it and its accompanying security controls. To reduce the load, trust relationships could be formed between domain servers, allowing a domain server to request previously constructed agent attribute statements from another domain server that it trusts.

This model is more flexible than the model described in scenario three above, as it leaves the (agent) code author and/or agent creator to decide what, if any,

controls should be transmitted with their executables. This, however, makes the authorisation decision-making process completed by the mobile device more complex, as a large number of policy statements may have to be defined in order to cater for all the possible security relevant attributes which may be used to describe the incoming executable. There may also be problems relating to the consistency of security controls associated with incoming executables as different (agent) code authors and agent creators could associate different controls with their executables.

3.7 Scenario 5

A domain server is also used in this scenario. It provides two fundamental components: a set of execution environments simulating device types which exist within the domain, and a database. As described in section 3.6, before any executable reaches its destination host, the domain server responsible for that device will process the incoming executable on behalf of the destination mobile device, as described below. Participant roles in this scenario include the CA, the domain server, the (agent) code author, the agent creator, and, as always, the mobile device, which contains the public key of its associated domain server. Each (agent) code author, agent creator and domain server possesses a public/private key pair and an X.509 public key certificate issued by their chosen CA.

When an (agent) code author generates executable code it signs the (agent) code and circulates it. If the code is for an agent, an agent creator then combines the code with the necessary state information. All static agent code, data and state information may then be digitally signed by the agent creator. The executable can then be disseminated across the network. When a domain

server receives an incoming executable destined for a mobile device within its domain, the signature(s) appended to the incoming executable are verified, and the executable is then executed in a simulator mirroring the intended destination device. If malicious behaviour is attempted by the executable in the simulator, it is discarded and a record of the malicious behaviour is made in the profile of the (agent) code author or, indeed, the agent creator (if the malicious agent behaviour was caused by malicious state information).

Conversely, if the executable behaves as expected, a note is made of this in the profiles of both the (agent) code author and, if relevant, the profile of the agent creator. This method, rather than confining the examination of executables to specific tests, allows for the discovery of any security violations that the executable may attempt. It does, however, mean that every executable may potentially be executed twice for each of its destination hosts, which may lead to efficiency problems. The profile database may, however, allow domain servers to improve these efficiency issues. Code written by authors, or agents written by authors and creators, who have built up a positive profile with a domain server, could be immediately authorised to execute before simulated execution has been completed. The activity of each domain server activity is illustrated in figure 3.6.

Once the domain server has decided that the executable can be trusted by the mobile device for which it is responsible, it signs the executable, and forwards it to the mobile device for execution. The mobile device would then complete the checks described in figure 3.7, and discard or authorise the executable accordingly.

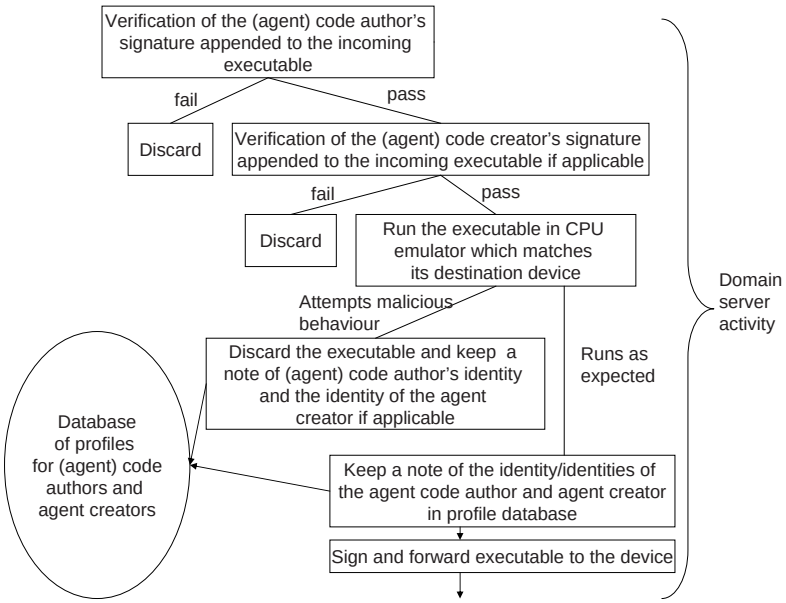

Figure 3.6: Scenario 5 — Domain server activity

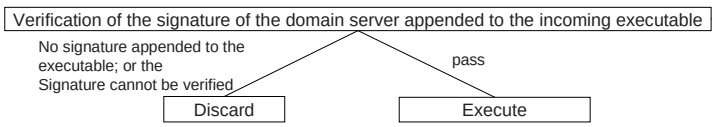

Figure 3.7: Scenario 5 — Mobile device activity

3.8 Scenario 6

The final scenario mirrors the authorisation model described by Berkovits et al. [11], and is illustrated in figure 3.8. This architectural model may be used only for the authorisation of incoming agents, as opposed to generic mobile code. Participant roles in this scenario include the CA, the agent code author, the agent creator, and, as always, the mobile device. In this case, the mobile device contains the public keys of a selection of (agent) code authors and a public key store containing trusted root CA public keys, so that the signatures on incoming executables can be verified. Each agent code author and agent creator possesses a public/private key pair and an X.509 public key certificate issued by their chosen CA.

When agent code is generated, the author signs the program name. The program name is defined as a hash of: the program, a state appraisal function *max* which, as a function of the agent's current state, outputs the maximum set of permissions to be accorded to the agent running the program, and a sender permission list, which includes every destination entitled to receive and use the agent code. On receipt of the agent code, and following agent creation, the agent creator, which has been listed in the SPL, then signs the agent name. The agent name is defined as the hash of: an identifier for the agent creator, the program, the program name, as defined above, a state appraisal function *req*, used to calculate the permissions the sender wants the agent to run with, and a place permission list. The PPL lists every destination host the agent is entitled to visit.

When an incoming agent is received by a mobile device, the signature of the agent code author on the program name is verified. The presence of the agent code author's identity in the list held by the mobile device is verified.

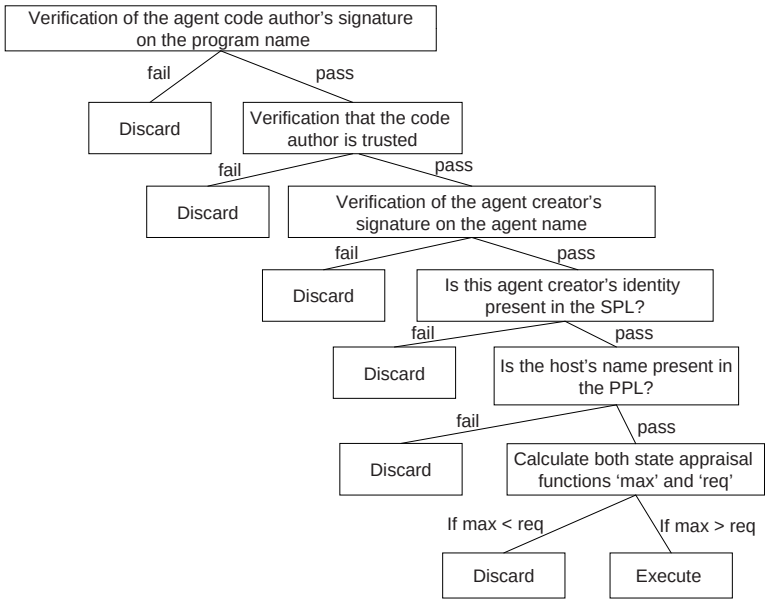

Figure 3.8: Scenario 6 — Domain server activity

The signature of the agent creator on the agent name is verified. The presence of the agent creator's identity in the SPL of the agent code author is verified, and the presence of the mobile device's identity in the PPL of the agent creator is checked. Both the state appraisal functions are then calculated. If the output of *max* is greater than or equal to the output of *req*, then access is granted; otherwise the agent is discarded.

In this particular scenario, the agent code author is essentially responsible for defining the authorisation policy for the agent, i.e. defining the maximum set of privileges which can be allotted to the executing agent. The agent code author must be trusted by the mobile device to complete such a task. Use of this framework requires a considerable amount of processing to be completed on the mobile device prior to agent authorisation. This scenario also requires that the SPL and PPL be defined in advance, which is probably not the most practical solution for a distributed mobile environment.

As an alternative, in order to reduce the processing to be completed on the

mobile device, we could assume that each mobile device belongs to a domain and that each domain has an associated domain server, as was the case in scenarios 4 and 5. Before any executable reaches its destination host, the domain server responsible for that device will process any signatures, SPLs, PPLs and state appraisal functions associated with the incoming executable on behalf of the destination mobile device.

In order to facilitate this modification to the architectural model, a domain server, which contains the public keys of a selection of (agent) code authors and a public key store containing trusted root CA public keys, is required. The mobile device must contain the public key of its associated domain server. When an incoming agent is received by the appropriate domain server, the signature of the agent code author on the program name is verified. The presence of the agent code author's identity in the list held by the domain server is verified. The signature of the agent creator on the agent name is verified. The presence of the agent creator's identity in the SPL of the agent code author is verified, and the presence of the mobile device's identity in the PPL of the agent creator is checked. If any of these checks fail, the agent is discarded. Both the state appraisal functions are then calculated. If the output of *max* is greater than or equal to the output of *req*, the mobile agent is signed by the domain server and forwarded to the mobile device for execution; otherwise the agent is discarded. The mobile device then completes the checks described in figure 3.7 and discards or authorises the executable accordingly. For the purposes of this thesis we will focus on this latter modification to scenario six, where the computational burden is placed on the domain server rather than the mobile device.

3.9 Conclusions

In this chapter, six architectural models facilitating the authorisation of incoming mobile code and agents have been described. Each model has been analysed with respect to its security and with regard to its suitability for implementation in a mobile environment. A summary of this analysis is tabulated in section 13.1.1.

Using the scenarios described in this chapter we can compile a set of requirements for the underlying architecture of a policy-based code and agent authorisation framework for implementation within a mobile environment. As a basic requirement, the underlying architecture should take into account the limited nature of many of the end host devices, i.e. the use of the end host CPU and storage should minimised. The underlying architecture should also ideally support a mechanism which provides assurances regarding the origin(s) of the executable, code quality, and the state of the agent. The use of a TTP, which verifies proofs of code and executes state appraisal functions, for example, or which completes a series of tests on an executable before it is sent to a mobile host for execution, would mean that the end host is not burdened with an intensive checking process.

It is required that each mobile device supports a policy engine. We define a policy engine as the software that executes on a mobile host which inputs an authorisation request from an authorisation requestor (AR) and outputs an authorisation decision. In the context of this work, an authorisation request is essentially a request from an incoming executable to execute on a specific mobile device. The authorisation decision dictates whether or not an executable may execute and/or the constraints under which executable execution is authorised.

We assume that the policy engine contains the following six fundamental

components.

- A policy administration point (PAP) which is used to administer the policy statements of the mobile device. A policy repository is also associated with this component.

- A policy information point (PIP) which is responsible for collating information pertaining to the authorisation requestor, for example, digital signatures, public key certificates and/or attribute certificates.

- An authentication point (AP) which authenticates the authorisation requestor, i.e. the incoming executable, and/or the (agent) code author, agent creator, TTP or domain server which 'speaks for' an incoming executable, where the 'speaks for' relationship implies that when P_1 speaks for P_2, it follows that when P_1 says s, P_2 says s [11].

- A trust establishment module (TEM) which maps an unknown authorisation requestor to a system specific role/group based on AR attributes.

- A policy decision point (PDP), where policy decisions are made. A PDP uses policy statements, written, managed and stored using the policy administration point, attributes of the access requestor, the target resource, and the execution environment, in order to make the appropriate decision.

- A policy enforcement point (PEP), which is responsible for enforcing the policy decision output by the PDP.

In order for the policy engine to function it is required that a number of additional elements can be defined within the system and processed by the policy engine.

- Mobile device policy statements, which specify the rules that define a

choice in the behaviour of a system [36]. Policy statements are defined, stored and accessed using the PAP, described above.

- Attribute certificates, which allow security-related attributes and data to be associated with an access requestor.

- Authentication evidence, which is used to verify the identity of an authorisation requestor, i.e. the incoming executable and/or the entity who 'speaks for' an incoming executable and facilitates the association of the authorisation requestor with a principal. A principal is a name, for example, a conventional name or a public key, associated with the authorisation requestor [96].

- An ordered compliance value set, which specifies the set of decisions that the PDP may output in response to an authorisation request. The PEP must react accordingly to the compliance value output in order to facilitate secure executable execution.

The six scenarios described in this chapter are re-used in chapter 4 to help analyse selected components which could be utilised in the specification of the framework policy engine. The policy engine itself is addressed in chapter 5.

Chapter 4

A policy engine for mobile code and agent authorisation

Contents

4.1 Introduction 112
4.2 The policy engine 113
4.2.1 Policy statements 113
4.2.2 Attribute certificates 114
4.2.3 Authentication evidence 115
4.2.4 Compliance values 117
4.2.5 The PAP 117
4.2.6 The PIP 117
4.2.7 The AP 118
4.2.8 The TEM 119
4.2.9 The PDP 119
4.2.10 The PEP 119
4.3 Approaches to policy specification 119
4.4 KeyNote 124
4.4.1 Scenario 1 128
4.4.2 Scenario 2 134
4.4.3 Scenario 3 139
4.4.4 Scenario 4 147
4.4.5 Scenarios 5 and 6 149
4.4.6 Conclusions 151
4.5 Ponder 159
4.5.1 Prior art 163
4.5.2 Scenario 1 169
4.5.3 Scenarios 2 – 6 175
4.5.4 Conclusions 176

4.6 SAML 177
4.6.1 Scenario 1 184
4.6.2 Scenario 2 189
4.6.3 Scenarios 3 and 4 191
4.6.4 Scenarios 5 and 6 195
4.6.5 Conclusions 196
4.7 Conclusions 196

In this chapter we examine three selected policy statement and attribute certificate specification languages, namely KeyNote, Ponder and SAML, and explore the functionality of their supporting policy engine components. The main goal of this analysis is to discover whether these languages can express the policy statements and attribute certificates required by the six scenarios described in chapter 3, and also whether the necessary policy engine component functionality can be supported. Our conclusions are then used in chapter 5 to choose the most appropriate language(s) for policy statement and attribute certificate expression in our policy-based framework.

4.1 Introduction

Six architectural models, each aimed at facilitating mobile code and/or mobile agent authorisation, were described in chapter 3. Each architectural model was analysed with respect to the level of security it can support and with regard to its suitability for implementation in a mobile environment. From this analysis, we derived a list of features desirable in the underlying architectural model of a policy-based framework for mobile code and agent authorisation. The constraints which must be considered when developing this architectural model, arising from potential limitations of the devices upon which the executables will execute, were also highlighted.

In this chapter we re-examine these six architectural models, in order to compile the expression requirements that the language chosen for policy statement and attribute certificate specification must meet, and the functional requirements that the supporting policy engine must fulfil in order to facilitate the implementation of each architectural model. Using these requirements, an analysis of KeyNote, Ponder and SAML is performed to enable the most appropriate language(s) to be chosen for policy statement and attribute certificate expression in our policy-based authorisation framework.

In section 4.2 the scenarios described in chapter 3 are re-examined in order to compile the functional requirements for the language(s) chosen for policy statement specification, attribute certificate definition, compliance value expression and the supporting policy engine, in order to enable the implementation of each architectural model. In section 4.3 we examine a selection of policy and attribute certificate expression techniques, and select three, namely KeyNote, Ponder and SAML, for detailed analysis. In sections 4.4, 4.5 and 4.6 we consider the three selected languages in turn, in order to determine whether they meet

the requirements compiled in section 4.2 and, ultimately, whether they could be used to implement the scenarios described in chapter 3.

4.2 The policy engine

Here we examine what is required of the language(s) chosen for policy statement specification, attribute certificate expression, authentication evidence expression and ordered compliance value definition. We will also list the functionality required of the PAP, PIP, AP, TEM, PDP and PEP, as described in section 3.9.

4.2.1 Policy statements

In a policy-based framework for mobile code and/or mobile agent authorisation, receiving hosts will need to store policy statements to be used in deciding whether or not to authorise executable execution and/or to dictate the constraints under which executable execution is permitted. This is a prerequisite of all six scenarios described in chapter 3. The policy expression language chosen to express the mobile device policy statements must meet the following requirements.

1. In scenario 1, and potentially in scenario 2, it is required that authority for mobile executable execution authorisation can be delegated.

2. In scenario 1 it is required that an incoming executable, which has been signed by an (agent) code author, which in turn has been certified by the mobile device's manufacturer, can be permitted to execute on the mobile device (potentially under specified controls).

3. In scenario 1 it may be required that an incoming agent, which has been signed by both an agent code author and an agent creator, which in turn

have been certified by the mobile device's manufacturer, can be permitted to execute on the mobile device (potentially under specified controls).

4. In scenarios 2, 5 and 6, and potentially in scenario 3, it is required that an incoming executable, which has been signed by a particular entity, for example, a device manufacturer, network operator, TTP, or domain server, can be provisioned with the authority to execute under specified controls.

5. In scenarios 3 and 4 it is required that particular entities, for example, a domain server or a TTP, can be provisioned with the authority to create attribute certificates for executables.

6. In scenarios 3 and 4 it is required that an incoming executable, whose attributes have been certified by a trusted entity, as described in requirement 5, can be provisioned with the authority to execute (potentially under specified controls) if the executable's attributes meet specified conditions.

4.2.2 Attribute certificates

Incoming executables, or, indeed, (agent) code authors or agent creators, may have associated attribute certificates, see scenarios 1, 2, 3 and 4. The language chosen to express attribute certificates must fulfil the following requirements.

1. In scenarios 1, 3, 4, and potentially in scenario 2, signed attribute certificates are required.

2. In scenario 1, and potentially in scenarios 2 and 3, it is required that an entity's identifier can be expressed in a chosen form, for example as a distinguished name or as a public key, in an attribute certificate. This enables the attribute certificate to be associated with one particular entity.

3. In scenario 1 it is required that the device manufacturer can delegate to chosen (agent) code authors and creators, by means of an attribute certificate, the authority to generate and sign executables, which will be authorised to execute on a device for which that particular device manufacturer is responsible.

4. In scenario 2 it may be required that the device manufacturer can delegate to chosen TTPs, by means of an attribute certificate, the authority to generate and sign executables, which will be authorised to execute in the TTP domain on the device for which that particular device manufacturer is responsible.

5. In scenario 3, and potentially in scenario 4, it is required that the identity of an incoming executable can be expressed within an attribute certificate (as the hash of the incoming mobile (agent) code or mobile agent, for example). This enables the attribute certificate to be uniquely bound to one particular executable.

6. In scenarios 3 and 4 it is required that the attributes of an incoming executable can be expressed within its associated attribute certificate. The definition of attribute values may require the expression of bit strings, character data, numerical or boolean values. The expression of an unbounded number of attribute element types must also be supported.

4.2.3 Authentication evidence

Authentication is defined by Berkovits et al. [11] as the "process of deducing which principal has made a specific request". Receiving hosts may, for example, be required to authenticate an incoming executable and the entity who 'speaks for' an incoming executable. This definition of authentication corresponds to

the notion of origin authentication (i.e. a connectionless service) rather than entity authentication (i.e. a connection-oriented service).

1. In scenarios 1, 2, 5 and 6, and potentially in scenario 4, it is required that the (agent) code author (and, potentially, the agent creator) or the domain server can be authenticated through the verification of the digital signature(s) generated on and appended to the incoming executable. In this instance, the digital signature serves as the authentication evidence.

2. In scenario 1, and potentially in scenario 2, it is required that the device manufacturer, which has generated and signed the attribute certificate of an entity to whom authority is being delegated, can be authenticated through the verification of the digital signature generated on the attribute certificate. In this instance computed on the attribute certificate serves as the authentication evidence.

3. In scenario 3, and potentially in scenario 4, it is required that the TTP or domain server which has generated and signed the attribute certificate of the incoming executable can be authenticated through the verification of the digital signature generated on the incoming executable's attribute certificate. In this instance, the digital signature serves as the authentication evidence.

4. In scenarios 1 to 6 it is required that the incoming executable is authenticated through the verification of the hash of the incoming (agent) code and data, or the hash of the entire agent (including any agent code, data, and static and dynamic state information), against the hash signed by the entity/entities which 'speak(s) for' the executable. In this instance, the incoming executable serves as the authentication evidence.

4.2.4 Compliance values

In each of the scenarios described in chapter 3 an ordered compliance value set must be defined. This enables the PDP to output an authorisation decision to the PEP, when given an authorisation request.

1. In scenarios 1, 3, 4, 5 and 6, a simple boolean ordered compliance value set is required containing two values, namely: discard and execute.

2. In scenario 2, and potentially in scenario 3, a more complex ordered compliance value set is required. In scenario 2 the ordered compliance value set must contain five values, namely: discard, restrictive domain, TTP domain, network operator domain, and device manufacturer domain. An ordered compliance value set which contains the values: discard, most restrictive domain, restrictive domain, and least restrictive domain, may potentially be required in scenario 3.

4.2.5 The PAP

In order to support the scenarios described in chapter 3, the PAP implemented on the mobile device must fulfil the following requirement.

1. In scenarios 1 to 6 the PAP must provide a means for specifying, managing and organising mobile device security policy statements.

4.2.6 The PIP

In order to support the scenarios described in chapter 3, the PIP implemented on the mobile device must fulfil the following requirement.

1. In scenarios 1 to 6 all security data relevant to the access requestor must

be collected by the PIP, for example, digital signatures and attribute certificates.

4.2.7 The AP

The authentication point is required to process the AR authentication evidence submitted by the AR, or collated by the PIP.

1. In scenarios 1, 2, 5 and 6, and potentially in scenario 4, it is required that the (agent) code author (and, potentially, the agent creator) or a domain server can be authenticated through the verification of the digital signature(s) generated on and appended to the incoming executable.

2. In scenario 1, and potentially in scenario 2, it is required that the device manufacturer can be authenticated through verification of the digital signatures on the attribute certificates of entities to whom authority has been delegated.

3. In scenarios 3 and 4 it is required that the TTP or domain server, which generated and signed the attribute certificate of the incoming executable, can be authenticated through the verification of the digital signature on the incoming executable's attribute certificate.

4. In scenarios 1 to 6 it is required that the incoming executable can be authenticated through the verification of the hash of the incoming (agent) code and data, or the hash of the entire agent (including agent code, data and any static and dynamic state information), against the hash signed by the entity/entities which 'speak(s) for' the executable, see section 3.9.

4.2.8 The TEM

In order to support the scenarios described in chapter 3, the TEM implemented on the mobile device must fulfil the following requirement.

1. In scenarios 1 to 6 the TEM must map the unknown authorisation requestor to a principal to which policies apply.

4.2.9 The PDP

In order to support the scenarios described in chapter 3, the PDP implemented on the mobile device must fulfil the following requirement.

1. In scenarios 1 to 6 the PDP must compare all the information submitted as part of the authorisation request with the relevant policy statements on the mobile device, and respond with an authorisation decision from the ordered compliance value set.

4.2.10 The PEP

In order to support the scenarios described in chapter 3, the PEP implemented on the mobile device must fulfil the following requirement.

1. In scenarios 1 to 6 the PEP must enforce the decisions of the PDP.

4.3 Approaches to policy specification

A number of different types of security policy expression languages have been devised including logic-based languages, for example, Authorisation Specification Language (ASL) [90] and Standard Deontic Logic (SDL) [21, 164], which may be used in the specification of security policies [32, 63]; role-based access

control specification languages such as Temporal Role Based Access Control (TRBAC) [12] and Ponder [34–37]; and the event-driven policy language, Security Policy Language (SPL) [129], to name but a few. Different policy languages facilitate the expression of different policy types including, for example, authorisation policies, delegation policies, refrain policies, obligation policies and constraint-based policies.

- *Authorisation policies* define the activities principals are permitted to do [35].
- *Delegation policies* define activities that principals are permitted to delegate to other principals within the system [35].
- *Refrain policies* specify what a principal must refrain from doing [35].
- *Obligation policies* define activities principals are obliged to do. Obligation policies are often triggered by events, for example, a security breach within the system or a time-based trigger.
- *Constraint-based policies* limit the applicability of authorisation, obligation, delegation and refrain polices, based on factors such as subject or target state, action/event parameters, i.e. past events (history-based policy statements), or time constraints.

The selection of policy types supported by a particular language is often dependent on the application environment or the use cases upon which the developers focused when specifying the language.

In all of the 'traditional' policy specification languages listed above, however, policy statements are defined in terms of the identity of the AR or, indeed, the group or role to which the identity of the AR can be associated. If the AR can be successfully authenticated to the system, it is mapped to a principal, which

may, for example, be a UserID defined within the system. Policy statements then map principals to permissions/capabilities. Alternatively, if an AR can be successfully authenticated, it may be mapped to a UserID defined within the system, which may in turn be associated with one or more roles or groups, where role assignment or group membership is defined in terms of UserID. Policy statements then map principals (in this instance a role or group name) to permissions/capabilities.

In both cases, policy statements are essentially defined in terms of the AR identity. In Internet enabled applications, however, the AR may be an unknown entity. If the AR is unknown, policies cannot be defined in terms of the AR identity. Trust management refers to "the problem of deciding whether requested actions, supported by credentials, conform to policies" [15]. This implies that an unknown entity may be authorised to perform actions, obligated to perform actions or permitted to delegate capabilities/permissions to other entities based on their associated credential set. Trust management systems include policy maker [15], KeyNote [13, 14, 16], Nereus [103] and REFEREE [26]. These trust management systems enable the specification of both policy statements and attribute certificates.

More recently, we have seen the advent of languages such as (Definitive) Trust Policy Language ((D)TPL) [68], which, rather than trying to define a complete trust management system, is designed to bridge the gap between traditional role-based policy specification languages and unknown entities. It does this by facilitating the definition of trust establishment policies which allow unknown entities to be mapped to roles based on their associated credential set. A traditional role based policy specification language can then be utilised in order to assign capabilities/permissions to roles.

It becomes clear, therefore, that in order to deploy a trust establishment system such as (D)TPL, not only is a traditional role-based specification language required for policy expression, but credential expression must also be considered. X.509 public key certificates represent the best known form of credential, see section 1.5.7. Other credential types exist, including X.509 attribute certificates, see section 1.5.9, and SAML assertions [112, 113].

Following a high-level analysis of a number of policy specification and attribute certificate expression languages, namely ASL [90], SPL [129], Ponder [37], KeyNote [13], Nereus [103], TPL [68], the Portfolio and Service Protection Language (PSPL) [17], credential acceptance policies [137], the X.509 attribute certificate [81] and SAML, [112, 113], three languages, namely KeyNote (and Nereus), Ponder (and (D)TPL) and SAML were chosen for further analysis.

ASL [90], SPL [129] and Ponder [37] represent examples of 'traditional' policy specification languages, as defined above. In ASL, SPL, and Ponder, policies are defined in terms of entity identity or, similarly, group or role membership. Of this language type, we have chosen to examine Ponder in further detail. Of the three languages examined at a high-level, it is the easiest to understand. It permits the expression of the widest range of policy types. It has also been previously deployed in the mobile agent domain [10, 30, 94, 105].

Trust management languages examined at a high-level include KeyNote [13], Nereus [103], PSPL [17] and credential acceptance policies [137]. We also examined the trusted establishment language (D)TPL [68]. Of these trust management and trust establishment languages KeyNote, Nereus and (D)TPL are examined in further detail in this chapter and the next. KeyNote is chosen as it is a well established and well documented trust management framework. Nereus is examined as it extends the functionality provided by KeyNote. Finally, (D)TPL

is chosen as it provides a bridge between traditional role-based policy specification languages and unknown entities, rather than trying to specify a complete trust management framework. While Bonatti and Samarati [17] present a framework enabling the expression of both credentials and policy statements, which define the credentials which must be provided by an AR on request of a service, at the time of writing, this framework and its accompanying language specification PSPL are still in development. Seamons and Winsborough [137] present the notion of credential acceptance policies, which are used to define exactly which credentials must be present in order to obtain specific services. They also give a programming methodology for writing credential acceptance policies. We have chosen not to examine this language because of its use of logic programs (more specifically XSB Prolog) to express policy statements, which are often deemed difficult to use and not always directly transferable into efficient implementation [34].

Following a high-level examination of X.509 attribute certificates [81] and SAML [112, 113], SAML was chosen due to the expressive nature of XML, and the extensibility of the language. XML is also easy to understand. The language also enables the expression of various types of assertion which are closely aligned with the attribute credential types required in the scenarios described in chapter 3.

In the remainder of this chapter we analyse KeyNote, Ponder and SAML, in order to determine whether they can meet the requirements outlined in section 4.2.

4.4 KeyNote

KeyNote is described as "a unified approach to specifying and interpreting security policies, credentials, and relationships and it allows direct authorisation of security critical actions" [14]. It was designed by Blaze, Feigenbaum and Ioannidis in conjunction with Keromytis, and is a descendent of the PolicyMaker system [15].

For the purposes of KeyNote, a trust management system is defined as being comprised of [13]:

- A language for defining actions which have consequences that impact on security;
- A mechanism for identifying principals, i.e. entities that can be authorised to perform actions;
- A language for specifying application policies which determine the actions that principals are allowed to perform;
- A language for specifying credentials which allow principals to delegate authorisation to other principals;
- A compliance checker which determines how a requested action should be handled depending on the policy and credential set available.

A principal, as defined in the KeyNote trust management system, denotes an entity within the system, for example, a public key or a process, with which the authority to perform trusted actions can be associated [13]. Principals perform two functions; they may issue assertions and they may also request actions.

Two types of assertion are defined within the KeyNote trust management system, policy assertions and signed assertions/credentials.

- Policy assertions are issued by the 'POLICY' principal, which represents the root of trust in KeyNote. 'POLICY' is authorised to perform any action within the system. Policy assertions are used to delegate authority to untrusted entities. If an untrusted entity in a policy assertion is identified by a public key, it can then create signed assertions.

- Signed assertions permit authorised entities (either the 'POLICY' entity, or entities authorised in policy assertions or signed assertions) to delegate their authority to other untrusted entities.

An assertion is comprised of a series of fields [13], one of which is mandatory, namely the <Authoriser> field, and six of which are optional, i.e. the <KeyNote-Version>, <Comment>, <Conditions>, <Licensees>, <Local-Constants> and <Signature> fields.

- The <Authoriser> field identifies the principal issuing the assertion.

- The <KeyNote-Version> field contains the version of the KeyNote trust management system in use.

- The <Comment> field may be used to enter comments describing the assertions.

- The <Conditions> field identifies the conditions under which the authoriser trusts the licensee(s) to perform an action.

- The <Licensees> field identifies the principal(s) which are delegated authorisation in the assertion.

- The <Local-Constants> field allows short names to be associated with cryptographic principal identifiers, e.g. public keys. The short names can then be used in the <Authoriser> and <Licensees> fields, making these fields easier to read.

- The <Signature> field identifies a signed assertion and holds the digital signature of the authoriser. The digital signature is computed over the assertion text from the first field to the signature field identifier.

All fields are case sensitive, and the KeyNote version field will always appear first and the signature field last.

When a principal requests an action, an action attribute set describing the action is generated. This action attribute set is then submitted to the KeyNote compliance checker as part of a KeyNote query. A KeyNote query consists of four components:

- the identifier(s) of the principal(s) requesting the action;
- the action attribute set which describes the action;
- an ordered policy compliance value set, which describes the compliance values of interest to the PEP, in ascending order; and
- the applicable policy and signed assertions.

The KeyNote compliance checker returns the correct compliance value.

We now examine the KeyNote architecture, as illustrated in figure 4.1. The objective is to determine which of the functional components, as described in section 4.2, are provided by the KeyNote trust management system, and whether KeyNote enables expression of policy statements, attribute certificates, authentication evidence and ordered compliance value sets.

- KeyNote policy assertions, can be used to specify security policy statements, as described in section 4.2.
- KeyNote signed assertions/credentials, can be used to specify attribute

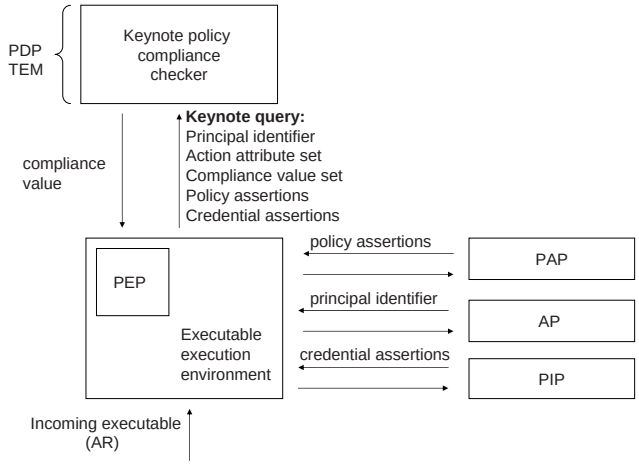

Figure 4.1: The KeyNote trust management system

certificates and to provide authentication evidence, as described in section 4.2.

- An ordered policy compliance value set can be defined within KeyNote.
- Policy administration has been considered by the developers of KeyNote, and a KeyNote toolkit has been developed [1].
- The KeyNote compliance checker inputs a query, containing a proposed action attribute set from an identified principal or principals, and determines the appropriate compliance value from a set of possible responses. The policy compliance checker essentially acts as the TEM and the PDP, as defined in section 4.2. It also meets some of the criteria for the AP.

Neither policy enforcement, nor the collation of security data pertaining to the access requester, have been considered in [13].

In sections 4.4.1 to 4.4.5 we consider the possible implementation of scenarios 1 – 6, as described in chapter 3, using the KeyNote trust management system. This will allow us to deduce whether the requirements outlined in section 4.2

[1] http://www.crypto.com/trustmgt/kn.html

can be met.

4.4.1 Scenario 1

In this section we explore how scenario 1 may be implemented using the KeyNote trust management system.

4.4.1.1 Policy statements

In order to implement scenario 1, as illustrated in figure 3.1, three requirements must be met with respect to policy statement expression — requirements 1, 2 and 3, as defined in section 4.2.1.

In order to meet these requirements the policy statement shown in figure 4.2 is defined. This policy statement delegates the authority for the 'mobile executable execution' application domain (app_domain) to RSA:12345, the public key of the device manufacturer. This statement unconditionally authorises RSA:12345 for all defined actions within the app_domain, 'mobile executable execution'. This policy also specifies that the device manufacturer is trusted to delegate authority related to the app_domain 'mobile executable execution'. It is assumed that the Internet Assigned Numbers Authority (IANA), or a similar organisation, will provide a registry of reserved app_domain names. In this registry the names and meanings of each app_domain attribute must also be defined.

```
KeyNote-Version: 2
Local-Constants: Device manufacturer == "RSA:12345"
      # public key of device manufacturer
Authorizer: "POLICY"
Licensees: Device manufacturer
Conditions: (app_domain == "mobile executable execution");
```

Figure 4.2: Scenario 1 — Policy assertion

This policy statement permits an incoming executable, which has been signed

by an (agent) code author, which has been delegated authority over the run command by the mobile device's manufacturer, which in turn has been delegated complete authority over the 'mobile execution execution' app_domain, in the policy statement defined in figure 4.2, to execute on the mobile device. In the same way, this policy permits an incoming agent, which has been signed by an (agent) code author and by an agent creator, which have been delegated authority over the run command by the mobile device's manufacturer, which in turn has been delegated complete authority over the 'mobile execution execution' app_domain in the policy defined in figure 4.2, to execute on the mobile device. Therefore, the policy statement requirements for scenario 1 can be met using the KeyNote trust management system.

4.4.1.2 Attribute certificates

In order to implement scenario 1 three requirements must be met with respect to attribute certificate expression — requirements 1, 2, and 3, as defined in section 4.2.2.

Following the generation of the policy statement defined in figure 4.2, the device manufacturer is free to generate signed assertion/credentials for (agent) code authors which he trusts to generate safe executables, as illustrated in figure 4.3. The signed assertion/credential shown in figure 4.3 permits the trusted (agent) code author to have its signed executables executed on a mobile device on which the policy statement specified in figure 4.2 has been defined.

The architectural model described in section 3.3 may require that an incoming agent is signed not only by the agent code author but also by the agent creator. In this case, the credential illustrated in figure 4.4 could be generated by the device manufacturer.

```
KeyNote-Version: 2
Authorizer: "RSA: 12345"
           #the public key of the device manufacturer
Licensees: "RSA: 654873"  # a trusted (agent) code author
Conditions: (app_domain == "mobile executable execution") &&
            (command == "run");
Signature: "RSA-SHA1: 1234534"
           #signature of device manufacturer
```

Figure 4.3: Scenario 1 — Signed assertion/credential

```
KeyNote-Version: 2
Authorizer: "RSA:12345"
            #the public key of the device manufacturer.
Licensees: ("RSA 332873" ||        # (agent) code author #1
            "RSA 120873" ||        # (agent) code author #2
            "RSA 454873") &&       # (agent) code author #3
           ("RSA 329873" ||        # (agent) creator #1
            "RSA 954973" ||        # (agent) creator #2
            "RSA 214873")
Conditions: (app_domain == "mobile executable execution") &&
            (command == "run");
Signature: "RSA-SHA1: 1234534"
       #signature of device manufacturer
```

Figure 4.4: Scenario 1 — Signed assertion/credential

The credential defined in figure 4.4 permits an incoming agent whose code and data have been digitally signed by an agent code author whose public key is listed in the <Licensees> field, and whose code, data and static state information have also been digitally signed by an agent creator whose public key is also listed in the <Licensees> field, to be executed on a mobile host on which the policy statement defined in figure 4.2 has been defined. All attribute certificate requirements for scenario 1 can be met using KeyNote.

4.4.1.3 Authentication evidence

In order to implement scenario 1, three requirements must be met with respect to authentication evidence — requirements 1, 2, and 4, as defined in section 4.2.3.

The signature of the device manufacturer on the KeyNote attribute assertion shown in figure 4.3 serves to authenticate the source of the assertion, thereby fulfilling requirement 2.

4.4.1.4 Compliance values

In order to implement scenario 1, one requirement must be met with respect to expressing the ordered compliance value set — requirement 1, as defined in section 4.2.4.

The ordered compliance value set for scenario 1 can be defined using KeyNote as shown in figure 4.5, thereby meeting this requirement. This compliance value set is input into the KeyNote compliance checker as part of a KeyNote query.

```
{discard, execute}
```

Figure 4.5: Scenario 1 — Ordered compliance value set

4.4.1.5 The PAP

In order to implement scenario 1, one requirement must be met with respect to the PAP — requirement 1, as defined in section 4.2.5. KeyNote assertions can be written using a text editor (except for the signatures). A KeyNote version 2.3 trust management toolkit and reference implementation for BSD Unix and Linux is available for download[2]. The KeyNote toolkit offers services such as asymmetric key pair generation, signature generation and signature verification. It also provides a query tool, which completes policy compliance checking on a KeyNote query. There are no toolkits available to enable the management or organisation of policy or signed assertions.

4.4.1.6 The AP

In order to implement scenario 1, three requirements must be met with respect to the AP — requirements 1, 2, and 4, as defined in section 4.2.7.

[2]http://www.crypto.com/trustmgt/kn.html

Requirement 2 can be met using the KeyNote compliance checker. Verification of the agent code author and agent creator signatures, and the verification of the incoming executable's identity, must however be completed by an independent component so that a KeyNote query can be submitted to the KeyNote compliance checker. Therefore, requirements 1 and 4 cannot be met.

4.4.1.7 The TEM

In order to implement scenario 1, one requirement must be met with respect to the TEM — requirement 1, as defined in section 4.2.8.

On receipt of a KeyNote query, comprised of:

- the identifier(s) of the principal(s) requesting the action;
- an action attribute set, for example, those illustrated in figures 4.6, 4.7 or 4.8;
- the compliance value set, as defined in figure 4.5; and
- the relevant policy and credential assertions;

the KeyNote policy compliance checker will verify the chain of trust between the principal's identifier, i.e. the _ACTION_AUTHORIZERS value input with the action attribute set and the root authorisation point 'POLICY'. The TEM requirement can be met using the KeyNote trust management system.

```
_ACTION_AUTHORIZERS = "RSA:654873"
app_domain = "mobile executable execution"
command == "run"
```

Figure 4.6: Scenario 1 — Action attribute set

```
_ACTION_AUTHORIZERS = "RSA:888888"
app_domain = "mobile executable execution"
command == "run"
```

Figure 4.7: Scenario 1 — Action attribute set

```
_ACTION_AUTHORIZERS = "RSA:332873, RSA:329873"
app_domain = "mobile executable execution"
command == "run"
```

Figure 4.8: Scenario 1 — Action attribute set

4.4.1.8 The PDP

In order to implement scenario 1, one requirement must be met with respect to the PDP — requirement 1, as defined in section 4.2.9.

The PDP requirement can be met using KeyNote. If the KeyNote query is comprised of the action attribute set illustrated in figure 4.6, the ordered compliance value set defined in figure 4.5, the policy assertion defined in figure 4.2, and the signed assertion/credential defined in figure 4.3, then the policy compliance checker will output the result {execute} to the PEP.

If the KeyNote query is comprised of the action attribute set illustrated in figure 4.7, the ordered compliance value set defined in figure 4.5, the policy assertion defined in figure 4.2, and the signed assertion/credential defined in figure 4.3, then the policy compliance checker will output the result {discard} to the PEP.

Alternatively, if the KeyNote query is comprised of the action attribute set illustrated in figure 4.8, the ordered compliance value set defined in figure 4.5, the policy assertion defined in figure 4.2 and the signed assertion/credential defined in figure 4.4, then the policy compliance checker will output the result {execute} to the PEP.

4.4.2 Scenario 2

In this section we explore how scenario 2 may be implemented using KeyNote.

4.4.2.1 Policy statements

In order to implement scenario 2, one additional requirement must be met with respect to policy statement expression — requirement 4, as defined in section 4.2.1.

In order to meet this requirement policy statements of the types shown in figures 4.9, 4.10 and 4.11 must be defined. The policy assertion defined in figure 4.9 delegates the authority to the device manufacturer with public key RSA:78547 to have its signed executables executed in the device manufacturer domain on the mobile device on which this policy statement has been defined.

```
KeyNote-Version: 2
Local-Constants: Device manufacturer == "RSA:78547"
      # public key of the device manufacturer
Authorizer: "POLICY"
Licensees: Device manufacturer
Conditions:(app_domain == "mobile executable execution") &&
           (command == "run")
                 -> "Device Manufacturer domain";
```

Figure 4.9: Scenario 2 — Policy assertion

The policy assertion defined in figure 4.10 delegates authority to the network operator with public key RSA:12345 to have its signed executables executed in the network operator domain on the mobile device on which this policy statement has been defined.

The policy assertion defined in figure 4.11 delegates authority to TTP1 with public key RSA:12885 to have its signed executables executed in the TTP do-

```
KeyNote-Version: 2
Local-Constants: Network operator == "RSA:12345"
      # public key of the network operator
Authorizer: "POLICY" Licensees: Network operator
Conditions: (app_domain == "mobile executable execution") &&
            (command == "run")
                  -> "Network operator domain";
```

Figure 4.10: Scenario 2 — Policy assertion

main on the mobile device on which this policy statement has been defined.

```
KeyNote-Version: 2
Local-Constants: TTP1 == "RSA:12885"
      # public key of the TTP1
Authorizer: "POLICY"
Licensees: TTP1
Conditions: (app_domain == "mobile executable execution") &&
            (command == "run")
                  -> "TTP domain";
```

Figure 4.11: Scenario 2 — Policy assertion

It may transpire that the public key used to verify the digital signature appended to the incoming executable is not contained in the <Licensees> field of any of the policy assertions defined above. The unknown third party, which has signed the incoming executable, may then be permitted to present an attribute certificate signed, for example, by the device manufacturer such that a chain of trust may be constructed between the mobile device and the unknown third party, as described in section 4.2.1. Requirement 1 as defined in section 4.2.1, must be met in order to facilitate this.

The first policy assertion defined in figure 4.9 delegates authority for the 'mobile executable execution' app_domain to RSA:78547, the public key of the device manufacturer. Based on this policy statement, the device manufacturer is permitted to create signed assertions for TTPs which he chooses to trust, thereby allowing them to execute in a domain less restrictive than the 'restric-

tive domain', described in section 3.4. The policy statement requirements for scenario 2 can be met using KeyNote.

4.4.2.2 Attribute certificates

In order to implement scenario 2, no additional attribute certificate requirements must be met. Requirements 1 and 5, as defined in section 4.2.1, must be fulfilled if a device manufacturer is permitted to delegate authority to a chosen TTP (not listed as one of the TTPs in the policy statement defined in figure 4.11) to have its signed executables executed in a domain less restrictive than the 'restrictive domain'. In order to meet this requirement a signed assertion/credential of the type shown in figure 4.12 must be defined by the device manufacturer for the chosen TTP.

```
KeyNote-Version: 2
Local-Constants: Device manufacturer == "RSA:78547"
      # public key of the device manufacturer
                TTP2 == "RSA:93445"
      # public key of TTP2
Authorizer: Device manufacturer
Licensees: TTP2
Conditions: (app_domain == "mobile executable execution") &&
            (command == "run")
                    -> "TTP domain";
```

Figure 4.12: Scenario 2 — Signed assertion/credential

This signed assertion/credential permits TTP2 to have its executables executed in the TTP domain on the mobile device in which the policy statement described in figure 4.9 has been defined. The attribute certificate requirements for scenario 2 can be met using KeyNote.

4.4.2.3 Authentication evidence

In order to implement scenario 2, no additional authentication evidence requirements must be met.

4.4.2.4 Compliance values

In order to implement scenario 2, one requirement must be met with respect to expressing the ordered compliance value set — requirement 2, as defined in section 4.2.4.

In scenario 2 the ordered compliance value set may be defined using KeyNote as shown in figure 4.13.

```
{discard, restrictive domain, TTP domain, network
operator domain, device manufacturer domain}
```

Figure 4.13: Scenario 2 — Ordered compliance value set

4.4.2.5 The PAP

In order to implement scenario 2, no additional PAP requirements must be met.

4.4.2.6 The AP

In order to implement scenario 2, no additional AP requirements must be met.

4.4.2.7 The TEM

In order to implement scenario 2, no additional TEM requirements must be met.

On receipt of a KeyNote query, comprised of:

- the identifier(s) of the principal(s) requesting the action;

- an action attribute set, for example those illustrated in figures 4.14, 4.15 and 4.16;
- the compliance value set, as defined in figure 4.13; and
- the relevant policy and credential assertions;

the KeyNote policy compliance checker will verify the chain of trust between the _ACTION_AUTHORIZERS value and the root authorisation point 'POLICY'.

```
_ACTION_AUTHORIZERS = "RSA:78547"
app_domain = "mobile executable execution"
command = "run"
```

Figure 4.14: Scenario 2 — Action attribute set

```
_ACTION_AUTHORIZERS = "RSA:93445"
app_domain = "mobile executable execution"
command = "run"
```

Figure 4.15: Scenario 2 — Action attribute set

```
_ACTION_AUTHORIZERS = "RSA:10105"
app_domain = "mobile executable execution"
command = "run"
```

Figure 4.16: Scenario 2 — Action attribute set

4.4.2.8 The PDP

In order to implement scenario 2, no additional PDP requirements must be met. If a KeyNote query is comprised of the action attribute set illustrated in figure 4.14, the ordered compliance value set defined in figure 4.13, and the policy assertions defined in figures 4.9, 4.10 and 4.11, then the policy compliance checker outputs the result {device manufacturer domain} to the PEP.

If the KeyNote query is comprised of the action attribute set illustrated in figure 4.15, the ordered compliance value set defined in figure 4.13, the policy as-

sertions defined in figures 4.9, 4.10 and 4.11, and the signed assertion/credential defined in figure 4.12, then the policy compliance checker outputs the result {TTP domain} to the PEP.

4.4.3 Scenario 3

In this section we will explore how scenario 3 may be implemented using the KeyNote trust management system.

4.4.3.1 Policy statements

In order to implement scenario 3, two additional requirements must be met with respect to policy statement expression — requirements 5 and 6, as defined in section 4.2.1.

In order to implement the architectural model described in figure 3.3, we give the policy assertion illustrated in figure 4.17. This policy statement delegates authority for the run command defined within the 'mobile executable execution' app_domain to TTP1, TTP2 and TTP3 when attribute 1 is less than the pre-defined value, 50, and attribute 2 is equal to the pre-defined value, true.

```
KeyNote-Version: 2
Local-Constants: TTP1 = "RSA:76595"
                 TTP2 = "RSA:24357"
                 TTP3 = "RSA:02755"
      # public keys of selected TTPs
Authorizer: "POLICY"
Licensees: TTP1 || TTP2 || TTP3
Conditions:((app_domain == "mobile executable execution") &&
            (command == "run")
            -> { (@(attribute 1) < "50"   &&
                   (attribute 2) = "true")
                   };
```

Figure 4.17: Scenario 3 — Policy assertion

In order to implement the architectural model described in figure 3.4, we give the policy assertion illustrated in figure 4.18. This policy statement delegates

authority for the run command defined within the 'mobile executable execution' app_domain to TTP1, TTP2 and TTP3 when test 1 is equal or not equal to 'completed' and test 2 is equal or not equal to 'completed'.

- If test 1 and test 2 have been completed, the executable is authorised to execute in the least restrictive domain.
- If test 1 has been completed and test 2 has not been completed, the executable is authorised to execute in the restrictive domain.
- If test 1 and test 2 have not been completed, then the executable is authorised to execute in the most restrictive domain.

```
KeyNote-Version: 2
Local-Constants: TTP1 == "RSA:76595"
                 TTP2 == "RSA:24357"
                 TTP3 == "RSA:02755"
                 # public keys of TTPs
Authorizer: "POLICY"
Licensees: TTP1 || TTP2 || TTP3
Conditions:((app_domain == "mobile executable execution") &&
            (command == "run")
                -> {((test 1) == "completed" &&
                      (test 2) == "completed")
                      ->  "least restrictive domain";
                    ((test 1) == "completed" &&
                      (test 2) != "completed")
                      ->  "restrictive domain";
                     ((test 1) != "completed" &&
                       (test 2) != "completed")
                      ->  "most restrictive domain";
                     };
```

Figure 4.18: Scenario 3 — Policy assertion

If ACLs are introduced in order to allow a selection of (agent) code authors to have their untested executables executed on the mobile device, a policy statement of the form shown in figure 4.19 must be specified, in addition to the policy assertion defined in either figure 4.17 or figure 4.18. This policy statement delegates authority for the run command defined within the 'mobile executable execution' app_domain to authors 1, 2, 3 and 4.

```
KeyNote-Version: 2
Authorizer: "POLICY"
Licensees: ("RSA:12579" || # public key of (agent) code author 1
            "RSA:14972" || # public key of (agent) code author 2
            "RSA:26579" || # public key of (agent) code author 3
            "RSA:32578")   # public key of (agent) code author 4
Conditions:(app_domain == "mobile executable execution") &&
           (command == "run");
```

Figure 4.19: Scenario 3 — Policy assertion

```
KeyNote-Version: 2
Authorizer: "POLICY"
Licensees: ("RSA:12579" ||    # public key of agent code author 1
            "RSA:14972" ||    # public key of agent code author 2
            "RSA:26579" ||    # public key of agent code author 3
            "RSA:32578")&&  # public key of agent code author 4
           ("RSA:56779" ||    # public key of agent creator 1
            "RSA:18872" ||    # public key of agent creator 2
            "RSA:86479" ||    # public key of agent creator 3
            "RSA:46578")      # public key of agent creator 4
Conditions:(app_domain == "mobile executable execution") &&
           (command == "run");
```

Figure 4.20: Scenario 3 — Policy assertion

If ACLs are introduced in order to allow a selection of agent code authors and agent creators to have their untested agents executed on the mobile device, a policy statement of the form shown in figure 4.20 must be specified, in addition to the policy assertion defined in either figure 4.17 or figure 4.18. This policy statement delegates authority for the run command defined within the 'mobile executable execution' app_domain to an agent code author listed in the <Licensees> field of figure 4.20 and an agent creator listed in the <Licensees> field in figure 4.20.

If ACLs are, alternatively, introduced in order to increase the security of the system, the policy statements defined in figures 4.17 and 4.18 must be modified as illustrated in figures 4.21 and 4.22 below. The policy statement defined in figure 4.21 delegates authority for the run command defined within the 'mobile executable execution' app_domain to TTP1, TTP2 and TTP3, where the code author's identity is listed as a value in the (agent) code author name element of

the <Condition> field, the agent creator's identity is listed in the agent creator name element of the <Condition> field, attribute 1 is greater than 50, and attribute 2 is equal to true.

```
KeyNote-Version: 2
Local-Constants: TTP1 == "RSA:76595"
                 TTP2 == "RSA:24357"
                 TTP3 == "RSA:02755"
                 # public keys of TTPs
Authorizer: "POLICY" Licensees: TTP1 || TTP2 || TTP3
Conditions:((app_domain == "mobile executable execution") &&
            (command == "run") &&
            ((agent) code author name == "Micro Ltd" ||
             (agent) code author name == "Orange Ltd") &&
            (agent creator name == "Agents Ltd" ||
             agent creator name == "Airline Ltd")
                  -> { (@(attribute 1) < 50 &&
                          (attribute 2) == "true")
                     };
```

Figure 4.21: Scenario 3 — Policy assertion

The policy statement defined in figure 4.22 delegates authority for the run command defined within the 'mobile executable execution' app_domain to TTP1, TTP2 and TTP3, where the author's identity is listed as a value in the (agent) code author name element of the <Condition> field, the agent creator's identity is listed in the agent creator name element of the <Condition> field, test 1 is equal or not equal to 'completed' and test 2 is equal or not equal to 'completed'.

- If the code author's identity is listed as a value in the (agent) code author name element of the <Condition> field, the agent creator's identity is listed in the agent creator name element of the <Condition> field, test 1 has been completed, and test 2 has been completed, the executable is authorised to execute in the least restrictive domain.
- If the code author's identity is listed as a value in the (agent) code author name element of the <Condition> field, the code creator's identity is listed

in the agent creator name element of the <Condition> field, test 1 has been completed, and test 2 has not been completed, the executable is authorised to execute in the restrictive domain.

- If the code author's identity is listed as a value in the (agent) code author name element of the <Condition> field, the code author's identity is listed in the agent creator name element of the <Condition> field, test 1 has not been completed, and test 2 has not been completed, the executable is authorised to execute in the most restrictive domain.

The policy statement requirements for scenario 3 can be met using KeyNote.

```
KeyNote-Version: 2
Local-Constants: TTP1 = "RSA:76595"
                 TTP2 = "RSA:24357"
                 TTP3 = "RSA:02755"
Authorizer: "POLICY"
Licensees: TTP1 || TTP2 || TTP3
Conditions:((app_domain = "mobile executable execution") &&
            (command == "run") &&
            ((agent) code author name == "Micro Ltd" ||
             (agent) code author name == "Orange Ltd") &&
            (agent creator name == "Agents Ltd" ||
             agent creator == "Code Ltd")
              -> {((test 1) == "completed"&&
                    (test 2) == "completed")
                     ->  "least restrictive domain";
              ->  ((test 1) == "completed"&&
                   (test 2) != "completed")
                     ->  "restrictive domain";
              ->  ((test 1) != "completed"&&
                   (test 2) != "completed")
                     ->  "most restrictive domain");
                   };
```

Figure 4.22: Scenario 3 — Policy assertion

4.4.3.2 Attribute certificates

In order to implement scenario 3 two additional attribute certificate requirements must be met — requirements 5 and 6, as defined in section 4.2.2.

Signed assertions/credentials, as defined in the KeyNote trust management

system, do not enable the expression of signed attribute certificates in which the attributes pertaining to an executable are described by a TTP. Another means would therefore be required in order for a TTP to certify the attributes of an executable which they have tested. Hence the attribute certificate requirements for scenario 3 cannot be met using KeyNote.

4.4.3.3 Authentication evidence

In order to implement scenario 3, one additional requirement must be met with respect to authentication evidence — requirement 3, as defined in section 4.2.3.

This requirement cannot be met however as the KeyNote attribute assertion which the TTP is required to sign cannot be expressed, as described in the previous section.

4.4.3.4 Compliance values

In order to implement the architectural model described in section 3.5 and illustrated in figure 3.3, one requirement must be met with respect to expressing the ordered compliance value set — requirement 1, as defined in section 4.2.4. This ordered compliance value set has previously been defined in figure 4.5.

In order to implement the architectural model described in section 3.5, and illustrated in figure 3.4, one requirement must be met with respect to expressing the ordered compliance value set — requirement 2, as defined in section 4.2.4. This ordered compliance value set is defined in figure 4.23. The compliance value set requirement for scenario 3 can thus be met using KeyNote.

```
{discard, most restrictive domain, restrictive
domain, least restrictive domain}
```

Figure 4.23: Scenario 3 — Ordered compliance value set

4.4.3.5 The TEM

In order to implement scenario 3, no additional TEM requirements must be met. On receipt of a KeyNote query, the KeyNote policy compliance checker will verify the chain of trust between the principal identifier, i.e. the _ACTION_AUTHORIZER in the KeyNote query and the root authorisation point 'POLICY'. For the architectural model described in figure 3.3, the KeyNote action attribute sets shown in figures 4.24 and 4.25 are illustrative of what may be input into the KeyNote compliance checker.

```
_ACTION_AUTHORIZERS = "RSA:76595"
app_domain = "mobile executable execution"
command == "run"
attribute 1 = "20"
attribute 2 = "true"
```

Figure 4.24: Scenario 3 — Action attribute set

```
_ACTION_AUTHORIZERS = "RSA:11115"
app_domain = "mobile executable execution"
command == "run"
attribute 1 = "20"
attribute 2 = "true"
```

Figure 4.25: Scenario 3 — Action attribute set

For the architectural model described in figure 3.4, the KeyNote action attribute set shown in figure 4.26 is illustrative of what may be input into the KeyNote compliance checker.

```
_ACTION_AUTHORIZERS = "RSA:76595"
app_domain = "mobile executable execution"
command == "run"
test 1 = completed
test 2 = completed
```

Figure 4.26: Scenario 3 — Action attribute set

For the architectural model described in figure 3.3, where ACLs are added

to provide improve efficiency, the action attribute set shown in figure 4.27 is illustrative of what may be input into the KeyNote compliance checker.

```
_ACTION_AUTHORIZERS = "RSA:12579" || "RSA:18872"
app_domain = "mobile executable execution"
command == "run"
```

Figure 4.27: Scenario 3 — Action attribute set

4.4.3.6 The PDP

In order to implement scenario 3, no additional PDP requirements must be fulfilled. If the KeyNote query is comprised of the action attribute set illustrated in figure 4.24, the ordered compliance value set defined in figure 4.5, and the policy assertion defined in figure 4.17, then the policy compliance checker outputs the result {execute} to the PEP.

If the KeyNote query is comprised of the action attribute set illustrated in figure 4.25, the ordered compliance value set defined in figure 4.5, and the policy assertions defined in figure 4.17, then the policy compliance checker outputs the result {discard} to the PEP.

If the KeyNote query is comprised of the action attribute set illustrated in figure 4.26, the ordered compliance value set defined in figure 4.23, and the policy assertions defined in figure 4.18, then the policy compliance checker outputs the result {least restrictive domain} to the PEP.

If the KeyNote query is comprised of the action attribute set illustrated in figure 4.27, the ordered compliance value set defined in figure 4.5, and the policy assertions defined in figures 4.17 and 4.20, then the policy compliance checker outputs the result {execute} to the PEP.

4.4.4 Scenario 4

We next explore how scenario 4 may be implemented using KeyNote. No additional requirements in terms of policy statements, attribute certificates, compliance values, the PDP or the TEM need to be met in order to implement scenario 4.

4.4.4.1 Policy statements

To support the architectural model illustrated in figure 3.5, a mobile device policy statement of the type shown in figure 4.28 must be defined. This policy statement delegates authority for the run command defined within the 'mobile executable execution' app_domain to the domain server with public key RSA:76595, when attribute 1 is less than the pre-defined value, 50, and attribute 2 is equal to the pre-defined value, true.

```
KeyNote-Version: 2
Local-Constants: Domain server = "RSA:76595"
      # public keys of the mobile device domain server
Authorizer: "POLICY"
Licensees: Domain server
Conditions:((app_domain == "mobile executable execution") &&
            (command == "run")
            -> { (@(attribute 1) < "50"   &&
                   (attribute 2) = "true")
                 };
```

Figure 4.28: Scenario 4 — Policy assertion

4.4.4.2 Attribute certificates

Signed assertions/credentials, as defined in the KeyNote trust management system, do not allow the expression of signed attribute certificates in which the attributes pertaining to an executable are described by the domain server to whom the authorisation is delegated in the policy statement defined in fig-

ure 4.28. Another means would therefore be required in order for a domain server to certify the attributes of an executable which they have tested.

4.4.4.3 Authentication evidence

In order to implement scenario 4, no additional requirements must be met with respect to authentication evidence.

4.4.4.4 Compliance values

The ordered compliance value set required for scenario 4 has been previously defined in figure 4.5.

4.4.4.5 The TEM

In order to implement scenario 4, no additional TEM requirements must be met. In the architectural model described in figure 3.5, the action attribute set shown in figure 4.29 is illustrative of what may be input into the KeyNote compliance checker.

```
_ACTION_AUTHORIZERS = "RSA:76595"
app_domain = "mobile executable execution"
command == "run"
attribute 1 = "20"
attribute 2 = "true"
```

Figure 4.29: Scenario 4 — Action attribute set

4.4.4.6 The PDP

In order to implement scenario 4, no additional PDP requirements must be fulfilled. If the KeyNote query contains the action attribute set illustrated in figure 4.29, the ordered compliance value set defined in figure 4.5, and the policy assertion defined in figure 4.28, then the policy compliance checker outputs the

result {execute} to the PEP.

4.4.5 Scenarios 5 and 6

In this section we explore how scenarios 5 and 6 may be implemented using KeyNote. No additional requirements in terms of policy statements, attribute certificates, compliance values, the TEM or the PDP need to be fulfilled in order to implement the mobile device policy engine for scenarios 5 and 6.

4.4.5.1 Policy statements

In order to implement scenarios 5 and 6, one requirement must be met with respect to mobile device policy statement expression — requirement 4, as defined in section 4.2.1. In order to meet this requirement, a policy statement of the form described in figure 4.30 must be defined. This policy statement delegates authority for the run command defined within the 'mobile executable execution' app_domain to RSA:11345, the public key of the device manufacturer.

```
KeyNote-Version: 2
Local-Constants: Device server = "RSA:11345"
      # public key of domain server
Authorizer: "POLICY"
Licensees: Domain server
Conditions: (app_domain == "mobile executable execution") &&
            (command == "run");
```

Figure 4.30: Scenario 5 and 6 — Policy assertion

4.4.5.2 Attribute certificates

No attribute certificate requirements are defined for scenarios 5 and 6.

4.4.5.3 Authentication evidence

In order to implement scenarios 5 and 6, no additional requirements must be met with respect to authentication evidence.

4.4.5.4 Compliance values

The ordered compliance value set required for scenarios 5 and 6 has been previously defined in figure 4.5.

4.4.5.5 The TEM

In order to implement scenarios 5 and 6, no additional TEM requirements must be fulfilled. In the architectural model described in figure 3.7, the action attribute set shown in figure 4.31 is illustrative of what may be input into the KeyNote compliance checker.

```
_ACTION_AUTHORIZERS = "RSA:11345"
app_domain = "mobile executable execution"
command == "run"
```

Figure 4.31: Scenario 5 and 6 — Action attribute set

4.4.5.6 The PDP

In order to implement scenarios 5 and 6, no additional PDP requirements must be fulfilled. If a KeyNote query is comprised of the action attribute set illustrated in figure 4.31, the ordered compliance value set defined in figure 4.5, and the policy assertion defined in figure 4.30, then the policy compliance checker outputs the result {execute} to the PEP.

4.4.6 Conclusions

PIP functionality has not been considered within the KeyNote system, i.e. there are no mechanisms to collect the required or missing signed assertions. The KeyNote compliance checker meets the requirements of the TEM, the PDP and some of the AP requirements. A basic toolkit also provides PAP functionality.

The syntax of the KeyNote assertion expression language is based on the format of RFC-822 style message headers, and is reasonably easy for the non-programmer to understand. The same language is used for both policy assertion and signed assertion/credential expression, which simplifies the language further. While the policy statement requirements could be met, issues arose in relation to limiting delegation and the expression of fine-grained access control policies. While the majority of attribute certificate requirements could be met, it also became apparent that there is no way to create an attribute certificate in which there is no inherent notion of delegation of authority. We now examine the areas where KeyNote falls short with respect to policy statement and attribute assertion expression in further detail.

The policy statements defined throughout this section are coarse-grained. In scenarios 1, 4, 5 and 6, and potentially in scenario 3, an executable is either permitted to execute or is discarded. In scenario 2, and potentially in scenario 3, a label indicating the 'execution environment' in which the executable should be permitted to execute is output. It then becomes the responsibility of the execution environment (acting as the PEP) to interpret this policy decision and act accordingly. If the decision output from the policy compliance checker is {execute}, for example, this could imply that the incoming executable is permitted to execute with all privileges or, alternatively, there may be a specified set of actions which the executable is permitted to perform. As a result, a separate set

of security policies may be required within the execution environment so that the PEP can interpret what the decision of the KeyNote compliance checker implies for executable execution.

Finer-grained policy statements could be specified using the KeyNote assertion specification language. The policy statement defined in figure 4.2 could, for example, be represented as a number of finer-grained policies which specify the exact privileges over which the device manufacturer has authority and can delegate. An example of such a policy is shown in figure 4.32. This policy assertion specifies that the device manufacturer has authority for read and write operations over two file directories, 'directory 1' and 'directory 2', where the app_domain is file_system. This implies that the device manufacturer is authorised to perform or to delegate these actions on these two directories.

```
KeyNote-Version: 2
Local-Constants: Device manufacturer == "RSA:12345"
      # public key of device manufacturer
Authorizer: "POLICY"
Licensees: Device manufacturer
Conditions:(app_domain == "file_system") &&
           (directory == "directory 1" || directory == "directory 2") &&
           (access == "read" || access == "write");
```

Figure 4.32: Scenario 1 (Alternative) — Policy assertion

A further policy could specify that the device manufacturer has authority over a selection of actions defined within the app_domain 'outbound connections'. The policy assertion shown in figure 4.33 permits the device manufacturer to delegate the authority to accept, resolve or connect to particular IP addresses using a specified set of port numbers, as shown in figure 4.33.

For each set of related security permissions, defined by an app_domain, a separate policy statement must be defined. Following the generation of the policy statements defined above, the device manufacturer can delegate the authority

```
KeyNote-Version: 2
Local-Constants: Device manufacturer == "RSA:12345"
      # public key of device manufacturer
Authorizer: "POLICY"
Licensees: Device manufacturer
Conditions: (app_domain == "outbound connections") &&
            (ipaddress ==  ipadress1 ||
             ipaddress == ipadress2) &&
            (port == 3 || port == 777) &&
            (connection_method == accept ||
             connection_method == connect ||
             connection_method == resolve);
```

Figure 4.33: Scenario 1 (Alternative) — Policy assertion

for a defined set of execution permissions to a set of trusted code authors, or a set of agent code authors and agent creators.

When an incoming executable begins executing, every attempted action is intercepted by the execution environment, and a request sent to the KeyNote policy compliance checker with the required action attribute set, policy assertions, signed assertions and an ordered compliance value set, for example {allow, disallow}. A decision is then output by the policy compliance checker and enforced by the execution environment (acting as the PEP).

This approach, however, has a number of disadvantages.

- Firstly, in the example described above, the number of signed assertions acquired by code authors (and potentially agent creators) may quickly increase, particularly if each device manufacturer that certifies code authors (and potentially agent creators) must issue them with multiple credentials in order to delegate the required execution permissions. This problem is compounded by the fact that the KeyNote trust management system does not support a mechanism to collect required or missing assertions.
- Secondly, all permissions delegated in signed assertions become publicly visible as they are distributed across the network, which may lead to a security breach.

- Thirdly, on examination of the policy statements defined in figures 4.32 and 4.33, it becomes clear that if, for example, these policy assertions are altered so that the authority delegated by 'POLICY' is made more restrictive, then a device manufacturer would have to re-issue all signed assertions pertaining to the modified permissions to each code author considered trusted, or, potentially, to each set of agent code authors and agent creators considered trusted.

It would therefore appear better to implement KeyNote policies as described in sections 4.4.1 to 4.4.5. In this way, a generic permission may be granted to the executable, e.g. {Device Manufacturer domain}, and this decision can in turn be mapped to a specific set of access rights by the execution environment responsible for enforcing the decision output by the KeyNote policy compliance checker.

While the majority of the attribute certificate requirements described in section 4.2.2 can also be fulfilled through the use of KeyNote, there is no way to create a credential in which there is no inherent notion of delegation of authority [136]. Requirement 6, as described in section 4.2.2, cannot be fulfilled using KeyNote signed assertions.

In conjunction with this, a problem may arise with respect to limiting delegation. For example, the attribute certificate which delegates the authority to a code author to run executables on the host device, as defined in figure 4.3, also permits that code author to delegate his authority over the run command to other untrusted entities, as illustrated in figure 4.34.

A way of specifying assertions such that authority cannot be delegated further is presented by Foley et al. [49]. They suggest that by stating as a <Condition> that the _ACTION_AUTHORISOR must be the licensee of the

```
KeyNote-Version: 2
Authorizer: "RSA: 654873"
            #the public key of a trusted (agent) code author
Licensees: "RSA: 333455"  # an entity chosen by the code author
Conditions: (app_domain == "mobile executable execution") &&
             (command == "run");
Signature: "RSA-SHA1: 5634534"
            #signature of (agent) code author
```

Figure 4.34: Scenario 1 (Alternative) — Signed assertion/credential

assertion, further delegation can be prevented. We now re-examine the signed assertion described in figure 4.3 in the light of this proposal. The signed assertion/credential described in figure 4.35 permits the trusted (agent) code author, represented by RSA:654873, to have their signed executables executed on a mobile device on which the policy statement specified in figure 4.2 has been defined. If this author creates a signed assertion for another entity represented by RSA:33337, who then attempts to have their executable executed on the device, a decision of {discard} would be output by the policy compliance checker because the _ACTION_AUTHORISER value in the KeyNote query would not meet the required condition.

```
KeyNote-Version: 2
Authorizer: "RSA: 12345"
            #the public key of the device manufacturer
Licensees: "RSA: 654873"  # a trusted (agent) code author
Conditions: (app_domain == "mobile executable execution") &&
             (command == "run") &&
             (_ACTION_AUTHORISOR == RSA: 654873);
Signature: "RSA-SHA1: 1234534"
            #signature of device manufacturer
```

Figure 4.35: Scenario 1 (Alternative) — Signed assertion/credential

Alternatively, it would be useful if KeyNote could be extended to provide additional functionality, as described in the Nereus authorisation framework, in order to counteract these weaknesses. Nereus was developed by Miklos [103]. This policy authorisation framework is very similar to KeyNote. However, there

are three fundamental differences between KeyNote and Nereus [103]: use of attributes; limited delegation; and a different compliance checker.

- Nereus allows attributes to be bound to public keys in signed binding assertions. Binding of authorisations to attributes is also enabled in binding policy assertions.
- Through the introduction of binding and delegation policy assertions, and signed binding and delegation assertions, delegation can be limited.
- A different compliance checking method is used in Nereus.

In the Nereus system, two types of policy assertion and two types of signed assertion are defined. Policy assertions include binding assertions and delegation assertions.

- Binding policy assertions assign authorisations to attributes.
- Delegation policy assertions delegate the right to define policies.

Policy assertion definition is not limited to the principal whose identifier is 'POLICY'. Miklos does, however, describe a special policy assertion, which serves as the trust root in the compliance checker and is generated by the principal, whose identifier is 'POLICY'.

Signed assertions include signed binding assertions and signed delegation assertions.

- Signed binding assertions bind attributes to public keys.
- Signed delegation assertions delegate the right to bind specific attributes to keys.

We now explore how the model described in figure 3.3 might be implemented using Nereus. Instead of the policy statement specified in figure 4.17, the following policy statements can be defined using the Nereus assertion specification language. The policy statement defined in figure 4.36 permits a principal with the attributes listed in the <Licensees> field to have their executables executed on the mobile device on which this policy is specified.

```
Nereus-Version: 2
Issuer: "POLICY"
Type: binding
Licensees: @ (attribute 1) < "50"   &&
             (attribute 2) = "true")
                  ASSERTED BY RSA:12345 || RSA:678910
                  # public keys of TTP1 || TTP2
Conditions: (app_domain == "mobile executable execution") &&
            (command == "run");
```

Figure 4.36: Scenario 3 (Alternative) — Policy assertion

The policy statement defined in figure 4.37 permits TTP1 or TTP2, represented by the public keys RSA:12345 and RSA:678910, to bind specific attributes to a subject.

```
Nereus-Version: 2
Issuer: "POLICY"
Type: delegation
Licensees: RSA:12345 || RSA:678910
Attribute name: attribute 1
Attribute value: < "100"
Attribute name: attribute 2
Attribute value: "true" || "false"
```

Figure 4.37: Scenario 3 (Alternative) — Policy assertion

TTP1 or TTP2 can then generate a signed binding assertion, as shown in figure 4.38, which binds the specified attributes to a particular subject.

Presumably, if a KeyNote query comprising the policy assertions described

```
Nereus-Version: 2
Issuer: TTP1
Type: binding
Licensees: ‘principal’
Attribute name: attribute 1
Attribute value: 42
Attribute name: attribute 2
Attribute value: true
Signature: "RSA-SHA1: f125dt"
```

Figure 4.38: Scenario 3 (Alternative) — Signed assertion

in figures 4.36 and 4.37, the signed assertion described in figure 4.38, an order compliance value set {execute, discard}, and an action attribute set as defined in figure 4.39 were input to the Nereus compliance checker, then the compliance checker would output the value, {execute}.

```
_ACTION_AUTHORIZERS = ‘principal’
app_domain = "mobile executable execution"
command == "run"
```

Figure 4.39: Scenario 3 (Alternative) — Action attribute set

In the above example (and in figures 4.38 and 4.39), we require that ‘principal’ is an identifier for the mobile executable, for example a hash of the executable. Unfortunately, in the Nereus system, unlike in the KeyNote trust management system, where the <Licensee> field of an assertion can contain a cryptographic key, or, indeed, an opaque identifier string, whose structure is not interpreted by the KeyNote system, it is required that the <Licensee> field contains a key identifier. This implies that the attribute assertion defined in figure 4.38 and, therefore, the query described above, cannot be specified. Further, difficulties in providing a definite assessment arise because a document which describes the syntax and semantics of the Nereus assertion specification language has not, as yet, been published.

If both cryptographic keys and opaque identifier strings were permitted in

the <Licensee> field of a Nereus assertion, then this authorisation system would solve two problems, that of unlimited delegation and the requirement for binding assertions. Alternatively, KeyNote may benefit from re-examination and extension.

4.5 Ponder

Ponder is a declarative, strongly typed, object-oriented policy specification language [34, 35, 37]. It can be used in the specification of both management and security policies. Every Ponder policy relates to objects with interfaces defined in terms of methods using an interface definition language [36]. A subject object refers to "an entity, i.e. a user or automated component, with management responsibility" [36]. "A subject object can access a target object by invoking methods visible on the target's interface" [36]. The notion of a domain is also used. A domain is defined as "a means of grouping objects to which policies apply" [36].

- Ponder policies are expressed in terms of subject object and target object sets called domains.
- Path names are used to identify domains, e.g. A/B/C.
- "Membership of a domain is explicit and not defined in terms of a predicate on object attributes" [37].
- Domain scope expressions are used to combine domains to form a set of objects for applying a policy [37].

Both basic and composite policy statements can be expressed. Basic policy statements can be coarsely categorised into access control policies and obligation

policies. Access control policy types include authorisation polices, information filtering policies and delegation policies.

- *Authorisation policies* define the actions that a member of a subject domain can perform on the objects in a target domain. Both positive authorisation policies and negative authorisation policies are supported. *Positive authorisation policies* define the actions that subject objects can perform on target objects. *Negative authorisation policies* define the actions that subject objects are forbidden from performing on target objects, and prove useful in the temporary removal of access rights from subjects.

- *Information filtering policies* transform the values of the input and/or output parameters in an action.

- *Delegation policies* allow for the temporary transfer of access rights. A policy of this type permits subject objects to grant privileges, which they possess, to grantees, so that the grantees can perform actions on their behalf. Delegation policies may be either positive, thereby allowing the delegation of certain actions, or negative, thereby forbidding the delegation of certain actions.

Obligation policy types include obligation policies and refrain policies.

- Obligation policies define the actions that must be performed by managers within the system should certain events occur, for example, security violations.

- Refrain policies define actions that subjects must not perform. They are similar to negative authorisation policies but, rather than being enforced by target object access controllers, as is the case with authorisation policies, they are enforced by subject objects.

The validity of each basic policy may be dependent on a set of conditions, for example, subject/target state-based constraints, action/event parameters or time constraints.

As well as the basic policy construction, composite policy construction is also defined, so that policy specification and management in large complex systems may be simplified. Constructs defined in order to enable policy specification in large complex organisations include the following [35].

- Groups associate related policies so that policies may be organised efficiently.
- Roles group policies with a common subject.
- Relationships group policies defining the rights and duties of roles towards each other. Relationships may also group policies related to resources that are shared by the roles [35].
- Management structures "define an organisation in terms of instances of roles, relationships and nested management structures relating to organisational units" [35].
- Meta policies specify constraints over a set of policies.

We now examine the Ponder policy specification framework, illustrated in figure 4.40, in order to determine which of the functional components, as described in section 4.2, are provided, and whether policy statement expression, attribute certificate expression, ordered compliance value definition and authentication evidence expression are supported.

- Management tools enable the specification and administration of policies. The Ponder toolkit, which is available for download[3], includes a domain

[3]http://www-dse.doc.ic.ac.uk/Research/policies/ponder.shtml

browser; a policy editor; a policy compiler; a management console; a user-role management tool; a GUI component; and a configuration manager tool. A domain service manages the hierarchy of domain objects. These management tools and the domain service provide the PAP component of a policy engine, as defined in section 4.2.

- An access controller is defined as an agent which receives an authorisation request, makes a policy decision, and then enforces this decision. One access controller exists for each target object. In conjunction with this, the access controller applies any authorisation filters to the action call parameters and the returned values. Access controllers provide PDP and PEP functionality, as defined in section 4.2.

- A policy management component is defined as an agent which enforces obligation and refrain policies. One policy management component is defined for each subject object. As obligation and refrain policies define the actions that must or must not be performed by managers within the system should certain events occur, these policies depend on input from an event service. Policy management components provide PDP and PEP functionality, as defined in section 4.2.

- A number of policy types can be specified, namely positive and negative authorisation policies, filtering policies, delegation policies, obligation policies and refrain policies. A set of conditions under which each policy specified is valid may also be specified. The Ponder policy specification language also enables composite policy definition.

PIP functionality is not considered in the Ponder policy specification framework. It is required that objects are explicitly added to domains. Membership of domains cannot be defined in terms of a predicate on object attributes. There-

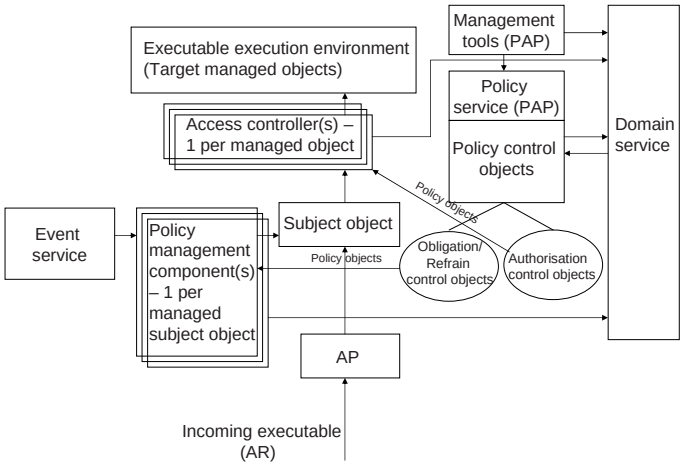

Figure 4.40: The Ponder policy specification framework

fore, there is no requirement to collect access requestor information. It is also assumed that authentication is completed independently of the Ponder policy specification framework, so AP functionality is not considered. As a result of this, neither the expression of attribute certificates nor authentication evidence are considered in the Ponder policy specification framework. Because policy decision making and policy enforcement are implemented within the same component, policy compliance values are not required.

In the following sections, we examine whether the requirements outlined in section 4.2 for policy statements, the PAP, the PDP and the PEP are met by Ponder. In order to do this, we attempt to express the architectural models outlined in chapter 3 using Ponder.

4.5.1 Prior art

There are two notable examples of the use of the Ponder policy specification framework to specify security policies for mobile agents. Knottenbelt [94] extended the April Agent Platform to support the management and evaluation of security policies specified in Ponder. Montanari et al. [10, 30, 105] have also

demonstrated how the Ponder specification language may be utilised in order to control agent execution in their agent infrastructure, SOMA.

The April Agent Platform is a Foundation for Intelligent Physical Agents (FIPA) compliant agent platform. A FIPA compliant agent platform is defined as a "physical infrastructure in which agents can be deployed" [50] and is comprised of three components: an agent management system; a message transport service, and an optional directory facilitator component.

- An agent management system (AMS) is a central directory service containing a list of identifiers for all agents registered with the agent platform. An agent identifier is made up of a globally unique immutable agentName, and other optional parameters, such as the transport addresses at which the agent can be contacted. Each agent must register with an AMS in order to get a valid agent identifier. An agent platform's AMS can then be queried by agentName to provide information about its registered agents. An agent platform's AMS is responsible for managing the operation of the platform, such as the creation of agents, the deletion of agents, deciding whether an agent can dynamically register with the platform, and overseeing the migration of agents to and from the agent platform (if agent mobility is supported). Only one AMS will exist in an agent platform.
- A directory facilitator (DF) can be used by agents, for example, to discover which agents provide particular services. Multiple DFs may exist within an AP and may be federated [50].
- A message transport service provides the standard way of communicating between agents on different agent platforms.

Knottenbelt [94] extends the basic April Agent Platform with four additional components.

- A policy repository service (PRS), which stores compiled policies and an indication as to whether policies are enabled or disabled.
- An agent directory service (ADS), which associates agents with their policy domains.
- An agent station, which receives requests to start and stop agents. There is generally one agent station per host machine within the agent platform.
- A controller, which represents the restricted environment in which an agent is executed. Controllers apply policies to incoming and outgoing agent messages. One controller is invoked per agent executing on a host machine.

On migration to an agent platform, a mobile agent must be registered with the agent platform's AMS. An agent station can then receive a request of the form *Agent*(*agentName*, *domain*, *runProc*), where *agentName* is the name of the agent registered with the AMS, *domain* is the name of the policy domain to which the agent wishes to be associated, and *runProc* is the April byte-code for the procedure which starts the agent [94]. The controller first queries the ADS in order to resolve the agent to a domain. This can only be done if the agent has been previously registered with the ADS. If the agent has not been registered, the controller registers the agent/domain tuple with the ADS and then downloads any policies relevant to the agent from the PRS and enforces them. The agent station then forks off a controller process for the new agent.

In his report Knottenbelt [94] defines two examples of agents.

- The factorAgent, which has domain name services/computing/factoriser, inputs a request of the form (factor, N), where N is the number to be factorised. After performing the computation, it replies with a message of the form (result, Factors) where Factors is the list of prime factors.

- The clockAgent, which has domain name services/clock/periodic, is an even simpler agent, which raises a timer event of the form (tick, Time) every five seconds. The Time is a floating point number indicating the number of seconds elapsed since the Epoch (1st January 1970).

Two examples of Ponder policies are also defined [94]. The first policy, as shown in figure 4.41, is called FactorPolicy. It is a negative authorisation policy restricting the factorisation of very large numbers by the agent contained in the domain /services/computing/factoriser, which contains a reference to the factorisation agent defined above.

```
inst auth- FactorPolicy {
       subject /;
       target /services/computing/factoriser;
       action factor(N);
       when N >= 1000;
   }
```

Figure 4.41: The Extended April Agent Platform — Sample policy statement

The second example policy statement, as shown in figure 4.42, is an obligation policy called ClockPolicy, which every three tick events requests the factorisation of the current time divided by 100 from the factor agent.

```
inst oblig ClockPolicy {
       on      3 * tick(Time);
       subject /services/clock/periodic;
       target  /;
       do      factorAgent.factor(Time/100);
   }
```

Figure 4.42: The Extended April Agent Platform — Sample policy statement

As is clear from the examples outlined in figures 4.41 and 4.42, all policies are defined in terms of subject domains and target domains, of which particular agents are members. As described above, when an agent is generated, it is

given an agent name. The agent is also associated with a domain name. When a mobile agent migrates to an agent platform, the agent name must then be registered with the AMS of the agent platform, and the agent/domain tuple must be registered by the ADS, so that the policies pertaining to the domain with which the agent is bound can be enforced.

This model assumes that a potentially unknown and/or malicious entity can be trusted to name an agent and to choose the domain with which the agent should be registered. In this way, a potentially unknown and/or malicious entity is permitted to decide which policies should be applied to its agent. This model also assumes that all agent creators can obtain the list of domains on each agent platform on which they want their agent to execute, and this is not a reasonable assumption in some scenarios. Another problem arises in relation to the integrity of the domain name associated with the agent name. No mechanisms are described in [94] in order to protect the domain name from malicious alteration. The issue of establishing trust in the incoming agent is not tackled.

Montanari et al. [10,30,105] have also demonstrated how the Ponder specification language may be utilised in order to control agent execution. The policy architecture integrated into SOMA provides a number of services [30].

- A policy specification service enables policies to be edited, updated and removed.
- A policy repository service stores all currently active policies, and can be queried to retrieve policies.
- A policy coordinator service is responsible for distributing policies to run-time entities at policy instantiation, and at any subsequent change.
- An authorisation enforcement service is responsible for applying run-time

access controls.

- An obligation enforcement service is responsible for the correct enforcement of obligation policies.

Montanari et al. have also given examples of Ponder policy statements which may be defined within their agent platform, SOMA [30]. The policy statement shown in figure 4.43 specifies that backup agents have read access to all files of managed machines nodes at backup hours, i.e. from 9 p.m. to 6 a.m., every day.

```
type auth+ backupP
          (subject bkagents, target files) {
           action read;
           when time.between (2100, 2359) or
                         time.between (0000, 0600);
       }

inst auth+ b = backupP (backup/agents, nodes/files);
```

Figure 4.43: SOMA — Sample policy statement

Once again, as is clear from the sample policy shown in figure 4.43, policy statements are defined in terms of known domains which are assumed to contain references to the agents to which the policy applies.

While it would appear that Ponder is quite expressive as regards policy expression for mobile agents, both of these systems assume that the incoming mobile agents can easily be mapped to pre-defined domains, about which policies are specified. Neither Knottenbelt [94] nor Montanari, Tonti and Stefanelli [10, 30, 105] tackle the issue of how trust is established between the incoming agent and the host platform.

4.5.2 Scenario 1

In this section we explore how scenario 1, as described in section 3.3, may be implemented using the Ponder policy specification framework.

4.5.2.1 Policy statements

In order to implement scenario 1, as illustrated in figure 3.1, three requirements must be met with respect to policy statement expression — requirements 1, 2 and 3, as defined in section 4.2.1.

Let us assume that executables which are members of the /executables/trustedMobileExecutables domain are executables which have been signed by an (agent) code author, which in turn has been certified by the mobile device's manufacturer, which has been authorised to do so. Alternatively, we may assume that executables which are members of the /executables/trustedMobileExecutables domain are mobile agents which have been signed by an agent code author and an agent creator, which in turn have been certified by the mobile device's manufacturer, which has been delegated this certification authority. Ponder then enables us to define the execution permissions of these trusted mobile executables (i.e. executables which have been explicitly added to the /executables/trustedMobileExecutablesdomain). The policy statement defined in figure 4.44 authorises a member of the /executables/trustedMobileExecutables domain to perform any action on any target within the system.

```
Inst auth+   {
       subject      s = /executables/trustedMobileExecutables;
       target       /;
       action       *;
    }
```

Figure 4.44: Scenario 1 — Policy statement

Alternatively, a more fine-grained set of policies may be defined for the members of the /executables/trustedMobileExecutablesdomain. For example, the policy defined in figure 4.45 authorises members of the /executables/ trustedMobileExecutables subject domain to read files which are members of the target object domain /files/PublicFiles.

```
Inst auth+   {
       subject      s = /executables/trustedMobileExecutables;
       target       f = /files/PublicFiles;
       action       f.read();
    }
```

Figure 4.45: Scenario 1 — Policy statement

The obligation policy defined in figure 4.46 triggers a system component to terminate an executable which is a member of the target object domain /executables/trustedMobileExecutables when the CPU usage exceeds 90.

```
inst oblig  {
        on CPU(load,90);
        subject s = System;
        target t = /executables/trustedMobileExecutables;
        do t.kill();
    }
```

Figure 4.46: Scenario 1 — Policy statement

Every permission associated with executable execution, i.e. members of the /executables/trustedMobileExecutables domain, may be defined in this way using positive and negative authorisation policies, information filtering policies, delegation policies, obligation policies and refrain policies.

Ponder also facilitates the use of roles or groups in order to simplify policy specification. Groups may be used to group policies which refer to the same target, relate to the same department or apply to the same application. Roles,

on the other hand, provide a mechanism for grouping policies with a common subject. The policies defined in figure 4.45 and 4.46 above, may be grouped as illustrated in figure 4.47. Indeed all policies relating to executable execution, or the domain /executables/trustedMobileExecutables , once defined, may be grouped in this way.

```
inst group TrustedMobileExecutableExecution {

Inst auth+   {
       subject      s = /executables/trustedMobileExecutables;
       target       f = /files/PublicFiles;
       action       f.read();
    }

inst oblig  {
        on CPU(load,90);
        subject s = System ;
        target t = /executables/trustedMobileExecutables;
         do t.kill();
    }
 }
```

Figure 4.47: Scenario 1 — Policy statement

As regards the requirements listed above, it is not possible to delegate the authority for the executable execution authorisation to the device manufacturer, so that he can be permitted to certify (agent) code authors or agent creators, whose identities are unknown at the time of policy specification. Neither can the second or third requirements be met completely. Ponder can be used to specify policies pertaining to members of the /executables/trustedMobileExecutables domain, but polices cannot be specified in terms of subject or target attributes. What is necessary, therefore, is a method by which unknown incoming executables can be mapped to the /executables/trustedMobileExecutables domain.

In order to do this, a separate trust management or trust establishment mechanism is required. KeyNote, as described in section 4.4, could be utilised to achieve this. In order to implement scenario 1, as illustrated in figure 3.1, using both KeyNote and Ponder, we must initially specify the KeyNote policy state-

ment shown in figure 4.48. This policy specifies that the device manufacturer is trusted to delegate authority related to the app_domain "mobile executable execution".

```
KeyNote-Version: 2
Local-Constants: Device manufacturer = "RSA:12345"
      # public key of device manufacturer
Authorizer: "POLICY"
Licensees: Device manufacturer
Conditions: (app_domain == "mobile executable execution");
Signature: "RSA:787878"
```

Figure 4.48: Scenario 1 — TE policy assertion

Following the generation of the policy statement defined in figure 4.48, the device manufacturer is free to generate signed assertion/credentials for (agent) code authors which he trusts to generate safe executables, as illustrated in figure 4.49.

```
KeyNote-Version: 2
Authorizer: "RSA: 12345"
           #the public key of the device manufacturer
Licensees: "RSA: 654873"  # a trusted (agent) code author
Conditions: (app_domain == "mobile executable execution") &&
            (command == "run")  &&
            (ACTION_AUTHORISERS == RSA:654873)
                -> "/executables/trustedMobileExecutables";
Signature: "RSA-SHA1: 1234534"
           #signature of device manufacturer
```

Figure 4.49: Scenario 1 — Signed assertion/credential

If a KeyNote query consisting of a principal identifer, RSA:65873; the action attribute set illustrated in figure 4.50; the ordered compliance value set {discard, /executables/trustedMobileExecutables }; the policy assertion defined in figure 4.48; and the signed assertion/credential defined in figure 4.49; is input into the KeyNote compliance checker, then {/ executables/trustedMobileExecutables } would be output.

```
_ACTION_AUTHORIZERS = "RSA:654873"
app_domain = "mobile executable execution"
command == "run"
```

Figure 4.50: Scenario 1 — Action attribute set

Once this decision has been output, the KeyNote policy enforcement component could add the executable to the /executables/trustedMobileExecutables domain. While executing, the authorisation and obligation policies defined in figure 4.47, and indeed all policies relating /executables/trustedMobileExecutables domain, could then be applied to the executable.

Alternatively a trust establishment framework could be deployed, such as that proposed by Herzberg et al. [68]. The trust policy language ((D)TPL) was designed in order to define the mapping of strangers to roles based on certificates issued by third parties, such that a role-based access control mechanism can be extended rather than completely replaced. It could be utilised in tandem with the Ponder specification language in order to restrict the assignment of users to domains. TPL focuses, however, on the mapping of entities, identified by their public keys, to roles. There are components of this trust establishment framework which may be utilised in our application in order to map incoming unknown executables to roles, and it will be revisited in chapter 5.

4.5.2.2 The PAP

In order to implement scenario 1, one requirement must be met with respect to the PAP — requirement 1, as defined in section 4.2.5.

As stated above, a set of management tools are provided to support the deployment of a Ponder policy specification framework.

- A domain browser allows the navigation of objects within the domain

server. External tools may be invoked from within the browser tool or, alternatively, other tools may interface with the domain browser through an invocation interface implemented by the domain browser.

- A policy editor enables policy and role creation.
- A policy compiler compiles specified policies, after which they are stored by the domain service.
- A management console is used to distribute policies to their enforcement components. This management console tool can interface with the domain browser to select policies and enforcement components from the directory. This management tool may also interact with the policy compiler in order to dynamically instantiate policies.
- A user-role management tool enables the management of user-roles.
- A GUI component provides a main console for accessing individual tools.
- A configuration manager tool enables the configuration of all Ponder tools.

The domain service manages a hierarchy of domain objects. Each domain object holds references to the policy objects that currently apply to that domain.

4.5.2.3 PDP

In order to implement scenario 1, one requirement must be met with respect to the PDP — requirement 1, as defined in section 4.2.9.

An access controller is defined as an agent which receives an authorisation request, and makes a policy decision with regard to authorisation policies. It also decides whether filters should be applied to the action call parameters or the returned values. One access controller exists for each target object. A

policy management component is defined as an agent which decides, based on events, whether obligation and refrain policies apply. One policy management component is defined for each object.

4.5.2.4 PEP

In order to implement scenario 1, one requirement must be met with respect to the PDP — requirement 1 as defined in section 4.2.10.

An access controller is defined as an agent which also enforces its policy decision with regard to authorisation policies. In conjunction with this, the access controller applies any authorisation filters to the returned values. A policy management component is defined as an agent which enforces obligation and refrain policies.

4.5.3 Scenarios 2 – 6

We do not investigate Ponder in the context of the five remaining scenarios. The significant issues unearthed in the attempted implementation of scenario 1 also arise with the other scenarios. Section 4.5.2 above demonstrates the way in which the Ponder policy specification framework may be utilised in each of the scenarios. For example, as was the case in figures 4.44 - 4.47, policies pertaining to executables which are members of, for example, domains named /executables/ deviceManufacturerExectuables, /executables/networkOperatorExectuables, /executables/TTPExectuables and /executables/restrictedExectuables, could be defined for scenario 2. The problem of ensuring that incoming executables become members of the correct domain, however, remains one that requires the use of an additional trust establishment mechanism.

4.5.4 Conclusions

PIP functionality is not supported by Ponder because of the assumption that subject or target domain membership is explicit. AP functionality is not supported because it is assumed that ARs are authenticated independently of the Ponder policy framework. Because of this, there is no supported means to collect AR information. TEM functionality is also not supported. Neither the specification of attribute certificates nor authentication evidence is covered by the Ponder policy specification framework. Finally, because policy decision and policy enforcement is implemented by the same component, policy compliance value specification is not required.

Management tools enable the specification and administration of policies, and provide PAP functionality. PDP and PEP functionality can be provided by access controllers and policy management components defined within the Ponder policy specification framework. A number of policy types can be specified using Ponder, namely positive and negative authorisation policies, filtering policies, delegation policies, obligation policies and refrain policies. A set of conditions under which each policy is valid may also be specified. The syntax and semantics of the Ponder policy specification language are also reasonably easy to understand. As regards specifying policies which define the permissions of an executable which is a member of a particular domain, Ponder is an expressive and useful language.

In summary, because policy statements are defined in terms of subject and target domains, to which objects must be explicitly added, additional trust establishment and attribute certificate expression mechanisms must be provided in order to implement scenarios 1 – 6.

4.6 SAML

The security assertions markup language is an XML-based framework for exchanging security information over the Internet. It is standardised by OASIS[4], the organisation for the advancement of structured information standards, which develops interoperable specifications based on XML. Version 1 of the standard, and the version referred to throughout the remainder of this section, was published in November 2002 [112, 113]. Since this analysis was performed, versions 1.1 [112, 113] and 2.0 [116–118] have been published (in September 2003 and March 2005, respectively). However, the fact that newer versions of SAML exist does not change our conclusions. The requirements which can be met using SAML v1.0, as highlighted in this section, can also be met using later versions of the specifications, as none of the functionality we require has been deprecated.

SAML defines the data format for three types of assertion.

- Authentication assertions assert that the issuer has authenticated a specific subject using a particular mechanism at a particular point in time.
- Attribute assertions assert that a subject has specific attributes.
- Authorisation assertions assert that a subject has been granted or denied access to particular resources.

The SAML v1.0 Assertions and Protocols specification [112] defines "the syntax and semantics for XML encoded SAML assertions, protocol requests and protocols responses".

Every SAML assertion contains an <Assertion> element which contains the following mandatory attributes:

[4] www.oasis-open.org

- MajorVersion — holds the major version of the SAML assertion schema. This number is incremented if the SAML assertion schema is changed in ways that are not compatible with previous versions.
- MinorVersion — holds the minor version of the SAML assertion schema. This number is incremented if minor changes are made to the SAML assertion schema. These changes should not effect the compatibility of the schema with older schema versions with the same MajorVersion number.
- AssertionID — holds the identifier of the assertion.
- Issuer — holds the name of the issuer provided as a string.
- IssueInstant — holds the time of assertion issue.

The <Assertion> element may also contain the following optional elements.

- <Conditions> — defines the conditions under which the assertion is valid. It may contain the following elements and attributes.
    1. NotBefore — holds the earliest time at which the assertion is valid.
    2. NotOnOrAfter — holds the instant at which the assertion expires.
    3. <Condition> — provides an extension point, allowing extension schemas to define new conditions.
    4. <AudienceRestrictionConditions> — specifies that the assertion is addressed to a specific audience.
- <Advice> — contains information which assists assertion processing.
- <ds:Signature> — holds an XML digital signature on the assertion.

One or more of the following elements is also included in the <Assertion> element.

- <Statement> — is an extension point that allows other assertion-based applications to reuse the SAML assertion framework. It contains a statement defined in an extension schema.
- <Subjectstatement> — is also an extension point that allows other assertion-based applications to reuse the SAML assertion framework. It contains a <Subject> element, in which an issuer describes the subject of an assertion.
- <AuthenticationStatement> — contains an authentication statement.
- <AuthorisationDecisionStatement> — contains an authorisation statement.
- <AttributeStatement> — contains an attribute statement.

The <AuthenticationStatement> element contains a statement supplied by the issuer about how a particular subject was authenticated, and at what time. It contains the following attributes and elements.

- AuthenticationMethod — holds a URI that specifies the type of authentication that was completed.
- AuthenticationInstant — holds the time at which the authentication took place.
- <SubjectLocality> — holds the IP address and the DNS domain name of the machine on which the subject was authenticated.
- <AuthorityBinding> — holds additional information about the subject of the statement.

An <AuthorisationDecisionStatement> element specifies the decision of the

issuer to allow or deny a particular subject access to a specified resource. It is comprised of the following attributes and elements.

- Resource — holds a URI reference which specifies the resource in question.
- Decision — holds the decision of the issuer, i.e. one of the values permit, deny or indeterminate.
- <Action> — holds the set of actions authorised on the specified resource.
- <Evidence> — holds the set of assertions on which the decision was based.

An <AttributeStatement> element contains a statement made by the issuer regarding the attributes a specified subject possesses. It is comprised of one of more <Attribute> elements, which in turn are comprised of the following elements.

- <AttributeDesignator> — identifies the attribute name and the namespace in which the attribute name should be interpreted.
- <AttributeValue> — holds the value of the attribute.

A request-response protocol is also defined in [112], containing SAML request and SAML response messages. A SAML request message allows an entity to request assertions from a SAML authority. The following attributes and elements are associated with SAML request messages. The following four attributes are mandatory.

- RequestID — an identifier for the SAML request.
- MajorVersion — the major version of the SAML request.
- MinorVersion — the minor version of the SAML request.

- IssueInstant — the time at which the request was issued.

The inclusion of the following elements in a request message is optional.

- <RespondWith> — holds the statement types acceptable to the requester. Any number of <RespondWith> elements may be contained within a request message.
- <ds:Signature> — holds an XML signature that authenticates the request.
- <Request> — specifies the SAML request. One of the following elements is also contained in a request message.
  - <Query> — an extension point that allows new SAML queries to be defined.
  - <SubjectQuery> — an extension point that allows new SAML queries that specify a single SAML subject.
  - <AuthenticationQuery> — holds a query for authentication statements for a particular subject.
  - <AttributeQuery> — holds a query for attribute statements for a particular subject.
  - <AuthorisationDecisionQuery> — holds a query for an authorisation decision for a particular subject.
  - <AssertionIDReference> — requests an assertion by reference to its assertion identifier.
  - <AssertionArtifact> — requests assertions by supplying an assertion artifact that represents it.

A SAML response message can then be used by the SAML authority to return the required assertions to the requester. A SAML response message is defined by the following attributes and elements. The following five attributes are required.

- ResponseID — an identifier for the SAML response.
- InResponseTo — a reference to the identifier of the request to which the response corresponds.
- MajorVersion — the major version of the SAML response.
- MinorVersion — the minor version of the SAML response.
- IssueInstant — the time the response was issued at.

The inclusion of the following two elements in a request message is optional.

- <Recipient> — the intended recipient of the response.
- <ds:Signature> — an XML signature on the response.
- <Response> — specifies the status of the SAML request and contains a (possibly empty) list of assertions. It is required to contain a <Status> element, and may contain any number of assertions.
  - <Status> — the status of the corresponding request.
  - <Assertion> — specifies an assertion by value.

Transferring SAML assertions may be completed using a variety of protocols, used either independently of, or in conjunction with, the SAML request and SAML response messages, described above. The Bindings and Profiles specification [113] describes how widely deployed protocols may be used to transport

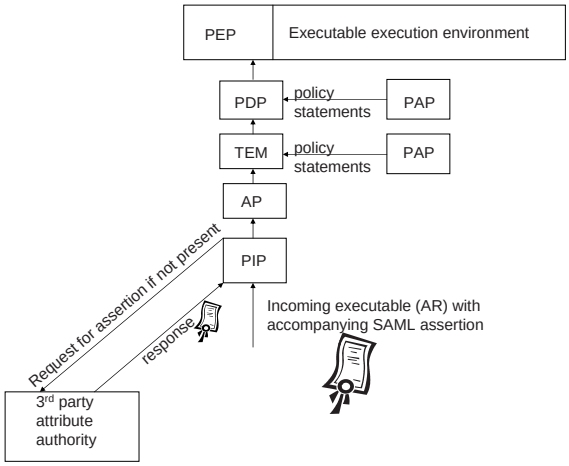

Figure 4.51: SAML

SAML assertions. For example, the SAML SOAP binding defines how SAML protocol messages can be communicated within SOAP messages, whilst the HTTP redirect binding defines how to pass protocol messages through HTTP redirection [113].

We examine the SAML domain model, as illustrated in figure 4.51, in order to determine which of the functional components, as described in section 4.2, are provided, and whether policy statement, attribute certificate, ordered compliance value definition and authentication evidence expression is supported.

- As stated above, [112] defines the syntax and processing semantics of assertions made about a subject by a system entity and the structure of SAML assertions. Signed attribute assertions can be used to specify attribute certificates, as described in section 4.2.

- As stated above, [112] also defines the structure of an associated set of request-response protocols, enabling the required information pertaining to an AR to be collected. These protocols meet some of the criteria for the PIP, as defined in section 4.2.

- Signed SAML assertions can be generated by SAML attribute authorities, where the XML signatures generated on these assertions acts as authentication evidence for signers of assertions.

Neither policy statement expression nor policy administration is covered by the SAML specifications. The same is true of trust establishment. SAML does not provide for trust negotiation between parties. Policy decision making and point enforcement are also not within the scope of the specifications. A set of compliance values (either permit, deny or indeterminate) can be described using an authorisation decision assertion and sent from the PDP to the PEP. However, it is likely that the compliance value set will be dependent on the policy specification language and the TEM, PDP and PEP deployed. Authentication point functionality is also beyond the scope of the SAML framework, although some forms authentication are covered. In sections 4.6.1 to 4.6.4 we examine whether the requirements for attribute certificates, authentication evidence, the AP and the PIP, as described in section 4.2, can be met using SAML.

4.6.1 Scenario 1

In this section we will explore how scenario 1 may be implemented using SAML.

4.6.1.1 Attribute certificates

In order to support scenario 1, as illustrated in figure 3.1, three requirements must be met with respect to attribute certificate expression — requirements 1, 2, and 3, as defined in section 4.2.2.

The SAML attribute assertion described in figure 4.52 shows the type of attribute certificate that must be generated for a code author by a mobile device's manufacturer if they are to be permitted to have their executables executed

on the mobile host, as described in the scenario 1 architectural model. This digitally signed statement asserts that the subject 'CodeAuthor1', which is in possession of the associated public key, has a 'CodeAuthorStatus' of 'Trusted'. The assertion is valid for the period between the times listed in the NotBefore and NotOnOrAfter attribute fields.

The subject of the assertion, 'CodeAuthor1' is defined in the <Subject> element. The <NameIdentifier> element contained within the <Subject> element defines the format in which the subject's name will appear. The assertion shown in figure 4.52 indicates that the content of the NameIdentifier field is in the form specified for the content of <ds:X509SubjectName>. This is indicated by the URI 'urn:oasis:names:tc:SAML:1.0:nameid-format:X509SubjectName' which is defined by SAML. The NameQualifier attribute allows multiple username issuing authorities to work without causing any duplication [139]. The subject's name then appears as follows 'CN=CodeAuthor1, OU=MOBILECODEGROUP, O=MOBILEEXECUTABLESLTD'. The <SubjectConfirmation> contains information that allows the subject of the assertion to be authenticated. The first element in the <SubjectConfirmation> element is <ConfirmationMethod>. The <ConfirmationMethod> method specifies the method which can be used by the host to confirm the relationship between the code author who signed the incoming executable and the code author who has been certified in the attribute assertion. 'urn:oasis:names:tc:SAML:1.0:cm:holder-of-key' is contained in the <ConfirmationMethod> element. This implies that the subject must prove that they are the owner of a specified public key in order to confirm their identity. The key is either included in, or referenced by, the second element in the <SubjectConfirmation> element, <ds:KeyInfo>. By verifying the signature of the code author on the incoming executable, the host can check that the identity of the code author is the subject of the attribute assertion.

The <Attribute> element contains two elements <AttributeDesignator> and <AttributeValue>. The <AttributeDesignator> element has two attributes, namely AttributeName and AttributeNamespace. The AttributeName attribute is used to specify the name of the security attribute being asserted. In the attribute assertion shown in figure 4.52, the attributeName is 'CodeAuthorStatus'. The AttributeNamespace attribute of the <AttributeDesignator> element, identifies the attribute namespace, 'http://www.mobileexecutable-authorisation-vocab.org/attribute' — a fictitious namespace, to which the security attribute belongs. Inclusion of this AttributeNamespace URI (which does not hold an XML namespace) allows organisations to have security attribute names without the fear of any collision between names defined by different naming services [139]. The attribute value, 'Trusted', is specified in the <AttributeValue> element. A similar assertion may also be generated for an agent creator by the device manufacturer if required.

We thus conclude that all the attribute certificate requirements for scenario 1 can be met using SAML.

4.6.1.2 Authentication evidence

In order to implement scenario 1 three requirements must be met with respect to authentication evidence — requirements 1, 2, and 4, as defined in section 4.2.3.

The XML signature of the device manufacturer on the attribute assertion shown in figure 4.52 serves to authenticate the source of the assertion, thereby fulfilling requirement 2.

4.6.1.3 The PIP

In order to support scenario 1, the PIP implemented on the mobile device must fulfil one requirement — requirement 1 as defined in section 4.2.6.

```
<?xml version="1.0" encoding="utf-8"?>
<Assertion
 xmlns="urn:oasis:names:tc:SAML:1.0:assertion"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 MajorVersion = "1"
 MinorVersion = "0"
 AssertionID = "6738467-47378dj-hu234832"
 Issuer = "DeviceManufacturer"
 IssueInstant = "2001-05-31T13: 20:00-05:00"
 <Conditions
   Notbefore = "2001-05-31T13: 20:00-05:00"
   NotOnOrAfter = "2002-10-31T13: 20:00-05:00">
 <AttributeStatement>
  <Subject>
   <NameIdentifier
     Format="urn:oasis:names:tc:SAML:1.0:nameid-format:X509SubjectName"
     NameQualifier="https://www.mobilecode.mobileececutables.org/saml">
     CN=CodeAuthor1,OU=MOBILECODEGROUP,O=MOBILEEXECUTABLESLTD
   </NameIdentifier>
   <SubjectConfirmation>
     <ConfirmationMethod>
      urn:oasis:names:tc:SAML:1.0:cm:holder-of-key
     </ConfirmationMethod>
     <ds:KeyInfo>
       <ds:KeyValue>...</ds:KeyValue>
     </ds:KeyInfo>
     </SubjectConfirmation>
   </Subject>
   <Attribute>
    <AttributeDesignator
     AttributeName = "CodeAuthorStatus"
     AttributeNamespace = "http://www.mobileexecutable-authorisation-
vocab.org/attribute"/>
     <AttributeValue>Trusted</AttributeValue>
   </Attribute>
 </AttributeStatement>
 <ds:Signature>...</ds:Signature>
</Assertion>
```

Figure 4.52: Scenario 1 — Attribute assertion

If the required attribute assertion(s) are not received with the incoming executable, they can be requested by the end host from the device manufacturer. Figures 4.53 and 4.54 contain examples of request and response messages which may be sent between the mobile host on which an incoming executable wishes to execute, and the device manufacturer. The request message, shown in figure 4.53, dictates that the responder must only <Respondwith> assertions containing attribute statements. The <AttributeQuery> element identifies the subject with which all returned assertions must be associated.

The response message, shown in figure 4.54, holds the identifier of the request message which this message is <InResponseTo>. The <Status> element has been assigned a <StatusCode> value of Success, which indicates that the request succeeded. The assertion illustrated in figure 4.52 would be contained within the <Assertion> element. Both the request and the response messages may be signed if required.

```
<?xml version="1.0" encoding="utf-8"?>
<samlp:Request
 xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
 xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 RequestID = "567.14.234.20.90123456"
 MajorVersion = "1"
 MinorVersion = "0"
 IssueInstant = "2001-06-31T13:20:00-05:00">
 <RespondWith>saml:AttributeStatement</RespondWith>
   <AttributeQuery>
    <saml:Subject>
     <NameIdentifier
     Format="urn:oasis:names:tc:SAML:1.0:nameid-format:X509SubjectName"
     NameQualifier="https://www.mobilecode.mobileececutables.org/saml">
     CN=CodeAuthor1,OU=MOBILECODEGROUP,O=MOBILEEXECUTABLESLTD
     </saml:Subject>
    </AttributeQuery>
</samlp:Request>
```

Figure 4.53: Scenario 1 — Attribute assertion request

```
<?xml version="1.0" encoding="utf-8"?>
<samlp:Response
    xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
    xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    ResponseID = "333.15.774.21.9012999"
    InResponseTo = "567.14.234.20.90123456"
    MajorVersion = "1"
    MinorVersion = "0"
    IssueInstant = "2003-06-31T13:20:00-05:00"
    Recipient = "mobilehost4">
    <Status
       <StatusCode = "Success"/>
     </Status>
    <saml:Assertion> ... </saml:Assertion>
</samlp:Response>
```

Figure 4.54: Scenario 1 — Attribute assertion response

4.6.1.4 The AP

In order to implement scenario 1, three requirements must be met with respect to the authentication point — requirements 2, 1, and 4, as defined in section 4.2.7.

A SAML processor can verify the signature of the device manufacturer on the incoming attribute assertion, thereby meeting requirement 2. [112] defines a set of constraints on the XML syntax for signing data "so that SAML processors do not have to deal with the full generality of XML signature processing".

4.6.2 Scenario 2

In this section we will explore how scenario 2 may be implemented using SAML. In order to implement scenario 2 there are no additional requirements to be met with respect to attribute certificates, authentication evidence, the AP, or the PIP.

4.6.2.1 Attribute certificates

The attribute assertion described in figure 4.55 shows the type of attribute certificate that must be generated for a code author by a mobile device's manufacturer if they are to be permitted to have their executables executed in the 'TTP domain' on the mobile host rather than the 'restricted domain'. This digitally signed statement asserts that the subject CodeAuthor1, who is in possession of the associated public key, is 'PermittedExecution' (permitted to have his signed executables executed) in the 'TTPDomain'. The assertion is valid for the period between the times listed in the NotBefore and NotOnOrAfter fields.

```
<?xml version="1.0" encoding="utf-8"?>
<Assertion
 xmlns="urn:oasis:names:tc:SAML:1.0:assertion"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 MajorVersion = "1"
 MinorVersion = "0"
 AssertionID = "6738467-47378dj-hu234832"
 Issuer = "DeviceManufacturer"
 IssueInstant = "2001-05-31T13: 20:00-05:00"
 <Conditions
   Notbefore = "2001-05-31T13: 20:00-05:00"
   NotOnOrAfter = "2002-10-31T13: 20:00-05:00">
 <AttributeStatement>
  <Subject>
   <NameIdentifier
     Format="urn:oasis:names:tc:SAML:1.0:nameid-format:X509SubjectName"
     NameQualifier="https://www.mobilecode.mobileececutables.org/saml">
     CN=CodeAuthor1,OU=MOBILECODEGROUP,O=MOBILEEXECUTABLESLTD
   </NameIdentifier>
   <SubjectConfirmation>
     <ConfirmationMethod>
      urn:oasis:names:tc:SAML:1.0:cm:holder-of-key
     </ConfirmationMethod>
     <ds:KeyInfo>
       <ds:KeyValue>...</ds:KeyValue>
     </ds:KeyInfo>
     </SubjectConfirmation>
   </Subject>
   <Attribute>
    <AttributeDesignator
     AttributeName = "PermittedExecution"
     AttributeNamespace = "http://www.mobileexecutable-authorisation-
vocab.org/attribute"/>
     <AttributeValue>TTPDomain</AttributeValue>
   </Attribute>
 </AttributeStatement>
 <ds:Signature>...</ds:Signature>
</Assertion>
```

Figure 4.55: Scenario 2 — Attribute assertion

4.6.2.2 Authentication evidence

There are no additional authentication evidence requirements defined for scenario 2.

4.6.2.3 The PIP

There are no additional policy information point requirements defined for scenario 2. If the required attribute assertion is not received with the incoming executable, an attribute assertion of the type shown in figure 4.55 can be requested

by the end host from the device manufacturer using the request message defined in figure 4.53 and returned by the device manufacturer in a response message similar to that shown in figure 4.54.

4.6.2.4 The AP

There are no additional authentication point requirements defined for scenario 2.

4.6.3 Scenarios 3 and 4

In this section we will explore how scenarios 3 and 4 may be implemented using SAML.

4.6.3.1 Attribute certificates

In order to implement scenarios 3 and 4 two additional attribute certificate requirements must be met — requirements 5 and 6, as defined in section 4.2.2.

The attribute assertion shown in figure 4.56 is an example of an attribute certificate that a TTP can generate for an executable once it has been tested, as required in the architectural model described in figure 3.4. The digitally signed statement in figure 4.56 asserts that Test1 has been completed and Test2 has not been completed on the subject, i.e. the incoming executable whose hash matches that held in the <SubjectConfirmationData> element. The assertion is valid for the period between the times listed in the NotBefore and NotOnOrAfter fields.

The <SubjectConfirmation> element specifies a subject by supplying data that allows the subject to be authenticated. In this assertion the <SubjectConfirmation> element contains two elements, <ConfirmationMethod> and <SubjectConfirmationData>. The <ConfirmationMethod> element holds a URI reference that identifies a protocol to be used to authenticate the subject. Subject

confirmation methods are defined in the SAML bindings and profiles specification, but none that satisfy our requirement. We require that the incoming executable is hashed and compared to the hash value held in the <SubjectConfirmationData> element. This new method, which we have identified by the fictitious URI, 'urn:org:names:tc:SAMLExtn:1.0:cm:hash-of-executable', in the assertion shown in figure 4.56, can only be used once a new profile has been defined for it, or, alternatively, if the parties involved agree on the use of this method in advance.

```
<?xml version="1.0" encoding="utf-8"?>
<Assertion
 xmlns="urn:oasis:names:tc:SAML:1.0:assertion"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 MajorVersion = "1"
 MinorVersion = "0"
 AssertionID = "6738467-47378dj-hu234832"
 Issuer = "TTP1"
 IssueInstant = "2001-05-31T13: 20:00-05:00"
 <Conditions
   Notbefore = "2001-05-31T13: 20:00-05:00"
   NotOnOrAfter = "2002-10-31T13: 20:00-05:00">
 <AttributeStatement>
  <Subject>
   <SubjectConfirmation>
     <ConfirmationMethod>
      urn:org:names:tc:SAMLExtn:1.0:cm:hash-of-executable
     </ConfirmationMethod>
     <SubjectConfirmationData>....</SubjectConfirmationData>
     </SubjectConfirmation>
   </Subject>
  <Attribute>
    <AttributeDesignator
     AttributeName = "Test1"
     AttributeNamespace = "http://www.mobileexecutable-authorisation-
vocab.org/attribute"/>
     <AttributeValue>completed</AttributeValue>
   </Attribute>
    <Attribute>
    <AttributeDesignator
     AttributeName = "Test2"
     AttributeNamespace = "http://www.mobileexecutable-authorisation-
vocab.org/attribute"/>
     <AttributeValue>NotCompleted</AttributeValue>
   </Attribute>
 </AttributeStatement>
 <ds:Signature>...</ds:Signature>
</Assertion>
```

Figure 4.56: Scenario 3 — Attribute assertion

The attribute assertion shown in figure 4.57 is an example of the type of attribute certificate that a domain server must generate for an executable once its security relevant properties have been verified, as required in the architectural model described in figure 3.5. This digitally signed statement asserts that 'Attribute1' is equal to '50' and that 'Attribute2' is 'true' for the incoming executable whose hash matches that held in the <SubjectConfirmationData> element. The assertion is valid for the period between the times listed in the NotBefore and NotOnOrAfter fields.

```
<?xml version="1.0" encoding="utf-8"?>
<Assertion
 xmlns="urn:oasis:names:tc:SAML:1.0:assertion"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 MajorVersion = "1"
 MinorVersion = "0"
 AssertionID = "6738467-47378dj-hu234832"
 Issuer = "DomainSever1"
 IssueInstant = "2001-05-31T13: 20:00-05:00"
 <Conditions
   Notbefore = "2001-05-31T13: 20:00-05:00"
   NotOnOrAfter = "2002-10-31T13: 20:00-05:00">
 <AttributeStatement>
  <Subject>
   <SubjectConfirmation>
     <ConfirmationMethod>
      urn:org:names:tc:SAMLExtn:1.0:cm:hash-of-executable
     </ConfirmationMethod>
     <SubjectConfirmationData>....</SubjectConfirmationData>
     </SubjectConfirmation>
   </Subject>
   <Attribute>
    <AttributeDesignator
     AttributeName = "Attribute1"
     AttributeNamespace = "http://www.mobileexecutable-authorisation-
vocab.org/attribute"/>
     <AttributeValue>50</AttributeValue>
   </Attribute>
    <Attribute>
    <AttributeDesignator
     AttributeName = "Attribute2"
     AttributeNamespace = "http://www.mobileexecutable-authorisation-
vocab.org/attribute"/>
     <AttributeValue>true</AttributeValue>
   </Attribute>
 </AttributeStatement>
 <ds:Signature>...</ds:Signature>
</Assertion>
```

Figure 4.57: Scenario 4 — Attribute assertion

4.6.3.2 Authentication evidence

In order to implement scenarios 3 and 4 two additional attribute certificate requirements must be met — requirements 3 and 4, as defined in section 4.2.3.

The XML signature serves as authentication evidence for the TTP or domain server which has signed the SAML attribute assertion. The authentication evidence for the incoming executable, i.e. the hash of the executable, can be contained in the <SubjectConfirmationData> element of the attribute assertion.

4.6.3.3 The PIP

There are no additional policy information point requirements defined for scenarios 3 and 4. If the required attribute assertion is not received with the incoming executable, an attribute assertion of the type shown in figure 4.57 can be requested by the end host from the device manufacturer using a message of the form given in figure 4.58 and returned by the device manufacturer in a response message similar to that shown in figure 4.59.

```
<?xml version="1.0" encoding="utf-8"?>
<samlp:Request
 xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
 xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 RequestID = "567.14.234.20.90123456"
 MajorVersion = "1"
 MinorVersion = "0"
 IssueInstant = "2001-06-31T13:20:00-05:00">
 <RespondWith>saml:AttributeStatement</RespondWith>
   <AttributeQuery>
     </saml:Subject>
       <SubjectConfirmation>
         <ConfirmationMethod>
           urn:org:names:tc:SAMLExtn:1.0:cm:hash-of-executable
         </ConfirmationMethod>
         <SubjectConfirmationData>....</SubjectConfirmationData>
       </SubjectConfirmation>
     </saml:Subject>
   </AttributeQuery>
</samlp:Request>
```

Figure 4.58: Scenarios 3 and 4 — Attribute assertion request

```
<?xml version="1.0" encoding="utf-8"?>
<samlp:Response
    xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
    xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    ResponseID = "333.15.774.21.9012999"
    InResponseTo = "567.14.234.20.90123456"
    MajorVersion = "1"
    MinorVersion = "0"
    IssueInstant = "2003-06-31T13:20:00-05:00"
    Recipient = "mobilehost4">
    <Status
       <StatusCode = "Success"/>
     </Status>
    <saml:Assertion> ... </saml:Assertion>
</samlp:Response>
```

Figure 4.59: Scenarios 3 and 4 — Attribute assertion response

4.6.3.4 The AP

In order to implement scenarios 3 and 4 two additional authentication point requirements must be met — requirements 3 and 4, as defined in section 4.2.2.

A SAML processor can verify the signature on the incoming attribute assertion, thereby meeting requirement 3. [112] defines a set of constraints on the XML syntax for signing data so that SAML processors do not have to deal with the full generality of XML signature processing. In order for a SAML processor to authenticate the incoming executable, i.e. be able to process the <SubjectConfirmationData>, the TTP or domain server and the devices with which it is communicating must have agreed on the stated <ConfirmationMethod>. Alternatively, a new profile which described this authentication method may be specified.

4.6.4 Scenarios 5 and 6

In scenarios 5 and 6 the expression of attribute certificates is not required, nor is PIP functionality. No new requirements need to be met with respect to authentication evidence or by the AP.

4.6.5 Conclusions

SAML meets all of the attribute certificate requirements defined in section 4.2.1. However, if it were to be deployed in a policy-based framework for mobile executable authorisation, two requirements would have to be met. A namespace or namespaces, defining the attribute names utilised within the attribute assertions, would have to be established. In conjunction with this, in order to meet the scenario 3 and 4 requirements, a new <ConfirmationMethod> would have to be defined. This may be achieved through the specification of a new SAML profile, or alternatively, by private agreement between TTPs or domain servers and the devices which they protect.

4.7 Conclusions

In this chapter we have examined a selection of policy and attribute certificate specification languages, namely KeyNote, Ponder and SAML, and explored the functionality of their supporting policy engine components. We have investigated how well the requirements outlined in section 4.2 can be met using KeyNote, Ponder, and SAML, by considering the possible implementation of the six architectural models from chapter 3 using each system.

Table 4.1 summarises the results of the investigation of KeyNote (see section 4.4); a tick denotes that a requirement can be met using KeyNote, a cross denotes that a requirement cannot be met using KeyNote, and a blank denotes that a requirement is not relevant to the specified scenario. The KeyNote assertion expression language is easy to understand, as the language is based on the format of RFC-822 style message headers and the same language and syntax is used to express both policy and attribute assertions. As a result of our analysis a number of issues have been uncovered with respect to use of this language in

our scenarios. Firstly, at stated above, KeyNote is more suited towards the expression of coarse-grained access control policies. Secondly, it is not possible to create an attribute certificate in which attribute(s) are bound to a principal. All KeyNote assertions have an inherent notion of delegation. Thirdly, a problem may arise with respect to limiting delegation. In section 4.4.6, we investigated a mechanism by which delegation may be limited. In conjunction with this, we analysed Nereus, a language which builds upon and extends the functionality of KeyNote. While full details of this language are not available, KeyNote may benefit from re-examination and extension in light of the additional features that may be incorporated into it. This may enable the issues surronding attribute binding and delegation to be overcome.

Table 4.2 summarises the results of the investigation of Ponder (see section 4.5); a tick denotes that a requirement can be met using Ponder, a cross denotes that a requirement cannot be met using Ponder, a bullet denotes that the requirement can be partially met using Ponder, and a blank denotes that a requirement is not relevant to the specified scenario. The Ponder policy specification language is reasonably easy to understand. As a result of our analysis a number of issues have been uncovered with respect to application of this language in our scenarios. Ponder enables the specification of policies pertaining to known entities, i.e. members of pre-defined domains. Once an executable has been made a member of a domain, a wide range of Ponder policies may be expressed in order to define the executable's execution permissions. Therefore, as illustrated in table 4.2, the policy statement requirements can only be partially met. Ponder can only support the six scenarios described in chapter 3 if it is used in conjunction with a trust management mechanism such as KeyNote, or a trust establishment mechanism such as (D)TPL.

Table 4.3 summarises the results of the investigation of SAML (see sec-

Table 4.1: A summary of KeyNote's applicability to the scenarios

Requirements		**Scenario**					
		1	2	3	4	5	6
KeyNote	Policy statement req 1	✓	✓				
	Policy statement req 2	✓					
	Policy statement req 3	✓					
	Policy statement req 4		✓	✓		✓	✓
	Policy statement req 5			✓	✓		
	Policy statement req 6			✓	✓		
	Authentication evidence req 1	✗	✗		✗	✗	✗
	Authentication evidence req 2	✓	✓				
	Authentication evidence req 3			✗	✗		
	Authentication evidence req 4	✗	✗	✗	✗	✗	✗
	Attribute certificate req 1	✓	✓	✓	✓		
	Attribute certificate req 2	✓	✓	✓			
	Attribute certificate req 3	✓					
	Attribute certificate req 4		✓				
	Attribute certificate req 5			✗	✗		
	Attribute certificate req 6			✗	✗		
	Compliance values req 1	✓		✓	✓	✓	✓
	Compliance values req 2		✓	✓			
	PAP req 1	✓	✓	✓	✓	✓	✓
	PIP req 1	✗	✗	✗	✗	✗	✗
	AP req 1	✓	✓				
	AP req 2	✗	✗			✗	✗
	AP req 3			✗	✗		
	AP req 4	✗	✗	✗	✗	✗	✗
	TEM req 1	✓	✓	✓	✓	✓	✓
	PDP req 1	✓	✓	✓	✓	✓	✓
	PEP req 1	✗	✗	✗	✗	✗	✗

tion 4.6); a tick denotes that a requirement can be met using SAML, a cross denotes that a requirement cannot be met using SAML, and a blank denotes that a requirement is not relevant to the specified scenario. SAML is XML-based, and therefore relatively easy to understand. SAML has been designed in order to enable the exchange of security information across the Internet, and it meets all our requirements with respect to attribute certificates. If it is to be deployed in a policy-based framework for mobile executable authorisation two additional issues need to be resolved. A namespace(s) defining the attribute names utilised within the attribute assertions would need to be established. In conjunction with this, a new <ConfirmationMethod> would need to be de-

Table 4.2: A summary of Ponder's applicability to the scenarios

Requirements		**Scenario**					
		1	2	3	4	5	6
Ponder	Policy statement req 1	✗	✗				
	Policy statement req 2	•					
	Policy statement req 3	•					
	Policy statement req 4		•	•		•	•
	Policy statement req 5			✗	✗		
	Policy statement req 6			•	•		
	Authentication evidence req 1	✗	✗		✗	✗	✗
	Authentication evidence req 2	✗	✗				
	Authentication evidence req 3			✗	✗		
	Authentication evidence req 4	✗	✗	✗	✗	✗	✗
	Attribute certificate req 1	✗	✗	✗	✗		
	Attribute certificate req 2	✗	✗	✗			
	Attribute certificate req 3	✗					
	Attribute certificate req 4		✗				
	Attribute certificate req 5			✗	✗		
	Attribute certificate req 6			✗	✗		
	Compliance values req 1	•		•	•	•	•
	Compliance values req 2		•	•			
	PAP req 1	✓	✓	✓	✓	✓	✓
	PIP req 1	✗	✗	✗	✗	✗	✗
	AP req 1	✗	✗				
	AP req 2	✗	✗			✗	✗
	AP req 3			✗	✗		
	AP req 4	✗	✗	✗	✗	✗	✗
	TEM req 1	✗	✗	✗	✗	✗	✗
	PDP req 1	✓	✓	✓	✓	✓	✓
	PEP req 1	✓	✓	✓	✓	✓	✓

fined. This may be achieved through the specification of a new SAML profile, or alternatively, by private agreement between TTPs or domain servers and the associated devices which they protect.

Our conclusions will be used in chapter 5 to enable the most suitable mechanisms to be chosen for use in our policy-based framework, designed to support the authorisation of mobile executables in a mobile environment.

Table 4.3: A summary of SAML's applicability to the scenarios

Requirements		**Scenario**					
		1	2	3	4	5	6
SAML	Policy statement req 1	✗	✗				
	Policy statement req 2	✗					
	Policy statement req 3	✗					
	Policy statement req 4		✗	✗		✗	✗
	Policy statement req 5			✗	✗		
	Policy statement req 6			✗	✗		
	Authentication evidence req 1	✗	✗		✗	✗	✗
	Authentication evidence req 2	✓	✓				
	Authentication evidence req 3			✓	✓		
	Authentication evidence req 4	✗	✗	✗	✗	✗	✗
	Attribute certificate req 1	✓	✓	✓	✓		
	Attribute certificate req 2	✓	✓	✓			
	Attribute certificate req 3	✓					
	Attribute certificate req 4		✓				
	Attribute certificate req 5			✓	✓		
	Attribute certificate req 6			✓	✓		
	Compliance values req 1	✗		✗	✗	✗	✗
	Compliance values req 2		✗	✗			
	PAP req 1	✗	✗	✗	✗	✗	✗
	PIP req 1	✓	✓	✓	✓	✓	✓
	AP req 1	✓	✓				
	AP req 2	✗	✗			✗	✗
	AP req 3			✓	✓		
	AP req 4	✗	✗	✗	✗	✗	✗
	TEM req 1	✗	✗	✗	✗	✗	✗
	PDP req 1	✗	✗	✗	✗	✗	✗
	PEP req 1	✗	✗	✗	✗	✗	✗

Chapter 5

A policy-based authorisation framework

Contents

5.1 Introduction 202
5.2 Requirements 203
5.3 The framework — A high level view 203
5.4 Design decisions 207
5.5 Notation 208
5.6 Assumptions 209
5.7 Trusted domain server activity 211
5.7.1 Evaluating a mobile agent 211
5.7.2 Attribute certificates 216
5.7.3 Authentication evidence 219
5.8 End host activity 219
5.8.1 The PIP 221
5.8.2 The AP 221
5.8.3 The TEM 223
5.8.4 Policy statements 227
5.8.5 The PDP and PEP 227
5.8.6 The PAP 228
5.9 Conclusions 229

This chapter describes a policy-based framework for the authorisation of mobile executables and, more specifically, mobile agents, in a mobile environment.

5.1 Introduction

This chapter describes a policy-based framework which enables incoming executables and, more specifically, incoming mobile agents to be authorised by mobile devices, which may be limited in terms of CPU processing power and data storage facilities. This framework incorporates several of the concepts introduced in chapters 2, 3 and 4. It is dependent on the notion of a trusted domain server, which is responsible for a set of mobile devices. This trusted domain server intercepts and completes a pre-defined set of security checks on incoming executables destined for a mobile device for which it is responsible. Following the completion of these security checks, the trusted domain server generates an attribute certificate for an executable before forwarding both the executable and certificate to the destination host. The incoming executable is then assigned to a domain defined on the mobile device based upon its associated attributes. A set of pre-defined executable permissions are specified for members of each domain. At the time of writing we are unaware of anything in existence with the same scope as the policy-based framework outlined in this chapter.

Section 5.2 outlines the requirements the policy-based authorisation framework is designed to satisfy. In section 5.3 a high-level description of the framework, including its associated components and how they inter-relate, is given. Section 5.4 highlights the fundamental design decisions made when planning the framework. Section 5.5 details the notation used in the specification of the framework, and section 5.6 describes the assumptions upon which the framework is based.

Section 5.7 describes how an incoming executable can be evaluated by a trusted domain server and a trust profile associated with the executable. We also

describe how this trust profile can be expressed in a SAML attribute assertion. Section 5.8 explores how the incoming executable and its associated attribute assertion can be processed on the end host, and how the incoming executable can be assigned a set of execution permissions. In section 5.9 we conclude part I of the thesis.

5.2 Requirements

The requirements for the underlying architecture of a policy-based code and agent authorisation framework for implementation within a mobile environment were described in section 3.9. Here we provide a very brief summary of those requirements that our framework is designed to satisfy.

The framework should make minimal use of the end host's CPU processing power and the end host's storage for authorisation data structures. The underlying architecture should support mechanisms which provide assurances regarding the origin of the executable, executable code quality and the state of an agent. It is required that a policy engine, which is comprised of a PAP, a PIP, an AP, a TEM, a PDP and a PEP is incorporated into each end host. It is also required that policy statements and signed attribute certificates can be specified, stored and processed by the end host.

5.3 The framework — A high level view

We begin by introducing the participant roles involved in the framework, many of which have been described in section 3.2. The relationships between the roles are illustrated in figure 5.1.

- In the context of this work, an incoming *executable* is defined as either mobile code or a mobile agent.

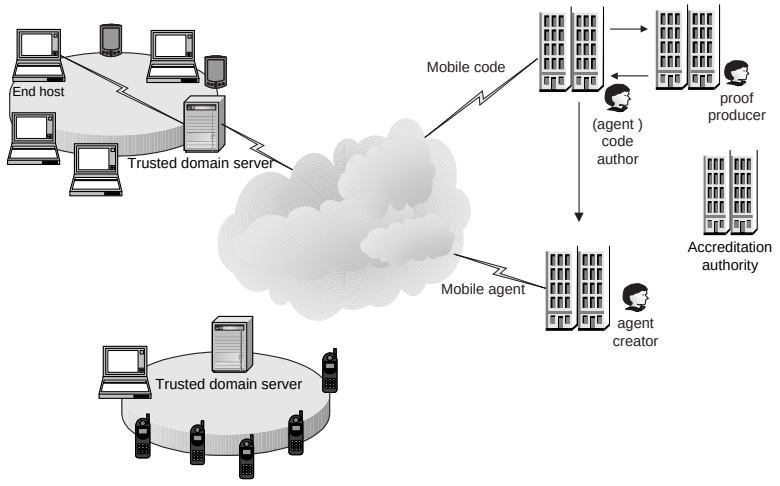

Figure 5.1: Architecture model

- *Mobile code* encompasses programs that can be executed on one or more hosts other than the host from which they originated.

- A *mobile agent* is defined as "an autonomous, reactive, goal-oriented, adaptive, persistent, socially aware software entity, which can actively migrate from host to host", see section 2.1.

- A *malicious executable*, i.e. malicious code or a malicious agent, as defined in section 2.4, is one which attacks the host system on which it is executing. Attacks may include, but are not limited to, unauthorised reading, writing or deletion of sensitive host information, monitoring the host execution environment and exporting information, denial of service attacks, and/or inserting a back door into a system, which may facilitate future security violations.

- An *agent code author* is responsible for the production of agent code.

More generally, we define a *code author* as the entity responsible for the generation of executable code.

- An *agent creator* is responsible for agent creation, i.e. the combination of the program code from the agent code author with data and initial state information. The agent creator may also be responsible for agent distribution.
- A *certification authority* is responsible for the generation of public key certificates.
- An *accreditation authority* is an entity which certifies that an (agent) code author and/or an agent creator can be trusted.
- A *proof producer* is responsible for generating proofs of code. Typically, the code author is also the proof producer for that code.
- A *mobile device* is a mobile host on which an executable may execute.
- A *trusted domain server* is responsible for completing a pre-defined set of security checks on an incoming executable, and for generating an attribute certificate for that executable which reflects the results of these checks.

Mobile code or agent code is produced by an (agent) code author. Once the code has been written, the author may request that a proof producer generates a proof/a set of proofs which allow computer systems to determine automatically the security relevant properties of the code, see section 2.5.1.1. Alternatively, the author may generate these proofs. An author may generate a state appraisal function, *max*, which will calculate, as a function of the agent's current state, the maximum set of permissions to be accorded to an agent running the program, as described in section 2.5.3.1. The author may then digitally sign the code it has produced, in conjunction with any security controls.

If the code produced by the author is agent code, an agent creator then combines it with execution state before it is distributed. If the agent code has an associated state appraisal function, *max*, the agent creator may also generate a state appraisal function *req*, which will calculate, as a function of the agent's current state, the requested set of permissions the sender wants the agent running the program to have, as described in section 2.5.3.1. The agent creator may then digitally sign the agent code, data, and any static state information, in conjunction with any security controls. Following this, the agent can be distributed.

The (agent) code author and/or agent creator may also possess third party accreditation. This accreditation may be based on a variety of factors, including the quality of the code/agents generated by the author or creator, compliance of the author/creator with accepted industry standards for code or agent generation and testing, contracts and/or liability agreements, and/or performance and reputation.

Every mobile device is affiliated with one trusted domain server at any given time. A trusted domain server is responsible for protecting the mobile devices within its domain against malicious executables. A trusted domain server essentially acts as a firewall, which analyses the security properties of executables destined for execution on a host within its protective domain boundary. This analysis may involve verification of a digital signature/digital signatures appended to the incoming executable, the verification of code proofs or the execution of state appraisal functions. Based on this analysis, a trusted domain server will construct a SAML attribute assertion for the executable which communicates its security properties to the end host. Both the executable and its attribute assertion are then forwarded to the destination host.

On receipt of the incoming executable, the mobile host verifies the signature of the trusted domain server on the attribute assertion, and validates the identity of the executable against the identifier contained within the assertion. Based on the security properties of the executable, it is mapped to a domain defined within the mobile device, using a policy specified in (D)TPL. Each domain member is assigned a pre-defined set of permissions using Ponder.

5.4 Design decisions

We chose to deploy the notion of a trusted domain server to assess the security properties of incoming executables because many mobile devices may not have sufficient computational power to perform some of the checks that may be necessary to guarantee the safety of incoming mobile code or mobile agents.

We chose SAML for assertion expression as it enables the transfer of signed assertions describing the attributes of an executable. The number or type of attributes that can be expressed is extensible. It is therefore possible for a trusted domain server to add new attributes under which an executable can be expressed at any stage. The identity of the executable may also be contained within a SAML attribute assertion.

In order to map an incoming executable to a group, i.e. in order to establish trust between the incoming executable and the mobile host, (D)TPL is used. Finally, Ponder is used to specify the actions that members of the domain to which an incoming executable has been assigned, are authorised to perform. We chose to decouple trust establishment and the specification of execution permissions. Integrated solutions to trust establishment and access control are more complex, notably KeyNote as shown in section 4.4. As described in section 4.4.3, the conditions field of the KeyNote policy assertion must be used in

order to specify the actions the subject of the statement is authorised to perform and, indeed, to delegate. In conjunction with this, the conditions field of the KeyNote policy assertion must also specify the conditions under which the actions are authorised, or the conditions which must be imposed on a subject who is delegated authority over the actions described in the assertion. Using an integrated solution may also impede the integration of the framework into existing systems, as it requires all policy statements to be written in the chosen language. Using separate trust establishment and access control mechanisms means that incoming executables may be mapped to groups within the system about which policies have already been defined. It also implies that, instead of using Ponder, as used in the examples in this chapter for illustrative purposes, this framework could be integrated into a platform which uses a different policy language for the expression of role-based or group-based platform security policies.

5.5 Notation

The following notation is used in the specification of the framework:

M	denotes the mobile device.
A	denotes the (agent) code author.
B	denotes the agent creator.
P	denotes a proof producer which is capable of generating proofs of code.
TDS	denotes the trusted domain server with which the mobile device M is affiliated.
C	denotes the certification authority trusted by A, B, and TDS.
N	denotes the accreditation authority trusted by TDS.
Cert_X	is a public key certificate for entity X.
H	denotes a hash function, as defined in section 1.5.1.
$S_X(Z)$	is the digital signature of data Z computed using entity X's private signature transformation.
P_X	is the public asymmetric key of X.
S_X	is the private asymmetric key of X.

max	denotes a maximum state appraisal function which calculates, as a function of an agent's current state, the maximum set of permissions to be accorded to an agent, as described in section 2.5.3.1.
req	denotes a requested state appraisal function which calculates, as a function of an agent's current state, the requested set of permissions the sender wants an agent to have, as described in section 2.5.3.1.
Id_X	is an identifier for X.
$X\|\|Y$	is the result of the concatenation of data items X and Y in that order.
domain	is a group of objects to which policies apply.

5.6 Assumptions

The following pre-conditions need to be satisfied for use of the framework described later in this chapter.

1. There exists a certification authority, C, trusted by A, B and *TDS*. A, B, and *TDS* possess a trusted copy of the public key of C, so that they can both verify certificates generated by C.

2. There exists an accreditation authority, N, which is trusted by *TDS*. The (agent) code author A and/or the agent creator B may be accredited as trustworthy by N.

3. The accreditation certificates, if issued by N to A and/or B, must be made available to *TDS*.

4. The (agent) code author A possesses a signature key pair.

5. The private signing key from the pair referred to in point 4, is securely stored by A.

6. The (agent) code author A has a certificate, $Cert_A$, issued by C. This certificate associates the identity of A with the public verification key from the pair referred to in point 4. This certificate must be available to *TDS*.

7. The (agent) code author A can generate proofs for its code, or, alternatively, can have proofs of code generated by a dedicated entity, P.

8. The (agent) code author A can generate a state appraisal function, *max*, which calculates, as a function of the agent's current state, the maximum set of permissions to be accorded to an agent running the program, as described in section 2.5.3.1.

9. The agent code author A can digitally sign the (agent) code it has written, in conjunction with any security controls generated for that code, using the key referred to in step 5.

10. The agent creator B possesses a signature key pair.

11. The private signing key from the pair referred to in point 10, is securely stored by B.

12. The agent creator B has a certificate, $Cert_B$, issued by C. This certificate associates the identity of B with the public verification key from the pair referred to in point 10. This certificate must be available to TDS.

13. The agent creator, B, can generate a state appraisal function, *req*, which calculates, as a function of the agent's current state, the requested set of permissions the sender wants the agent running the program to have, as described in section 2.5.3.1.

14. The agent creator, B, can digitally sign agents it has constructed, in conjunction with any security controls generated for that agent, using the key referred to in step 5.

15. The trusted domain server TDS possesses a signature key pair.

16. The private signing key from the pair referred to in point 15, is securely stored by TDS.

17. The public key of *TDS* is embedded in M in a public key store labeled 'Trusted Domain Servers'.

18. TDS will intercept all executables destined for M.

19. M is initialised with a pre-defined set of (D)TPL and Ponder security polices.

5.7 Trusted domain server activity

When an executable, destined for M, is intercepted by *TDS*, *TDS* attempts to complete a pre-defined set of security checks on the intercepted executable. As *TDS* completes these security checks, a trust profile, which describes the results of each security check completed, is constructed. Once this trust profile has been completed, it is recorded in a SAML assertion, which also holds a unique identifier for the executable with which it is associated. *TDS* then signs the SAML attribute assertion and forwards both the executable and the assertion to *M*.

5.7.1 Evaluating a mobile agent

In this section, we describe the security checks that *TDS* attempts to complete on an incoming executable and the process by which an executable's trust profile is constructed by *TDS*. These checks are used to complete a 10-variable trust profile (t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}), where each of the values is equal to an integer, as defined below. A similar approach is described in [1].

1. *TDS* first checks whether the incoming executable is mobile code or a mobile agent. This is then recorded in the first field, t_1, of the trust profile of the executable, which may have one of two values:

- 1, which implies that the executable is mobile code.
- 0, which implies that the executable is a mobile agent.

2. A rigorous virus check is then completed on the executable code. Once this has been completed, the second field in the trust profile, t_2, may be assigned one of three values:
   - -1, which implies that the virus check failed and the code may be malicious.
   - 0, which implies that the virus check was not completed.
   - 1, which implies that the virus check was successfully completed.
3. *TDS* then verifies the (agent) code author's digital signature. It then compares the code author's identifier, Id_A, with lists indicating whether *TDS* has assigned the author complete trust, high trust, medium trust, or if it is considered distrusted. Once this has been completed, the third field in the trust profile, t_3, is assigned one of the following values:
   - 3, which implies that Id_A is contained in a list of authors in which *TDS* has complete trust. Only an agent code author contained in this list can generate a maximum state appraisal function, *max*, which will be accepted and executed by *TDS*. The function *max* defines the maximum privileges that should be allotted to the corresponding agent when executing.
   - 2, which implies that Id_A is contained in a list of authors in which *TDS* has high trust.
   - 1, which implies that Id_A is contained in a list of authors in which *TDS* has medium trust.
   - 0, which implies that Id_A is not contained in any of the defined lists. The author is unknown.

- -1, which implies that Id_A is contained in a list of authors which TDS distrusts. An author is also assigned the value -1 if its signature cannot be verified.

4. The agent creator's digital signature is verified. The identifier of the agent creator, Id_B, is then compared with lists indicating whether TDS has assigned the creator high trust, medium trust, or if it is considered distrusted. Once this has been completed, the fourth field in the trust profile, t_4, is assigned one of the following values:

   - 2, which implies that Id_B is contained in a list of creators in which TDS has high trust.
   - 1, which implies that the creator's identity is contained in a list of creators in which TDS has medium trust.
   - 0, which implies that Id_B is not contained in any of the defined lists. The creator is unknown.
   - -1, which implies that Id_B is contained in a list of creators which TDS distrusts. A creator is also assigned the value -1 if its signature cannot be verified.

   A creator of any trust level is permitted to generate a requested state appraisal function, *req*, which will be executed by TDS if the agent code author field, t_1 has been allocated a trust value of 3 and A has generated a *max* state appraisal function.

5. In certain cases, TDS may not have a trust relationship with A and/or B. In such instances, TDS may contact a third party accreditation authority, N, with which A and/or B may have registered, so that a basic level of trust may be allocated to an incoming executable of unknown origin. Once

this has been completed, t_5 and t_6, the fifth and sixth fields in the trust profile, which respectively represent whether the (agent) code author and the agent creator have been accredited, are assigned one of the following values.

- 1, which implies that A or B has been accredited.
- 0, which implies that A or B has not been accredited, or that *TDS* did not complete this check.

6. *TDS* may verify any proofs of code appended to the incoming executable. In order to utilise proofs of code in this context, we require that a predefined set of security properties are proved by the proof producer. For example, it may be useful to know that a executable terminates within a given number of instructions, and that, if it sends network packets, the volume does not exceed a preset bandwidth, two properties which can be proved [107]. In this case, the seventh field in the trust profile, t_7, which indicates whether proofs of code were verified, is assigned one of the following values.

   - 1, which implies that the proofs of code were successfully validated.
   - 0, which implies that the proofs of code were not validated or that there were no proofs of code appended to the executable.
   - -1, which implies that the proofs of code could not be successfully validated.

   In conjunction with this, the values of the properties can also be recorded in the executable's profile. So if, for example, it is proved that an agent terminates within a given number of instructions, the number of cycles could be recorded in t_8, the eighth field of the trust profile. This field may

be assigned a value of 0 if the properties has not been proved. Depending on the number of properties that have been proved, the number of fields in the trust profile may be extended accordingly.

7. If an incoming agent has an associated set of state appraisal functions [11], and A has been assigned 'complete trust', then both the *max* and *req* functions are evaluated by *TDS*. In this case, the ninth field in the trust profile, t_9, which indicates the result of this security check, is assigned one of the following values.

    - 1, which implies that the output of *max* is greater than or equal to the output of *req*.
    - 0, which implies that the state appraisal functions were not executed, either because there were no state appraisal functions appended to the agent, or, because the trust value assigned to the (agent) code author was not equal to 3.
    - -1, which implies that the output of *req* is greater than the output of *max*.

8. Finally, *TDS* may complete a pre-defined set of tests on the incoming executable using static analysis tools, as described in section 3.5. In this case, the tenth field t_{10} of the executable trust profile may be assigned one of three values.

    - 1, which implies that the pre-defined test set was successfully completed on the executable.
    - 0, which implies that the pre-defined test set was not completed on the executable.
    - -1, which implies that the pre-defined test set uncovered a security vulnerability in the executable.

The authorisation policy of *TDS* may require special actions in the case of negative attribute values. For example, *TDS* may communicate an executable's trust profile to the end host, inclusive of '-1' values, which indicate that a security check has unearthed a potentially dangerous executable property. Alternatively, *TDS* may simply choose to discard any executable whose profile contains a field with a value of '-1'.

Whether a security check is completed is dependent on the security controls appended to the incoming executable and the policy of a particular trusted domain server. In conjunction with this, each trusted domain sever may choose to, or may only be capable of, executing a specified number and type of security checks. Alternatively, the trust profile may be extended to include more or a different set of checks. Also, the completion of certain checks by a trusted domain server may depend on the results output from other security checks completed.

Indeed the sample set of 10 attributes listed above could be replaced by a list of indeterminate length, with each required attribute name and value encoded in XML.

5.7.2 Attribute certificates

Once a trust profile has been constructed, a SAML attribute assertion which describes this profile is generated by *TDS* and sent with the executable to M. In order to specify this trust profile, three requirements must be met.

1. It must be possible for *TDS* to sign attribute certificates.

2. The identity of an incoming executable must be expressible within an attribute certificate (as the hash of the incoming mobile code or a mobile agent) so that the attribute certificate can be bound to a particular exe-

cutable.

3. The trust profile of an incoming executable must be expressible within its associated attribute certificate.

As demonstrated in section 4.6, all three of these requirements can be met using SAML attribute assertions. SAML assertions can be signed using an XML signature. An identifier for the certified executable, in this instance a hash of the executable, can be held in the <SubjectConfirmationData> element. Each of the executable's trust profile field names and values can be recorded in an <Attribute> element. The SAML attribute assertion shown in figure 5.2 describes an executable which has been assigned the trust profile $(0, 1, 3, 2, 0, 0, 1, 20, 0, 0)$.

```
<?xml version="1.0" encoding="utf-8"?>
<Assertion
 xmlns="urn:oasis:names:tc:SAML:1.0:assertion"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 MajorVersion = "1"
 MinorVersion = "0"
 AssertionID = "6738467-47378dj-hu234832"
 Issuer = "TDS"
 IssueInstant = "2001-05-31T13: 20:00-05:00"
 <Conditions
   Notbefore = "2001-05-31T13: 20:00-05:00"
   NotOnOrAfter = "2002-10-31T13: 20:00-05:00">
 <AttributeStatement>
  <Subject>
   <SubjectConfirmation>
     <ConfirmationMethod>
      urn:org:names:tc:SAMLExtn:1.0:cm:hash-of-executable
     </ConfirmationMethod>
     <SubjectConfirmationData>....</SubjectConfirmationData>
     </SubjectConfirmation>
   </Subject>
   <Attribute>
     <AttributeDesignator
       AttributeName = "ExecutableType"
       AttributeNamespace = "http://www.TDS.com/AttributeService"/>
       <AttributeValue>0</AttributeValue>
    </Attribute>
    <Attribute>
     <AttributeDesignator
       AttributeName = "VirusCheck"
       AttributeNamespace = "http://www.TDS.com/AttributeService"/>
       <AttributeValue>1</AttributeValue>
    </Attribute>
    <Attribute>
     <AttributeDesignator
       AttributeName = "AuthorTrust"
       AttributeNamespace = "http://www.TDS.com/AttributeService"/>
```

```
      <AttributeValue>3</AttributeValue>
    </Attribute>
    <Attribute>
     <AttributeDesignator
       AttributeName = "CreatorTrust"
       AttributeNamespace = "http://www.TDS.com/AttributeService"/>
       <AttributeValue>2</AttributeValue>
    </Attribute>
    <Attribute>
    <AttributeDesignator
      AttributeName = "AuthorAccreditation"
      AttributeNamespace = "http://www.TDS.com/AttributeService"/>
      <AttributeValue>0</AttributeValue>
    </Attribute>
    <Attribute>
    <AttributeDesignator
      AttributeName = "CreatorAccreditation"
      AttributeNamespace = "http://www.TDS.com/AttributeService"/>
      <AttributeValue>0</AttributeValue>
    </Attribute>
    <Attribute>
    <AttributeDesignator
      AttributeName = "CodeProofs"
      AttributeNamespace = "http://www.TDS.com/AttributeService"/>
      <AttributeValue>1</AttributeValue>
    </Attribute>
    <Attribute>
    <AttributeDesignator
      AttributeName = "CPUCycles"
      AttributeNamespace = "http://www.TDS.com/AttributeService"/>
      <AttributeValue>20</AttributeValue>
    </Attribute>
    <Attribute>
    <AttributeDesignator
      AttributeName = "StateAppraisal"
      AttributeNamespace = "http://www.TDS.com/AttributeService"/>
      <AttributeValue>0</AttributeValue>
   </Attribute>
   <Attribute>
    <AttributeDesignator
      AttributeName = "Testing"
      AttributeNamespace = "http://www.TDS.com/AttributeService"/>
      <AttributeValue>0</AttributeValue>
    </Attribute>
 </AttributeStatement>
 <ds:Signature>...</ds:Signature>
</Assertion>
```

Figure 5.2: Sample SAML attribute assertion generated by *TDS*

5.7.3 Authentication evidence

In order to implement the framework, two requirements must be fulfilled with respect to authentication evidence.

1. It is required that the trusted domain server which generated and signed the attribute certificate of the incoming executable can be authenticated through the verification of the digital signature in the incoming executable's attribute certificate. The digital signature serves as the authentication evidence.

2. It is required that an incoming executable can be authenticated through the verification of the hash of the incoming (agent) code and data, or the hash of the entire agent (including agent code, data and any static and dynamic state information), against the hash signed by the entity/entities who 'speak(s) for' the executable, see section 3.9.

The XML signature of *TDS* on the signed SAML attribute assertion, as shown in figure 5.2, serves as authentication evidence for *TDS*. The authentication evidence for the incoming executable, i.e. the hash of the executable, is contained in the <SubjectConfirmationData> element of the attribute assertion, as described in section 4.6.

5.8 End host activity

Once received, the attribute assertion, which specifies the trust profile of the incoming executable, is used by M to assign the executable to a domain, which is synonymous with a set of privileges to access resources on the device. Alternatively, depending on its trust profile, the incoming executable may not be permitted to execute. The end host maintains a set of mappings from the set of

trust profiles to the set of domains in the form of policy statements. A mapping has the following form:

$$(t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}) \rightarrow domain_i.$$

The sandboxes (which correspond to permissions assigned to domains) on the mobile device may implement a variety of access policies on the mobile device. For example, it is possible for separation of duty policies to be implemented, that prevent an executable performing a potentially damaging combination or sequence of actions [42].

Once an executable's SAML attribute assertion has been received, the following process is completed.

- The origin of the executable's attribute assertion is authenticated by verifying the signature of *TDS* on the attribute assertion.
- The executable is authenticated by hashing it and comparing the result to the hash value stored in the <SubjectConfirmationData> element of the attribute assertion.
- The executable is assigned to a domain by M:
  - The trust profile is recovered from the attribute assertion;
  - The executable is assigned to a sandbox by the TEM based on its trust profile.
- The execution of the executable is controlled:
  - The policy decision point determines the execution permissions which should be assigned to an executable, based on its domain assignment.
  - Policy enforcement point enforces the decision of the policy decision point.

5.8.1 The PIP

The PIP implemented on the mobile device must fulfil one requirement.

1. The PIP must collect security data relevant to the access requestor.

If the required SAML attribute assertion is not received with the incoming executable, it can be requested by M from *TDS*. Figure 5.3 and 5.4 show examples of request and response messages which may be sent between M and *TDS* so that the required attribute assertion can be retrieved, if it exists.

```
<?xml version="1.0" encoding="utf-8"?>
<samlp:Request
 xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
 xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 RequestID = "567.14.234.20.90123456"
 MajorVersion = "1"
 MinorVersion = "0"
 IssueInstant = "2001-06-31T13:20:00-05:00">
 <RespondWith>saml:AttributeStatement</RespondWith>
   <AttributeQuery>
     </saml:Subject>
       <SubjectConfirmation>
         <ConfirmationMethod>
           urn:org:names:tc:SAMLExtn:1.0:cm:hash-of-executable
         </ConfirmationMethod>
         <SubjectConfirmationData>....</SubjectConfirmationData>
       </SubjectConfirmation>
     </saml:Subject>
   </AttributeQuery>
</samlp:Request>
```

Figure 5.3: Sample SAML attribute assertion request

5.8.2 The AP

Once the attribute certificate has been received, two authentication point requirements must be met.

1. It is required that the trusted domain server which generated and signed the attribute certificate of the incoming executable, can be authenticated

```
<?xml version="1.0" encoding="utf-8"?>
<samlp:Response
    xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
    xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    ResponseID = "333.15.774.21.9012999"
    InResponseTo = "567.14.234.20.90123456"
    MajorVersion = "1"
    MinorVersion = "0"
    IssueInstant = "2003-06-31T13:20:00-05:00"
    Recipient = "mobilehost4">
    <Status
       <StatusCode = "Success"/>
     </Status>
    <saml:Assertion> ... </saml:Assertion>
</samlp:Response>
```

Figure 5.4: Sample SAML attribute assertion response

by verifying the digital signature in the incoming executable's attribute certificate.

2. It is required that the incoming executable can be authenticated by verifying the hash of the incoming (agent) code and data, or the hash of the entire agent (including agent code, data and any static and dynamic state information), against the hash signed by the entity/entities who 'speak(s) for' the executable, see section 3.9.

A SAML processor verifies the signature of *TDS* on the incoming attribute assertion, thereby meeting the first AP requirement outlined above. The SAML Assertions and Protocols specification [112] define a set of constraints on the XML syntax for signing data, so that SAML processors do not have to deal with the full generality of XML signature processing, see section 4.6. In order for a SAML processor to authenticate the incoming executable, i.e. be able to process the <SubjectConfirmationData>, the trusted domain server and the devices with which it is communicating must have agreed on the stated <ConfirmationMethod>, as was described in section 4.6. Alternatively, a new profile, which describes this new confirmation method, must be specified.

5.8.3 The TEM

The TEM implemented on M must fulfil the following requirement.

1. The TEM must map the unknown authorisation requestor to a principal to which execution permission policies apply.

In order to meet this requirement we use components from the (D)TPL trust establishment system. (D)TPL is "used to define the mapping of strangers to predefined roles, based on certificates issued by third parties" and is explored in a paper by Herzberg, Mass, Michaeli, Naor and Ravid [68]. Within this trust establishment system, four components are defined:

- a generic trust establishment certificate object;
- a trust policy language, (D)TPL, specified by an XML DTD;
- a policy engine; and
- a certificate collector and repository.

This trust establishment policy language has two variants. The first is DTPL, which is monotonic, i.e. it does not include negative rules. The second is TPL, which is non-monotonic. For the purposes of this chapter, we consider DTPL.

In order to express a policy using DTPL, the following elements and attributes can be used.

- <GROUP> — has one attribute, NAME. The group tag is used to define the name of a domain to which the unknown entity (or more specifically the public key of the unknown entity) will be mapped.
- <RULE> — defines the certificate set necessary to join a domain. It may contain requirements on the groups of which certificate issuers must be

members, and requirements on the attributes of certificates. It is made up of one or more <INCLUSION> elements and at most one <FUNCTION> element.

- <INCLUSION> — is used to define a certificate which must exist for the rule to hold. Its attributes include [68]:
  - ID — a unique ID for the certificate, which will be used in the <FUNCTION> element to refer to the certificate's fields. This is a local name used only in the scope of the RULE.
  - TYPE — refers to the type of certificate required.
  - FROM — identifies the name or one or more groups to which the issuer should belong.
  - REPEAT — defines the number of certificates of the specified type that must exist.
  - DEPTH — may be used to limit the length of a certificate chain.
- <FUNCTION> — is used to define other conditions which must be fulfilled by the certificate field values [68].

5.8.3.1 TEM policy statements

We assume that there is at least one pre-defined group on every mobile device, namely 'Trusted Domain Servers'. When the mobile device requires the services of a specific trusted domain server, *TDS*, the public key of the trusted domain server is associated with the 'Trusted Domain Servers' group. On receipt of an attribute assertion, the mobile device verifies the signature, and checks that *TDS* is a member of 'Trusted Domain Servers'.

An example of trust establishment policy statements which might be held on M is illustrated in figure 5.5. These policy statements specify that an

executable which has a SAML attribute certificate, which has been signed by an entity whose public key is contained in 'Trusted Domain Servers', and has the listed attribute values, should be mapped to the domain '/executables/trustedMobileExecutables.

```
<?xml version="1.0"?>
<POLICY>
     <GROUP NAME="Trusted Domain Servers">
     </GROUP>

     <!---->
     <!-- preferred executables -->
     <!---->

     <GROUP NAME="/executables/trustedMobileExecutables">
       <RULE>
        <INCLUSION ID="ExecCert" TYPE="SAMLAttributeCertificate"
         FROM="Trusted Domain Servers" DEPTH="1"></INCLUSION>
           <FUNCTION>
             <AND>
              <EQ>
               <FIELD ID="ExecCert" NAME="ExecutableType"></FIELD>
               <CONST>0</CONST>
              </EQ>
              <EQ>
               <FIELD ID="ExecCert" NAME="VirusCheck"></FIELD>
               <CONST>1</CONST>
              </EQ>
              <EQ>
               <FIELD ID="ExecCert" NAME="AuthorTrust"></FIELD>
               <CONST>3</CONST>
              </EQ>
              <EQ>
               <FIELD ID="ExecCert" NAME="CreatorTrust"></FIELD>
               <CONST>2</CONST>
              </EQ>
              <EQ>
               <FIELD ID="ExecCert" NAME="AuthorAccreditation"></FIELD>
               <CONST>0</CONST>
              </EQ>
              <EQ>
               <FIELD ID="ExecCert" NAME="CreatorAccreditation"></FIELD>
               <CONST>0</CONST>
              </EQ>
```

```
            <EQ>
             <FIELD ID="ExecCert" NAME="CodeProofs"></FIELD>
             <CONST>1</CONST>
            </EQ>
            <EQ>
             <FIELD ID="ExecCert" NAME="CPUCycles"></FIELD>
             <CONST>20</CONST>
            <EQ>
            <EQ>
             <FIELD ID="ExecCert" NAME="StateAppraisal"></FIELD>
             <CONST>0</CONST>
            </EQ>
            <EQ>
             <FIELD ID="ExecCert" NAME="Testing"></FIELD>
             <CONST>0</CONST>
            </EQ>
           <AND>
          </FUNCTION>
        </RULE>
    </GROUP>
</POLICY>
```

Figure 5.5: Sample DTPL policy statement

5.8.3.2 The TEM policy decision and enforcement points

A simple TEM policy decision point is required, which inputs the executable's SAML assertion, and outputs a domain name to which the executable should be assigned (i.e. made a member). Given the assertion shown in figure 5.2, and the TE policy statement described in figure 5.5, the executable would be mapped to the domain, /executables/trustedMobileExecutables.

5.8.3.3 The TEM policy administration point

The TEM PAP implemented on the mobile device must fulfil the following requirement.

1. The PAP must provide a means for specifying, managing and organising mobile device security policy statements.

As the policy statements are defined in XML, they can be viewed or edited using a text editor or graphically. A graphic editor has been developed by Herzberg et al. [68], which displays the policy as a graph, where nodes are groups and the edges are the rules defining the relationships between groups.

5.8.4 Policy statements

Once the incoming executable has been assigned to a domain, in this instance, /executables/trustedMobileExecutables, it can be controlled using a pre-defined set of Ponder policy statements. Figures 5.6 gives an example of the access rights and constraints that could be enforced on an executable which is a member of the domain, /executables/trustedMobileExecutables.

```
inst group TrustedMobileExecutablesExecution {

Inst auth+   {
       subject      s = /executables/trustedMobileExecutables;
       target       f = /files/PublicFiles;
       action       f.read();
    }

inst oblig  {
        on CPU(load,90);
        subject s = System ;
        target t = /executables/trustedMobileExecutables;
         do t.kill();
    }
 }
```

Figure 5.6: Sample Ponder composite policy statement

5.8.5 The PDP and PEP

The PDP and PEP must meet the following requirements.

1. The PDP must decide whether a particular access request should be permitted.

2. The PEP must enforce the decisions of the PDP.

As was described in section 4.5, in the Ponder policy specification framework an access controller is defined as an agent which receives an authorisation request, makes a policy decision with regard to authorisation policies, and enforces this decision. It also decides whether filters should be applied to the action call parameters or the returned values. One access controller exists for each target object. A policy management component is defined as an agent which decides, based on events, whether obligation and refrain policies need to be enforced. One policy management component is defined for each object. This agent also enforces obligation and refrain policies.

5.8.6 The PAP

In order to enable policy administration the following requirement must be met.

1. The PAP must provide a means for specifying, managing and organising mobile device security policy statements.

As described in section 4.5, a set of management tools are provided to support the deployment of a Ponder policy specification framework.

- A domain browser allows the navigation of objects within the domain server. External tools may be invoked from within the browser tool or, alternatively, other tools may interface with the domain browser through an invocation interface which is implemented by the domain browser.
- A policy editor enables policy and role creation and modification.
- A policy compiler compiles specified policies, after which they are stored by the domain service.
- A management console is used to distribute policies to their enforcement components. This management console tool can interface with the domain

browser to select policies and enforcement components from the directory. It may also interact with the policy compiler in order to dynamically instantiate policies.

- A user-role management tool enables the management of user-roles.
- A GUI component consists of a main console for accessing individual tools.
- A configuration manager tool enables the configuration of all Ponder tools.

The domain service manages a hierarchy of domain objects. Each domain object holds references to the policy objects that currently apply to that domain.

5.9 Conclusions

Mobile code, and more specifically mobile agents, offer many potential benefits in a mobile environment where a permanent connection is not always possible, devices are often disconnected for long periods of time, connections are often characterised by low bandwidth, high latency and may be error prone. The ability of mobile agents to adapt and make decisions based on the distributor's preferences also makes them a versatile and powerful tool. The lack of resources on mobile devices provides an additional incentive for the use of mobile agent technologies. Nevertheless, before their widespread deployment, it is necessary to address mobile code and agent security concerns.

In this part of the thesis we examined the threats to host security introduced by the use of mobile code. In addition to the threats posed by mobile code, a mobile agent may increase security risks through the introduction of malicious state information. Following this, we examined the state of the art in technologies for mobile code and agent authorisation, and examined the reasons why many of the solutions proposed do not transfer well to the mobile environ-

ment. Based on this analysis, we proposed the development of a policy-based framework for mobile code and agent authorisation.

In order to construct this policy-based authorisation framework, two steps were completed. Firstly, we proposed six possible architectural models upon which a policy-based framework for the authorisation of incoming executables and, more specifically, mobile agents, could be built. Each model was then analysed with respect to the level of security it could support, and with regard to its suitability for implementation in a mobile environment. From this, we were able to compile a set of requirements for the an optimal architecture model for a policy-based code and agent authorisation framework.

Secondly we examined a selection of policy statement and attribute certificate specification languages, namely KeyNote (and Nereus), Ponder (and (D)TPL), and SAML, and explored the functionality of their supporting policy engine components. The main goal of this analysis was to discover whether these languages could express the policy statements and attribute certificates required by the six scenarios described in chapter 3, and also whether the necessary policy engine component functionality could be supported. Our conclusions from this analysis were then used in chapter 5 to choose the most appropriate language(s) for policy statement and attribute certificate expression in our policy-based framework.

In this chapter, we have proposed a policy-based framework that synthesises techniques for establishing the origin, authenticity, safety and integrity of incoming mobile executables, and policy-specification and processing techniques, in order to provide a rich and flexible authorisation framework. It is most suitable for deployment in an environment where end hosts may be limited in terms of processing power or the checks they can complete.

Part II

Mobile code protection

Chapter 6

Conditional access in mobile systems

Contents

6.1 Introduction 234
6.2 Conditional access systems 236
6.2.1 DVB standards 238
6.2.2 Simulcrypt 238
6.2.3 Common interface 239
6.2.4 Limitations of existing mechanisms 240
6.2.5 Modifications required for mobile receivers 241
6.3 Security issues 242
6.3.1 Security threats 242
6.3.2 Security services and mechanisms 243
6.4 Conclusions 245

A conditional access system is used to prevent unauthorised access to broadcast content. The Digital Video Broadcast organisation has developed standards which enable an end user to acquire broadcast services from a variety of service providers that use proprietary conditional access systems to protect their content. This chapter examines these DVB standards and describes certain limitations which arise when they are applied in a mobile environment. In order to overcome these limitations, the mobile platform could be re-configured to be compatible with the appropriate conditional access system, if the proprietary conditional access application is implemented entirely in software. Such a software

application could be delivered to the mobile device on demand. The remainder of this chapter explores the threats resulting from the introduction of reconfigurable receivers in a mobile environment, and identifies the security services and security mechanisms required for the protected download of a conditional access application to a mobile receiver.

6.1 Introduction

One of the driving forces behind recent developments in mobile communications systems is the potential for such systems to deliver more complex content to consumers. Current 3G systems are capable of delivering multimedia clips to mobile phones. The next generation of communications systems is expected to develop this service, and collaborate with broadcast systems to provide wireless access to video content from a wide range of mobile devices. For a service like this to achieve its full commercial potential, the owners of the content will require assurance that their material is not illegally accessed. Current broadcast systems accomplish this by using conditional access systems to ensure that only bona fide subscribers have access to the content.

Services broadcast today, however, use a range of proprietary conditional access systems. In order to enable an end user to acquire broadcast services from a variety of service providers, which use proprietary conditional access systems to protect their content, the DVB organisation[1] has developed several standard mechanisms. While receivers remain static, and consumers subscribe to one or two service providers, the mechanisms specified in the DVB standards provide a practical solution. However, if a mobile subscriber requires access to services controlled by several different conditional access systems, then the current solution becomes increasingly impractical. If future wireless devices are to maximise their access to broadcast services, then a more practical and cost effective means will be required to allow consumer products to support a wide range of proprietary conditional access systems.

In part II of this thesis we consider the possibility of downloading legacy conditional access applications on demand to the mobile device. Part II is

[1] www.dvb.org

structured as follows:

- This chapter explores the threats resulting from the introduction of reconfigurable receivers in a mobile environment, and identifies the security services and security mechanisms required for the protected download and execution of a conditional access application on a mobile receiver.

- Chapter 7 describes two protocols, a key exchange and a key agreement protocol, designed to protect against threats 1 to 5, described in section 6.3.1, for the protected download and execution of a conditional access system in a mobile environment.

- Chapter 8 explores three possible implementations of the generic key exchange and key agreement protocols described in chapter 7. The first implementation assumes the presence of a mobile device into which components described in the TCG version 1.2 specification set are integrated. The second implementation assumes a mobile device architecture into which a version 1.2 compliant TPM and CRTM are integrated and an isolation layer deployed. Finally, the third implementation assumes an NGSCB compliant platform, as described by Microsoft. Each implementation description is accompanied by an analysis which examines how well the security of the downloaded application is protected against threats 6 and 7, described in section 6.3.1.

- Finally in chapter 9 we examine two protocols for secure application download which have been proposed by the designers of XOM and AEGIS, Lie et al. and Suh et al. These protocols are based upon the assumption that the host device contains a hardened processor rather than a trusted module, as assumed in chapters 7 and 8. Both protocols are analysed against the security requirements described in chapter 6. As a result of these anal-

yses, recommendations are made regarding possible protocol modifications designed to address identified security issues.

Section 6.2 of this chapter examines the mechanisms currently used to protect broadcast content, and describes certain limitations which arise when they are applied in a mobile environment. As stated above, in order to overcome these limitations, the mobile platform could be re-configured to be compatible with the appropriate conditional access system, as long as the proprietary conditional access application is implemented entirely in software. Section 6.3 explores the threats which result from the introduction of reconfigurable receivers in a mobile environment, and identifies the security services and security mechanisms required to securely download and execute a conditional access application on a mobile receiver.

6.2 Conditional access systems

A conditional access system is defined as a "complete system for ensuring that broadcast services are only accessible to those who are entitled to receive them" [8]. A conditional access system conforming to the DVB standards is comprised of two main components [33]:

- A scrambling subsystem; and
- A proprietary access control subsystem.

The scrambling subsystem scrambles broadcast services using the symmetric common scrambling algorithm (CSA) and a key known as a control word (CW) [44]. Since the cryptographic scheme is a symmetric algorithm, the CW must be delivered to the receiver in a secure manner, i.e. so that its confidentiality is protected. This is the function of the proprietary access control subsystem.

Table 6.1: Conditional access system vendors

CA System	Vendor	
Viaccess	Viaccess SA	www.viaccess.fr
NagraVision	Kudelski	www.kudelski.com
Videoguard	NDS	www.nds.com
Mediguard	Canal+	www.canalplus-technologies.com
Mcrypt	Irdeto	www.irdetoaccess.com
PiSys	Irdeto	www.irdetoaccess.com
CryptoWorks	Philips	www.digitalnetworks.philips.com
BetaCrypt	BetaResearch	www.betaresearch.de
Conax	Telenor	www.telenorsbc.com
Digicipher	Motorola	www.broadband.motorola.com
PowerKey	Scientific Atlanta	www.sciatl.com

The control word and the program attribute information, which is used by the receiver to determine whether a subscriber is entitled to view a program on the basis of his or her subscription, is encrypted by the access control subsystem using a service key [134]. The resulting encrypted bundle, called an entitlement control message (ECM), is broadcast in advance of the scrambled service. The service key and the subscription information of a particular receiver, are then encrypted using a key unique to that particular receiver. This encrypted bundle, called an entitlement management message (EMM) must also be communicated to the receiver in advance of the scrambled service. An EMM may be broadcast or may be communicated using an alternative method, such as by telephone.

As can be seen from table 6.1, there are many proprietary conditional access systems available for broadcasters to choose from. Although the conditional access systems remain proprietary, most vendors have adopted the DVB standard protocols. These protocols provide a well defined interface between the proprietary conditional access system and the rest of the broadcast equipment. The remainder of this section describes the DVB protocols and assesses their suitability for use with highly mobile receivers.

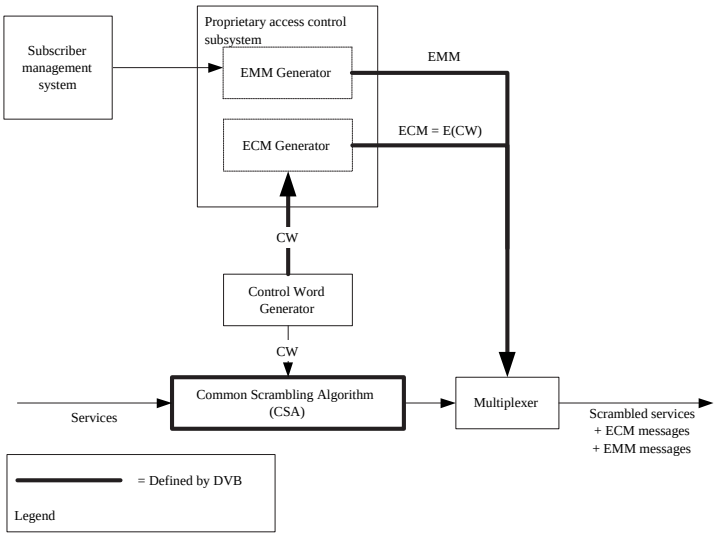

Figure 6.1: Scrambling broadcast services using DVB standards

6.2.1 DVB standards

The DVB standards specify two mechanisms that aim to provide some flexibility in the application of proprietary conditional access systems to broadcast services [33]. At the transmission site, the simulcrypt standard [45] allows a service to be controlled by two or more conditional access systems. At the receiver, the common interface standard [22] allows conditional access modules to be plugged into pc-card slots in the receiver to configure the device for the required conditional access system. Both systems rely on the service being scrambled using the standard common scrambling algorithm [44]. Figure 6.1 illustrates how an implementation of these standards would be used to scramble a broadcast service.

6.2.2 Simulcrypt

If a second conditional access system is available to the broadcaster, then the same control word may also be protected using this access control subsystem,

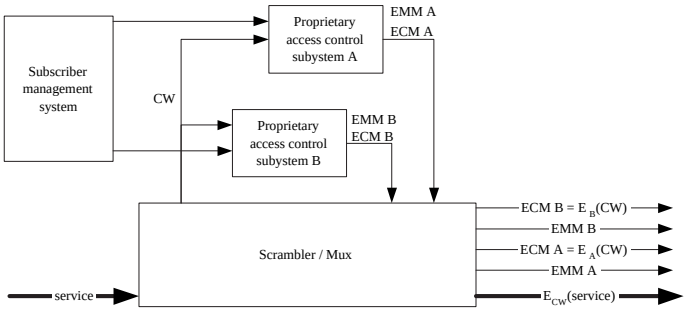

Figure 6.2: Simulcrypt

as shown in figure 6.2. The simulcrypt standard [45] describes the interface to the access control subsystems and the synchronisation protocols. All entitlement messages are then broadcast in advance of the scrambled service. Thus, receivers running either conditional access system are able to recover the control word and access the scrambled service. As far as the consumer is concerned, the operation of this system is completely transparent. The service provider, however, must operate multiple conditional access systems.

While it may be commercially feasible for large networks to operate two or three conditional access systems, the cost and logistics of running many such systems simultaneously could prove to be prohibitive, especially for smaller networks.

6.2.3 Common interface

A parallel means of supporting multiple conditional access systems is to provide a solution at the receiver. This is accomplished by specifying a standard interface for the receiver that provides access to the scrambled service and the encrypted control words [22]. Figure 6.3 shows how the scrambled service and the entitlement messages can be passed to a separate pc-card module containing the hardware and software for a specific conditional access system. The

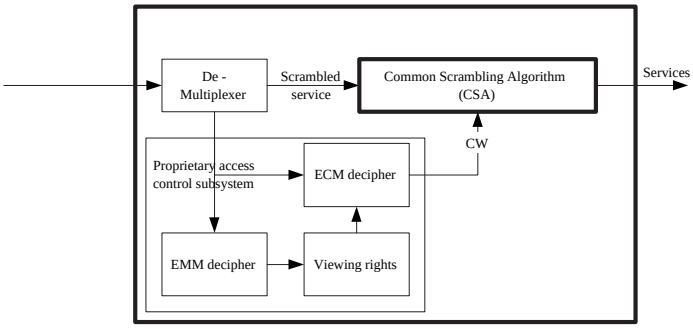

Figure 6.3: Common interface module

encrypted control words can be decrypted by the proprietary access control subsystem which resides on the module to provide access to the service.

By swapping modules, the receiver can thus be configured to match the conditional access system used by the broadcaster. This system is therefore transparent to the broadcaster but not the subscriber. As is the case with simulcrypt, this mechanism may provide a solution for two or three conditional access systems, but, as the numbers increase, the cost to the subscriber could become prohibitive.

6.2.4 Limitations of existing mechanisms

The solutions described above are well suited to current technology, where services are generally controlled by one or two conditional access systems and subscribers generally only require authorisation from one or two broadcasters. With a mobile receiver, however, subscribers will require authorisation from an increasing number of service providers as they travel further afield.

The common interface solution would require mobile devices to have a pc-card interface and the user to carry a number of modules. The cost of adding such an interface as well as the practical design issues could make this infeasible. The cost of the modules may also deter some subscribers. The alternative

solution using current technology would be to broadcast each service under the control of as many conditional access systems as possible, which could prove prohibitively expensive for many broadcasters, especially those operating in small niche markets.

Thus both current solutions have potential difficulties, which would significantly restrict the content available to mobile receivers. These existing solutions do not transfer well to mobile systems, and the problem would benefit from reconsideration in the light of the new requirements which arise in a mobile environment.

6.2.5 Modifications required for mobile receivers

The objective is to provide access controlled broadcast content to mobile devices. This should be achieved with minimum impact on existing networks while at the same time minimising the hardware overhead on the mobile device and the cost to the user. An attractive solution is to re-configure a mobile platform to be compatible with the appropriate proprietary conditional access system. The implication of this is that the proprietary application is implemented entirely in software and delivered to the mobile device on demand.

The main difficulty with such an approach lies in convincing the software provider that the application, including any embedded secret keys, will be protected to at least the level offered by current solutions. This contrasts with the more familiar problem of securing the platform against incoming malicious code, as discussed in part I of this thesis.

To address this problem, the delivery mechanism must protect both the integrity and confidentiality of the application. Moreover, the mobile device must be able to demonstrate to the software provider that the platform on which

the application will execute can be trusted. The former can be accomplished using well known security mechanisms such as symmetric encryption and the use of message authentication codes, while the latter may be achieved by the deployment of trusted computing primitives, as described in section 1.6.

6.3 Security issues

To ensure that any solution proposed for use in a mobile environment is at least as secure as currently implemented DVB standard solutions, two fundamental security requirements must be satisfied, namely:

Secure download: The confidentiality and integrity of the conditional access application must be protected as it is transported from the software provider to the host platform, and while in storage on the host platform.

Secure execution: The conditional access application must be protected while executing on the host platform.

This section summarises the security threats, security services, and potential security mechanisms relevant to the download, storage, and execution of a conditional access application.

6.3.1 Security threats

The secure download, storage, and execution of a conditional access application is subject to a number of threats including:

1. Unauthorised reading of the application code and data.
2. Unauthorised modification of the application code and data.

3. Unknowingly communicating with an unknown and potentially malicious entity. More specifically, we will primarily concentrate on the threat of a software provider unknowingly communicating with an unknown and potentially malicious mobile platform.

4. The inability to corroborate the source of the conditional access application.

5. Replay of communications.

6. Unauthorised reading or modification of any cryptographic keys used in the provision of confidentiality and integrity protection to the conditional access application code and data.[2]

7. Unauthorised reading or modification of the application code and data while it executes on the mobile host.

6.3.2 Security services and mechanisms

The security services required to thwart the first five threats listed above may be provided using standard cryptographic mechanisms, as described in section 1.5. The services required to counter the remaining two threats require the application of mechanisms associated with trusted computing or closely related technologies, as discussed in section 1.6. There is a direct mapping between the security threats outlined in section 6.3.1 and the security services and potential security mechanisms, outlined below, which may be deployed to prevent their realisation.

1. *Confidentiality of the application code and data:* This service may be provided by symmetric or asymmetric encryption.

[2]This is essentially a secondary threat, i.e. a threat to the mechanisms which may be deployed in order to thwart threats 1 and 2 above. As we can assume that symmetric encryption is the most viable way of preventing unauthorised reading of the application code and data, there will be, at the very least, one symmetric encryption key which needs to be protected.

2. *Integrity protection of the application code and data:* A message authentication code or a digital signature can be used to provide this service.

3. *Entity authentication:*

   (a) With respect to the authentication of the mobile host to the software provider, trusted computing based platform attestation can be used to meet this service requirement, see section 1.6.6.

   (b) Authentication of the software provider to the mobile host can be provided using a unilateral entity authentication protocol.

4. *Origin authentication:* The origin of the conditional access application can be authenticated via the verification of the software provider's digital signature on either the (possibly encrypted) incoming application, or on keys used to protect the integrity and confidentiality of the incoming application.

5. *Freshness:* This can be provided by the use of nonces or timestamps.

6. *Confidentiality and integrity protection of the cryptographic key(s) used in the prevention of unauthorised reading of, and the detection of unauthorised modification to, the application code and data:* Threat 6, described above, may be countered by providing the following services:

   (a) *Secure symmetric key generation:* The key(s) must be generated in an isolated environment, for example, in a secure dedicated hardware device, or, alternatively, by an application executing in an isolated compartment, as described in section 1.6.8.

   (b) *Secure symmetric key transmission:* The confidentiality and integrity of the symmetric key(s) whilst in transit can be provided by using asymmetric encryption and digital signatures.

(c) *Secure symmetric key storage:* This service requires the availability of protected storage on the host, see for example section 1.6.7. Alternatively, the mechanisms which confidentiality and integrity-protect the symmetric key(s) whilst in transit may also be used to protect the key(s) whilst in storage.

(d) *Prevention of unauthorised access to the symmetric key(s):* This service can be provided by binding the symmetric key(s) to specified access control information. A protected storage mechanism can be used to ensure that the symmetric key(s) is/are only accessed when an execution environment on a specific platform is in a particular state and/or when valid authorisation data is provided, see, for example, section 1.6.7. Alternatively, the symmetric key(s) may be bound to a particular hardware component, such as a secure (co-)processor, so that the symmetric key(s) can only be decrypted inside that particular hardware component.

7. *Confidentiality and integrity protection of the application code and data during execution:* This service can be provided by using process isolation techniques. These are mechanisms that allow applications and services to run without interference from other processes executing in parallel, see section 1.6.8.

6.4 Conclusions

In this chapter we have described the two standard mechanisms that have been defined by the DVB organisation in order to ensure that an end user can acquire broadcast services from a variety of service providers using proprietary conditional access systems to protect the content. We have proposed the use of

reconfigurable mobile receivers in order to overcome the limitations which arise when the DVB standard solutions are applied in a mobile environment. The security threats relating to the secure delivery of the conditional access application and the secure storage and execution of the application on a mobile device have been defined. The security services and mechanisms required to thwart the threats highlighted have also been described.

Chapter 7

Protocols for secure application download

Contents

7.1 Introduction 248
7.2 Model 248
7.3 Prior art 250
7.4 Notation 252
7.5 Assumptions 253
7.6 Protocol initiation 257
7.7 Key exchange protocol 258
7.7.1 Protocol specification 258
7.7.2 Security analysis of the key exchange protocol 261
7.8 Key agreement protocol 264
7.8.1 Protocol specification 264
7.8.2 Security analysis of the key agreement protocol . . . 267
7.9 Conclusions 269

This chapter describes two protocols designed to meet the security requirements described in chapter 6. That is, these protocols support the protected download and execution of a conditional access system in a mobile environment.

7.1 Introduction

Having established the security threats pertaining to the secure download and execution of a conditional access application, two protocols are now described which have been designed to provide the necessary security services using a selection of the security mechanisms outlined in section 6.3.2. These protocols are not intended to supersede or replace the DVB standards or existing conditional access systems. Instead, they are intended to co-exist with existing mechanisms, so that the receipt of digital video broadcast may be achieved more efficiently in a mobile environment.

In section 7.2 the model under consideration and its associated entities are described. Section 7.3 describes current mechanisms which enable the secure download of applications. Section 7.4 details the notation used in the protocol descriptions, and section 7.5 outlines the assumptions upon which the protocols are based.

Section 7.6 describes the events culminating in the initiation of one of the two proposed download protocols. A key exchange protocol is specified in section 7.7 and an alternative key agreement protocol is described in section 7.8. Both protocols are accompanied by an analysis focusing on how well the security of the downloaded application is protected against threats 1 to 5, described in section 6.3.1.

7.2 Model

The model under consideration is illustrated in figure 7.1, and involves three parties: the user, who has a mobile receiver; the broadcaster; and the software provider. A fundamental component in this model is the trusted module. This tamper evident module is assumed to be bound either physically or crypto-

graphically to the mobile receiver, and is capable of performing a limited set of cryptographic operations.

In this model the mobile user does not need to have a long term relationship with the broadcaster, but is assumed to be aware of the content provision services that are available. Some of these services may be scrambled, in which case access is controlled by a conditional access system. For each scrambled service, the associated conditional access application must be acquired by the mobile receiver. One fundamental requirement for the application download protocol is that the mobile receiver is able to demonstrate to the software provider that it is a bona fide receiver and not in a malicious state that may facilitate the modification, replication, or extraction of secret data from the downloaded application. Once the receiver has proved itself to be trustworthy, the application is made available only to that receiver. The chosen application download protocol must also ensure that the confidentiality and integrity of the application is protected as it is transported to the mobile device.

The software provider in this model is required to supply the appropriate conditional access application to the mobile receiver. This software provider may, in practice, be the same entity as the broadcaster. Alternatively, it may be a third party broker. The mobile user needs to be aware of which software provider can deliver the required conditional access application. He may be informed of this by either the broadcaster or the software provider. The mobile receiver is then in a position to download the appropriate conditional access application from the software provider and descramble the broadcast service, subject to the relevant commercial agreements.

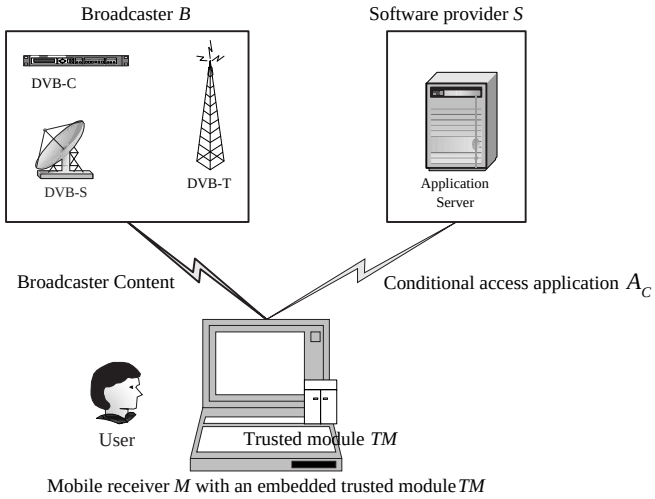

Figure 7.1: Architecture model

7.3 Prior art

One method widely adopted in order to protect mobile code is to digitally sign executables before they are distributed. Digital signatures are utilised in the security models for Java, as described in section 2.5.1.3, and MExE, as described in section 2.5.2, for example. Digital signatures are also utilised in order to protect agent code, data, and static state information in mobile agent systems, as described in part I of this thesis, see, for example, Safe-Tcl, as described in section 2.5.1.3, and D'agents and Telescript, as described in section 2.5.2. Digitally signing an executable enables the origin(s) of the executable to be authenticated and the integrity of the executable to be verified. Digitally signing an executable alone, however, does not fulfil the majority of the security services listed in section 6.3.2. While security services 2, 4 can be met, services 1, 3, 5, 6 and 7 cannot.

Another method currently used in order to enable the protected download of software is SSL/TLS, which makes use of TCP, or WTLS, which makes use of

UDP, to provide a reliable end-to-end secure tunnel [128]. Downloading software using SSL/TLS or WTLS enables the confidentiality and the integrity of the application code and data to be ensured; the origin of the downloaded executable to be corroborated; and mutual authentication of the software provider and the mobile device to be facilitated. The freshness of messages is also ensured. In this way, security services 1, 2, 3, 4 and 5, as described in section 6.3.2, can be met. If RSA is chosen as the key exchange method parameter in the cipher suite, the symmetric keys for integrity and confidentiality-protecting the downloaded application may be generated in a secure environment by the software provider, and securely distributed to the mobile device using the public key of the mobile receiver. In this way, security services 6a and 6b can be met. If, however, Diffie-Hellman is chosen as the key exchange method parameter in the cipher suite, the symmetric keys for integrity and confidentiality-protecting the downloaded application are generated on the mobile host as well as on the software provider. In this instance, while the symmetric keys do not need to be distributed, there is no guarantee that the keys are generated in a secure environment on the mobile host, thereby meeting security service 6a. Once the symmetric keys have been agreed during the handshake protocol, there is no requirement in SSL/TLS or WTLS that the keys are either confidentiality or integrity-protected while in storage on the host and, therefore, no requirement that access to the symmetric keys is controlled. Therefore, security services 6c and 6d cannot be met. Neither can security service 7 be met.

As described in section 6.1 Lie et al. and Suh et al. have designed hardware-based approaches to enable copy and tamper resistant software, namely XOM and AEGIS, respectively. XOM aims to enable the confidentiality and integrity-protection of software during download and while executing on the host machine. AEGIS aims to integrity-protect or integrity and confidentiality-protect software

during download and while executing on the host machine. Both proposals are examined, and analysed against the security requirements described in chapter 6, in chapter 9.

In the mobile agent domain a number of mechanisms have been proposed in order to prevent an attack against either the confidentiality and/or integrity of a mobile agent. Using code obfuscation, as described in section 2.5.3.3, the code is scrambled in such a way that no one is able to gain a complete understanding of its function, or to modify the resulting code without detection [91]. While a mechanism like this one could be considered as meeting security services 1, 2 and 7 and rendering security service 6 unnecessary, it seems that there is no known algorithm or approach for robustly implementing such an approach [91].

7.4 Notation

The following notation is used in the specification of the protocols:

M	denotes a mobile device or mobile receiver.
B	denotes a broadcaster.
S	denotes a software provider.
C	denotes a certification authority trusted by both M and S.
TM	denotes a trusted module bound to the mobile receiver M.
A_D	denotes an application, or agent, responsible for the secure download of a conditional access application.
A_B	denotes a broadcast application.
A_C	denotes a conditional access application to be downloaded to M.
Cert_X	is a public key certificate for entity X.
$K_{X,Y}$	denotes a secret key possessed only by X and Y.

R_X	is a random number issued by entity X.
$E_K(Z)$	is the result of the encryption of data Z using the key K.
$\text{Seal}_I(Z)$	is the result of the encryption of data Z concatenated with integrity metrics, I, such that Z can only be deciphered and accessed if the the platform is in a specified software state.
I	is a pair of integrity metrics (I_1, I_2), where I_1 specifies the state that the execution environment must be in for subsequent use of the protected object, and I_2 is the state of the execution environment at the time of command execution.
H	is a hash function, as defined in section 1.5.1.
$\text{MAC}_K(Z)$	is a Message Authentication Code, generated on data Z using key K.
$S_X(Z)$	is the digital signature of data Z computed using entity X's private signature transformation.
P_X	the public asymmetric key of X.
S_X	the private asymmetric key of X.
p	is a prime number.
g	is a generator for Diffie-Hellman key exchange modulo p, i.e. an element of multiplicative order q (a large prime dividing $p-1$) modulo p.
a_X	is entity X's Diffie-Hellman private key (i.e. a positive integer satisfying $a_X < q$).
b_X	is entity X's Diffie-Hellman public key for secret key generation $b_X = g^{a_X} \bmod p$.
Id_X	is an identifier for X.
$X\|\|Y$	is the result of the concatenation of data items X and Y in that order.

7.5 Assumptions

The following pre-conditions need to be satisfied for use of the protocols described later in this chapter.

1. There exists a certification authority C, trusted by both M and S. Both M and S possess a trusted copy of the public key of C, so that they can both verify certificates generated by C.

2. The designers of the relevant applications have agreed on the use of a specific protocol presented in sections 7.7 and 7.8, and have also agreed on all the necessary cryptographic algorithms and parameters.

3. A trusted module TM is inextricably bound to M. It is a self-contained processing module with specialist security capabilities such as random number generation, asymmetric key generation, digital signing, encryption capabilities, hashing capabilities, MACing capabilities, monotonic counters as well as memory, non-volatile memory, power detection and I/O. Support for platform integrity measurement, recording and reporting is also provided. One possible implementation of the trusted module is as a hardware chip, separate from the main platform CPU.

4. The mobile receiver M is running at least one protected execution environment. Within this environment, different applications run in isolation, free from being observed or compromised by other processes running in the same protected execution environment, or by software running in any insecure execution environment that may exist in parallel, see section 1.6.8.

5. The state of the platform has been measured and the integrity metrics which reflect it stored in the trusted module.

6. All secret keys required by the mobile receiver in the implementation of the protocols described below are protected by the trusted module, either directly or indirectly.

7. A unique asymmetric encryption key pair is associated with the trusted module.

8. The private decryption key from the pair referred to in point 7 is securely stored in the trusted module.

9. The public encryption key from the pair referred to in point 7 is certified. The certificate contains a general description of TM and its security properties.

10. Credentials have been generated indicating whether the particular design of the trusted module TM in a particular class of mobile platform (to which M conforms) meets specified security requirements.

11. A credential has been generated indicating whether the particular mobile receiver M which incorporates TM is an instance of a certified class of trusted mobile platform, as referred to in point 10.

12. The trusted module TM possesses a signature key pair, used only for entity authentication.

13. The private signing key from the pair referred to in point 12 is securely stored by the trusted module.

14. The public signature verification key from the pair referred to in point 12 is certified by C. The certificate issued, Cert_{TM}, binds the identity of TM (the trusted platform containing TM) to a public key used for the verification of digital signatures. This certificate must be obtainable by the software provider S.

15. The software provider S possesses a signature key pair, used only for entity authentication.

16. The private signing key from the pair referred to in point 15, is securely stored by the software provider.

17. The software provider S has a certificate, Cert_S, issued by C. This certificate associates the identity of S with the public verification key from the pair referred to in point 15. This certificate must be available to the mobile receiver.

18. If the key agreement protocol specified in section 7.8 is to be used, all parties must agree on the Diffie-Hellman system parameters g and p. This

could be achieved by including these values in the relevant certificates, or by hard-coding them into relevant application software. Alternatively, the Diffie-Hellman parameters could be standardised as has been done for the Internet key exchange protocol version 2 in the IPSec architecture.

19. The trusted module is capable of generating an asymmetric encryption key pair, where the public encryption key can be signed using the signature key described in assumption 12. This thwarts the privacy and security threats surrounding routine use of the public encryption key described in assumption 7. The private decryption key from this pair is bound to a particular platform configuration.

20. S is able to verify the configuration-related claims made by the platform containing a particular trusted module. S is able to look up, or obtain from a validation authority, the integrity measurement value that should be obtained if a platform component is working as intended, or the set of platform state integrity metrics that should be obtained if a platform is working as intended [5].

21. Every mobile device wishing to receive a video broadcast has a trusted broadcast application, A_B, running in a protected execution environment.

22. Every mobile device has a download application, A_D, running in the same protected execution environment as A_B. This download application will perform two fundamental tasks. Firstly, it will complete one of the protocols described in sections 7.7 and 7.8. Secondly, once A_C is executing in a protected execution environment, A_D must prevent any interference by another application. It may, for example, incorporate a monitoring function which adheres to a specified policy, such that once the conditional access application is running on the device, any attempt by another ap-

plication to start up will fail. Alternatively, the start-up of any additional applications will result in A_D stopping A_C, and erasing it from memory.

23. A_B and A_D execute within a protected execution environment. A_C will also execute within this protected execution environment once it has been downloaded.

24. The attacker's behaviour is bounded by the assumption of perfect encryption [40].

7.6 Protocol initiation

Both protocols begin when the user makes a request to the broadcast application A_B to view a specific video broadcast. If consumption of this broadcast is controlled by a particular conditional access application, A_B completes the following process:

1. A_B checks to see if the mobile device has dedicated hardware or software installed to support the specific conditional access system.

2. If no dedicated hardware, for example a common interface module, is present on the mobile device, then A_B determines whether A_C has previously been downloaded and is still available in secure storage.

   (a) If so, the download application A_D is called to retrieve A_C from secure storage and execute the application.

   (b) If A_C is not available on the mobile device, then A_D is called to download the application. The download of A_C is accomplished by deploying one of the protocols described in sections 7.7 and 7.8.

It is presupposed that a protocol run is completed every time a conditional access application is to be downloaded, so that either the asymmetric encryption

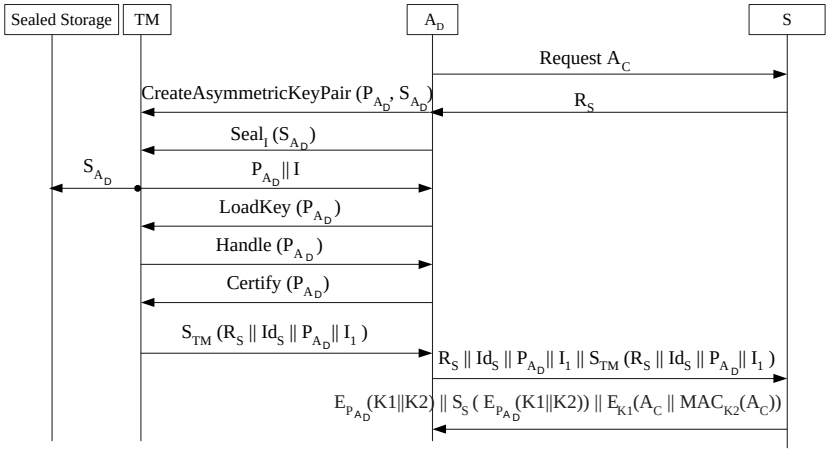

Figure 7.2: Key exchange protocol

key pair generated in the key exchange protocol described in section 7.7, or the Diffie-Hellman key agreed in the key agreement protocol described in section 7.8, is unique to a protocol run.

7.7 Key exchange protocol

In this section, a key exchange protocol is specified. The protocol is then analysed in terms of how the conditional access application is protected while in transit between the software provider and the mobile receiver.

7.7.1 Protocol specification

The key exchange protocol is shown in figure 7.2 and consists of the following sequence of steps, where $X \rightarrow Y : Z$ is used to indicate that the message Z is sent by entity X to entity Y.

1. $A_D \rightarrow S$: Request for A_C.

2. S : Generates a random value R_S, and stores it for subsequent freshness checking of received data. R_S should be chosen in such a way that the

probability of the same value ever being used twice by S is negligible. The random number must also be unpredictable to a third party.

3. $S \rightarrow A_D : R_S$.

4. A_D : Stores R_S.

5. $A_D \rightarrow TM$: Requests the generation of an asymmetric encryption key pair, P_{A_D} and S_{A_D}, and the sealing of S_{A_D} to a set of integrity metrics (I), i.e. $\text{Seal}_I(S_{A_D})$. These integrity metrics should reflect the state that the protected execution environment must be in if subsequent use of the private key S_{A_D} is to be permitted, I_1, and also the state of the protected environment at the time of key generation, I_2, i.e. $I = I_1 \mid\mid I_2$.

6. TM : Generates P_{A_D} and S_{A_D}, and seals S_{A_D} to I.

7. $TM \rightarrow A_D : P_{A_D} \mid\mid I$.

8. A_D : Keeps a record of the integrity metrics I_2 to which S_{A_D} was bound, and what I_2 represents.

9. $A_D \rightarrow TM$: Request to load the key pair.

10. TM : Loads the key pair.

11. TM : Outputs a handle to the loaded key pair.

12. $A_D \rightarrow TM$: Request to certify P_{A_D}. In conjunction with this request, A_D sends R_S and Id_S to the TM.

13. TM : Signs R_S, Id_S, P_{A_D} and I_1, where R_S is included so that the freshness of the signature can be checked by the software provider, and Id_S is included so that the intended destination of the message can be verified by the software provider,
    $S_{TM}(R_S \mid\mid Id_S \mid\mid P_{A_D} \mid\mid I_1)$.

14. $TM \rightarrow A_D : R_S \mid\mid Id_S \mid\mid P_{A_D} \mid\mid I_1 \mid\mid S_{TM}(\ R_S \mid\mid Id_S \mid\mid P_{A_D} \mid\mid I_1\ )$.

15. $A_D \rightarrow S : R_S \mid\mid Id_S \mid\mid P_{A_D} \mid\mid I_1 \mid\mid S_{TM}(\ R_S \mid\mid Id_S \mid\mid P_{A_D} \mid\mid I_1\ )$.

16. S : Retrieves Cert_{TM} and verifies it.

17. S : Verifies $S_{TM}(\ R_S \mid\mid Id_S \mid\mid P_{A_D} \mid\mid I_1\ )$ using the public signature verification key of TM contained in Cert_{TM}.

18. S : Verifies R_S against the value generated and stored in step 2 to ensure that the message is fresh.

19. S : Verifies that the message was intended for S through examination of the identifier Id_S.

20. S : Decides if I_1 represents a sufficiently trustworthy state.

21. Assuming the signature from TM can be verified, the values of R_S and Id_S are as expected, and the integrity metrics, I_1, are acceptable, then:
    S: Extracts P_{A_D}.

22. S : Generates secret keys $K1_{S,A_D}$ and $K2_{S,A_D}$ used for data encryption and data integrity, respectively.

23. S : Computes a MAC on, and then encrypts, A_C,
    $E_{K1_{S,A_D}}(A_C \mid\mid \text{MAC}_{K2_{S,A_D}}(A_C))$.

24. S : Encrypts the MACing and encryption keys used in step 23 with P_{A_D}, the public encryption key of the TM,
    $E_{P_{A_D}}(K1_{S,A_D} \mid\mid K2_{S,A_D})$.

25. S : Signs the encrypted bundle from step 24,
    $S_S(E_{P_{A_D}}(K1_{S,A_D} \mid\mid K2_{S,A_D}))$.

26. $S \rightarrow A_D : E_{P_{A_D}}(K1_{S,A_D} \mid\mid K2_{S,A_D}) \mid\mid$
    $S_S(E_{P_{A_D}}(K1_{S,A_D} \mid\mid K2_{S,A_D})) \mid\mid E_{K1_{S,A_D}}(A_C \mid\mid \text{MAC}_{K2_{S,A_D}}(A_C))$.

27. A_D : Retrieves Cert_S and verifies it.

28. A_D : Verifies $S_S(E_{P_{A_D}}(K1_{S,A_D} \mid\mid K2_{S,A_D}))$ using the public signature verification key of S contained in Cert_S.

29. TM : Decrypts $E_{P_{A_D}}(K1_{S,A_D} \mid\mid K2_{S,A_D})$, if the platform is in the agreed state, I_1.

30. A_D: Compares I_2 to its record of I_2 to which S_{A_D} was bound in step 6, to ensure that the request for key pair generation came from A_D.

31. A_D : Decrypts $E_{K1_{S,A_D}}(A_C \mid\mid \text{MAC}_{K2_{S,A_D}}(A_C))$.

32. A_D : Verifies $\text{MAC}_{K2_{S,A_D}}(A_C)$.

33. Once A_C is executing, A_D precludes the potential interference of any other application with A_C.

34. A_D : Deletes A_C, and all other keys, when they are no longer required. The encrypted copy of A_C may remain stored for future use, space permitting.

7.7.2 Security analysis of the key exchange protocol

The analysis completed here focuses upon how well the conditional access application is protected against threats 1 to 5, described in section 6.3.1. Protection of the application against threats 6 and 7, described in section 6.3.1, will depend on the functionality of the particular type of trusted module embedded in the platform and, more generally, on the overall computing architecture of the mobile receiver. Consequently, how well the application is protected against threats 6 and 7 is analysed in chapter 8, following an exploration of the possible implementations of the generic key exchange and key agreement protocols on a variety of trusted platform architectures.

1. *Confidentiality of the application code and data:*

   Symmetric encryption is deployed to protect the confidentiality of A_C. The confidentiality of A_C is also dependent, however, on the confidentiality of $K1_{S,A_D}$ being protected. How well $K1_{S,A_D}$ is protected is analysed in chapter 8.

2. *Integrity protection of the application code and data:*

   A MAC is deployed to protect the integrity of A_C. The integrity of A_C is also dependent, however, on the confidentiality and integrity of $K2_{S,A_D}$ being protected. How well $K2_{S,A_D}$ is protected is analysed in chapter 8.

3. *Entity authentication:*

   The software provider can authenticate the trusted platform by verifying the signature of TM on R_S, Id_S, P_{A_D} and I_1. Steps 3 and 15 of the above protocol conform to the two pass unilateral authentication protocol described in clause 5.1.2 of ISO/IEC 9798-3:1998 [85], where P_{A_D} serves as the nonce in the response message sent by A_D, by virtue of the fact that an asymmetric key pair is generated for each protocol run.

   It may be argued that the protocol outlined above also provides entity authentication of the software provider to the mobile platform. If P_{A_D} is unique to the protocol run, P_{A_D} acts not only as a random nonce, but also serves to represent the identity of the destination platform. The signature of the software provider on the unique public key, P_{A_D}, in step 26, or more specifically $S_S(E_{P_{A_D}}(K1_{S,A_D} \| K2_{S,A_D}))$, allows the identity of the software provider to be authenticated by the mobile receiver.

   Alternatively, one of the following additions may be made to the protocol. A random nonce may be included in the signed bundle sent to the software provider in step 15, and returned in conjunction with R_S and Id_M in the

bundle signed by the software provider in step 26. If this modification is made to the protocol, steps 3, 15, and 26 would conform to the three pass mutual authentication protocol described in clause 5.2.2 of ISO/IEC 9798-3:1998 [85].

Instead of this, a timestamp, in conjunction with the identifier Id_M, could be included in the signed bundle from step 26. If this modification is made, step 26 would conform to the one pass unilateral authentication protocol as described in clause 5.1.1 of ISO/IEC 9798-3:1998 [85].

4. *Origin authentication:*

   Since S signs $K1_{S,A_D}$ and $K2_{S,A_D}$, M is able to verify that these keys have been sent from S. As $K2_{S,A_D}$ is used to compute the MAC on A_C, M can verify that A_C has been sent from the same source. An attacker attempting to deliver a malicious application would require the collaboration of S.

5. *Freshness:*

   It may be possible for an attacker to replace the message in step 26 with an older message destined for the same mobile host, or with a corresponding message destined for a different mobile host. However, since a unique public key P_{A_D} is generated for each protocol run, the verification in step 32 would detect this.

   Alternatively, one of the following additions may be made to the protocol. A random nonce could be included in the signed bundle sent to the software provider in step 15 and returned in the bundle signed by the software provider in step 26. Alternatively, a timestamp could be incorporated into the message sent in step 26.

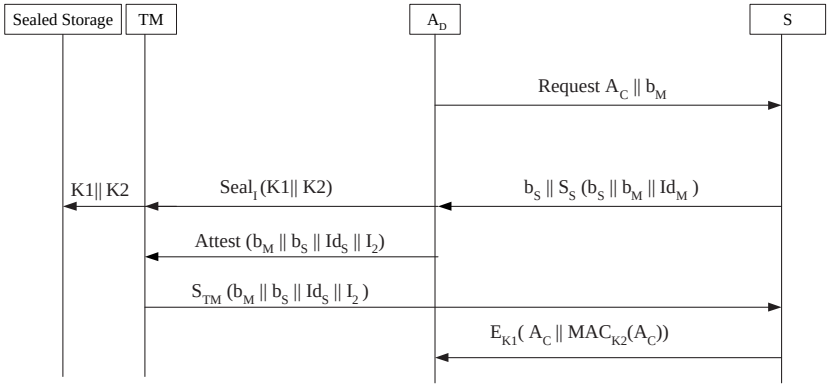

Figure 7.3: Key agreement protocol

7.8 Key agreement protocol

In this section, an alternative protocol based upon the Diffie-Hellman key agreement protocol is specified. This protocol is shown in figure 7.3. As before, the protocol begins when A_D executing on the mobile device makes a request for a conditional access application to be downloaded. This protocol is analysed against the same criteria as the key exchange protocol described in section 7.7.

7.8.1 Protocol specification

1. A_D : Chooses a Diffie-Hellman private value a_M and calculates b_M based on g and p, where these latter values may have been hard-coded into the relevant application software or, alternatively, may be retrieved from the relevant certificates.

2. $A_D \rightarrow S$: Request for $A_C \mid\mid b_M$.

3. S : Chooses a Diffie-Hellman private value a_S and calculates b_S using the same values of g and p that were used by A_D in step 1.

4. S : Signs the concatenation of b_S, its public Diffie-Hellman key, b_M, the public Diffie-Hellman key of A_D, and Id_M, the identity of M, to whom

the message is being sent,

$S_S(b_S \mid\mid b_M \mid\mid Id_M)$.

5. $S \rightarrow A_D : b_S \mid\mid S_S(b_S \mid\mid b_M \mid\mid Id_M)$.

6. A_D : Retrieves Cert_S and verifies it.

7. A_D : Verifies the signature on the message received in step 5 using the public signature verification key of S contained in Cert_S.

8. A_D : Verifies that b_M has been returned, thereby indicating that the message is fresh.

9. A_D : Verifies that Id_M is contained within the signed message, to ensure that the message was destined for M.

10. A_D : Calculates the shared key K_{S,A_D} using b_S, the public Diffie-Hellman value of S, and a_M, the private Diffie-Hellman value of A_D.

11. A_D : Generates secret keys $K1_{S,A_D}$ and $K2_{S,A_D}$ used for data encryption and data integrity, respectively, from K_{S,A_D} by an agreed method.

12. $A_D \rightarrow TM$: Request the encryption of $K1_{S,A_D} \mid\mid K2_{S,A_D}$ and their association with a specified protected execution environment state, I, which consists of I_1, the state the protected execution environment must be in before access to $K1_{S,A_D} \mid\mid K2_{S,A_D}$ is permitted, and I_2, which represents the current state of the protected execution environment at the time the keys were encrypted,

    $\text{Seal}_I(K1_{S,A_D} \| K2_{S,A_D})$.

13. TM : Securely stores $K1_{S,A_D}$ and $K2_{S,A_D}$ using the protected storage mechanism, such that $K1_{S,A_D}$ and $K2_{S,A_D}$ can only be decrypted when the protected environment is in a specified state I_1, i.e. TM seals $K1_{S,A_D}$ and $K2_{S,A_D}$.

14. A_D : Keeps a record of I_2 to which the sealed keys were bound and what I_2 represents.

15. $A_D \rightarrow TM$: Request platform attestation, i.e. the signature of TM on $(b_M \,||\, b_S \,||\, Id_S \,||\, I_2\,)$, where I_2 represents the current state of the protected execution environment.

16. $TM \rightarrow A_D : S_{TM}(b_M \,||\, b_S \,||\, Id_S \,||\, I_2\,)$.

17. $A_D \rightarrow S : S_{TM}(b_M \,||\, b_S \,||\, Id_S \,||\, I_2\,)$.

18. S : Obtains and verifies Cert_{TM}.

19. S : Verifies $S_{TM}(b_M \,||\, b_S \,||\, Id_S \,||\, I_2\,)$ using the public signature verification key of TM contained in Cert_{TM}.

20. S : Verifies b_S to ensure that the message is fresh.

21. S : Verifies Id_S to ensure that the message was destined for S.

22. S : Checks that the message has come from a trustworthy module, and checks, using I_2, that the state information represents a sufficiently trustworthy execution environment.

23. S : Calculates the shared key K_{S,A_D} using b_M, the public Diffie-Hellman value of A_D, and a_S, the private Diffie-Hellman value generated by S.

24. S : Derives secret keys $K1_{S,A_D}$ and $K2_{S,A_D}$ used for data encryption and data integrity, respectively, from K_{S,A_D} by an agreed method.

25. S : Computes a MAC on, and then encrypts, A_C, $E_{K1_{S,A_D}}(A_C \,||\, \text{MAC}_{K2_{S,A_D}}(A_C))$.

26. $S \rightarrow A_D : E_{K1_{S,A_D}}(A_C \,||\, \text{MAC}_{K2_{S,A_D}}(A_C))$.

27. $A_D \rightarrow TM$: Request to unseal $K1_{S,A_D}$ and $K2_{S,A_D}$.

    $K1_{S,A_D}$ and $K2_{S,A_D}$ can only be unsealed by the TM if the platform is in the agreed state, I_1.

28. A_D: Compares I_2 to its record of the value of I_2 to which the symmetric keys were bound in step 13, to ensure that the request for sealing the symmetric keys came from A_D.

29. A_D : Decrypts $E_{K1_{S,A_D}}(A_C \mid\mid \text{MAC}_{K2_{S,A_D}}(A_C))$.

30. A_D : Verifies $\text{MAC}_{K2_{S,A_D}}(A_C)$.

31. Once A_C is executing, A_D precludes the potential interference of any other application with A_C.

32. A_D : Deletes A_C, and all other keys, when they are no longer required.

    The encrypted copy of A_C may remain stored for future use, space permitting.

7.8.2 Security analysis of the key agreement protocol

As before, the analysis completed here focuses upon how well the conditional access application is protected against threats 1 to 5, described in section 6.3.1.

1. *Confidentiality of the application code and data:*

   Symmetric cryptography is deployed to protect the confidentiality of of A_C. The confidentiality of A_C is also dependent, however, on the confidentiality of $K1_{S,A_D}$ being protected. How well $K1_{S,A_D}$ is protected is analysed in chapter 8.

2. *Integrity protection of the application code and data:*

   As was the case in the key exchange protocol, a MAC is deployed to protect the integrity of A_C. The integrity of A_C is also dependent, however, on

the confidentiality and integrity of $K2_{S,A_D}$ being protected. How well $K2_{S,A_D}$ is protected is analysed in chapter 8.

3. *Entity authentication:*

   As this protocol is based on the station-to-station protocol [38] (a type of authenticated Diffie-Hellman protocol) it offers mutual authentication. The mobile device can authenticate the software provider's identity via the verification of the digital signature which is generated over b_S, b_M and Id_M and sent by S in step 5. The software provider can verify the identity of the TM and the state of the platform's protected execution environment using the trusted computing based platform attestation mechanism, see section 1.6.6. This is achieved through the verification of TM's signature upon b_M, b_S, Id_S and the integrity metrics representative of the protected execution environment, sent in step 17.

4. *Origin authentication:*

   Since S signs b_M and b_S in step 5, and A_C is protected using keys derived from b_M and b_S, M is able to verify the origin of A_C. As before, an attacker attempting to deliver a malicious application would require the collaboration of S.

5. *Freshness:*

   The attacker could replace the message in step 26 with an older message destined for the same mobile host, or with a corresponding message destined for a different mobile host. However, since a unique secret key K_{S,A_D} is agreed for each protocol run, the verification in step 30 would detect this. As an additional measure, if deemed necessary, a timestamp could be included in step 26 to prevent replay.

7.9 Conclusions

In this chapter, we have described two protocols which support the download of a conditional access application to a mobile device. Both the key exchange protocol, specified in section 7.7.1, and the key agreement protocol, specified in section 7.8.1, ensure that the conditional access application is protected against threats 1 to 5, described in section 6.3.1. This has been demonstrated through the security analysis completed in sections 7.7.2 and 7.8.2.

Chapter 8

Protocol implementation using trusted computing frameworks

Contents

8.1 Introduction 272
8.2 Notation 272
8.3 Implementing the protocols using the TCG specifications 273
8.3.1 Key exchange protocol 276
8.3.2 Key agreement protocol 286
8.3.3 Implementation specific security analysis 291
8.4 Implementing the protocols using the TCG specification set and an integrated isolation kernel . . . 296
8.4.1 Key exchange protocol 297
8.4.2 Key agreement protocol 298
8.4.3 Implementation specific security analysis 298
8.5 Implementing the protocols using NGSCB 302
8.5.1 Key exchange protocol 303
8.5.2 Key agreement protocol 304
8.5.3 Implementation specific security analysis 304
8.6 Conclusions 307

This chapter explores three possible implementations of the generic key exchange and key agreement protocols described in chapter 7. The first implementation assumes the presence of a mobile device into which components described in the TCG version 1.2 specification set are integrated. Following this, we exam-

ine the implementation of the protocols given a mobile device architecture into which a version 1.2 compliant TPM and CRTM are integrated and an isolation layer deployed. Finally, protocol implementation given an NGSCB compliant platform, as described by Microsoft, is explored. Each implementation description is accompanied by an analysis which examines how well the security of the downloaded application is protected against threats 6 and 7 from section 6.3.1.

8.1 Introduction

Chapter 7 contains a description of two secure application download protocols. This chapter considers how three different trusted computing architectures may be utilised in the implementation of the protocols. Development of the TCG's specification set and Microsoft's NGSCB represent major industry initiatives in the field of trusted computing, and it is important that the protocols proposed in chapter 7 can be implemented using these technologies, if they are to find practical application.

Section 8.2 details the notation used in the protocol implementation descriptions. Section 8.3 illustrates how the generic application download protocols may be implemented using a TCG version 1.2 compliant TPM and CRTM. Section 8.4 describes how the protocols may be implemented on a platform supporting not only a version 1.2 compliant TPM and CRTM, but also an isolation layer, while section 8.5 examines how the protocols may be implemented on an NGSCB compliant platform. Finally, for each of the three implementation options explored, an analysis is given of how well the security of the downloaded application is protected against threats 6 and 7, from section 6.3.1.

8.2 Notation

The following notation is used throughout this chapter, together with some of the notation given in section 7.4:

TPM	denotes a version 1.2 compliant trusted platform module bound to the mobile receiver M.
R	denotes a CRTM bound to the mobile receiver M.
P	denotes a privacy certification authority (privacy-CA) trusted by both M and S.

8.3 Implementing the protocols using the TCG specifications

In this section we investigate how the generic protocols described in chapter 7 can be mapped to a platform into which a version 1.2 compliant TPM and a CRTM have been integrated. How the generic key exchange protocol, defined and analysed in section 7.7, can be implemented using a TCG version 1.2 compliant system is examined in section 8.3.1. The implementation of the generic key agreement protocol, described and analysed in section 7.8, using a platform into which a version 1.2 compliant TPM and a CRTM have been integrated is explored in section 8.3.2.

In chapter 7, some generic assumptions were made about the software provider, the mobile receiver, the trusted module embedded within the mobile receiver, and the download application. In this section, the trusted module, introduced in chapter 7, is mapped to a version 1.2 compliant TPM and a CRTM, as specified by the TCG. The assumptions pertaining to the trusted module and the architecture of the mobile receiver defined in section 7.5 are re-examined here in view of this mapping.

TPM, a TCG version 1.2 compliant TPM, is a tamper resistant self-contained processing engine, inextricably bound to M, with specialist capabilities such as random number generation, asymmetric key generation, digital signing, encryption capabilities, a SHA-1 engine, a HMAC engine, monotonic counters, as well as volatile and non-volatile memory, power detection and I/O. The RTM, also inextricably bound to M, is a computing engine which accurately generates at least one integrity measurement event representing a software component running on the platform. For the foreseeable future, it is envisaged that the RTM will be integrated into the normal computing engine of M with minimum pro-

tection, where additional instructions (i.e. the CRTM) are integrated into the platform's BIOS boot block or BIOS and cause the main platform processor to function as the RTM. The CRTM, R, may, however, be part of TPM. See section A.6 for further details.

A unique asymmetric encryption key pair is associated with TPM, called an endorsement key pair, see section A.7.4. The endorsement key pair is used only for encryption/decryption purposes. The private endorsement key is protected within a TPM shielded location, see section A.7.1. The public endorsement key is certified by a trusted platform management entity, see section A.5, in an endorsement credential, see section A.10.1. A set of credentials including the endorsement credential, in conjunction with conformance credentials and a platform credential, are also generated for M, which we assume incorporates TPM, see section A.10.

TPM has at least one identity and at least one attestation identity key (AIK) pair associated with it, see section A.10.4. The private AIK is securely stored by TPM and may be used both to sign attestation statements and to certify non-migratable keys, see section A.13.1, generated within TPM. The public key from this AIK pair is certified by a privacy-CA P in the form of an AIK credential Cert_{TPM}, see section A.10.4. P is trusted by S.

TPM is also capable of generating an asymmetric key pair on demand, of which the private key may be cryptographically linked to a set of integrity metrics. The private key from such a pair is securely stored by TPM, and can only be used when the platform is in a specified state. A newly generated public key may be certified using the public AIK described in the previous paragraph.

TPM also provides secure storage and, more specifically, sealing, see section A.13. This capability may be used to encrypt and store any symmetric

keys that are generated on the platform, and to ensure that access to these symmetric keys is only permissible when the platform is in a specified state.

Finally, *TPM* is capable of signing 160 bits of external data, in conjunction with I_2, see section 7.4, thereby providing a platform attestation statement that can be verified by S, see section A.11. Validation certificates can be generated and made available so that a challenger of the platform, such as S, can validate the configuration of the trusted platform's software environment. Alternatively, a trusted third party may be used in the validation of a challenged platform's software configuration, see section A.11.4.

The TCG 1.2 specifications do not, however, specify any components which can be used in the implementation of isolated software domains or compartments on a platform. In this scenario, it is therefore assumed that multiple isolated domains or compartments do not necessarily exist on the platform. The 'protected execution environment' which is assumed in chapter 7, can only be constructed in a platform of this nature through the deployment of a trusted operating system (which has been measured, and which has measurement capabilities) and the enforcement of rigorous restrictions on the execution of software. It is assumed that the state of the platform has been measured and the integrity metrics which reflect it stored by *TPM*, see section A.11.

In defining the protocols, use of the version 1.2 TPM command set [158] and data structures [157] is implied. TPM commands used include *TPM_CreateWrapKey* ; *TPM_CertifyKey* or *TPM_CertifyKey2* depending on the properties of the key to be certified and the certifying key; *TPM_Quote* or *TPM_Quote2*; and *TPM_Seal*. The data structures used include *TPM_Key*, which uses the *TPM_PCR_INFO* to define the platform configuration registers (PCRs) in use, or *TPM_Key12*, which uses *TPM_PCR_INFO_LONG* structure

to more fully define the PCRs in use; and *TPM_Certify_Info* or *TPM_Certify_Info2*.

8.3.1 Key exchange protocol

Here we describe how the key exchange protocol defined in section 7.7 can be implemented using a TCG version 1.2 compliant system. We assume a basic TPM key hierarchy, for example, that described in figure 8.1, which contains a TPM storage root key (SRK), an attestation identity key, which has been generated and activated (AIK), and a storage key, i.e. a conditional access application key ($CAAK$). We assume, prior to the initiation of the protocol described below, that all three of these keys are in existence and that the SRK, the $CAAK$ and the attestation identity key, AIK, are loaded in TPM.

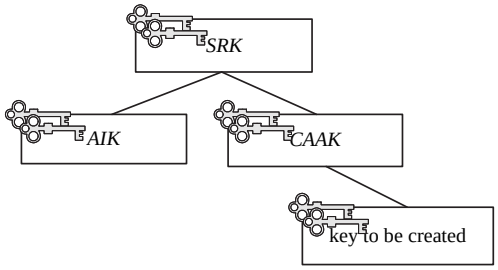

Figure 8.1: Key hierarchy for the download of A_C

The numbered protocol steps described below map directly to the generic key exchange protocol steps described in section 7.7. TCG-specific functionality is used in steps 5 to 17. In steps 5 to 8, *TPM_CreateWrapKey* functionality is used. In steps 9 to 11, *TPM_LoadKey2* functionality is deployed. Finally, in steps 12 to 17, *TPM_CertifyKey* functionality is used. Note that further details of the TPM commands and structures used in the protocol are given in the text

following the numbered steps.

1. $A_D \rightarrow S$: Request for A_C.
2. S : Generates a random value R_S, and stores it for subsequent freshness checking of received data. R_S should be chosen in such a way that the probability of the same value ever being used twice by S is negligible. The random number generated must also be unpredictable to a third party.
3. $S \rightarrow A_D : R_S$.
4. A_D : Stores R_S.
5. $A_D \rightarrow TPM$: *TPM_CreateWrapKey*.
6. TPM : Generates P_{A_D} and S_{A_D}, and binds S_{A_D} to I_1 and I_2, (the concatenation of which is referred to as I).
7. $TPM \rightarrow A_D$: *TPM_Key*, which contains P_{A_D}, an encrypted S_{A_D}, and I.
8. A_D : Keeps a record of I_2 to which S_{A_D} was bound, the list of selected target PCRs, and the selected PCR values.
9. $A_D \rightarrow TPM$: *TPM_LoadKey2* — Request to load *TPM_Key*.
10. TPM : Loads *TPM_Key*.
11. TPM : Outputs a handle to the loaded *TPM_Key*.
12. $A_D \rightarrow TPM$: *TPM_CertifyKey*. The hash of $R_S \mid\mid Id_S$ is sent to the TPM as an input parameter to this command.
13. TPM : Signs $H(P_{A_D})$, $H(R_S \mid\mid Id_S)$, and I_1, where $H(R_S \mid\mid Id_S)$ is included so that the freshness of the signature can be checked by the software provider, and so that the intended destination of the message

can be verified by the software provider,

$S_{TPM}(H(P_{A_D}) \mid\mid H(R_S \mid\mid Id_S) \mid\mid I_1)$.

14. $TPM \rightarrow A_D$: *TPM_Certify_Info* $\mid\mid S_{TPM}(H(P_{A_D}) \mid\mid H(R_S \mid\mid Id_S) \mid\mid I_1)$.

15. $A_D \rightarrow S : R_S \mid\mid Id_S \mid\mid P_{A_D} \mid\mid$ *TPM_Certify_Info* $\mid\mid$
$S_{TPM}(H(P_{A_D}) \mid\mid H(R_S \mid\mid Id_S) \mid\mid I_1)$.

16. S : Retrieves Cert_{TPM} and verifies it.

17. S : Verifies *TPM_Certify_Info* and $S_{TPM}(H(P_{A_D}) \mid\mid H(R_S \mid\mid Id_S) \mid\mid I_1)$
using the public signature verification key of the TPM contained in Cert_{TPM}.

18. S : Verifies R_S against the value generated and stored in step 2, to ensure that the message is fresh.

19. S : Verifies that the message was intended for S through the examination of Id_S.

20. S : Decides if I_1 represents a sufficiently trustworthy state.

21. Assuming the signature, R_S, Id_S and I_1 are acceptable,
S: Extracts P_{A_D}.

22. S : Generates secret keys $K1_{S,A_D}$ and $K2_{S,A_D}$, used for data encryption and data integrity, respectively.

23. S : Computes a MAC on, and then encrypts, A_C,
$E_{K1_{S,A_D}}(A_C \mid\mid \text{MAC}_{K2_{S,A_D}}(A_C))$.

24. S : Encrypts the MACing and encryption keys used in step 23 with P_{A_D}, the public encryption key of TPM,
$E_{P_{A_D}}(K1_{S,A_D} \mid\mid K2_{S,A_D})$.

25. S : Signs the encrypted bundle from step 24,
$S_S(E_{P_{A_D}}(K1_{S,A_D} \mid\mid K2_{S,A_D}))$.

26. $S \rightarrow A_D : E_{P_{A_D}}(K1_{S,A_D} \mid\mid K2_{S,A_D}) \mid\mid$
$S_S(E_{P_{A_D}}(K1_{S,A_D} \mid\mid K2_{S,A_D})) \mid\mid E_{K1_{S,A_D}}(A_C \mid\mid \text{MAC}_{K2_{S,A_D}}(A_C))$.

27. A_D : Retrieves Cert_S and verifies it.

28. A_D : Verifies $S_S(E_{P_{A_D}}(K1_{S,A_D} \mid\mid K2_{S,A_D}))$ using the public signature verification key of S contained in Cert_S.

29. TPM : Decrypts $E_{P_{A_D}}(K1_{S,A_D} \mid\mid K2_{S,A_D})$.
Use of the corresponding private key, and therefore decryption of the shared symmetric keys, will only be completed if the current state of the platform software environment is reflected by the integrity metrics, I_1, to which S_{A_D} was bound in step 6. Authorisation data may also be required, depending on the value of the *TPM_Auth_Data_Usage* field set in step 5.

30. A_D: Compares I_2 to its record of I_2 to which S_{A_D} was bound in step 6, to ensure that the request for key pair generation came from A_D.

31. A_D : Decrypts $E_{K1_{S,A_D}}(A_C \mid\mid \text{MAC}_{K2_{S,A_D}}(A_C))$.

32. A_D : Verifies $\text{MAC}_{K2_{S,A_D}}(A_C)$.

33. Once A_C is executing, A_D precludes the potential interference of any other application with A_C.

34. A_D : Deletes A_C, and all other keys, when they are no longer required.
The encrypted copy of A_C may remain stored for future use, space permitting.

8.3.1.1 Steps 5 to 8

The *TPM_CreateWrapKey* command is used in step 5 of the protocol to instruct TPM to generate an asymmetric key pair P_{A_D} and S_{A_D}. The input parameters associated with the *TPM_CreateWrapKey* command include information about

the key-to-be-created, i.e. the TPM structure version, the operations to be permitted with the key, an indication of whether the key-to-be-created should be migratable, the parameters used to generate the key, the PCRs to which the key-to-be-created is to be bound, and the conditions in which it is required that authorisation data is to be presented for use of the key-to-be-created. Input of the parent wrapping key usage authorisation data may also be required. Encrypted usage authorisation data and/or migration authorisation data for the key-to-be-created may also be input. For this particular use case we require that the key-to-be-created is non-migratable. This implies that the key cannot be migrated from the TPM in which it is created.

Alternatively, a certifiable migratable key could be created using the *TPM_CMK_CreateKey* command instead of the *TPM_CreateWrapKey* command. A certifiable migratable key is one which may be certified by *TPM* and migrated, but only under strict controls. This prohibits the key protecting the conditional access application from being migrated to an arbitrary platform authorised by the owner of *TPM*, but permits its migration to selected devices, e.g. other TPMs owned by the same entity. Before key migration, the key owner must authorise the migration transformation. The migration destination must also be authorised, not only by the owner of *TPM*, but also by a migration selection authority. This authority could, for example, be the trusted download agent, A_D, or, alternatively, the software provider, S. We focus, however, on the case where the key to be created is non-migratable and generated using the *TPM_CreateWrapKey* command.

In response to the *TPM_CreateWrapKey* command, *TPM* returns either a *TPM_Key* or a *TPM_Key12* data structure (note that the above protocol description assumes the former). Both data structures contain the created public key, P_{A_D}, and the encrypted private key, S_{A_D}. Both data structures also iden-

tify the operations permitted with the key and contain a flag indicating whether or not the key is migratable. Both data structures may also identify the platform configuration (the PCR info) to which S_{A_D} is bound. The *TPM_Key* and *TPM_Key12* data structures differ only in the way in which the PCR info parameter is described.

If a *TPM_Key* data structure is returned from the *TPM_CreateWrapKey* command, a *TPM_PCR_INFO* structure will describe the platform configuration to which the key is bound. A *TPM_PCR_INFO* structure contains three fields:

- pcrSelection, which indicates the selected PCRs to which the key is bound;
- digestAtRelease, which is the digest of the PCR indices and PCR values which must be verified when using the key bound to the PCRs; and
- digestAtCreation, which is the digest value of the selected PCR values at the time of key creation.

Alternatively, if a *TPM_Key12* data structure is returned from the *TPM_CreateWrapKey* command, a *TPM_PCR_INFO_LONG* structure will describe the platform configuration to which the key is bound. A *TPM_PCR_INFO_LONG* structure contains six main fields:

- localityAtCreation, which contains the locality modifier set when the key was created;
- localityAtRelease, which contains the locality modifier that must be set in order to use the key created;
- creationPCRSelection, which contains the selection of PCRs active when the key was created;

- releasePCRSelection, which contains the selection of PCRs to which the key is bound;
- digestAtCreation, which contains the composite digest of the PCR values when the key was created; and
- digestAtRelease, which contains the digest of the PCR indices and the PCR values that must be verified when using the key that was bound to the PCRs.

The use of the *TPM_PCR_INFO_LONG* structure allows the values of a different set of PCRs to be reflected in the digestAtCreation and digestAtRelease fields. The *TPM_PCR_INFO_LONG* structure also allows the locality modifier that was set when the key was created, and the locality modifier required for key use, to be defined. The locality mechanism permits trusted processes communicating with *TPM* to indicate to *TPM* that a particular command has originated from a trusted process, the definition of which is platform-specific; see section A.12 for further details.

As no assumptions are made regarding the existence of multiple isolated compartments in this implementation, the locality feature is not required. Also, as the software configuration we wish to reflect in the PCR info parameter is represented by the entire PCR set, and is the same for both digestAtCreation and digestAtRelease, we assume here that a *TPM_Key* structure is used as an input parameter to, and as an output parameter from, the *TPM_CreateWrapKey* command.

In this particular implementation, it is required that the pcrSelection represents the entire set of PCRs. The returned digestAtCreation should reflect an execution environment which consists of correctly functioning broadcast and download applications running on a particular trusted operating system, and

nothing more. Verification of the returned digestAtCreation by A_D when using the key assures the download application that the key was created in the correct software environment and not by a rogue application.

The required digestAtRelease could be incorporated into the application, A_D, and then inserted as an input parameter to the *TPM_CreateWrapKey* command by A_D. The digestAtRelease could reflect, for example, a platform configuration in which a particular broadcast application and a particular download application are running on a particular trusted operating system, but nothing more.

The *TPM_PCR_INFO* structure in the returned *TPM_Key* structure describes the state of the execution environment to which the key is bound. However, if this data is to be communicated to the challenger, S, proof must exist that the data originated from a genuine TPM and that it has not been replayed. This is discussed below.

The final part of the *TPM_Key* structure to consider is the *TPM_Auth_Data_Usage* structure. This structure may take one of three values: *TPM_Auth_Never; TPM_Auth_Always*; or *TPM_Auth_Priv_Use_Only*. In this scenario, A_D must use the private key to decipher the symmetric keys protecting A_C. The first option is to permit A_D to use the private key without the submission of any authorisation data. In this case the *TPM_Auth_Data_Usage* structure is set to *TPM_Auth_Never*. Alternatively, the *TPM_Auth_Data_Usage* structure could be set to *TPM_Auth_Always* or *TPM_Auth_Priv_Use_Only*, where, on key pair generation, 20 bytes of authorisation data are associated with the public/private key pair, or with just the private decryption key, respectively. In this instance we assume that the *TPM_Auth_Data_Usage* structure is set to *TPM_Auth_Priv_Use_Only*.

To enable this, before a request for key pair generation, the user could be requested to provide a password, from which the authorisation data for private key use is derived. Thus, when use of the private decryption key is required, the correct password would have to be re-entered by the user. Alternatively, a known authorisation value could be used, or the authorisation value required for the use of S_{A_D} could be sealed to PCR values which represent a correctly functioning A_D running in a particularly configured execution environment.

8.3.1.2 Steps 9 to 11

Once a key pair has been created using the *TPM_CreateWrapKey* command, the key to be certified must be loaded using the *TPM_LoadKey2* command. The handle of the parent key, *CAAK*, is input as a parameter to this command, in conjunction with a parameter which proves to *TPM* that the parent key usage authorisation data is known by the caller. The *TPM_Key* structure of the newly created key to be loaded is also input. *TPM* responds by sending an internal TPM handle pointing to where the key is loaded.

8.3.1.3 Steps 12 to 17

The key handle returned from the load command is then used as an input parameter to either a *TPM_CertifyKey* or *TPM_CertifyKey2* command in a request for the loaded key to be certified. A 160-bit string of externally supplied data, which in this protocol is used to submit a hash of R_S and Id_S, is also given as an input parameter to this command.

In response to the *TPM_CertifyKey* command, *TPM* returns either a *TPM_Certify_Info* or a *TPM_Certify_Info2* data structure. Both certifyInfo structures describe the key-to-be-certified, including any authorisation data requirements, a digest of the public key-to-be-certified, 160 bits of external data,

and a description of the platform configuration data required for the release and use of the certified key. In addition to this structure, *TPM* also signs and returns a hash of the certifyInfo parameter.

Whether a *TPM_Certify_Info* or a *TPM_Certify_Info2* data structure is output, is determined by the localities and the PCRs the certified key is restricted to. A key with no locality restrictions, and one which is not bound to a PCR greater than PCR 15, will cause the command to return and sign a *TPM_Certify_Info* structure. Otherwise, a *TPM_Certify_Info2* data structure is returned. The *TPM_CertifyKey* command does not support the case where the certifying key requires a usage authorisation to be provided, but the key to be certified does not.

In response to the *TPM_CertifyKey2* command, a *TPM_Certify_Info2* data structure is returned. It supports the case where the certifying key requires a usage authorisation to be provided, but the key-to-be-certified does not. However, this command does not support the case where the key-to-be-certified requires a usage authorisation to be provided, but the certifying key does not. The *TPM_CertifyKey2* command must also be used to certify certifiable migratable keys.

Use of a particular command and a particular structure depends on whether the parent certifying key or key-to-be-certified are associated with usage authorisation data, and whether the key-to-be-certified is a non-migratable key or a certifiable migratable key. Use of a particular structure is also dependent on the required PCR binding. For the purpose of this protocol, the *TPM_CertifyKey* command is used, and a *TPM_Certify_Info* structure is returned by *TPM*.

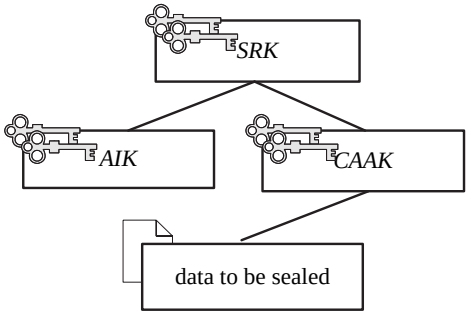

Figure 8.2: Key hierarchy for the download of A_C

8.3.2 Key agreement protocol

We now describe how the key agreement protocol defined in section 7.8 can be implemented using a TCG version 1.2 compliant system. A TPM key hierarchy similar to that described in section 8.3.1 is assumed. The key hierarchy described in figure 8.2 also contains a TPM storage root key, *SRK*, an attestation identity key, *AIK*, which has been generated and activated, and a storage key, *CAAK*. We assume, prior to the initiation of the protocol described below, that all three of these keys are in existence, and that the *SRK*, *CAAK* and *AIK* are loaded in *TPM*.

The numbered protocol steps described below map directly to the generic key exchange protocol steps described in section 7.8. This implementation involves the deployment of TCG-specific functionality in steps 12 to 22 and in step 29. *TPM_Seal* functionality is used in steps 12 to 14. Steps 15 to 22 utilise *TPM_Quote* functionality. Finally, in step 29, *TPM_Unseal* functionality is used. Note that further details of the TPM commands and structures used in the protocol are given in the text following the numbered steps.

1. A_D : Chooses a Diffie-Hellman private value a_M and calculates b_M based on g and p, which may have been hard-coded into the relevant application software or alternatively, may be retrieved from the relevant certificates.

2. $A_D \rightarrow S$: Request application $A_C \mid\mid b_M$.

3. S : Chooses a Diffie-Hellman private value a_S and calculates b_S using the same g and p that were used by A_D in step 1.

4. S : Signs the concatenation of b_S, its public Diffie-Hellman key, b_M, the public Diffie-Hellman key of A_D, and Id_M, the identity of M, to whom the message is being sent,
   $S_S(b_S \mid\mid b_M \mid\mid Id_M)$.

5. $S \rightarrow A_D : b_S \mid\mid S_S(b_S \mid\mid b_M \mid\mid Id_M)$.

6. A_D : Retrieves Cert_S and verifies it.

7. A_D : Verifies the signature on the message received in step 5 using the public signature verification key of S contained in Cert_S.

8. A_D : Verifies that b_M has been returned, thereby indicating that the message is fresh.

9. A_D : Verifies that Id_M is contained within the signed message to ensure that the message was destined for M.

10. A_D : Calculates the shared key K_{S,A_D} using b_S, the public Diffie-Hellman value of S, and a_M, the private Diffie-Hellman value of A_D.

11. A_D : Derives secret keys $K1_{S,A_D}$ and $K2_{S,A_D}$ used for data encryption and data integrity, respectively, from K_{S,A_D} by an agreed method.

12. $A_D \rightarrow TPM : TPM\_Seal_I(K1_{S,A_D} \| K2_{S,A_D})$.

13. TPM : Securely stores $K1_{S,A_D}$ and $K2_{S,A_D}$ using the protected storage mechanism, such that $K1_{S,A_D}$ and $K2_{S,A_D}$ can only be decrypted when the protected environment is in a specified state I_1, i.e. the TPM seals $K1_{S,A_D}$ and $K2_{S,A_D}$.

14. A_D : Keeps a record of I_2 to which the sealed keys were bound, the list of selected target PCRs, and the selected PCR values.

15. $A_D \rightarrow TPM$: $TPM\_Quote(H(b_M \mid\mid b_S \mid\mid Id_S) \mid\mid I_2$).

16. $TPM \rightarrow A_D : S_{TPM}(H(b_M \mid\mid b_S \mid\mid Id_S) \mid\mid I_2)$.

17. $A_D \rightarrow S : S_{TPM}(H(b_M \mid\mid b_S \mid\mid Id_S) \mid\mid I_2)$.

18. S : Obtains and verifies Cert_{TPM}.

19. S : Verifies $S_{TPM}(H(b_S \mid\mid b_M \mid\mid Id_S) \mid\mid I_2)$ using the public signature verification key of TM contained in Cert_{TPM}.

20. S : Verifies b_S to ensure that the message is fresh.

21. S : Verifies Id_S to ensure that the message was destined for S.

22. S : Checks that the message has come from a legitimate TPM, and checks whether I_2 represents a sufficiently trustworthy execution environment. Given I_2, S can verify that the A_D is executing as expected, that it has not been tampered with, and also that there is a legitimate broadcast application executing on the mobile host. Thus S can be sure that $K1_{S,A_D}$ and $K2_{S,A_D}$ have been sealed to the PCR data defined by I_1.

23. S : Calculates the shared key K_{S,A_D} using b_M, the public Diffie-Hellman value of A_D, and a_S, the private Diffie-Hellman value generated by S.

24. S : Derives secret keys $K1_{S,A_D}$ and $K2_{S,A_D}$ used for data encryption and data integrity, respectively, from K_{S,A_D} by an agreed method.

25. S : Computes a MAC on, and encrypts, A_C.

    $E_{K1_{S,A_D}}(A_C \mid\mid \text{MAC}_{K2_{S,A_D}}(A_C))$.

26. $S \rightarrow A_D : E_{K1_{S,A_D}}(A_C \mid\mid \text{MAC}_{K2_{S,A_D}}(A_C))$.

27. $A_D \rightarrow TPM$: Request to unseal $K1_{S,A_D}$ and $K2_{S,A_D}$.

    $K1_{S,A_D}$ and $K2_{S,A_D}$ can only be unsealed if the platform is in the agreed state, I_1. Furthermore, authorisation data may also be required, depending on the value of the *TPM_Auth_Data_Usage* field set in step 12.

28. A_D: Compares I_2 to its record of I_2, to which the symmetric keys were bound in step 12, to ensure that the request for sealing the symmetric keys came from A_D.

29. A_D : Decrypts $E_{K1_{S,A_D}}(A_C \mid\mid \text{MAC}_{K2_{S,A_D}}(A_C))$.

30. A_D : Verifies $\text{MAC}_{K2_{S,A_D}}(A_C)$.

31. Once A_C is executing, A_D precludes the potential interference of any other application with A_C.

32. A_D : Deletes A_C, and all other keys, when they are no longer required. The encrypted copy of A_C may remain stored for future use, space permitting.

8.3.2.1 Steps 12 to 14

The *TPM_Seal* command is used in step 12 to securely store the keys $K1_{S,A_D}$ and $K2_{S,A_D}$. The input parameters for this command include the data to be sealed and the authorisation data required to unseal the data. The *TPM_Seal* command is also given information identifying the PCRs whose values are to be bound to the protected data. In response, *TPM* returns either a *TPM_Stored_Data* or a *TPM_Stored_Data12* structure. Both the *TPM_Stored_Data* and the

TPM_Stored_Data12 structures contain the platform configuration to which the sealed data is bound and a *TPM_Sealed_Data* structure. The latter contains the encrypted data, and the authorisation requirements for access to the data. The *TPM_Stored_Data* and *TPM_Stored_Data12* data structures differ only in the way in which the PCR info parameter is described. If a *TPM_Stored_Data* data structure is returned from the *TPM_Seal* command, a *TPM_PCR_INFO* structure will describe the platform configuration to which the key is bound. Alternatively, if a *TPM_Stored_Data12* data structure is returned from the *TPM_Seal* command, a *TPM_PCR_INFO_LONG* structure will describe the platform configuration to which the key is bound. We assume here that the *TPM_Seal* command is used and the *TPM_Stored_Data* structure output.

For our application we require that the key used to seal the symmetric keys, $CAAK$, is non-migratable. This implies that the private key cannot be migrated from *TPM*, in which it was created. Alternatively, a certifiable migratable key, created with the *TPM_CMK_CreateKey* command, may be used in the seal operation. This migratable key may be certified by *TPM* and migrated, but only under strict controls. This prohibits the sealing key, which essentially protects the conditional access application, from being migrated to an arbitrary platform authorised by the TPM owner, but permits its migration to selected devices, e.g. to other TPMs owned by the same entity. We will focus, however, on the case where the sealing key is non-migratable.

8.3.2.2 Steps 15 to 22

Finally, the *TPM_Quote* or *TPM_Quote2* command is used to instruct *TPM* to attest to the platform's configuration. The parameters given to either command include the indices of the PCRs to be attested to. 160 bits of external data may also be supplied, which, in this protocol, are used to submit a

one way hash of b_M, b_S and Id_S for attestation. *TPM_Quote2* differs from *TPM_Quote* in that *TPM_Quote2* uses the *TPM_PCR_INFO_SHORT* rather than a *TPM_PCR_COMPOSITE* structure to hold information relating to the PCRs. *TPM_PCR_INFO_SHORT* holds locality information, thereby providing the challenger with a more complete view of the current platform configuration. Use of a particular structure is also dependent on the required PCR binding. The *TPM_Quote* command is used here.

8.3.3 Implementation specific security analysis

We now consider how/whether security requirements 6 and 7 from section 6.3 are met by the TCG-specific protocol implementations specified in sections 8.3.1 and 8.3.2.

6. *Confidentiality and integrity protection of the cryptographic keys used in the prevention of unauthorised reading of, and the detection of unauthorised modification to, the application code and data:*

   (a) *Secure symmetric key generation:*

   In the key exchange protocol, the symmetric keys, $K1_{S,A_D}$ and $K2_{S,A_D}$, are generated by the software provider.

   In the key agreement protocol, $K1_{S,A_D}$ and $K2_{S,A_D}$ are derived both by the software provider and on the mobile receiver. It must be ensured by S, therefore, that the symmetric keys are derived in an environment which is indeed trusted by the software provider. S can verify the software environment in which the symmetric keys were derived, and also the value I_1 to which A_D requested that the keys were sealed by verifying the attestation statement sent in step 17 of the key agreement protocol.

In the scenario where the host platform has a version 1.2 compliant TPM and a CRTM, but no mechanisms in place to facilitate the construction of multiple isolated compartments on the platform, the entire PCR set must be attested to by *TPM*, so that S can obtain a full image of the platform's software configuration. Ideally, only a trusted operating system, a download application and a broadcast would be running on the platform, or the verification of the platform's configuration by S may quickly become an overly complex task.

(b) *Secure symmetric key transmission:*

As stated above, in the key exchange protocol, the symmetric keys are generated by the software provider and must therefore be securely transmitted to the mobile host. In order to do this, the software provider takes the public encryption key sent by the mobile host, encrypts and signs the symmetric keys, and returns the encrypted bundle. Because the corresponding private key is known only to *TPM* embedded in the mobile platform, an attacker cannot compromise the confidentiality of the symmetric keys in transit. If the encrypted MACing and encryption keys are modified in an accidental or a malicious way, the verification of the signature on the MACing and encryption keys will fail, and so this will be detected.

In the key agreement protocol, keys do not need to be transmitted as they are derived locally on the mobile host.

(c) *Secure symmetric key storage:*

In the key exchange protocol, the symmetric keys are encrypted by the software provider using the public encryption key sent by the mobile host, and then signed. The keys remain encrypted and signed while in storage on the mobile host, until their use. Because the cor-

responding private key is known only to *TPM* which is embedded in the mobile platform, an attacker cannot compromise the confidentiality of the symmetric keys while in storage. If the encrypted MACing and encryption keys are modified in an accidental or a malicious way while in storage, the verification of the signature on the MACing and encryption keys will fail, so this will be detected. $K1_{S,A_D}$ and $K2_{S,A_D}$ must also be securely managed and protected by S, at least to the same degree as A_C itself is protected.

In the key agreement protocol the symmetric keys, when derived, are encrypted by *TPM* and bound to a specific execution environment state (sealed). Once the keys have been sealed an attacker cannot compromise the confidentiality of the symmetric keys while in storage, as the private key required to decrypt the symmetric keys is known only to *TPM*. While the sealing capability does not explicitly integrity protect the sealed keys, the association of 20 bytes of authorisation data with the sealed data provides implicit integrity protection, as described in section A.13.

(d) *Prevention of unauthorised access to the symmetric keys:*

In the key exchange protocol, an asymmetric key pair is generated by *TPM*. The symmetric keys, $K1_{S,A_D}$ and $K2_{S,A_D}$, used to MAC and encrypt the application, are then securely delivered to M by S encrypted under P_{A_D}, as stated above. The non-migratable private key, S_{A_D}, required to decrypt $K1_{S,A_D}$, is securely stored by *TPM*, and its use is only permitted when the platform is in a particular state, which has been verified as trustworthy by the software provider.

In conjunction with this, twenty bytes of authorisation data may have been associated with S_{A_D}. However, a problem arises regard-

ing where this authorisation data may be stored. It may be securely stored by *TPM*, i.e. sealed to A_D, but this offers no additional protection as regards preventing unauthorised access to S_{A_D}, than if no authorisation data were associated with it. This is an important issue, but one that is not dealt with in the TCG specifications.

As an alternative option, it may be relatively straightforward for a *user* to provide the necessary password, during key pair generation, from which the key usage authorisation data may then be derived. This option may be acceptable so long as user interaction with A_D is permitted, and there is a secure link between the user entering the password and *TPM*.

In the key agreement protocol the symmetric keys are encrypted by *TPM* and bound to a specific execution environment state (sealed). Twenty bytes of authorisation data are also associated with the sealed MACing and encryption keys, for more stringent control against unauthorised access.

Once the keys have been decrypted, they are protected using the same mechanisms used to protect the decrypted A_C as described in the next section.

7. *Confidentiality and integrity protection of the application code and data during execution:*

   No mechanisms are described by the TCG for partitioning a system into trusted and untrusted compartments or execution environments. On the face of it, one could take this to imply that the 'protected execution environment' we speak of in relation to the TCG protocol implementation must encompass the entire platform. In order to gain some assurances about the platform's behaviour, the software provider/challenger of a platform

may potentially require that only a trusted operating system and a limited applications set (namely A_D and A_B) are running on the platform, so that the state can be considered trustworthy for the download and execution of A_C. Consequently, the system would be rendered unusable for any purpose other than broadcast for the duration of A_C download and use. Once $K1_{S,A_D}$, $K2_{S,A_D}$ and A_C have been decrypted, A_D, as defined in chapter 7, precludes the potential interference of any other application with A_C.

Alternatively, if platform use is to remain open, a challenger may be faced with the task of verifying a large set of potentially complex integrity metrics, making the process of PCR verification and assessment almost certainly an impossible one. In conjunction with this, unless A_D is running in a controlled environment, for example on a trusted operating system in conjunction with A_B, then application controls provided by A_D may be circumvented.

In reality, however, it would appear that the TCG never intended the security mechanisms they describe to be deployed in isolation. System partitioning, for example, represents a vital facet of trusted computing, but its implementation is left open. For our particular use case, therefore, it is beneficial if the system can be compartmentalised into trusted and untrusted environments. This facilitates simpler PCR verification, and enables untrusted applications to be executed in parallel to, but in isolation from, those running in the trusted environment. The deployment of system partitioning may also be used to ensure that the conditional access application may not be manipulated while executing on the mobile host. It becomes clear that in order to implement the above protocols as securely as possible, the entire system needs to be considered, not merely

the trusted components upon which that platform is built.

Although the functionality described in the TCG specification set provides a solid starting point to implement the protocols, a more complete architecture detailing the entire trusted platform, from the trusted foundation to the application layer, would be advantageous. This complete architecture may be provided using a combination of additional hardware and/or software built around the TCG standard components.

8.4 Implementing the protocols using the TCG specification set and an integrated isolation kernel

The deployment of an isolation layer represents one of the most widely discussed and secure methods of implementing isolated compartments on a computing platform, where an isolation layer "provides a means to isolate operating systems, application and applets" [104]. Implementations include those based on virtual machine monitors, hypervisors, microkernels and exokernels, see section 1.6.8. Through the integration of an isolation layer into a TCG-defined trusted platform containing a version 1.2 compliant TPM and a CRTM, the protocols described in section 8.3 may be implemented in a more efficient and secure way.

As was the case in section 8.3, the trusted module, introduced in chapter 7, is mapped to a version 1.2 compliant TPM and CRTM as specified by the TCG. In this particular implementation, the architecture also aims to provide a high assurance runtime environment for trustworthy applications. It is assumed that multiple isolated execution environments can be supported through the deployment of an isolation layer, and that at least one protected execution

environment/protected compartment is running on the platform. Within this environment, different applications run in isolation, free from being observed or compromised by other processes running in any insecure partition that may exist in parallel, see section 1.6.8.

Given that a version 1.2 compliant TPM and CRTM, as specified by the TCG, is assumed to be integrated into the platform, the protocols we are proposing will be executed in a way similar to that described in section 8.3. However, because this architecture encompasses an isolation layer, which in turn facilitates the existence of isolated compartments on the platform, it is logical to assume that locality modifiers could be used when describing the platform's configuration.

8.4.1 Key exchange protocol

When implementing the generic key exchange protocol described in section 7.7, the protocol mapping will remain, in the most part, consistent with that described in section 8.3.1. However, the required data structures output from the TPM commands called may differ. When the *TPM_CreateWrapKey* is called in step 5 of section 8.3.1, the *TPM_Key12* structure, which uses the *TPM_PCR_INFO_LONG* structure to properly define the PCRs, is output rather than the *TPM_Key* structure. The *TPM_PCR_INFO_LONG* structure allows the definition of the locality modifier that was set when the key was created, and the locality modifier required for key use. When the *TPM_CertifyKey* command is called in step 12 of section 8.3.1, a *TPM_Certify_Info2* structure is returned by *TPM*. This data structure must be returned when the certified key is limited by locality.

8.4.2 Key agreement protocol

When implementing the generic key agreement protocol described in section 7.8, the protocol mapping will remain generally consistent with that described in section 8.3.2. However, when the *TPM_Seal* command is used, a *TPM_Stored_Data* structure is output. The *TPM_Quote2* command may be called instead of the *TPM_Quote* command. *TPM_Quote2* differs from *TPM_Quote* in that *TPM_Quote2* uses the *TPM_PCR_INFO_SHORT* structure rather than the *TPM_PCR_COMPOSITE* structure to hold information relevant to the PCRs. The *TPM_PCR_INFO_SHORT* structure holds locality information, thereby providing the challenger with a more complete view of the current platform configuration.

8.4.3 Implementation specific security analysis

We now consider how security requirements 6 and 7 from section 6.3 are met when the protocols are implemented on a platform into which a TCG version 1.2 compliant TPM and CRTM and an isolation layer are integrated.

6. *Confidentiality and integrity protection of the cryptographic keys used in the prevention of unauthorised reading of and the detection of unauthorised modification to the application code and data:*

   (a) *Secure symmetric key generation:*

   In the key exchange protocol, the symmetric keys, $K1_{S,A_D}$ and $K2_{S,A_D}$, are generated by the software provider.

   In the key agreement protocol, $K1_{S,A_D}$ and $K2_{S,A_D}$ are derived on the platform. It must be ensured, therefore, that the symmetric keys are derived in an environment which is trusted by the software provider. S can verify the software environment in which the symmetric

keys were derived, and also the value I_1 to which A_D requested that the keys were sealed, by verifying the attestation statement sent in step 17 of the key agreement protocol.

In this implementation, where it is assumed that isolated software compartments are running on the platform, the PCR set attested to by *TPM* can be greatly reduced. Ideally, PCRs reflecting the boot process, the isolation kernel, and the compartment in which a download application and a broadcast are running would be attested to by *TPM*, making verification of the protected execution environment by S much less complex.

(b) *Secure symmetric key transmission:*

In the key exchange protocol, keys are protected during transmission, as described in point 6b in section 8.3.3.

(c) *Secure symmetric key storage:*

In both the key exchange and key agreement protocols, keys are securely stored, as described in point 6c in section 8.3.3.

(d) *Prevention of unauthorised access to the symmetric keys:*

In both the key exchange protocol and the key agreement protocol, the symmetric keys can only be decrypted when the host platform, or, more specifically, a particular compartment which resides on the host platform, is in a particular state. Knowledge of the required usage authorisation data may also have to be demonstrated in order to access the symmetric keys.

Once the keys have been decrypted, they are protected using the same mechanisms used to protect the decrypted A_C, as described in the next section.

7. *Confidentiality and integrity protection of the application code and data during execution:*

   $K1_{S,A_D}$, $K2_{S,A_D}$, and A_C, once decrypted, are protected within an isolated compartment, which may be constructed using an isolation layer deployed within the platform. How exactly these compartments are isolated from one another is dependent on the particular isolation layer implementation, see section 1.6.8. Every isolation layer implementation, however, aims to fulfil two basic requirements, isolation and assurance [126].

   In order to fulfil the isolation requirement, a program must be able to execute free from external interference. In conjunction with this, a program's data or computations should not be observable by other entities, except for data the program chooses to reveal through interprocess communication.

   It is also required that the isolation layer behaves as specified. A high degree of assurance in the behaviour of the isolation layer can only be achieved if it is kept as small and as simple as possible.

   While isolation layer implementations have aimed to meet both isolation and assurance requirements, this has not always been achieved because of problems which have arisen in relation to device support and backward compatibility [126].

   One approach used to support devices is to vitualise them, see section 1.6.8. While the number of devices supported by a computing platform remains small, this is a viable solution. However, in today's consumer environment there are an ever increasing set of devices which need to be supported. With each device that is virtualised, the size of the isolation kernel grows, and in turn moves further from meeting the assurance requirement [126].

   As an alternative, devices may be exported to guests by the isolation ker-

nel, see section 1.6.8. In this case, device accesses by guests are made directly to the device, without translation by the isolation layer. This implies that device drivers need not be included in the isolation layer. Problems arise, however, in relation to direct memory access (DMA) devices which are given full access to physical memory. This may in turn result in problems as regards whether the isolation layer can fulfil the isolation requirement. In the majority of isolation layers, protection mechanisms used to isolate compartments from one another, for example virtual to physical memory mappings, may be circumvented by the presence of DMA devices which have direct access to physical memory [126].

Whether or not the isolation layer meets the assurance requirement may also be affected by whether compatibility with current operating systems is supported. While, ideally, an operating system (OS) developed prior to the isolation layer could be executed without modification on the isolation layer, problems may arise when exposing the original hardware to a guest. More specifically, the x86 CPU is not virtualisable and, in order to deal with this, the complexity of the isolation layer must be increased. As an alternative, paravirtualisation techniques may be utilised. In this case, the exact original machine model is not exported, requiring the original OS code to be modified but keeping the isolation layer as simple as possible.

If the symmetric keys and A_C are to be protected while exposed on the platform, the issues surrounding DMA devices and backward compatibility must be addressed.

As stated above, once A_C is executing in the isolated compartment, A_D will prevent the potential interference of any other application with A_C within the isolated compartment.

8.5 Implementing the protocols using NGSCB

The NGSCB architecture encompasses a broader set of capabilities than the TCG-defined trusted platform, see section A.4. In addition to the functional components defined within the version 1.2 TCG specification set, NGSCB provides:

- An extended CPU to enable the efficient implementation of a minimal isolation kernel (as described in [72], for example);
- A minimal isolation kernel;
- Memory controller or chipset extensions such that direct memory access can be controlled (as described in [72], for example); and
- Hardware components enabling input and output to be efficiently secured (as described in [72], for example).

In chapter 7, assumptions were made about the software provider, the mobile receiver, the trusted module embedded within the mobile receiver, and the download application. The assumptions pertaining to the trusted module and the architecture of the mobile receiver defined in section 7.5 are re-examined in view of this mapping.

As was the case in sections 8.3 and 8.4, we suppose here that the trusted module, introduced in chapter 7, is mapped to a version 1.2 compliant TPM and CRTM, as specified by the TCG. The NGSCB architecture also aims to provide a high assurance runtime environment for trustworthy applications. Through the deployment of various hardware extensions, as described in A.4, and the integration of an isolation kernel, the platform can facilitate the execution of multiple isolated compartments or domains. It is assumed that at least one pro-

tected execution environment/protected compartment is running on the platform. Within this environment, different applications run in isolation, free from being observed or compromised by other processes running in any insecure partition that may exist in parallel, see section 1.6.8. The services described in section 6.3.2 are assumed to be available in this environment.

Given that an NGSCB compliant system will incorporate a version 1.2 compliant TPM and CRTM, as specified by the TCG, it is reasonable to assume that the protocols we are proposing will be executed in a way similar to that described in section 8.3. Because the NGSCB architecture encompasses an isolation kernel, which in turn facilitates the existence of isolated compartments on the platform, it is logical to assume that locality modifiers could be used when describing the platform's configuration, as was described in section 8.4 above.

8.5.1 Key exchange protocol

As was described in section 8.4.1, when implementing the generic protocol described in section 7.7 on an NGSCB compliant platform, the protocol mapping will remain, in the most part, consistent with that described in section 8.3.1. However, the required data structures output from the TPM commands may differ. When the *TPM_CreateWrapKey* command is called in step 5 of section 8.3.1, the *TPM_Key12* structure, which uses the *TPM_PCR_INFO_LONG* structure to properly define the PCR registers, is output rather than the *TPM_Key* structure. The *TPM_PCR_INFO_LONG* structure also allows the locality modifier that was set when the key was created, and the locality modifier required for key use, to be defined. When the *TPM_CertifyKey* command is called in step 12 of section 8.3.1, a *TPM_Certify_Info2* structure is returned by the TPM. This data structure must be returned when the certified key is limited by locality [158].

8.5.2 Key agreement protocol

When implementing the generic protocol described in section 7.8, the protocol mapping will remain, in the most part, consistent with that described in section 8.3.1. When implementing the key agreement protocol described in section 7.8 on an NGSCB compliant platform, the *TPM_Seal* command is used and a *TPM_Stored_Data* structure is output, and the *TPM_Quote2* command may be called instead of the *TPM_Quote* command, as was the case in section 8.4.2.

8.5.3 Implementation specific security analysis

We now analyse the differences that the NGSCB architecture would make to the security of the protocols described in section 8.3.

6. *Confidentiality and integrity protection of the cryptographic keys used in the prevention of unauthorised reading of and the detection of unauthorised modification to the application code and data:*

   (a) *Secure symmetric key generation:*

   In the key exchange protocol, the symmetric keys, $K1_{S,A_D}$ and $K2_{S,A_D}$, are generated by the software provider.

   In the key agreement protocol, $K1_{S,A_D}$ and $K2_{S,A_D}$ are derived on the platform. It must be ensured, therefore, that the symmetric keys were derived in an environment which is trusted by the software provider. S can verify the software environment in which the symmetric keys are derived, and also the value I_1 to which A_D requested that the keys were sealed, by verifying the attestation statement sent in step 17 of the key agreement protocol.

   In this implementation, where isolated software compartments are running on the platform, the PCR set attested to by *TPM*, can

be greatly reduced. Ideally, the PCR values communicated would represent the boot process, the isolation kernel, and download and broadcast applications running in an isolated compartment, making verification much less complex.

(b) *Secure symmetric key transmission:*

In the key exchange protocol, keys are protected during transmission, as described in point 6b in section 8.3.3.

(c) *Secure symmetric key storage:*

In both the key exchange and key agreement protocols, keys are securely stored, as described in point 6c in section 8.3.3.

(d) *Prevention of unauthorised access to the symmetric keys:*

As was described in section 8.3.3, the NGSCB architecture will utilise the TPM's protected storage functionality to ensure that $K1_{S,A_D}$ and $K2_{S,A_D}$ are only accessible when the protected execution environment is in a particular state. Additional authorisation data may also be required for access.

Once the keys have been decrypted, they are protected using the same mechanisms used to protect the decrypted A_C, as described in the next section.

7. *Confidentiality and integrity protection of the application code and data during execution:*

The NGSCB architecture facilitates system partitioning through the implementation of an isolation kernel. As described in section 8.4.3, this system partitioning enables simpler PCR verification, as the number of applications running in a particular protected execution environment may be strictly controlled.

The NGSCB isolation kernel exposes the original hardware to one guest OS, see section 8.4.3. This offers the advantage that legacy operating systems and applications may remain in use, despite the adoption of trusted computing technologies. In order to facilitate efficient OS compatibility without bloating the isolation kernel, a new CPU mode is introduced so that the isolation kernel can run in a new ring -1, and guest operating systems can still execute in ring 0. This avoids problems in relation to the virtualisability of particular OS instruction sets which may arise if a virtual machine monitor were to be deployed in ring 0.

With respect to memory partitioning, the NGSCB isolation kernel uses an algorithm called page table edit control (PTEC) to partition physical memory among guests. This is described in greater detail in appendix B but, from a security perspective, protection is analogous to that of traditional virtual memory protections such as those that use translation lookaside buffers (TLBs) or page tables.

The NGSCB isolation kernel contains device specific code for a very small number of devices, i.e. those needed for the operation of the isolation kernel, see section 8.4.3. All other consumer devices are exported to guest OSs. This leaves the issue of DMA devices. As described in section 8.4.3, memory protection mechanisms do not help if an attacker can subvert or bypass the operating system kernel controls via direct memory access). In order to prevent this type of attack, Microsoft has encouraged chipset manufacturers, for example Intel [72], to make changes to their hardware, so that an access control policy map may be defined by software (for example, the isolation kernel) and stored in main memory. This policy map then indicates whether a particular subject (DMA device) should be able to access (read or write to) a particular resource (physical address).

The enforcement of this policy map is completed by hardware. Further details of this functionality are available in appendix B.

8.6 Conclusions

In this chapter we have examined the implementation of the abstract key exchange and key agreement protocols described in chapter 7 on a selection of trusted computing architectures.

In a TCG compliant platform security service 6 can be met. Problems may arise however in relation to the provision of security service 7. No mechanisms are defined by the TCG for partitioning a system into trusted and untrusted compartments. In order for a software provider to trust the execution environment in which the conditional access application will execute, he may require that the platform is in a controlled state, running for example a trusted OS, a download application, and a broadcast application, but nothing more. Essentially, the end host may be required to become a more closed platform. If a TCG compliant platform were to remain open in this scenario, it would become very difficult for a software provider to verify the attestation statement generated by the end host, and also to evaluate whether a platform should be trusted for this particular purpose, i.e. the secure download and execution of a conditional access application.

If an isolation layer is integrated into a TCG compliant mobile device, the platform can be partitioned into both trusted and untrusted execution environments. In this way A_C can be executed in an isolated execution environment, which M has attested to, and S has verified and evaluated as trusted for the secure download and execution of a conditional access application. The verification of PCRs, which must be completed by S, is simplified and M can remain

open and useable. On implementation of an isolation layer, problems may arise, however, in relation to OS compatibility and DMA attacks.

Issues surrounding device support and OS backward compatibility may be tackled through the extension of the platform chipset and enhancement of the platform CPU, as described as both Microsoft's NGSCB and Intel's LaGrande initiatives.

Chapter 9

Secure application download using XOM and AEGIS architectures

Contents

9.1 Introduction 311
9.2 Model 312
9.3 Notation 313
9.4 Assumptions 313
9.5 Protocol initiation 315
9.6 The XOM application download protocol 316
9.6.1 The XOM system architecture 316
9.6.2 The XOM download protocol 317
9.6.3 Security analysis 319
9.6.4 Proposed security enhancements/clarifications 325
9.7 The AEGIS application download protocol 327
9.7.1 The AEGIS system architecture 327
9.7.2 The AEGIS download protocol 328
9.7.3 Security analysis 331
9.7.4 Proposed security enhancements/clarifications 337
9.8 Conclusions 339

The designers of XOM and AEGIS, Lie et al. and Suh et al., have both proposed protocols for secure application download. These protocols are based upon the assumption that the host device contains a hardened processor rather than a trusted module, as assumed in chapters 7 and 8. In this chapter, we

examine these two download protocols, which assume a mobile receiver compliant with the XOM and the AEGIS system architectures, respectively. Both protocols are then analysed against the security requirements described in chapter 6. These analyses have revealed serious security shortcomings in both of these protocols. We subsequently give a series of proposed enhancements to the protocols designed to address the identified shortcomings.

9.1 Introduction

Two protocols which enable the secure download and execution of an application A_C were specified in chapter 7. Chapter 8 then described three possible implementations of the abstract key exchange and key agreement protocols defined in chapter 7, using trusted computing technologies. Having analysed whether the security requirements listed in chapter 6 were fulfilled by the protocols described in chapters 7 and 8, we now move on to examine two protocols for secure application download and execution proposed by the designers of the XOM and AEGIS system architectures, Lie et al. and Suh et al., see appendix B. These protocols are based upon the assumption that the mobile receiver contains a hardened processor rather than a trusted module, as assumed in chapters 7 and 8.

In section 9.2 the generic model under consideration and its associated entities are described. Section 9.3 details the notation used in the protocol descriptions, and section 9.4 outlines the basic assumptions upon which the protocols are based.

Section 9.5 describes the events culminating in the initiation of either download protocol. Sections 9.6 and 9.7 examine how the XOM and AEGIS system architecture developers intended application download to be implemented using their respective technologies. A security analysis against the security requirements described in chapter 6 is also given for both protocol implementations. A series of protocol enhancements designed to rectify the identified weaknesses in the protocols is also provided.

9.2 Model

The model underlying both the XOM and AEGIS application download protocols is illustrated in figure 9.1. It involves three parties; the mobile receiver, the broadcaster and the software provider, as in figure 7.1. In this model, however, the fourth fundamental component is a tamper resistant processor chip (a hardened processor (HP)), modified to facilitate the secure delivery and execution of software on M.

As previously described in section 7.2, the mobile user does not need to have a long term relationship with the broadcaster, but is assumed to be aware of the content provision services that are available. Some of these services may be scrambled, in which case access is controlled by a conditional access system. For each scrambled service, the associated conditional access application A_C must be acquired by the mobile receiver from a software provider.

One fundamental requirement for the application download protocols described in chapters 7 and 8 is that the mobile receiver is able to demonstrate to the software provider that it is a bona fide trusted receiver, and that it is not in a malicious state that might enable the modification, replication, or extraction of secret data from the downloaded application. Once the receiver has proved itself to be trustworthy, A_C can be delivered to the platform protected using mechanisms that prevent access to (decryption of) A_C unless the host platform is in the state deemed trustworthy by the challenger. In the model illustrated in figure 9.1, where M has an integrated tamper resistant hardened processor, the application can be cryptographically bound to an individual processor authorised to execute the code. The software provider need only ensure that A_C is encrypted such that it can only be decrypted by and executed on a hardened processor verified as legitimate by S.

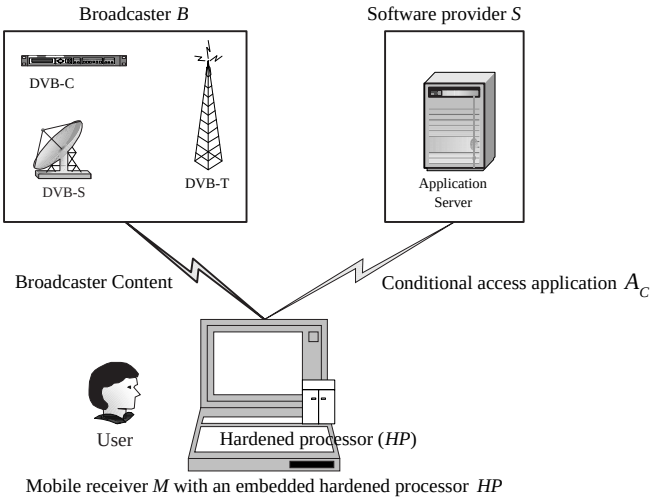

Figure 9.1: Architecture model

9.3 Notation

The majority of notation used in the specification of this protocol has been defined in section 7.4. The following additional notation will also be used:

HP	denotes a hardened processor embedded in the mobile receiver *M*.
XOM_HP	denotes an XOM hardened processor embedded in the mobile receiver M.
AEGIS_HP	denotes an AEGIS hardened processor embedded in the mobile receiver M.
SK	denotes a secure kernel executing on the mobile receiver M.

While in chapters 7 and 8 it is assumed that H is a 160-bit hash function, for XOM, H is a 128-bit hash function, and for AEGIS, H is unspecified.

9.4 Assumptions

The following pre-conditions need to be satisfied for use of the protocols described in sections 9.6 and 9.7.

1. There exists a certification authority (CA), trusted by both M and S. Both M and S possess a trusted copy of the public key of CA, so that they can both verify certificates generated by CA.

2. The designers of the relevant applications have agreed on the use of the protocol, and have also agreed on all the necessary cryptographic algorithms.

3. A hardened processor HP, is integrated into the mobile receiver. This hardened processor enables the isolated execution of processes on the mobile receiver. In order to achieve this, attacks must be prevented against the following elements:

   (a) the initial state of applications;

   (b) on-chip caches and registers;

   (c) application state on interrupts; and

   (d) external memory.

   The hardened processor must also provide a secure storage area.

4. All secret keys required by the mobile receiver in the implementation of the protocols described below are protected by the hardened processor.

5. A unique asymmetric key pair is associated with the hardened processor. This key pair is used for encryption/decryption.

6. The private decryption key from the pair referred to in point 5 is securely stored in the hardened processor.

7. The public encryption key from the pair referred to in point 5 is certified. The certificate, Cert_{HP}, contains an identifier for HP, and a general description of HP and its security properties. This certificate will most probably be generated by the manufacturer of HP.

8. The manufacturer of *HP*, or indeed the entity who generated the certificate described in point 7, must be trusted by S. S must possess a trusted copy of the public key of this entity.

9. The software provider S possesses a signature key pair, used only for entity authentication.

10. The private signing key from the pair referred to in point 9, is securely stored by the software provider.

11. The software provider S has a certificate, Cert_S, issued by CA. This certificate associates the identity of S with the public verification key from the pair referred to in point 9. This certificate must be available to the mobile receiver.

12. Every mobile device wishing to receive video broadcast must have a trusted broadcast application, A_B, running in a protected execution environment.

13. Every mobile device has a download application, A_D, running in a protected execution environment.

9.5 Protocol initiation

As was the case in section 7.6, both application download protocols begin when the user makes a request to the broadcast application, A_B, to view a specific video broadcast. If reception of this broadcast is controlled by a particular conditional access application, A_C, then A_B carries out the following steps:

1. A_B checks to see if the mobile device has dedicated hardware or software installed to support the specific conditional access system.

2. If no dedicated hardware, for example a common interface module, exists

on the mobile device, then A_B determines whether A_C has previously been downloaded and is still available in secure storage.

(a) If so, the download application A_D is called to load the protected A_C from secure storage into *HP*, where it is executed.

(b) If A_C is not available on the mobile device, then A_D is called to download the application. The download of A_C can be accomplished using either of the protocols described in sections 9.6 and 9.7.

9.6 The XOM application download protocol

The XOM architecture and application download protocol were proposed in order to meet one general security requirement — to support the copy and tamper resistant download and execution of software [98, 99].

9.6.1 The XOM system architecture

The XOM system architecture aims to provide protected compartments for XOM code to execute in. It provides on-chip protection of caches and registers, protection of cache and register values during context switching and on interrupts, and confidentiality and integrity protection of application data when transferred to external memory.

The platform subsystem used to provide the services listed above is called the XOM machine. In a hardware implementation of the XOM machine, complete trust is put in the modified CPU hardware, which provides the security services listed above. Everything transmitted outside the main CPU is both integrity and confidentiality-protected. However, an XOM virtual machine monitor (XVMM) implementation of the XOM machine reduces the number of necessary CPU extensions. In this case, a software XVMM, whose integrity is validated via

secure boot, is used to provide some of the security services provided directly by the CPU in the hardware implementation. For a detailed description of the XOM machine, see appendix B. For the purpose of this chapter, we focus on the hardware implementation of the XOM machine.

9.6.2 The XOM download protocol

The XOM application download protocol is defined as follows.

1. $A_D \rightarrow S$: Request for $A_C \mid\mid P_{XOM\_HP}$.
2. *S:* Verifies P_{XOM_HP} as belonging to a legitimate *XOM_HP* by retrieving and verifying Cert_{XOM_HP}, most probably using the public key of the manufacturer of the *XOM_HP*.
3. *S:* Generates a symmetric compartment key K.
4. *S:* Encrypts A_C using K, $E_K(A_C)$.
5. *S:* Encrypts K using P_{XOM_HP}, $E_{P_{XOM\_HP}}(K)$.
6. $S \rightarrow A_D : E_{P_{XOM\_HP}}(K) \mid\mid E_K(A_C)$.
7. *XOM_HP:* Loads $E_{P_{XOM\_HP}}(K) \mid\mid E_K(A_C)$.
8. *XOM_HP:* Decrypts K using S_{XOM_HP}.
9. *XOM_HP:* Decrypts A_C using K.
10. *XOM_HP:* Ensures the protected execution of A_C.

Using the XOM system architecture, the protocol begins when A_D requests download of the application, A_C. In conjunction with this request, the public key of the hardened processor upon which A_D is executing is also sent to the software provider.

S verifies that the public key received belongs to a genuine XOM hardened processor. S then chooses a symmetric compartment key, K. S uses this key K to encrypt A_C. S then encrypts K using the public encryption key of *XOM_HP*. As a result, the compartment key, and consequently A_C, cannot be accessed by any entity other than the intended recipient (i.e. *XOM_HP*, which has been verified as legitimate by S). The following message is returned by the software provider: $E_{P_{XOM\_HP}}(K) \mid\mid E_K(A_C)$.

When the application is required, the *enter_xom* instruction is called. The input to this command indicates the starting memory address of $E_{P_{XOM\_HP}}(K)$, the encrypted symmetric compartment key for A_C. Execution of the *enter_xom* instruction causes the XOM machine to check whether the symmetric compartment key has been decrypted on a previous occasion by comparing $H(E_{P_{XOM\_HP}}(K))$ with the hash values of all symmetric compartment keys securely stored in the XOM key table.

If no match is found, an XOM key table entry is allocated to the XOM application, A_C. An XOM identifier (ID_{XOM}) is assigned to A_C. The symmetric compartment key is decrypted and stored with the ID_{XOM} assigned to A_C. In conjunction with this, a mutating register key is associated with the ID_{XOM} assigned to A_C, which is used in the protection of register values on interrupts and context switches. In order to prevent replay of registers, this key is updated every time the XOM process is interrupted.

During execution of the protected application, *XOM_HP* protects A_C using a variety of mechanisms, as detailed below.

9.6.3 Security analysis

The XOM application download protocol is now analysed against the security requirements described in section 6.3.

1. *Confidentiality of the application code and data:*

   Symmetric encryption is used to confidentiality protect A_C, where A_C is encrypted with the symmetric compartment key K chosen by S. The confidentiality of A_C is also dependent, however, on the confidentiality of K, which is analysed below.

2. *Integrity protection of the application code and data:*

   According to [99], no mechanisms are deployed to protect the integrity of the downloaded application, which in this case is A_C. The software distribution model, described in the more recent thesis of Lie [98], does not explicitly mention integrity protection either. Such mechanisms are, however, implied in the definition of the *enter_xom* instruction in [98], where the author states that the *enter_xom* instruction must always be followed by encrypted and MACed instructions. No mention is made in [99] or [98] as to how this MAC key is derived.

3. *Entity authentication:*

   The XOM application protocol does not provide authentication of the XOM hardened processor by the software provider or authentication of the software provider by the XOM hardened processor.

4. *Origin authentication:*

   In the defined XOM application download protocol, the origin of A_C cannot be authenticated.

5. *Freshness:*

   The freshness of the application received from the software provider cannot be verified.

6. *Confidentiality and integrity protection of the cryptographic keys used in the prevention of unauthorised reading of and the detection of unauthorised modification to the application code and data:*

   (a) *Secure symmetric key generation:*

   In the XOM application download protocol the symmetric encryption compartment key, K, is generated by the software provider. Assuming the software provider also computes a MAC on A_C before encrypting it, the MAC key, N, is also generated by the software provider.

   (b) *Secure symmetric key transmission and storage:*

   A_C is encrypted under a symmetric compartment key, K, which in turn is encrypted by the software provider using the public encryption key associated with a specific tamper resistant XOM hardened processor, XOM_HP. Assuming the software provider also computes a MAC on A_C before encrypting it, K and N are encrypted by the software provider using the public encryption key associated with a specific tamper resistant XOM hardened processor, XOM_HP. In this way, only the intended XOM hardened processor, in which the corresponding private key is embedded, can decrypt the symmetric compartment key K and the MAC key N. There is no mention, however, in [98, 99] of the accreditation infrastructure that is necessary in order to support the secure use of public key pairs, as described above.

   The MACing key and the compartment key used to MAC and encrypt

A_C remain encrypted on the mobile receiver until they are decrypted using S_{XOM_HP} and loaded into the XOM key table, see appendix B. However, no mechanisms are described in [98,99] to prevent unauthorised modification of the keys sent by the software provider (assuming the software provider also computes a MAC on A_C before encrypting it). As a result, an attacker can:

- replace A_C with a malicious application, A_M;
- compute a MAC on A_M using a key, N^*, generated by the attacker, $\text{MAC}_{N^*}(A_M)$;
- encrypt A_M and $\text{MAC}_{N^*}(A_M)$ using a key, K^*, generated by the attacker, $E_{K^*}(A_M \mid\mid \text{MAC}_{N^*}(A_M))$; and then
- encrypt the newly generated symmetric keys, K^* and N^* using the public key of *XOM_HP*.

(c) *Prevention of unauthorised access to the symmetric key(s):*

The symmetric compartment key K, and N (assuming the software provider also computes a MAC on A_C before encrypting it), can only be decrypted by the intended XOM processor using S_{XOM_HP}, where S_{XOM_HP} is protected in hardware in *XOM_HP*. When the symmetric keys are decrypted using S_{XOM_HP} they are loaded into the XOM key table, which is stored in protected memory on *XOM_HP*.

7. *Confidentiality and integrity protection of the application code and data during execution:*

On-chip caches and registers: Within the XOM chip, all XOM data and code in caches and registers is tagged with a unique XOM identifier, which is mapped to the associated application's decrypted symmetric compartment key in the XOM key table. Programs that run in the clear have

a XOM identifier of 0. The size of the compartment key table and the number of XOM identifier tags affect how many concurrently executing principals can have data in the machine. At any one time, however, there will be only:

- One active principal;
- One active XOM identifier; and
- One active compartment key.

When this active principal produces data, it is automatically tagged with the active XOM identifier. Subsequently, if an attempt is made by an active principal to read data, the tag on the data is compared with the active XOM identifier, and access is only permitted if these values are identical. By virtue of the fact that on-chip caches and registers are embedded within a tamper resistant chip, they are also secure from physical attack.

Context switching and interrupt support: Secure interrupts and context switching are also supported by the XOM machine. When an interrupt occurs, the register content for a XOM program remains in the registers, and the tags, as mentioned above, prevent an untrusted and potentially malicious OS from reading the register values. The OS may then issue an instruction which causes register values to be MACed, encrypted, and cleared. Register contents are encrypted using the current mutating register key, which is associated with the active XOM identifier in the XOM key table. This allows an operating system to schedule and interrupt XOM processes without violating the security of the XOM application.

A mutating register key is used in order to prevent the replay of register

values on interrupts and context switches. If a static register key were used, an adversarial OS could first interrupt a running process and save the register state. The adversarial OS could then restore the process state and restart the process. At a later time, the adversarial OS could interrupt the process again, but instead of restoring the register values from the most recent interruption, restore the values from a previous interruption. When the process restarts, it will be using the replayed register values; see appendix B.4 for further details.

To counteract this, a key, called the register key, is changed every time a particular XOM compartment is interrupted. Therefore, the register key used to protect the register values on the first interrupt will no longer be valid when a register value is being restored after a second interrupt. Trying to restore a replayed state value will therefore result in an exception.

While it is stated that the register values must be MACed and encrypted before they are interrupted, no details of the key that must be used to MAC the register values before they are encrypted are given in [98, 99]. It is advisable that a separate MACing key is used to MAC the register values, before they are encrypted using the mutating register key.

External memory: Newly generated application data, when sent to external off-chip memory, must also be protected. In order to achieve this, application data is MACed and encrypted using the symmetric compartment key associated with the active XOM application identity in the XOM key table, before it is exported. It is suggested that the compartment key may be used for MACing and encryption. Alternatively, it is stated that a separate key may be used for MACing, but how or where this key is generated is not defined. It is, however, advisable that separate MACing and encryption keys are used.

An attacker may also, however, try to replay the data securely stored in external memory. To accomplish this, the attacker waits for the operating system to record MACed and encrypted data to memory, and then overwrites the same location at a later time with the old MACed and encrypted data.

To defend against this attack, a hash of the region of memory is made and stored in a secure register. To replay a specified region in memory, its associated hash stored in an on-chip register must also be replayed, but the anti-replay mechanism used to protect register values, as described above, defends against this type of attack. The overhead associated with this mechanism may become excessive if the region of memory is large, or if the values in the region change frequently. If, however, Merkle hash trees, as described in section 1.5.1, are deployed for memory authentication then the performance impact may be lessened.

It is stated in [99] that the most secure, but far from the most efficient, implementation of hash trees is to calculate the hash function every time a user writes to cache, and to verify the hash every time a value is read into the cache from memory. This process is described in further detail in appendix B.4.

There are two reasons why shortcomings in this protocol arise. Firstly, the XOM designers do not require that security services 3, 4 and 5, as described in section 6.3.2, are met by their download protocol. The generic requirement explicitly listed by the designers, i.e. to support the copy and tamper resistant download and execution of software, does however necessitate that security services 1, 2 and 6 and 7, as described in section 6.3.2, are met. Of these four security services, the XOM download protocol meets security service 7 but only

partially meets security services 1, 2 and 6, as described above. It appears that the second reason for the protocol's security shortcomings is due to the designers focus on ensuring that their architecture and download protocol supports the copy and tamper resistant execution of software, rather than the copy and tamper resistant download and execution of software.

9.6.4 Proposed security enhancements/clarifications

Here we propose a number of enhancements to the XOM protocol which have been designed to address the shortcomings identified in section 9.6.3.

- *Integrity protection of the application code and data:*

  For our application, it is required that the integrity of A_C is protected. We therefore require that, before the application is encrypted under the symmetric compartment key, it is MACed using an independent MACing key N, as follows, $E_K(A_C \mid\mid \mathrm{MAC}_N(A_C))$. This MAC key N must then be securely transmitted to the mobile host using mechanisms which protect both the integrity and the confidentiality of N.

  It would be helpful if it could be made clear by the XOM designers whether incoming XOM code is just encrypted or both MACed and encrypted. Current papers, referenced in appendix B, are ambiguous. Use of separate MACing and encryption keys is also advisable.

- *Origin authentication:*

  In order to meet this requirement, the software provider should be required to sign the encrypted symmetric compartment key used to encrypt A_C before it is transmitted, as follows, $S_S(E_{P_{XOM\_HP}}(K)) \mid\mid E_K(A_C)$. Preferably, the encrypted symmetric compartment and MACing keys would be signed, as follows, $S_S(E_{P_{XOM\_HP}}(\ K \mid\mid N)) \mid\mid E_K(A_C \mid\mid \mathrm{MAC}_N(A_C))$. In

this scenario, it would also be required that the signature of the software provider on the symmetric keys is verified by A_D using Cert_S before the symmetric keys are decrypted.

- *Freshness:*

  Assuming we include the signature of the software provider on the keys protecting A_C, it may initially appear that, if a protected XOM application is replayed, the host only learns something he already knows. However, instead of being sent the new application requested, an old version of the requested application, containing security vulnerabilities, could be replayed by an adversary, and accepted by the mobile device. This attack can be prevented by concatenating a timestamp with the application before it is encrypted.

- *Confidentiality and integrity protection of the cryptographic keys used in the prevention of unauthorised reading of and the detection of unauthorised modification to the application code and data:*

  - *Secure symmetric key transmission and storage:*

    As was the case with the TCG-defined system architecture, if XOM hardened processors were to be deployed, these hardened processors would need to be tested, and their conformance with the definition of a trustworthy XOM hardened processor validated. The public encryption key would then have to be certified/endorsed, i.e. associated with a statement regarding the ability of the processor to perform specific tasks.

    By encrypting K and N using the public encryption key associated with a specific tamper resistant XOM hardened processor, *XOM_HP*, the software provider can be assured that only the intended XOM

hardened processor, in which the corresponding private key is embedded, can decrypt the symmetric compartment key K and the MAC key N, assuming that the software provider has verified that the public key sent in the application request message has originated from a legitimate XOM hardened processor, using certificates generated as part of the aforementioned supporting accreditation infrastructure.

In order to prevent unauthorised modification of the keys used to protect the application, they must not only be encrypted using the public key of the *XOM_HP*, but also signed using the private key of the software provider.

In order to prevent unauthorised modification and copying of the keys sent by the software provider, the symmetric MACing key and the symmetric compartment key used to encrypt A_C should remain both signed and encrypted on the mobile receiver until their use.

9.7 The AEGIS application download protocol

The AEGIS architecture and application download protocol were proposed in order to meet two security requirements — to support the download and execution of both tamper evident, and copy and tamper resistant software. For the purpose of our application, we require the copy and tamper resistant download and execution of a conditional access application.

9.7.1 The AEGIS system architecture

Within the AEGIS system architecture, both tamper evident and private tamper resistant environments can be provided for multiple mistrusting processes. Tamper evident environments are defined as "authenticated environments, where physical or software tampering can be detected" [143]. Private tamper resistant

environments are defined as "private and authenticated environments where an adversary cannot gain any information about data or software within the environment by tampering with or observing system operation" [143].

The platform subsystem used to enable tamper evident or private tamper resistant execution of applications may be implemented in one of two ways. The first secure computing model assumes a hardened AEGIS processor and an untrusted operating system, whereas the alternate model assumes a hardened AEGIS processor and a security kernel, which runs at a higher privilege level than the regular operating system. In our analysis of this particular protocol, we refer to the trusted computing base (TCB), which consists of an AEGIS hardened processor and, optionally, the security kernel. For further details see appendix B.

9.7.2 The AEGIS download protocol

The AEGIS application download protocol is as follows:

1. $A_D \rightarrow S$: Request for $A_C \mid\mid P_{AEGIS\_HP} \mid\mid (Id_{SK})$.
   Id_{SK} is only included if the host TCB is comprised of the *AEGIS_HP* and a security kernel.

2. S : Verifies P_{AEGIS_HP} as belonging to a legitimate *AEGIS_HP* by retrieving and verifying Cert_{AEGIS_HP} using the public key of the manufacturer of *HP*.

3. S : Retrieves $H(SK)$, where the $H(SK)$ represents a configuration of the security kernel, identified by Id_{SK}, which S considers to be trustworthy. This step need only be completed if the TCB of the mobile device from which the request initiated consists of an *AEGIS_HP* and a security kernel, *SK*, as indicated in step 1.

4. S : Generates a static key K.

5. S : Computes the $H(A_C)$.

6. S : Encrypts A_C using *AEGIS_HP*.

7. S : Encrypts $(H(SK) \mid\mid H(A_C) \mid\mid K)$, using P_{AEGIS_HP} or, alternatively,
   S : Encrypts $(H(A_C) \mid\mid K)$, using P_{AEGIS_HP}.

8. $S \rightarrow A_D : E_{P_{AEGIS\_HP}}(H(SK) \mid\mid H(A_C) \mid\mid K) \mid\mid E_K(A_C)$; or
   $S \rightarrow A_D : E_{P_{AEGIS\_HP}}(H(A_C) \mid\mid K) \mid\mid E_K(A_C)$;

9. $AEGIS\_HP$: Decrypts $E_{P_{AEGIS\_HP}}(H(SK) \mid\mid H(A_C) \mid\mid K)$ using S_{AEGIS_HP}, or, alternatively,
   $AEGIS\_HP$: Decrypts $E_{P_{AEGIS\_HP}}(H(A_C) \mid\mid K)$, using S_{AEGIS_HP}.

10. If there is a security kernel within the mobile device TCB:
    $AEGIS\_HP$: Compares the hash value of the security kernel running on the platform and measured at boot time $(H(SK)^*)$ with the value of $H(SK)$ decrypted in step 9.

    (a) If $H(SK)^* = H(SK)$, then $H(A_C)$ and K are released, and the A_C can be decrypted by the SK which is part of the TCB, as described in step 12 below.

    (b) If $H(SK)^* \neq H(SK)$, then $H(A_C)$ and K are discarded.

11. If the TCB contains only an $AEGIS\_HP$, or following step 10:
    TCB: Decrypts $E_K(A_C)$ using K.

12. TCB: Recomputes the hash value of A_C, $H(A_C)^*$, and compares it to the value of $H(A_C)$, decrypted in step 9.

    (a) If $H(A_C)^* = H(A_C)$, then A_C can be executed.

    (b) If $H(A_C)^* \neq H(A_C)$, then A_C is discarded.

13. TCB: ensures the protected execution of A_C

Using the AEGIS system architecture, the protocol begins when A_D requests download of the application, A_C. In conjunction with the request for A_C, the public key of *AEGIS_HP* is sent from A_D to S. If the TCB of the mobile receiver consists of both an AEGIS hardened processor and a security kernel (SK), the identity of the SK (Id_{SK}) running on the mobile receiver must also be communicated to the software provider.

S verifies that the public key received belongs to a genuine AEGIS hardened processor. The software provider then retrieves $H(SK)$, where $H(SK)$ represents a configuration of the security kernel running on the mobile receiver, identified by Id_{SK}, which S considers to be trustworthy (if indeed Id_{SK} was included in the request message).

The software provider then generates a symmetric key K, known as a static key, and encrypts A_C using K. The software provider then composes one of the following messages: $E_{P_{AEGIS\_HP}}(H(A_C) \mid\mid K) \mid\mid E_K(A_C)$; or $E_{P_{AEGIS\_HP}}(H(SK) \mid\mid H(A_C) \mid\mid K) \mid\mid E_K(A_C)$. The first message is sent to a device into which *AEGIS_HP* is embedded. If it has been indicated in the request message that a particular SK is also running on the platform, the second message is sent, where $H(SK)$ represents the configuration of the security kernel which the software provider deems to be trustworthy.

When this message is received by A_D, the encrypted bundle is loaded into *AEGIS_HP*. If $E_{P_{AEGIS\_HP}}(H(A_C) \mid\mid K) \mid\mid E_K(A_C)$ has been received, $(H(A_C) \mid\mid K)$ can only be recovered using the hardened processor's secret decryption key if it has been received by the intended AEGIS processor.

In the case that $E_{P_{AEGIS\_HP}}(H(SK) \mid\mid H(A_C) \mid\mid K) \mid\mid E_K(A_C)$, has been received by *AEGIS_HP*, $(H(SK) \mid\mid H(A_C) \mid\mid K)$ can only be decrypted using

the hardened processor's secret decryption key if it has been received by the intended AEGIS processor. $H(A_C) \mid\mid K$ is then released only if the identity of the security kernel running on the device matches the value of $H(SK)$ sent by S.

The static key is then used to decrypt $E_K(A_C)$. When decrypted, a hash of A_C is generated, and if the output matches the value of $H(A_C)$ sent by the software provider, the static key is assigned to the A_C in the key table in the TCB. This process is made possible by two AEGIS instructions which are integrated into the protected application, *enter_aegis*, which is used to start the execution in a tamper evident environment, and *set_aegis_mode*, which is used to enable a private tamper resistant environment from tamper evident mode. We require that A_C is executed in a PTR environment. During this process, a dynamic key is also associated with A_C in the key table in the TCB. This key is randomly chosen by the TCB when *enter_aegis* is called, and is used to encrypt and decrypt data that is generated during program execution.

9.7.3 Security analysis

We now analyse the AEGIS trusted download process against the security service requirements specified in section 6.3.

1. *Confidentiality of the application code and data:*

   Symmetric encryption is deployed to confidentiality protect A_C, where A_C is encrypted with the symmetric static key K chosen by S. The confidentiality of A_C is also dependent, however, on the confidentiality of K, which is analysed below.

2. *Integrity protection of the application code and data:*

   The authors use a hash of A_C encrypted under the public key of $AEGIS\_HP$

to protect the integrity of A_C during transmission. When the encrypted bundle is received, the hash of the decrypted application is compared against the hash value received from the software provider. Any discrepancy found indicates that the application may have been maliciously or accidently modified.

The integrity of A_C is also dependent, however, on the integrity of $H(A_C)$ being maintained both during transmission and while in storage on the mobile device. The integrity of $H(A_C)$ cannot be guaranteed using the mechanism described above, and therefore the application code and data is not integrity-protected. This is examined further.

3. *Entity authentication:*

   Neither authentication of the AEGIS processor by the software provider nor authentication of the software provider by the AEGIS processor is provided by the AEGIS download protocol.

4. *Origin authentication:*

   The origin of A_C cannot be authenticated using the AEGIS download protocol.

5. *Freshness:*

   The freshness of the application received from the software provider cannot be verified.

6. *Confidentiality and integrity protection of the cryptographic key and the hash used in the prevention of unauthorised reading of and the detection of unauthorised modification to the application code and data:*

   (a) *Secure symmetric key generation:*

In the AEGIS application download protocol the encryption key, K, is generated by the software provider.

(b) *Secure symmetric key and integrity verification data transmission and storage:*

A_C is encrypted under a static key, K, which is in turn encrypted by the software provider using the public encryption key associated with a specific tamper resistant AEGIS hardened processor, $AEGIS\_HP$. In this way, only the intended AEGIS processor, in which the corresponding private key is embedded, can decrypt the symmetric static key, assuming that the software provider has verified that the public key sent in the application request message has originated from a legitimate AEGIS hardened processor, $AEGIS\_HP$. The value $H(A_C)$ is also encrypted by the software provider using the public encryption key associated with a specific tamper resistant AEGIS hardened processor, $AEGIS\_HP$.

In the case that $E_{P_{AEGIS\_HP}}(H(SK) \mid\mid H(A_C) \mid\mid K) \mid\mid E_K(A_C)$, has been received by $AEGIS\_HP$, $(H(SK) \mid\mid H(A_C) \mid\mid K)$ can only be decrypted using the hardened processor's secret decryption key if it has been received by the intended AEGIS processor. $H(A_C) \mid\mid K$ is then released only if the identity of the security kernel running on the device matches the value of $H(SK)$ sent by S.

The symmetric compartment key used to encrypt A_C and $H(A_C)$ remains encrypted on the mobile receiver until it is decrypted using S_{AEGIS_HP} and loaded into $AEGIS\_HP$'s key table.

In the description [143] of the AEGIS system architecture, however, there is no mention of the accreditation infrastructure that is necessary in order to support the secure use of public key pairs as described

above.

In conjunction with this, no measures are taken to prevent unauthorised modification of the key K and the value of $H(A_C)$ sent by the software provider. As a result, an attacker can:

- replace A_C with a malicious application, A_M;
- compute the hash of A_M, $H(A_M)$;
- encrypt A_M using a key, K^*, generated by the attacker, $E_{K^*}(A_M)$; and then
- encrypt the newly generated key, K^*, and hash, $H(A_M)$, using the public key of *AEGIS_HP*.

(c) *Prevention of unauthorised access to the symmetric key and integrity verification data:*

The encrypted symmetric static key K and $H(A_C)$ can only be decrypted by the intended AEGIS processor using S_{AEGIS_HP}, where S_{AEGIS_HP} is protected in *AEGIS_HP*.

If the mobile host system architecture is composed of a security kernel, its identity may also be included in the encrypted string. In this case, the static key K and $H(A_C)$ are only decrypted and output when the processor contains the corresponding public decryption key, and the identity of the security kernel matches that sent by the software provider.

When the symmetric static key K and $H(A_C)$ are decrypted using S_{AEGIS_HP}, they are loaded into a table stored in protected memory on *AEGIS_HP*.

7. *Confidentiality and integrity protection of the application code and data during execution:*

For our particular application, it is required that A_C runs in a PTR environment.

Process start-up: The *enter_aegis* instruction is used to enter tamper evident mode. Immediately after the *enter_aegis* instruction is executed, a piece of code is executed, which guarantees that the initial state of the program, in this instance A_C, is properly set-up. This code may reside in the security kernel or, indeed, in the AEGIS processor, depending on the chosen implementation of the AEGIS subsystem. Therefore, either the AEGIS processor, or the security kernel, takes a hash of the program A_C after it has been decrypted, so that this hash can be compared to the copy of A_C sent by the software provider. Either the AEGIS processor or the security kernel then checks any other code or data that the application relies on. This code must also check the environment it is running in, for example, whether its mode is TE or PTR.

On-chip caches: In terms of physical or hardware attacks, it is assumed by Suh et al. [143] that an adversary cannot physically tamper with on-chip caches; see appendix B.5 for further details. In the case where the TCB is comprised of *AEGIS_HP*, on-chip caches are protected from buggy or malicious software using tags. Whenever a process accesses a cache block, the block is tagged with the identity of that process. On future accesses the identity of the active process is compared to the identity the cache block has been tagged with, and access is only granted if they match. In the case of a PTR environment, access is only granted if the active ID matches the cache tag, and the active process is in PTR mode. In the case where the TCB is comprised of both *AEGIS_HP* and a security kernel, it is assumed that on-chip caches are protected by a virtual memory manager integrated into the security kernel.

On-chip registers: In the PTR environment, all registers are private and protected. In the AEGIS system architecture, it is the TCB which saves/clears the registers on an interrupt and restores them on a resume. Therefore, there is no need to tag registers, as an untrusted and potentially malicious operating system will not have access to register values even following an interrupt or when context switching.

Context switching and interrupt support: In order to support context switching and interrupts, register values are securely saved to the TCB by the TCB. In the case of a PTR environment values are additionally flushed from the registers before a potentially untrusted interrupt handler starts, in order to protect the integrity and confidentiality of register values.

Externally stored memory values: Both hardware and software attacks on off-chip memory must also be considered. Confidentiality of newly generated cache data is protected via the symmetric encryption of cache blocks before their transfer to external memory, using the dynamic key associated with A_C in the key table in the TCB. This key is randomly chosen by the TCB when *enter_aegis* is called, and is used to encrypt and decrypt data that is generated during the program execution.

The integrity of dynamic data stored in untrusted off-chip memory is protected through the implementation of hash trees between the L2 cache and the encryption engine, see section 1.5.1.

Suh et al. [143] criticise the XOM scheme [99] for focusing primarily on the replay of registers and failing to recognise the potential for replay of data in memory. This criticism, however, appears to be unjustified. While Lie et al. do not explicitly detail their chosen method to prevent the replay of data in memory, it is briefly mentioned in [99], and thoroughly explored

in [98].

There are two reasons why the shortcomings of this protocol arise. Firstly, the AEGIS designers do not require that security services 3, 4 and 5, as described in section 6.3.2, are met by their download protocol. However, on examination of the generic requirement explicitly listed by the designers, i.e. to support the download and execution of copy and tamper resistant software, security services 1, 2 and 6 and 7, as described in section 6.3.2, must be met. Of these four security services, the AEGIS download protocol meets security service 7, but only partially meets security services 1, 2 and 6, as described above. As with XOM, it appears that the second reason for the protocol's security shortcomings is due to the designers focus on ensuring that their architecture and download protocol supports the copy and tamper resistant execution of software rather than the copy and tamper resistant download and execution of software.

9.7.4 Proposed security enhancements/clarifications

Here we propose a number of enhancements to the AEGIS protocol, which have been designed to address the shortcomings identified in section 9.7.3.

- *Origin authentication:*

  Using the AEGIS download protocol, the origin of A_C cannot be authenticated. In order to meet this requirement, the software provider should be required to sign $E_{P_{AEGIS\_HP}}(H(A_C) \mid\mid K)$ or
  $E_{P_{AEGIS\_HP}}(H(SK) \mid\mid H(A_C) \mid\mid K)$ before its transmission. In this scenario, it would also be required of A_D to verify the signature of the software provider on the encrypted bundle using Cert_S, before the symmetric key K, $H(A_C)$ and, potentially, $H(SK)$, are decrypted by *AEGIS_HP*.

- *Freshness:*

The freshness of the application received from the software provider cannot be verified. Assuming the signature of the software provider is computed on $E_{P_{AEGIS\_HP}}(H(A_C) \mid\mid K)$ or $E_{P_{AEGIS\_HP}}(H(SK) \mid\mid H(A_C) \mid\mid K)$ before its transmission, it may initially appear that, if A_C is replayed, the host only learns something he already knows. However, instead of being sent the new application as requested, an old version of the application, containing security vulnerabilities, may be replayed by an adversary and accepted by the mobile host. By concatenating a timestamp with the application before it is encrypted, this attack can be prevented.

- *Confidentiality and integrity protection of the cryptographic key and the hash used in the prevention of unauthorised reading of and the detection of unauthorised modification to the application code and data:*

  - *Secure symmetric key and integrity verification data transmission and storage:*

    As was the case with the TCG-defined system architecture, if AEGIS processors were to be deployed, these hardened processors would need to be tested and their conformance with the definition of a trustworthy AEGIS processor validated/endorsed. The public encryption key would then have to be certified, i.e. associated with a statement regarding the ability of the processor to fulfil specific tasks. Certificates which illustrate configurations for security kernels which are behaving as expected would also be required.

    A_C is encrypted using a static key, K, which is in turn encrypted by the software provider using the public encryption key associated with a specific tamper resistant AEGIS hardened processor, *AEGIS_HP*. In this way, only the intended AEGIS processor, in which the corresponding private key is embedded, can decrypt the symmetric static

key, assuming that the software provider has verified that the public key sent in the application request message has originated from a legitimate AEGIS hardened processor, *AEGIS_HP*, using certificates generated as part of the aforementioned supporting accreditation infrastructure. The value $H(A_C)$ is also encrypted by the software provider using the public encryption key associated with a specific tamper resistant AEGIS hardened processor, *AEGIS_HP*.

Unless K and $H(A_C)$ are both encrypted and signed by the software provider, an attacker can replace A_C with a malicious application A_M, compute $H(A_M)$, encrypt A_M using a key generated by the attacker, and then encrypt the newly generated symmetric key and $H(A_M)$ using the public key of the destination AEGIS hardened processor, *AEGIS_HP*.

$H(A_C)$ and K should remain both signed and encrypted on the mobile receiver until their use.

9.8 Conclusions

This chapter described two application download protocols that have been proposed by the designers of XOM and AEGIS, Lie et al. and Suh et al. Both protocols are based upon the assumption that the host device contains a hardened processor rather than a trusted module, as assumed in chapters 7 and 8. Both protocols were then analysed against the security requirements described in chapter 6. These analyses have revealed security shortcomings in both of these protocols. We subsequently proposed a series of enhancements to the protocols designed to address the identified shortcomings.

The implementation of a secure application download protocol using either

the XOM and AEGIS system architectures meets some of the security requirements described in chapter 6. While neither protocol meets all of the requirements for the secure transmission of A_C, the protocols can easily be modified to fulfil the most crucial of these requirements, as suggested in sections 9.6.3 and 9.7.3. The requirements surrounding the secure execution of A_C are met, however, and strong isolation of A_C is provided when executed on the hardened processors described.

Part III

Remote code protection

Chapter 10

OMA DRM

Contents

10.1 Introduction 343
10.1.1 The MPWG 343
10.1.2 Digital rights management 344
10.1.3 Scope of part III 345
10.2 DRM 346
10.3 The OMA 347
10.4 Model 348
10.4.1 Functional entities 348
10.4.2 Functional components 349
10.4.3 Functional architecture 350
10.5 OMA DRM v1 350
10.6 OMA DRM v2 351
10.7 Conclusions 354

This chapter provides an overview of Digital Rights Management, with particular focus on the Open Mobile Alliance DRM standards. The model to which the OMA DRM architecture applies is introduced. A high level critique of OMA DRM version 1 is given, followed by an examination of the OMA DRM version 2 specification set.

10.1 Introduction

10.1.1 The MPWG

"At the current time, the number of applications that use trusted computing is quite limited, both in volume and in scope" [104]. However, as the advantages of integrating trusted computing functionality into a wide range of devices have become more apparent, the baseline TCG specification set has been expanded by platform-specific working groups to include specifications describing specific platform implementations for the PC client, servers, peripherals and storage systems.

One such working group is the mobile phone working group, the main challenge for which is to determine the 'roots of trust', see section 1.7, required within a trusted mobile phone. The functionality provided by each of these roots of trust must also be specified. In order to identify the capabilities required of a trusted mobile phone, a number of use cases, whose secure implementation may be aided by the application of trusted platform functionality, have been identified by the MPWG. Among the use cases described are SIMLock, device authentication, mobile ticketing, mobile payment and robust DRM implementation [159]. As stated by the MPWG [159], the use cases lay a foundation for the ways in which:

- The MPWG will derive requirements that address situations described in the use cases.
- The MPWG will specify an architecture based on the TCG architecture that will meet these requirements.
- The MPWG will specify the functions and interfaces that will meet the requirements in the specified architecture.

10.1.2 Digital rights management

Part II of this thesis focused on the next generation of communications systems, which are expected to collaborate with broadcast systems to provide wireless access to streamed video content from a wide range of mobile devices. Currently 3G systems are already capable of delivering a wide range of digital content to subscribers' mobile telephones, for example, music, video clips, ring tones, screen savers or java games.

As network access becomes ever more ubiquitous, and media objects become more easily accessible, providers are exposed to the risks of illegal consumption and use of their content. Just as conditional access systems were developed in order to ensure that only authorised entities have access to broadcast content, as described in chapter 6, similar rights management solutions are available in order to facilitate the safe distribution of various forms of digital content in a wide range of computing environments, and to give assurance to content providers that their media objects cannot be illegally accessed.

A Digital Rights Management system is an umbrella term for mechanisms used to manage the lifecycle of digital content of any sort. A DRM agent, i.e. the DRM functionality of a device responsible for enforcing permissions and constraints associated with protected content, must be trusted in terms of its correct behaviour and secure implementation [120]. Stipulation of a trust model, within which robustness rules are defined, is one method of specifying how secure a device implementation of a DRM agent must be, and what actions should be taken against a manufacturer that builds devices that are insufficiently robust [73].

10.1.3 Scope of part III

In this part of the thesis we examine the evolution of the OMA DRM specification set, with particular focus on OMA DRM v2. The security threats that may impact upon devices, and protected content received by devices, on which an OMA DRM v2 agent is not robustly implemented, are extracted. This enables the derivation of requirements for a robust implementation of OMA DRM v2. Following this, a description is given of the architectural components, based on the TCG architecture, and the functions and interfaces, as specified in the current TPM and TSS specifications, which meet these requirements. This enables any architecture components, functions or interfaces not currently defined within the TCG specification set, but required for the implementation of a robust and secure DRM solution on a trusted mobile platform, to be identified.

The content of chapters 10 to 12 was contributed to the TCG MPWG. A concise version of the use case described in this chapter has subsequently been included in the TCG MPWG use case scenarios [159]. The functionality required of a trusted mobile platform on which an OMA DRM v2 agent is to be robustly implemented, defined in chapter 11, has been incorporated in an internal TCG MPWG document describing the requirements for a trusted mobile platform. Finally, the analysis in chapter 12, completed in order to define which architectural components and functionality described within TCG version 1.2 specification set may be used to facilitate a robust implementation of OMA DRM v2, and in order to identify any architecture components and functionality not currently defined within the TCG specification set but required for the implementation of a robust and secure DRM solution on a trusted mobile platform, contributed towards the TCG mobile TPM commands and structures specification document, publication of which is expected in late 2006.

This chapter provides an overview of DRM with particular focus on the OMA DRM standards. Section 10.2 gives the rationale for DRM solutions and outlines the generic components of a DRM system. Section 10.3 introduces the OMA, while section 10.4 describes the model to which the OMA DRM architecture applies. Section 10.5 provides an overview of the first version of the OMA DRM specification set, i.e. OMA DRM v1. Section 10.6 examines OMA DRM v2.

10.2 DRM

DRM solutions are designed to allow the distribution of digital content to clients with some assurance that the client will use the content according to conditions set by the content owner [73]. DRM has often been separated into two functional areas [31]:

- The identification and description of intellectual property and the rights pertaining to works and to parties involved in their creation or administration (digital rights management); and
- The (technical) enforcement of usage restrictions (digital management of rights).

A DRM system may therefore consist of a wide variety of technologies and services, which contribute to one or other of the functional areas of DRM. The most fundamental of these technologies and services are described in [31], and are listed below.

- Identification technologies ensure that every item within a DRM system has a unique label, so that unambiguous identification, see section 1.7, may be completed across computer systems.
- Metadata technologies facilitate the description of digital content.

- Rights language technologies describe rights associated with content.
- Encryption technologies protect digital content against unauthorised access.
- Persistent association technologies facilitate the permanent association of metadata with content.
- Privacy technologies mitigate threats against the confidentiality and privacy of user personal data.
- Payment technologies provide secure and usable payment methods for digital content.

10.3 The OMA

The Open Mobile Alliance was founded in June 2002. One of the original objectives of the OMA was to define a DRM specification set for use in a mobile environment. OMA DRM v1 was published as a candidate specification in October 2002, and was approved as an OMA enabler specification [123], after full interoperability testing had been completed in 2004.

Following this, in 2004, work on OMA DRM v2 was completed and OMA DRM v2 was published as a candidate specification in July 2004 [124]. OMA DRM v2 builds upon the version 1 specifications to provide higher security and a more extensive feature set [73]. Devices other than mobile phones are also supported by OMA DRM v2. The OMA DRM version 2 specification set defines [120]:

- the format and the protection mechanism for protected content;
- the format and the protection mechanism for rights objects;

- the security model for the management of encryption keys; and
- how protected content and rights objects may be transferred to devices using a range of transport mechanisms.

10.4 Model

Next, we examine the model to which the OMA DRM architecture applies. The model under consideration is taken from [120] and is illustrated in figure 10.1.

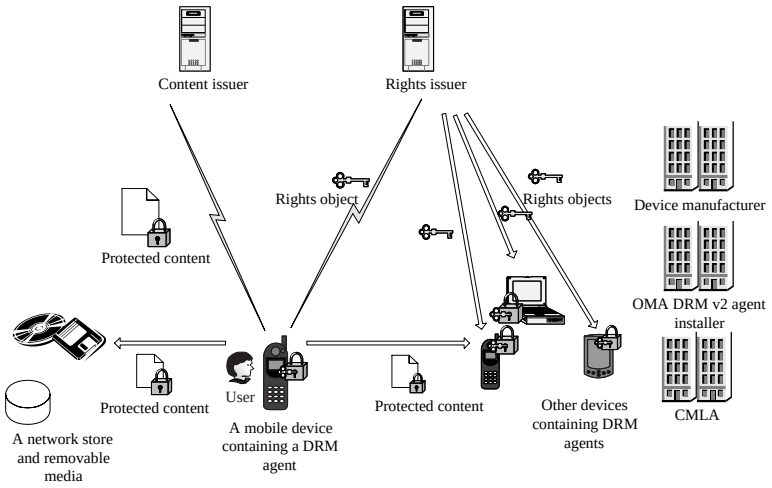

Figure 10.1: Architecture model

10.4.1 Functional entities

The following functional entities may exist within an OMA DRM system.

- An agent installer is responsible for the 'robust' implementation of an OMA DRM v2 agent on a device.
- A device manufacturer is responsible for the manufacture of devices. The device manufacturer may in practice be the agent installer.

- An OMA DRM implementation compliance authority provides a set of robustness rules necessary to support the OMA DRM system. Implementations of the OMA DRM specification set can then be evaluated against the defined rule set as either robust or not. The content management licensing administrator for digital rights management (CMLA DRM) is an example of one such authority. This entity only came into existence with the adoption of OMA DRM v2.

- A user denotes a human user of content. Users can only access protected content through a DRM agent.

- A DRM agent is defined as an entity, present in a device, that is responsible for enforcing permissions and constraints associated with content, and for controlling access to protected content [120].

- A content issuer (CI) is an entity that delivers content. OMA DRM defines the format of content delivered to DRM agents, and the way protected content can be transported from a content issuer to a DRM agent using various transport mechanisms [120].

- A rights issuer (RI) is an entity that assigns permissions and constraints to content, and generates rights objects. A rights object is an XML document expressing permissions and constraints associated with a piece of content [120].

10.4.2 Functional components

We now move on to examine the functional components of OMA DRM systems, as defined by the OMA [120].

- A device is defined as user equipment on which a DRM agent is installed.

- A rights object is a collection of permissions and other attributes which are linked to protected content.
- A media object is a digital work, for example, a ring tone, screen saver, java game or a composite object, which itself contains one or more media objects.

10.4.3 Functional architecture

A user requests a media object from a content issuer. The requested content, which is packaged in order to prevent unauthorised access, is then sent to the user's device. The packaging of the content may be completed by the content issuer or, alternatively, by the content owner, before it is dispatched to the content issuer. The rights object associated with the requested media object is delivered to the user by the rights issuer. This rights issuer may, in practice, be the same entity as the content issuer.

10.5 OMA DRM v1

Version 1 of the OMA specifications [119, 121] represents the initial attempt to define a DRM solution for a mobile environment. Three main goals were specified for OMA DRM v1 [73]. The solution was required to be timely and inexpensive to deploy. It was also required to be easy to implement on mass market mobile devices. Finally, it was required that the initial OMA DRM solution did not necessitate the roll-out of a costly infrastructure. In the development of OMA DRM v1 a trade-off was made, so that the objectives listed above could be met ahead of some security requirements.

Three classes of DRM functionality are specified in OMA DRM v1 [119,121]. The first class of DRM functionality, forward lock, must be supported by an

OMA DRM v1 agent on a device. Provision of the second and third classes of DRM functionality, i.e. combined delivery and separate delivery, by an OMA DRM v1 agent is optional.

1. Forward lock prevents unencrypted content being forwarded from the device to which it was initially delivered. The protected content is wrapped inside a DRM message, which indicates to the OMA DRM v1 agent on the receiving device that the content is not to be forwarded. Protection is dependent on the OMA DRM v1 agent acting accordingly.

2. Combined delivery involves wrapping unencrypted content and its associated rights object inside a DRM message.

3. Separate delivery involves the separate delivery of encrypted content and the associated rights object. The content is encrypted and placed in a container, in a format known as the DRM container format (DCF). Headers, which allow a receiving device to associate the correct rights object with the corresponding DCF object, are also contained in this file. The associated rights object, which contains the relevant permissions and constraints, and the decryption key for the associated content, is delivered via SMS.

10.6 OMA DRM v2

OMA DRM v2 [120, 122] builds upon the original OMA DRM v1 specification set, with the primary objective of providing a more secure DRM solution. The following security vulnerabilities have been identified in OMA DRM v1 [73].

1. A rights issuer has no way of determining whether the requesting device supports DRM. When using the forward lock and combined delivery features, where the content is not encrypted, this particular security vul-

nerability enables an attack in which unencrypted content is initially sent to a PC made to look like a compliant phone. On receipt, content is then extracted and illegally distributed.

2. In the separate delivery DRM class, where the content is encrypted, the content encrypting key is not protected. This implies that the attack described above in step 1 is also possible in this case, although it is more complex, and more difficult to complete successfully [73].

3. The device has no way of authenticating the rights issuer, and therefore may be sent bogus rights objects from an entity claiming to be the legitimate rights issuer.

OMA DRM v2 addresses the above security weaknesses through the deployment of additional security mechanisms.

- Both device authentication and rights issuer authentication are provided.
- Mechanisms are deployed in order to protect the confidentiality of media objects. Content is protected using a content encrypting key (CEK). This CEK is encrypted in a rights object under a rights object encrypting key (REK). In turn, the REK is encrypted using the public key of the device.
- Mechanisms are also deployed so that the OMA DRM v2 agent can determine whether a media object received from a RI has been modified in an unauthorised way.

The OMA DRM v2 specification set is no longer mobile device specific, as was the case with the v1 specifications. It also provides a richer feature set which includes, most notably [73]:

- Support for the automatic preview of protected content.

- Support for subscription services.
- Support for continuous media such as streaming and progressive download of content.
- Support for reward schemes.
- Support for domains. A domain, to which a specified number of devices can be added, may be established by a user. Following this, content and the associated access rights may be shared among the devices in this particular domain. In this case, rights objects must be explicitly acquired for the domain rather than a specific device. A RI may control the number of devices allowed in a domain, although the user is entitled to add and remove devices at will, as long as the limit set by the RI is adhered to.
- Support for unconnected devices. This is a feature supported by the implementation of domains. An unconnected device may be added to a domain, after which content and rights may be copied from a connected domain device to the unconnected device.

In order to provide the additional security features described above, a dedicated suite of DRM security protocols, the rights object acquisition protocol (ROAP) suite, was developed by the OMA.

In addition to the ROAP suite, it was agreed that the OMA DRM v2 specification set should be supported by a trust model. A trust model enables an RI to obtain assurances about DRM agent behaviour, and the robustness of the DRM agent implementation [120]. It is the responsibility of the CMLA DRM, or a similar organisation, to provide a trust model, i.e. robustness rules, and to define actions which may be taken against a manufacturer who builds devices which are not sufficiently robust.

10.7 Conclusions

In this chapter, an overview of DRM has been provided, with particular focus on the OMA DRM standards. The rationale for DRM solutions has been examined and the generic components of a DRM system outlined. The activities of the OMA have been briefly introduced, followed by a description of the model to which the OMA DRM architecture applies. An overview of OMA DRM v1 and v2 has also been provided.

Chapter 11

Requirements for a robust implementation of OMA DRM v2

Contents

11.1 Introduction 356
11.2 OMA DRM v2 agent installation 357
11.3 The rights object acquisition protocol (ROAP) suite 360
11.3.1 Notation 360
11.3.2 The 4-pass registration protocol 363
11.3.3 The rights acquisition protocols 368
11.3.4 The 2-pass join domain protocol 373
11.3.5 The 2-pass leave domain protocol 376
11.4 Conclusions 378

In this chapter the lifecycle of an OMA DRM v2 agent is considered. Each lifecycle stage is analysed in order to derive a list of security threats that may impact on devices, and protected content received by devices, on which an OMA DRM v2 agent is not robustly implemented. The functionality required of a trusted mobile platform on which an OMA DRM v2 agent is to be robustly implemented, thereby thwarting any threats to the DRM agent and its associated data, is also defined.

11.1 Introduction

Having described the model to which the OMA DRM architecture applies, and having briefly examined the core elements of OMA DRM v1 and v2 in chapter 10, we now consider certain aspects of the most recent version of the OMA DRM specifications in greater detail. More specifically, we examine OMA DRM v2 with a view to defining what functionality is required of a trusted mobile platform if it is to support a robust implementation of an OMA DRM v2 agent. The numbered list of functional requirements that is accumulated through the course of this chapter will be utilised in chapter 12 in two ways. Firstly, they enable us to determine the 'roots of trust', see section 1.7, required within a trusted mobile phone. Secondly, they enable the evaluation of the capabilities which must be provided by each of these roots of trust to support this particular use case.

Section 11.2 describes the process by which an OMA DRM v2 agent and its associated data are installed on a device. This process is analysed in order to extract any threats which may arise if the OMA DRM v2 agent is not robustly implemented. Following this, the functionality required of a trusted mobile device in order to thwart these threats is described.

Section 11.3 examines the fundamental steps in each of the protocols defined within the OMA ROAP suite [122]. Following each of the protocol descriptions, the threats which may impact upon the security of the protocols, if the OMA DRM v2 agent is not robustly implemented, are highlighted. As stated above, the functionality required of a trusted mobile device in order to thwart these threats is also described.

11.2 OMA DRM v2 agent installation

Before an OMA DRM v2 agent can be executed by a mobile device user to acquire protected content, it must be installed on the device. The steps described in table 11.1 must be completed when installing an OMA DRM v2 agent on a mobile device. It is assumed that this will take place at the time of manufacture.

Table 11.1: OMA DRM v2 agent installation

Step	Description
1	The OMA DRM v2 agent code must be installed on the device.
2	The OMA DRM v2 agent private key must be installed on the device.
3	The OMA DRM v2 agent private key must be stored on the device.
4	The OMA DRM v2 agent certificate (chains), the device details, i.e. the device manufacturer, model, and version number, and the trusted RI authorities certificate must be installed on the device.
5	The OMA DRM v2 agent certificate (chains), the device details and the trusted RI authorities certificate must be stored on the device.

Every OMA DRM v2 agent is equipped with a unique key pair [120]. The private key from this key pair is used by an OMA DRM v2 agent to generate digital signatures, so that a rights issuer can authenticate a particular DRM agent. The public key from this pair is used by rights issuers in order to distribute rights object (RO) encryption keys, which protect content encryption keys that are used to encrypt content, as described in section 10.6.

An associated certificate, which identifies the DRM agent and binds the agent to the public key described above, is also provided to the DRM agent. The OMA DRM v2 certificate may also be integrated into one or more certificate chains. The OMA DRM v2 certificate comes first in a chain, and each subsequent certificate contains the public key necessary to verify the certificate preceding it [124]. When a rights issuer, with whom the OMA DRM v2 agent is communicating, indicates its preferred trust anchor(s), the OMA DRM v2 agent must select and send a device certificate (chain) which points back to an

appropriate anchor [124], so that the RI can verify the OMA DRM v2 agent certificate.

The device details indicate the device manufacturer, model, and version number. Finally, the trusted RI authorities certificate is used to indicate which rights issuer trust anchor(s) are recognised by the OMA DRM v2 agent. This trusted RI authorities certificate may be a single root certificate, as is the case in the CMLA trust model [29], where the trusted RI authorities certificate is a self-signed CMLA root CA certificate, or, alternatively, may be a collection of self-signed public key certificates representing the preferred trust anchors of the OMA DRM v2 agent, see section 1.5.7.

Of the items described in table 11.1, the CMLA requires that the OMA DRM v2 agent private key is both confidentiality and integrity-protected, and that the device details and the trusted RI authorities certificate are integrity-protected [29].

Unless the device implementation of the OMA DRM v2 agent is robust, a number of threats may impact on the device, and ultimately on the protected content received by the device. These threats include:

- Unauthorised modification of the OMA DRM v2 agent code on installation onto the device.
- Unauthorised modification of the OMA DRM v2 agent code while in storage on, or while executing on, the device.
- Unauthorised reading/copying of the OMA DRM v2 agent private key on installation into the device.
- Unauthorised reading/copying of the OMA DRM v2 agent private key while in storage on the device.

- Unauthorised modification of the OMA DRM v2 agent private key, the OMA DRM v2 agent certificate (chains), the device details or the trusted RI authorities certificate on installation into the device.
- Unauthorised modification of the OMA DRM v2 agent private key, the OMA DRM v2 agent certificate (chains), the device details or the trusted RI authorities certificate while in storage on the device.

Using the list of threats outlined above, a number of requirements can be derived for a trusted mobile platform (TMP), if it is to facilitate the secure installation of an OMA DRM v2 agent.

1. The TMP SHALL provide a mechanism so that "an OMA DRM v2 agent can perform self-checking of the integrity of its component parts so that unauthorised modifications will be expected to result in a failure of the implementation to provide the authorised authentication and/or decryption function" [29].
2. The TMP SHALL provide a mechanism so that the OMA DRM v2 agent private key can be confidentiality-protected during its installation.
3. The TMP SHALL provide a mechanism so that the OMA DRM v2 agent private key can be confidentiality-protected while in storage on the device.
4. The TMP SHALL provide a mechanism so that the OMA DRM v2 agent private key, the device details and the trusted RI authorities certificate can be integrity-protected during their installation.
5. The TMP SHALL provide a mechanism so that the OMA DRM v2 agent private key, the device details and the trusted RI authorities certificate can be integrity-protected while in storage on the device.

If the OMA DRM v2 agent certificate or an OMA DRM v2 agent certificate (chain) is modified in an unauthorised way, it will be detected when the certificate (chain) is verified. Therefore, they do not need to be integrity-protected during their installation or while they are in storage on the device. However, as the trusted authorities certificate is defined in the CMLA trust model as a self-signed CMLA root CA certificate, it needs to be integrity-protected.

11.3 The rights object acquisition protocol (ROAP) suite

The ROAP suite is defined as the "the suite of DRM protocols between the RI and the OMA DRM v2 agent on the mobile device" [122]. The ROAP suite is composed of five protocols:

- The 4-pass registration protocol;
- The 2-pass rights acquisition protocol;
- The 1-pass rights acquisition protocol;
- The 2-pass join domain protocol; and
- The 2-pass leave domain protocol.

11.3.1 Notation

In our discussion of ROAP we use a large amount of standardised terminology. This terminology is tabulated below.

Version	represents the highest ROAP version supported by the communicating entity.
Device ID	identifies the device to the RI.
Supported Algorithms	identifies the cryptographic algorithms that are supported by the device.
Status	indicates whether a message was successfully handled by the receiving entity.
Session ID	denotes the protocol session identifier selected by the RI.
Selected Version	indicates the ROAP version selected by the RI.
RI ID	identifies the RI to the device.
Selected Algorithms	identifies the cryptographic algorithms to be used in subsequent ROAP interactions.
RI Nonce	denotes a random nonce chosen by the RI.
Trusted Device Authorities	identifies the trusted root certification authorities recognised by the RI.
Server Info	contains server specific information provided by the RI, that must not be modified.
Device Nonce	denotes a random nonce chosen by the device.
DRM Time	denotes a secure non-changeable time source.
Request Time	indicates the current DRM time, as measured by the device.
Certificate Chain	contains a certificate chain, including the public key certificate of the communicating entity.
Trusted RI Authorities	identifies the trusted root certification authorities recognised by the device.
Signature	contains a digital signature on the data sent in the protocol so far.
RI URL	indicates the URL that should be stored in the RI context, and used by the device in later interactions with the RI when sending ROAP requests.
OCSP Response	contains a complete set of valid online certificate status protocol (OCSP) responses for the RI's certificate chain.
Domain Identifier	identifies a domain.
RO Info	identifies the requested rights objects.
Protected ROs	contain the rights objects.
Domain Info	carries domain keys, encrypted using the device's public key.

In our discussion of ROAP the following extensions may be supported by a device or a rights issuer. The optional extensions are tabulated below.

Certificate Caching	In the device hello message of the registration protocol, a certificate caching extension is used by a device to communicate to an RI that it has the ability to store information in the RI context indicating whether an RI has stored device certificate information. In the RI hello message of the registration protocol, a certificate caching extension is used to indicate to the device that the RI has the capability to store information about the device certificate.
Peer Key Identifier	In the RI hello message of the registration protocol, a peer key identifier extension is used to communicate an identifier for a device public key stored by the RI. In the registration request, RO request and join domain request messages, a peer key identifier extension denotes an identifier for an RI public key stored in the device.
Device Details	In the RI hello message of the registration protocol, a device details extension indicates that the RI is requesting that the device details be sent in a subsequent message. In the registration request message of the registration protocol, a device details extension specifies the device model, manufacturer and version.
No OCSP Response	In registration request, RO request and join domain request messages, a no OCSP response extension indicates to the RI that there is no need to send any OCSP responses to the device.
OCSP Responder Key Identifier	In registration request, RO request and join domain request messages, an OCSP responder key identifier extension identifies a trusted OCSP responder key stored in the device.
Domain Name Whitelist	In the registration response message of the registration protocol, a domain name whitelist extension allows an RI to specify a list of fully qualified domain names regarded as trusted for the purposes of silent and preview headers.
Hash Chain Support	In the join domain request message of the domain management protocols, a hash chain support extension indicates that the client supports a particular technique for generating domain keys through hash chains.

	In the join domain response message of the domain management protocols, a hash chain support extension indicates that the RI is using a particular technique for generating domain keys through hash chains.
Not a domain member	In the leave domain request message of the domain management protocols, a not a domain member extension is used by the device to indicate to the RI that the device does not consider itself a member of a particular domain.
Transaction Identifier	In the RO request message of the RO acquisition protocols, a transaction identifier extension allows the device to provide the RI with information for tracking transactions. In the RO response message of the RO acquisition protocols, a transaction identifier extension allows the RI to provide the device with information for tracking transactions.

11.3.2 The 4-pass registration protocol

The 4-pass registration protocol is defined by the OMA as a "complete security information exchange and handshake between the RI and a DRM agent in a device" [122]. The protocol enables the negotiation of protocol parameters including protocol version, cryptographic algorithms, certificate preferences, optional exchange of certificates, mutual authentication of the mobile device and RI, integrity protection of protocol messages, and optional device DRM time synchronisation [122]. The registration protocol is a 4-pass protocol, in which two messages are sent from the device to the RI, namely the device hello and the registration request, and two messages are sent from the RI to the device, namely the RI hello and the registration response. The composition of these messages is shown in table 11.2.

There are three occasions on which the 4-pass registration protocol may be used [122].

- On first contact between the RI and the mobile device.

- When security information needs to be updated.
- When the device time source is deemed to be inaccurate by the RI.

Table 11.2: The 4-pass registration protocol

Step	Message composition
Device hello	Version, Device ID (both of which are mandatory), Supported Algorithms and Extensions—Certificate Caching (both of which are optional).
RI hello	Status, Session ID, Selected Version, RI ID, RI Nonce (all of which are mandatory), Selected Algorithms, Trusted Device Authorities, Server information and Extensions—Peer Key Identifier, Certificate Caching and Device Details (all of which are optional).
Registration request	Session ID, Device Nonce, Request Time (all of which are mandatory), Certificate Chain, Trusted RI Authorities, Server Information and Extensions—Peer Key Identifier, No OCSP Response, OCSP Response Key Identifier and Device Details (all of which are optional) and the Signature of the DRM agent on all data sent so far in the protocol run (which is also mandatory).
Registration response	Status, Session ID and RI URL (all of which are mandatory), Certificate Chain, OCSP Response and Extensions—Domain Whitelist (all of which are optional), and a Signature on all data sent so far in the protocol run (which is also mandatory).

Once the 4-pass registration protocol has been successfully completed, the device establishes a context for the RI. The elements in the RI context are accumulated on the device over the course of the four protocol passes. On completion of a 4-pass registration protocol, the RI context will contain five mandatory elements — the RI URL, the RI ID, the agreed protocol parameters, the protocol version, and information as to whether an RI has stored the OMA DRM v2 agent certificate. It may also contain the following optional elements — trusted device authorities, the OCSP responder public key certificate (chain), the current (valid) OCSP response, the RI certificate (chain), the RI certificate validation data, the domain name whitelist, and the context expiry time [29]. It is required by the CMLA that the device must maintain the confidentiality and integrity of component information of the RI context until it expires [29].

In order to compose the device hello message, the OMA DRM v2 agent must access the implicitly integrity-protected OMA DRM v2 agent certificate in order to obtain the device ID, which is equal to the hash of the OMA DRM v2 agent's public key info, as it appears in the OMA DRM v2 agent certificate. All other elements of the device hello message contain non-sensitive data.

In order to compose a registration request message, a device nonce must be generated. On receipt of the registration request message, and before the registration response message is sent, the RI may optionally perform a nonce-based OCSP request for its own certificate, using the device nonce sent in the registration request message [122]. An OCSP request may also be performed if the RI deems the device DRM time source to be inaccurate, or if the device is an unconnected device which does not support DRM time [122]. The device nonce cryptographically binds an OCSP request and an OCSP response to prevent replay attacks [106].

The device nonce, sent to the RI in the registration request message, and returned in the signed RI registration response message, also allows the device to authenticate the RI. The registration response message is susceptible to a replay attack if the device nonce has been previously used. The registration request message, which contains the device generated nonce is not, however, open to a preplay attack because it is digitally signed. The OCSP response message received by RI from the OCSP responder may also be susceptible to a replay attack unless the device nonce has not been previously used. Whilst realistic attack scenarios for preplay attacks seem a little difficult to construct, there are possible issues with the fact that the OCSP request sent by an RI to an OCSP responder is unsigned and contains a DRM agent generated nonce. Hence, unpredictability of the device nonce is also desirable to rule out any possibility of an attack.

Access may be required to the OMA DRM v2 agent certificate (chain), the trusted RI authorities certificate, and the device details, in order to construct the registration request message. The CMLA requires that both the trusted RI authorities certificate and the device details are integrity-protected if the implementation of the OMA DRM v2 agent is to be considered robust [29]. Access to an accurate DRM time source is also required. In order to complete the remaining registration response extension fields, i.e. Peer Key Identifier, No OCSP Response and OCSP Responder Key Identifier, the RI context must be accessed and the extension fields completed based on the RI certificate, OCSP responder certificate and current valid OCSP response values (or lack thereof) stored in the RI context. Finally, access to and use of the confidentiality-protected OMA DRM v2 private key is required so that the registration request can be signed.

When the registration response has been received, access to the integrity-protected trusted RI authorities certificate is required so that the RI certificate (chain) can be validated and the digital signature of the RI on the registration response message verified. Alternatively, the RI public key field of the RI context may be accessed, if present on the device, so that the digital signature of the RI can be verified. The presence of a valid OCSP response must also be checked by the DRM agent before the RI signature is validated. This OCSP response may be received by the DRM agent in the registration response or, alternatively, accessed from the OCSP response field in the RI context, if present on the device.

Unless the implementation of the OMA DRM v2 is robust, the following additional threats may impact upon the device:

- Replay of the registration response message because of the generation and use of a previously used nonce by the device.

- Replay of an OCSP response message as part of the OCSP protocol, performed by the RI on receipt of the registration request, because of the generation and use of a previously used nonce by the device.
- Preplay attack against the OCSP protocol between the RI and the OCSP responder because of the generation of a predictable nonce by the device.
- Unauthorised modification of the RI context before its expiry time while in storage on the device.
- Unauthorised access to the RI context, the OMA DRM v2 private key, the OMA DRM v2 agent certificate (chain), the device details or the trusted RI authorities certificate.
- Unauthorised reading/copying of the OMA DRM v2 private key while in use on the device.
- Unauthorised modification of the OMA DRM v2 private key, the RI context, the OMA DRM v2 agent certificate (chain), the device details or the trusted RI authorities certificate while in use on the device.

While the inclusion of an inaccurate device DRM time in the registration request message will not result in the realisation of a security threat, it may result in a threat to the efficiency of the protocol run. If the device DRM time included in the registration request is deemed to be inaccurate by the RI, an OCSP exchange is completed by the RI, and the OCSP response received by the RI, containing the correct time, is then communicated to the device in the registration request message.

Using the list of threats outlined above, the following additional requirements can be derived for a TMP, if it is to facilitate a robust implementation of an OMA DRM v2 agent.

6. The TMP SHALL provide a pseudo-random number generator of good quality.

7. The TMP SHALL provide an accurate and trusted time source.

8. The TMP SHALL provide a mechanism such that the RI context can be integrity-protected while in storage on the device.

9. The TMP SHALL provide an access control mechanism such that the RI context, the OMA DRM v2 private key, the device details and the trusted RI authorities certificate, can only be accessed by authorised entities.

10. The TMP SHALL provide a mechanism so that the OMA DRM v2 private key can be confidentiality-protected while in use on the device.

11. The TMP SHALL provide a mechanism so that the RI context, the OMA DRM v2 private key, the device details and the trusted RI authorities certificate can be integrity-protected while in use on the device.

If the OMA DRM v2 agent certificate or an OMA DRM v2 agent certificate (chain) is modified in an unauthorised way, it will be detected when the certificate (chain) is verified, so mechanisms to protect either the OMA DRM v2 agent certificate or an OMA DRM v2 agent certificate (chain) from unauthorised access while in use on the device are not required.

11.3.3 The rights acquisition protocols

Two rights acquisition protocols are defined in the OMA DRM v2 specification set. The 2-pass rights acquisition protocol allows a device to acquire a rights object from a RI. One message is sent from the device to the RI, i.e. the RO request, and one message returned from the RI to the device, i.e. the RO response. The composition of these messages is shown in table 11.3.

This protocol supports [122]:

- mutual authentication of the device and the RI;
- integrity protection for the RO request and RO delivery; and
- the secure transfer of keys necessary to process the RO.

Table 11.3: The 2-pass rights acquisition protocol

Step	Message composition
RO request	Device ID, RI ID, Device Nonce, Request Time and RO Information (all of which are mandatory), Domain ID, Certificate Chain and Extensions—Peer Key Identifier, No OCSP Response, OCSP Response Key Identifier and Transaction Identifier (all of which are optional), and the Signature of the DRM agent on the entire message (which is also mandatory).
RO response	Status, device ID, RI ID, Device Nonce and Protected ROs (all of which are mandatory), Certificate Chain, OCSP Response and Extensions—Transaction Identifier (all of which are optional), and the Signature of the RI on all the data sent during the protocol run (which is also mandatory).

The 1-pass rights object acquisition protocol is initiated by the RI and contains only one protocol message, as shown in table 11.4. It may, for example, be used to support a content subscription [122].

Table 11.4: The 1-pass rights acquisition protocol

Step	Message composition
RO response	Status, Device ID, RI ID, Protected ROs, OCSP Response (all of which are mandatory), RI URL, Certificate Chain, Extensions—Transaction Identifier (both of which are optional) and the Signature of the RI on all the data sent during the protocol run (which is also mandatory).

We now examine how messages are composed and verified in both the 1-pass and 2-pass rights acquisition protocols. In order to compose the RO request message, access to the OMA DRM v2 agent's certificate is required in order to obtain the device ID, which is computed as the hash of the OMA DRM v2 agent's public key info, as it appears in the OMA DRM v2 agent's certificate.

The device nonce, sent to the RI in the RO request message, and returned in the RO response message signed by the RI in the 2-pass protocol, allows the device to authenticate the RI. If the nonce has been previously used, an attacker may be able to launch a replay attack against the 2-pass RO acquisition protocol.

The OMA DRM v2 agent requires access to the RI ID from the integrity-protected RI context, the confidentiality and integrity-protected OMA DRM v2 private key and an accurate DRM time source. Access may also be required to a domain ID from a domain context, which must remain integrity-protected, and the OMA DRM v2 agent certificate (chain). In order to complete the remaining RO request extension fields, i.e. Peer Key Identifier, No OCSP Response and OCSP Responder Key Identifier, the RI context must be accessed and the extension fields completed based on the RI certificate, OCSP responder certificate and current valid OCSP response values (or lack thereof) stored in the RI context. A transaction identifier, which must be integrity-protected [29], may also be generated on the device and communicated to the RI in the RO request message.

When the RO response has been received, authorised access may be required to the trusted RI authorities certificate, so that the RI certificate chain can be validated and the digital signature of the RI verified. Alternatively, access may be required to the RI context, if the RI public key has been previously stored on the device, so that the digital signature of the RI can be verified. The presence of a valid OCSP response must also be checked by the DRM agent. This OCSP response may be received by the DRM agent in the RO response, or, alternatively, accessed from the relevant RI context field already present on the device. A transaction identifier, if generated by the RI and communicated to the device in the RO response, rather than being generated on the device

and communicated to the RI in the RO request, will also need to be integrity-protected by the device.

On receipt of the RO response, the content encryption key, the rights object encryption key, the MAC key, and the random value (Z) used in the generation of a key encryption key (KEK), all of which are contained in a protected RO which has been received, must be confidentiality and integrity-protected [29]. Any permissions or constraints contained in received rights objects must also be integrity-protected [29].

Unless the implementation of the OMA DRM v2 agent is robust, a number of additional threats may impact upon the device on execution of the rights acquisition protocols.

- Replay of the RO response in the 2-pass RO acquisition protocol because of the generation and use of a previously used nonce by the device.
- Unauthorised reading/copying of the CEK, the value Z used to generate the KEK, the KEK, the REK or the MAC key received in a protected RO while in storage on the device.
- Unauthorised modification of the transaction identity, the permissions or constraints, the CEK, Z, the KEK, the REK or the MAC key received in a protected RO while in storage on the device.
- Unauthorised access to the domain ID from a domain context, the transaction identity, any permissions or constraints, CEK, Z, KEK, REK or MAC key.
- Unauthorised reading/copying of any CEK, Z, KEK, REK or MAC key while in use on the device.

- Unauthorised modification of the domain ID from a domain context, the transaction identity or any permissions or constraints, CEK, Z, KEK, REK or MAC key while in use on the device.

While the inclusion of an inaccurate device DRM time in the RO request message will not result in the realisation of a security threat, it will result in a threat to the efficiency of the 2-pass rights acquisition protocol completion. If the device DRM time included in the RO request is deemed inaccurate by the RI, a status value of DeviceTimeError will be returned to the device in the RO response. Following this, the device is required to re-initiate a 4-pass registration protocol.

With respect to rights object acquisition protocols, we can extract the following additional requirements.

12. The TMP SHALL provide a mechanism so that any CEK, Z, KEK, REK and MAC key, received in a protected RO, can be confidentiality-protected while in storage on the device.

13. The TMP SHALL provide a mechanism so that the transaction identity and any permissions and constraints, CEK, Z, KEK, REK and MAC key, received in a protected RO, can be integrity-protected while in storage on the device.

14. The TMP SHALL provide an access control mechanism so that the domain ID, the transaction identity, and any permissions and constraints, CEK, Z, KEK, REK and MAC key, received in a protected RO, can only be accessed by authorised entities.

15. The TMP SHALL provide a mechanism so that any CEK, Z, KEK, REK and MAC key, received in a protected RO, can be confidentiality-protected

while in use on the device.

16. The TMP SHALL provide a mechanism so that the domain ID, the transaction identity, any permissions and constraints, CEK, Z, KEK, REK and MAC key, received in a protected RO, can be integrity-protected while in use on the device.

11.3.4 The 2-pass join domain protocol

Rather than requesting rights objects for individual devices, as illustrated above, a domain may be established, devices added, and domain ROs requested. These ROs can then be shared among the devices in the domain, and used to access protected content.

A domain is defined as a collection of devices that typically belongs to a single user. Once a domain has been established by a user, and after devices have been added to the established domain, protected content and associated rights objects, which have been explicitly created for domain use, may be copied and moved between domain devices. Therefore, rather than requesting a separate rights object for each individual device, only one domain RO need be requested. The join domain and leave domain protocols are used to manage domains.

The join domain protocol may be attempted after the 4-pass registration protocol has been successfully completed. It is used in the establishment of a domain context in the device. On completion of a 2-pass join domain protocol, the domain context will contain three mandatory elements — the domain ID, the domain context expiry time and, if applicable, an indication that the RI supports hash-chained domain keys [122]. The domain key and the RI public key may also be stored in the domain context. This domain context is used by the device to install and use domain ROs [122]. The device must maintain the

confidentiality and integrity of the component information for the domain [29].

In this protocol, the join domain request message is sent from the device to the RI, and the join domain response message is returned from the RO to the device, as shown in table 11.5.

Table 11.5: The 2-pass join domain protocol

Step	Message composition
JoinDomainRequest	Device ID, RI ID, Device Nonce, Request Time, Domain Identifier (all of which are mandatory), Certificate Chain, Extensions—Peer Key Identifier, No OCSP Response, OCSP Response Key Identifier and Hash Chain Support (both of which are optional) and the Signature of the DRM agent on the message (which is also mandatory).
JoinDomainResponse	Status, Device ID, RI ID, Device Nonce, Domain Information (all of which are mandatory), Certificate Chain, OCSP Response, Extensions—Hash Chain Support (both of which are optional), and the Signature of the RI on the message (which is also mandatory).

In order to compose a join domain request message, a device nonce must be generated and sent to the RI. The device nonce enables the device to authenticate the RI. If the nonce has been previously used, a replay attack could be mounted on the RI authentication exchange.

Authorised access is required by the OMA DRM v2 agent to the device ID, the RI ID and the domain ID, which must be integrity-protected [29], and, optionally, the OMA DRM v2 agent certificate (chain). Access to an accurate DRM time source is also required. In order to complete the remaining join domain request extensions, i.e. Peer Key Identifier and No OCSP Response, the RI context must be accessed. Finally, access to, and use of, the confidentiality and integrity-protected OMA DRM v2 private key is also required, so that the join domain request message can be digitally signed by the OMA DRM v2 agent.

When the join domain response has been received, authorised access is required to the trusted RI authorities certificate so that the RI certificate chain

can be validated and the digital signature of the RI verified. As previously stated, the CMLA requires that the trusted RI authorities certificate is integrity-protected [29]. Alternatively, the RI public key field of the RI context may be accessed, if present on the device, so that the digital signature of the RI can be verified. Access may also be required to the OCSP details stored in the RI context. Domain keys securely transmitted in the domain information field of the join domain response must be confidentiality and integrity-protected by the device [29].

Unless the implementation of the OMA DRM v2 agent is robust, a number of additional threats may impact upon a device when using the join domain protocol.

- Replay of the join domain response message because of the generation and use of a nonce which has been previously used by the device.
- Unauthorised reading/copying of the domain key while in storage on the device.
- Unauthorised modification of the domain key, the domain ID, the expiry time or the RI public key, i.e. the domain context, while in storage on the device.
- Unauthorised access to the domain context established as part of the join domain protocol.
- Unauthorised reading/copying of the domain key while in use on the device.
- Unauthorised modification of the elements of the domain context while in use on the device.

While the inclusion of an inaccurate device DRM time in the join domain request message will not result in the realisation of a security threat, it will result in a threat to the efficiency of the join domain protocol run. If the device DRM time included in the join domain request message is deemed inaccurate by the RI, a status value of DeviceTimeError will be returned to the device in the join domain response message. Following this, the device is required to initiate the 4-pass registration protocol.

11.3.5 The 2-pass leave domain protocol

This 2-pass protocol may be executed at any time after the join domain protocol has been completed, but only after the domain context has been deleted from the device. This protocol is used to remove a device from a domain. The two messages passed between the device and the RI during the leave domain protocol are described in table 11.6.

Table 11.6: The 2-pass leave domain protocol

Step	Message composition
LeaveDomainRequest	Device ID, RI ID, Device Nonce, Request Time, Domain Identifier (all of which are mandatory), Certificate Chain, Extensions—Not a Domain Member (both of which are mandatory), and the Signature of the DRM agent on the message (which is also mandatory).
LeaveDomainResponse	Status, Device Nonce, Domain Identifier (all of which are mandatory), and Extensions—None currently defined (which is optional).

In order to compose a leave domain request message, a device nonce must be generated and sent to the RI. Authorised access is required to the device ID, the RI ID, the domain ID, all of which must be integrity-protected, and optionally, the OMA DRM v2 agent certificate (chain). Access to an accurate DRM time source is also required. Finally, access to and use of the confidentiality and integrity-protected OMA DRM v2 private key is also required, so that the leave domain request message can be digitally signed.

No additional threats may impact upon the device in relation to this protocol. While the inclusion of an inaccurate device DRM time in a leave domain request message will not result in the realisation of security threat, it will result in a threat to the efficiency of the leave domain protocol run. If the device DRM time included in the leave domain request message is deemed inaccurate by the RI, a status value of DeviceTimeError will be returned to the device in the leave domain response message. Following this, the device is required to initiate the 4-pass registration protocol.

The following additional requirements, derived from the threats to the join domain and leave domain protocol runs, apply to a trusted mobile platform (TMP):

17. The TMP SHALL provide a mechanism so that the domain key from the domain context can be confidentiality-protected while in storage on the device.

18. The TMP SHALL provide a mechanism so that the domain key, the domain ID, the expiry time and the RI public key from the domain context can be integrity-protected while in storage on the device.

19. The TMP SHALL provide an access control mechanism so that the domain context can only be accessed by authorised entities.

20. The TMP SHALL provide a mechanism so that that the domain key from the domain context can be confidentiality-protected while in use on the device.

21. The TMP SHALL provide a mechanism so that the domain key, the domain ID, the expiry time and the RI public key from the domain context can be integrity-protected while in use on the device.

11.4 Conclusions

In this chapter, the lifecycle of an OMA DRM v2 agent has been examined, and each stage analysed in terms of the threats that may impact on devices on which OMA DRM v2 is not robustly implemented. In order to thwart these threats and provide a robust implementation of OMA DRM v2, a trusted mobile platform must meet the functional requirement set accumulated through the course of this chapter.

Chapter 12

A robust implementation of OMA DRM v2

Contents

12.1 Introduction 381
12.2 Requirements analysis 382
12.2.1 Requirement 1 382
12.2.2 Requirement 2 – 5 and 8 – 21 384
12.2.3 Requirement 6 385
12.2.4 Requirement 7 386
12.2.5 Meeting the requirements using a TMP 386
12.3 Model 387
12.4 Assumptions 388
12.5 The trusted mobile platform architecture 390
12.6 Authenticated boot 392
12.7 Secure boot 394
12.7.1 Prior art 394
12.7.2 Secure boot using a version 1.1 compliant TPM . . . 398
12.7.3 Secure boot using a version 1.2 compliant TPM . . . 399
12.8 Platform run-time integrity 403
12.9 Fundamental TSS and TPM command sequences . 405
12.9.1 TPM permanent flags 405
12.9.2 TPM initialisation 406
12.9.3 TPM startup 406
12.9.4 Context management 407
12.9.5 Endorsement key pair generation 410
12.9.6 Accessing the public endorsement key 410
12.9.7 TPM self testing 412
12.9.8 Enabling the TPM 412
12.9.9 The ownership flag 413
12.9.10 Taking ownership of the TPM 413
12.9.11 TPM activation 415

12.10 Secure storage 415
12.10.1 Key hierarchy 415
12.10.2 Installing integrity and confidentiality sensitive OMA DRM v2 data on the device 416
12.10.3 Secure storage of and access control to OMA DRM v2 data 418
12.10.4 Security of the OMA DRM v2 data while in use on the device 426
12.11 Platform attestation 426
12.12 Demonstrating privilege 429
12.13 Random number generation 434
12.14 Trusted time source 435
12.15 Conclusions 436

In this chapter, the requirements extracted in chapter 11 are utilised in order to examine which architectural components and functionality described within the TCG version 1.2 specification set may be used to provide a robust implementation of OMA DRM v2. This examination also allows us to identify any architecture components and functionality not currently defined within the TCG specification set but which are required for the implementation of a robust and secure DRM solution on a trusted mobile platform.

12.1 Introduction

In the previous chapter we examined the security threats that may impact upon devices, and on the content received by devices, if OMA DRM v2 is not robustly implemented. This enabled us to derive a set of requirements for a robust implementation of OMA DRM v2. In this chapter, we consider the mechanisms required in order to meet these requirements. We assume that the TMP is a mobile platform within which a version 1.2 compliant CRTM, TPM and TSS have been implemented. We then examine which of the required mechanisms are provided by such a platform or, more specifically, by the trusted mobile platform subsystem, namely, the CRTM, TPM and TSS, within such a platform. We also explore the additional functionality that is required of a trusted mobile platform subsystem if it is to meet all the identified requirements.

Section 12.2 summarises the requirements extracted in sections 11.2 and 11.3. In section 12.3 the model defined in chapter 10 and illustrated in figure 10.1 is re-examined and modified to support a trusted mobile platform. Section 12.4 lists the assumptions we make about the trusted mobile platform, and section 12.5 describes the generic trusted mobile platform architecture assumed in the remainder of this chapter.

Sections 12.6 and 12.7 examine authenticated and secure boot mechanisms. Section 12.8 examines runtime integrity protection and verification mechanisms. Section 12.9 explores the fundamental command sequences which need to be completed on any version 1.2 compliant TPM before its security mechanisms can be utilised. Section 12.10 shows how secure storage can be provided, while section 12.11 describes the platform attestation mechanism. Section 12.12 describes the process by which an entity can demonstrate knowledge of an authorisation value/secret bound to a key object, data object, or an 'owner authorised

command' so that access to the object or use of an 'owner authorised command' can be permitted by the TPM. Sections 12.13 and 12.14 briefly examine the random number generation capabilities and trusted time-stamping functionality provided by a version 1.2 compliant TPM. Section 12.15 summarises the conclusions of the analysis completed throughout the chapter.

12.2 Requirements analysis

We now re-examine the security mechanisms that must be provided by a trusted mobile platform if it is to provide a robust implementation of an OMA DRM v2. In doing so, we summarise the requirements extracted in sections 11.2 and 11.3.

12.2.1 Requirement 1

Requirement 1 necessitates that the integrity of an OMA DRM v2 agent can be checked, and, if any unauthorised modifications are detected, that the OMA DRM v2 agent implementation will fail to provide authorised authentication and/or use of the decryption function [29]. This requirement may be fulfilled in a variety of ways. We consider three possible approaches.

Firstly, an authenticated boot mechanism in combination with a secure storage mechanism could be used to meet this requirement.

- An authenticated boot mechanism facilitates the accurate measurement and secure storage of the software configuration of the TMP; and
- A secure storage mechanism ensures that security sensitive information, such as the OMA DRM v2 agent's private key, cannot be accessed and/or utilised if the OMA DRM v2 agent code has been modified in an unauthorised way.

Secondly, a secure boot mechanism could be deployed in order to ensure that only a legitimate and authorised OMA DRM v2 agent can be loaded at boot time. Run-time integrity protection and/or verification mechanisms can then be used in conjunction with a secure boot mechanism in order to ensure that the software environment remains in a trustworthy state after boot.

- A secure boot mechanism enables the accurate measurement and verification of the correctness of the software configuration of the platform at start-up. An unauthorised, yet successful, attempt to modify the OMA DRM v2 agent should result in one of the following three scenarios [4] at boot time.
  - The system could continue booting as normal but issue a warning. This approach gives little protection against attack. Malicious or corrupted software components can still be executed.
  - The system could opt not to execute the component whose integrity is compromised. This, however, leaves the system open to denial of service attacks.
  - Finally, the system could attempt to recover and correct the inconsistency using a trusted source before executing or using the component.
- A runtime integrity-checking mechanism facilitates the accurate measurement and verification of the correctness of the software configuration of the platform while it is in operation. An unauthorised yet successful attempt to modify the OMA DRM v2 agent while the platform is in use should result in one of the following two scenarios.
  - The system could continue as normal but issue a warning. This approach, however, gives little protection against attack. Attacks may

still be successfully executed against software components running on the platform.

– The system could make unavailable the majority of its services if the integrity of a software component is compromised. The platform would then have to be rebooted in order to transition back into a trusted state. This, however, leaves the system open to denial of service attacks.

Thirdly, mechanisms which aim to prevent an attack impacting the runtime integrity of the platform could be adopted.

In conjunction with the mechanisms described above, a platform attestation mechanism would allow a TMP to attest to both the hardware and software environment of the platform. In this way, a content provider or indeed a rights issuer could be assured of the configuration of the OMA DRM v2 agent and any platform security components implementing secure boot and/or run-time integrity protection or verification mechanisms.

- A platform attestation mechanism enables the platform's configuration, which has been measured and securely stored on the TMP using the authenticated boot mechanism, to be reported to a challenger of the platform. On receipt of this report, a challenger can validate that the platform's configuration has not been modified in an unauthorised manner before embarking on further interactions.

12.2.2 Requirement 2 – 5 and 8 – 21

Requirements 2 – 5 and 8 – 21 can be summarised as follows.

- A mechanism is required so that data may be installed on the TMP, where

either its:

  - Integrity; or
  - Integrity and confidentiality;

must be protected.

- A mechanism is required so that data stored on the TMP is protected with respect to its:
  - Integrity; or
  - Integrity and confidentiality.
- A mechanism is required so that confidentiality and integrity-protected data can only be accessed by authorised entities, i.e. a particular OMA DRM v2 agent.
- A mechanism is required so that data in use on the TMP is protected with respect to its:
  - Integrity; or
  - Integrity and confidentiality.

In considering whether (and how) TPM functionality can be used to meet the above requirements, it should also be considered whether TPM functionality could be used to generate (and not just protect) the OMA DRM v2 agent key pair, described in section 11.2.

12.2.3 Requirement 6

Requirement 6 necessitates a pseudo-random number generator of good quality to be provided by the TMP.

12.2.4 Requirement 7

Requirement 7 necessitates a mechanism which supports the implementation of a trusted time source.

12.2.5 Meeting the requirements using a TMP

In the remainder of this chapter we examine ways of using trusted computing functionality to meet the requirements outlined in sections 12.2.1 to 12.2.4. Initially, in sections 12.3, 12.4 and 12.5 we list some fundamental assumptions regarding a TMP.

In sections 12.6, 12.7, 12.8, and 12.11, we examine mechanisms which enable us to meet requirement 1 as defined in section 12.2.1, namely, authenticated boot, secure boot, run-time integrity protection and platform attestation.

Section 12.9 explores the fundamental command sequences which need to be completed on any version 1.2 compliant TPM before its security mechanisms, i.e. secure storage, platform attestation, random number generation and time source functionality, can be utilised.

Section 12.10 investigates if and how requirements 2 – 5 and 8 – 21, as described in section 12.2.2, can be supported using TPM version 1.2 secure storage functionality.

Section 12.12 describes the process by which an entity can demonstrate knowledge of an authorisation value/secret bound to a key object, data object, or an 'owner authorised command' so that access to the object or use of an 'owner authorised command' can be permitted by the TPM. Use of the platform attestation mechanism and provision of integrity protected secure storage are dependent on this process.

Sections 12.13 and 12.14 briefly examine if and how requirements 6 and 7, as described in sections 12.2.3 and 12.2.4, can be met using the random number generation capabilities and trusted time-stamping functionality provided by a version 1.2 compliant TPM.

12.3 Model

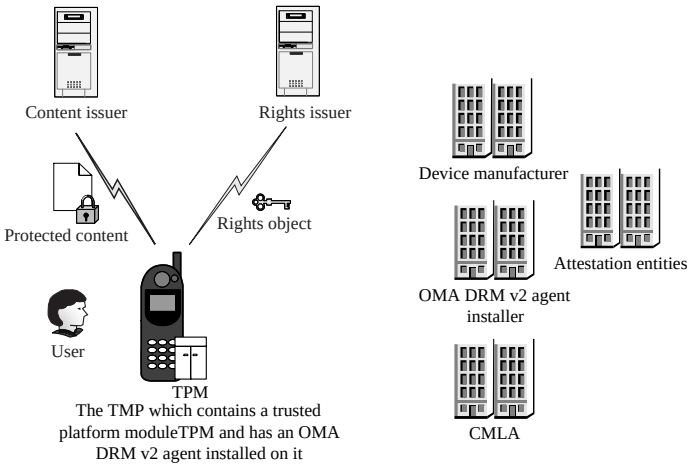

Figure 12.1: Revised architecture model

We now revisit the model described in chapter 10 and illustrated in figure 10.1. The revised model is shown in figure 12.1, and involves eight entities. Six of these, namely the device manufacturer; the OMA DRM v2 agent installer; the CMLA; the user; the content issuer and the rights issuer are already included in the model illustrated in figure 10.1, and described in section 10.4.1. The revised model also includes two further entities, namely a trusted mobile platform which is included in place of the mobile device illustrated in figure 10.1, and a set of attestation entities, responsible for issuing a set of credentials for a particular TMP to testify to their confidence in platform components, see A.10. A CRTM and at least one TPM, see section A.6.1.1, is either physically

or logically bound to the trusted mobile platform. This TPM is supported by a TCG software stack, as described in section A.6.2. All references to a TPM and a TSS in this chapter are to a TPM and TSS conforming to version 1.2 of the TCG specifications.

The TPM must first be manufactured and then integrated into a mobile platform by the device manufacturer, as described in section 10.4. In order for the manufactured device to be considered a trusted mobile platform, the TPM, the integration of the TPM into the platform, and the platform design must be certified by various attestation entities, see section A.5, namely the trusted platform management entity (TPME), conformance entities (CEs) and the platform entity (PE).

Once the device has been manufactured, an OMA DRM v2 agent is installed by the agent installer, who may in practice be the device manufacturer. Privacy-certification authorities (P-CAs) and validation entities (VEs), as described in section A.5, must certify TMP identities and the trustworthy measurements of software components respectively, if statements regarding the software and hardware configuration of the TMP are to be made by the TMP and validated by a TMP challenger.

The OMA DRM v2 agent implementation is required to fulfil robustness rules specified by the CMLA [29] or a similar organisation. Once the OMA DRM v2 agent has been robustly implemented, a user may utilise their TMP or, more specifically, the OMA DRM v2 agent installed on their TMP, to acquire protected content and their associated rights objects.

12.4 Assumptions

We make the following assumptions about the trusted mobile platform.

- At least one version 1.2 compliant TPM is inextricably bound to the TMP. Support for platform integrity measurement, recording and reporting is also provided. See appendix A for further details.

- The TPM and the platform to which it is bound must have a specified set of credentials associated with it. This credential set is described in section A.10.

- It is assumed that the TPM is supported by a version 1.2 compliant TSS, see section A.6.2, such that applications may interface efficiently and easily with the TPM.

- A TMP potentially has a number of stakeholders, for example the device manufacturer, the network operator, third party service providers and the end user.

- In order to serve the interests of each stakeholder associated with the mobile platform, various trusted platform mechanisms need to be available. Each stakeholder may, for example, potentially need to call TPM commands, generate their own storage key hierarchy to which only they have access, and attest to platform state or certify keys using attestation identity keys.

- The trusted mobile platform is running at least one protected execution environment. Within this environment, an application can run in isolation, free from being observed or compromised by other processes running in the same protected partition, or by software running in any insecure partition that may exist in parallel.

12.5 The trusted mobile platform architecture

As stated in section 12.4, every stakeholder requires access to TPM functionality. How this functionality is provided to each stakeholder is implementation specific. With respect to the TCG trusted platform functionality, as described in appendix A, we highlight some issues which must be considered in relation to TPM command usage, storage key hierarchies and attestation identity key use in a mobile platform architecture which incorporates multiple stakeholders.

The majority of TPM commands may be called by any entity with access to the platform, as they do not require any authorisation data to be input before they can be executed. Some TPM commands, namely those in a category called 'TPM-owner authorised commands', can only be executed on demonstration that the TPM owner authorisation data is known by the calling entity, see section A.16.2. If someone other than the TPM owner needs to execute such commands, either the TPM owner authorisation data must be transmitted to that specific entity, or the TCG delegation functionality must be used, see section A.18. The fact that access is required by stakeholders to these 'TPM-owner authorised commands' must be considered as part of any security assessment.

Each individual stakeholder may also require their own storage key hierarchy, so that no other stakeholder on the platform can access the keys in their hierarchy.

Finally, we examine the generation and use of attestation identity keys, which require the input of TPM owner authorisation data. In order that a stakeholder can attest to the platform's configuration, or indeed certify other keys using platform AIKs, he or she must be able to either generate and activate attestation identity keys or, alternatively, utilise AIKs which the TPM owner has generated and activated, when using a subset of TPM commands, for example,

TPM_CertifyKey, *TPM_CertifyKey2*, *TPM_Quote*, *TPM_Quote2*.

In order to satisfy the above requirements for individual stakeholders, we describe an abstract trusted mobile platform architecture in which the required TPM functionality is available to each individual stakeholder. This TPM functionality may be provided to each stakeholder in a variety of ways.

- Each stakeholder's 'TPM functionality' may be provided using 'physical' TPMs, implemented, for example, as hardware TPM chips, where a physical TPM is defined as a module with its own physical resources and meeting a TCG TPM protection profile and target of evaluation.

- Alternatively, only a device manufacturers's TPM is implemented by a physical TPM. Other stakeholder 'TPM functionality' could then be provided through the delegation of owner authorised key and command use by the device manufacturer, as described in section A.18. Unrestricted use of unauthorised TPM commands, and the generation of an isolated branch of keys in the physical TPM key hierarchy, would also be permitted.

- Other possibilities include the implementation of virtual stakeholder TPMs with their foundations in a physical device manufacturer TPM. Such virtual stakeholder TPMs may be constructed using emulations based on the device manufacturer TPM, or as TSS instantiations which have their basis in the device manufacturer TPM.

- As an alternative to a technical solution, legal or commercial agreements could be drawn up between stakeholders, enabling multiple entities to share a single TPM.

Whatever the chosen implementation, the mobile device must be capable of supporting and protecting the interests of every stakeholder, either indepen-

dently or in cooperation with other trusted stakeholders.

Analysis of this use case, 'a robust implementation of OMA DRM v2', is used below to identify the TMP subsystem functionality that may need to be provided to the OMA DRM v2 agent installer by the TMP. In the remainder of this chapter, we investigate whether the mechanisms provided by the TMP (as defined in section 12.4) meet the requirements described in section 12.2. If a particular mechanism is provided by a TMP we also examine the architecture components, i.e. the TPM and TSS commands, required to utilise the mechanism. If a particular mechanism is not provided by a TMP, we describe the additional functional components required within a TMP, as described in section 12.4, in order that the mechanism can be provided.

12.6 Authenticated boot

Requirement 1, as described in 12.2, may be partially met through the deployment of an authenticated boot mechanism. Such a mechanism can be provided by a TMP.

Such a mechanism would be supported primarily by the root of trust for measurement and the root of trust for storage, as described in section A.6.1. The TPM incorporates the RTS. The RTM is generally implemented in a PC platform via the integration of additional instructions into the BIOS or BIOS boot block (the CRTM), which cause the platform processor to act as the RTM.

It is envisaged that the authenticated boot mechanism for a mobile platform will closely resemble that of the PC platform, as described in section A.11.3. In this instance, the CRTM could be integrated, for example, into the BIOS boot block (BBB) of the phone. Measurement functionality could be integrated into various platform components, for example the BIOS, the OS loader, and/or the

OS. Precisely which components are used will depend on the specific platform architecture. The authenticated boot mechanism could, for example, proceed as follows:

- When the BBB starts the boot process, it measures its own configuration and the configuration of the entire BIOS, saving the measurement to a TPM PCR and a summary of the measurement to a log file in the TMP.
- The BIOS then continues the measurement process, saving measurements of option ROMs and the OS loader, for example, to the TPM PCRs and a summary to the log on the TMP. It then passes control to the next component in the chain, the OS loader.
- This process continues until all the specified software on the platform has been measured.

Measurements stored during the authenticated boot process may be utilised in secure storage and attestation mechanisms, described in sections 12.10 and 12.11.

The architectural components required in order to provide such a mechanism are already defined within the TCG specification set, as follows.

- A root of trust for measurement is required to accurately measure at least one integrity measurement, and report the integrity measurement to the TPM.
- A root of trust for storage is required to accept measured integrity measurements and record them. This may be accomplished using the TSS PCR extension methods, *Tspi_TPM_PcrExtend* and *Tcsip_Extend*, and the TPM PCR extension command, *TPM_Extend*.

The corresponding entries in the TSS event log can be written using the *Tcsi_LogPcrEvent* command. The *TSS_PCR_EVENT* data structure is required to provide information about an individual PCR extend event.

The exact process by which a trusted mobile platform is booted, its integrity is measured and its integrity measurements stored, needs to be specified, just as for the PC client in [153]. All RTM implementations are required to meet the TBB protection profile, which defines what properties must be met by the RTM, independently of how it is implemented.

12.7 Secure boot

Requirement 1, as described in section 12.2, may be partially met through the deployment of a secure boot process. The authenticated boot process, as described in the previous section, permits a platform to boot into any state. As described in section 12.2, a trusted mobile platform implementation may require a secure boot mechanism, rather than an authenticated boot mechanism so that the platform is permitted only to boot into a specified state. Such a mechanism has not been specified by the TCG.

In the following subsections we will examine previous work on secure boot (section 12.7.1), suggested methods for secure boot implementation using a version 1.1 compliant TPM (section 12.7.2), and suggestions/requirements for implementing a secure boot process using a version 1.2 compliant TPM (section 12.7.3).

12.7.1 Prior art

We begin by examining previous work on secure boot, that was conducted independently of the TCG. The concept of secure boot has been widely discussed,

most notably by Tygar and Yee [161], Clark and Hoffman [27], Arbaugh, Farber and Smith [4] and Itoi et al. [88]. Each of these papers describe a similar process, in which the integrity of system components is measured, and these measurements are then compared against a set of expected measurements which must be securely stored and accessed by the platform during the boot process.

Tygar and Yee [161] were amongst the first to describe a secure boot mechanism [4]. They discuss the possibility of using a secure co-processor to facilitate a secure boot. The expected integrity measurements of system components are stored within the secure co-processor non-volatile memory, where their integrity and privacy can be assured. The secure co-processor is first to take control of the system, and it checks system components, for example the bootstrap program, the OS kernel and system utilities, before handing over to the host CPU. Tygar and Yee also discuss issues surrounding the use of a secure boot floppy, containing system verification code, rather than using a secure co-processor, which requires significant architectural revisions to most computer systems [4].

Clark and Hoffman [27] present a system in which a personal computer memory card international association (PCMCIA) card is used to facilitate a secure boot. In this case, the host's boot sector and a series of checksums for boot files and host executables are stored on the PCMCIA card. When the card is inserted into the host, the user is initially authenticated to the card by entering a password. The card is also authenticated to the host after knowledge of a secret shared between the card and the host has been demonstrated. If both authentications are successful, the card allows the host to read the boot sector and any required checksums from the card. When the boot sequence completes, control is given to the operating system, whose configuration has either been retrieved from the PCMCIA card or measured and verified against the expected measurement value stored on the PCMCIA card [27]. The physical security of

both the host and the card are assumed.

Arbaugh, Farber and Smith [4] require the addition of a PROM board and the modification of the system BIOS. Their AEGIS model is based upon four fundamental assumptions. It is assumed that an attacker is unable or unwilling to replace the motherboard, CPU and a portion of the system read-only memory (ROM)/BIOS, which contains a small section of trusted software. It is also assumed that an expansion card/programmable read-only memory (PROM) board, which contains cryptographic certificates and copies of essential boot process components for recovery, is present. The integrity of this expansion card, called the AEGIS ROM, must also be maintained. It is implied by Arbaugh, Farber and Smith that the cryptographic certificates contained within the PROM board enable the identities of entities, trusted to certify trustworthy configurations of software components on incoming component certificates, to be verified. These certificates may, for example, take the form of self-signed public key certificates, see section 1.5.7, of entities permitted to certify trustworthy configurations of software components. Alternatively, they may take the form of Keynote policy assertions, see section 4.4, in which the system administrator specifies the set of public keys which they trust to certify trustworthy configurations of software components. A specific method by which entities are authorised to certify trustworthy configurations of software components is not specified. Finally, it is assumed that a trusted source exists to support the recovery of platform components, for example a network host or a trusted ROM card located within the host.

Before a secure boot process can be completed the computing platform must be initialised with a number of items (see [4] and [88]).

1. For every component on the platform which requires a secure boot, an

authorised entity must generate a hash of that software component (when it is working as expected) and then create a credential which contains the component hash, a component identifier and an expiry date. An authorised entity is one trusted by the system to certify trustworthy configurations of software components. Arbaugh, Farber and Smith imply that this trust relationship is established through the use of 'cryptographic certificates' installed in the AEGIS ROM. As stated above, details of the trust establishment mechanism are not defined.

2. The credential is digitally signed by the authorised entity.

3. This signed credential is then stored on the host, for example in the platform component to be securely booted, or in a data block of a flash memory device on the host's motherboard.

4. The AEGIS ROM and BIOS block block contain a small section of trusted software, signed credential(s), authorised entity public key certificates and recovery code, whose integrity is assumed.

The secure boot process proceeds as follows (see [4] and [88]).

1. The first section of the BIOS executes, i.e. the part which contains a small section of trusted software, and computes a checksum over its address space and the address space of the AEGIS ROM. This process protects against ROM failures.

2. A hash of the remainder of the BIOS is then computed.

3. Execution control is then passed to this second section of the BIOS if:

   - Its associated credential has not expired;
   - The signature on the credential is valid;

- The hash value stored in the credential matches the value computed in step 2.

4. This BIOS component then hashes each of the expansion ROMs and verifies them against their expected values.

5. This hashing and verification continues until the system has been booted into an expected state.

If at any stage during the boot process there is an integrity failure, the failed component is replaced using components either stored on an AEGIS expansion card/PROM board, or retrieved from a trusted network host. Itoi et al. [88] extend the AEGIS system to work with smartcards.

12.7.2 Secure boot using a version 1.1 compliant TPM

We now examine suggested methods for secure boot implementation using either a version 1.1 or 1.1b compliant TPM. Versions 1.1 and 1.1b of the TCG TPM specification set define a data integrity register (DIR) as a storage register that holds a 20-byte digest value, see section A.11.2. These versions of the TCG specifications require that the TPM contains only one 20-byte DIR in a TPM-shielded location, although the TPM could incorporate more than one DIR. While the exact purpose of DIRs was not specified, their use in the implementation of a secure boot process is briefly examined in [5], and is now described.

If a TPM contains the same number of DIRs as PCRs, the expected value of every PCR can be written to its corresponding DIR. Every time a PCR is filled and its final value computed, it is compared to its corresponding DIR value. If the two values match, the boot process continues; otherwise an exception is called and the boot process halted.

Alternatively, if the TPM has access to non-volatile memory, all expected PCR values may be held in unprotected non-volatile memory, and a summary, i.e. a cumulative digest, is held in a single DIR. Every time a PCR is filled and its final value computed, it is checked that:

1. Each PCR value, when calculated, matches the expected value held in the non-volatile memory; and

2. The cumulative digest of the expected table of PCR values matches the value held in the DIR.

Read access to DIRs must be provided without the need for any authorisation data to be input as, typically, no authorisation information is available at the early stage in the boot process when the DIR value must be read.

In the version 1.2 specifications, use of the DIR has been deprecated. The TPM must still, however, support DIR functionality in the general-purpose non-volatile storage area.

12.7.3 Secure boot using a version 1.2 compliant TPM

Following our examination of prior art in the area of secure boot, we now outline a set of additional functional components which may be required within a TMP, as described in section 12.4, in order that a secure boot mechanism can be implemented on a TMP.

- Each software component on the platform whose integrity is to be measured and verified at boot time must have a corresponding reference integrity metric (RIM), which is equal to the hash of the component. In order to ensure the secure boot of an OMA DRM v2 agent, it is required that a correct reference integrity measurement for an OMA DRM v2 agent

(i.e. the OMA DRM v2 agent RIM) is present on the platform.

- Each component RIM, a component identifier and expiry data must be digitally signed by an authorised entity to create a credential, as described by Itoi et al. [88].
- A list of authorised entities must be securely stored within the TMP.
- A root of trust for verification (RTV) is required to verify at least one integrity measurement.
- Just as the RTM has its foundation in an immutable instruction set, i.e. the core root of trust for measurement, the RTV must also have its foundation in an immutable and trusted instruction set, known as the core root of trust for verification (CRTV).
- The CRTV shall act in conjunction with the CRTM to measure and verify the first set of software components on the platform. It then passes control to the RTM and RTV integrated into the first set of software components, which continue the measurement and verification process.

As the platform boots, a specified set of platform software components (including the OMA DRM v2 agent) need to be measured by the RTM and verified by the RTV. For every software component:

- The RTM measures the component;
- The signature on the corresponding component credential is verified, and the expiry date within the credential checked;
- If the signature is valid and the credential has not expired, the value measured by the RTM is compared to its corresponding RIM;

- If no discrepancy is found between the measured value and its expected RIM, the measurement is stored securely to the TPM PCRs and the boot process continues;
- If a discrepancy is found between the measured value of the software component and its RIM, appropriate action should be taken (for example, the boot process aborted).

Three issues which also need to be discussed include recoverability of components that fail the integrity check, the revocation of RIM certificates, and the identities of the authorities responsible for signing RIM certificates.

While a secure boot mechanism is not described within the TCG specifications, we have already seen how current TCG-defined components, i.e. DIRs or TPM non-volatile memory and the RTM, may be utilised to implement a secure boot mechanism. We now examine two TCG structures which may be useful in the definition of a secure boot mechanism.

The form and structure of 'validation certificates', as described in version 1.1 of the TPM main specification, could be used to represent RIM certificates. However, the validation certificate structure is not included in the v1.2 TPM specifications set. Currently, the specification of all TCG certificates and credentials are being re-defined by the TCG infrastructure working group. Current versions of the infrastructure profile specification document, however, indicate that the validation certificate may not be included in future releases. Whether or not validation certificates need to be specified should be re-considered in light of the trusted mobile platform requirements. The *VALIDATION_DATA* structure, as given in the TCPA main specification version 1.1b [148], is shown in table 12.1. The purpose of the validation data structure is to encapsulate the integrity metric of a platform component that is behaving as expected.

Table 12.1: The *VALIDATION_DATA* structure

Name	Description
"*TCPA Validation Data*"	The ASCII string "TCPA Validation Data".
component_manufacturer	The name of the manufacturer of the component (in ASCII).
component_name	The common name of the component (in ASCII).
component_version	The version of the component (in ASCII).
instruction_digest	The digest of any component instructions that are intended to execute on the main platform.
component_distributed_validation	A reference to the security properties of the component
VE_reference	An indication of the identity of the (validation) entity that attests to the validation data.
TCPA_VERSION	The TPM version
validation_data_signature_value	The result of signing all the fields on the *VALIDATION_DATA* structure using the signature key of the *VE_reference*.

Once platform verification has been completed by the RTV, the *TSS_EVENT_CERT* data structure, which is described in the version 1.2 TSS specification (see [154]), could be utilised to indicate the result of a comparison/verification completed by the RTV. The structure of a *TSS_EVENT_CERT* is outlined in table 12.2 below.

Table 12.2: The *TSS_EVENT_CERT* structure

Name	Description
versionInfo	Version data.
ulCertificateHashLength	The length in bytes of the certificate hash parameter.
rgbCertificateHash	Pointer in memory containing the hash value of the entire validation entity certificate.
ulEntityDigestLength	The length in bytes of the entity digest parameter.
rgbEntityDigest	Pointer in memory containing the actual digest value of the entity.
fDigestChecked	TRUE if the entity logging the event checked the measured value against the digest value in the certificate; FALSE if no checking was attempted. FALSE if no checking was attempted.
fDigestVerified	Only valid when the value of the filled above is TRUE. The value is TRUE if the measured value matches the digest in the certificate, and FALSE otherwise.
ulIssuerLength	The length in bytes of the issuer parameter.
rgbIssuer	Pointer to the actual issuer certificate.

12.8 Platform run-time integrity

Requirement 1, as described in section 12.2, may be partially met through the deployment of a run-time integrity checking mechanism. A secure boot mechanism, as described in the previous section, offers assurances regarding the state into which the platform has booted. Assurances are also required regarding the run time integrity of the platform, so that any changes to the platform which affect the trusted state into which it has booted may be prevented, or at least detected and responded to. Neither preventative nor reactive measures are currently provided by a TMP, as described in section 12.4. This section outlines a set of additional functional components that may be required within a TMP in order that a run-time integrity checking mechanism can be implemented.

In order to develop a reactive mechanism, the components described in section 12.7 could be re-deployed and their capabilities extended. In this case, the RTM would be required not only to measure the platform software components at boot time, but to re-measure software components running on the platform periodically or, indeed, when triggered by a particular event. Rather than compare the measured values to static reference integrity measurements at boot time, as described above, the RTV would also compare measurements taken during runtime to run-time RIMs whose values could change over different instances of the boot sequence. How the set of run-time RIMs are generated needs to be specified. In conjunction with this, the reaction of the RTV to an integrity verification failure should also be discussed. Finally, the management of the policy statement which describes the components to be checked during run-time, and the frequency of checking, must also be considered.

Rather than deploy a mechanism which detects and reacts to unauthorised modification of platform components during runtime, preventative measures

could be used. Depending on the system architecture, varying degrees of separation and isolation of software components can be provided. A relatively simple approach involves storing critical and unchanging data in one time programmable memory or ROM.

Alternatively, the import of native code to the platform could be prohibited, and the download of interpreted code permitted but only to managed compartments within the platform. Two types of Java application management systems exist for a mobile device, corresponding to specifications in Java 2 Platform Micro Edition (J2ME) [39]. MIDP and PDA Profile (PDAP) are specifications designed to enable the use of Java on embedded resource constrained devices[1], i.e. CLDC devices such as cell phones and PDAs. The Open Service Gateway initiative (OSGi) specification[2] defines a life-cycle management model for a Java program. Its reference implementation runs on Foundation Profile, Personal Profile or Personal Basis Profile which are specifications designed to enable Java on Connected Device Configuration (CDC) devices [39]. Both MIDP/CLDC and OSGi define their own unique security model and policy [39].

A lower level mechanism which facilitates the isolation of compartments, and one which permits the download of native code to a platform, can be provided through the deployment of a secure operating system on the platform. Both SELinux and Trusted BSD are examples of operating systems which have controlled access protection profile evaluation assurance level-4 (CAPP EAL-4) Common Criteria certification and access control mechanisms which are finer grained than mass market operating systems [39].

The most secure isolation mechanism, and one which has been widely discussed in the context of a PC, involves the deployment of an isolation layer, see

[1] www.java.sun.com/products/midp/
[2] www.osgi.org

section 1.6.8, which "provides a means to isolate operating systems, application and applets" [104]. Proposed implementations include virtual machine monitors, hypervisors, microkernels and exokernels, see section 1.6.8. More recent work has seen the development of an 'isolation kernel' by Microsoft, see section A.23. This work relies on the development of curtained memory facilities by Intel, see section A.24, so that an isolation kernel can be isolated in a hardware protected environment, and in turn can provide isolated environments to higher level software components. The OpenTC project[3] is currently examining how an L4 microkernel can be ported onto an embedded system so that isolated compartments can be supported.

12.9 Fundamental TSS and TPM command sequences

Before we examine the TPM and TSS version 1.2 commands, which can be used to fulfil storage, attestation, random number generation and time-stamping requirements, as described in section 12.2, we review a number of TPM and TSS commands which need to be executed in order to initialise a TPM for use. Further information on all commands listed below may be found in appendix A.

12.9.1 TPM permanent flags

Firstly, in table 12.3, we define a number of TPM permanent flags the use of which is discussed in this chapter. TPM permanent flags are used to maintain the state information for the TPM [157]. The values of these flags are not affected by the *TPM_Startup* command.

[3] www.opentc.net

Table 12.3: TPM permanent flags

Name	Description
TPM_PF_READPUBEK	This flag may be set to *TRUE* or *FALSE*. It indicates whether the public endorsement key can be read with or without owner authorisation. The default value is *TRUE*.
TPM_PF_DISABLE	This flag may be set to *TRUE* or *FALSE* and indicates whether TPM is disabled or enabled. The default value is *TRUE*.
TPM_PF_OWNERSHIP	This flag may be set to *TRUE* or *FALSE* and indicates whether or not an entity can be take ownership of the TPM. The default value is *TRUE*.
TPM_PF_DEACTIVATED	This flag may be set to *TRUE* or *FALSE* and indicates whether the TPM is deactivated or activated. The default value is *TRUE*.

12.9.2 TPM initialisation

The TPM must first be initialised. *TPM_Init* is a method of physically initialising the TPM. This command puts the TPM into a state where it waits for *TPM_Startup*, a command which specifies the type of the initialisation required. The TPM initialisation command is shown in table 12.4.

Table 12.4: TPM initialisation

TPM_Init

12.9.3 TPM startup

After TPM initialisation, the TPM must then be started up. The *TPM_Startup* command is always preceeded by *TPM_Init*. The TPM can start up in one of three possible modes. The chosen mode depends on the platform event that caused the reset, and the operations on the TPM that need to be completed in response to the particular event. The three modes include: *clear* start, *save* start and *deactivated* start. For an initial start up, a *clear* start would normally be used, where all variables go to their default or non-volatile values. The TPM startup command is shown in table 12.5.

Table 12.5: TPM start-up

TPM_Startup

12.9.4 Context management

Every time an application is to participate in communication with a TPM via the TCS, it must connect to a context, to ensure that the TSS service provider (TSP) layer or, indeed, the application is talking to the correct TSS core services (TCS) layer.

The focus of the context object is [154]:

- to provide a connection to a TSS core service. There might be multiple connections to one or more core services.
- to provide functions for resource management and freeing of memory.
- to create working objects.
- to establish a default policy for working objects, as well as a policy object for the TPM object representing the TPM owner.
- to provide functionality to access the persistent storage database.

Initially, a context must be created, using the series of commands shown in table 12.6.

Table 12.6: Creating a context

Tspi_Context_Create	Returns a context handle to a new context object.
Tspi_SetAttribUint32	This method sets the 32-bit attributes of the context object.
Tspi_SetAttribData	This method sets a non 32-bit attribute of the context object. A non 32-bit attribute is an attribute which may vary in structure and size. Currently, no such attributes have been defined for the context object.

A handle to the TPM object associated with the context must then be retrieved and its attributes set. As above, the *Tspi_SetAttribUint32* and the

Tspi_SetAttribData commands may be used to set the attributes of the TPM object, or all the necessary parameters may already be defined for the TPM object by default. This handle represents the TPM with which the application is communicating with via the TCS layer. The sequence of commands required in order to achieve this are shown in table 12.7.

Table 12.7: Creating a TPM object

Tspi_Context_GetTPMObject	Retrieves the handle of the TPM object associated with a context. Only one instance of this object exists for a given context, and it implicitly represents the TPM owner.
Tspi_SetAttribUint32	This method sets the 32-bit attributes of the TPM object.
Tspi_SetAttribData	This method sets a non 32-bit attribute of the TPM object.

A connection must then be made to the chosen context, using the pair of commands shown in table 12.8.

Table 12.8: Connecting to a context

Tspi_Context_Connect	Establishes a connection to either a local or remote TSS system.
Tcsi_OpenContext	Returns a handle to an established context.

When the communication has been completed, the context is closed, as shown in table 12.9, and memory associated with the context is freed, using the commands given in table 12.10.

Table 12.9: Closing a context

Tspi_Context_Close	Destroys a context and releases all assigned resources.
Tcsi_CloseContext	Releases all resources assigned to the given context.

The FreeMemory calls may or may not be be necessary. The TCS developed

Table 12.10: Freeing memory allocated to the context

Tspi_Context_FreeMemory	Frees memory allocated by TSP to the specified context.
Tcsi_FreeMemory	Frees memory allocated by TCS to the specified context.

by NTRU[4], for example, cleans up everything related to a context when the context is closed, whether or not the FreeMemory methods are explicitly called.

On creation of a context, a default policy object is created. Each newly created object associated with that particular context is automatically assigned to its corresponding default policy. The default policy for each working object remains unless the *Policy_AssignToObject* method is used to associate a new policy object with the working object.

A handle to the default policy object can be retrieved using the method shown in table 12.11. The default policy object has the following settings after initialisation:

- *Secret mode* = *TSS_SECRET_MODE_POPUP*, which means that the TSP will display a dialogue to the user so that a pass phrase can be entered. This pass phrase is then hashed using SHA-1 to obtain the authorisation secret for the working object.
- *Secret lifetime mode* = *SECRET_LIFTIME_ALWAYS*, which implies that once the pass phrase has been entered and hashed, it is cached by the TSP and does not have to be re-entered by the user.

The attributes of the default policy object may, however, be changed using the *Tspi_SetAttribUint32* and *Tspi_SetAttribData* methods.

Table 12.11: The default policy object (created on TPM initialisation)

Tspi_Context_GetDefaultPolicy

Before any call to the TPM is made, a connection must be established with the TPM device driver, after which the *Tddli_TransmitData* function must be used to send the TPM command directly to the TPM device driver, which in

[4] www.ntru.com

turn forwards the command to the TPM. After all the TPM commands have been executed, the connection is closed. The three commands necessary to achieve the above process are shown in table 12.12.

Table 12.12: TPM device driver communications

Tddli_Open	This function establishes a connection with the TPM device driver. Following a successful response to the *Tddli_Open* function the TPM device driver must be prepared to process TPM command requests from the calling application.
Tddli_TransmitData	This function sends a TPM command directly to a TPM device driver, causing the TPM to perform the corresponding operation.
Tddli_Close	This function closes the connection with the TPM device driver.

12.9.5 Endorsement key pair generation

An endorsement key pair must be associated with each TPM, as described in section A.7.4. This endorsement key pair can be generated using the sequence of commands shown in table 12.13. Alternatively, the endorsement key pair may be generated using an external key generator. When this process has been completed the endorsement key must be certified by the TPME, as described in section A.5.

Before generating an endorsement key pair, calls can be made to the *TPM_GetCapability* to determine whether or not an endorsement key already exists.

12.9.6 Accessing the public endorsement key

Table 12.14 gives a sequence of commands enabling the public endorsement key to be accessed. Access to the public endorsement key is required so that an entity can take ownership of the TPM.

Table 12.13: Creating an endorsement key pair

Create key object:	
Tspi_Context_CreateObject	The key object provides information about the endorsement key to be generated.
Tspi_SetAttribUint32	
Tspi_SetAttribData	
Create endorsement key:	
Tspi_TPM_CreateEndoresmentKey	
Tcsip_CreateEndorsementKeyPair	
TPM_CreateEndorsementKeyPair	

Table 12.14: Accessing the public endorsement key

Open access to the public endorsement key: By default, the flag which indicates whether or not open access to the public endorsement key is allowed, *TPM_PF_READPUBEK*, is set to *TRUE* so that the public endorsement key can be read without the input of owner authorisation data.	
Tspi_TPM_GetPubEndorsementKey	Used during the taking ownership process, before the TPM has acquired an owner. Outputs a handle to a key object representing the endorsement public key.
Tcsip_ReadPubek	Returns the public portion of the endorsement key to any entity.
TPM_ReadPubek	
Disable open access to the of public endorsement key:	
Often by default, once the TPM has acquired an owner, the flag which indicates whether or not open access to the public endorsement key is allowed, *TPM_PF_READPUBEK*, is set to *FALSE* so that the public endorsement key can only be read by the TPM owner. This flag may, however, be changed using the *TPM_SetCapability* command, which requires owner authorisation.	
Tspi_TPM_SetStatus	Used to set the status of the *TSS_TPMSTATUS_ DISABLEPUBEKREAD* to *FALSE*.
Tcsip_DisablePubekRead	
TPM_SetCapability	
TPM owner read of public endorsement key:	
Tspi_TPM_GetPubEndorsementKey	
Tcsip_OwnerReadPubek	
TPM_OwnerReadInternalPub	

12.9.7 TPM self testing

During the initialisation process, a minimal set of self tests are completed by the TPM. In order to ensure a more thorough self test, the commands shown in table 12.15 could be executed.

Table 12.15: Self testing

Continue self-test process:	
Tcsip_ContinueSelfTest	Informs the TPM that it should complete the self test of all TPM functions.
TPM_ContinueSelfTest	This command causes the TPM to test the TPM internal functions not tested at initialisation.
As stated above, the *TPM_ContinueSelfTest* command causes the TPM to test all the TPM internal functions that were not tested at start-up. If the TPM is running in compliance with FIPS-140 evaluation criteria, then the *TPM_ContinueSelfTest* command will cause the TPM to perform a complete self-test.	
Or	
Complete a full self-test:	
Tspi_TPM_SelfTestFull	Requests that the TPM completes a full self test.
Tcsip_SelfTestFull	
TPM_SelfTestFull	
The results of the self tests are held in the TPM.	

12.9.8 Enabling the TPM

The TPM must be enabled; that is the *TPM_PF_DISABLE* flag must be set to *FALSE*. This can be achieved using the commands shown in table 12.16.

Table 12.16: Physically enabling the TPM

Tspi_TPM_SetStatus	Used to set the status of the *TSS_TPMSTATUS_PHYSICALDISABLE* to *FALSE*.
Tcsip_PhysicalEnable	The TPM owner must enable the platform before any TPM commands can be utilised.
TPM_PhysicalEnable	

In order to physically disable the TPM before it has acquired an owner, the commands shown in figure 12.17 can be executed.

Table 12.17: Physically disabling the TPM

Tspi_TPM_SetStatus	Used to set the status of the *TSS_TPMSTATUS_PHYSICALDISABLE* to *TRUE*.
Tcsip_PhysicalDisable	
TPM_PhysicalDisable	

Once the TPM has acquired an owner, he or she may also enable or disable the TPM using the *TPM_OwnerSetDisable* command, which changes the state of the *TPM_PF_DISABLE* flag to either *TRUE* or *FALSE*. This command is shown in table 12.18.

Table 12.18: Enabling/Disabling the TPM

Tspi_TPM_SetStatus	
Tcsip_OwnerSetDisable	Used to change the status of the *TPM_PF_DISABLE* flag.
TPM_OwnerSetDisable	

12.9.9 The ownership flag

In order for a user to take ownership of a TPM, the ownership flag, *TPM_PF_OWNERSHIP* , must be set to *TRUE*, using the commands given in table 12.19. The default value for this flag is *TRUE*, so this command need never be called.

Table 12.19: Setting the state of the *TPM_PF_OWNERSHIP* flag

Tcsip_SetOwnerInstall	Used to set the value of the *TPM_PF_OWNERSHIP* flag to *TRUE*, so that an entity can take ownership of a TPM.
TPM_SetOwnerInstall	

12.9.10 Taking ownership of the TPM

In order for an entity to take ownership of a TPM, the following steps must be completed.

1. The public endorsement key must be accessed, as described in table 12.14.
2. TPM owner authorisation data must be input into the TPM.

3. A storage root key (SRK) must be generated inside the TPM.

4. The authorisation data for the SRK must be input (if required) into the TPM.

5. A tpmProof must be generated. A tmpProof is a 160-bit secret that is generated by the TPM when the *TPM_TakeOwnership* command is executed [5]. This secret is associated with non-migratable objects so that a TPM can identify the objects which it has created.

Steps 2 through 5 can be completed using the take ownership command sequence shown in table 12.20.

Table 12.20: Taking ownership of the TPM

Create policy object (owner authorisation data):
Tspi_Context_CreateObject
Tspi_SetAttribUint32
Tspi_SetAttribData
Tspi_Policy_SetSecret
Assign policy to the TPM object, whose handle can be retrieved using the *Tspi_Context_GetTPMObject*, as shown in table 12.7:
Tspi_Policy_AssignToObject
Create a key object (SRK):
Tspi_Context_CreateObject
Tspi_SetAttribUint32
Tspi_SetAttribData
Create policy object for the SRK (SRK authorisation data):
Tspi_Context_CreateObject

Tspi_SetAttribUint32

Tspi_SetAttribData

Tspi_Policy_SetSecret

Assign policy to the SRK object:

Tspi_Policy_AssignToObject

Read public EK — may be accomplished using the command sequences described in table 12.14:

Take ownership:

Tspi_TPM_TakeOwnership

Tcsip_TakeOwnership

TPM_TakeOwnership

12.9.11 TPM activation

Finally, the TPM must be activated; this will result in the *TPM_PF_DEACTIVATED* flag being set to *FALSE*. This can be activated using the commands shown in table 12.21.

Table 12.21: Activating the TPM

Tspi_TPM_SetStatus	Used to set the status of the *TSS_TPMSTATUS_ PHYSICALSETDEACTIVATED* to *FALSE*.
Tcsip_PhysicalSetDeactivated	
TPM_PhysicalSetDeactivated	

12.10 Secure storage

Requirement 1, as described in section 12.2, may be partially met through the deployment of a secure storage mechanism, as can requirements 2–5 and 8–21.

12.10.1 Key hierarchy

Each stakeholder may build up their own key hierarchy. The method by which this is done will depend on the TMP architecture. For the particular use case

considered here, the focus is on the OMA DRM v2 agent installer, the key hierarchy for which is represented in figure 12.2.

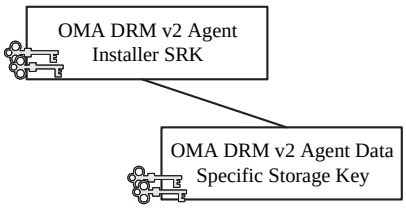

Figure 12.2: OMA DRM v2 agent installer key hierarchy

The storage root key illustrated in this particular key hierarchy may, in fact, be an SRK, as defined in the TCG v1.2 specifications, or it may represent the root of the agent installer's key hierarchy, which is a branch in a key hierarchy which has as its root a TPM SRK as defined in the TCG v1.2 specifications.

12.10.2 Installing integrity and confidentiality sensitive OMA DRM v2 data on the device

If sensitive OMA DRM v2 data is installed on the device in a controlled environment, and is not entered into the TMP remotely, we can assume that the confidentiality and integrity of the data will not be compromised before it is protected using TPM v1.2 functionality. This appears to be the most likely scenario.

Alternatively, a secure transport session may be set up with the TPM so that all input parameters into the secure storage commands described below may be protected while being communicated to the TPM.

Transport security enables the establishment of a secure channel between the TPM and a secure process, offering confidentiality and integrity protection of commands sent to the TPM. It also provides a logging function so that all commands sent to the TPM during a transport session can be recorded.

Session establishment involves the generation of 20 bytes of transport authorisation data by the caller, for use between the caller and the TPM. This transport authorisation data has two purposes:

- It is used to generate a secret key for use in encrypting commands sent from the application to the TPM; and
- It is also used to generate a secret HMAC key to provide origin authentication and integrity protection for the *TPM_ExecuteTransport* command.

The authorisation data is generated by the caller and encrypted under a public key whose corresponding private key is available only to the TPM. The key used is pointed to by the encHandle field of the *TPM_EstablishTransport* command.

In the command sequence described in table 12.22, the context object which is created, and to which the session is connected, will be required to possess certain additional transport session specific attributes.

Table 12.22: A transport session

Create policy object (transport authorisation data):
Tspi_Context_CreateObject
Tspi_SetAttribUint32
Tspi_SetAttribData
Tspi_Policy_SetSecret
Transport key object: A handle to the transport key object must be retrieved. By default the TSP uses a non-migratable storage key to establish the transport session. If this key is not to be used, any other key can be provided via a UUID or key handle using the *Tspi_Context_SetTransEncryptionKey* command.

Set transport key:

Tspi_Context_SetTransEncryptionKey

Establish transport session:

Tcsip_EstablishTransport

TPM_EstablishTransport

Execute transport session:

Tcsip_ExecuteTransport

TPM_ExecuteTransport

Close transport session:
This command terminates the transport session, and, if logging is switched on, a signed hash of all operations completed during the session is output. In order to complete this command sequence, a signing key must have been created for this purpose, and its handle communicated as input to the *Tspi_Context_CloseSignTransport* method.

Tspi_Context_CloseSignTransport

Tcsip_ReleaseTransportSigned

TPM_ReleaseTransportSigned

We assume, however, for the purposes of this chapter that sensitive OMA DRM v2 data is installed on the device in a controlled environment. Therefore, we can assume that the confidentiality and integrity of the data will not be compromised before it is protected using TPM v1.2 functionality.

12.10.3 Secure storage of and access control to OMA DRM v2 data

OMA DRM v2 data which needs to be integrity protected, for example the device details and the trusted rights issuer authorities certificate, may be MACed using cryptographic functionality provided by a TCG independent cryptographic infrastructure (CI), implemented on the platform. This CI may then utilise the TSP to access the TCS, and thus the TPM, so that a *TPM_Seal* can be called and the MAC key stored securely. This sealing mechanism can confidentiality protect the MACing key and ensure that it is only accessible to the legitimate OMA DRM v2 agent.

Alternatively, data which needs integrity protection, in conjunction with data which is required to be both integrity and confidentiality protected, may be directly sealed by the TPM such that it is only accessible to the legitimate OMA DRM v2 agent.

Because of the limited size of the OMA DRM v2 data which needs integrity protection, e.g. the device details and the trusted RI authorities certificate, it would be more practical and efficient to directly seal the data rather then MACing the data and sealing the key.

Integrity protection is not explicitly provided by the TPM. In order to integrity protect sealed data, 20 bytes of authorisation data needs to be associated with it. This authorisation data may be entered by the TMP user via the PC user interface, e.g. using a pop-up dialog box, or, it may be sealed to the OMA DRM v2 agent (or, more specifically, to PCR values which represent a trustworthy execution environment in which a correctly functioning OMA DRM v2 agent is running). In this way, only the correctly functioning OMA DRM v2 agent can unseal the authorisation data and then unseal the OMA DRM v2 data. The OMA DRM v2 data can only be unsealed if knowledge of the correct authorisation data is demonstrated and the current platform environment is represented by the PCR values bound to the OMA DRM v2 data when it was sealed.

Therefore, all DRM data, both data which is required to be integrity-protected and that which requires both integrity and confidentiality protection, should be sealed so that only a legitimate OMA DRM v2 agent can access and utilise it.

In order to protect OMA DRM v2 agent data, a key hierarchy as described in figure 12.2 must initially be set-up, and then the data sealed to the appropriate PCRs using the 'OMA DRM v2 data specific storage key', (a non-migratable

storage key) so that it can only be accessed by the legitimate OMA DRM v2 agent. This can be achieved using the following sequence of steps.

1. Load the OMA DRM v2 agent installer SRK and obtain a handle to the SRK.

   If the OMA DRM v2 agent installer SRK is a TPM SRK, as defined in the TCG v1.2 specifications, then it will not need to be loaded, since a TPM SRK is permanently loaded. In this case, in order to access and utilise the SRK, an SRK object must be created and a handle to the SRK object retrieved. If, however, the OMA DRM v2 agent installer SRK is the root of the agent installer's key hierarchy, which is itself a branch in a key hierarchy which has as its root a TPM SRK as defined in the TCG v1.2 specifications, the key may need to be loaded before use, which can be achieved using the command sequence described in table 12.24.

   Clearly, the SRK must first be created before it can be used. If it is a TPM SRK then it will have been created during the take ownership process described in table 12.20. Otherwise it could have been created using the command described in table 12.23.

2. The OMA DRM v2 specific storage key (OSSK) needs to be created under the agent installer SRK, again using the command sequence shown in table 12.23;

Table 12.23: Creating a wrap key

Create key object:
Tspi_Context_CreateObject
Tspi_SetAttribUint32
Tspi_SetAttribData

Create policy object (key authorisation data):

If we want to associate a policy object other than the default policy to the key object, the 'create policy object' and the 'assign policy to key object' command runs are used.

Tspi_Context_CreateObject

Tspi_SetAttribUint32

Tspi_SetAttribData

Tspi_PolicySetSecret

Assign policy to key object:

Tspi_Policy_AssignToObject

Create policy object (key migration data):
In this instance, however, we require the key to be created to be non-migratable.

Tspi_Context_CreateObject

Tspi_SetAttribUint32

Tspi_SetAttribData

Tspi_PolicySetSecret

Assign policy to key object:

Tspi_Policy_AssignToObject

Create PCR composite object (only required if the key to be generated is bound to PCR values):

Tspi_Context_CreateObject
We assume that the PcrComposite object created is set to use a *TPM_PCR_INFO_LONG* structure or a *TPM_PCR_INFO_SHORT* structure.

Tspi_SetAttribUint32

Tspi_PcrComposite_SetPcrLocality
This method sets the *LocalityAtRelease* inside the PCR composite object using a version 1.2 *TPM_PCR_INFO_LONG* or *TPM_PCR_INFO_SHORT* structure.

Tspi_PcrComposite_SelectPcrIndexEx
This method selects a PCR index inside a PCR composite object containing a *TPM_PCR_INFO_LONG* or *TPM_PCR_INFO_SHORT* structure. For the *TPM_PCR_INFO_LONG* structure, the index may be selected for *DigestAtCreation* or *DigestAtRelease*. For *TPM_PCR_INFO_SHORT*, the index may be selected only for *DigestAtRelease*.

Tspi_PcrComposite_SetPcrValue
This method sets the *DigestAtRelease* for a given PCR index inside the PCR composite object. Multiple PCRs with different indexes can be set by calling this method multiple times on the same PCR composite object.

Tspi_Key_Createkey

Tcsip_CreateWrapKey

TPM_CreateWrapKey

3. Load the OSSK, as shown in table 12.24.

Table 12.24: Loading a key

Assume we have the handle of the parent unwrapping key of the key to be loaded: in this instance, in the context of figure 12.23, this is the OMA DRM v2 SRK handle:
There are four possible ways to load a key, depending on whether the key is registered in persistent storage or not and depending on whether the parent key requires authorisation or not.
If the key is to be loaded by the input of a wrapped key blob, where the wrapping key has been loaded and its handle is available, the following command sequence is used. Depending on the parent key, authorisation may or may not be required.
Tspi_Context_LoadKeyByBlob
Tcsip_LoadKeyByblob
TPM_LoadKey2
If the key to be loaded is registered in persistent storage, and if the parent key does not require authorisation, the following command sequence is used:
Tspi_Context_LoadKeyByUUID
Tcspi_LoadKeyByUUID
TPM_LoadKey2
If the key to be loaded is registered in persistent storage, if its parent key requires authorisation, and if the application knows the registered key stack, the following command sequence is used:

Tspi_Context_GetKeyByUUID

Tspi_Key_LoadKey

Tcspi_LoadKeyByUUID

TPM_LoadKey2

If the key to be loaded is registered in persistent storage, if its parent key requires authorisation, and if the application does not know the registered key stack, the following command sequence is used, after which the command sequence continues as above:

Tspi_ContextGetRegisteredKeysByUUID

Tspi_Context_GetKeyByUUID

Tspi_Key_LoadKey

Tcspi_LoadKeyByUUID

TPM_LoadKey2

4. Finally, seal the OMA DRM v2 data using the OSSK, as shown in table 12.25.

Table 12.25: Sealing data using a storage key

Create an encrypted data object:

Tspi_Context_CreateObject

Tspi_SetAttribUint32

Tspi_SetAttribData

Create policy object (the authorisation data to be associated with the sealed data):

Tspi_Context_CreateObject

Tspi_SetAttribUint32

Tspi_SetAttribData

Tspi_PolicySetSecret

Assign policy to object:

Tspi_Policy_AssignToObject

Create PCR composite object (if the sealed data is to be sealed to a PCR set):

Tspi_Context_CreateObject

Tspi_SetAttribUint32

Tspi_PcrComposite_SetPcrLocality

Tspi_PcrComposite_SelectPcrIndexEx

Tspi_PcrComposite_SetPcrValue

Seal data:

Tspi_Data_Seal

Tcspi_Seal

TPM_Seal

In order to protect the OMA DRM v2 agent private key (if it has not been generated on the platform) the following commands are executed so that it can only be accessed by the legitimate OMA DRM v2 agent.

1. As was described in section 12.10.3, if the OMA DRM v2 agent installer SRK is a TPM SRK, as defined in the TCG v1.2 specifications, it will not need to be loaded, since the TPM SRK is permanantly loaded. If, however, the OMA DRM v2 agent installer SRK is the root of the agent installer's key hierarchy, which is itself a branch in a key hierarchy which has as its root a TPM SRK as defined in the TCG v1.2 specifications, the key may need to be loaded before use, as described in table 12.24.

   This key must also exist before it may be used. If it is a TPM SRK as defined in the TCG v1.2 specification set then it will have been created during the take ownership process as described in table 12.20. Otherwise, it may have been created using the command sequence given in table 12.23.

2. An OSSK needs to be created as described in table 12.23 and loaded, as described in table 12.24. In this way, the handle to the wrapping key,

Table 12.26: Wrapping a key to a PCR(s)

Create key object (which will contain the key to be wrapped, in this instance the OMA DRM v2 key):
Tspi_Context_CreateObject
Tspi_SetAttribUint32
Tspi_SetAttribData
Create policy object (the authorisation data to be associated with the wrapped key):
Tspi_Context_CreateObject
Tspi_SetAttribUint32
Tspi_SetAttribData
Tspi_PolicySetSecret
Assign policy to object:
Tspi_Policy_AssignToObject
Create PCR composite object (if the wrapped key is to be bound to PCR values):
Tspi_Context_CreateObject
Tspi_SetAttribUint32
Tspi_PcrComposite_SetPcrLocality
Tspi_PcrComposite_SelectPcrIndexEx
Tspi_PcrComposite_SetPcrValue
Wrap key:
Tspi_Key_WrapKey

OSSK, is retrieved.

3. Finally, the DRM agent key needs to be wrapped to specified PCR values using the OSSK, using the command sequence given in table 12.26.

Generation of the OMA DRM v2 agent key pair on the TPM could have significant security advantages.

12.10.4 Security of the OMA DRM v2 data while in use on the device

The PCRs which represent the execution environment into which the OMA DRM v2 data can be released, are presumed to represent a secure and trustworthy environment. The mechanisms described in section 12.8 can be used to protect the OMA DRM v2 agent while it is executing on the platform.

12.11 Platform attestation

Requirement 1 may also necessitate a mechanism that allows a TMP to attest to the integrity metrics of specified platform components, where the integrity measurements have been generated by the root of trust for measurement.

In order to meet this requirement, RTM functionality, as described within the TCG specifications, must first be utilised so that the integrity of the platform can be measured and the resulting integrity values stored to TPM PCRs.

In order to attest to platform integrity metrics, the following command runs must be completed.

1. Generate and activate a platform attestation identity key, using the command sequence given in table 12.27.
2. Attest to the requested PCR values.
3. Gather the corresponding event log and send it to the challenger.

Steps 2 and 3 can be completed using the platform attestation command sequence shown in table 12.28.

Table 12.27: Creating a platform attestation identity key

Create key object (a new attestation identity key pair):

Tspi_Context_CreateObject

Tspi_SetAttribUint32

Tspi_SetAttribData

Create policy object (authorisation data to be associated with the new attestation identity key):

Tspi_Context_CreateObject

Tspi_SetAttribUint32

Tspi_PolicySetSecret

Assign policy to key object:

Tspi_Policy_AssignToObject

Create key object (which represents the public key of the P-CA):

Tspi_Context_CreateObject

Tspi_SetAttribUint32

The TPM handle can be retrieved using the *Tspi_Context_GetTPMObject* method, and a handle to a SRK object can be retrieved by creating a SRK object using *Tspi_Context_CreateObject*:
Make identity:

Tspi_TPM_CollateIdentityRequest

Tcspi_MakeIdentity

TPM_MakeIdentity

Activate identity:

Tspi_TPM_ActivateIdentity

Tcspi_ActivateTPMIdentity

TPM_ActivateIdentity

Table 12.28: Platform attestation

Load the attestation identity key as described in table 12.24 and retrieve the handle to the attestation identity key object:

Create PCR object:

Tspi_Context_CreateObject

Tspi_SetAttribUint32

Tspi_PcrComposite_SelectPcrIndexEx

TPM quote:

Tspi_TPM_Quote

Tcsip_Quote

TPM_Quote

The corresponding event log:

In conjunction with the output from the *TPM_Quote* command, an event log, which describes what the integrity metrics output from the *TPM_Quote* command represent, must also be sent to the challenger. This event log may contain a single event, which is represented as a single *TSS_PCR_EVENT* structure; a group of events, which are represented as the group of selected *TSS_PCR_EVENT* structures; or, the entire event log, which is represented as an ordered sequence of *TSS_PCR_EVENT* structures.

If a PCR event for a given PCR index and event number is required:

Tspi_TPM_GetEvent

Tcsi_GetPcrEvent

If a specific number of events for a given PCR index are required:

Tspi_TPM_GetEvents

Tcsi_GetPcrEventsByPcr

If the entire event log is required:

Tspi_TPM_GetEventLog

Tcsi_GetPcrEventLog

12.12 Demonstrating privilege

In order to demonstrate the level of privilege required to execute various TPM commands:

- An entity may demonstrate physical presence at the platform; or, alternatively,
- An entity may demonstrate knowledge of the required authorisation data.

There are three particular occasions where demonstration of physical presence at the platform may be necessary in order to execute particular TPM commands, usually in the case when cryptographic authorisation is unavailable. These occasions include the operation of commands that control the TPM before an owner has been installed; when the TPM owner has lost cryptographic authorisation information; or when the host platform cannot communicate with the TPM.

As an alternative to physical presence, cryptographic authorisation mechanisms may be used to authenticate an owner to his or her TPM, or to authorise the release and use of TPM protected objects. An authorisation value must be 20 bytes long, and could, for example, be a hashed password or 20 bytes from a smartcard. It must always be treated as shielded data and only ever used in the authorisation process.

Many of the TPM commands (specifically the TPM owner authorised commands) described in this chapter may require knowledge of the required authorisation data to be demonstrated before access to the command is permitted. Similar constraints can also apply to a key or a data object. A variety of authorisation data can be held by a TPM, including:

- Unique TPM owner authorisation data, input of which is required before any 'owner-authorised TPM command' may be executed;
- TPM object usage authorisation data, input of which is required before an object protected by the TPM may be accessed; and
- TPM object migration authorisation data, input of which is required before a TPM key object can be migrated.

In order to demonstrate knowledge of the relevant authorisation data to the TPM, an entity may deploy one of two challenge-response protocols, namely the object independent authorisation protocol (OIAP), or the object specific authorisation protocol (OSAP).

OIAP is the more flexible and efficient of the two challenge-response authorisation protocols. Once an OIAP session has been established, it can be used to demonstrate knowledge of the authorisation data associated with a particular TPM object or TPM command.

In order to input the required authorisation data using OIAP, a number of steps must be followed:

1. A working object, which represents the object to be used/accessed, must be created, and the handle retrieved.
2. A policy object must be assigned to the working object so that the required authorisation data can be collected.
3. The required Tspi method is then called.
4. An OIAP session is established using the *Tcsip_OIAP* method. *Tcsip_OIAP* allows the creation of an authorisation handle and the tracking of the handle by the TPM. The TPM generates the handle and nonce.

5. The required Tcspi method is called.

6. The *TPM_OIAP* command is called.

7. Finally, the required TPM command is called.

We now re-examine how the TPM owner reads the public endorsement key using the commands shown in table 12.14. Knowledge of the owner authorisation data must be demonstrated in order to gain access to the public endorsement key.

Table 12.29: TPM owner reading of the public endorsement key

Tspi_TPM_GetPubEndorsementKey	Outputs a handle to a key object representing the endorsement public key.
Assign a policy object to the key object using the *Tspi_Policy_AssignToObject* method, so that the TPM owner authorisation data can be collected.	
Tcsip_OIAP	
Tcsip_OwnerReadPubek	
TPM_OIAP	
TPM_OwnerReadInternalPub	

The second protocol defined in the TCG specifications is OSAP. This protocol supports the establishment of a session to prove knowledge of the authorisation data for a single TPM object, and minimises the exposure of long-term authorisation values. It may be used to authorise multiple commands without additional session establishment but, as we discuss below, the *TPM_OSAP* handle specifies a single object to which all authorisations are bound.

During this protocol an ephemeral secret is generated (using the HMAC of the session nonces exchanged at the beginning of the protocol, with the target TPM object's authorisation data acting as the HMAC key) by the TPM and the caller, which is used to prove knowledge of the TPM object authorisation data.

This particular protocol must also be used with operations that set or reset authorisation data, e.g. sealing or creating a wrap key.

In order to input the required authorisation data a number of steps must be followed:

1. A handle to the object to be used/accessed must be retrieved.
2. A policy object must be assigned to the working object so that the required authorisation data can be collected.
3. The required Tspi method called.
4. An OSAP session is established using the *Tcsip_OSAP* method.
5. The required Tcspi method is called.
6. The *TPM_OSAP* command must be called. TPM_OSAP creates the authorisation handle and the shared secret, and generates *nonceEven* and *nonceEvenOSAP*.
7. The required TPM command is called.
8. The shared secret which is generated can be used both to authorise use of the parent object and to input the authorisation data for a newly created child object, for example a new key or sealed data object.
9. Once this has been completed, the OSAP session can be kept open in order to authorise another command which is bound to the same parent object.

We now re-examine the load key command sequence shown in table 12.24, where we assume that the key, OSSK, is to be loaded by the input of a wrapped key blob. It is also assumed that the agent installer SRK is loaded and its handle is available, and that the parent key, i.e. the agent installer SRK, requires authorisation.

In order to load the OSSK, knowledge of the agent installer SRK authorisation data must be demonstrated. When the OSSK has been loaded, a seal command, as described in table 12.25, is called. Use of the OSSK must also be authorised.

In this case, the user can demonstrate knowledge of the parent wrapping key (the agent installer SRK) authorisation data when loading the non-migratable key, OSSK, using an OIAP, for example. When sealing the OMA DRM v2 data using the OSSK, knowledge of the OSSK authorisation data can be demonstrated and the authorisation data for the sealed data inserted using the shared key established during the initial steps of the OSAP. This process is shown in table 12.30.

Table 12.30: Authorising a load key and an object seal

Assume we have the handle of the agent installer SRK
Assign a policy object to the agent installer SRK object using the *Tspi_Policy_AssignToObject* method to authorise use of the SRK
Assuming OSSK is to be loaded by the input of a wrapped key blob, where the wrapping key, i.e. the agent installer SRK, has been loaded and its handle is available, the following command sequence is then used.
Tspi_Context_LoadKeyByBlob
Tcsip_OIAP
Tcsip_LoadKeyByblob
TPM_OIAP
TPM_LoadKey2
Now we have the handle to OSSK
Assign a policy object to the OSSK object using the *Tspi_Policy_AssignToObject* method to authorise use of the OSSK
Create an encrypted data object (for the OMA DRM v2 data to be sealed):

Tspi_Context_CreateObject

Tspi_SetAttribUint32

Tspi_SetAttribData

Create policy object (the authorisation data to be associated with the sealed data):

Tspi_Context_CreateObject

Tspi_SetAttribUint32

Tspi_SetAttribData

Tspi_PolicySetSecret

Assign policy to object:

Tspi_Policy_AssignToObject

Create PCR composite object (if the sealed data is to be sealed to a PCR set):

Tspi_Context_CreateObject

Tspi_SetAttribUint32

Tspi_PcrComposite_SetPcrLocality

Tspi_PcrComposite_SelectPcrIndexEx

Tspi_PcrComposite_SetPcrValue

Seal data:

Tspi_Data_Seal

Tcsip_OSAP

Tcspi_Seal

TPM_OSAP

TPM_Seal

12.13 Random number generation

Requirement 6 necessitates that pseudo-random number generator (PRNG) functionality is provided by the TMP.

A TPM contains an RNG (which provides nonces which are both random and unpredictable), and the commands listed in table 12.31 may be executed in

order to access a random number.

Table 12.31: Random number retrieval

Tspi_TPM_GetRandom
Tcspi_GetRandom
TPM_GetRandom

12.14 Trusted time source

Requirement 7 necessitates a mechanism which supports the implementation of a trusted time source.

The version 1.2 TCG specifications include a design document which discusses time stamping [156]. This discussion explores the capability of a TPM to apply a timestamp to a binary object. The timestamp provided by the TPM, however, is not a coordinated universal time (UTC) value but the number of ticks that the TPM has counted. It becomes the responsibility of the caller to associate the tick count with the corresponding UTC time.

While no particular protocol is mandated by the TCG specifications in order to accomplish this association of the tick count value with the UTC time, a sample protocol is described.

In this particular use case, we are interested in the use of the trusted time source to successfully update the time source available to the OMA DRM v2 agent, so that the protocols outlined in the OMA v2 specifications which deal with clock drift and clock synchronisation may be deprecated. These protocols involve OCSP interactions following the detection of an inaccurate time in a registration request message.

On examination of the effort required to complete the OMA v2 protocols, and the effort required to complete the TCG time stamping protocols, there

may, however, be little to be gained by using TCG protocols to update the device time source.

12.15 Conclusions

In this chapter, the requirements extracted in chapter 11 were utilised in order to examine which architectural components and functionality described within TCG version 1.2 specification set could be used to facilitate a robust implementation of OMA DRM v2.

Table 12.32 summarises the subset of version 1.2 TPM commands required in a mobile TPM in order to enable this use case. As can been seen from this table, support for key migration or the availability of certifiable migratable keys is not required for this particular use case. Neither is direct anonymous attestation (DAA) functionality. It also appears unlikely, as described in section 12.22, that transport protection will be required. The delegation mechanism is not required in order to implement this use case. As is the case in the v1.2 TPM specification, audit and maintenance denote optional functionality which may be provided by the TPM manufacturer but are not necessarily required. In order to implement this use case, it is required, however, that the TPM can be taken ownership of. In conjunction with this, basic functionality, such as self testing, is also needed. Measurement functionality is required; a root of trust for measurement and TPM support for such a trust root, i.e. *TPM_Extend* and *TPM_PCRRead* commands, must be provided. In conjunction with this, secure storage, key management, attestation and command authorisation functionality is mandatory in order to robustly implement OMA DRM v2.

Table 12.32: TPM commands required for a robust implementation of OMA DRM v2

TPM version 1.2 command	Mobile device TPM
TPM_Init	required
TPM_Startup	required
TPM_SaveState	optional
TPM_SelfTestFull	required
TPM_ContinueSelfTest	required
TPM_GetTestResult	required
TPM_SetOwnerInstall	required
TPM_OwnerSetDisable	optional
TPM_PhysicalEnable	required
TPM_PhysicalDisable	required
TPM_PhysicalSetDeactivated	required
TPM_SetTempDeactivated	optional
TPM_SetOperatorAuth	optional
TPM_TakeOwnership	required
TPM_OwnerClear	optional
TPM_ForceClear	optional
TPM_DisableOwnerClear	optional
TPM_DisableForceClear	optional
TPM_GetCapability	required
TPM_SetCapability	optional
TPM_GetAuditDigest	optional
TPM_GetAuditDigestSigned	optional
TPM_SetOrdinalAuditStatus	optional
TPM_FieldUpgrade	optional
TPM_SetRedirection	optional
TPM_ResetLockValue	optional
TPM_Seal	required
TPM_Unseal	required
TPM_Unbind	optional
TPM_CreateWrapKey	required
TPM_LoadKey2	required
TPM_GetPubKey	optional
TPM_Sealx	optional
TPM_CreateMigrationBlob	optional
TPM_ConvertMigrationBlob	optional
TPM_AuthorizeMigrationKey	optional
TPM_MigrateKey	optional
TPM_CMK_SetRestrictions	optional
TPM_CMK_ApproveMA	optional
TPM_CMK_CreateKey	optional
TPM_CMK_CreateKey	optional
TPM_CMK_CreateTicket	optional
TPM_CMK_CreateBlob	optional
TPM_CMK_ConvertMigration	optional
TPM_CreateMaintenanceArchive	optional

TPM_LoadMaintenanceArchive	optional
TPM_KillMaintenanceFeature	optional
TPM_LoadManuMaintPub	optional
TPM_ReadManuMaintPub	optional
TPM_SHA1Start	optional
TPM_SHA1Update	optional
TPM_SHA1Complete	optional
TPM_SHA1CompleteExtend	optional
TPM_Sign	optional
TPM_GetRandom	required
TPM_StirRandom	required
TPM_CertifyKey	optional
TPM_CertifyKey2	optional
TPM_CreateEndorsementKeyPair	optional
TPM_CreateRevokableEK	optional
TPM_RevokeTrust	optional
TPM_ReadPubek	required
TPM_OwnerReadInternalPub	optional
TPM_MakeIdentity	required
TPM_ActivateIdentity	required
TPM_Extend	required
TPM_PCRRead	required
TPM_Quote	required
TPM_PCR_Reset	optional
TPM_Quote2	optional
TPM_ChangeAuth	optional
TPM_ChangeAuthOwner	optional
TPM_OIAP	required
TPM_OSAP	required
TPM_DSAP	optional
TPM_SetOwnerPointer	optional
TPM_Delegate_Manage	optional
TPM_Delegate_CreateKeyDelegation	optional
TPM_Delegate_CreateOwnerDelegation	optional
TPM_Delegate_LoadOwnerDelegation	optional
TPM_Delegate_ReadTable	optional
TPM_Delegate_UpdateVerification	optional
TPM_Delegate_VerifyDelegation	optional
TPM_NV_DefineSpec	optional
TPM_NV_WriteValue	optional
TPM_NV_WriteValueauth	optional
TPM_NV_ReadValue	optional
TPM_NV_ReadValueAuth	optional
TPM_KeyControlOwner	optional
TPM_SaveContext	optional
TPM_LoadContext	optional
TPM_FlushSpecific	required
TPM_GetTicks	optional
TPM_TickStampBlob	optional
TPM_EstablishTransport	optional

TPM_ExecuteTransport	optional
TPM_ReleaseTransportSigned	optional
TPM_CreateCounter	optional
TPM_IncrementCounter	optional
TPM_ReadCounter	optional
TPM_ReleaseCounter	optional
TPM_ReleaseCounterOwner	optional
TPM_DAA_Join	optional
TPM_DAA_Sign	optional

Each of the necessary TPM commands should also be supported by the TSS commands described in sections 12.6 to 12.9, if the TMP is to support a TSS.

The requirements extracted in chapter 11 were also utilised in order to identify architecture components and functionality not currently specified within the TCG version 1.2 specification set, but required for the implementation of a robust and secure DRM solution on a trusted mobile platform. Two additional mechanisms were identified, namely a secure boot mechanism and a mechanism which ensures that the integrity of the platform is maintained after boot. In order to implement secure boot, we identified a number of fundamental components which need to be considered, including:

- Platform component RIMs;
- RIM certificates;
- The list of entities authorised to sign RIM certificates;
- A RTV;
- A CRTV;
- The interaction between the RTV and the RTM;
- Platform recovery; and
- RIM certificate revocation and update.

In order to maintain integrity after boot, a number of preventative approaches were examined in section 12.8 in conjunction with a high-level reactive mechanism which is closely coupled to the concept of secure boot was identified.

When attempting to implement this use case using TCG functionality, every effort was made not to require modifications to the OMA specifications. If, however, TPM functionality were to be made available on all mobile devices, it may be useful for the OMA to leverage the key generation capabilities of the TPM in order to generate the OMA DRM v2 agent key pair.

While only a subset of the secure storage, key management, attestation and command authorisation functionality is required in order to implement this use case, on examination of the remainder of the use cases described in the MPWG use case document [159], namely, SIMLock/Device personalisation, software download, mobile ticketing, mobile payment, software use, proving platform and/or application integrity to end user user data, and protection and privacy, it appears that the functionality required in order to enable these use cases would also include both secure boot and run-time integrity checking. It seems highly likely, given a high-level examination of the use cases, that secure storage, key management, attestation and command authorisation functionality, in conjunction with the fundamental TPM command runs, would also be required. Given these assumptions, it would seem highly likely that a MTPM profile would need to include the functionality described in table 12.33. However, in order to verify this hypothesis, further investigation must be completed on each of the remaining use cases.

Table 12.33: TPM commands required in a MTPM

TPM version 1.2 command	Mobile device TPM
TPM_Init	required
TPM_Startup	required
TPM_SaveState	optional
TPM_SelfTestFull	required
TPM_ContinueSelfTest	required
TPM_GetTestResult	required
TPM_SetOwnerInstall	required
TPM_OwnerSetDisable	optional
TPM_PhysicalEnable	required
TPM_PhysicalDisable	required
TPM_PhysicalSetDeactivated	required
TPM_SetTempDeactivated	optional
TPM_SetOperatorAuth	optional
TPM_TakeOwnership	required
TPM_OwnerClear	optional
TPM_ForceClear	optional
TPM_DisableOwnerClear	optional
TPM_DisableForceClear	optional
TPM_GetCapability	required
TPM_SetCapability	optional
TPM_GetAuditDigest	optional
TPM_GetAuditDigestSigned	optional
TPM_SetOrdinalAuditStatus	optional
TPM_FieldUpgrade	optional
TPM_SetRedirection	optional
TPM_ResetLockValue	optional
TPM_Seal	required
TPM_Unseal	required
TPM_Unbind	required
TPM_CreateWrapKey	required
TPM_LoadKey2	required
TPM_GetPubKey	required
TPM_Sealx	required
TPM_CreateMigrationBlob	optional
TPM_ConvertMigrationBlob	optional
TPM_AuthorizeMigrationKey	optional
TPM_MigrateKey	optional
TPM_CMK_SetRestrictions	optional
TPM_CMK_ApproveMA	optional
TPM_CMK_CreateKey	optional
TPM_CMK_CreateKey	optional
TPM_CMK_CreateTicket	optional
TPM_CMK_CreateBlob	optional
TPM_CMK_ConvertMigration	optional
TPM_CreateMaintenanceArchive	optional

TPM_LoadMaintenanceArchive	optional
TPM_KillMaintenanceFeature	optional
TPM_LoadManuMaintPub	optional
TPM_ReadManuMaintPub	optional
TPM_SHA1Start	optional
TPM_SHA1Update	optional
TPM_SHA1Complete	optional
TPM_SHA1CompleteExtend	optional
TPM_Sign	required
TPM_GetRandom	required
TPM_StirRandom	required
TPM_CertifyKey	required
TPM_CertifyKey2	optional
TPM_CreateEndorsementKeyPair	optional
TPM_CreateRevokableEK	optional
TPM_RevokeTrust	optional
TPM_ReadPubek	required
TPM_OwnerReadInternalPub	optional
TPM_MakeIdentity	required
TPM_ActivateIdentity	required
TPM_Extend	required
TPM_PCRRead	required
TPM_Quote	required
TPM_PCR_Reset	optional
TPM_Quote2	optional
TPM_ChangeAuth	required
TPM_ChangeAuthOwner	optional
TPM_OIAP	required
TPM_OSAP	required
TPM_DSAP	optional
TPM_SetOwnerPointer	optional
TPM_Delegate_Manage	optional
TPM_Delegate_CreateKeyDelegation	optional
TPM_Delegate_CreateOwnerDelegation	optional
TPM_Delegate_LoadOwnerDelegation	optional
TPM_Delegate_ReadTable	optional
TPM_Delegate_UpdateVerification	optional
TPM_Delegate_VerifyDelegation	optional
TPM_NV_DefineSpec	optional
TPM_NV_WriteValue	optional
TPM_NV_WriteValueauth	optional
TPM_NV_ReadValue	optional
TPM_NV_ReadValueAuth	optional
TPM_KeyControlOwner	optional
TPM_SaveContext	optional
TPM_LoadContext	optional
TPM_FlushSpecific	required
TPM_GetTicks	optional
TPM_TickStampBlob	optional
TPM_EstablishTransport	optional

TPM_ExecuteTransport	optional
TPM_ReleaseTransportSigned	optional
TPM_CreateCounter	optional
TPM_IncrementCounter	optional
TPM_ReadCounter	optional
TPM_ReleaseCounter	optional
TPM_ReleaseCounterOwner	optional
TPM_DAA_Join	optional
TPM_DAA_Sign	optional

Chapter 13

Conclusions

Contents

13.1 Summary and conclusions 445
13.1.1 Part I: Mobile host protection 445
13.1.2 Part II: Mobile code protection 454
13.1.3 Part III: Remote code protection 458
13.2 Future work 460

This thesis deals with authorisation issues for mobile executables in mobile systems. In this chapter we summarise the main conclusions and original contributions of this thesis, and give suggestions for future work in the area.

13.1 Summary and conclusions

This thesis deals with authorisation issues for mobile executables in mobile systems and is subdivided into three parts. In this section we summarise the main conclusions and original contributions of each of these three parts.

13.1.1 Part I: Mobile host protection

Part I focused on the authorisation of incoming mobile code and agents by a mobile host. We began by introducing the mobile agent paradigm. We then highlighted the security threats which may impact a mobile host on which malicious mobile code, and more specifically a malicious mobile agent, executes. Finally, we reviewed the state of the art in mobile code and agent authorisation. Based on this state of the art review, it became apparent that, while solutions to the problem of mobile code and agent authorisation have been proposed, many are ill-suited for application in a mobile environment.

With the goal of assessing mechanisms which may be of use in meeting our objective of assembling a scheme for code and agent authorisation, we initially examined behaviour-based authorisation techniques. The deployment of this type of mechanism is beneficial in an open environment. However, on examination of proof-carrying code, model-carrying code and language security, a number of shortcomings were unearthed. While being able to "conclude, automatically and with certainty, that program code, provided by another system, is safe to install and execute" [107] is desirable, there are still questions surrounding how proofs of code should be constructed, in what formalism, and how it can be guaranteed that proofs can be verified efficiently and simply. Currently, the number and type of properties that can be captured in proofs is limited [138], and in the case of large executables, the size of the proof can sometimes be

larger than the code itself. While 'safe' or secure languages, such as Java, are effective in making the execution of mobile code and agents less hazardous, security then becomes dependent on the incoming code or an incoming agent being written in a specific language. In order for model-carrying code to be securely deployed, either proofs of code or digital signatures must be employed in order to ensure model soundness [138]. In this way, the limitations associated with proof-carrying code and identity-based mechanisms also impact upon the suitability and usefulness of model-carrying code.

Identity-based authorisation has limited value in an open environment, as was highlighted on examination of a set of identity-based mobile code and agent authorisation mechanisms. Firstly, a host may potentially need to authorise an unbounded number of entities, many of whom will be unknown. Secondly, such systems do not address the case where a known entity generates a malicious executable, or buggy code, which results in an end host system security vulnerability.

Finally, we examined authorisation mechanisms based upon the integrity of the incoming code or agent. Of the three mechanisms examined, tracing enables malicious activity to be detected rather than prevented, which is not ideal. There are currently no known algorithms which enable code obfuscation [91]. While at the time of writing, state appraisal functions represent the only mechanism by which the security threats, introduced by potentially malicious agent state information, can be thwarted, the implementation of such an approach requires a complex verification process to be completed by the end host, which is undesirable in a mobile environment, where devices may be limited in terms of processing power. Each of these mechanisms also assume/depend on the fact that the originator of the code or agent is trusted.

Based on these conclusions, therefore, we chose to develop a policy-based framework for the authorisation of mobile executables in a mobile environment, where a minimal set of checks need to be completed by the end device, and both the origin and/or the behaviour of an incoming executable can be taken into account when making an authorisation decision. In order to construct a framework of this nature, two fundamental elements had to be considered:

- The framework's underlying architecture; and
- The mechanisms required in order to express the policy statements and attribute certificates, needed in the implementation of the underlying architecture, and to support the necessary policy engine component functionality.

We described six architectural models which facilitate the authorisation of incoming mobile code and agents. We then analysed each model with respect to its security, and with regard to its suitability for implementation in a mobile environment. The results of this analysis are summarised in table 13.1.

Table 13.1: A summary of scenario description and analysis

	Description	**Authorisation**	**Security issues**	**Implementation**
1	Device manufacturers generate attribute certificate(s) for code authors (and agent creators) they trust. Incoming executables are signed by code authors (and agent creators).	Determined by code author (and potentially agent creator) identity/identities.	Identity-based authorisation is not suitable in an open environment. Device manufacturer is the sole point of trust. Dynamic agent state information cannot be protected.	Signature verification: Code author's (and potentially agent creator's) on executable; and Device manufacturer's on attribute certificate(s).

2	Incoming executables are signed by code authors.	Determined by code author identity.	Identity-based authorisation is not suitable in an open environment. Reasoning behind the incremental trust values associated with trusted third parties, the network operator and the device manufacturer? Static or dynamic agent state information cannot be protected.	Signature verification: Code author's on executable.
3	Executables are tested by a trusted third party, using for example proof-carrying code, model-carrying code, static analysis tools. An attribute certificate is then generated by the trusted third party for the executable.	Determined by executable behaviour (security relevant properties of the executable).	Issues with proof-carrying code: Formalism of proofs; Efficient proof verification; Provable properties? Proof size. Issues with model-carrying code: Not fully developed; Proof-carrying code or identity-based issues may hinder its deployment. Definition of the security relevant property set? Issues with tools: What is a suitable test-set? Test-sets specific to particular languages; False positives and negatives; Static or dynamic agent state information cannot be protected.	Signature verification: Trusted third party's on attribute certificate. Burden of testing pushed onto a trusted third party (away from the mobile host). Efficiency of testing process? (potentially eased by access control lists). There may be a difficulty in mapping an incoming executable to a set of execution permissions (if different trusted third parties consider different security properties relevant).

4	Executables are tested by a domain server with which the end host is affiliated. An attribute certificate is then generated by the domain server for the executable.	Determined by code author (and agent creator) identity and/or executable behaviour (security relevant properties).	Flexibility in security controls associated with executable. Static and dynamic agent state information can be protected.	Signature verification: Domain server's on attribute certificate. Burden of testing pushed onto the domain server (away from the mobile host).
5	Executables are executed by a domain server with which the end host is affiliated. If malicious activity is not detected, executables are signed and forwarded by domain server to the end host.	Determined by executable behaviour.	Must potentially test the executable with an unbounded number of input parameters to ensure its safety.	Signature verification: Domain server's on executable. Burden of testing pushed onto the domain server (away from the mobile host). Efficiency issues? (potentially eased by profiling). Emulation difficulties - growing number of device configurations.
6	Executable's signatures, certificates and state appraisal functions verified by domain server with which the mobile host is affiliated. If all checks are validated the mobile agent is signed and forwarded to the mobile host.	Determined by code author and agent creator identity and agent behaviour.	Concrete implementation of state appraisal functions – does such a function exist? The agent code author must be trusted.	Signature verification: Domain server's on incoming agent. Burden of testing pushed onto domain server (away from mobile host). Focus on mobile agents rather than mobile code and agents.

Based on this analysis, we compiled a set of requirements for the underlying architecture of a policy-based code and agent authorisation framework. In short, we concluded that:

- the framework should make minimal use of the end host's CPU processing power and the end host's storage for authorisation data structures;
- the underlying architecture should support mechanisms which provide assurances regarding the origin of the executable, executable code quality and the state of an agent;

- a policy engine, which is comprised of a policy administration point, a policy information point, an authentication point, a trust establishment module, a policy decision point, and a policy enforcement point, should be incorporated into each end host; and finally that
- an end host can specify, store and/or process policy statements and signed attribute certificates.

We then drafted a set of specific requirements by which the specification language and policy engine components defined as part of the KeyNote trust management system, the Ponder policy specification framework and SAML could be analysed. These requirements are summarised in table 13.2.

Table 13.2: Policy engine requirements

Requirements			**Scenario**					
			1	2	3	4	5	6
Policy statements	1	Authority for mobile agent execution can be delegated.	✓	✓				
	2	Executable signed by code author which is certified by device manufacturer can be authorised to execute under specified controls.	✓					
	3	Mobile agent signed by code author and agent creator which are both certified by device manufacturer can be authorised to execute under specified controls.	✓					
	4	Executable signed by particular entity can be authorised to execute in a predefined domain.		✓	✓		✓	✓
	5	Authority for executable attribute certificate generation can be delegated.			✓	✓		
	6	Executable whose attributes have been certified by a specified entity can be authorised to execute under specified controls.			✓	✓		
Authentication evidence	1	Digital signature on incoming executable.	✓	✓		✓	✓	✓
	2	Digital signature on entity attribute certificate.	✓	✓				

	3	Digital signature on executable attribute certificate.			✓	✓		
	4	Hash of incoming executable.	✓	✓	✓	✓	✓	✓
Attribute certificates	1	Signed attribute certificates.	✓	✓	✓	✓		
	2	Expression of entity identity in the attribute certificate.	✓	✓	✓			
	3	Delegation of authority to code authors and agent creators by means of an attribute certificate.	✓					
	4	Delegation of authority to trusted third parties by means of an attribute certificate.		✓				
	5	Expression of executable identity in the attribute certificate.			✓	✓		
	6	Expression of attributes.			✓	✓		
Compliance values	1	Boolean ordered compliance value set.	✓		✓	✓	✓	✓
	2	A more complex ordered compliance value set – containing predefined domain names.		✓	✓			
The PAP	1	A means of specifying, managing and organising security policies.	✓	✓	✓	✓	✓	✓
The PIP	1	Collection of access requestor information.	✓	✓	✓	✓	✓	✓
The AP	1	Verification of the digital signature on the incoming executable.	✓	✓				
	2	Verification of the digital signature on the entity attribute certificate.	✓	✓			✓	✓
	3	Verification of the digital signature on the executable attribute certificate.			✓	✓		
	4	Verification of the incoming executable's identity.	✓	✓	✓	✓	✓	✓
The TEM	1	Mapping the unknown authorisation requestor to a principal to which policies apply.	✓	✓	✓	✓	✓	✓
The PDP	1	Comparing all the information submitted as part of the authorisation request with the relevant policy statements on the mobile device, and responding with an authorisation decision from the ordered compliance value set.	✓	✓	✓	✓	✓	✓
The PEP	1	Enforcing the decisions of the PDP.	✓	✓	✓	✓	✓	✓

On examination of the three selected policy and attribute certificate specification languages, namely KeyNote, Ponder and SAML, and following an exploration of the functionality of their supporting policy engine components, we investigated whether each of the architectural models summarised in table 13.1 could be implemented using each system, thereby allowing us to make some conclusions about the suitability for each of the languages for use in our policy-

based framework. The results of this investigation are summarised in table 13.3: a tick denotes that a requirement can be met using the specified policy framework, a cross denotes that a requirement cannot be met, and a bullet denotes that the requirement can be partially met.

Table 13.3: Policy engine component analysis

Requirement	**Specification Language**		
	KeyNote	Ponder	SAML
Policy statement req 1	✓	✗	✗
Policy statement req 2	✓	•	✗
Policy statement req 3	✓	•	✗
Policy statement req 4	✓	•	✗
Policy statement req 5	✓	✗	✗
Policy statement req 6	✓	•	✗
Authentication evidence req 1	✗	✗	✗
Authentication evidence req 2	✓	✗	✓
Authentication evidence req 3	✗	✗	✓
Authentication evidence req 4	✗	✗	✗
Attribute certificate req 1	✓	✗	✓
Attribute certificate req 2	✓	✗	✓
Attribute certificate req 3	✓	✗	✓
Attribute certificate req 4	✓	✗	✓
Attribute certificate req 5	✗	✗	✓
Attribute certificate req 6	✗	✗	✓
Compliance values req 1	✓	•	✗
Compliance values req 2	✓	•	✗
PAP req 1	✓	✓	✗
PIP req 1	✗	✗	✓
AP req 1	✓	✗	✓
AP req 2	✗	✗	✗
AP req 3	✗	✗	✓
AP req 4	✗	✗	✗
TEM req 1	✓	✗	✗
PDP req 1	✓	✓	✗
PEP req 1	✗	✓	✗

Finally, we proposed a policy-based framework that synthesises techniques for establishing the origin, authenticity, safety and integrity of incoming mobile executables, and policy-specification and processing techniques, in order to provide a rich and flexible authorisation framework. It is most suitable for deployment in an environment where end hosts may be limited in terms of processing power or the checks they can complete. We assume the presence of a trusted domain server, which is responsible for a set of mobile devices. This trusted domain server intercepts and completes a pre-defined set of security checks on incoming executables destined for a mobile device for which it is responsible.

On completion of these security checks, the trusted domain server generates an executable attribute certificate before forwarding both the executable and certificate to the destination host. The incoming executable is then assigned to a domain defined on the mobile device based upon its associated attributes. A set of pre-defined executable permissions are specified for members of each domain.

SAML was chosen for assertion expression as it enables the transfer of signed assertions describing the attributes of an executable. The number or type of attributes that can be expressed is extensible. It is therefore possible for a trusted domain server to add new attributes under which an executable can be expressed at any stage. The identity of the executable may also be contained within a SAML attribute assertion. DTPL was chosen in order to enable an incoming executable to be mapped to a domain, i.e. in order to establish trust between the incoming executable and the mobile host. Finally, Ponder was chosen to specify the actions members of a particular domain, to which an incoming executable has been assigned, are authorised to perform. We chose to decouple trust establishment and the specification of execution permissions. Integrated solutions to trust establishment and access control are more complex. In conjunction with this, the use of an integrated solution may also impede the integration of the framework into existing systems, as it requires all policy statements to be written in the chosen language. Using separate trust establishment and access control mechanisms means that incoming executables may be mapped to groups/roles defined within the system about which policies have already been defined. It also implies that, instead of using Ponder, as is used in the proposed framework, this framework could be integrated into a platform which uses a different policy language for the expression of role-based or group-based platform security policies.

13.1.2 Part II: Mobile code protection

Part II focused on the authorisation of a mobile host upon which downloaded code would execute. Initially, we described the two standard mechanisms that have been defined by the DVB organisation in order to ensure that an end user can acquire broadcast services from a variety of service providers using proprietary conditional access systems to protect the content, namely simulcrypt and common interface. On examination of these standards however, it became apparent that, when applied to the mobile environment, these mechanisms could prove prohibitively costly and cumbersome. We then proposed the use of reconfigurable mobile receivers in order to overcome these limitations. Following this, the security threats relating to the secure delivery of a conditional access application and the secure storage and execution of the application on a mobile device were defined. The security services and mechanisms required to thwart the threats highlighted were also described. This work is summarised in table 13.4.

Following this, we described two protocols which support the secure download of a conditional access application to a mobile device. Both these protocols ensure that the conditional access application is protected against threats 1 to 5, described in table 13.4. This was demonstrated through a security analysis completed on both protocols.

We then described how the two protocols might be implemented on a selection of trusted computing architectures, namely:

- a platform containing TCG trusted platform components;
- a platform into which a version 1.2 compliant TPM and CRTM are integrated and an isolation layer deployed; and finally,

Table 13.4: A summary of the security threats, services and mechanisms pertaining to secure software download and execution

	Threat	**Service**	**Mechanism**
1	Unauthorised reading of the application code and data.	Confidentiality of the application code and data.	Symmetric or asymmetric encryption.
2	Unauthorised modification of the application code and data.	Integrity protection of the application code and data.	A message authentication code or digital signature.
3	Unknowingly communicating with an unknown and potentially malicious entity.	Entity authentication.	Entity authentication protocols; and platform attestation.
4	The inability to corroborate the source of the conditional access application.	Origin authentication.	The software provider's digital signature on either the (possibly encrypted) incoming application, or on keys used to protect the integrity and confidentiality of the incoming application.
5	Replay of communications.	Freshness.	Nonces or timestamps.
6	Unauthorised reading or modification of any cryptographic keys used in the provision of confidentiality and integrity protection to the conditional access application code and data.	Secure symmetric key generation.	Key generation in an isolated environment.
		Secure symmetric key transmission.	Asymmetric encryption and digital signatures.
		Secure symmetric key storage.	Protected storage on the host; or the mechanisms used to confidentiality and integrity-protect the symmetric key(s) whilst in transit.
		Prevention of unauthorised access to the symmetric key(s).	Protected storage on the host; or the symmetric key(s) may be bound to a particular hardware component, such as a secure (co-)processor, so that the symmetric key(s) can only be decrypted inside that particular hardware component.
7	Unauthorised reading or modification of the application code and data while it executes on the mobile host.	Confidentiality and integrity protection of the application code and data during execution.	Isolation mechanisms.

- given an NGSCB compliant platform, as described by Microsoft.

Following this, we analysed whether the implementation of both protocols on each of the trusted computing architectures enabled security services 6 and 7, as described in table 13.4, to be met.

In a TCG compliant platform security service 6 can be met. Problems may arise, however, in relation to the provision of security service 7. No mechanisms are defined by the TCG for partitioning a system into trusted and untrusted compartments. In order for a software provider to trust the execution environment in which the conditional access application will execute, the provider may require that the platform is in a controlled state, running, for example, a trusted OS, a download application, and a broadcast application, but nothing more. Essentially, the end host may be required to be a closed platform. If a TCG compliant platform were to remain open in this scenario, it would become very difficult for a software provider to verify the attestation statement generated by the end host, and also to evaluate whether a platform should be trusted for the secure download and execution of a conditional access application. If an isolation layer is integrated into a TCG compliant mobile device, the platform can be partitioned into both trusted and untrusted execution environments. This means that the conditional access application can be executed in an isolated environment which the mobile device has attested to, and the software provider has verified and evaluated as trusted for the secure download and execution of a conditional access application. It also implies that the platform configuration register verification which must be completed by the software provider can be simplified. Finally, it means that the platform can remain open and useable. Problems may arise, however, in relation to OS compatibility and direct memory access attacks. Issues surrounding device support and OS backward compatibility may be tackled through the extension of the platform

chipset and enhancement of the platform CPU, as described as both Microsoft's NGSCB and Intel's LaGrande initiatives.

Following this, we described two application download protocols proposed by the designers of XOM and AEGIS. Both protocols are based upon the assumption that the host device contains a hardened processor. Both protocols were then analysed against the security requirements described in table 13.4. These analyses revealed security shortcomings in both of these protocols.

There are two reasons why the shortcomings of the XOM and AEGIS protocols arise. Firstly, the AEGIS designers do not require that security services 3, 4 and 5, as described in table 13.4, are met by their download protocol. However, on examination of the generic requirement explicitly listed by the designers, i.e. to support the download and execution of copy and tamper-resistent software, security services 1, 2 and 6 and 7, described in table 13.4, must be met. Of these four security services, both the XOM and AEGIS download protocols meet security service 7, but only partially meet security services 1, 2 and 6, as described table 13.4. It appears that the second reason for the protocols' security shortcomings is due to the focus on ensuring that the architectures and download protocols support the copy and tamper-resistent execution of software rather than the copy and tamper-resistent download and execution of software. We subsequently proposed a series of enhancements to the protocols designed to address the identified shortcomings. While neither protocol meets all of the requirements for the secure transmission of A_C, the protocols can easily be modified to fulfil the additional requirements. The requirements surrounding the secure execution of A_C are met, however, and strong isolation of A_C is provided when executed on these hardened processors.

13.1.3 Part III: Remote code protection

Part II focused on the authorisation of remote code residing on a mobile device. We initially provided an overview of DRM, with particular focus on the OMA DRM standards. The rationale for DRM solutions were examined and the generic components of a DRM system outlined. The activities of the OMA were briefly introduced, followed by a description of the model to which the OMA DRM architecture applies. An overview of OMA DRM v1 and v2 was also provided. Following this, the lifecycle of an OMA DRM v2 agent was examined, and OMA DRM v2 agent installation and each protocol in the ROAP suite analysed in terms of the threats that may impact on devices on which OMA DRM v2 is not robustly implemented. We then accumulated a functional requirement set which must be met by the platform in order to thwart these threats and provide a robust implementation of OMA DRM v2. The requirements extracted were then utilised in order to examine which architectural components and functionality described within TCG version 1.2 specification set could be used to facilitate a robust implementation of OMA DRM v2.

This enabled us to construct a partial profile for a mobile TPM for the OMA DRM v2 use case, as illustrated in figure 13.5.

Table 13.5: TPM commands required for a robust implementation of OMA DRM v2

TPM version 1.2 command	Mobile device TPM
TPM_Init	required
TPM_Startup	required
TPM_SaveState	optional
TPM_SelfTestFull	required
TPM_ContinueSelfTest	required
TPM_GetTestResult	required
TPM_SetOwnerInstall	required
TPM_OwnerSetDisable	optional
TPM_PhysicalEnable	required
TPM_PhysicalDisable	required
TPM_PhysicalSetDeactivated	required

TPM_SetTempDeactivated	optional
TPM_SetOperatorAuth	optional
TPM_TakeOwnership	required
TPM_OwnerClear	optional
TPM_ForceClear	optional
TPM_DisableOwnerClear	optional
TPM_DisableForceClear	optional
TPM_GetCapability	required
TPM_SetCapability	optional
TPM_GetAuditDigest	optional
TPM_GetAuditDigestSigned	optional
TPM_SetOrdinalAuditStatus	optional
TPM_FieldUpgrade	optional
TPM_SetRedirection	optional
TPM_ResetLockValue	optional
TPM_Seal	required
TPM_Unseal	required
TPM_Unbind	optional
TPM_CreateWrapKey	required
TPM_LoadKey2	required
TPM_GetPubKey	optional
TPM_Sealx	optional
TPM_CreateMigrationBlob	optional
TPM_ConvertMigrationBlob	optional
TPM_AuthorizeMigrationKey	optional
TPM_MigrateKey	optional
TPM_CMK_SetRestrictions	optional
TPM_CMK_ApproveMA	optional
TPM_CMK_CreateKey	optional
TPM_CMK_CreateKey	optional
TPM_CMK_CreateTicket	optional
TPM_CMK_CreateBlob	optional
TPM_CMK_ConvertMigration	optional
TPM_CreateMaintenanceArchive	optional
TPM_LoadMaintenanceArchive	optional
TPM_KillMaintenanceFeature	optional
TPM_LoadManuMaintPub	optional
TPM_ReadManuMaintPub	optional
TPM_SHA1Start	optional
TPM_SHA1Update	optional
TPM_SHA1Complete	optional
TPM_SHA1CompleteExtend	optional
TPM_Sign	optional
TPM_GetRandom	required
TPM_StirRandom	required
TPM_CertifyKey	optional
TPM_CertifyKey2	optional
TPM_CreateEndorsementKeyPair	optional
TPM_CreateRevokableEK	optional
TPM_RevokeTrust	optional

TPM_ReadPubek	required
TPM_OwnerReadInternalPub	optional
TPM_MakeIdentity	required
TPM_ActivateIdentity	required
TPM_Extend	required
TPM_PCRRead	required
TPM_Quote	required
TPM_PCR_Reset	optional
TPM_Quote2	optional
TPM_ChangeAuth	optional
TPM_ChangeAuthOwner	optional
TPM_OIAP	required
TPM_OSAP	required
TPM_DSAP	optional
TPM_SetOwnerPointer	optional
TPM_Delegate_Manage	optional
TPM_Delegate_CreateKeyDelegation	optional
TPM_Delegate_CreateOwnerDelegation	optional
TPM_Delegate_LoadOwnerDelegation	optional
TPM_Delegate_ReadTable	optional
TPM_Delegate_UpdateVerification	optional
TPM_Delegate_VerifyDelegation	optional
TPM_NV_DefineSpec	optional
TPM_NV_WriteValue	optional
TPM_NV_WriteValueauth	optional
TPM_NV_ReadValue	optional
TPM_NV_ReadValueAuth	optional
TPM_KeyControlOwner	optional
TPM_SaveContext	optional
TPM_LoadContext	optional
TPM_FlushSpecific	required
TPM_GetTicks	optional
TPM_TickStampBlob	optional
TPM_EstablishTransport	optional
TPM_ExecuteTransport	optional
TPM_ReleaseTransportSigned	optional
TPM_CreateCounter	optional
TPM_IncrementCounter	optional
TPM_ReadCounter	optional
TPM_ReleaseCounter	optional
TPM_ReleaseCounterOwner	optional
TPM_DAA_Join	optional
TPM_DAA_Sign	optional

13.2 Future work

The issue of mobile code and agent authorisation considered in part I of this thesis has been widely discussed. At the current time, a large set of mobile code and agent authorisation mechanisms exist. While behaviour-based code and/or agent authorisation mechanisms such as proofs of code are promising in an open environment, further efforts are required in order to expand the range

of properties which can be currently captured by proofs.

The issue of agent security and host authorisation, however, has proved more difficult to solve. While a plethora of solutions tackling various aspects of this problem have been proposed [70,130,133,141], a summary of which can be found in [25,28], the issue would benefit from re-examination in light of the emergence of trusted computing technologies. We are currently examining this problem and have recently presented an efficient method by which trusted computing can be used to protect a mobile agent against malicious hosts [57].

The work completed in parts II and III focuses on the use of trusted computing technologies in a mobile environment. In part II we considered how secure software download can be supported using trusted computing functionality, while in part III we stipulated the subset of version 1.2 TPM commands required in a mobile TPM in order to enable a robust implementation of OMA DRM v2. We also identified architecture components and functionality not currently specified within the TCG version 1.2 specification set, but required for the implementation of a robust DRM solution on a trusted mobile platform, namely a secure boot mechanism and a mechanism which ensures that the integrity of the platform is maintained after boot.

While we proposed a list of TPM v1.2 functionality that might be required in a mobile TPM in order to support all trusted computing use cases, further investigation needs to be completed on key use cases to verify this hypothesis. This problem is to some extent being addressed within the OpenTC project[1], where a selection of use cases, namely OMA DRM v2, IMEI, SIMLock, secure software download, secure software use and secure wallet, are being described and analysed in order to determine the security and policy requirements for a trusted mobile computing platform.

[1] www.opentc.net

While secure boot is a concept that has been widely discussed, further work is required to enable run-time integrity verification such that the integrity of a platform can be validated periodically after boot.

Trusted mobile platform implementation represents another area for future development. Given the number of potential stakeholders on a mobile device, each of which needs access to TPM functionality and potentially, its own TPM, the deployment of soft or virtual TPMs would be advantageous. The secure implementation of a soft TPM is an open and interesting research topic.

Bibliography

[1] A. Abdul-Rahman and S. Hailes. A distributed trust model. In *Proceedings of the 1997 Workshop on New Security Paradigms*, pages 48–60, Langdale, Cumbria, United Kingdom, 23–26 September 1998. ACM Press, New York, USA.

[2] B. Albahari, P. Drayton, and B. Merrill. *C# Essentials*. O'Reilly, Sebastopol, California, USA, 2nd edition, March 2002.

[3] R. Anderson. Cryptography and competition policy - issues with 'trusted computing'. In *Proceedings of the 23rd Annual Symposium on Principles of Distributed Computing (PODC 2003)*, pages 3–10, St. John's, Newfoundland, Canada, 25–28 July 2003. ACM Press, New York, USA.

[4] W.A. Arbaugh, D.J. Farber, and J.M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy (S&P 1997)*, pages 65–71, Oakland, California, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California.

[5] B. Balacheff, L. Chen, S. Pearson, D. Plaquin, and G. Proudler. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall, Upper Saddle River, New Jersey, USA, 2003.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield. XEN and the art of virtualization. In

Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003), pages 164–177, Bolton Landing, New York, USA, 19–22 October 2003. ACM Press, New York, USA.

[7] M.F. Barrett. Towards an open trusted computing framework. Masters thesis, Department of Computer Science, The University of Auckland, New Zealand, February 2005.

[8] European Broadcasting Union (EBU) Project Group B/CA. Functional model of a conditional access system. EBU technical review, EBU, Geneva, Switzerland, October 1995.

[9] M. Bellare and C. Namprempre. Authenticated encryption: Relations amoung notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *Proccedings of Advances in Cryptology — ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security*, volume 1976 of *Lecture Notes in Computer Science (LNCS)*, pages 531–545, Kyoto, Japan, 3–7 December 2000. Springer–Verlag, Berlin–Heidelberg, Germany.

[10] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. Policy-driven binding to information resources in mobility-enabled scenarios. In M.S. Chen, P.K. Chrysanthis, M. Sloman, and A.B. Zaslavsky, editors, *Proceedings of the 4th International Conference on Mobile Data Management (MDM 2003)*, volume 2574 of *Lecture Notes in Computer Science (LNCS)*, pages 212–229, Melbourne, Australia, 21–24 January 2003. Springer–Verlag Berlin–Heidelberg, Germany.

[11] S. Berkovits, J.D. Guttman, and V. Swarup. Authentication for mobile agents. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419

of *Lecture Notes in Comptuer Science (LNCS)*, pages 114–136. Springer–Verlag, Berlin–Heidelberg, Germany, 1998.

[12] E. Bertino, P.A. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):191–233, August 2001.

[13] M. Blaze, J. Feigenbaum, J. Ioanndis, and A. Keromytis. The KeyNote trust management system version 2. RFC 2740, Internet Engineering Task Force (IETF), September 1999.

[14] M. Blaze, J. Feigenbaum, and A. Keromytis. KeyNote: Trust management for public key infrastructures. In W.S. Harbison and M. Roe, editors, *Proceedings of the 6th International Workshop on Security Protocols*, volume 1550 of *Lecture Notes in Computer Science (LNCS)*, pages 59–63, Cambridge, UK, 15–17 April 1998. Springer–Verlag, Berlin–Heidelberg, Germany.

[15] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralised trust management. In *Proceedings of the 17th IEEE Symposium on Security and Privacy*, pages 164–173, Oakland, California, USA, May 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[16] M. Blaze, J. Ioannidis, and A.D. Keromytis. Experience with the KeyNote trust management system: Applications and future directions. In P. Nixon and S. Terzis, editors, *Proceedings of the 1st International Conference on Trust Management (iTrust 2003)*, volume 2692 of *Lecture Notes in Computer Science (LNCS)*, pages 284–300, Heraclion, Greece, 28–30 May 2003. Springer–Verlag, Berlin–Heidelberg, Germany.

[17] P. Bonatti and P. Samarati. Regulating service access and information release on the web. In P. Samarati, editor, *Proceedings of the 7th ACM*

Conference on Computing and Communications Security, pages 134–143, Athens, Greece, 1–4 November 2000. ACM Press, New York, USA.

[18] A.B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd International Workshop on Software and Performance (WOSP 2000)*, pages 195–203, Ottawa, Ontario, Canada, 17–20 September 2000. ACM Press, New York, USA.

[19] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. *OpenPGP Message Format*. Internet Engineering Task Force (IETF), November 1998.

[20] J. Cappaert, B. Wyseur, and B. Preneel. Software security techniques. COSIC internal report, Computer Security and Industrial Cryptography (COSIC), Katholieke Universiteit Leuven, Leuven–Heverlee, Belgium, 2004.

[21] H. Castaneda. *New Studies in Deontic Logic: Norms, Actions and the Foundations of Ethics*, chapter The Paradoxes of Deontic Logic: The Simplest Solution to all of them in One Fell Swoop, pages 37–85. D. Reidel Publishing company, Dordrecht, Holland, 1981.

[22] CENELEC. Common interface specification for conditional access and other digital video broadcasting decoder applications. CENELEC Standard 50221, European Committee for Electrotechnical Standardization (CENELEC), Brussels, Belgium, February 1997.

[23] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, pages 235–244, Washington, District of Columbia, USA, 18–22 November 2002. ACM Press, New York, USA.

[24] Y. Chen, P. England, M. Peinado, and B. Willman. High assurance computing on open hardware architectures. Microsoft Technical report MSRTR–2003–20, Microsoft Corporation, March 2003.

[25] D.M. Chess. Security issues in mobile code systems. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Comptuer Science (LNCS)*, pages 1–14. Springer–Verlag, Berlin–Heidelberg, Germany, 1998.

[26] Y. Chu, J. Feigenbaum, B.A. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for web applications. *The World Wide Web Journal*, 2(3):127–139, 1997.

[27] P.C. Clark and L.J. Hoffman. BITS: a smartcard protected operating system. *Communications of the ACM*, 37(11):66–94, November 1994.

[28] J. Classens, B. Preneel, and J. Vandewalle. (How) can mobile agents do secure electonic transactions on untrusted hosts? – a survey of the security issues and the current solutions. *ACM Transactions on Internet Technology*, 3(1):28–48, 2003.

[29] CMLA. Client adopter agreement. Technical Report Revision 1.00-050708, The Content Management License Administrator Limited Liability Company (CMLA, LLC), August 2005.

[30] A. Corradi, N. Dulay, R. Montanari, and C. Stefan. Policy-driven management of agent systems. In M. Sloman, J. Lobo, and E. Lupu, editors, *Proceedings of the 3rd Workshop on Policies for Distributed Systems and Networks (POLICY 2001)*, volume 1995 of *Lecture Notes in Computer Science (LNCS)*, pages 214–229, Bristol, England, UK, 29–31 January 2001. Springer–Verlag, Berlin–Heidelberg, Germany.

[31] J.P. Cunard, K. Hill, and C. Barlas. Curent developments in the field of digital rights management. WIPO document SCCR/10/2, World Intellectual Property Organisation Standing Committee on Copyright and Related Rights (WIPO SCCR), Geneva, Switzerland, August 2003.

[32] F. Cuppens and C. Saurel. Specifying a security policy: A case study. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop (CSFW 1996)*, pages 123–134, Kenmare, Kerry, Ireland, 10–12 March 1996. IEEE Computer Society Press.

[33] D.J. Cutts. DVB conditional access. *IEE Electronics and Communications Engineering Journal*, 9(1):21–27, February 1997.

[34] N. Damianou, A.K. Bandara, M. Sloman, and E.C. Lupa. A survey of policy specification approaches. Research report, Department of Computing, Imperial College of Science Technology and Medicine, London, UK, 2002.

[35] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. Ponder: A language for specifying and managenent policies for distributed systems, the language specification. Research Report Version 2.3, Department of Computing, Imperial College of Science Technology and Medicine, London, UK, 2000.

[36] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In M. Sloman, J. Lobo, and E.C. Lupu, editors, *Proceedings of the 2nd International Workshop on Policies for Distributed Systems and Networks (POLICY 2001)*, volume 1995 of *Lecture Notes in Computer Science (LNCS)*, pages 18–38, Bristol, England, UK, 29–31 January 2001. Springer–Verlag, Berlin–Heidelberg, Germany.

[37] N.C. Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, London, UK, February 2002.

[38] A.W. Dent and C.J. Mitchell. *User's Guide to Cryptography and Standards*. Artech House, Boston, Massachusetts, USA, 2005.

[39] NTT DoCoMo, IBM, and Intel Corporation. Trusted mobile platform. Software Architecture Description TMP_SWAD_rev1_00_20040405, June 2004.

[40] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[41] J.G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S.W. Smith, and S. Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, October 2001.

[42] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 38–48, San Francisco, California, USA, 2–5 November 1998. ACM Press, New York, USA.

[43] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *IEEE Computer*, 36(7):55–62, July 2003.

[44] European Telecommunications Standards Institute (ETSI). Digital Video Broadcasting (DVB); Support for use of Scrambling and Conditional Access (CA) within Digital Broadcasting Systems. ETSI Technical Report ETR 289, European Telecommunications Standards Institute (ETSI), Sophia Antipolis, France, October 1996.

[45] European Telecommunications Standards Institute (ETSI). Digital Video Broadcasting (DVB): Head-End Implementation of DVB Simulcrypt. ETSI Standard TS 103 197 V1.3.1, European Telecommunications Standards Institute (ETSI), Sophia Antipolis, France, January 2003.

[46] S. Farrell and R. Housley. An internet attribute certificate profile for authorization. RFC 3281, Internet Engineering Task Force IETF, April 2002.

[47] J. Feghhi, J. Feghhi, and P. Williams. *Digital Certificates – Applied Internet Security.* Addison-Wesley-Longman, October 1998.

[48] *FIPS PUB 186-2, Digital Signature Standard (DSS).* Gaithersburg, Maryland, USA, January 2000.

[49] S.N. Foley, T.B. Quillinan, J.P. Morrison, D.A. Power, and J.J. Kennedy. Exploiting KeyNote in webcom: Architecture neutral glue for trust management. In *Proceedings of the 5th Nordic Workshop on Secure IT Systems (NORDSEC 2000)*, pages 101–119, Reykjavik, Iceland, 12–13 October 2000.

[50] Foundation for Intelligent Physical Agents (FIPA). FIPA agent management specification. Standard SC00023K, Foundation for Intelligent Physical Agents (FIPA), March 2004.

[51] W. Ford. *Computer Communications Security — Principles, Standard Protocols and Techniques.* Prentice-Hall, Upper Saddle River, New Jersey, USA, 1994.

[52] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In J.P. Müller, M.J. Wooldridge, and N.R. Jennings, editors, *Proceedings of the Intelligent Agents III, the 3rd*

International Workshop on Agent Theories, Architectures, and Languages (ATAL 1996), volume 1193 of *Lecture Notes in Computer Science (LNCS)*, pages 21–35, Budapest, Hungary, 12–13 August 1996. Springer–Verlag, Berlin–Heidelberg, Germany.

[53] E. Gallery. Mobile agent and mobile code authorisation in mobile systems: A policy-based authorisation framework. In *Proceedings of the 10th Wireless World Research Forum Meeting*, New York, USA, 27–28 October 2003. Wireless World Research Forum (WWRF).

[54] E. Gallery. A policy based authorisation framework for software download. In *Proceedings of the 2nd Software Defined Radio Forum Technical Conference (SDR 2003)*, Orlando, Florida, USA, 17–19 November 2003. Software Defined Radio Forum (SDRF).

[55] E. Gallery. Towards a policy framework for mobile agent authorisation in mobile systems. In *Proceedings of the 4th International Conference on 3G Mobile Communication Technologies (3G 2003)*, number 494 in IEE Conference Publication, pages 13–18, Savoy Place, London, UK, 25–27 June 2003. The Institute of Electrical Engineers (The IEE), Michael Faraday House, Six Hills Way, Stevenage, UK.

[56] E. Gallery. An overview of trusted computing technology. In C.J. Mitchell, editor, *Trusted Computing*, IEE Professional Applications of Computing Series 6, chapter 3, pages 29–114. The Institute of Electrical Engineers (IEE), London, UK, April 2005.

[57] E. Gallery and S. Balfe. Mobile agents and the deus ex machina. In *Workshop on Current and Emerging Research Issues in Computer Security (CERICS 2006)*, Royal Holloway, University of London, July 2006.

[58] E. Gallery and A. Tomlinson. Conditional access in mobile systems: Securing the application. In *Proceedings of the 1st International Conference on Distributed Frameworks for Multimedia Applications (DFMA 2005)*, pages 190–197, Besançon, France, 6–9 February 2005. IEEE Computer Society.

[59] E. Gallery and A. Tomlinson. Protection of downloadable software on SDR devices. In *Proceedings of the 4th Software Defined Radio Forum Technical Conference (SDR 2005)*, Orange County, California, USA, 14–18 November 2005. Software Defined Radio Forum (SDRF).

[60] E. Gallery and A. Tomlinson. Secure delivery of conditional access applications to mobile receivers. In C.J. Mitchell, editor, *Trusted Computing*, IEE Professional Applications of Computing Series 6, chapter 7, pages 195–238. The Institute of Electrical Engineers (IEE), London, UK, April 2005.

[61] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS support and applications for trusted computing. In *Proceedings of the 9th USENIX Workshop on Hot Topics on Operating Systems (HotOS-IX)*, pages 145–150, Kauai, Hawaii, USA, 18-21 May 2003. USENIX, The Advanced Computing Systems Association.

[62] A.K. Ghosh. *E-commerce Security; Weak Links, Best Defences*, chapter Deadly Content: The Client Side Vulnerabilities, pages 31–96. John Wiley and Sons, New York, USA, 1998.

[63] J.I. Glasgow, G.H. MacEwen, and P. Panangaden. A logic for reasoning about security. *ACM Transactions on Computer Systems (ACM TOCS)*, 10(3):226–264, August 1992.

[64] L. Gong. *Inside Java 2 Paltform Security: Architecture, API Design, and Implementation*. Addison-Wesley Longman Publishing Co. Inc., Boston, Massachusetts, USA, 2003.

[65] D. Grawrock. *The Intel Safer Computing Initiative*. Intel Press, Oregon, USA, March 2006.

[66] R. Gray, D. Kotz, S. Nog, D. Rus, and G. Cybenko. Mobile agents for mobile computing. Technical Report PCS-TR96 285, Dartmouth College, Hanover, New Hampshire, USA, May 1996.

[67] R.S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'agents: Security in multiple-language, mobile agent system. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science (LNCS)*, pages 154–187. Springer–Verlag, Berlin–Heidelberg, Germany, 1998.

[68] A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, and Y. Ravid. Access control meets PKI, or: Assigning roles to strangers. In *Proceedings of the 21st IEEE Syposium on Security and Privacy (S&P 2000)*, pages 2–14, Washington, District of Columbia, USA, May 2000. IEEE Computer Society.

[69] F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science (LNCS)*, pages 92–113. Springer–Verlag, Berlin Heidelberg, Germany, 1998.

[70] F. Hohl. Time limited blackbox security:protecting mobile agents from malicious hosts. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science (LNCS)*, pages 92–113. Springer–Verlag, Berlin–Heidelberg, Germany, 1998.

[71] IEEE. Standard specifications for public key cryptography. IEEE 1363 standards documents IEEE 1363-2000, IEEE Computer Society, August 2000.

[72] Intel. LaGrande technology architectural overview. Technical Report 252491-001, Intel Corporation, September 2003.

[73] J. Irwin and T. Wright. Digital rights management. Vodafone internal newsletter, Vodafone, Newbury, England, UK, August 2004.

[74] *ISO/IEC 9594-8, Information Technology — Open Systems Interconnection — The Directory: Public-Key and Attribute Certificate Frameworks*. International Organization for Standardisation, Geneva, Switzerland, 2005.

[75] *ISO/IEC 11770-1, Information Technology — Security techniques — Key management —Part 1: Framework*. International Organization for Standardisation, Geneva, Switzerland, 1996.

[76] *ISO/IEC 13888-1. Information technology — Security techniques — Non-repudiation — Part 1: General*. International Organization for Standardisation, Geneva, Switzerland, 2004. 2nd edition.

[77] *ISO/IEC 14888-1. Information technology — Security techniques — Data signatures with appendix — Part 1: General*. International Organization for Standardisation, Geneva, Switzerland, 1998.

[78] *ISO/IEC 14888-2. Information technology — Security techniques — Data signatures with appendix — Part 2: Identity-based mechanisms*. International Organization for Standardisation, Geneva, Switzerland, 1999.

[79] *ISO/IEC 14888-3. Information technology — Security techniques — Data signatures with appendix — Part 3: Certificate-based mechanisms.* International Organization for Standardisation, Geneva, Switzerland, 1998.

[80] *ISO/IEC 7498-2 / ITU-T X.800, Data Communication Networks: Open System Interconnection (OSI); Security, Structure and Applications — Security Architecture for Open Systems Interconnection for CCITT Applications.* International Organization for Standardisation, Geneva, Switzerland, 1991.

[81] *ISO/IEC 9594-8, Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks.* International Organization for Standardisation, Geneva, Switzerland, 2001.

[82] *ISO/IEC 9797-1. Information technology - Security techniques — Message Authentication Codes (MACs) — Part 1: Mechanisms using a block cipher.* International Organization for Standardisation, Geneva, Switzerland, 1999.

[83] *ISO/IEC 9797-2. Information technology — Security techniques — Message Authentication Codes (MACs) — Part 2: Mechanisms using a hash-function.* International Organization for Standardisation, Geneva, Switzerland, 2002.

[84] *ISO/IEC 9798-1 Information technology — Security techniques — Entity authentication — Part 1: General.* International Organization for Standardisation, Geneva, Switzerland, 1997. 2nd edition.

[85] *ISO/IEC 9798-3 Information technology — Security techniques — Entity authentication mechanisms — Part 3: Mechanisms using digital signature techniques.* International Organization for Standardisation, Geneva, Switzerland, 1998. 2nd edition.

[86] *ISO/IEC 9798-4, Information technology — Security techniques — Entity authentication — Part 4: Mechanisms using a cryptographic check function.* International Organization for Standardisation, Geneva, Switzerland, 1999. 2nd edition.

[87] *ISO/IEC 9798-5, Information technology — Security techniques — Entity authentication — Part 5: Mechanisms Using Zero-Knowledge Techniques.* International Organization for Standardisation, Geneva, Switzerland, 2004.

[88] N. Itoi, W.A. Arbaugh, S.J. Pollack, and D.M. Reeves. Personal secure booting. In *Proceedings of the 6th Australasian Conference on Information Security and Privacy ACISP 2001*, volume 2119 of *Lecture Notes In Computer Science (LNCS)*, pages 130–141, Sydney, Australia, 11–13 July 2001. Springer–Verlag, London, UK.

[89] *ITU-T Recommendation X.509, Information technology — Open Systems Interconnection — The Directory: Public-key and Attribute Certificate Frameworks.* International Organization for Standardisation, Geneva, Switzerland, 2000. 4th edition.

[90] S. Jajodia, P. Samarati, and V.S. Subrahmanian. A logical language for expressing authorisations. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 1997)*, pages 31–42, Oakland, CA, USA, 4–7 May 1997. IEEE Computer Society, Washington, District of Columbia, USA.

[91] W. Jansen and T. Karygiannis. Mobile agents and security. NIST Special Publication 800-19, National Institute of Standards and Technology (NIST), Computer Security Division, Gaithersburg, Maryland, USA, 1999.

[92] W. Johnston, S. Mudumbai, and M. Thompson. Authorization and attribute certificates for widely distributed access control. In *Proceedings of IEEE 7th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 1998)*, pages 340–345, Palo Alto, California, USA, 17–19 June 1998. IEEE Computer Society, Washington, District of Columbia, USA.

[93] B. Kaliski and J. Staddon. PKCS #1: RSA cryptographic specifications – version 2. RFC 2437, Internet Engineering Task Force (IETF), October 1999.

[94] J.A. Knottenbelt. Policies for agent systems. Masters thesis, Imperial College of Science, Technology and Medicine, London, UK, June 2001.

[95] H. Krawczyk, M. Bellare, and R. Canetti. HMAC – keyed hashing for message authentication. RFC 2104, Internet Engineering Task Force (IETF), February 1997.

[96] B. Lampson, M. Abadi, and M. Burrows. Authentication in distributed systems: Theory and practice. *ACM transactions on computer*, 10(4):265–310, November 1992.

[97] J. Lettice. Bad publicity: Clashes trigger MS Palladium name change. Press pass – information for jornalists, The Register, 27th January 2003.

[98] D. Lie. *Architectural Support for Copy and Tamper Resistant Software.* Phd thesis, Department of Electrical Engineering, Stanford University, Stanford, California, USA, December 2003.

[99] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural*

Support for Programming Languages and Operating Systems (ASPLOS-IX), pages 169–177, Cambridge, Massachusetts, USA, 12–15 November 2000. ACM Press, New York, USA.

[100] V.B. Livshits and M.S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *The 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE–11)*, pages 317–326, Helsinki, Finland, 1–5 September 2003. ACM Press, New York, USA.

[101] A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*, volume 6 of *Discrete Mathematics and its Applications*. CRC Press, Boca Raton, Florida, USA, 1997.

[102] R.C. Merkle. Protocols for public key cryptography. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 122–134, Oakland, California, USA, April 1980. IEEE Computer Society Press.

[103] Z. Miklos. A decentralised authorisation mechanism for e-business applications. In *Proceedings of the 13th International Workshop on Database and Expert Systems Applications (DEXA 2002) - International Workshop on Trust and Privacy in Digital Business - TrustBus*, pages 446–450, Aix-en-Provence, France, September 2002. IEEE Computer Society, Washington, District of Columbia, USA.

[104] Chris Mitchell, editor. *Trusted Computing*. IEE Professional Applications of Computing Series 6. The Institute of Electrical Engineers (IEE), London, UK, April 2005.

[105] R. Montanari, G. Tonti, and C. Stefanelli. Programming agent mobility. In M. Klusch, S. Ossowski, and O. Shehory, editors, *Proceedings of the 6th*

International Workshop on Coorperative Information Agents - Intelligent Agents for the Internet and Web (CIA 2002), volume 2446 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 287–296, Madrid, Spain, 18–20 September 2002. Springer–Verlag, Berlin–Heidelberg, Germany.

[106] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. Internet X.509 public key infrastructure, online certificate status protocol – OCSP. RFC 2560, Internet Engineering Task Force (IETF), June 1999.

[107] G.C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture notes in conputer science (LNCS)*, pages 61–91. Spring–Verlag, Berlin–Heidelberg, Germany, 1998.

[108] NIST. Security requirements for cryptographic modules. Federal Information Processing Standards Publication FIPS PUB 140–1, National Institute of Standards and Technology (NIST), January 1994.

[109] NIST. Security hash standard. Federal Information Processing Standards Publication FIPS PUB 180–1, National Institute of Standards and Technology (NIST), April 1997.

[110] NIST. Common criteria of information technology security evaluation. Technical Report Version 2.1, National Institute of Standards and Technology (NIST), August 1999.

[111] H.S. Nwana and D.T. Ndumu. An introduction to agent technology. In H.S. Nwana and N. Azarmi, editors, *Software Agents and Soft Computing: Towards Enhancing Machine Intelligence*, number 1198 in Lecture notes in Artificial Intelligence (LNAI), pages 3–26. Springer–Verlag, Berlin–Heidelberg, Germany, 1997.

[112] OASIS. Assertion and protocol for the OASIS Security Assertion Markup Language (SAML) version 1.0. OASIS Standard Document oasis-sstc-saml-core-1.0, OASIS, 5 November 2002.

[113] OASIS. Bindings and profiles for the OASIS Security Assertion Markup Language (SAML) version 1.0. OASIS Standard Document oasis-sstc-saml-bindings-1.0, OASIS, 5 November 2002.

[114] OASIS. Assertion and protocol for the OASIS Security Assertion Markup Language (SAML) version 1.1. OASIS Standard Document oasis-sstc-saml-core-1.1, OASIS, 2 September 2003.

[115] OASIS. Bindings and profiles for the OASIS Security Assertion Markup Language (SAML) version 1.1. OASIS Standard Document oasis-sstc-saml-bindings-1.1, OASIS, 2 September 2003.

[116] OASIS. Assertion and protocol for the OASIS Security Assertion Markup Language (SAML) version 2.0. OASIS Standard Document saml-core-2.0-os, OASIS, 15 March 2005.

[117] OASIS. Bindings for the OASIS Security Assertion Markup Language (SAML) version 2.0. OASIS Standard Docuement saml-bindings-2.0-os, OASIS, 15 March 2005.

[118] OASIS. Profiles for the OASIS Security Assertion Markup Language (SAML) version 2.0. OASIS Standard Docuement saml-profiles-2.0-os, OASIS, 15 March 2005.

[119] OMA. Digital Rights Management v1.0. Technical Specification OMA-Download-DRM-V1_0-20040615-A, The Open Mobile Alliance (OMA), June 2004.

[120] OMA. DRM architecture v2.0. Technical Specification OMA-DRM-ARCH-V2_0-2004071515-C, The Open Mobile Alliance (OMA), July 2004.

[121] OMA. Drm architetcure specification v1.0. Technical Specification OMA-Download-ARCH-V1_0-20040625-A, The Open Mobile Alliance (OMA), June 2004.

[122] OMA. DRM specification v2.0. Technical Specification OMA-DRM-DRM-V2_0-20040716-C, The Open Mobile Alliance (OMA), July 2004.

[123] OMA. OMA DRM V1.0 approved enabler specification. Technical Specification OMA-DRM-V1_0-20040625-A, The Open Mobile Alliance (OMA), June 2004.

[124] OMA. OMA DRM V2.0 approved enabler specification. Technical Specification OMA-ERP-DRM-V2_0-20060303-A, The Open Mobile Alliance (OMA), July 2004.

[125] J.K. Ousterhout, J.Y. Levy, and B. B. Welsh. The safe TCL security model. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science (LNCS)*, pages 217–235. Springer–Verlag, Berlin–Heidelberg, Germany, 1998.

[126] M. Peinado, Y. Chen, P. England, and J. Manferdelli. NGSCB: A trusted open system. In H. Wang, J. Pieprzyk, and V. Varadharajan, editors, *Proceedings of 9th Australasian Conference on Information Security and Privacy, ACISP 2004*, volume 3108 of *Lecture Notes in Computer Science (LNCS)*, pages 86–97, Sydney, Austrailia, 13–15 July 2004. Springer–Verlag, Belin–Heidelberg, Germany.

[127] B. Pfitzmann, J. Riordan, C. Stuble, M. Waidner, and A. Weber. The PERSEUS system architecture. Technical Report RZ 3335 (#93381), IBM Research Division, Zurich Laboratory, April 2001.

[128] C.P. Pfleeger. *Security in Computing.* Prentice Hall, Upper Saddle River, New Jersey, USA, 2nd edition, 1997.

[129] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: An access control language for security policies with complex constraints. In *Proceedings of Network and Distributed System Security (NDSS '01)*, pages 89–107, San Diego, California, USA, 7–9 February 2001. The Internet Society.

[130] J. Riordan and Bruce Schneier. Environmental key generation towards clueless agents. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Comptuer Science (LNCS)*, pages 15–24. Springer–Verlag, Berlin–Heidelberg, Germany, 1998.

[131] K. Rothermel and M. Schwehm. Mobile agents. In A. Kent and J.G. Williams, editors, *Encyclopedia for Computer Science and Technology*, volume 40, pages 155–176. M. Dekker Inc., New York, USA, 1999.

[132] A.R. Sadeghi and C. Stuble. Taming "Trusted Platforms" by Operating System Design. In K. Chae and M. Yung, editors, *Proceedings of Information Security Applications, 4th International Workshop, (WISA 2003)*, volume 2908 of *Lecture Notes in Computer Science (LNCS)*, Jeju Island, Korea, 25–27 August 2003. Springer-Verlag, Berlin–Heidelberg, Germany.

[133] T. Sander and C.F. Tschudin. Protecting mobile agents against malicious hosts. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Comptuer Science (LNCS)*, pages 44–60. Springer–Verlag, Berlin–Heidelberg, Germany, 1998.

[134] NHK Science and Technical Research Laboratories. Scrambling (conditional access system). NHK Science and Technical Research Laboratories Bulletin 12, Tokyo, Japan, Autumn 2002.

[135] Software Defined Radio Forum (SDRF). Security considerations for operational software for software defined radio devices in a commercial wireless domain. SDRF Archived Approved Document 2004-A0010, 27 October 2004.

[136] K. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu. Requirements for policy languages for trust negotiation. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, pages 68–79, Monterey, California, USA, 5–7 June 2002. IEEE Computer Society, Washington, District of Columbia, USA.

[137] K.E. Seamons and W. Winborough. Internet credential acceptance policies. In M. Falaschi, M. Navarro, and A. Policriti, editors, *Joint Conference on Declarative Programming (APPIA-GULP-PRODE 1997)*, pages 415–432, Grado, Italy, 16–19 June 1997.

[138] R. Sekar, C.R. Ranalrishnan, I.V. Ramakrishnan, and S.A. Smolka. Model Carrying Code (MCC): A new paradigm for mobile code security. In *New Security Paradigms Workshop (NSPW'01)*, pages 23–30, Cloudcroft, New Mexico, USA, 10–13 September 2001. ACM Press, New York, USA.

[139] Bilal Siddiqui. Web services security. *XML.com*, 4 March 2003.

[140] M. Sihvonen. CC/PP negotiation of a mobile station in mexe service environment. In *International Conference on Information Systems Technology and its Applications (ISTA 2001)*, pages 185–198, St. Augustin, Germany, 2001. Gesellschaft fuer Mathematik und Datenverarbeitung.

[141] D. Singelée and B. Preneel. Secure e-commerce using mobile agents on untrusted hosts. COSIC internal report, Computer Security and Industrial Cryptography (COSIC), Katholieke Universiteit Leuven, Leuven–Heverlee, Belgium, 2004.

[142] W. Stallings. *Cryptography and Network Security, Principles and Practices*. Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 1999.

[143] E. Suh, D. Clarke, B. Gassend, M. van Dyke, and S. Devadas. The AEGIS processor architecture for tamper–evident and tamper-resistant processing. In *17th Annual ACM International Conference on Supercomputing (ICS'03)*, pages 160–171, San Francisco, California, USA, 23–26 June 2003. ACM Press, New York, USA.

[144] E. Suh, C.W. 'O Donnell, I. Sachdev, and S. Devadas. Design and implementation of the AEGIS secure processor using physical random functions. *ACM SIGARCH Computer Architecture News*, 33(2):25–36, 2005.

[145] J. Tardo and L. Valente. Mobile agent security and telescript. In *41st International IEEE Computer Society International Conference: Technologies for the Information Superhighway (COMPCON 1996)*, pages 58–63, Santa Clara, California, USA, 25–28 February 1996. IEEE Computer Society Press.

[146] TCG. TCPA Main Specification. TCG Specification Version 1.1b, The Trusted Computing Group (TCG), Portland, Oregon, USA, February 2002.

[147] TCG. Main specification changes. TCG Specification Version 1.2, The Trusted Computing Group (TCG), Portland, Oregon, USA, October 2003.

[148] TCG. TCG Software Stack (TSS) Specification. TCG Specification Version 1.1, The Trusted Computing Group (TCG), Portland, Oregon, USA, August 2003.

[149] TCG. TCG Specification Architecture Overview. TCG Specification Version 1.2, The Trusted Computing Group (TCG), Portland, Oregon, USA, April 2003.

[150] TCG. TPM Main, Part 1 Design Principles. TCG Specification Version 1.2 Revision 62, The Trusted Computing Group (TCG), Portland, Oregon, USA, October 2003.

[151] TCG. TPM Main, Part 2 TPM Data Structures. TCG Specification Version 1.2 Revision 62, The Trusted Computing Group (TCG), Portland, Oregon, USA, October 2003.

[152] TCG. TPM Main, Part 3 Commands. TCG Specification Version 1.2 Revision 62, The Trusted Computing Group (TCG), Portland, Oregon, USA, October 2003.

[153] TCG. TCG PC client specific implementation specification for conventional BIOS. TCG specification Version 1.2 Final, The Trusted Computing Group (TCG), Portland, Oregon, USA, July 2005.

[154] TCG. TCG Software Stack (TSS) Specification. TCG Specification Version 1.2, The Trusted Computing Group (TCG), Portland, Oregon, USA, 2005.

[155] TCG. TCG Work Group Charter Summary. TCG Specification, The Trusted Computing Group, Portland, OR, USA, 2005.

[156] TCG. TPM Main, Part 1 Design Principles. TCG Specification Version 1.2 Level 2 Revision 85, The Trusted Computing Group (TCG), Portland, Oregon, USA, February 2005.

[157] TCG. TPM Main, Part 2 TPM Data Structures. TCG Specification Version 1.2 Level 2 Revision 85, The Trusted Computing Group (TCG), Portland, Oregon, USA, February 2005.

[158] TCG. TPM Main, Part 3 Commands. TCG Specification Version 1.2 Level 2 Revision 85, The Trusted Computing Group (TCG), Portland, Oregon, USA, February 2005.

[159] TCG MPWG. Use Case Scenarios. TCG Specification Version 2.7, The Trusted Computing Group, Mobile Phone Working Group, Portland, Oregon, USA, September 2005.

[160] W. Tuttlebee, D. Babb, J. Irvine, G. Martinez, and K. Worrall. Broadcasting and mobile telecommunications: Interworking — not convergence. *European Broadcasting Union (EBU) Technical Review*, 293:1–11, January 2003.

[161] J. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. Technical Report CMU-CS-91-140R, Carnigie Mellon University, Pittsburgh, Pennsylvania, USA, May 1991.

[162] G. Vigna. Cryptographic traces for mobile agents. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture notes in computer science (LNCS)*, pages 137–153. Springer-Verlag, Berlin Heidelberg, Germany, 1998.

[163] D. Volpano and G. Smith. Language issues in mobile program security. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lec-*

ture Motes in Computer Science (LNCS), pages 25–43. Springer–Verlag, Berlin–Heidelberg, Germany, 1998.

[164] G.H. von Wright. Deontic logic. *Mind*, 60:1–15, 1951.

[165] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *The 7th Annual Symposium on Network and Distributed System Security Symposium (NDSS 2000)*, pages 2–4, San Diego, California, USA, February 2000. The Internet Society.

[166] R. Walsh. Q&A: Microsoft seeks industry wide collaboration for 'Palladium' initiative. Press pass – information for journalists, Microsoft, 1 July 2002.

[167] M. Weber, V. Shah, and C. Ren. A case study in detecting software security vulnerabilities using constraint optimization. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, pages 3–13, Florence, Italy, 10 November 2001. IEEE Computer Society.

[168] J. Wilander. Modeling and visualizing security properties of code using dependence graphs. In L. Blankers, editor, *The 5th Conference on Software Engineering Research and Practice in Sweden (SERPS 2005)*, pages 65–74, Västerås, Sweden, 20–21 October 2005. Malardalen University Press.

[169] Philip R. Zimmermann. *The Official PGP User's Guide*. MIT Press, Boston, Massachusetts, USA, 1995.

Appendix A

The TCG specification set

Trusted computing has recently emerged as one of the most significant new information security technologies. A trusted platform is one that behaves as expected for a particular purpose. Through the incorporation of trusted functionality into a computing platform, which offers services such as: cryptographic capabilities; platform integrity measurement, storage and reporting; the creation of trusted identities; secure storage; and platform attestation, it may be transformed into a trusted platform, as defined above. This chapter examines the specifications for trusted platform functionality produced by the TCG. NGSCB and LaGrande Technology are also briefly examined.

A.1 Introduction

This appendix provides an overview of the work of the TCG, with particular focus on the TPM version 1.2 specification set. Section A.2 details the notation used throughout this appendix. Section A.3 provides a brief history of the TCG and an overview of the TCG organisational structure. Section A.4 defines what is meant by a trusted platform, and section A.5 examines the entities necessary for the endorsement and use of a trusted platform. Section A.6 describes the three fundamental components of a TCG-defined trusted platform.

Section A.7 examines the capabilities of a v1.2 compliant TPM. Section A.8 describes how a TPM can be initialised. Section A.9 examines how a TPM can be enabled and activated. It also explains how an entity can take ownership of a TPM and, finally, how a TPM can be cleared. Section A.10 describes the set of credentials which each trusted platform should have associated with it. The concept of TPM identity is also explored in this section. How a trusted platform's configuration can be measured and reported is described in section A.11, and section A.12 explores the concept of locality. Section A.13 explores the protected storage functionality of a TPM.

In section A.14 transport security is explored. This feature enables the establishment of a secure channel between a TPM and secure processes, which offers confidentiality and integrity protection of commands sent to the TPM. The TPM's monotonic counter is described in section A.15. Section A.16 describes the TPM authorisation framework. The TPM's context manager is described in section A.17. Delegation functionality is explained in section A.18 and the TPM time-stamping capabilities are explored in A.19. Section A.20 describes the TPM migration mechanisms, which are used for the backup and cloning of migratable TPM-protected key objects. Sections A.21 and A.22 describe the

maintenance of the TPM and the TPM's audit functionality, respectively.

In section A.23, Microsoft's Next Generation Secure Computing Base, formerly known as Palladium, is investigated. In contrast to the TCG specification set, however, little technical information on NGSCB has yet been published.

Section A.24 explores Intel's LaGrande Technology initiative, which has led to the production of a set of enhanced hardware components designed to evolve the Intel Architecture–32 bit platform into a trusted platform.

A.2 Notation

The following notation is used in the remainder of the appendix.

SHA-1(Z)	denotes the SHA-1 hash of data Z.
$HMAC_K(Z)$	denotes a keyed-hash message authentication code on data Z using key K,
PCR-i	denotes the ith platform configuration register of the TPM, for i=0,1.....,16.

A.3 The TCG

Trusted computing platform specifications were first developed by the Trusted Computing Platform Alliance (TCPA), which was founded in January 1999. After the release of the initial draft specification in October 1999, the original alliance, which included HP, IBM, Intel and Microsoft, invited other companies to become involved. By 2002, over 150 companies had joined and the specifications had become an open industry standard [5]. In April 2003 it was announced that the TCPA was to be superseded by the TCG, which adopted the TCPA specifications with the intention of expanding and developing them.

The TCG is made up of five fundamental organisational components: the board of directors, the market working group, the advisory council, the administration and the technical committee. The technical committee advises to the

board by monitoring technical work groups for consistency and interoperability of technical specifications and initiatives [155]. In turn, following guidance from the board, the technical committee develops the agenda for the technical work groups [155]. There are currently ten active technical working groups, whose primary responsibilities are as follows [155].

- The trusted platform module work group is accountable for specifying how the TPM architecture, as described by the technical committee, can be implemented.
- The TCG software stack work group is responsible for the provision of a set of APIs for application vendors wishing to utilise TPM functionality.
- The mobile phone work group is currently adopting the TCG concepts for mobile devices.
- The peripheral work group is examining the trust-related properties of peripherals, where the trustworthiness of a computing platform is dependent, at least in part, on the trustworthiness of peripheral devices connected to that platform.
- The server-specific work group is responsible for the provision of definitions, specifications, guidelines and technical requirements essential for the implementation of TCG technology in servers.
- The storage systems work group focuses on standards for security services on dedicated storage systems.
- The conformance work group will generate the necessary specifications and documents required for the definition of protection profiles and other evaluation criteria.

- The PC client work group is accountable for the provision of common functionality, interfaces and a minimum set of security and privacy requirements for a PC client which uses TCG components to establish its root of trust.

- The infrastructure work group is responsible for the integration of the TCG platform specific specifications into internet and infrastructure technologies. There is also a sub-group of the infrastructure work group, called the trusted network connect sub-group, developing "an open solution architecture which enables network operators to enforce policies regarding endpoint integrity when granting access to network infrastructure" [155].

- The hard copy work group is accountable for the definition of a minimum set of functional, interface and privacy requirements for hardcopy components that use TCG components to establish their root of trust.

A.4 A trusted platform

A platform, as defined by Balacheff et al. [5], is any computing device, e.g. PC, server, mobile phone or appliance, capable of computing and communicating electronically with other platforms. A trusted platform (TP) is defined as a computing platform with a trusted component that is used to create a foundation of trust for software processes [5]. This 'trusted component', as defined by the TCG, incorporates the core root of trust for measurement and the trusted platform module.

We next explore the fundamental entities, components, mechanisms, and protocols that constitute the trusted computing group's trusted platform, where core reference material consists of [5, 148–152].

A.5 Entities involved

The critical entities involved in the endorsement and use of a trusted platform are described in this section. From the end user point of view, the main entities include the following.

- The TPM owner has full control over the TP's TPM.
- A TP user (who may differ from the TP owner).
- A challenger who needs to decide whether or not a particular TP can be trusted for a particular purpose.
- A TPM operator who is permitted to temporarily deactivate the TPM, see section A.9.3.

We also consider all entities involved in the establishment of trust in the platform, i.e. attestation entities.

- The trusted platform module entity (TPME) attests to the fact that a TPM is genuine by digitally signing an endorsement credential, see section A.10, which contains the public endorsement key from the endorsement key pair, see section A.7.4, associated with the TPM. The TPME is likely to be the TPM manufacturer.
- The conformance entity (CE) guarantees, through the generation of conformance credentials, see section A.10, that the design and implementation of the TPM and trusted building blocks (TBB) within a trusted platform meet established evaluation guidelines. The TBB is defined in [149] as the parts of the roots of trust, see section A.6.1, that do not have shielded locations or protected capabilities, see section A.7.1, for example the RTM or TPM initialisation functions.

- The platform entity (PE) offers assurance in the form of a platform credential, see section A.10, that a particular platform is an instantiation of a TP design as described in conformance credentials, and that the platform's TPM is indeed genuine. The PE may be the original equipment manufacturer (OEM).

- The validation entity (VE) certifies integrity measurements, i.e. measured values and measurement digests, see section A.11, which correspond to correctly functioning or trustworthy platform components, for example embedded data or program code, in the form of validation certificates. The VE may be a software component supplier. These validation certificates can be used by a challenger wishing to evaluate the state of a challenged TP based on signed integrity metrics provided to it.

- A privacy-certification authority is responsible for the generation of attestation identity credentials, see section A.10, which confirm that particular identities (public attestation identity keys) belong to a genuine TP.

- DAA issuers are responsible for the creation of DAA credentials for particular TPMs, which are generated as a result of an interaction between the DAA issuer and the TPM.

- DAA verifiers use DAA credentials to decide whether or not a TPM is indeed genuine, while preserving the anonymity of the TPM.

Other important entities include intermediaries, which may be utilised in either the migration or maintenance procedures.

A.6 The trusted platform subsystem

The trusted platform subsystem (TPS) is composed of three fundamental elements:

- The RTM;
- The TPM, which is the RTS and the RTR; and
- The TSS, which encompasses the software on the platform that supports the platform's TPM.

A.6.1 Roots of trust

The RTM, the RTS and the RTR are defined by the TCG as the required roots of trust for a TP.

A.6.1.1 The RTM

The RTM is a computing engine which accurately generates at least one integrity measurement event representing a software component running on the platform [5]. The measurement digest is then recorded to a PCR in the TPM, see section A.11.1, and details of the measuring process, namely a record of what was measured, is then recorded to the stored measurement log (SML) outside the TPM, see section A.11.3.

For the foreseeable future, it is envisaged that the RTM will be integrated into the normal computing engine of the platform with minimum protection [5], where the provision of additional BIOS boot block or BIOS instructions (the CRTM) cause the main platform processor to function as the RTM. Ideally, however, for the highest level of security, the CRTM would be part of the TPM [5].

A.6.1.2 The RTS and RTR

The TPM incorporates the RTS and the RTR, where:

- The RTS is a collection of capabilities which must be trusted if storage of data inside a platform is to be trusted. The RTS provides integrity and confidentiality protection to data used by the TPM but that is stored externally. It also provides a mechanism to ensure that the release of certain data only occurs in a specific environment.
- The RTR is a collection of capabilities that must be trusted if reported integrity metrics, which are representative of the platform state, are to be trusted [150]. It is responsible for establishing platform identities, reporting platform configurations, protecting reported values and establishing a context for attesting to reported values [156].

A.6.2 The TSS

The TCG software stack [148] is the software on the platform which supports the TPM. The challenger must determine whether TSS functions can be trusted by examining a challenged platform's integrity metrics. The TSS architecture consists of a number of software modules, which provide fundamental resources to support the TPM (see also [148]).

A.6.2.1 The TPM Device Driver

The TPM Device Driver (TDD) is a kernel mode component which is TPM and OS specific, and is likely to be provided by the TPM manufacturer or vendor. It contains code that has an understanding of TPM behaviour. Because user mode executables cannot directly access kernel mode executables, the manufacturer must also provide the TCG Device Driver Library (TDDL) which provides a

user mode interface to the TPM.

The standard interface to the TDDL is called the TDDL interface (Tddli), which facilitates the transition between user mode and kernel mode. This Tddli is defined so that all TPMs will support the same interface.

The TDD receives byte streams from the TDDL, sends them to the TPM, and returns any responses. The interface between the TDD and the TDDL is called the TDD interface (TDDI), and is defined by the TPM vendor. The interface between the TDD and the TPM device is also defined by the TPM vendor.

A.6.2.2 The TSS Core Services

The TSS core services software module resides in user mode, executes as a system service, and provides single threaded access to the TPM. The TCS provides a common set of services to platform service providers, which cannot directly access the TPM. There must be only one TCS per platform operating system. The interface to this module is the TCS interface (Tcsi). It is anticipated that in most environments the Tcsi will reside as a system process, separate from the application and service provider processes.

- The TCS context manager provides dynamic handles that allow for efficient use of both the service provider's and the TCS's resources.
- The TCS key and credential manager stores and manages keys and credentials associated with the platform, the user, or individual applications, preventing unauthorised access to them.
- The TCS event manager provides the functions to store, manage and report PCR event structures and their association with the correct PCRs.

- The TCS parameter block generator converts the parameters passed to the TCS into the byte stream expected by the TPM.

A.6.2.3 The TSS Service Provider Module

The TSS service provider software module provides a common set of services for applications running on a TCG-enabled platform, and may be accessed via the TSP interface (Tspi). The TSP interface is an object oriented interface that resides within the same process as the application. The TSP provides high-level TCG functions required by applications, as well as a small number of auxiliary functions not provided by the TPM, such as cryptographic hashing.

- The TSP context manager provides dynamic context handles that allow for efficient use and management of both the application's and the TSP's resources.
- The TSP cryptographic functions provide auxiliary cryptographic functionality not provided by the TPM, such as hashing and byte stream generation.

A.7 Properties of a TPM

The TPM is a computing engine which must be resilient against software attack and some forms of physical attack. The TPM must be either physically or cryptographically bound to the CRTM. The TPM comprises the RTS and the RTR. We next examine the functional components offered by a typical TPM in order to support RTS and RTR capabilities.

A.7.1 Protected capabilities and shielded locations

Protected capabilities and shielded locations are terms introduced by the TCG to illustrate the protection level required for the implementation of TPM functions and protected TPM data areas, without dictating any TP implementation details.

- Shielded locations are areas in which data is protected against interference or snooping.
- Protected capabilities are the set of commands with exclusive permission to access shielded locations [149]. They are those capabilities whose correct operation is necessary for the platform to be trusted [5].

A TPM manufacturer must ensure that the prerequisite properties of protected capabilities and shielded locations are satisfied by the chosen TPM implementation.

A.7.2 TPM functional components

Table A.1 below describes the functional components of a version 1.2 conformant TPM.

Table A.1: TPM functional components

Functional Component	Contents	Explanation
Input and output		This component manages information flow over the communication bus: It performs encoding/decoding for communications over external and internal buses; It routes messages to the appropriate TPM components; It also enforces access control policies associated with TPM components.
Key generation	RSA asymmetric key generation	The implementation of the asymmetric key generation function must be in accordance with IEEE P1363 [71]. No requirements are made on key generation times. The generate function is defined as a protected capability.
	Nonce creation	All nonces must use the n next bits in the RNG.
Cryptographic co-processor:	RSA engine	Used for digital signature generation and encryption. RSA 512, 768, 1024 and 2048 [101] must be supported. Use of RSA 2048 is, however, recommended.
	Signature operations	The TPM must use RSA for the generation of signatures to be verified by entities outside the TPM. Other algorithms may be used for signature generation, but there is no requirement that other TPM devices either accept or verify such signatures.
	Symmetric encryption engine	The symmetric encryption engine is used to: Encrypt authentication information; Provide confidentiality in transport sessions; and Provide internal encryption of blobs stored off the TPM.

		The TPM does not expose the symmetric operations for general message encryption. A Vernam 1-time-pad [101] with XOR is used to protect authentication information and transport sessions. The shared secret key used for the 1-time pad is constructed from the pair of nonces exchanged between the parties. If the data to be protected is larger than the 1-time-pad, the mask generation function, MGF1 [93], is used to expand the entropy to the size of the data. For internal encryption the TPM designer may, however, choose any algorithm which provides required level of protection.
Execution engine		Runs program code to execute the TPM commands received through the I/O port. It ensures: Operations are segregated; Shielded locations are protected.
HMAC engine		The HMAC engine provides 2 pieces of information to the TPM: Proof that the request arriving is indeed authorised; and Proof that the command has not been modified in transit. TPM must support HMAC calculation according to RFC 2104 [95]. The key size must be 20 bytes. The block size must be 64 bytes.
SHA-1 engine		SHA-1 must be implemented as defined by FIPS 180-1 [109].
Power detection		This component manages TPM power states and platform power states.
Random number generator (RNG)	Entropy source	The entropy source provides data which is as unpredictable as possible. This data may either be inserted into, or generated inside, the TPM.

	Entropy collector State registers The volatile register The non-volatile register The mixing function	The collector collects the entropy and removes any bias. Hold the most recent RNG state. The volatile register is affected by all input from entropy sources and the mixing function. The volatile register is loaded from the non-volatile (NV) register at start up, and saved to the NV register at power down. Takes the value from the volatile register, puts it through a mixing function to make it truly random, and produces the RNG output. Results are also recorded to the volatile state register. Output must conform to FIPS 140-1 [108] PRNG requirements.
NV memory		Will hold persistent state and identity information, e.g. the endorsement key or storage root key.
Volatile memory		Used for storing keys in use by the TPM (active TPM keys) for signature generation or decryption.
Opt-in		The opt-in component maintains the state of persistent and volatile flags, whose values signify whether or not an entity is, or needs to be, physically present at the TP at a given instance, and how this physical presence may be demonstrated. This component also enforces the semantics associated with these flags.

A.7.3 PCRs

A TPM must contain a minimum of sixteen 20-byte PCRs, so that integrity measurement digests collected by the RTM can be securely stored. All PCRs are classified as shielded locations, see section A.7.1, inside the TPM. See section A.11.1 for further details.

A.7.4 The endorsement key

A unique 2048-bit RSA key pair, known as an endorsement key, must exist within every genuine TPM, where the private key is used for decryption only. This endorsement key is stored in a TPM shielded location. The endorsement key may be generated inside the TPM, using the TPM_CreateEndorsementKeyPair command, or, alternatively, by an outside generator. The entity that initiates endorsement key pair generation is also the entity that creates a credential attesting to its validity, and to the validity of the TPM.

The private endorsement key must never be exposed outside the TPM. While the public endorsement key may be revealed outside the TPM, overuse of the public endorsement key may result in privacy issues. These issues are explored in more detail in section A.10.

A.8 Initialising the TPM

Before the TPM becomes fully operational, it passes through several operational states. The events or triggers which prompt the TPM to begin and complete the initialisation process and to enter a fully operational state are summarised below.

First the TPM must be initialised. The TPM_Init command, a protected capability, is a physical method of initialising the platform and may be executed by applying power to the platform, or by physically resetting it. This physical method of initialising the TPM puts it into a state where it waits for the TPM_Startup command.

After the TPM has been initialised, limited self-tests are performed by the TPM on a minimal set of TPM commands in order to ensure that they are

working properly. The TPM commands examined include:

- TPM_SHA1xxx;
- TPM_Extend, used for the extension of platform configuration register values, see section A.11;
- TPM_Startup; and
- TPM_ContinueSelfTest.

The TPM_Startup command then transfers the TPM from an initialisation state to a limited operational state. The platform informs the TPM of the required platform operation state by inputting one of the following values into the 'startupType' parameter of the TPM_Startup command.

- 'Clear' results in the TPM values being set to a default or non-volatile operational state.
- 'Save' causes the TPM to recover state, including PCR values, saved to non-volatile memory following the successful execution of the TPM_SaveState command.
- 'Deactivated' informs a TPM that any further operations should not be allowed. The TPM turns itself off and can only be reset by performing another TPM_Init command.

Finally, the TPM is transformed to a fully operational state after the successful completion of all remaining self-tests. This can be accomplished in one of two ways:

- A complete self-test of the TPM capabilities can be explicitly requested, using the TPM_SelfTestFull command; or

- The TPM_ContinueSelfTest command can be called, which causes the TPM to test all the TPM internal functions that were not tested at start-up. If the TPM is running in compliance with FIPS-140 evaluation criteria [108], then the TPM_ContinueSelfTest command will request that the TPM perform a complete self-test.

Results of the self-test are stored within the TPM.

A.9 Enabling, activating and taking ownership of the TPM

Once the above processes have been completed, the TPM is ready for operation. Eight possible operational modes exist, based on whether the TPM is enabled or disabled, active or inactive, and owned or unowned. The TPM command sets which enable/disable the TPM, enable/disable the process of taking ownership of the TPM, and activate/deactivate the TPM, can be used in combination so that taking ownership of the TPM can be accomplished without the risk of malicious TPM use.

A.9.1 Enabling the TPM

Initially, a TPM must be enabled so that the process by which a prospective owner takes ownership of the TPM can be completed. A single non-volatile TPM flag, pFlags.tpmDisabled, which is asserted in hardware, is used to represent the enablement status. This flag cannot initially be changed to pFlags.tpmDisabled = FALSE with normal computer controls. It may only be changed via the execution of the physical command TPM_PhysicalEnable, which may be achieved by flicking a dedicated switch on the platform. The effect of physically enabling the TPM persists between boot cycles.

There are three fundamental commands associated with enabling a TPM.

- TPM_PhysicalEnable physically enables a TPM.
- TPM_PhysicalDisable physically disables a TPM.

Both of these commands may be used by anyone who can prove that they are physically present at the platform, for example by accessing and changing a dedicated switch or jumper, either before or after the TPM has acquired an owner.

- TPM_OwnerSetDisable is used to put a TPM into either an enabled or a disabled state after the TPM has acquired an owner. The input of the correct TPM owner authorisation data is required via a challenge-response protocol, see section A.16.2, before this command can be executed.

Once ownership has been established, pFlags.tpmDisabled = TRUE causes all TPM functionality to be turned off, with the exception of PCR value tracking. In this way, the TPM always has an accurate record of the platform state. Changing the pFlags.tpmDisabled flag to FALSE permits the TPM to act normally, even after a period where the pFlags.tpmDisabled flag was set to TRUE in the current boot cycle.

A.9.2 Enabling ownership of the TPM

The fFlags.OwnershipEnabled flag must be set to TRUE if the process by which a prospective owner can take ownership is to succeed. In order for the

TPM_TakeOwnership command to succeed, however, the TPM must also be enabled. A single non-volatile TPM flag is used to represent the ownership enabled status. This flag must be changed via a command which requires a physical presence at the platform, but software may be used to demonstrate

physical presence, for example by using normal computer controls (e.g. by using the keyboard). After the TPM has an owner, the status of this flag has no effect.

A.9.3 Activating the TPM

When a TPM is deactivated, the execution of commands that utilise TPM resources is not permitted. A deactivated TPM is in a similar state to a disabled TPM, with the exception that the TPM_TakeOwnership command may be executed on a deactivated TPM.

There are two activate flags, a non-volatile TPM flag, which requires physical presence to change, and a volatile TPM flag, which is set to the same state as the NV flag at power up. The NV flag, pFlags.tpmDeactivated, may be changed using the TPM_PhysicalSetDeactivated command, which requires physical presence.

There is also a TPM_SetTempDeactivated command which will set the volatile flag, vFlags.tpmDeactivated, of an activated TPM, to true, i.e.

vFlags.tpmDeactivated = TRUE, thereby deactivating the TPM until it is rebooted. This command requires an entity to demonstrate his/her physical presence at the platform before it may be executed, or alternatively, demonstrate knowledge of the operator's authorisation data prior to the command's execution (this latter case can only occur after the operator authorisation data has been set). If an entity's physical presence at the platform was demonstrated before the TPM_SetTempDeactivated command was executed, e.g. through the use of a dedicated switch or jumper, then the TPM will remain deactivated until this jumper or switch is changed again. This can be done by anyone with physical access to the platform.

The TPM may be activated and deactivated without destroying secrets pro-

tected by the TPM. When the TPM is deactivated, integrity measurements are still calculated and stored by the TPM. The activate commands are designed for use in conjunction with the 'enabling ownership' command, described in the previous section, in order to prevent rogue software taking ownership of a platform before the true owner does, and in turn taking control of all TPM functionality [5].

When the TPM is enabled, i.e. OwnershipEnable = TRUE, and permanently deactivated, i.e. the value of the NV flag tpmDeactivated = TRUE, the genuine TPM owner can execute the taking ownership process, and then turn on the remaining TPM functions by physically activating the TPM. However, if a remote entity (software) successfully executes the take ownership command on a deactivated TP before the legitimate owner, it will gain only restricted access to TP functionality until the platform is activated. As physical presence is required for TPM activation, the remote software cannot perform this step, and thus the potential control that rogue software may have over a hijacked TPM is limited.

A.9.4 Taking ownership

By default, a TPM is shipped with no owner, and taking ownership of TPM is achieved in the following manner. Once the TPM has been enabled and the fFlags.OwnershipEnabled flag has been set to TRUE, the following steps are performed:

- Twenty bytes of TPM owner authorisation data, which is to be shared between the owner and the TPM, is inserted into the TPM under the protection of the TPM's public endorsement key, see section A.7.4. This data is labeled the 'owner authorisation secret' and will enable the TPM

owner to gain access to TPM commands that require the owner authorisation data to be input before they are executed (i.e. owner authorised commands), see section A.16.2;

- The owner then informs the TPM which type of asymmetric key to create as the storage root key, see section A.13.1;
- The authorisation secret for the SRK is sent to the TPM under the protection of the TPM's public endorsement key;
- The TPM then generates a nonce (tpmProof), i.e. a 160-bit secret value. This nonce is later associated with non-migratable TPM key objects by the TPM, so that, when this value is later found associated with a particular TPM key, the TPM knows that it generated the key. Non-migratable TPM keys, see section A.13, are created inside the TPM, are locked to an individual TPM, and are never duplicated. They are never available in plaintext outside of the TPM.

The owner's authorisation secret, the private part of the SRK, the SRK's authorisation secret, and the nonce, tmpProof, are all kept in non-volatile storage in the TPM. If the TPM owner should forget his authorisation secret, the old value may be removed and a new one installed. The removal of the old value does, however, invalidate all information associated with the previous value.

A.9.5 Clearing the TPM

In order to clear a TPM, two commands are defined.

- The first is the owner clear command, TPM_OwnerClear, an 'owner-authorised' command. This command remains available for use by the TPM owner unless the disable clear function, TPM_DisableOwnerClear,

is executed. Once this has been invoked, the only way to clear the TPM is via physical presence.

- The second command is the force clear command, TPM_ForceClear, which requires the assertion of physical presence. As above, this command is available unless the owner executes the disable force clear command, TPM_DisableForceClear. In this instance, however, the force clear command only remains disabled until the next start-up.

The clearing of the TPM results in the following actions:

- Invalidation of the SRK;
- Invalidation of the tmpProof;
- Invalidation of the TPM owner authorisation data; and
- A reset of both volatile and non-volatile data to manufacturer defaults.

The endorsement key pair, however, is not affected.

A.10 Platform identification and certification

Each trusted platform will have an associated set of credentials, some of which were briefly mentioned in section A.5.

A.10.1 An endorsement credential

A TPME, see section A.5, vouches that a TPM is indeed genuine via the installation of an asymmetric key pair in the TPM, called an endorsement key pair, see section A.7.4, and an associated endorsement credential. This endorsement credential, includes TPM type information and the public endorsement key, and

is signed by the TPME. The private endorsement key is held securely in the TPM.

As mentioned earlier in this appendix, the endorsement key has the following properties:

- It is an asymmetric key pair located in a TPM's internal persistent memory;
- Each TPM has exactly one such key pair;
- The private key is used for decryption only, and cannot be exported from TPM;
- The public encryption key can be exported from the TPM for use by other parties;
- Generally, the endorsement key pair will be installed on or generated within the TPM by the manufacturer before shipping.

The entity which embeds the endorsement key pair in the TPM, also certifies the public encryption key from the pair by generating an endorsement credential, and is known as a TPME. This credential contains:

- A statement reflecting the fact that it is an endorsement credential;
- The public endorsement key value;
- The TPM type and security properties;
- A reference to the TPME; and
- The signature of the TPME on the credential.

A.10.2 A conformance credential

Multiple conformance credentials may be issued for a single TP, one for the TPM, for example, and others for disparate TBB components [149]. When signing a conformance credential, the evaluator is attesting to an evaluation result, details of which may be available for inspection [149].

The TCG has two protection profiles, i.e. methods used to describe the security properties of equipment with respect to the common criteria [110].

- The first describes the TPM; and
- The second describes the attachment of a TPM to the platform.

Manufacturers write security targets describing their equipment designs and how they meet particular protection profiles [5]. These security target documents are then scrutinised by conformance labs who can then issue a corresponding conformance credential for each security target that meets the protection profile [5].

A conformance credential contains:

- A statement reflecting the fact that it is a conformance credential;
- The evaluator name;
- The platform manufacturer name;
- The platform model number;
- The platform version;
- The TPM manufacturer name;
- The TPM model number;

- The TPM version; and
- The signature of the evaluator on the credential.

A.10.3 A platform credential

The PE offers assurance in the form of a platform credential that a particular platform is an instantiation of a TP. In order to create a platform credential, a platform entity examines the endorsement credential of the specific TP; the conformance credentials of the specific TP; and the platform to be certified.

Following this, the PE signs a credential containing:

- A statement that it is a platform credential;
- A reference to the endorsement key in the TPM, for example the identifier of an endorsement certificate;
- A reference to conformance credentials;
- A platform type and security property description;
- A reference to the PE; and
- The signature of the PE on the credential.

A.10.4 Attestation identities

P-CAs, see section A.5, enable identities and attestation identity keys to be assigned to trusted platforms. A P-CA will verify all TP credentials to ensure that a particular TP is indeed genuine, and then create an attestation certificate which binds a public attestation identity key to a TP identity label and generic TP information. The private AIK may then be used by the TPM to generate signatures.

A platform may have multiple TP identities, where a TPM identity or TP identity is synonymous with an attestation identity or AIK pair. Such an identity must be:

- Statistically unique;
- Difficult to forge; and
- Verifiable to a local or remote entity [5].

The attestation identity guarantees that certain properties hold for the platform associated with the identity. That is, it proves that a platform is a given type of TP. AIKs also allow for past behaviour, linked to a particular TP identity, to be collated. Information associated with a particular AIK can only be correlated with other TP identities by the P-CA, thereby providing a level of anonymity to the platform.

A.10.4.1 AIK uses

TPM private attestation identity keys can only be used to sign data created by the TPM, where fundamental uses include:

1. To prove to a challenger of the platform that data existed within the platform when that platform was in a particular state. The TPM creates a digital signature using a private AIK over the data in question and the current PCR values. This signed bundle is then sent with the public AIK credential, which attests to the TPM's identity, to the challenger, who then validates the signature and PCR values.
2. Certifying other keys generated within the TPM:
    - Private AIKs can be used to sign statements about the properties of secondary asymmetric keys generated within the TPM;

- These secondary keys must be fixed to the platform and cannot be migrated.

A.10.4.2 AIK credentials

An attestation identity credential is constructed in the following way.

1. A P-CA uses the endorsement credential, the platform credential, and the conformance credential(s), to verify that the platform is a TP with a genuine TPM.
2. The P-CA then creates an identity credential containing:
   - A statement that it is an identity credential;
   - The identity allotted by the TPM owner to the public key;
   - The public AIK to be associated with this identity;
   - The TPM type and the security property description from the endorsement certificate;
   - The platform type and the security property description from the platform certificate;
   - A reference to the P-CA; and
   - The signature of the P-CA on the credential.

The procedure used to allow a TPM owner to create a TPM identity and obtain an AIK credential has two main steps: identity creation and identity activation [5].

1. The platform sends a message to the TPM requesting the generation of an AIK pair.
2. The AIK key pair is then generated by the TPM.

3. The private AIK is protected by encrypting it with the TPM storage root key.

4. The TPM then signs the newly generated public AIK; the TP identity name chosen by the TPM owner; and the identity of the P-CA chosen to attest to the AIK. This is called an identity-binding.

5. A TSS command is called to assemble all data needed by the P-CA, i.e. the platform credential set and the identity-binding, generated in step 4.

6. The assembled data is then sent to the chosen P-CA, encrypted under the public key of that P-CA.

7. The P-CA decrypts the bundle received.

8. The P-CA inspects the credentials and checks whether it is indeed the P-CA being asked to generate the new identity for the TP.

9. The P-CA then creates an AIK credential, encrypts it with a symmetric key, and encrypts the symmetric key such that it can only be decrypted by that specific TPM, verified as genuine (using the public endorsement key of the TPM).

10. A hash of the AIK public key from the identity-binding, generated in step 4, is also generated and encrypted using the public endorsement key of the TPM by the P-CA.

11. These items are then sent to the TPM.

12. The TPM decrypts the hash of the AIK public key and the symmetric key used to encrypt the AIK credential.

13. The TPM then compares the decrypted hash of the AIK public key received against the hashes of all its public AIKs (if the data was intended

for this TPM, then the hash will equal the hash of a public AIK belonging to the TPM).

14. If a match is found, the TPM releases the decrypted P-CA symmetric key to the host platform.

15. Release of the symmetric key permits the decryption of the AIK credential.

After public release of the specifications describing P-CA use, controversy arose regarding the P-CA role. It was suggested that they represented a point of weakness in the system, since, in order to assign attestation identities to TPMs, P-CAs must collect an abundance of information about the specific TPM, including the endorsement key of the TPM. This information, if disclosed, could potentially compromise TPM user privacy.

A.10.5 DAA

In order to address the above criticisms of the role of the P-CA, an alternative scheme known as direct anonymous attestation (DAA) was introduced. DAA may be used by a TPM to convince a remote verifier that it is indeed valid without the disclosure of the TPM public endorsement key, thereby removing the threat of a TTP collating data which may jeopardise the privacy of the TPM user. DAA is based on a family of cryptographic techniques known as zero knowledge proofs [147]. In order to complete a DAA protocol-run:

- The TPM must first generate a set of DAA credentials through interaction with a DAA credential issuer. This can be done multiple times. The DAA-credentials are generated in an interaction between the TPM and the issuer, in which the TPM employs a TPM-unique secret that remains within the TPM. This TPM unique secret is used in every instance of

DAA-credential creation, and is distinct from, but analogous to, the endorsement key [147].

- The DAA credentials are then used as often as necessary in interactions with a second party called the verifier. At the end of the protocol the verifier can determine if the TPM contains a valid set of DAA credentials from a particular issuer, and may therefore decide whether the TPM is genuine. The verifier will, however, have no knowledge that might allow it to distinguish one particular TPM from others that also have valid DAA credentials [147].

While DAA offers a more private way of demonstrating that a TPM is valid, the trusted third party model which uses a P-CA and requires the disclosure of a TPM's public endorsement key as described above also remains a valid approach.

A.11 Integrity measuring, recording and reporting

In this section we examine how the integrity of a trusted platform may be measured, recorded and reported, so that a challenger can evaluate whether or not a platform is in a software state that can be trusted for a particular purpose. Following this, we explore how these integrity measurements can be used in the provision of a secure boot mechanism.

A.11.1 Platform configuration registers

The measurement of a trusted platform's software state results in the generation of measurement events. These events are of two types: measured values, which are representations of embedded data or program code, and measurement

digests, which are hashes of these values. The measurement digests are stored in platform configuration registers in the TPM using RTR and RTS functionality. The measurement values are stored in the stored measurement log (SML), outside the TPM.

- A TPM must provide sixteen or more PCRs each of which can be used to record an aspect of the platform's state;
- Each storage register has a length equal to a SHA-1 digest, i.e. 20 bytes;
- Each PCR holds a summary value of all the measurement values presented to it, as this is less expensive than holding all the individual measurements in the TPM. This also enables an unbounded number of measurements to be stored;
- A PCR value is defined to be equal to *SHA-1*(the existing PCR value $\|$ latest measurement digest);
- A PCR must be held in a TPM shielded location, in which its confidentiality and integrity are protected;
- A PCR is initialised to all zeros during system boot, but keeps existing values during sleep; in fact a record of any sleep events may be kept in a PCR (to avoid rogue activity during sleep mode);
- The fewer PCRs there are, the more difficult it is to determine the meaning of the PCR. The more PCRs there are, the more costly a TPM becomes.

Potential uses of the PCR values include the following [5]:

1. To help provide a 'platform attestation statement', which gives evidence of the state of the trusted platform at the time of signing;

- In this instance the TPM concatenates arbitrary data (usually a digest of data to be signed and a nonce to provide freshness) and the PCR values and then signs them.

2. They are also used in the protected storage mechanism, see section A.13, so that it can be determined whether secrets should be revealed to the platform in its present state.

   - In this instance the current PCR values are compared with the intended PCR values, stored with the data.
   - Access is only granted, and data unsealed, if no difference is found between the two.

A.11.2 Data integrity register

Data integrity registers are storage registers that hold digest values. Version 1.1 of the TCG specifications [146] requires that a TPM contains one 20-byte DIR in a TPM-shielded location. A specific use of the DIR is not specified. However a DIR may, for example, be used to aid the implementation of a secure boot mechanism [5].

If a TPM contains multiple data integrity registers holding values representing all of the expected PCR values, then the following secure boot implementation may be deployed. Every time a PCR is filled and its final value computed, its value is compared to the equivalent DIR value. If the two values match, the boot process continues; otherwise an exception is called and the boot process is halted.

Alternatively, if the TPM has access to non-volatile memory, all expected PCR values may be held in unprotected non-volatile memory and their summary or cumulative digest held in a single DIR. Every time a PCR is filled and its

final value computed, it is checked that:

1. Each PCR value, when calculated, matches the expected value held in non-volatile memory; and
2. The cumulative digest of the expected table of PCR values matches the value held in the DIR.

As is clear from the above, read access to DIRs must be provided without the required input of any authorisation data, as typically authorisation data is not available early in the boot process when the DIR value is read.

In the version 1.2 specifications, use of the DIR is deprecated. The TPM must still, however, provide a general-purpose non-volatile storage area.

A.11.3 Integrity measurement

In a PC, where the CRTM may be integrated into part of the BIOS called the BIOS boot block, integrity metrics may be measured and recorded as follows [5]:

1. The BBB starts the boot process, measures its own integrity and the integrity of the entire BIOS, and stores the measured values in the SML, saving the measurement digests in *PCR-0*, for example;
2. The BBB then passes control to the BIOS, which contains a measurement agent (MA) responsible for measuring the option ROMs, storing the measured values in the SML and the measurement digests in *PCR-1*;
3. Control is then passed from the BIOS to the option ROMs, which carry out their normal operations and pass control back to the BIOS;
4. The BIOS then measures the OS loader, and stores the measured value in the SML and the measurement digest in *PCR-2*;

5. Control is then passed to the OS loader, also containing an integrated MA, which carries out its normal functions and then measures the OS;

6. Finally, the OS MA generates and stores the measured values and measurement digests of OS components and any additional software loaded onto the platform.

7. This measurement agent will remain, running on the OS, measuring additional software when it is loaded onto the platform.

Details of all events measured and recorded in PCRs are detailed in the SML.

A.11.4 Assessing the software state of a platform

Once the measurement digests of the platform have been generated and stored in the PCRs, and their corresponding measured values stored in the SML, a challenger may use these PCR values to assess whether the software state of a platform can be trusted for a particular purpose.

A challenger presents the TP with an integrity challenge or nonce, which the TPM signs in conjunction with the relevant PCR values, using a private AIK. The TP agent then forwards this signed data on to the challenger, in conjunction with the relevant SML entries and TP AIK credential. The challenger validates the response, and makes a decision whether the challenged TP can be trusted for the intended purpose.

It must be noted, however, that in any transaction that requires a challenged platform to demonstrate its current state before any further interaction occurs, there will be a window between the beginning and the end of the communication during which the platform configuration may change. Therefore, the challenger must request signed integrity metrics both before and after the transaction is completed.

Alternatively, an exclusive transport session may be instigated between the challenger and the TPM, such that only requests initiated by that particular challenger of the TPM can be made within this session. If a TPM request is made by some party other than the particular challenger, the exclusive transport session is aborted and an exception generated. A change in the platform's software state could be configured to lead to a TPM_Extend command being called so that the platform PCRs can be updated/extended. An exception would then be generated by the TPM and the challenger notified of this. This mechanism could be used by a TP challenger to detect any changes that occur in the platform's state for the duration of the exclusive transport session. This mechanism is described in greater detail in section A.14.

In order to simplify the verification that must be completed by a TP challenger in the generic protocol run described above, one of the following methods, which are described in [5], may be used.

- In a constrained environment, where there are only a limited number of acceptable software states, the PCR representations that correspond to these software states may be directly certified by a validation entity. In this way a challenger can check the entire software state of the challenged platform in a single step.
- Instead of this, a third party could carry out intermediate validations at regular intervals. In this instance, the chosen third party would then certify that integrity information validated at a particular point complies with a particular policy. Integrity information could then be replaced by this single entry representing all of the previous certified history.
- Alternatively, a TTP may be trusted to evaluate the whole software state on behalf of a challenger.

A.12 Locality

The introduction of platforms facilitating the execution of trusted processes in protected execution environments, as described in sections 1.6.8, A.23 and A.24, is increasingly being discussed. Locality modifiers permit trusted processes communicating with the TPM to indicate to the TPM that a particular command has originated from a trusted process, the definition of which is platform-specific.

The trusted process sends the commands to be executed by the TPM in conjunction with a modifier, which indicates that the process from which the command has originated, is executing, for example, within a trusted system partition.

The TPM cannot, however, validate the modifier received, but the TCG specifications require that, in order to spoof a modifier to the TPM, some expertise and possibly specialist hardware should be required. The TPM is initially expected to be capable of understanding four modifiers.

Locality modifiers may also be used in conjunction with PCRs to define the platform configuration. Rather than allocating a specific PCR to record integrity metrics for a particular subset of platform components, PCR attribute fields may be defined at manufacturing time such that particular PCRs may be:

- Extended only by processes running in particular localities; and
- Reset at times other than TPM start-up, by calling processes in particular localities, such that trusted processes running in protected memory locations can be started up and shut down without resetting the entire PC.

When sealing an object, see section A.13, platform configuration data may specify the PCRs reflecting the environment in which the sealing was completed

and the environment into which the data may be released. The value of the locality modifier set when the data was sealed and the value of the locality modifier that must to be set if the data is to be released, may also be specified.

A.13 Protected storage

The TPM protected storage functionality was designed so that an unlimited number of secrets could be protected on a platform. The protected storage feature only provides functions to access these protected secrets. Functionality to control how they are used, or to protect them from deletion, is not provided.

Protected storage provides data confidentiality via asymmetric encryption of data, where asymmetric cryptography is deployed for two reasons [5]:

- Asymmetric cryptography is already required to implement TPM identities, and therefore its reuse minimises the cost of a TPM; and
- If plaintext originates outside the TPM then it can be encrypted there without revealing the decryption key (ensures that the TPM does not become a bottleneck for encryption operations).

Protected storage also provides implicit integrity protection of data objects in the form of an authorisation check [5]. A TPM protected object may include encrypted authorisation data, which is compared to the authorisation data (of length equal to a SHA-1 digest, i.e. 20 bytes) submitted to the TPM when access to a protected object is requested; by this means, access is refused to entities without the proper authorisation. The fact that 20 bytes of authorisation data is required before a TPM protected object can be accessed, implicitly protects it from tampering.

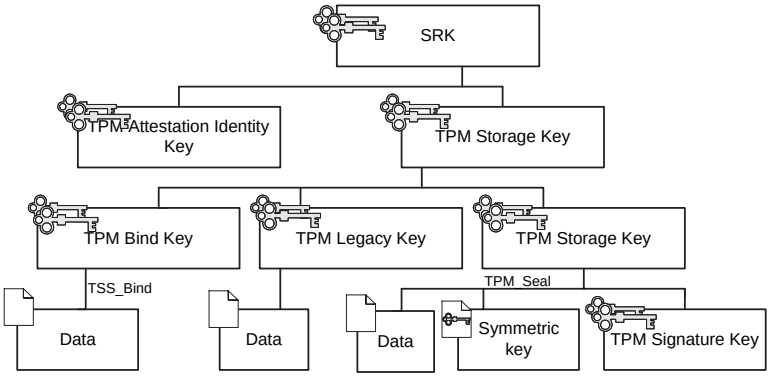

Figure A.1: The protected storage object hierarchy

A.13.1 Object hierarchy

An example of a TPM protected storage object hierarchy is illustrated in figure A.1. It is comprised of both protected key objects and protected data objects. The only TPM key to be permanently loaded in the TPM is the storage root key, a non-migratable storage key, created inside the TPM. The public portion of the SRK is used to encrypt the first layer of TPM key objects, and the corresponding private key is used to decrypt the same objects when their use is required. The entire object hierarchy of a TPM is essentially protected by the SRK. Each TPM protected data object is encrypted using an encryption key which is itself usually a TPM protected object stored outside the TPM. The set of TPM protected key and data objects, therefore, comprise an object hierarchy, where each child TPM protected object is encrypted using the encryption key in the parent TPM protected object.

All keys within a trusted platform can be divided into two fundamental categories, i.e. non-migratable keys and migratable keys. Non-migratable key

pairs have the following properties.

- In the case of non-migratable asymmetric key pairs, the private key is only known to the TPM that created it. Non-migratable keys may be certified using the AIKs of the TPM, or using general purpose signing keys.
  - They are locked to a given TPM;
  - They are never duplicated;
  - They must be created by the TPM; and
  - tpmProof, which is known only to the specific TPM, is attached to all non-migratable objects in the placeholder of migration authorisation information.
- In the case of migratable asymmetric key pairs, there are no guarantees about the origin or use of the private key. Migratable keys have the following properties.
  - They can be replicated ad infinitum by the platform owner;
  - They can be created outside the TPM or by the TPM;
  - They are protected by the TPM; and
  - The owner of a migratable TPM object creates the migration authorisation information.
- The version 1.2 TCG specifications also describe a third type of key, namely certifiable migratable keys, which are keys that can migrate but still have properties which the TPM can certify. When CMKs are created, control of their migration is delegated to a migration (selection) authority. CMK migration is examined in section A.20.

The TCG specifications define many different types of key, including:

- Storage keys, used to encrypt or decrypt keys and data objects in the protected storage hierarchy;
- Signature keys, used for signing operations;
- Attestation identity keys, used by TPM aware applications to prove that data has come from a genuine TPM, see section A.10;
- Binding keys, used for TSS bind and TPM unbind operations, as described below;
- Legacy keys, which are for systems in which the same key is used for signing and encryption; and
- Change authorisation keys, which are ephemeral keys used during the process of changing authorisation information, as described in section A.16.4.

A.13.2 Sealing

Sealing is an important aspect of protected storage. The seal and unseal operations, TPM_SEAL and TPM_UNSEAL, are used in order to encrypt and decrypt arbitrary data. The tpmProof field in the sealed data structure binds the arbitrary data to be protected to an individual TPM, proving that the data was sealed on that particular TPM. In conjunction with this, the data is bound to platform configuration data (PCRinfo) so that it can only be revealed by the TPM when the platform is in a particular software state.

When TPM_SEAL is executed, a TPM_STORED_DATA structure is used to represent the protected data, and is composed of:

- The TPM version;
- The size of the sealed info parameter;

- The sealed info parameter, which contains PCR information. The value of this element may be set to null if the data is not bound to specified PCRs. Alternatively it may contain information on: the locality modifier when the data is created; the locality modifier required to reveal sealed data; the selection of PCRs active when the blob is created; the selection of PCRs to which the data or key is bound; the digest value of the PCR values when the blob is created; the digest of the PCR indices and PCR values to verify when revealing sealed data.
- The size of the encrypted data, encDataSize;
- The encrypted TPM_SEALED_DATA structure, encData, which contains the following fields:
  - The payload type;
  - The authorisation data for the sealed object;
  - The tpmProof;
  - STORED_DIGEST: A digest of the TPM_STORED_DATA fields excluding the encDataSize and encData fields;
  - The size of the data parameter to be sealed; and
  - The data to be sealed.

A.13.3 Binding

In the case of binding, external data, created outside the TPM, is encrypted under a TPM parent bind key. Binding is not a TPM function, but a bound object can only be unbound (unencrypted) by the target TPM using a TPM command. A TPM bind key must always be used to create a bound object. The bound data structure contains the following elements:

- The TPM version;
- The payload type; and
- The bound data.

The TPM unbind function takes the resulting stored data structure, decrypts it and exports it for use by the caller. The caller, however, must initially authorise the use of the decryption key.

A.13.4 Wrapping

Wrapping is a TSS capability which allows an externally generated key to be encrypted under a TPM parent key. Execution of the TSS_WrapKey command creates a migratable blob from a key presented externally. The key creator can, however, prevent migration of the key by a user, by wrapping the key using a non-migratable storage key and loading random data for the migration authorisation data.

TSS_WrapKeyToPCR is a TSS command which allows for an externally generated key to be concatenated with the platform configuration data and encrypted under a parent key.

Finally, the TPM_CreateWrapKey command allows for the generation of a key inside the TPM, after which it is concatenated with platform configuration data and encrypted under a parent key.

A.14 Transport security

Transport protection enables the establishment of a secure channel between the TPM and secure processes, offering confidentiality and integrity protection of commands sent to the TPM. It also provides a logging function such that all

commands sent to the TPM during a transport session can be recorded.

Central to transport security is the concept of session creation, where the creation of a session allows for:

- A set of commands to be grouped;
- A log of all commands to be recorded; and
- Confidentiality and integrity protection of commands sent within the session to be provided.

A.14.1 Session establishment

The session establishment process launches a transport session, and, depending on the attributes specified for the session, may lead to the creation of shared keys and session logs.

Session establishment involves the generation by the caller of 20 bytes of transport authorisation data, for use between the caller and the TPM. This transport authorisation data has two purposes:

- It is used as input when generating an encryption key for use between the TPM and the caller, to provide command confidentiality; and
- It is also used as a HMAC key when sending the TPM_ExecuteTransport command, so that the command can be authorised.

The authorisation data is generated by the caller and encrypted under a public key whose corresponding private key is available only to the TPM. The key used is pointed to in the encHandle field of the TPM_EstablishTransport command sent.

The TPM then decrypts the authorisation data and creates an internal transport structure, which contains the following fields.

- authData, to the authorisation data received and decrypted.
- tranPublic, consists of:
  - transAttributes, where information regarding the encryption of the session, logging of the session, and whether the session is to be exclusive is set;
  - algoId, which refers to the properties the symmetric key to be generated will have; and
  - encScheme, which refers to the encryption scheme to be used.
- transHandle, to which a value is assigned.
- transEven, is set to the value of the nonce generated, transNonce.
- transDigest, is initially set to NULL but is then used to reflect all logged transport events.

If transport encryption is required, the TPM must validate that the requested key generation algorithm and encryption scheme are supported. If logging is required, the log in and log out structures must be created for each logged event. If an exclusive session if requested, then the relevant flag must be set in the transAttribute field.

A.14.2 Transport encryption and authorisation

Confidentiality protection of data transmitted inside the transport session is provided through the encryption of the command to be protected or, more

specifically, encryption of the input and output parameters associated with the command.

Authorisation of the execute transport command is provided by the calculation of a HMAC on a subset of elements from the wrapped command, the command ordinal, header information and data fields, where the transport authorisation data fixed during session establishment is used as the HMAC key. This HMAC also allows the integrity of the bundle to be verified and a link made between the establish session and execute transport commands.

A.14.2.1 Transport encryption

Following session establishment, the trusted process can call the execute transport command which delivers a wrapped command to the TPM, which can, in turn, unwrap and execute the command.

The execute transport command has the following structure:

- (Execute transport header ‖ wrapped command ‖ execute transport trailer), where:

  Wrapped command = (ordinal and header information ‖ key and other handles ‖ data ‖ authorisations), where authorisations represents the usual authorisation data required in order to utilise the specified wrapped command.

Due to resource management issues, encryption of the entire wrapped command is impossible. Therefore, the command data field contains the only information to be encrypted.

Confidentiality of the transport session can be provided by XORing the command data with the output of the MGF1 function, where the into the MGF1 function consists of:

- Transport session authorisation data, as generated by the caller; the oddnonce provided by the trusted process; the evennonce provided by the TPM; and either the number 1 or 2, used to indicate the direction of the communication.

A.14.2.2 Execute transport authorisation

The output from an HMAC computation is used so that knowledge of the transport authorisation data required for use of the execute transport command can be demonstrated by the caller and the integrity of the wrapped command assured.

The caller calculates:

- *H1 = SHA-1(ordinal and header information* $\|$ *decrypted command data)*, where both input strings are taken from the wrapped command above; and
- $HMAC_K$*(execute transport header and ordinal information* $\|$ *H1* $\|$ *Transport Nonce Even* $\|$ *Transport Nonce Odd* $\|$ *Continue Transport Session value)*, where K is the transport authorisation data set up between the caller and the TPM for this session.

This value is then sent to the TPM which validates it.

The TPM then:

- unwraps the command, validates the ExecuteTransport authorisation HMAC, validates the wrapped command authorisation data, and executes the command;
- calculates *H2 = SHA-1(The return code* $\|$ *ordinal and header information* $\|$ *output data)* of the wrapped command;

- calculates *S2 = SHA-1(The return code for the execute transport || the ordinal and header information for execute transport || current ticks || H2)*
- returns to the caller $HMAC_K$*(S2 || Transport Nonce Even || Transport Nonce Odd || Continue Transport Session value)*, where K is the transport authorisation data set up between the caller and the TPM for this session.

A.14.3 Transport logging

A session log provides a record of each command using a particular session, using a structure that includes the parameters and current tick count for each logged command.

If transport logging functionality is set, entries are written to a TPM_TRANSPORT_LOG_IN structure on the receipt of each wrapped command to be logged. Similarly, for outgoing data, after the execution of the wrapped command, entries are written to a TPM_TRANSPORT_LOG_OUT structure. The summaries of these internal structures are stored to the transDigest field of the internal transport structure described above.

A.14.4 Error handling and exclusive transport sessions

Provisions are also made for error handling. The concept of exclusive transport sessions is also discussed. In this case if a caller establishes an exclusive session with the TPM, the session is invalidated if any TPM command outside the established session should attempt to execute. Only one exclusive session may be supported by the TPM at any one time.

A.15 Monotonic counter

A monotonic counter provides an incremental value. The TPM must support at least four concurrent monotonic counters which may be implemented as:

- Four unique counters; or as
- One counter with pointers which keep track of the current values of the other counters.

The fundamental counter components include the internal base and the external counters. The internal base represents the main counter. It is used internally by the TPM, and is not accessible to processes running outside the TPM.

External counters are used by external processes. They may be related to the main counter via pointers, via difference values, or, alternatively, they may be unique. The values of these counters are not affected by any use, incrementation or deletion of any other external counter. External counters must allow for seven years of increments taking place every five seconds. The output of the counters is a 32-bit value. To create an external counter, TPM owner authorisation is required. In order to increment an external counter, knowledge of the correct authorisation data must be demonstrated. Finally manufacturers are free to implement monotonic counters using any chosen mechanism.

A.16 Demonstrating privilege

In order to demonstrate the level of privilege required to execute various TPM commands:

- An entity may demonstrate physical presence at the platform; or, alter-

natively,

- An entity may demonstrate knowledge of the required authorisation data.

A.16.1 Physical presence

There are three particular occasions where demonstration of physical presence at the platform may be necessary in order to execute particular TPM commands, usually in the case when cryptographic authorisation is unavailable. These occasions include the operation of commands that control the TPM before an owner has been installed; when the TPM owner has lost cryptographic authorisation information; or when the host platform cannot communicate with the TPM.

The PhysicalPresenceV flag in volatile memory indicates whether physical presence has been confirmed, i.e. whether a particular dedicated switch or jumper has been manipulated, for example. If this flag is set to TRUE, then the following commands may be executed:

- TPM_ForceClear;
- TPM_PhysicalEnable;
- TPM_PhysicalDisable;
- TPM_PhysicalSetDeactivated; and
- TPM_SetOwnerInstall, which enables the take ownership process.

It is advised that TPM designers take precautions to ensure that this flag is not maskable, so that physical presence cannot be faked.

A.16.2 Cryptographic authorisation

As an alternative to physical presence, cryptographic authorisation mechanisms may be used to authenticate an owner to their TPM, or to authorise the release

and use of TPM protected objects. An authorisation value, which must be 20 bytes long, could for example, be a hashed password or 20 bytes from a smartcard. It must always be treated as shielded data and only ever used in the authorisation process.

A variety of authorisation data is held by a TPM, including:

- Unique TPM owner authorisation data, input of which is required before any 'owner-authorised TPM command' may be executed;
- TPM object usage authorisation data, input of which is required before an object protected by the TPM may be accessed; and
- TPM object migration authorisation data, input of which is required before a TPM key object can be migrated, as discussed in A.20.

In order to demonstrate knowledge of the relevant authorisation data to the TPM, an entity may deploy one of two challenge-response protocols, namely the object independent authorisation protocol or the object specific authorisation protocol. These protocols are described below.

A.16.3 The OIAP

The object independent authorisation protocol is the more flexible and efficient of the two challenge-response authorisation protocols. Once an OIAP session has been established, it can be used to demonstrate knowledge of the authorisation data associated with a particular TPM object or TPM command.

It proceeds as follows.

1. A TPM_OIAP command is used to start an OIAP authorisation session (this command may be executed without the input of any authorisation data).

2. The TPM creates a handle to track the authorisation session and a nonce, and sends these to the caller.

3. The caller then requests the use of a specific TPM command, e.g. TPM_example1, where TPM_example1 is a TPM command that uses the key K_1. This implies that, in order to use the command, the caller must demonstrate knowledge of the authorisation data for K_1.

    - To prove knowledge of this authorisation data, the caller sends to the TPM:
        - the key handle of the key to be accessed;
        - the command ordinal representative of TPM_example1, the first input argument, the second input argument, the handle assigned to the authorisation session by the TPM, the nonce generated by the caller, and an indication as to whether or not the session is to be kept open; and
        - $HMAC_K(m)$, where K is the authorisation data required for access to and use of the key K_1, and message m is composed of: *SHA-1(the command ordinal representative of TPM_example1* $\|$ *first input argument* $\|$ *second input argument)* $\|$ *(the handle assigned to the authorisation session by the TPM* $\|$ *the nonce sent by the TPM* $\|$ *the nonce generated by the caller* $\|$ *an indication as to whether or not the session is to be kept open)*

4. If the TPM successfully verifies this, and it is indicated that the session is to be kept open:

    - A new nonce is generated by the TPM to replace the last TPM nonce.
    - The command TPM_example1 is executed.
    - The TPM then sends to the caller:

- the returnCode, the output argument, the newly generated TPM nonce, an indication as to whether the session is to be kept open or not; and
- $HMAC_K(m1)$, where K is the authorisation data required for access to and use of the key K_1, and message $m1$ is composed of:

  SHA-1(the returnCode $\|$ *the command ordinal representative of TPM_example1* $\|$ *the output argument)* $\|$ *(the handle assigned to the authorisation session* $\|$ *the newly generated TPM nonce* $\|$ *the caller nonce received* $\|$ *an indication as to whether the session is to be kept open or not).*

5. The caller verifies that the returnCode and the output parameters are correct.

6. The session may end here or, alternatively, if it has been indicated that the session is to be kept open, the caller saves the new TPM nonce received, creates a new caller nonce, calls a new command (which may demand the demonstration of knowledge of new authorisation data to enable its execution) and the process continues, as described above.

A.16.4 The OSAP

The second protocol defined in the TCG specifications is the object specific authorisation protocol. This protocol allows for the establishment of a session to prove knowledge of the authorisation data for a single TPM object, and minimises the exposure of long-term authorisation values. It may be used to authorise multiple commands without additional session establishment but, as we discuss below, the TPM_OSAP handle specifies a specific object to which all authorisations are bound.

During this protocol an ephemeral secret is generated (via the HMAC of the session nonces exchanged at the beginning of the protocol, with the target TPM object's authorisation data used as the HMAC key) by the TPM and the caller, which is used to prove knowledge of the TPM object authorisation data. This particular protocol is necessary for TPM functions that set or reset authorisation data.

In conjunction with the terms defined above for the OIAP protocol, two extra nonces are defined for use with OIAP: the nonceOddOSAP and nonceEvenOSAP. These nonces are HMACed with the authorisation data for the protected object or the authorised command used as the HMAC key, to generate the shared secret key. OSAP proceeds as follows:

1. A TPM_OSAP command is used to start an OSAP authorisation session, and this command is sent to the TPM in conjunction with the handle associated with the object, K_1, to be accessed and utilised, and an OSAP nonce.

2. The TPM generates:

   - a handle in order to track the authorisation session;
   - an OSAP nonce;
   - a shared secret by HMACing the caller OSAP nonce and the TPM OSAP nonce, where the authorisation data needed for access to and use of the key K_1, is used as the HMAC key; and
   - a TPM replay nonce.

3. The authorisation handle, the TPM OSAP nonce, and the TPM replay nonce are then sent to the caller who also generates the shared secret key

as above; (this can only be accomplished by the caller if it knows the authorisation data for K_1).

4. The caller then requests use of a specific TPM command, TPM_example1, where TPM_example1 is a TPM command that uses the key K_1, which implies that the caller must demonstrate knowledge of the authorisation data for key K_1 in order to use the command.

    - In order to prove knowledge of this data, the caller generates and sends to the TPM:
        - the command ordinal representative of TPM_example1, the first input argument, the second input argument, the handle assigned to the authorisation session by the TPM and the nonce generated by the caller; and
        - $HMAC_K(m)$, where K is the shared secret and message m is composed of:

          SHA-1(the command ordinal representative of TPM_example1 || first input argument || second input argument) || (handle assigned to the authorisation session by the TPM || the replay nonce sent by the TPM || the replay nonce generated by the caller above || an indication as to whether or not the session is to be kept open).

5. If the MAC is sucessfully verified by the TPM, and it is indicated that the session is to be kept open:

    - A new replay nonce is generated to replace the last TPM replay nonce.
    - The command is executed.

- The returnCode, the output argument, the newly generated TPM nonce, an indication as to whether the session is to be kept open or not; and
- $HMAC_K(m1)$, where K is the shared secret, and message $m1$ is composed of:
  SHA-1(returnCode $\|$ *the command ordinal of TPM_example1* $\|$ *output argument)* $\|$ *(handle assigned to the authorisation session* $\|$ *the newly generated TPM nonce* $\|$ *the caller replay nonce received* $\|$ *an indication as to whether or not the session is the be kept open or not).*

6. The caller verifies the correctness of the returnCode and parameters.
7. The session may end here or, alternatively, if it has been indicated that the session is to be kept open, the caller saves the new TPM replay nonce, creates a new caller replay nonce, calls a new command (which demands the demonstration of knowledge of the same authorisation data to enable its execution), and the process continues as described above.

A.16.5 Changing authorisation data

Once authorisation data has been associated with an object, it can be securely changed at any stage. Two mechanisms are supported in order to accomplish this objective. The first mechanism is the authorisation data change protocol (ADCP):

- ADCP is used for changing authorisation data bound to TPM objects that have a parent. For example, it can be used to change the authorisation data associated with data (the child object) encrypted under a TPM key (the parent object) which in turn has authorisation data associated with

it;

- In this instance, the owner of the parent object co-operates to set up an OSAP session for the parent object;
- The OSAP ephemeral secret is generated, as described in section A.16.4, and is then sent to the child object owner so that it can enter the new child authorisation data into the TPM confidentially;
- This OSAP ephemeral secret is then xored with the newly generated authorisation data for the child object.

However, as is clear from the description of the ADCP, the parent object owner can gain access to the new authorisation information for the child object by eavesdropping on TPM commands.

In order to overcome this issue, it is advised that a normal TPM_ChangeAuth command (the ADCP) is used inside a transport session with confidentiality, as described in section A.14.

The second mechanism is a variant of ADCP:

- This protocol is specifically used in order to change the TPM owner's authorisation data, or the authorisation data associated with the storage root key (i.e. TPM objects with no parents);
- In this instance, the TPM owner acts as the parent to the SRK, and also as its own parent;
- The new authorisation value, be it for the TPM owner or the SRK, is rendered confidential by xoring it with an ephemeral secret, generated using an OSAP session based on the TPM owner's authorisation data.

A.16.6 The ADIP

The protocol used to insert new TPM object authorisation data during the creation of a TPM object is called the authorisation data insertion protocol (ADIP). A new TPM object must always created under an existing parent TPM object but, in order to use the required parent object, knowledge of the associated parent object's authorisation data must be proved.

To demonstrate knowledge of the parent object's authorisation data, an OSAP authorisation session, as described in section A.16.4, is established, and the OSAP nonce to be used for shared secret key generation is sent to the TPM. The TPM then chooses a random OSAP nonce and generates the OSAP shared secret from the TPM and caller OSAP nonces and the authorisation data associated with the parent object. A nonce to be used for freshness is also chosen. The caller then generates the shared secret using the same method as is used by the TPM.

1. The parent object owner then passes K, computed as the *SHA-1(the shared secret key* $\|$ *the freshness nonce sent by the TPM)*, to the child object creator;

2. The creator of the child object then chooses 20 bytes of authorisation data to be associated with the child object and protects it by xoring it with the key K, received in step 1;

3. All information is verified by the TPM and, if all is in order, the new authorisation data is extracted, the command used to create a new entity is executed, the new entity is created, the new authorisation data is securely associated with it, and the output parameters are delivered to the caller.

4. In order to protect the child object's authorisation data against eavesdrop-

ping by the parent object's owner, it was previously recommended that ADIP should be closely followed by AACP, but since use of this scheme is deprecated, the execution of ADIP followed by a normal TPM_ChangeAuth command used inside a transport session with confidentiality is recommended.

A.17 Context manager

Because the TPM has limited resources, caching of resources may occur without the knowledge or assistance of the application which has loaded the resource. In version 1.1 two types of resources needed this mechanism, namely keys and authorisation sessions, both of which had separate load and restore operations. Version 1.2 introduced the concept of the transport session, which also requires management.

In order to consolidate context management, a single context manager is defined in the version 1.2 specifications, which all resources may use. In this instance a resource manager can request that a resource be wrapped in a manner that:

- Securely protects the resource when it is evicted from the TPM; and
- Allows for the resource to be restored onto the same TPM during the same operational cycle.

The encryption scheme used to protect these cached objects may be symmetric or asymmetric and the keys used for protecting the blobs may be temporary, regardless of whether they are 2048-bit RSA keys or 128-bit AES keys. A new key could, for example, be generated for this purpose at start-up. Alternatively the TPM could generate a key for context management and store it persistently

in TPM persistent data.

A.18 Delegation

Prior to the version 1.2 specifications, the TPM owner had the privilege to control all aspects of the TPM's operation, i.e. use of all TPM commands including the 'owner-authorised' command set, the set of TPM commands that require TPM owner authorisation data to be entered before their execution is authorised. Therefore, if any aspect of the TPM required management, it became the direct responsibility of the TPM owner. As an alternative, privileged information belonging to the TPM owner (the 20 bytes of TPM owner authorisation data) would have to be forwarded to another entity, trusted to perform only a particular subset of operations, thereby potentially leaving the platform vulnerable to attack.

In order to alleviate the risk associated with the above management method, a delegation model has been included in the version 1.2 specifications, so that a TPM owner can delegate particular privileges to individual entities or trusted processes, through the construction of new authorisation data and the association of specified TPM ownership rights with this authorisation data. The delegation process differs depending on the entity to which the specified privileges are being given.

The delegation process is summarised in table A.2.

A.18.1 Family and delegation tables

In order to implement this delegation model, two tables, a family table and a delegate table, are required.

The delegate table [156]:

Table A.2: The delegation process

Delegation to	Requires	Additional Information
A generic entity	Privileges may be associated with a generic entity by: Dictating PCRs which define a particular process; and/or Stipulating a particular authorisation value; to which privileges are associated.	The resultant blob is then passed to the chosen entity
An external entity	If privileges are to be delegated to a specified external entity: A null PCR selection; and An authorisation value; are required.	The authorisation value may, however, be sealed to a set of PCRs on the remote platform
A trusted process provided by the local OS	If privileges are to be delegated to a trusted process provided by the local OS: A PCR that defines the specific trusted process; and A known authorisation value; are required. 0x111 111 is suggested by the TCG specifications as the known authorisation value	Generic authorisation data is sufficient, since the OS has no means of securely storing the authorisation token. If sealing is used to protect the randomly chosen authorisation value associated with the delegation, the security is no stronger than associating the delegation with a known authorisation value.

- Lists ordinals for commands which may be used by a specified delegate;
- Holds the identity of a process which may use the commands represented by the ordinal list; and
- Holds the authData value necessary to use the commands represented by the ordinal list.

The family table provides a means for delegations to be validated, revoked and edited.

The delegate table must have a minimum of two rows, whereas the family table must have a minimum of eight rows and may have many more. The size of these tables, however, does not restrict the number of delegations because the TPM facilitates the caching of delegations off the TPM, should the number of table rows become limited. When cached, the TPM protects both the confidentiality and integrity of the delegations. A counter mechanism is also used so that encrypted cached delegations can be validated as fresh when they are loaded back onto the TPM.

Each entry in the delegate table contains six fields:

- PCR information, which defines a particular process to which the privileges have been delegated;
- Authorisation data for delegated capabilities, which defines a particular authorisation value to which delegated privileges have been associated;
- A delegation label;
- The family ID, identifying the family to which the delegation belongs;
- The verification count, which specifies the version of this row; and

- A profile of the capabilities delegated to the trusted process identified by the PCR information or the authorisation data.

The TPM owner can also delegate the management of tables to particular entries.

The family table provides a method of grouping rows in the delegate table. It supports the validation and revocation of exported delegate table rows and those stored in the table. The family table must have at least eight rows, and each entry in the family table contains the following four fields:

- The family ID;
- The family label, which helps identify information associated with the family;
- The family verification count, which represents the sequence number identifying the last outside verification and attestation of family information; and
- The family flags.

In order to validate delegation rows, the following process is used. The TPM produces a signed statement, grouping all delegations from the same family. This signed statement then allows a verification agent to examine the delegations and the processes involved, and then to make an assessment as to the validity of the delegations.

Before any signed statement is produced, the verification counter is incremented and inserted into selected table elements (all of which are from the same family), including temporarily loaded delegations. Copies of these elements are then made, signed and sent to a chosen TTP.

Alternatively, the counter may not be incremented before the platform delegations are sent for validation. This, however, leads to the possible use of undesirable delegations which have not been validated, permitting undesirable actions when loaded onto the delegate table. This occurs because these undesirable delegations were not on the platform when its delegation state was validated, but have the same counter value as blobs that were, as no increment of the verification counter took place prior to the validation process. In this case there is no way, therefore, of differentiating between validated delegations and undesirable, unvalidated, delegations.

The platform owner also requires assurance that no management of the table is possible during the validation process. In order to ensure this occurs, the transport session established during this verification process will have the exclusive attribute set. This ensures that no other TPM operations can occur during the validation process. The TTP then returns the results of the assessment.

A.18.2 The delegate-specific authorisation protocol (DSAP)

In order to delegate privileges, the delegation creation entity must initially demonstrate knowledge of the authorisation information associated with the key or command to which access will be delegated.

The delegation entity then creates a TPM_Delegate_Key_Blob or a

TPM_Delegate_Owner_Blob, depending on whether key use or TPM owner command execution is being delegated.

This TPM_Delegate_Key_Blob, TPM_Delegate_Owner_Blob, or, alternatively, a pointer to an entry in the internal delegation table, which contains the authorisation data necessary for use of delegated command or key, is encrypted such that only the TPM can decrypt it. This encrypted value is then sent to the

entity being granted delegated use of the key or command, in conjunction with the authorisation data necessary for use of delegated command or key use.

The entity which has been delegated restricted privileges can now start a TPM_DSAP session using TPM_Delegate_Key_Blob, TPM_Delegate_Owner_Blob, or a pointer to an entry in the internal delegation table, as input. The protocol proceeds as follows:

1. The TPM_DSAP command is used to initiate the protocol. In conjunction with the command ordinal, input parameters for this command include:
   - a key handle associated with the object, K, to be accessed and utilised or a handle to the privileges which have been delegated;
   - a caller OSAP nonce;
   - an entity type element; and
   - an entity value element, where the entity type and entity value elements represent the delegation structure which has been allotted to the caller, be it a TPM_Delgate_Owner_Blob, a TPM_Delegate_Key_Blob or a TPM delegation table index value.
2. If a TPM_Delgate_Owner_Blob or a TPM_Delgate_Key_Blob is received, its integrity is initially checked. Alternatively, if entityType is TPM_ET_DEL_ROW or TPM_ET_DEL_KEY, the entityValue will consist of a TPM_DELEGATE_INDEX, which points to a delegation entry in the delegate table.
3. The encrypted authorisation data necessary for use of delegated command or key is extracted by the TPM from either the TPM_Delegate_Key_Blob, TPM_Delegate_Owner_Blob, or, alternatively, from an entry in the internal delegation table and decrypted. When the authorised data has been

decrypted by the TPM, it is used along with the TPM and caller OSAP nonces to generate a shared key between the caller and the TPM.

4. The DSAP authorisation protocol then proceeds as did the OSAP protocol where the shared key generated by the TPM and the entity requesting access, by the process described above, is used by the entity requesting access to prove to the TPM that usage of a particular command or key should be granted.

A.19 Time-stamping

The TPM provides a service enabling time stamps to be applied to various data strings. The timestamp provided, however, is not a universal clock time but a representation of the number of ticks the TPM has counted. The caller can then have this value associated with a universal time clock value externally.

No requirements are made regarding how the tick counter mechanism is chosen and implemented. Neither are any requirements made regarding the ability of the TPM to continually increment the tick counter. This is convenient, in that a PC with continual power supply may deploy a more continuous reliable timing method than a mobile platform with a limited battery, whose timing tick maintenance capability may be limited because of power limitations.

For each tick session, the values specified in table A.3 are maintained by the TPM.

A basic tick stamp result consists of a TPM digital signature computed over:

- The data to be time-stamped (a digest of the data to be time-stamped);
- The current tick counter value;
- The tick session nonce; and

Table A.3: Tick session values

Value maintained by TPM for tick session	Description
The tick count value, TCV	Counts the ticks for the session. Must be set to 0 at the start of every tick session. If the TPM loses the ability to increment the TCV in accordance with the TIR, the TCV must be set to null and a new tick session started.
The tick increment rate, TIR	The rate at which the TCV is incremented (its relationship with seconds is set during TPM manufacture).
The tick session nonce, TSN	The TSN is set at the start of each tick session. It must be set to the next value of the TPM RNG at the beginning of each new tick session. If the TPM loses the ability to increment the TCV in accordance with to the TIR, the TSN must be set to null.

- Some fixed text.

An example of a protocol illustrating how a tick counter value can be associated with a universal time clock value by employing the functionality of a time-stamping authority is described in the specifications. The protocol specified, however, is merely illustrative. No particular protocol is mandated.

A.20 Migration mechanisms

Migration mechanisms are used for the backup and cloning of migratable TPM-protected key objects. They allow the private keys from TPM protected key objects to be attached to other TPM protected storage trees. Two methods which facilitate the migration of the private part of a TPM protected key object are described in [5].

Both mechanisms initially require that:

- The TPM owner has authorised a particular migration destination, i.e. the use of a particular destination or intermediary public key for a particular migration method, be it migrate or rewrap;

- The authorisation data to use the parent key currently wrapping the key that is to be migrated has been submitted; and
- The authorisation data required for key migration has been submitted [5].

Once the TPM owner has authorised the use of a particular destination or intermediary public key for a particular migration method, the target key to be migrated may be decrypted at any stage and migrated, if the appropriate authorisation data to enable access to the parent key currently encrypting the key that is to be migrated and the authorisation data required for key migration are submitted by the target key owner.

Then it is required that the source TPM encrypts the target key under the destination public key. This is then forwarded to the destination TPM in conjunction with a plaintext object describing the public key from the key pair to be migrated.

The alternate migration method involves the use of an intermediary. In this instance, the private key to be migrated is encoded using optimal asymmetric encryption padding (OAEP) and XORed with a one-time pad. The resultant data is then encrypted under the public key of the intermediary, which unwraps the key and rewraps it under the public key of the destination TPM. The XOR encryption prevents the intermediary gaining unauthorised access to the migrated key. The one-time pad must, however, be made available to the destination TPM so that the migrated key can eventually be integrated into the protected storage hierarchy of the destination TPM.

While the TPM will check that a particular destination or intermediary public key is at least as strong as 2048-bit RSA, it is up to the TPM owner to ensure that the public key does actually represent the desired destination TPM or intermediary [5]. A migratory key can essentially be sent to any arbitrary

platform, not necessarily a trusted platform.

With respect to certifiable migratable keys:

- The TPM owner must authorise the use of a particular destination public key;
- The authorisation data required for use of the parent key, under which the key to be migrated is encrypted, must be submitted; and
- The authorisation data required for key migration must be submitted [5].
- In conjunction with this, a chosen migration (selection) authority must authorise the migration destination.

In this way controlled migration of keys is made possible, where an entity other than the TPM owner may have input into the decision as to where CMKs are destined.

A.21 Maintenance mechanisms

Maintenance mechanisms are used to clone a broken trusted platform. This cloning process can only be completed with the co-operation of the TPM owner and the platform manufacturer, and the process must only be performed between two platforms of the same manufacturer and model. Maintenance mechanisms are optional, but if they are provided by the TPM, certain requirements as described in [156] must be met. The maintenance capability may be disabled until the current owner is erased.

A.22 Audit

The audit function allows the TPM owner to determine whether or not certain operations on the TPM have been executed.

The audit function in earlier versions of the TCG specifications, TCPA main specification, version 1.1, 2001, was shown to have security weaknesses [5], so an updated method of audit is described in the version 1.2 document set. In version 1.2 of the specifications, the audit mechanism consists of:

- A digest held internally to the TPM; and
- A log of all audited commands held externally to the TPM.

The securely stored internal digest allows for verification of the external log, so that tampering of any kind can be detected. Re-synchronisation functionality is also provided by the TPM, so that the internal digest and the external logs may be kept consistent with each other.

The TPM owner has the capability to choose which functions generate an audit log entry, and to alter this choice at any stage.

The auditing process itself consists of two fundamental steps:

- Auditing of the command and input parameter received; and
- Auditing of the response to the command and output parameters.

This method was chosen to diminish the amount of memory required to complete the auditing process, as no memory is required to save any audit information while the command is executing.

An internal audit record consists of:

- A non-volatile counter, which increments once per session when the first audit event of a session occurs; and
- A digest, which holds the digest of the current session, most probably volatile.

The audit process may therefore proceed as follows:

- An auditable command is called;
- The audit session opens when the volatile digest is extended from its null state with the input parameter from the command;
- When this audit session is opened, the non-volatile counter is incremented;
- When the command has executed, the response to the command (the return code) and the output parameters are then used to further extend the digest value;
- The audit session closes when the TPM receives the command to get the audit event signed and the close audit parameter. The explicit closing of an audit session addresses the potential threat of undetectable audit log truncation;
- The TPM then signs the concatenation of the non-volatile counter and the volatile digest, and exports the following three values:
  - The non-volatile counter value;
  - The volatile digest value; and
  - The signature.

A.23 NGSCB

In June 2002 [166] Microsoft released information on Palladium, a system which combined software and hardware controls to create a trusted computing platform. It was initially stated that Palladium was due to ship with the next major version of the windows operating system, code named Longhorn, then planned for release in 2004.

The name Palladium, however, has since been abandoned in favour of NGSCB. According to John Lettice of the Register [97], this change was made for two fundamental reasons: the name 'Palladium' had already been used by another company for a product in a similar area and because of the initial controversy and criticism surrounding the project. To date, however, few technical details regarding Palladium or NGSCB have been published. Primary reference material used in the writing of this section includes [43], [126], [24] and [104].

A.23.1 The relationship between TCG and NGSCB

Microsoft was one of the founding members of TCG and actively participates in the organisation [5]. The NGSCB architecture, however, encompasses a broader set of capabilities than the TCG-defined trusted platform, see section A.4. In addition to the functionality offered by the TCG, NGSCB requires:

- An extended CPU to enable the efficient implementation of a minimal isolation kernel;
- A minimal isolation kernel;
- Memory controller or chipset extensions such that direct memory access can be controlled; and
- Hardware components enabling input and output to be efficiently secured.

A version 1.1 compliant TPM is not, however, supported by Microsoft Windows unless a cryptographic service provider, which has been written to support the Windows cryptographic applications programming interface has been provided by the hardware manufacturer and integrated into the platform. A version 1.2 compliant TPM is expected to fulfil the role required crypto chip in a NGSCB.

A.23.2 The NGSCB architecture

An NGSCB platform includes three fundamental components.

- A tamper resistant crypto chip, which is implemented in hardware and is either physically or cryptographically bound to the platform.
- An isolation kernel, which facilitates the execution of several software compartments in parallel on the same machine, and controls the access of software running in these compartments to system resources.
- Software components, which may be hosted by the isolation kernel in isolated compartments, for example, an optional mass-market operating system or high assurance component.

A.23.3 The tamper resistant crypto chip

The tamper resistant crypto chip is required to provide the security primitives described in table A.4. It is required at a minimum to implement a variety of cryptosystems, a random number generator, a small amount of memory, and a monotonic counter. It must also contain at least one 'process control register' used to store the image of system software components, e.g. the isolation kernel. A TPM conforming to version 1.2 of the TCG specifications is a concrete implementation of an NGSCB crypto chip.

Table A.4: The NGSCB tamper resistant crypto chip

Component	Definition	Capabilities	Provision
NGSCB crytpo chip	A tamper resistant crypto chip	The following functionality must be provided to software components running on the platform into which the chosen crypto chip implementation is embedded. Seal Unseal Quote PKSeal PKUnseal ReadCounter IncrementCounter	The crypto chip will be inextricably linked to the motherboard.

Details of the fundamental security primitives which must be supported by the chosen crypto chip implementation are described in table A.5.

Table A.5: Security primitives supported by the NGSCB crypto chip

Primitive	Input	Output	Description
Seal	Source_Identity (Identity od the caller) AC_Information (The access control information, for example, the identity of the software entity allowed to unseal the secret) Data (to be sealed)	C = Store (Source_Identity ‖ AC_Information ‖ Data)	A cryptographic implementation of the store operation, which satisfies the requirements for an authenticated encryption scheme, as described in [9], is required. Implementations also require the crypto chip to have the functionality to securely store and access cryptographic keys, so that encryption and integrity protections can be performed.
Unseal	C (an identifier for the sealed bundle)	Source_Identity Data Only output if the access control information, AC_Information, is fulfilled	

Quote	Data (arbitrary data block)	S_K (Data ‖ Source_Identity) A public key signature is output on the above data with key K, which is protected by the crypto chip	
PKSeal	Target_Identity Data (data to be protected) K (a public key, whose corresponding private key is held in the crypto chip)	D = AsymmetricEncrypt_K (Target_Identity ‖ Data)	PKSeal allows a remote entity to encrypt a secret such that it is only accessible by a specified entity, the Target_Identity with private key, K. PKSeal does not need to be implemented in the crypto chip.
PKUnseal	D	Decrypts D and returns Data If Target_Identity is equal to the identity of the caller identity.	
Read Counter		V	Gets the value of the monotonic counter
Increment Counter			Increments the value of the monotonic counter

A.23.4 The isolation kernel

The isolation kernel [126] will execute in a CPU mode more privileged than the existing ring 0, effectively in ring -1, which will be introduced in forthcoming versions of the x86 processors. This allows the isolation kernel to operate in ring -1 and all guest operating systems/software components to execute in ring 0. Thus, problems that may occur with virtualisation, in the scenario where the isolation kernel executes in ring 0 and guest OSs or high assurance components must execute in ring 1, can be avoided.

The isolation kernel utilises the PTEC algorithm in order to partition physi-

cal memory among guests hosted by the isolation kernel. Any attempt made to edit a page map, which determines the physical to virtual mapping active for a particular guest, traps to the isolation kernel, which consults its security policy in order to decide whether or not the action may proceed.

The isolation kernel is also described as combining the merits of both virtual machine monitors (VMMs) and exokernels [43] in relation to OS compatibility. It resembles a minimised virtual machine monitor, in that it allows a mass market OS to operate with few changes. However, rather than necessitating the virtualisation of all devices, as a VMM does, the exokernel approach to devices is adopted, where devices are assigned to guest OSs which contain drivers for the devices they choose to support. Guest operating systems may then efficiently operate directly on the chosen device.

This does, however, leave the problem of uncontrolled DMA devices, which by default have access to all physical memory. In order to prevent DMA devices circumventing virtual memory-based protections, the provision of chipset extensions is required by the hardware manufacturers. A DMA policy map [43] is set by software with write access to the memory region which holds the policy map, usually the isolation kernel, and it is then stored in main memory. The DMA policy map is then read and enforced by hardware, for example the memory controller or bus bridges, where this policy map decides, given the state of the system, if a particular subject (DMA device) has access (read or write) to a specified resource (physical address).

Enhancements to input devices such as keyboards and mice may be deployed to facilitate the MACing and encryption of data as it is communicated to a trusted application on the platform. Secure graphics hardware may also be deployed in parallel to the complex mass-market graphics system, and used

only by the isolation kernel and high assurance guests.

In a fundamental sense, however, these input and output hardware changes are not strictly necessary. In principle, a piece of trusted code could be given physical control of the keyboard and the graphics card by the isolation kernel, and thus guarantee that input and output will not be observed or corrupted. There are two reasons, however, why new hardware for input and graphics is desirable:

- Minimising the size of the trusted computing base, which should ideally be kept as small as possible to preserve security; graphics drivers, for example, typically contain millions of lines of code.
- Performance and ease of running off-the-shelf legacy operating systems; OSs expect to have direct access to the graphics card. While VMMs routinely solve this problem by exposing a virtual graphics card to their guest OSs, in practice this solution entails significant performance degradation.

A.24 LaGrande

Following the description of NGSCB, we briefly examine LaGrande Technology, born out of Intel's initiative to address the challenges of trusted computing. LaGrande is defined as "a set of enhanced hardware components designed to help protect sensitive information from software-based attacks, where LT features include capabilities in the microprocessor, chipset, I/O subsystems, and other platform components" [72]. As was the case with NGSCB, detailed information on LT remains limited. LaGrande Technology essentially provides all the components needed to meet the requirements defined by Microsoft for hardware enhancements and extensions necessary to support their NGSCB architecture.

A.24.1 The architecture

The generic LaGrande Technology architecture consists of three fundamental concepts: the standard partition, the protected partitions; and the domain manager.

The standard partition provides an environment identical to today's Intel Architecture – 32 (IA–32) environment [72]. In the standard partition, users may freely run software of their choice. The existence of this standard partition implies that, despite the addition of supplementary security mechanisms to the platform, code already in existence will retain its value, and software unconcerned with security will have somewhere to execute unaffected.

The protected partition provides a parallel environment, in which software can be executed with the assurance that it cannot be tampered with by software executing in either the standard or protected partition. This protected partition is hardened against software attacks by the implementation of a number of components, described below, which provide domain separation; memory protection; protected graphics; and a trusted channel to peripherals.

The existence of a domain manager, which facilitates this domain separation, is also assumed. This domain manager may be constructed in various ways, depending on the architecture implemented. A concrete example of this domain manager is the isolation kernel as described in section A.23. The domain manager is physically protected via processor and chipset extensions and, in turn, protects standard and protected partitions from each other.

A.24.2 Hardware enhancements and extensions

In order to facilitate the implementation of the above partitions, in conjunction with protected input and output and TPM functionality to a platform, Intel are

in the process of extending and enhancing the following hardware components:

- The CPU;
- The memory controller or chipset;
- The keyboard and mouse;
- The video graphics card; and
- The graphics adaptor.

A v1.2 TPM must also be added.

A.24.2.1 The CPU

CPU extensions facilitate the efficient execution of standard and protected partitions, as described above. CPU enhancements, in conjunction with chipset extensions, also provide more stringent access control enforcement with respect to the use of hardware resources such as memory, thereby thwarting the threats caused by DMA devices.

Secure event management may also be facilitated. Through the extension of the CPU and chipset, the situation where an abnormal event may result in the transfer of control to a malicious agent outside the protective environment's boundaries, can be detected and handled appropriately.

Instructions to manage the protected execution environment and to establish a more secure software stack are also added.

A.24.2.2 The chipset

Chipset extensions allow for a memory protection policy to be enforced; facilitate the creation of protected channels to and from input/output devices; include

enhancements, which protect against direct memory access (DMA); and provide interfaces to a version 1.2 compliant TPM.

A.24.2.3 The keyboard and mouse

The keyboard and mouse extensions support secure communication between the mouse and keyboard and trusted applications. Protected input allows for protected channels to be established between input devices and applications running in the protected environments. This protects the confidentiality and integrity of input data against unauthorised or malicious software running on the platform.

Keystrokes and mouse clicks may be encrypted or MACed, or both, using keys shared between the protected domain's input manager and the input device. Only applications with the correct encryption key can then decrypt and use the transported data [72].

A.24.2.4 The video graphics card

The video graphics card extensions allow for display information to be sent to the graphics frame buffer without observation or compromise [72]. Protected output allows applications running in protected execution environments to securely send display information to the graphics frame buffer, with the assurance that it cannot be observed or tampered with by malicious software running on the platform [72].

A.25 Conclusions

In this appendix, the TCG specification set, NGSCB and LaGrande Technology have been examined. It is clear that, despite the negative criticism often associated with this particular area of computing, trusted computing technologies

offer a wide range of functionality which may be leveraged to improve computer security. It must also be noted, however, that this chapter reflects these technologies as they are currently documented. This area is the subject of much current research and development, and the specifications, functionality, architectures, mechanisms and implementations associated with trusted computing technologies are evolving and changing rapidly.

Appendix B

Technologies related to trusted computing

In parallel to the development of the trusted computing technologies described in appendix A, a number of alternative architectures have been developed, with the goal of providing more secure and trustworthy computing platforms. In this appendix three architectures of this type are examined, namely the IBM 4758, XOM and AEGIS architectures.

B.1 Introduction

Having examined current trusted computing initiatives in appendix A, this appendix provides an overview of closely related technologies.

Sections B.2 briefly introduces secure co-processors, and examines the IBM 4758. Section B.3 presents the concept of hardened processors and then examines the XOM and AEGIS architectures.

B.2 Secure co-processors

The most familiar of the alternative architectures involves platforms which contain secure co-processors. A secure co-processor provides a tamper resistant computing environment in which functions may be executed without interference, despite physical or logical access to the device. The IBM 4758 is an example of a user configurable, tamper resistant co-processor with the following properties and capabilities [41]:

- A random number generator;
- A layered design, where each device should have at least two layers, one responsible for implementing the security policy, which should be validated as correct, and a layer for device personalisation;
- Outbound authentication, which allows external entities to determine the exact applications running on the device;
- General co-processor and auxiliary processors, which, for example, allow the efficient implementation of DES and modular arithmetic;
- Persistent storage, which may protect application data or arbitrary data directly using the 4758 tamper response in battery backed RAM, or indi-

rectly by encrypting the data using keys stored in battery backed RAM; and also

- A third party programmable interface.

Co-processors essentially provide a tamper resistant secluded area on a platform, in which particular applications may be securely executed. As highlighted above, a variety of security services may be offered to applications executing or data processed within this region. In conjunction with this, outbound authentication is often permitted, allowing external entities to verify the exact applications running on the device. Co-processors are, however, separate from the generic platform processor, providing specialised protection for a limited set of applications and data, independently of what occurs on the generic platform with which the co-processor is associated. A challenger need only trust the specified co-processing environment. This segregated protection is relatively expensive to provide [5].

B.3 Hardened processors

As an alternative approach, rather then implementing a secure co-processor, which runs in parallel to the general platform processor, the primary processor can be extended (hardened) so that certain applications can be run securely in on-chip protected compartments. A process running in one compartment has only strictly controlled access to application code or data from another compartment. Off-chip compartment application code and data is also protected through the deployment of encryption and integrity-protection mechanisms.

We begin with a brief examination of the XOM architecture [99] which provides protected environments/compartments for XOM code to execute in. The XOM architecture essentially provides on-chip protection of caches and regis-

ters, protection of cache and register values during context switching and on interrupts, and confidentiality and integrity protection of application code and data when transferred to external memory. The platform subsystem used to provide the services listed above is called the XOM machine.

In order to practically and efficiently implement an XOM machine, extensive hardware additions must be made to the CPU. In a hardware implementation of the XOM machine, all trust is put in the modified CPU hardware. Everything transmitted outside the main CPU is encrypted.

On the other hand, however, the 'XVMM implementation' of the XOM machine, described below, significantly reduces the number of necessary CPU hardware extensions. A software XVMM, whose integrity is validated via secure boot, is used to provide many of the security services provided directly by the CPU in the hardware implementation.

We subsequently explore the AEGIS architecture [143], which builds upon concepts developed in the XOM architecture. Given the abstract AEGIS architecture, two potential architecture implementations are explored: an untrusted operating system solution, which involves implementing all security mechanisms within the hardened AEGIS processor, and a trusted security kernel solution, where some of the core operating system functionality is trusted, thereby enabling the minimisation of CPU modifications.

As is the case with the more efficient hardware implementation of the XOM architecture, the untrusted operating system implementation of the AEGIS architecture puts all the trust in the CPU hardware, and therefore requires a complete overhaul of current system architectures.

The 'security kernel implementation' of AEGIS, while also requiring CPU modifications to be made, utilises a security kernel so that the requisite hard-

ware adjustments may be minimised. Through the addition of secure boot and security services, loosely-coupled with the sealing and platform attestation mechanisms as described in appendix A, the boundaries of trust may be moved from the trusted hardware core to encompass the security kernel running on the hardware.

B.4 XOM

Executable only memory [99] was proposed with the objective of preventing software consumers from examining executable code, thereby protecting any algorithms incorporated into the code. It also aimed to thwart the unauthorised execution of software. While the XOM architecture is not an implementation of trusted computing methodologies, it uses concepts closely linked to the notion of trusted computing. Fundamental reference material for our discussion of XOM includes [98, 99].

B.4.1 The abstract XOM machine

In order to facilitate the execution of XOM code, an XOM machine, which supports internal compartments, is proposed. Within this XOM machine, "a XOM process executing in one compartment cannot read data from another compartment" [99], and all data that leaves a particular compartment on the XOM machine is integrity and confidentiality protected, as it is assumed that external memory is not secure.

An XOM machine, as defined by Lie et al. [99], has three fundamental tasks to fulfil:

- Decryption of the symmetric compartment key used to protect an incoming application, using the private key from the asymmetric key pair em-

bedded in that particular XOM machine;

- Decryption of the program code using the symmetric compartment key; and
- Isolation of the active principal, for example the decrypted code and its data, where principals are separated into compartments between which only strictly controlled information flow is possible.

B.4.2 XOM machine implementation

Lie et al. [99] consider two potential XOM machine implementations: a virtual machine implementation of a XOM machine and a hardware implementation of a XOM machine.

The virtual machine implementation of the XOM architecture involves running a special XOM virtual machine monitor on a slightly modified CPU, which integrates special microcode that incorporates [99]:

- A private key from a unique asymmetric key pair assigned to the XOM hardened processor;
- On-chip memory, which contains tagged shadow registers and the XOM key table;
- The ability to trap on instruction cache misses (to the XVMM); and
- A privileged mode under which XVMM runs.

The actual XVMM may be implemented either in software, where implementations must be authenticated via a secure boot, or in microcode. However, only the microcode XVMM implementation is explored by the authors.

The XVMM must execute as an authorised privileged program. It requires the processor to be configured to trap on instruction cache misses, so that the XVMM can then decrypt data and instructions coming from external memory. The XVMM then stores decrypted instructions to the instruction cache. On interrupts or context switches, where a copy of the encrypted XOM program instructions remains in external memory, the instruction cache may be flushed, as it contains no modified data.

While decrypted instructions may be saved to the instruction cache by the XVMM; the XVMM may not, however, store decrypted XOM program data to the data cache. Unlike decrypted XOM instructions, there are two types of program data: shared data, i.e. data generated during XOM program execution, which an XOM program may authorise other programs to access; and XOM program private data. Shared data may be pushed from the caches unencrypted, whereas XOM private data needs to be both confidentiality and integrity protected when evicted from the data cache. As there is no way to differentiate between data types held in the data cache without additional hardware extensions, the data cache is not used by the XVMM to store decrypted XOM program data.

In order to facilitate compartment access control in an XOM machine which contains no cache or register tags, the XVMM must facilitate interrupts and context switching by flushing instruction caches and clearing all registers. In order to support register clears, the XVMM must maintain a set of shadow registers for each compartment, where each of these registers stores a compartment ownership bit indicating whether the register is in the private or shared compartment and a saved bit indicating which registers were saved by the OS on interrupt. When a compartment is interrupted, private compartment register values are copied to protected shadow registers by the processor, and then

cleared, thereby protecting values from OS interrupt handlers. When the compartment is restarted, the values are restored. In order to support the above functionality, the XVMM must implement the following eight additional instructions:

- enter_xom, facilitates the initial decryption of the symmetric XOM application compartment key and decryption of the XOM program code and data, the registration of a handler for cache miss events and re-vectoring of all CPU exceptions and interrupts;
- exit_xom, unregisters the handler for cache misses and restores handlers for all CPU exceptions and interrupts;
- secure_store, is used to move data between private registers (where register status is stored in shadow registers) and the data cache and, ultimately, external memory. The XVMM computes a MAC on and encrypts private register values before eviction;
- secure_load, is used to import values into registers. The XVMM decrypts the value from memory and verifies the MAC. If the MAC can be verified, the register value is written and its 'private' status written to the shadow registers;
- move_from_shared and move_to_shared, change the status of the registers in the status register to and from 'private';
- save_register, moves all register values whose status is 'private' to the shadow registers; a record of the source register from which the value has been saved is also stored; and
- restore_register, restores shadow register values to the source registers.

The alternative hardware implementation:

- Facilitates the use of data caches;
- Alleviates the necessity to flush the instruction cache every time there is a trap;
- Eliminates the overhead incurred through the use of the XVMM for cryptographic operations; and, finally,
- XOM instructions are both interpreted and implemented in hardware, thereby decreasing overhead.

Additional CPU hardware modifications required to facilitate a hardware implementation of the XOM machine include the incorporation of:

- A private key from a unique asymmetric key pair assigned to the XOM hardened processor;
- On-chip memory, which contains tagged shadow registers and the XOM key table;
- On-chip cache and register ownership tags; and
- A hardware cryptographic engine for symmetric encryption/decryption and MACing.

Because of the efficiency and performance issues associated with the XVMM implementation of the XOM machine, we now examine the concept of XOM in further detail, focusing in particular on the hardware implementation when exploring XOM machine concepts and instructions.

B.4.3 Compartments

As stated above, each XOM processor chip has an embedded asymmetric key pair, where the private decryption key is protected on-chip and the public encryption key is made available so that anyone can encrypt code for the particular chip. If, however, code for a particular chip is encrypted under its public key, instruction loading would be extremely inefficient. Therefore, the header block of the message contains a symmetric key, encrypted using the public key of the XOM chip, and the program image is encrypted using this symmetric key. Each application is encrypted using a different symmetric key.

In order to support trusted or secure execution environments, a 'compartment' is used to isolate independent software applications running on the same processor, where a compartment is built from, and defined by, the symmetric compartment key used originally by the service provider to protect a distributed program image. The null compartment is defined as one where regular unencrypted code may run; it has no associated compartment key.

It is worth noting that, in order to protect the incoming XOM program image, not only should it be encrypted by the service provider before being communicated to the host, but it should also be integrity protected using a MAC or digital signature. According to [99], no mechanism is deployed in order to protect the integrity of the incoming program image while in transit from the service provider to the mobile host. The same holds for the more recent thesis of Lie [98], where the deployment of an integrity protection mechanism is not explicitly mentioned in the software distribution model. The integrity verification of an incoming application is, however, implicitly implied in the instruction definition, where Lie states that the enter_xom instruction, described below, must always be followed by an encrypted and MACed application. No

mention is made of how the MAC key is derived.

When a symmetric compartment key has been decrypted, the XOM machine associates it with an arbitrary XOM ID and a 128-bit hash of the encrypted compartment key in a key table stored in the XOM machine.

The isolation of active principals through the use of compartments, as defined above, may be achieved by the provision of three fundamental services; secure storage, security over interrupts and external memory protection.

B.4.3.1 Secure storage

On-chip, all XOM data and code in caches and registers is tagged with a unique XOM identifier, which is mapped to the code's decrypted symmetric compartment key in a compartment key table. Programs that run in the clear have a XOM identifier of 0. The size of the compartment key table and the number of XOM identifier tags depend on how many concurrently executing principals can have data in the machine.

At any one time, only one program will be executing; therefore there will only be:

- One active principal;
- One active XOM identifier; and
- One active compartment key.

When this active principal produces data, it is automatically tagged with the active XOM identifier. Subsequently, when an attempt is made by an active principal to read data, the tag on the data is compared with the active XOM identifier, and access is only permitted if these values are identical.

In order to implement the proposed functionality, a series of instructions and additional functionality must be added to the abstract machine. Two basic instructions are initially required of the abstract machine to facilitate compartment establishment, namely enter_xom and exit_xom, as described in table B.1. These instructions facilitate the decryption and integrity verification of XOM code and data from external memory into the instruction stream.

The instructions defined for use when moving data between caches and registers in the active compartment on the abstract machine include those listed in table B.2. These instructions act as normal load and store instructions for XOM processes.

Although isolation of principals is required, complete isolation, where principals are separated by compartments, and between which no information flow is possible, would prove impractical and a hindrance. In order to facilitate communication between protected principals, two further instructions are also defined (see table B.3). The mv_to_shared and mv_from_shared instructions provide a controlled way of changing tags associated with data values. Executing these commands on data that was not originally tagged, results in an exception.

B.4.3.2 Interrupts

Extra consideration must also be given to interrupts, so that an untrusted operating system can save the register state of an XOM process without leaving the register contents vulnerable to attack. Two further instructions are necessary in order to protect register values. These instructions, described in table B.4, package data in such a way that any principal, for example the OS, may be permitted to move the data with the knowledge that it cannot be tampered with in any way by the moving principal. This, therefore, provides the OS with a means of scheduling XOM processes without violating compartment security.

Table B.1: XOM enter_xom and exit_xom instructions

enter_xom	exit_xom
All XOM code is preceded by an enter_xom instruction. The source register holds the starting memory address of the encrypted compartment key for the XOM code. The enter_xom instruction indicates that all the following code belongs to a principal associated with the compartment key. The machine checks to see if the compartment key has already been decrypted by comparing the 128-bit hash of the encrypted symmetric key with those stored in the XOM key table. If the compartment key is already in the table, the active identifier is set to that entry and the encrypted XOM code is fetched. If no matching entry is found: A free entry in the XOM key table is found; An XOM ID is assigned to the key; The active identifier is set to this entry; The encrypted symmetric key is loaded and the asymmetric decryption algorithm is run on the key. The 128-bit hash of the encrypted compartment key and the decrypted compartment key are then entered into the key table. All instructions following the enter_xom instruction must be encrypted and accompanied by a valid MAC; otherwise they are not loaded into the instruction cache for execution.	This instruction changes the active identifier back to null and the machine stops decrypting. If an abnormal 'trap' or 'interrupt' occurs, an implicit exit_xom is executed before the instructions from the interruption handler are executed.

Table B.2: XOM secure_load and secure_store instructions

secure_load	secure_store
Used if the required value is found in the cache. If a line hits in the cache, the XOM cache tag of the cache line is compared with the active XOM ID. If the values match, then the value in the cache and the XOM ID are written to the register.	The XOM processor verifies that the source register tags matches the active XOM ID. If they match, the register value is stored to the cache and the particular cache line is tagged with the active XOM ID ownership tag.

Table B.3: XOM mv_to_shared and mv_from_shared instructions

mv_to_shared	mv_from_shared
Changes the XOM ID tag on a register to the null identifier. After execution of this instruction, access to data by the original principal results in an exception.	Changes the tag on a register to the tag of the active XOM identifier.

Table B.4: XOM save_register and restore_register instructions

save_register	restore_register
The XOM processor takes the contents of the register and creates an encapsulated version that other principals can move but cannot manipulate. To achieve this, the register contents are MACed and encrypted with a register key associated with the XOM ID in the XOM key table. Each XOM compartment register key is regenerated every time a particular XOM compartment is interrupted, to prevent the replay of saved register values.	Used to restore a register value. This instruction decrypts the register value, verifies the MAC and then restores it.

B.4.3.3 External memory

The mechanisms described above would prove sufficient in the implementation of XOM code if either external memory is secure or, alternatively, if there is no need to use external memory. However it is rare for either of these cases to hold, and it therefore became clear to the XOM designers that the compartments, as defined above, would have to be extended to incorporate external memory. In order to extend a compartment:

- Tagged data in caches is MACed and then encrypted with the appropriate compartment key before it leaves the abstract machine;
- A secure hash is generated on regions of memory and stored in on-chip

registers so that the replay of data in external memory can be detected. In turn, register keys prevent the replay of registers in which the memory hashes are stored.

B.5 AEGIS

Suh et al. [143] describe the AEGIS architecture for a single chip processor, which is designed to help build a system secure against physical and software attack, assuming untrusted external memory. As is the case with the XOM architecture, AEGIS is not an implementation of trusted computing, but it uses concepts closely linked to the notion of trusted computing. More specifically, the security kernel implementation of the AEGIS architecture incorporates concepts such as secure booting and notions loosely coupled with platform attestation and sealing, as described by the TCG; see appendix A.

It is claimed that, within this AEGIS architecture, both tamper evident and tamper resistant environments can be provided for multiple mistrusting processes. Tamper evident environments are defined as authenticated environments, in which physical or software tampering can be detected [143]. Tamper resistant environments are defined as private and authenticated environments where an adversary cannot gain any information about data or software within the environment by tampering with or observing system operation [143].

Suh et al. also describe two implementations of their architecture:

- In the first implementation it is assumed that the core functionality of the OS is trusted and implemented in a security kernel; and
- In the second implementation the use of an untrusted OS is assumed.

The fundamental reference material for the following description is [143].

B.5.1 Secure computing model: Assumptions

Before we examine the AEGIS architecture in detail, we highlight suppositions made by the authors with respect to the computing model. Abstractly speaking, the authors consider systems that are built around processing subsystems with external memory and peripherals. The processor chip is assumed to be trusted and protected from physical attacks, implying that the internal state cannot be tampered with or observed by physical means. The processor chip may contain secret information which identifies it and allows it to communicate with the outside world securely, such as a physical random function, or the secret key from a certified public key pair [143].

External memory and peripherals are assumed to be untrusted and therefore may be observed and tampered with. Generally, the operating system is assumed to be untrusted and, consequently, attacks by an OS or malicious software are deemed feasible. However, in certain implementations, part of the OS (the security kernel) may operate at a higher level than the remainder of the OS [143]. An adversary may also attack off-chip memory.

B.5.2 The AEGIS architecture

We now examine the security services Suh et al. specify as necessary for the provision of both tamper evident and tamper resistant environments. Interspersed with the descriptions of these necessary security services, potential mechanisms for their successful provision are described. Methods of implementing the secure execution environments are highlighted, both in the scenario where a security kernel is in existence within the TCB in conjunction with a hardened AEGIS processor chip; and also in the scenario where no security kernel exists, i.e. there is merely an untrusted operating system and a hardened AEGIS processor chip,

which alone constitutes the TCB.

In the first scenario the security kernel will operate at a higher level of protection than the rest of the operating system, in order to prevent attacks from untrusted parts of the OS, such as device drivers.

In the alternative scenario, where no security kernel exists, the processor needs to be aware of all processes running in AEGIS mode so that their states can be securely tracked. In this scenario, a secure context manager (SCM) is added to the hardened AEGIS processor, to ensure protection of secure processes.

- The SCM assigns a secure process identity (SPID) to each secure process, where a zero SPID represents regular processes.
- The SCM maintains a table for each process running in AEGIS mode containing:
  - The SPID of the program;
  - The program hash;
  - A field used to store register values;
  - A field used to store a hash for memory integrity verification;
  - A bit which indicates whether the process is in tamper evident or tamper resistant mode; and
  - A pair of keys for encryption (a static key and a dynamic key).
- A table entry is created by the initial enter_aegis instruction, and is deleted by the exit_aegis instruction.

The SCM table may be stored on the processor, but this will obviously restrict the number of processes that can be recorded in the table. Hence use of

virtual memory space to store the table, managed by the OS and stored in off-chip memory, is recommended, where a memory integrity mechanism, described below, is used to prevent a malicious OS tampering with the SCM table. A special on-chip cache is used to store SCM table entries for recent processes. When encryption keys and register values in the SCM table are moved off-chip, they are protected using a master key held in the processor.

In the scenario where a security kernel is present within the TCB, in conjunction with a hardened AEGIS processor chip, the security kernel completes the function of the SCM.

B.5.3 Tamper evident processing

A tamper evident environment does not provide for the privacy of code or data. Integrity protection is provided, however.

B.5.3.1 Additional instructions

In order to enable tamper evident processing, the instruction set described in table B.5 is defined.

Table B.5: AEGIS TE processing instructions

Instruction	Description
enter_aegis	Start execution in TE environment
exit_aegis	Exit TE environment
sign_msg	Generate a signature on a message and a program identity/hash with the processor's secret key

The valid execution of a program on a general purpose time sharing processor can be guaranteed, or a tamper evident environment constructed, by securing the program against three potential forms of attack:

- Attacks on the initial state;

- Attacks to on-chip caches or off-chip memory; and
- Attacks on state information when interrupts occur, or during context switching [143].

B.5.3.2 Protection of initial state

In order to guarantee that the initial state of a program is properly set-up:

1. The enter_aegis instruction is used to enter TE mode;
2. This instruction specifies a region containing stub code, which is used to generate the program hash;
3. This hash, when calculated, is then stored in protected storage for later use;
    - The stub code is executed directly after the enter_aegis instruction, and is responsible for verifying the hashes of any applications or data upon which the application relies, by comparing their hashes with hashes stored in the stub region;
    - The stub code also checks the parameters of the environment it is running in, i.e.
        - The processor mode it is running in;
        - The virtual address of the stub code; and
        - The position of stack

      must all be checked and validated.

This is summarised in table B.6.

Table B.6: AEGIS TE processing — protection of initial state

	Implementation containing a security kernel	Untrusted OS solution
Security kernel start-up	The kernel identity should be verifiable by a user, where a user can identify a TCB by the security kernel hash and the processor's public/private key pair. The processor computes the hash of the kernel at boot time. After this, the kernel's integrity is protected using the same methods used for the protection of other secure processes, i.e. Trusted VM management; Off-chip integrity verification.	N/A
Initial start-up of programs	Managed by the security kernel which ensures the initial state is correct.	The SCM implements the enter_aegis operation as a processor instruction. A hash of the program code and data are calculated. The hash is then stored in the SCM table. (In architectures such as x86, the initial stack pointer is checked to avoid stack overflow, should an interrupt occur).

B.5.3.3 Protection of state on interrupts

The integrity of program state must also be integrity protected during interrupt handling — see table B.7.

B.5.3.4 Memory

The integrity of on-chip caches and off-chip memory must also be protected against both physical and software attacks.

On-chip cache integrity

Table B.7: AEGIS TE processing — protection of state on interrupts

	Implementation containing a security kernel	Untrusted OS solution
Interrupts	Managed by the security kernel, which ensures that states are correctly restored after an interrupt.	The untrusted OS handles all aspects of multitasking. The processor must, however, verify that the a TE process's state is preserved when it is not executing, so the SCM stores all process register values in the SCM table when the interrupt occurs and restores them afterwards.

By virtue of the fact that on-chip caches are on-chip, they are implicitly secure from physical attack and therefore need only to be protected from buggy software — see table B.8.

Off-chip memory integrity

Off-chip memory is vulnerable to both physical and software attack. Therefore, the integrity of a block needs to be verified whenever it is read from off-chip memory.

Merkle trees or hash trees, see section 1.5.1, are used in order to verify the integrity of dynamic data in untrusted storage. Before a cache block is sent to off chip memory, it is hashed. A parent hash is then generated by hashing the concatenation of a set of level 2 (L2) cache block hashes. The tree root is stored in the TCB, where it cannot be tampered with. A separate hash tree is used to protect the virtual memory space of each TE process.

In order to verify the integrity of a node:

- The processor reads the node hash and its siblings from memory;
- Concatenates the values;

Table B.8: AEGIS TE processing — on-chip cache integrity

	Implementation containing a security kernel	Untrusted OS solution
Protection of on-chip caches	**Physical Attacks:** The processor chip is assumed to be tamper resistant, therefore on-chip caches are assumed safe from physical attacks. **Software attacks:** The security kernel protects on-chip caches from software attacks. Virtual memory protections and privileges are considered adequate to protect applications from each other. A virtual memory manager is included within the security kernel to protect the integrity of memory from malicious software.	**Physical Attacks:** The processor chip is assumed to be tamper resistant; therefore on-chip caches are assumed safe from physical attacks. **Software attacks:** On-chip caches are protected using SPID tags. When a process accesses an on-chip cache block, the block is tagged with the owner's SPID. This specifies the owner of the cache block. Each block will also contain the virtual address used by the owner process on last accessing the block. If a secure process later wants access to a cache block which requires integrity protection, the processor verifies the block before using it: If the active SPID = the SPID of the cache block; and the accessing virtual address = the virtual address of cache block, access is permitted.

- Checks the resulting hash matches that of the parent.

This process is repeated until the tree root is reached.

If, however, the entire memory space is protected, no sharing would be permitted between processes, and no input from an I/O device could be received. Therefore, a program should be able to access a part of memory with no integrity protection [143].

The solution involves the use of the most significant bit (MSB) of an address, which is used to indicate whether the integrity of the address should be protected or not. This implies that the upper half of virtual memory is protected and the lower half is not and, in view of this, the program must lay out its code and data appropriately. This static division of memory restricts processes to only half of the memory space for secure data, although this is not a problem for 64-bit architectures [143].

B.5.3.5 Trusting program execution results

In order that the results of a program's execution can be trusted, in spite of the fact that communication channels from a processor may be untrusted:

- The sign_msg operation is used (see table B.9);
- The signature of the security processor is provided on the message (results) concatenated with the identity/hash of the program from which the message originated.

If there is a security kernel within the TCB, the signature of the security processor will be calculated on the message concatenated with both the hash of the security kernel and the hash of the program from which the message originated.

Table B.9: AEGIS TE processing — sign_msg operation

	Implementation containing a security kernel	Untrusted OS solution
Signing operation	Sign_msg is implemented as a system call	Sign_msg is implemented as a processor instruction

B.5.4 Private tamper resistant processing

In order to allow for the provision of tamper resistant processing, the tamper evident environment described above may be extended to support private and authenticated operations [143].

B.5.4.1 Additional instructions

As described in table B.10, one new instruction, the set_aegis_mode instruction, is added to support this mode.

Table B.10: AEGIS PTR processing instructions

Instruction	Description
set_aegis_mode	Used to enable or disable the PTR environment from TE mode. When this instruction is called, a static key, concatenated with the protected program identity/hash, encrypted under the public key of the security processor, must be provided. The static key can only be decrypted by a particular AEGIS processor. It is then recorded as the static key for the protected program if and only if the encrypted program hash received matches the hash of the program decrypted by the AEGIS processor. If there is a security kernel, the input to set_aegis_mode must include the concatenation of the security kernel hash, the program hash, and the static key encrypted with the public key of the security processor. In this instance, the processor will only decrypt the static key if the security kernel hash matches the encrypted hash of the security kernel received. This static key may then be used to decrypt the accompanying application, as above.

In a PTR environment:

- All register values are considered private and protected;

- Whether instructions and data exported to external memory are confidentiality protected is dependent on the value of the second MSB of the address. Data stored in virtual addresses with the second MSB set are privacy protected.

B.5.4.2 Protection of initial state

The set_aegis_mode instruction is used to enable or disable privacy from tamper evident mode. The initial state is therefore validated when the initial enter_aegis instruction is called, as described in section B.5.3.2.

B.5.4.3 Protection of state on interrupts

In order to ensure the privacy of registers against software attacks when interrupts occur, the TCB saves the register values in private storage in the TCB, and then clears the registers before the untrusted interrupt handler starts — see table B.11.

Table B.11: AEGIS PTR processing — protection of state on interrupts

	Implementation containing a security kernel	Untrusted OS solution
Interrupts	Managed by the security kernel which ensures that states are correctly restored after an interrupt.	The untrusted OS handles all aspects of multitasking. As was the case in the TE environment, the SCM stores all process register values in the SCM table when the interrupt occurs, and restores them at the end. In addition, for a PTR process, once the values have been stored in the SCM table, the working copy is cleared so that interrupt handlers cannot see previous values.

B.5.4.4 On-chip/Off-chip memory

The TCB also protects on-chip caches and off-chip memory so that no process can read the private data belonging to another process.

On-chip Cache Privacy. Once again, since on-chip caches are on-chip, they are implicitly secure from physical attack and therefore only need to be protected from buggy software. The issues are summarised in table B.12.

Table B.12: AEGIS PTR processing — on-chip cache privacy

	Implementation containing a security kernel	Untrusted OS solution
On-chip caches	**Physical Attacks:** The processor chip is deemed tamper resistant, and therefore on-chip caches are assumed safe from physical attacks. **Software attacks:** The security kernel protects on-chip caches against software attacks. Virtual memory protections and privileges are considered adequate to protect applications from each other. Therefore a virtual memory manager is included within the security kernel to protect the integrity and confidentiality of memory from software attack.	**Physical Attacks:** The processor chip is deemed tamper resistant, and therefore on-chip caches are assumed safe from physical attacks. **Software attacks:** In PTR mode, accesses to private cache blocks are allowed if and only if the cache block SPID = the active SPID; and the active process is in the PTR mode. Otherwise the block is evicted from the cache and reloaded.

Off-chip memory encryption When data needs to leave the chip but remain privacy protected, it is encrypted by the TCB using symmetric encryption. Each process uses:

- A static key to decrypt instructions and data from the received program binary. It is obtained, encrypted under the processor public key, as input to the set_aegis_mode instruction. Following its initial use, it is used to encrypt instructions and data from the program binary when exported to

off-chip memory.

- A dynamic key to encrypt data generated during program execution. It is randomly chosen by the TCB when the enter_aegis instruction is called.

The authors recommend the use of AES as the symmetric encryption algorithm, although this is not compulsory. In both the implementation which contains a security kernel and in the solution which contains an untrusted OS it is advocated that a hardware encrpytion/decryption engine is deployed.

B.5.5 Conclusions

In this appendix, the IBM 4756, XOM and AEGIS architectures have been described.