A Cryptographically-Based Operating System Security Model That Protects against Privileged Attackers

Christian Payne

This thesis is presented for the degree of Doctor of Philosophy of Murdoch University, Western Australia.

11 December 2009
I declare that this thesis is my own account of my research and contains, as its main content, work that has not been previously submitted for a degree at any tertiary education institution.

____________________________________

Christian A. Payne
Abstract

It has long been recognized that widely used contemporary systems have relatively weak security and stronger operating system security models are required. In particular, the design of widely-used security models is such that the highest level of privileges available on the system are often highly exposed. If an attack is successful and the attacker attains a high level of privilege, all of the security mechanisms on the system may typically be bypassed.

Despite such limitations, weak models remain ubiquitous as more secure alternatives are complex and therefore harder to configure and audit for correctness. This is especially problematic when the user and administrator are the same person, as is often the case in widespread workstation environments. To be used effectively, security models must be simple enough to be easily conceptualised by users and consistent with their requirements.

Careful application of cryptography can often improve security. However, in the domain of operating system security to date, the use of cryptography has largely involved the creation of ad hoc, standalone mechanisms. Cryptographic filesystems exist to protect the confidentiality of data, but have little or no connection with existing access control theory. For example, some allow sharing of data between users but none provide all of the expected properties available in conventional, fully-fledged access control mechanisms such as secure and convenient revocation and prevention of authorisation transference. Filesystem integrity checkers use cryptographic hashes or digital signatures to ensure objects have not been modified, thereby protecting against substitution of malicious code. However, existing schemes lack the supporting infrastructure of an underlying security model to properly discriminate between verified and unverified objects. Furthermore, key management-related interdependencies between these different mechanisms have
not been recognized and, as a result, work in this area has so far progressed in a somewhat disjointed and piecemeal manner.

This research describes a new security model, known as Vaults, that utilise cryptography to provide improved security. In particular, the new model aims to be secure against an attacker who has achieved a high level of privilege on the system. Vaults provides a cryptographically-enhanced access control model that protects files from unauthorised read and write access. It also facilitates secure, authenticated sharing of data between users using semantics consistent with traditional non-cryptographic access control models.

The Vaults access control mechanism is supported by a flexible and convenient key management architecture that can be used for both file access keys and generic application secrets. Access to these values is controlled by a mechanism for cryptographically verifying the integrity of programs and the data objects with which they interact. However, unlike previous schemes, Vaults not only prevents execution of illicitly modified trusted code, but also assigns different privilege levels to verified and unverified processes. Furthermore, partially-trusted processes can be confined to specifically defined objects if required. This approach provides a mechanism for authenticated user interaction with security-critical system components and therefore represents a new interpretation of the traditional notion of a physical trusted path that can be extended to any appropriate object on the system. Finally, all of these mechanisms apply on both a global and local level, allowing administrators to create system-wide policies, and users to extend and refine these to suit their own security needs.

However this flexibility does not come at the cost of great complexity and the basics of using the scheme can be easily explained to users as they can be expressed using conceptually simple abstractions such as “locking files” and “sharing keys”. The use of cryptography in this way also serves to weaken the traditional association between privilege and identity, as access is permitted or denied based upon possession of the required token rather than the identity of the requesting process. Such a design
has the dual effects of constraining the powers of privileged users and lowering their exposure to attack by reducing privileges to a token, which is generally easier to protect than an identity.

After developing the model, a series of large-scale attack trees were constructed to analyse its security. The attack trees were used to both refine the design of the security model and also evaluate the assertion that the model retains its security properties when under attack by a user who has gained the ability to bypass the security kernel and directly access the secondary storage device. The results of this analysis demonstrate the advantages of applying cryptography to the problem of operating system security and show that the Vaults model is able to maintain its security properties in the face of attacks that are normally excluded 'by assumption' under existing computer security models. Vaults is therefore a novel and comprehensive model for integrating cryptography into the operating system in a manner that improves security, while remaining both flexible and usable.
Acknowledgements

In any project of this size, and that goes on for such a length of time, there are necessarily an enormous number of people who have contributed. Indeed, there are probably so many people who have assisted me in this research that it would be impossible to adequately thank them all. However, I would like specifically mention a number of people, without whom this work could not have been completed. Over the years this project has occupied me, it seems there has been an almost continuous series of external obstacles to overcome, beyond those actually part of the research itself. There have been many times I have wondered if I would ever complete this project. Without the following people, I feel certain that I would not have.

Firstly, I would like to thank Mark Reynolds, who was my first supervisor and, without whose advice and support I would never have made it through the early period of my candidature. Many thanks to Nicola Ritter, who provided support long before she became a supervisor, and continued this, along with a steady stream of insightful feedback, after she formally took on the supervisory role. Also, my immense gratitude to Fay Sudweeks for all of this — that which otherwise would not have been possible! I would also like to thank Peter Cole, whose support over the years, and particularly at critical times, has been essential to my being able to complete this work.

I am also immensely indebted to Terry Ingoldsby from Amenaza Technologies Ltd, who has been incredibly generous in supporting my research by providing use of the SecurITree software for the attack tree analysis. Put simply, there is no other software currently available capable of supporting an analysis of the magnitude performed and, without this generosity, the analysis work presented here would not have been possible. I would also like to thank Nuance Communications, the developers of the Dragon NaturallySpeaking software with which virtually all of this thesis has been written.

Finally, I would like to thank my family. In particular, my parents, who have supported me in everything right from the beginning and, without whom, I could
not have reached this point. Also, most especially, to my wife, Emily, and son, Nicholas, whose patience, love and support has been essential, now and always.
Previously Published Material

The following papers have been previously published or presented, and contain material based upon the content of this thesis:


# Contents

1 Introduction ........................................ 1
   1.1 Background ........................................ 1
      1.1.1 The Need for Stronger Security Models .......... 1
      1.1.2 An Integrated Cryptographic Approach .......... 2
   1.2 Research Objectives and Design Parameters .......... 3
      1.2.1 Research Objectives .............................. 3
      1.2.2 Goals, Principles and Assumptions ............. 4
   1.3 Overview of Vaults Security Model ................... 7
      1.3.1 Key Management .................................. 7
      1.3.2 Enhanced Access Controls ......................... 9
      1.3.3 Trusted Fingerprinting ........................... 12
      1.3.4 Advantages of Cryptography to Operating System Security .................. 14
   1.4 Significance of the Research ......................... 16
   1.5 Thesis Outline ...................................... 16

2 Computer Security and Cryptography ................. 18
   2.1 Introduction ....................................... 18
   2.2 Security Background ................................ 19
2.2.1 The Nature of Security ........................................ 19
2.2.2 Dimensions of Security ..................................... 20
2.2.3 The Nature of Insecurity .................................. 20
2.3 Computer Security ............................................. 21
2.3.1 Early Trusted Systems and Outcomes .................. 21
2.3.2 Security Models of Mainstream Systems ............... 23
2.3.3 Mainstream Alternative Privilege Models ............. 25
2.3.4 Mandatory Security in Mainstream Systems .......... 26
2.4 Overview of Cryptography .................................... 28
2.4.1 Relevance of Cryptography to Networking ........... 29
2.4.2 Use of Cryptography in Operating Systems .......... 30
2.5 Cryptographic Filesystems and Access Control .......... 31
2.5.1 Cryptographic Filesystems ................................. 31
2.5.2 Gifford’s Cryptographic Sealing ......................... 36
2.5.3 Other Cryptographic Access Control Models ........... 39
2.6 Secure Storage of Secrets .................................... 41
2.6.1 Secure Coprocessors ...................................... 41
2.6.2 Secret-Protected Processors ............................... 42
2.6.3 Plan 9 Factotum and Secstore ............................ 43
2.6.4 Contemporary ‘Trusted Computing’ ..................... 45
2.7 Cryptographic File Integrity Verification ................. 46
2.7.1 Tripwire ..................................................... 46
2.7.2 CryptoMark ................................................ 48
2.7.3 A TCG-Based Approach ................................... 51
2.7.4  I³FS  ......................................................... 52
2.7.5  General File Integrity Protection Problems  .............. 54
2.8  Summary  ....................................................... 54

3  Vaults Infrastructure  ............................................ 57

3.1  Introduction  .................................................... 57
3.2  Vault Types and Properties  .................................... 57
  3.2.1  User Vaults  .................................................. 58
  3.2.2  Global Private Vault  ....................................... 59
  3.2.3  Global Public Vault  ....................................... 60
  3.2.4  Escrow Vault  .............................................. 60
  3.2.5  Fundamental Vault  ....................................... 60
3.3  Vaults Public Key Infrastructure  .............................. 62
  3.3.1  The Need for a PKI  ....................................... 62
  3.3.2  VPKI Architecture Overview  ............................. 63
  3.3.3  VPKI Certificates  ......................................... 64
  3.3.4  User and Vault Creation  ................................... 65
  3.3.5  Vault IDs  .................................................. 67
  3.3.6  Prime User Certification of Key  .......................... 68
  3.3.7  User Certificate Verification  ............................ 69
  3.3.8  Revocation of VPKI Certificates  ........................ 70
3.4  Process Credentials and Vault Session Initiation ............... 72
  3.4.1  Process Credentials and Metadata  ....................... 72
  3.4.2  Process Trust Levels  ..................................... 73
  3.4.3  Session Initiation Process  ............................ 76

xi
3.4.4 Process Credentials Maintenance ......................... 78

3.5 Application Keys .............................................. 80
  3.5.1 Application Key Entry Format ............................. 80
  3.5.2 Creating Application Keys ................................. 82
  3.5.3 Application Key Release ................................ 84

3.6 Administrative Privileges .................................... 85
  3.6.1 The Design of the Prime User Account ................... 85
  3.6.2 Prime User Privileges .................................... 86
  3.6.3 Fundamental Privileges and Tickets ....................... 87
  3.6.4 Prime User Account Creation ............................ 89
  3.6.5 Prime User Security ..................................... 90

3.7 Summary ....................................................... 92

4 Cryptographic Enhanced Access Controls ...................... 94
  4.1 Overview ................................................... 94
    4.1.1 File Protection Modes ................................. 95
    4.1.2 File Protection Key Binding ............................ 96

  4.2 General Filesystem Issues ................................ 96
    4.2.1 Relationship to Identity-Based Access Controls ........ 96
    4.2.2 File Ownership ....................................... 97
    4.2.3 Protecting Directories and Their Contents ............ 97
    4.2.4 Moving and Deleting Files ............................. 98

  4.3 File Access Tickets ........................................ 99
    4.3.1 Ticket Security ...................................... 100
    4.3.2 Primary Tickets ..................................... 101
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.3</td>
<td>Secondary Tickets</td>
<td>102</td>
</tr>
<tr>
<td>4.4</td>
<td>Ticket and Key Metadata</td>
<td>104</td>
</tr>
<tr>
<td>4.4.1</td>
<td>The Key ID</td>
<td>105</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Primary and Secondary Ticket Metadata</td>
<td>107</td>
</tr>
<tr>
<td>4.4.3</td>
<td>GPRIV Protected Data</td>
<td>107</td>
</tr>
<tr>
<td>4.5</td>
<td>Verifying File Protection Status</td>
<td>108</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Filesystem Metadata</td>
<td>109</td>
</tr>
<tr>
<td>4.5.2</td>
<td>The Process Status Authenticator</td>
<td>111</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Protection Status Protocols</td>
<td>113</td>
</tr>
<tr>
<td>4.6</td>
<td>Ticket Creation Protocols</td>
<td>115</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Read Protecting an Object</td>
<td>115</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Write Protecting an Object</td>
<td>117</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Creating and Distributing Secondary Tickets</td>
<td>118</td>
</tr>
<tr>
<td>4.7</td>
<td>File Access Protocols and Related Issues</td>
<td>120</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Binding File Keys to Applications</td>
<td>120</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Accessing Protected Files with Tickets</td>
<td>122</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Issues Relating to Trust Levels and Bound Files</td>
<td>123</td>
</tr>
<tr>
<td>4.7.4</td>
<td>Primary Ticket Verification</td>
<td>124</td>
</tr>
<tr>
<td>4.7.5</td>
<td>Secondary Ticket Verification</td>
<td>125</td>
</tr>
<tr>
<td>4.7.6</td>
<td>Fundamental Ticket Verification</td>
<td>126</td>
</tr>
<tr>
<td>4.8</td>
<td>Other File Protection Protocols and Issues</td>
<td>127</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Revocation</td>
<td>127</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Removing Cryptographic Protection from a File</td>
<td>128</td>
</tr>
<tr>
<td>4.8.3</td>
<td>Ticket Escrow</td>
<td>128</td>
</tr>
<tr>
<td>4.9</td>
<td>Summary</td>
<td>130</td>
</tr>
</tbody>
</table>
5 Trusted Fingerprinting

5.1 Introduction ................................................................. 132

5.1.1 Goals of Trusted Fingerprinting ................................. 132

5.1.2 Overview of Process Execution and Fingerprinting ........ 134

5.2 Trusted Fingerprint Dependencies ................................. 135

5.2.1 Dependency Data Structures ........................................ 135

5.2.2 Dependency Types and Flags ....................................... 136

5.2.3 Dependency Trust Levels ............................................ 137

5.2.4 Identifying the Program Target and Library Dependencies . 139

5.2.5 Data File Dependencies .............................................. 139

5.2.6 Creating Dependency Records ..................................... 140

5.3 Fingerprint Matching .................................................... 141

5.3.1 Matching Techniques ................................................ 142

5.3.2 Path Matching ......................................................... 143

5.3.3 Multi-digraph Matching .............................................. 143

5.4 The Runtime Dependency Verification Model .................... 144

5.4.1 Runtime Digraphs .................................................... 144

5.4.2 Merged Runtime Digraphs ......................................... 145

5.4.3 Runtime Digraph Construction Conflicts ....................... 146

5.4.4 Practical Fingerprint Verification Issues ..................... 148

5.4.5 Runtime Verification of Programs ............................... 149

5.4.6 Impact of Dependency Flags on Execution .................... 150

5.4.7 Runtime Verification of Libraries ............................... 152

5.4.8 Runtime Verification of Data Files .............................. 154
5.4.9  Impact of Flags on the Runtime Data File Verification . . . . 155
5.4.10 Runtime Verification of Non-Native Code . . . . . . . . . . . 158

5.5  Fingerprint Updating . . . . . . . . . . . . . . . . . . . . . . . . 159
5.5.1  Manual Updating . . . . . . . . . . . . . . . . . . . . . . . . 159
5.5.2  Automatic Updating . . . . . . . . . . . . . . . . . . . . . . . 160
5.5.3  Automatic Update Restrictions . . . . . . . . . . . . . . . . 162
5.5.4  Installer Applications . . . . . . . . . . . . . . . . . . . . . . 163

5.6  Managing Process Trust . . . . . . . . . . . . . . . . . . . . . . . 163
5.6.1  Trusted Fingerprinting and Trusted Path . . . . . . . . . . . 163
5.6.2  The Trust Elevation Problem . . . . . . . . . . . . . . . . . 165
5.6.3  Limiting Process Trust Elevation . . . . . . . . . . . . . . . 166
5.6.4  Impacts of Variable Effective Trust Levels . . . . . . . . . . 167
5.6.5  Trust Levels of Non-PROGRAM Dependencies . . . . . . . . 169

5.7  Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 170

6  Security Analysis Background and Methodology 172

6.1  Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 172

6.2  Review of Security Analysis and Assessment Literature . . . . . 173
6.2.1  Petri Net-Based Approaches . . . . . . . . . . . . . . . . . . 173
6.2.2  Graph-Based Multistage Attack Modelling . . . . . . . . . . . 176
6.2.3  Attack Patterns and Profiles . . . . . . . . . . . . . . . . . 177
6.2.4  Security Metrics and Assessment Techniques . . . . . . . . . 178
6.2.5  General Threat Modelling Methodologies . . . . . . . . . . . 179
6.2.6  Argument Trees . . . . . . . . . . . . . . . . . . . . . . . .    180
6.2.7  Attack Trees . . . . . . . . . . . . . . . . . . . . . . . . . .    182
7.12 Trust Level Changes and Open Objects .......................... 225
7.13 Process Trust Elevation Restrictions and L1 Processes ............. 226
7.14 Summary .............................................................. 227

8 Attack Tree Results ..................................................... 229

8.1 Introduction ............................................................. 229

8.2 The Confidentiality Property Attack Tree ................................. 230

8.2.1 Overview .............................................................. 230

8.2.2 Summary of Unpruned Confidentiality Attack Scenarios ...... 231

8.2.3 Overview of Pruning Results ........................................ 239

8.2.4 The Manual Update Race Condition Attack ......................... 240

8.2.5 Specific Attack Instances ............................................ 248

8.2.6 Discussion ............................................................ 256

8.3 The Verifiable Integrity Property Attack Tree ............................. 260

8.3.1 Overview .............................................................. 260

8.3.2 Summary of Unpruned Verifiable Integrity Attack Scenarios ... 261

8.3.3 Pruning Results ........................................................ 266

8.4 Trusted Fingerprinting Attack Tree .................................... 268

8.4.1 Overview .............................................................. 268

8.4.2 Summary of Unpruned Trusted Fingerprinting Attack Scenarios 270

8.4.3 Pruning Overview ..................................................... 278

8.4.4 Trusted Fingerprinting-Specific Attacks ............................... 278

8.4.5 Verifiable Integrity Property Dependence ............................ 284

8.5 Observations on Attack Trees ........................................... 285

8.5.1 Subtree Libraries .................................................... 285
8.5.2 Tree Properties and Characteristics ......................... 287
8.5.3 Limitations of Pruning ........................................... 288
8.6 Discussion and Summary of Attack Tree Results ................. 289
  8.6.1 Summary of Attacks ............................................ 289
  8.6.2 Security against Highly Privileged Attackers ............... 291
8.7 Conclusion ............................................................. 293

9 Discussion and Conclusions ........................................ 295
  9.1 Summary of Research Results .................................... 295
    9.1.1 Vaults Security Features .................................... 296
    9.1.2 Security Analysis ............................................ 299
  9.2 Limitations of the Research .................................... 302
    9.2.1 Lack of Availability Protection ............................. 302
    9.2.2 Limitations of the Analysis ................................ 302
    9.2.3 Absence of an Implementation ............................... 303
  9.3 Directions for Further Research ................................ 304
    9.3.1 Distributed Applications of Vaults ......................... 304
    9.3.2 Assurance Advantages of Cryptography ..................... 305
    9.3.3 Further Research on Attack Trees ......................... 307
  9.4 Conclusion ............................................................. 308

A Glossary .............................................................. 310

B List of Symbols Used in Equations ................................ 322

C Attack Tree CD-ROM Guide ........................................... 325
  C.1 Directories .......................................................... 325
  C.2 File Formats .......................................................... 325

xviii
List of Figures

2.1 Cryptographic security feature interdependencies ..................... 56

3.1 Vault item categories icon legend ................................. 58

3.2 Overview of items stored in a user’s vault and their relationships ... 59

3.3 Role of the Fundamental Vault and architecture overview ............ 61

3.4 Abstract bindings created by the VPKI .............................. 63

3.5 Concrete VPKI bindings description .................................. 64

3.6 Vaults’ hierarchical process trust levels .............................. 74

3.7 Application Key label structure ....................................... 81

4.1 Simplified ticket/key/file relationships ............................... 100

4.2 Ticket and key metadata overview .................................... 106

4.3 Referencing the Protection Status Table with the Protection Status Authenticator ................................................. 111

4.4 Overview of Escrow vault operation ................................... 129

5.1 Overview of process execution and fingerprinting ..................... 134

5.2 Example of separate global and local digraph before merging ........ 147

5.3 Merged runtime digraph .................................................. 147

5.4 Matching and verification for POSIX exec() system calls .......... 151
8.9 “Perform manual update race condition attack” subtree ............ 245

8.10 Pruned “Defeat confidentiality property” attack tree with MURCA
subtrees highlighted ................................................... 246

8.11 Pruned subtree showing attacks for malicious code executing at trust
level L3 ......................................................................... 249

8.12 Pruned subtree showing attacks for subverting VID assignment ... 251

8.13 Pruned subtree for when an attacker has a ticket mistakenly issued
to them ...................................................................... 253

8.14 Pruned subtree showing use of malcode to leak read-protected file
contents ........................................................................ 254

8.15 Pruned subtree for stealing a usable primary ticket ................. 255

8.16 Pruned subtree for stealing a usable secondary ticket .......... 257

8.17 Pruned subtree for obtaining a ticket through the escrow mechanism 258

8.18 Properties of the “Defeat verifiable integrity property” attack tree .. 260

8.19 Top-level view of the “Defeat verifiable integrity property” attack tree
with expanded left-most branch ....................................... 262

8.20 Subtree of the “Defeat verifiable integrity property” attack tree showing
attacks where the modification bypasses the kernel interface ... 265

8.21 Properties of the “Defeat Trusted Fingerprinting” attack tree .... 269

8.22 Top-level view of the “Defeat Trusted Fingerprinting” attack tree
with undetected modification branch partially expanded .......... 271

8.23 Subtree from the “Defeat Trusted Fingerprinting” attack tree showing
attacks where privileges incorrectly assigned to an executing program 274

8.24 Subtree from the “Defeat Trusted Fingerprinting” attack tree showing
attacks where unauthorised code accesses an application key ...... 275

xxii
8.25 Subtree from the "Defeat Trusted Fingerprinting" attack tree showing attacks where a trusted program is subverted by its parent . . . . 276

8.26 Pruned subtree for attacks involving undetected modification of a fingerprint object .............................................. 279

8.27 Pruned subtree for attacks involving privileges being incorrectly assigned .......................................................... 282

E.1 Fingerprint matching algorithm ................................................. 330
List of Tables

3.1 Comparison between Fundamental Vault decryption key storage mechanisms ........................................ 62

3.2 Overview of Vault structure on disk .................................................................................................. 78

4.1 Protection status-related filesystem metadata .................................................................................. 110

4.2 List and summary of Protection Status Table fields .......................................................................... 112

4.3 Protection mode and access request permutations and behaviour .................................................. 122

5.1 Summary of dependency flags .......................................................................................................... 137

5.2 Dependency flag details and defaults .................................................................................................. 138

5.3 Key to identifiers in the trust elevation restriction algorithm .......................................................... 168

8.1 Statistics on leaf node possibility and Assumption 5 ....................................................................... 291

C.1 File formats of attack tree presented on the CD-ROM .................................................................... 326

E.1 Key to identifiers in the matching algorithm ..................................................................................... 329
Chapter 1

Introduction

1.1 Background

1.1.1 The Need for Stronger Security Models

At the same time as societies and economies are becoming ever increasingly dependent upon information technology, a constant stream of new computer security vulnerabilities is being discovered [2,3]. Despite extensive development of mechanisms to secure applications and networks, it has been recognised that the security models utilised by contemporary operating systems are relatively weak and stronger models are required to underpin even securely designed applications [4].

The privilege model found in the Unix operating system is a prime example of this problem. It involves a two-tier approach consisting of a single, privileged user account (the superuser or ‘root’), with all other users having no special privileges and being essentially limited to their own files and those shared by other users. This concentration of privilege with a single, ultimately trusted user identity makes the superuser account a primary target for attackers. However, it also has the effect of rendering the account almost impossible to defend since superuser privileges are required for even the most basic administrative tasks. The result is a proliferation of privilege boundaries by which attackers can seek to gain superuser access. This
design property of Unix has been acknowledged as the principal security problem of the system [5].

Nonetheless the Unix model, and other similarly weak models, are widespread, both due to historical inertia and because their simplicity, compared to more secure alternatives [6, 7], makes them more attractive to users. In addition, more secure approaches are also often problematic because their conceptual complexity makes policy correctness difficult to verify [8–10]. Because of these issues, it is difficult to envisage such models becoming widely accepted in environments where security is not an overriding priority. Instead it has been argued that “users need a clear model of how the security operates (if not how it actually provides security) in order to use it well” [11, p. 3] but a conceptually simple model that provides strong security has so far proved elusive. Nonetheless, the extent of the security problem requires that alternative and novel approaches be explored.

### 1.1.2 An Integrated Cryptographic Approach

This research describes the development of a new security model called *Vaults* that utilises cryptography to provide a non-identity-based access control scheme that is integrated with a code integrity verification and privilege management mechanism. Despite cryptography being widely used in the domain of network security for communication and authentication tasks, it has seen only limited application in the area of operating system security. Cryptographic host security tools to date have been rather piecemeal, having been developed on an *ad hoc* basis with the aim of resolving a single, isolated security issue. For example, cryptographic filesystems protect the confidentiality of data in the event of a machine such as a laptop being lost or stolen, but do not provide the properties expected of generic access control mechanisms. Similarly, while stand-alone file integrity checking tools exist to detect and mitigate the risks of intrusion, they struggle to deal with practical concerns such as updates to frequently modified objects and do not discriminate between the privileges of verified and unverified code. One important role that integrity veri-
fication tools could play is in the area of key management to ensure that securely
stored secrets are only released to authorised, authenticated code. However, this
requires taking an integrated, holistic view point that recognizes the interdepen-
dence between the key management infrastructure required to support a flexible
cryptographic access control scheme, and a code integrity verification mechanism
that prevents the compromise of both keys and files by malicious software. The
Vaults security model described and evaluated in this thesis has been developed
upon such a basis.

More generally, cryptography provides advantages that conventional purely logic-
based mechanisms cannot. A security mechanism that depends solely on active
enforcement of security policy by a reference monitor will inevitably fail if the mon-
itor can be bypassed. In practice, attackers can often obtain an elevated privilege
level on the system and thereby bypass the enforcement mechanism [5,12]. For ex-
ample, an attacker who gains the ability to bypass the operating system kernel and
directly access the secondary storage device can read or modify trusted system files
at will. By comparison, employing cryptography enforces certain additional limits
that are not otherwise achievable. For example, an encrypted file remains secure
even if the attacker can bypass the security kernel and read the ciphertext directly
off the disk. Additionally, if a highly privileged attacker makes unauthorised mod-
ifications to security-critical code, these modifications can be detected through the
use of cryptographic integrity verification. These kinds of security guarantees can-
not be obtained with conventional active security mechanisms. Consequently, one
of the central security goals of the Vaults model is to maintain its stated security
properties even against a highly privileged attacker.

1.2 Research Objectives and Design Parameters

1.2.1 Research Objectives

The objectives of this research are as follows:

1. to develop a comprehensive computer security model that leverages crypto-
graphic techniques to provide security advantages over traditional identity-based models while being more intuitive than existing mandatory security-based alternatives;

2. to demonstrate the security advantages of applying cryptography in an integrated way to operating systems; and

3. to analyse the above using a threat modelling approach to evaluate the level of security it achieves and, in particular, to demonstrate that the model is secure, even against highly privileged attackers.

1.2.2 Goals, Principles and Assumptions

Before proceeding, it is important to identify explicitly the security goals for the Vaults model, the principles according to which it has been designed, and some general assumptions made regarding the security it provides.

Security Goals

The overall security goal of Vaults is to maintain a defined set of security properties in the face of an attacker who has achieved a high level of privilege equivalent to Unix superuser access. This goal is achieved through the use of cryptography in the model and thereby mitigates this underlying problem in existing security models for systems such as Unix.

Specifically, the confidentiality and verifiable integrity of protected objects on the system is guaranteed within the security limits of the relevant cryptographic primitives and algorithms. Note that, while the confidentiality guarantee is absolute in the sense that an encrypted file cannot be decrypted and viewed without access to the relevant cryptographic access token, the integrity of a protected object is only guaranteed to be verifiable. Therefore the illicit modification of an object is not guaranteed to be prevented. The changes, however, are guaranteed to be detectable by those with legitimate access to that object. Furthermore, the user is shielded
against negative impacts resulting from inadvertently utilising an illicitly modified object. On first inspection, this implies that Vaults achieves a lower level of protection than traditional access control models, which actively prevent unauthorised modification of designated objects. However, Vaults access controls are intended to be layered on top of the existing access control scheme so the minimum level of security achieved is that provided by the existing scheme. Conversely, even if all forms of active protection failed completely and a highly privileged attacker were able to write directly to the disk to modify cryptographically protected files, these modifications would be guaranteed to be detected and the user shielded against any deleterious effects.

The security goal for subjects\footnote{The term ‘subject’ is widely used to mean both an active entity such as a user or a process executing on the user’s behalf [13, p. 108], [14, p. 28], as the distinction between these two is rarely important. However, in this context, the term is used specifically to refer to processes rather than users although users are also authenticated when accessing the system.} is that designated code to be executed will be cryptographically authenticated, as will designated data objects accessed by this code. Based upon this authentication, subjects will be assigned a trust level and this label, along with the identity of the code, will be used to determine that subject’s access to cryptographically protected objects. Further details of the security goals of the Vaults model necessary to support the evaluation of these properties are given in Section 6.3.2.

**Design Principles**

Saltzer & Schroeder [15] have previously identified general principles of secure design and these apply extensively to different aspects of the Vaults model. However, in addition to these general principles, the model has been developed upon the novel principle that all security decisions are to be made based on cryptographically derived or protected parameters. For example, access control decisions are not made based on identity but rather on the cryptographic tokens available to the process. Amongst other improvements to security, this approach has the effect of constraining the power of the superuser, which is normally based entirely on the
identity of the subject. Also, as previously indicated, the trust level assigned to each process is based on a cryptographic authentication of its code and dependencies.

Existing widespread security models are well entrenched and the alternatives developed so far have seen little success in displacing them. Consistent with this observation, Vaults has been designed to be backwardly compatible with the identity-based models currently in use so that it can be layered on top of the existing models to provide ‘defence-in-depth’. To reflect the practicality of this new approach, where appropriate, the description of the model given here utilises Unix as an example operating system upon which it could be implemented. Also, a number of issues relevant to the integration of Vaults with Unix are discussed where relevant. However, the underlying principles and design of the model are not dependent upon the Unix architecture. The Unix-specific adaptations described here serve primarily to demonstrate the model’s ability to be adapted to specific access control contexts.

Assumptions

The following general assumptions have been made regarding the security provided by Vaults.

**Kernel isolation.** The kernel is considered to be isolated from direct interference at runtime, even from a highly privileged attacker. In particular, internal kernel memory is assumed to be protected and cannot be accessed even by privileged users. Memory belonging to other processes is similarly protected. Since it is assumed that a privileged attacker may modify data stored on the disk, the existence of a secure bootstrap mechanism to ensure kernel integrity at bootup is assumed. This assumption encompasses the requirement for a suitable hardware infrastructure as necessary to support these abstracted features.

**Correct implementation.** It is assumed that all relevant components are correctly implemented and contain no errors of any kind that impact on the
security of the system. That is to say, the issue of implementation flaws and code assurance are considered to be out of the scope of the security provided by Vaults. Other relevant practical prerequisites for security, such as sufficient available entropy for key generation whenever required, security against side-channel attacks [16], and an adequate level of physical security, are also assumed to be satisfied\textsuperscript{2}.

**Secure cryptography.** Suitable cryptographic algorithms for the relevant primitives are assumed to be available and secure in design and implementation. Also, while some low-level cryptographic implementation issues are considered throughout this document, this section is not intended to be a comprehensive or exhaustive treatment of such issues. Therefore, it is assumed that suitable encryption modes are available and would be selected for implementation where appropriate [17].

**Acceptable performance.** While efficiency has been a consideration in the design of the Vaults model, the primary focus of the design is on attaining a high level of security and, where deemed appropriate, conservative design decisions have been made in the interests of achieving this. Furthermore, any use of cryptography involves an unavoidable concomitant reduction in performance as a consequence of increased security. It is therefore assumed that suitably efficient cryptographic algorithms, and hardware capable of achieving an acceptable level of performance, are available as required.

### 1.3 Overview of Vaults Security Model

#### 1.3.1 Key Management

At the core of the Vaults security model is the provision of a secure repository for storage of security-sensitive data and a collection of straightforward rules for controlling access to the data. The repositories are referred to as *vaults*, which is

\[\text{This is not intended to be a complete list of such issues.}\]
also an appropriate name for the model as a whole because of the significance of
these repositories to its security. The data stored in the vaults is maintained in
memory by the security kernel and is cryptographically protected when written to
the disk.

The vaults themselves are used in several ways for different purposes as outlined in
the following sections. However, their simplest application is as a general repository
for secret values utilised by various applications. These are known as application
keys and can include normal passwords and passphrases, as well as cryptographic
keys of all types. Each user has their own vault for storing their application keys,
and trusted programs that they execute can gain access to the specific keys they
require. This controlled access is achieved by the application requesting a key from
the kernel, which maintains the security of this data by only releasing keys to pro-
cesses that have been cryptographically authenticated and specifically designated
as authorised to access that particular key. This approach effectively locks out
malicious code and isolates applications from security failures in other trusted pro-
grams. Note that this is one of the very few aspects of the Vaults model that is not
entirely backward compatible, as it requires the applications using these keys to be
aware of the architecture in order to issue requests to access them. However, this
request mechanism is designed so as to require only very minor changes to existing
code.

The application keys mechanism has both usability and security advantages. Users
no longer need to memorise numerous, complex and potentially confusing passwords
as they may be stored in their vault and automatically retrieved when required after
a once-per-session authentication process. Not only is this more convenient for the
user but it also encourages the selection of high entropy passwords that would oth-
erwise be too difficult to memorise. Additionally, eliminating the need to memorise
multiple passwords makes it more practical for users to select different passwords
for different purposes, thereby maximally limiting the damage in the event of an
individual password being compromised. Therefore, storage of application keys in
vaults allows users to maintain their passwords and other keys in a single, convenient location and make them securely and transparently available to authenticated, trusted applications when required. However, this is by no means the only—or indeed most significant—role played by the vault structures. The vaults represent generic key management mechanisms that enable a number of cryptographic security enhancements to be deployed in an integrated and comprehensive manner. The remainder of these features will now be outlined briefly.

1.3.2 Enhanced Access Controls

The ability to securely store and manage cryptographic keys within an operating system has an obvious application in supporting the construction of a cryptographic filesystem. However, the file encryption mechanism in the Vaults model does not simply encipher data before writing it to the disk. Instead, it is a complete generic access control model that maintains the properties expected of traditional, non-cryptographic access control schemes while providing improved security. In contrast to previous cryptographic filesystems, the Vaults’ enhanced access control architecture provides the following properties.

**Independent protection modes.** Objects can be both read-protected, by encrypting the data, and independently write-protected. By comparison, most previous cryptographic filesystems behave primarily as integrated file encryption tools with no independent support for restricting modifications to file contents.

**Data sharing.** Users can cryptographically protect objects and then grant access to other specified subjects (users) as desired. Again, the mode for shared access can be independently specified as being read, write or both.

**Revocation.** Users who grant others access to their files may revoke this at any time. This can be performed without requiring any computationally expensive
processing, such as re-encryption of file data, as users never gain direct access to cryptographic keys.

**Universality.** Protection can be applied to any object that the user owns and is not restricted to files within a specific directory or filesystem. This also allows cryptographic access controls to be applied to important system files.

**Non-transferability of privilege.** Under some previous schemes, such as some of those discussed in Section 2.5, a user who has been granted access to a cryptographic key is able to duplicate that key and transfer it to other parties at will. However, this is not possible with Vaults. Therefore, a subject who has been granted access to a file object belonging to another cannot transfer these access privileges to a third party, either before or after revocation.

**Identity independence.** Unlike most traditional, non-cryptographic access control models, access to data under Vaults is not dependent on the subject’s identity but rather on the values contained within their vault that are cryptographically determined. This effectively places limits on privileged users and represents a step away from widely used identity-based access control paradigms that have proved to be weak.

**Simple conceptual model.** The access control scheme is highly amenable to abstraction and can be easily explained to users in simple terms such as “locking files” and “sharing keys” with other users. Use of the scheme does not require any understanding of complex security models or cryptographic techniques.

Upon protecting a specific file that they own, a user will receive a cryptographic token known as a primary ticket. This ticket is then stored in their vault and used to subsequently access the object. The owner can then use the primary ticket to construct additional tickets, known as secondary tickets, which are given to those users to whom access is to be granted. The structure of the secondary ticket ensures that it cannot be duplicated, transferred to another party, forged or converted into a primary ticket by a malicious user, and the owner can revoke the ticket at any
time, rendering it useless. In the case of both read and write access, a cryptographic
key is generated and used to protect the file. This is achieved either by encrypting
it or by generating a message authentication code (MAC) to verify its integrity in
the case of write access. This key is maintained by the kernel and not released to
the users who access the file.

**Limiting the Privileges of the Superuser**

The generality of the Vaults’ enhanced access controls has the effect that the privi-
leges available to a process now involve two separate parameters. As previously, the
identity label, such as the UID/GIDs under Unix, with which the process executes,
still determines its privileges according to existing permissions. However, under the
Vaults model, access to an object now also depends on the tickets available in the
vault to which the process has access. This latter aspect is both cryptographically
determined and independent of identity. In effect, a ‘locks and keys’ access con-
trol model [18, p. 396–400] is layered on top of the existing identity-based access
control scheme, thereby limiting the privileges of the traditional superuser account.
While previously the superuser was able to bypass access control restrictions, this
is no longer the case. Access to a protected file object will not be granted without
possession of the appropriate ticket, irrespective of the identity making the request.
Therefore, a new privileged account, known as the prime user, exists which is able
to bypass these access controls when necessary and within tightly defined limits.
The prime user is also responsible for administrative matters relating to the Vaults
security model but does not have the broad range of privileges or responsibilities
of the traditional superuser, thereby limiting the ways through which the account
may be attacked. Partially separating privilege from identity in this way has some
of the security advantages associated with mandatory access controls, in that the
cryptographically enhanced access controls provide a strong security baseline that
cannot be arbitrarily violated by privileged attackers. Consequently, this significant
weakness in security models, such as those used in Unix, is largely resolved. How-
ever, Vaults’ enhanced access controls remain essentially a discretionary scheme, as access privileges are determined and specified by individual users on the basis of their individual security requirements. While this may not be suitable for implementing certain policies where centralised control is critical, it is consistent with the flexibility expected of (and currently found) in mainstream systems, albeit with significantly improved security. Furthermore, this is achieved without the imposition of a conceptually complex or difficult to manage security model.

1.3.3 Trusted Fingerprinting

With widespread threats of a malicious code and intrusion [19, 20], it is increasingly important to protect the integrity of security-critical programs on a system. This is particularly so where these programs may access sensitive data stored in a user’s vault. Trusted Fingerprinting is the mechanism provided by the Vaults model to verify the integrity of programs to ensure that they have not been modified, and thereby protect users from malicious code. The mechanism works by generating one-way hash values of security-relevant code, with these values stored in a vault to ensure their integrity and authenticity. When these programs are subsequently executed, their hash value or ‘fingerprint’ is verified and execution denied if the code has been changed. Furthermore, as the behaviour of a program can be influenced by external objects—such as shared libraries, configuration files and even other programs that interact with it at runtime—these objects are similarly verified.

Local and Global Trusted Computing Bases

Using the Trusted Fingerprinting mechanism, a security administrator can fingerprint security-critical programs that are part of the system’s trusted computing base (TCB) and the integrity of this code will be verified upon execution by all users. This mechanism means that the user is interacting with TCB code that has been cryptographically authenticated, similar to how a traditional physical trusted path mechanism provides the user with a degree of certainty as to the authenticity of
the code with which they are interacting. However, individual users also have code that is relevant to their own security requirements and so each user can fingerprint these programs and have them verified upon execution. This creates two tiers of TCB: a global TCB (GTCB) determined by the security administrator that applies equally to all users; and independent local TCBs (LTCB) that apply specifically to each user, reflecting their individual security requirements.

**Process Trust Levels**

Current real-time code integrity verification mechanisms have no means for discriminating between verified and unverified code, and therefore create the opportunity for redirection attacks to be mounted unless all objects on the system are verified [21–23]. However, universal verification is often impractical, especially with respect to objects that are frequently modified. This issue is compounded when dealing with various non-executable data objects that are not necessarily verified and can influence otherwise verified trusted code at runtime. To address these problems, Vaults assigns a trust level to every process and this dictates its ability to access the user’s vault. Programs without a fingerprint are designated as L0 and—as these are regarded as untrusted—they have no access whatsoever to the user’s vault. This restriction further secures cryptographically protected files and application keys by preventing access from potentially malicious, unverified processes. However, some processes require access to a limited number of cryptographically protected objects but do not need to be granted full vault access. These are designated as L1 and, although they are fingerprinted and verified on execution, their access is restricted to specifically designated tickets. This trust level allows programs that are exposed to extensive untrusted data to be confined to a minimum level of privilege and isolated from other sensitive data. Fully trusted applications are designated as L2 and can access all tickets in the user’s vault. A very small number of programs are designated by the security administrator as L3 and this trust level is reserved exclusively for vault administrative tasks. In contrast to previous
schemes, this simple, tiered design allows the privileges of verified and unverified processes to be distinguished. Under the Vaults Trusted Fingerprinting model, a fail-safe default applies such that, even if the user is redirected to access malicious code that will not be verified, this code is flagged as L0 and has no access to any cryptographically protected objects or keys. Similarly, a simple mechanism exists to allow updates to cryptographically verified data objects where these modifications are made by authorised, trusted code.

1.3.4 Advantages of Cryptography to Operating System Security

In the past, cryptography has been used most extensively in the domain of network security, whereas computer security models have typically depended upon a specific set of rules required to implement the desired policy [18,24]. Examining the reasons for this difference in approaches is instructive and identifies how cryptography can benefit operating system security. A fundamental problem in network security involves two or more parties communicating over an untrusted channel where attackers may either passively eavesdrop or actively manipulate the messages exchanged. Here, cryptography is used as the primary mechanism for facilitating secure communication over an untrusted medium. Within the limits of existing protocols, algorithms and implementations, cryptography provides a degree of mathematical certainty as to the security of these messages and effectively excludes a range of possible attacks. In contrast, the computer security approach of labelling subjects and objects, and implementing logic to enforce the desired policy according to these labels, would be unsuccessful if applied to network communication since attackers are not bound by these rules and would simply ignore them.

However, labelling is successful in computer security as long as the attacker cannot subvert the system by achieving a sufficiently high level of privilege to bypass the reference monitor that enforces the security policy. Unfortunately, the security models used in mainstream systems mean that such security failures are difficult to prevent [5]. In contrast, a security model based upon cryptography rather than
arbitrary rules allows fewer assumptions to be made concerning the level of privilege that needs to be achieved before security policy can no longer be enforced. For example, an attacker who is able to bypass the security kernel and gain direct access to the disk can ignore the access control labels of objects stored there, and this is analogous to the network scenario described above. However, if these objects are cryptographically protected then, given a well-designed security model, the attacker will be unable to compromise the policy. An additional related benefit is that the cryptographic nature of the protection reduces the dependence of security on enforcement logic, thus both reducing its complexity and the impact of any failures in it.

Use of cryptography for computer security models also has other potential advantages. For example, it is the only efficient and realistic way of performing integrity verification to guard against privileged intrusion and malicious code substitution [25]. Cryptographic tokens also represent a convenient mechanism for breaking the link between the privilege of a subject and the subject’s identity. For example, an attacker who achieves superuser privilege but does not possess cryptographic tokens that grant higher privileges would have their privileges constrained. This advantage also flows down to ordinary users, although the overall impact on security is less significant. Furthermore, basing high levels of privilege on the possession of specific tokens (objects) rather than the identity of the subject protects these administrative accounts from attack by reducing their exposure. On conventional systems, every instance of the superuser identity represents an opportunity for attackers to find and exploit security flaws that will grant them unlimited access to the system. In contrast, a cryptographically protected token denoting higher privileges that rarely needs to be accessed has far fewer such privilege boundaries. These issues will be explored later on in the thesis.
1.4 Significance of the Research

This thesis describes a number of novel and significant contributions to the field of computer security. These contributions can broadly be categorised into two major points.

1. This thesis identifies and evaluates the advantages of employing cryptography as the basis of an operating system security model.

It is argued that cryptographically-based operating system security models provide a number of advantages over purely logic-based approaches. The use of cryptography allows the models to provide security properties that are normally either very difficult or impossible to achieve. In particular, it is claimed that cryptographic models are capable of maintaining their stated security properties, even when under attack by a privileged attacker. The security properties achieved by the model that has been developed are described in detail and their security, and success in meeting these goals, evaluated.

2. This thesis demonstrates the success of employing large-scale attack trees for analysing a security model.

The use of several attack trees to analyse the security of the cryptographically-based security model developed is described. These attack trees are exceptionally large, with the two largest both containing over 160,000 nodes each. These trees were used to rigorously evaluate the success of the model to meet its stated security properties in the face of a privileged attacker and were also used to inform the design of the model during its development in order to improve its general robustness.

1.5 Thesis Outline

The remainder of the thesis is organised as follows. Chapter 2 provides a literature review describing the current status of computer security and the limitations of existing models. It also examines some relevant uses of cryptography, focusing
in particular on existing examples of its application to operating systems. The properties of cryptography that make it beneficial when applied to OS security are also discussed in detail.

Chapter 3 describes some of the underlying cryptographic infrastructure used by the model including: the specific types of vaults it provides and their purposes; the mechanism by which applications can transparently access passwords stored in vaults; the handling of administrative privileges; and a lightweight public key infrastructure for inter-user authentication.

Chapter 4 describes the cryptographically enhanced access control model utilised by Vaults and details how users are able to grant and revoke access to the objects they own. Chapter 5 describes the Trusted Fingerprinting mechanism used to cryptographically verify the integrity of executed code and to constrain the trust assigned to this code as appropriate.

Chapter 6 reviews the literature on security analysis and evaluation techniques, focusing in particular on threat-oriented methodologies suitable for elucidating the capabilities of a privileged attacker with respect to the Vaults model. It also details the specifics of the attack tree-based methodology selected for use, defines the security properties that were analysed and outlines the assumptions employed to limit the scope of the analysis.

Chapter 7 describes the attacks identified by the analysis while the trees were still being constructed and discusses in detail how the model was adapted to address these issues. Chapter 8 then presents the results of the attack tree analysis and the nature and prerequisites of the attacks identified, considering in particular the ability of the Vaults model to maintain its stated security properties in the face of a privileged attacker.

Finally, Chapter 9 examines the advantages of the cryptographically-based security model and considers the security of the Vaults model in particular. A range of opportunities for future work in exploiting other advantages of cryptographically-based OS security models is also discussed.
Chapter 2

Computer Security and Cryptography

2.1 Introduction

This chapter examines the literature relating to the security model being developed in this thesis. In the first half of the chapter, security models of mainstream systems are considered, as well as schemes to improve the security of such systems. Following on from this, an overview is given on issues relating to use of the cryptography, both for networking and operating systems. Various mechanisms that employ cryptography to enhance operating system security are described in detail. These include cryptographic filesystems and access control models, mechanisms for securely storing secret values, and schemes for cryptographic verification of filesystem objects. Interdependencies between the various weaknesses and limitations of these different types of operating system-oriented cryptographic mechanisms are identified and discussed, pointing the way towards the development of a model that resolves these problems.

It should be noted that a further discussion of the literature is provided in Chapter 6, focusing on reviewing techniques and methodologies for analysing and assessing security models. This latter discussion is used to identify the most appropriate
methodology for evaluating the Vaults model described in this thesis and is not directly connected with the literature underpinning the design of the model itself.

2.2 Security Background

2.2.1 The Nature of Security

While security is, to some extent, a relatively intuitive concept, defining it precisely is often problematic. This is evidenced both by the variety of definitions that exist and the fact that many security texts avoid explicitly defining the term at all [3, 24, 26–33]. Even Bishop’s [18] comprehensive 1000 page text on computer security does not attempt to present an all-encompassing definition. Early definitions, such as that in Russell & Gangemi’s [13] introductory text, often consider physical assets and infrastructure while generally being information-centric and heavily biased towards confidentiality issues [34]. More recently, security has been defined by distinguishing it from the property of reliability. Specifically, reliable code should do everything required, while secure code should do everything required and nothing else [35]. This definition requires secure code to behave correctly in response to all possible inputs that might occur in an explicitly hostile environment, not just those that are likely or plausible in a benign environment. An example of this is a buffer overflow attack where the ‘shell code’ and offset pointer injected into the program could only ever be regarded as malicious data and would never occur coincidentally [36], [3, Ch. 7].

While Arce’s [35] definition is simple and abstract, it does not identify the correct behaviour of secure code which is a prerequisite for secure design [28, 37, 38]. More importantly it cannot be used to identify the presence of security vulnerabilities or confirm their absence. This point highlights why security is often so difficult to define and achieve since, as Saltzer & Schroeder [15] observed, proving the absence of security flaws in a design or implementation is extraordinarily difficult.
2.2.2 Dimensions of Security

As a result of the difficulty in arriving at a suitable definition, security is often considered in terms of several non-overlapping properties variously referred to as ‘components’, ‘goals’, ‘aspects’ or ‘dimensions’ of security. Of these, the term ‘dimensions’ will be used in this document as it is the least ambiguous.

The dimensions most commonly considered are confidentiality, integrity and availability [5,13,18,26]. Confidentiality refers to the requirement that data, either in whole or in part, must not be viewed by any unauthorised parties. Depending on requirements, sometimes this extends to protecting additional metadata (such as file names and sizes) that may reveal information about the data itself. The integrity dimension requires that data cannot be modified by an unauthorised party or modified in an unauthorised way by those with legitimate access. Sometimes the term is used to refer to the trustworthiness or authenticity of data [18] but more often simply means modification [39]. A subset of integrity is ‘verifiable integrity’ where unauthorised modifications are not necessarily prevented but are guaranteed to be detected. The availability dimension concerns ensuring that computer resources are accessible to a specified service quality level. In practice, this can be easy to achieve within individual systems where centralised control may be enforced [5, Ch. 25] but is far more difficult to guarantee in distributed systems [26]. As a result, Needham [40] has argued that availability does not always receive the same attention as the other two dimensions.

2.2.3 The Nature of Insecurity

Ever since the recognition of the importance of computer security in the 1970s, vulnerabilities in software have represented a serious security problem [2,3,41]. Analyses of computer security trends over time show increasing sophistication of attacks and exploitation of a variety of vulnerabilities in both the design and implementation of critical and widely used systems [19,42]. Such failures in application software mean the weight of security responsibility falls heavily on to the operating
system [4]. Furthermore, malicious software or ‘malware’ is increasingly widespread and financially damaging [43, 44]. In an especially worrying trend, one survey reported a very high level of Trojan horse or ‘rootkit’ software that typically allows an attacker to bypass operating system security measures and covertly re-gain direct access to the system [20].

Such vulnerabilities occur due to difficulty in ensuring secure design and implementation of software [15, 45, 46]. Researchers have examined the properties of vulnerable software and attempted to identify where such vulnerabilities can occur. For example, Manadhata & Wing [47, 48] describe the notion of an ‘attack surface’ of a system as consisting of those system actions and resources that are externally visible and therefore exposed to attack. A related notion discussed by Swiderski & Snyder [3, 49] is that of ‘privilege boundaries’, being the places within code involving interactions between entities of differing privilege. Similar concepts are referred to by other authors [2, 50] under different names such as ‘trust’ or ‘security’ boundaries, highlighting the role that such boundaries play in determining whether a program will be vulnerable to attack. However—regardless of the degree of security at the application level—as Baker [51] has argued, a secure operating system is required to provide a strong foundation for these high-level features. Operating system security research will be discussed in the next section.

2.3 Computer Security

2.3.1 Early Trusted Systems and Outcomes

The recognition of the existence of a computer security problem by the US government and military agencies in the 1960s led to an initial response of using ‘tiger teams’ to attack systems and identify vulnerabilities [52, 53]. However, this ad hoc ‘penetrate and patch’ approach did not produce secure systems, which led to a search for a more comprehensive solution [54]. Notable products of this work include the related notions of a ‘reference monitor’ and ‘security kernel’ described by
Anderson [55] and Bell & LaPadula’s (BLP) [56,57] confidentiality-oriented security model.

There have been many significant outcomes of this early research, both positive and negative. Important security concepts were identified during this time and still have an impact on contemporary systems [58]. Examples of these include the distinction between user-centric discretionary access controls and mandatory access controls that are enforced according to global security policy and cannot be bypassed by individual users [13]. Other concepts also originating from this period are the notion of a trusted computing base (TCB) representing the collection of system components responsible for enforcing security policy [55] and a physical trusted path by which users can ensure they are interacting with the TCB and not malicious code [26].

However, there have been many criticisms of this early work. For example, while the BLP model has been extremely influential, its confidentiality-centric approach is not appropriate for most modern security requirements where, as Clark & Wilson [59] have argued, integrity is typically more relevant. The model’s practical weaknesses in requiring trusted subjects, and the difficulty in eliminating covert channels, demonstrate the obstacles involved in implementing an abstract, formal security model [57,60,61]. Additionally, the usefulness of its Basic Security Theorem has been heavily criticised, most notably by McLean [62,63]. The evaluation criteria developed to assess the security of these systems have also been criticised. Some evaluation criteria have been criticised for being too inflexible and confidentiality-centric [13,26]. Meanwhile, others, such as Shapiro [64], have criticised more flexible criteria for being so complex as to be meaningless to the average consumer. The length and cost of the evaluation process is also of concern [13,34,65,66]. As Neumann [67] has observed, a consequence of this is that trusted systems have not been widely deployed, with most contemporary operating systems utilising discretionary-based approaches.
2.3.2 Security Models of Mainstream Systems

Mainstream systems such as Unix and Windows NT utilise discretionary access control models, where users specify permissions or access control lists (ACLs) on objects they own to determine who may access these [68, Ch. 5]. Mandatory access schemes are not integrated into these systems, although extensions that provide these features are available and will be discussed in the next section. Other features associated with trusted systems are also relatively rare, although one well-known counter-example is Windows NT’s ‘Secure Attention Sequence’ that allows a physical trusted path between the user and TCB to be established [69]. In this case, a trusted path is achieved through the kernel responding to a Control-Alt-Delete key sequence, which cannot be trapped by applications, and allows the user to perform some limited, predefined administrative tasks such as changing their password, logging out and killing system processes.

As is typical with discretionary access control schemes, access rights are granted based upon the identity of the subject and so this is sometimes referred to as identity-based access control [18, p. 103]. Given the importance of identity, users are authenticated before being granted access to the system by being required to supply the password or passphrase associated with the identity of the account they are logging into. Therefore, password security is a critical issue as password compromise—whether achieved by interception, or by dictionary attack on low entropy passwords—inevitably leads to security failure. Notably, schemes that enforce mandatory security also depend on the successful authentication of a subject’s identity for the assignment of the correct sensitivity labels [34]. Consequently, a great deal of work has been conducted on the issue of password security [12, 70–75].

Security Model Weaknesses

Loscocco et al. [4] have argued that the discretionary security models found in mainstream systems, such as those described in the previous section, do not provide an adequate foundation for application layer security features. Instead, they state that
mandatory security and trusted path are essential for achieving such a foundation. However, use of discretionary access controls is a far less frequently criticised feature of Unix than its privilege model [2, 3, 5, 26–28]. This privilege model involves two tiers of users: standard user accounts who can access files according to discretionary permissions, and the superuser account (known as ‘root’) that holds all administrative privileges and is not constrained by any access controls whatsoever. Garfinkel \textit{et al.} [5, pp. 105–108] observe that “the superuser is the main security weakness in the Unix operating system” and that, as most Unix vulnerabilities grant superuser access, “most Unix security holes result in a catastrophic bypass of the operating system’s security mechanisms” [5, p. 108].

The reasons why such compromises are so common are twofold. One reason is that, as Garfinkel \textit{et al.} [5] note, the power of the superuser account makes this the only meaningful target for attackers. However, the breadth of the superuser’s privileges is an equally significant reason, since the requirement of superuser privileges for even the most basic administrative tasks causes the superuser account to be broadly utilised and thereby far more exposed to attack than non-privileged accounts. This creates a large number of privilege boundaries and makes the superuser account almost impossible to defend. Furthermore, the superuser’s privileges are indivisible so that a process which requires minor administrative privileges automatically inherits unrestricted access to the system. For example, superuser privileges are necessary to authenticate and change passwords, bind network servers to trusted ports, access most hardware devices, and configure the network. Thus, superuser privileges are encapsulated in many network servers, most system service programs (‘daemons’), special programs designated to run with root privilege (set UID programs), the graphical environment subsystem (X Window System server) and every program executed by the administrator. The sheer extent of the attack surface this creates means that a single vulnerability leads to a global security failure. Some responses to this flawed privilege model will now be discussed.
2.3.3 Mainstream Alternative Privilege Models

BSD securelevels

One technique for limiting the privileges of the superuser under Unix is the creation of different security levels at which a system may operate. Details of the scheme are described by Garfinkel et al. [5, pp. 118–119] in their text on Unix security. The levels begin at ‘securelevel 0’ with this incremented to 1 when the system changes from single user to multi-user mode at boot time, and optionally from there up to securelevel 2. Each increment cannot be reversed and enforces additional restrictions on what processes may do, including those executing as root. While the securelevels mechanism provides limitations on the superuser which have the potential to improve security, as Kamp & Watson [76] have observed, this comes at the cost of significant hidden complexity, as the impact of setting a higher securelevel throughout the system can be difficult to predict. Furthermore, Garfinkel et al. [5] criticise the mechanism for being impractical because reboots are required for many common administrative tasks after the securelevel has been raised. This requirement makes the mechanism unsuitable for most production systems. Both Garfinkel et al. and Dowd, McDonald & Shuh [2] suggest that a sufficiently motivated and privileged attacker could still bypass the additional security afforded by the securelevels mechanism, by modifying system files to allow bypassing of the restrictions upon the next reboot.

Linux Capabilities

The Linux operating system provides a more finely grained mechanism for limiting superuser privileges where individual administrative rights, known as ‘capabilities’\(^1\), are created. Under the scheme, a process can only perform a privileged operation if it holds the corresponding capability, irrespective of which user executed it. By

\(^{1}\)Traditionally, the term ‘capabilities’ refers to an access control architecture where capabilities “encapsulates object identity” [18, p. 390]. The technique is used by some systems and models in place of more common access control lists [77–83]. The way the term is used in this case is in a more general sense where a right is encapsulated into a flag or token.
partitioning superuser privileges in this way, the principle of least privilege can be enforced and the impact of security failures thereby limited. Linux capabilities are therefore more finely grained than BSD securelevels. However, Dowd et al. [2] have criticised the scheme for its increased complexity, which has already resulted in security flaws having been identified. Furthermore, as Garfinkel et al. [5, p. 121] have observed, the effectiveness of the mechanism is dependent upon software being aware of its existence and dropping capabilities when they are no longer needed. As few programs currently do this, the lack of backward compatibility significantly limits the benefits of the scheme.

2.3.4 Mandatory Security in Mainstream Systems

Following from the observation by Loscocco et al. [4] that mainstream operating systems do not include mandatory security features, there has been some work to build extensions that incorporate this kind of functionality. For example, Spencer et al. [82] propose the Flask security architecture and this has been implemented on the Linux operating system under the name Security-Enhanced Linux or SELinux [7, 83]. The SELinux system has been integrated with a number of Linux-based systems [84], giving it a relatively wide deployment for a mandatory security-based scheme. SELinux attempts to address limitations of traditional mandatory access control implementations, most particularly their confidentiality-centric approach. The scheme is able to support a variety of security policies, such as role-based access control [85], type enforcement [86, 87] and multilevel security [56]. The proponents of SELinux attribute this flexibility to the mechanism’s design that separates policy from enforcement.

Another similarly flexible security extension is the Rule Set Based Access Control, presented by Ott [88]. This mechanism can also be adapted to a variety of required security models, including a ‘role compatibility’ model that is similar to the type enforcement module provided by SELinux. Although having been developed from fairly different origins, the two approaches are quite similar and the differences in
practice (described in [7]) are relatively subtle. A number of other similar projects such as Novell’s [89, 90] ‘AppArmour’ and ‘gsecurity’ [91] also aim to provide similar sorts of security infrastructures for the Linux system, either as patches to the mainstream kernel or using the standard Linux Security Module (LSM) interface. The Flask architecture has also been ported to FreeBSD in the form of the TrustedBSD project [92]. A version of the Solaris operating system from Sun that incorporates mandatory access controls is also available [93].

Limitations and Obstacles to Adoption

Mandatory security-based models, such as SELinux, provide a far more comprehensive response to the problem of superuser omnipotence than securelevels and Linux capabilities as they both limit the superuser and provide a strong alternative generic security model. From the user’s perspective, this latter point may be a significant disadvantage as it has been observed that the imposition of an alien security model often causes frustration [13, p. 75-76]. Such complaints have been observed in user online postings related to SELinux where users indicate that they feel the model is too complex, unsuited to their security requirements, and “forces too many changes on fundamental unix [sic] concepts” [94, 95]. This is problematic as the lack of user acceptance can lead to limited adoption and attempts to bypass or disable security features.

The issue of SELinux complexity has also been discussed in the academic literature. While Smalley [10] provides an overview on the configuration of SELinux security policies, researchers have raised concerns regarding the difficulty of constructing a complete policy and obtaining sufficient assurance that the configuration is correct. Archer, Leonard & Padella have criticised the policy language as being “very low-level and detailed, making the high-level properties of a policy difficult to check by inspection” [96, p. 158]. Zanin & Mancini observe that interrelationships can occur between the rules within a policy configuration, making it “extremely difficult to understand their overall effects in the system” [9, p. 136]. Adapting the configura-
tion to deal with specific requirements is also a significant problem, with a number of rules in the policy potentially growing to well over 50,000. As Jaeger, Sailer & Zhang state, configuring the system securely becomes “an arduous and error-prone task” [8, p. 59]. This task is both a critical and complex one, with any flaws in either policy or its specification within the SELinux framework leading to security failure. This is borne out in one user’s report on his experiences configuring an SELinux policy where this needed to be adjusted several times as specific flaws and limitations were identified [97].

**Security Model Requirements**

It is not clear that the models developed to date have achieved the right balance between usability, flexibility and security. While the design of architectures such as Flask and SELinux is secure and flexible, the significant paradigmatic differences compared with those widely deployed and the additional complexity involved make them problematic as alternatives to existing weak models. Ultimately, to gain widespread acceptance, a security model must be able to be readily abstracted and understood by the user and also not interfere with or make more complex or onerous the tasks that the user must complete. It has been argued that “users need a clear model of how the security operates (if not how it actually provides security) in order to use it well” [11, p. 3] and the replacements that have been developed to strengthen existing models have failed to achieve this. What is required is a model that enhances security by limiting the power of administrative users while being conceptually simple to use and aligned with common user requirements. No models that meet these criteria have been developed to date.

### 2.4 Overview of Cryptography

Modern cryptography involves the application of different cryptographic primitives for the construction of protocols in order to achieve some security goal. The primitives typically considered are symmetric encryption, one-way hash functions, public
key encryption, digital signatures and pseudorandom numbers [98, 99]. Numerous algorithms for each primitive are available with different security and performance properties [100–109]. Protocols exist for different purposes, with the most numerous and widely used category being that of authentication and key exchange protocols [110–117]. These protocols are used to establish a secure channel across which two or more authenticated parties can communicate. Design of these types of protocols (and others) is problematic due to the wide variety of potential circumstances that may arise during a protocol run and the difficulty in ensuring that the protocol is secure against all possible attacks. In general, assumptions cannot be made about either the security of the network or that of remote systems. Furthermore, protocols must be resistant against a wide variety of passive and active attacks. Schneier [118] provides an in-depth description of common algorithms and protocols, along with an analysis of their security.

2.4.1 Relevance of Cryptography to Networking

Cryptography is particularly relevant to networking as messages exchanged between parties usually travel across untrusted networks—such as the Internet—via nodes controlled by other parties. Cryptography is the mechanism by which remote parties may authenticate one another and protect the confidentiality and integrity of the messages exchanged [80, 81, 119–124]. As a result, cryptography is far more widely used today than it has been in the past [125]. However, the underlying property of cryptography that makes it so significant in securing network communications is rarely explicitly examined. Existing computer security models involve a centralised, trusted control agent such as a security kernel that can enforce the rules and policy required on that system. However, in a networking environment, each system is largely autonomous and it is not generally possible for one node to impose an arbitrary security policy on another. This can only be accomplished if there is a trusted enforcement mechanism and no such mechanism exists in an open network. However, the universal applicability of the

29
mathematics underpinning cryptography allows a well-designed protocol to ensure that the desired policy will be enforced on a remote autonomous system, even if no other trust can be placed in the hardware or software on that system. It is this property that, for example, prevents a user from maliciously modifying a Kerberos ticket to gain access to a service for which they are not authorised [81,126]. A number of other schemes, such as Gifford’s cryptographic sealing, CNFS and SIRiUS [127–129], that demonstrate this property will be discussed in more detail in Section 2.5.

2.4.2 Use of Cryptography in Operating Systems

Use of cryptography in operating systems is less common than in networking, with the most frequent use being in the form of standalone tools that accomplish some specific purpose such as file encryption. The lack of integration of these tools limits their effectiveness and convenience [130] and more closely integrated approaches have subsequently been pursued. The three categories of these approaches that are relevant to this work are cryptographic filesystems, file integrity verification mechanisms and secure storage of secrets. These three will be discussed in detail in the following sections. Other miscellaneous uses of cryptography in operating systems, which are largely orthogonal to this research, include cryptographically secured audit trails [131–135], virtual memory encryption [136], interfaces to cryptographic hardware [137] and random allocation of identifiers in order to stymie certain attacks [138–140]. Finally, cryptography is typically used to facilitate storage of passwords on the system for the purposes of authentication [141]. Storing hashes of passwords leverages the non-invertibility property of one-way hash functions to limit the usefulness of these values if recovered. However, the low entropy nature of user passwords means that dictionary attacks are often feasible and secure storage of passwords remains a significant problem [12,70–74,142]. A related use is the construction of one-time passwords that can be used to authenticate only once and reveal nothing to an attacker concerning the next valid password [143–145].
2.5 Cryptographic Filesystems and Access Control

One use of cryptography in operating systems is encrypting file data. Although data encryption can be implemented by individual applications, moving this functionality into the operating system or filesystem layer is a more transparent, interoperable and convenient approach [130]. Often the primary goal is to protect data in the event that an attacker gains unrestricted physical access to a machine, such as in the case of the stolen laptop. However, some schemes utilise cryptography to construct an access control scheme rather than simply to protect low-level confidentiality. Unfortunately, the designs produced to date have numerous limitations and these will now be discussed.

2.5.1 Cryptographic Filesystems

Blaze's Cryptographic File System

The CFS scheme developed by Blaze [130] is an early cryptographic filesystem and largely established the mould for later designs, with its successors only making incremental improvements [146,147]. Its design recognizes that standalone encryption software is inconvenient, error-prone and creates interoperability problems, while low-level hardware-based encryption is too coarse an approach and cannot provide per-user security. Instead, Blaze argues, a middle ground OS layer approach is superior.

CFS involves each user creating their own encrypted directory that can then be ‘attached’ to some part of the filesystem tree through the provision of an encryption key in the form of a passphrase. Access to encrypted files under this directory is subsequently transparent to all applications. Implementation uses a custom, user-space NFS server and this approach is portable, requires no kernel modifications and allows CFS to be used transparently over a network [147]. A symmetric block cipher is employed for encryption with a novel combination of output feedback mode and electronic code book mode (ECB), which allows random access to file
data while avoiding the security limitations of ECB mode [118, Ch. 9]. Blaze [148] later described a key escrow system involving smartcards to extend CFS.

Zadok & Wright [146,149] have criticised CFS for its poor performance resulting from the use of a user-space server, which creates the need for many context switches and reduces the scope for in-kernel caching. Furthermore, access to encrypted but attached data is also controlled based upon the user ID (UID) of the process making the request. Consequently, the superuser can bypass the cryptographic protection for attached files under the normal Unix rules for access control. This anomaly further highlights the flaws in the Unix security model that have already been discussed.

**Transparent CFS**

Cattaneo et al. [150] describe the Transparent Cryptographic File System (TCFS). TCFS is intended as an improved version of CFS using a very similar architecture, but which operates in a more transparent manner. For example, TCFS no longer relies on the concept of attaching an encrypted directory to the filesystem tree but instead allows encrypted files to be accessed transparently once the password has been supplied. TCFS allows for files to be individually marked as encrypted or otherwise, meaning that both types of files can be contained in the same directory. Unlike CFS it allows for group sharing of encrypted files using a threshold scheme [118, pp. 71–73] where each user in the group holds a ‘share’ representing part of the secret key and where, if a sufficient number of the group load their shares into the kernel, the encrypted data becomes accessible to the entire group. Cipher block chaining (CBC) mode is used to encrypt the data but this is done with a separate key per file block to facilitate random access, which CBC mode normally inhibits. TCFS is also integrated with the kernel and is therefore generally faster in operation than standard CFS.

Although TCFS does incorporate some improvements over Blaze’s CFS scheme, most of these improvements are marginal and there are some areas where TCFS is
significantly weaker. For example, Kher & Kim [151] describe a dictionary attack on the cryptographic keys stored in the TCFS database, amongst other weaknesses. Cattaneo et al. [150] contend that the security of TCFS can be improved through use of Kerberos to support key management. However, doing so requires that the applications involved be aware of the mechanism and thereby sacrifices the scheme’s supposed transparency. TCFS also has the undesirable properties that files created under encrypted directory trees are not automatically encrypted and, if encrypted files are copied, the duplicates automatically and silently also lose their encrypted status—this applies even if the destination is under the same protected directory. Finally, the threshold scheme used for group sharing is awkward and does not reflect the way users typically collaborate; it requires sufficient users with a share to be logged into the system simultaneously in order for any user to access the shared files. Avoiding this requires that a threshold of one be used, which makes the use of a threshold scheme entirely redundant.

Encrypting File System (EFS)

Microsoft Corporation developed EFS as an extension to the NTFS filesystem and it has been supported since Windows 2000 [152,153], [69, pp. 343–356]. It involves the user marking a file with the ‘encrypted’ attribute causing it to be encrypted with a symmetric key (termed the File Encryption Key or FEK), which in turn is encrypted with the user’s public key and this ciphertext is then stored in the file’s header. The corresponding private key is kept symmetrically encrypted with the user’s login passphrase. EFS is integrated into the Windows GUI and, once enabled for a file, the decryption process is entirely transparent.

EFS also has extensive support for key escrow and data recovery with all FEKs being automatically encrypted with a public key belonging to the specified Data Recovery Agent (DRA), of which there may be several. Windows XP provides some support for sharing encrypted files by manually selecting the users involved, but there is no group functionality. Sharing is implemented by encrypting the FEK
with the public key of each user; however, the FEK is not automatically changed upon revocation and the file is not re-encrypted. Kher & Kim [151] suggest that this design allows a revoked user, who can gain physical access to the device, to manually decrypt the file and read its contents.

EFS has four notable limitations. Most obviously, it is restricted to selected Microsoft Windows environments with no support for or interoperability with other platforms such as Unix. Another limitation observed by Wright & Zadok [146] is its dependence on userspace components for encryption and authentication, which prevents the scheme from encrypting the contents of system directories. Its design also involves encrypting and decrypting data just prior to accessing the disk and therefore it cannot be wrapped around distributed filesystem protocols as most other systems described here [147], although one reference [151] suggests this is not the case when ‘web folders’ are used. And finally, as Wright & Zadok [146] have noted, the use of login passphrases to protect private keys makes these vulnerable to dictionary attack and also creates potential reliability problems if an attacker resets the user’s password.

Cryptfs and NCryptfs

Zadok, Badulescu & Shender [154] describe Cryptfs, a proof-of-concept cryptographic filesystem developed using the notion of ‘stackable’ filesystems [149], and designed to improve the performance and security of previous systems such as CFS and TCFS. It relies on the vnode abstraction used in Unix filesystems that hides the details of the physical filesystem underneath. More specifically, it uses the notion of ‘vnode stacking’ where one vnode interface can call another. To improve security over previous schemes, Cryptfs uses both the UID and process session ID to identify whether access to an encrypted file is permitted, thereby making it more difficult for users with root privileges to access encrypted files belonging to others.

Wright, Martino & Zadok [155] have subsequently presented NCryptfs, an improved version of Cryptfs that adds several significant new features, including a generic,
finely-grained mechanism for sharing of encrypted data between users and the ability to revoke access when required. NCryptfs also allows the owner of data a great deal of flexibility in determining how long another party may interact with this data. For example, not only may their access to the data expire after a specified amount of time, but they may also be required to re-authenticate themselves to the system while accessing the data after another specified interval, and even to re-authenticate after a specified period of inactivity\textsuperscript{2}. An improved encrypting filesystem, which supports integrity verification of encrypted files, is described by Sivathanu, Wright & Zadok [156].

Unfortunately NCryptfs’s flexibility is also a potential weakness. The design of the access control scheme is complex and arbitrary, and therefore contrary to established secure design principles, such as those described by Saltzer & Schroeder [15], which recommend an economy of mechanism. This is also problematic from a user perspective, as the design involves a degree of conceptual complexity in order to be used effectively and securely. As Whitten & Tygar [157] have argued, if users cannot properly conceptualise a security mechanism then they are likely to misuse it or not use it at all. NCryptfs lacks the simplifying abstractions, recommended by Cox et al. [11], by which the architecture can be easily explained to end users.

For example, in order to facilitate group sharing in situations where the underlying Unix permissions do not allow this, the owner must grant these users the “Bypass VFS Permissions” right\textsuperscript{3}, which allows the recipient to take on the owner’s identity in a manner analogous to the set UID mechanism. This approach requires careful programming to be implemented securely and also demands the users have a detailed, low-level understanding of Unix security primitives in order for the feature to be used safely and effectively.

The scheme also exhibits other security problems. File encryption keys are generated from user-supplied passwords leading to the potential for dictionary attacks.

\textsuperscript{2}These restrictions are somewhat weak as they can be bypassed for read protection by making a copy of the protected file which can be accessed indefinitely although this will not include subsequent updates or modifications.

\textsuperscript{3}This is one of seven different rights which may be set by users, with two additional rights able to be set by the administrator.
Also, although linking a key with a specific session is superior to previous UID-based designs such as CFS, ultimately this is not a cryptographic link and can still be bypassed by a suitably privileged and motivated attacker. This problem occurs because most cryptographic filesystem designs are not designed as generic access control schemes, but rather as tools for dealing with specific security threats such as the physical theft of a machine. So far, work on generic access control architectures that utilise cryptography and take advantage of the unique properties that it provides have been relatively limited. One early approach that meets these criteria will now be examined.

2.5.2 Gifford’s Cryptographic Sealing

Gifford [127] describes a scheme that he refers to as “cryptographic sealing”, which is unique in that it represents not only a cryptographic access control scheme, but also effectively a protection primitive that can be used to construct other specific access control architectures. Cryptographic sealing is essentially a lock and key approach [18, p. 396–400] that combines features of capabilities and access control lists by splitting the informative access control data into two parts: one residing with the object and the other with the subject. These are the lock and the key respectively, and access to a given object is dependent upon the subject having the appropriate key for that object’s lock.

Gifford distinguishes between an ‘active’ protection system that requires some agent, such as a security kernel, to enforce the appropriate policy and ‘passive’ approaches (such as Gifford’s) where protection is always enforced and there exists no enforcement entity to be bypassed. This concept is related to the notion of using cryptography to enforce security policy remotely as described in Section 2.4.1. However, purely passive mechanisms are limited in that they can only provide confidentiality and authentication, while traditional active schemes can also enforce integrity and availability requirements. Gifford acknowledges this limitation and introduces a ‘guard’ to which a password must be supplied in order to gain write
access to a particular object.

The scheme involves two types of key with which objects may be encrypted. A ‘basic key’ is essentially a randomly generated symmetric key, while a ‘derived key’ is calculated from an existing key in some manner. The means by which it is calculated depends on how the key is to be used. Derived keys are used to implement threshold schemes whereby, not only can access to a particular object be granted to other users as the owner of that object wishes, but a highly flexible set of restrictions can also be enforced whereby the object may only be accessed when a certain combination of keys are available to the subject\(^4\). This is done through the notion of ‘Key-OR’ and ‘Key-AND’, which require either or both keys respectively. As these may be combined, an enormous variety of security policies can be constructed; this is what gives the scheme its high degree of flexibility.

The primary advantage of Gifford’s scheme is its highly flexible nature. In particular, the use of a threshold scheme creates the possibility of a large number of access control configurations that are not available with more conventional approaches. Gifford demonstrates this by describing how the scheme can be used to implement capabilities, access control lists and also a multilevel secure system. That a single underlying access control scheme could be used to implement these considerably different approaches is impressive and demonstrates the flexibility of Gifford’s design. Another, perhaps even more significant, advantage is the reduction in importance of the reference monitor through the passive, self-enforcing nature of Gifford’s scheme which potentially reduces the attack surface area of the system. However, the need for a guard mechanism to enforce integrity protection means the privilege boundary cannot be entirely eliminated.

**Disadvantages of Cryptographic Sealing**

The chief disadvantage of Gifford’s scheme is its inherent complexity. While base keys are simple and linear in their behaviour, the recursive nature of derived keys

---

\(^4\)Note this threshold scheme works on the basis of a single user rather than a collection of users as in TCFS as described in the previous section.
makes constructing and modelling the security policies that they implement increasingly difficult as more keys and rules become involved. Not only does this make it difficult to evaluate the security of a specific policy but it is also problematic in the requirements it sets for administrators and users working with the system. These parties already have extensive experience with existing access control mechanisms and, as a completely new and unique way of enforcing access control, Gifford’s scheme shares little similarity with existing, widely used schemes. Gifford’s model therefore cannot leverage the substantial experience that users and administrators have gained over years of working with and managing existing access control architectures. The fact that the non-linear, recursive and largely unstructured nature of the scheme rapidly becomes much more complex outside of extremely simple security scenarios using base keys, only serves to magnify fundamental differences between Gifford’s approach and those currently in widespread use. This makes it extremely difficult for users to be able to visualise how the security model works, making it difficult to use in practice [11].

Therefore, while the technique can emulate alternative access control mechanisms, given that reliable implementations of these already exist using conventional techniques, it is not clear that there is any advantage in using Gifford’s scheme in this way. Furthermore, the model gives no consideration to important issues, such as revocation, except to note that an active mechanism is required to enforce this. Finally, the scheme is also vulnerable to unauthorised transference of keys from one party to another and therefore lacks the expected properties of existing, non-cryptographic access control models.

Despite these issues, Gifford makes strong arguments regarding the advantages of passive mechanisms using cryptography. Arguably, what is needed is a scheme that provides the advantages inherent in cryptography, while being able to prevent unauthorised sharing of access tokens between users and lowering the barrier to adoption by being conceptually similar to existing access control models.
2.5.3 Other Cryptographic Access Control Models

Cryptographic Access Control and CNFS

Harrington & Jensen [128] describe a locks and keys access control scheme that they term "cryptographic access control". The scheme is intended for use in a distributed system and has been implemented as an cryptographically-enabled version of the network file system (NFS, [158]), referred to as CNFS.

One of the guiding principles of the work is to transfer the reference monitor and TCB away from the server and towards the client through use of cryptography. This is closely related to Gifford's notion of a passive enforcement mechanism and represents another example of cryptography being used to remotely enforce security policy, as discussed in Section 2.4.1. The client places a file on a remote server, having encrypted it with a symmetric key, and then digitally signs it with a private key that is referred to by the scheme, somewhat confusingly, as the 'encryption key'. The server is provided with the corresponding public key ('decryption key') and verifies the attached signature to ensure that the file is valid. However, as the server does not possess the symmetric key, it is unable to read the original data. No mechanism is specified for limiting access to the server so a naive implementation would allow uploads from anyone. No additional protection is required for the files on the server since it is impossible for anyone to read the data unless they possess the symmetric key and the server does not allow modifications to the data unless they are signed with the appropriate encryption key. This significantly diminishes the server's role in enforcing security policy as all that is required of it is the checking of signatures. As a result, the extent and complexity of the security-related logic implemented by the server is also reduced. Access can be granted to other principals by giving them the symmetric key and either the decryption key for read access or the encryption key for write access.

Unfortunately there appear to be a number of security-related problems with this scheme. Some relate to the use of cryptography in the system; for example, the file
is encrypted before it is signed, which is generally considered to be ill-advised for a number of reasons [118, pp. 41–42]⁵. The scheme, as described by Harrington & Jensen [128], is also vulnerable to replay attacks where an attacker can effectively roll back a document to an older version unless some mechanism for ensuring freshness is incorporated. Another significant oversight is the lack of a key distribution mechanism, without which it becomes impossible to share data with other remote users, thereby negating a significant advantage of a distributed filesystem. More seriously, the scheme provides very limited support for revocation as this requires deletion of the object, re-encryption with a new key, and distribution of this key to all other authorised parties. However, deletion is problematic as the server has no ability to distinguish between the owner of the object and someone who has been granted write access. Therefore, any user with write access can delete and recreate the object, effectively taking possession of it and locking out authorised users, including the original owner. Finally, the scheme does not prevent users who have been granted access to an object from ‘gifting’ these access keys to other unauthorised parties who can then do likewise. These flaws undermine the security of the scheme in terms of facilitating remote collaboration.

SiRiUS and Plutus

Goh, Shacham, Modadugu & Boneh [129] describe the ‘SiRiUS’ distributed cryptographic filesystem, while Kallahalla, Reidel, Swaminathan, Wang & Fu [159] present a similar scheme named ‘Plutus’. Both models aim to provide independent confidentiality and integrity protection for files with the ability to grant these rights to other users. Files are encrypted with a symmetric key, the possession of which grants read access. They are also signed with a private key which, in conjunction with the symmetric key, grants both read and write access. Neither scheme allows for write access to be granted separately from read access. One notable difference between the two is the elegant and efficient notion of ‘file groups’ used by Plutus.

⁵A well-known attack exists against the extremely widely-used RSA public key cryptosystem when a message is encrypted and then signed [118, pp. 473–474]. A brief analysis suggests that this scheme is not vulnerable, however, a closer examination would be required to confirm this.
whereby all files with the same owner, group and permissions are automatically encrypted with the same key.

In SiRiUS, metadata is stored on the server in a separate file and a public key is included for verifying the file’s integrity. No active mechanism exists to prevent malicious modifications, although they are guaranteed to be detected. Plutus avoids this problem by issuing those granted write access with a ‘write token’. A hash of the write token is stored on the server, which must be trusted to verify the token before allowing modification of the file. Another difference is that Plutus lacks a key distribution mechanism, which must be achieved independently. Revocation is awkward under both schemes, with a new encryption key needed for all relevant files. The affected files must then be re-encrypted and the new key securely distributed to all relevant parties. Plutus minimises the impact of revocation through an optimisation strategy known as ‘lazy revocation’ where a file will not be re-encrypted until it needs to be updated, although the effect of this approach on access control semantics is not considered. Both schemes also fail to prevent users passing on keys they have been granted to unauthorised parties and therefore many of the problems exhibited by Gifford’s cryptographic sealing and CNFS also apply here. So far, no cryptographic filesystem or access control model has been developed that avoids all of these limitations.

2.6 Secure Storage of Secrets

2.6.1 Secure Coprocessors

A fundamental issue in cryptographic systems is storing and maintaining the confidentiality of ‘secret’ values such as keys and passwords. While such secrets can be encrypted, this new encryption key must in turn be stored, making it a cyclic problem [160]. One option, summarised by Smith [161], is the use of secure coprocessors where secret values (and sometimes security-critical processing) are moved from the untrusted machine to a separate, limited functionality, physically-secure
computational device. Note that the motivation is not to improve performance, as is sometimes the case with add-on hardware, but rather to leverage separation to improve security.

A design issue in the use of coprocessors is the amount of functionality moved into the secure device and Gutmann [160] identifies five ‘tiers’ of coprocessor functionality, which highlights the paradox of this approach. If too little functionality is contained within the secure device then there is significant potential for it to be mis-used by malicious external code in undesirable ways (such as digitally signing documents without the user’s consent) because it will contain insufficient logic to adequately determine the context of the requests it receives. However, if too much functionality is moved into the coprocessor then it becomes too complex to be sufficiently trusted and its original purpose is diluted.

2.6.2 Secret-Protected Processors

The design of the ‘secret-protected processor’, or SP-processor, proposed by Lee, Kwan, McGregor, Dwośkin & Wang [162], and McGregor & Lee’s [163] ‘virtual’ secure coprocessor from which it evolved, highlight some of the limitations of hardware-based secure storage. These involve changing the microprocessor to support coprocessing in conjunction with a ‘trusted software module’ (TSM), which runs in a ‘concealed execution environment’ so no other processes on the system (including the OS) can interfere with or observe it. Secrets are encrypted as part of a hierarchical ‘key chain’ with the ‘user master key’ at the root. This can be generated from a passphrase that must be input directly to the TSM, bypassing the OS.

The first problem with this approach is that it requires extensive hardware and software support. This support includes five new CPU registers, two new cache line flags, hardware support for encryption and hashing, six new instructions and secure I/O modes including two external LEDs and buttons. Specialised OS support is also required and applications must also be aware of the scheme and be able to
interface with it. With such extensive prerequisites, extensive industry support would be required for the scheme to ever be deployed and, with the emergence of the Trusted Computing Group (Section 2.6.4), this seems unlikely.

However, a more fundamental flaw exists. While the TSM executes in a concealed environment and all data it transmits is encrypted and hashed, the secrets it stores must at some point be released to external applications. To maintain confidentiality, these applications must be authenticated to ensure they are not malicious or have otherwise been subverted, but no mechanism for this is provided by the SP-processor. The designers suggest that “in many scenarios, a verified and trusted TSM [will provide] sufficient protection” [162, p. 6]. However, this claim assumes the TSM is not being attacked, which is a highly questionable assumption given the goal of the mechanism. Alternatively, the TSM should “attempt to verify the trustworthiness of the caller to its functions” [162, p. 6], either through the use of a secure bootup mechanism (such as that described by Arbaugh, Farber & Smith [164]) or utilising a trusted security kernel to “correctly identify caller processes, and then restrict access based on this identification” [162, p. 6–7]. However, both approaches require trust to be significantly extended beyond the ‘virtual’ secure coprocessor provided by the scheme, with security now depending primarily on these external components, as it would do if the secure coprocessor did not exist. This highlights the need for any scheme for securely storing secrets to adequately authenticate the code to which the secrets are released.

2.6.3 Plan 9 Factotum and Secstore

Cox et al. [11] describe a redesign of the security architecture of the Plan 9 distributed operating system involving the introduction of two new components known respectively as ‘factotum’ and ‘secstore’. Factotum is a per-user self-contained agent that is essentially an evolution of the SSH authentication agent developed by Ylönen [122] as part of the SSH remote login system [163], with two key differences. While the SSH agent manages keys only for the SSH client, factotum is not restricted
to a single protocol and contains modules supporting a variety of authentication techniques, with the capacity for more to be added. Secondly, both the client and server in a Plan 9 distributed system have factotum agents that negotiate with one another on behalf of the server and client processes to complete the authentication process. This makes the factotum agent a far more active player in the protocol, and allows authentication and cryptography code to be removed from general applications and replaced with a simple interface to the agent. The second relevant component of the Plan 9 security architecture is the seccstore network file server where keys normally held by a user’s factotum are stored in encrypted form when this agent is not running. Keys are loaded from the seccstore when the factotum process is started during boot up using Boyko, MacKenzie & Patel’s [166] provably secure password-based protocol, PAK.

Although proponents of hardware-based secret storage mechanisms argue that software-based approaches cannot provide the same level of security [160–163], Cox et al. [11] note that their design improves security and also makes the system easier to use. Furthermore, as discussed previously, the operating system and related software components are often better placed to determine whether a secret should be released to a specific application than hardware schemes such as the SP-processor architecture. Plan 9 also includes a ‘confirm’ mechanism that allows the user to mark specific secrets as requiring user confirmation before they are used to authenticate a process. However, it would be inconvenient to flag all secrets as requiring confirmation and, even though factotum never releases actual keys, secrets not flagged in this way may be potentially used by factotum to authenticate on behalf of malicious processes. This is because the factotum agent has no way of authenticating the processes involved and discerning whether a specific key should be used or not. This is therefore a similar problem to that affecting the SP-processor, albeit in a different form. The scheme also requires that applications be aware of the factotum agent and able to interface with it, although the work involved in modifying applications to add this functionality is minimal. Nonetheless, the Plan 9 factotum/seccstore architecture represents an interesting, user-friendly and relatively secure approach
to managing secrets within an operating system.

2.6.4 Contemporary ‘Trusted Computing’

A combined hardware and software approach to managing secrets is demonstrated by the specifications released by the Trusted Computing Group (TCG)\(^6\) industry consortium [167]. At the centre of this approach is the Trusted Platform Module (TPM) chip, summarised by Reid & Caelli [168], which is effectively a cryptographic coprocessor providing support for the cryptographic primitives and a small amount of protected key storage. This chip is designed to be a fixed part of a specific machine and has been incorporated in some PC systems since 2003 [169].

The TPM provides a basic cryptographic hardware infrastructure that can be used in a variety of ways, principally surrounding digital rights management [170] and high assurance computing [171–174]. The two main generic features supported by the TPM are protected storage and remote attestation [175]. Protected storage leverages the ability of the TPM to provide hardware-based protection of certain secret values. This allows construction of a tree-based key hierarchy with a Storage Root Key (SRK) at the root and stored in the TPM itself. Utilising this, various data and keys can be stored securely outside of the TPM. The protected storage mechanism therefore supports the construction of a custom secure coprocessor system, such as that proposed by Marchesini, Smith, Wild & MacDonald [169].

Remote attestation is the ability for the system to cryptographically prove to a remote party the status of its configuration and software. This is achieved through an authenticated boot process, partially based on the scheme by Arbaugh \textit{et al.} [164], where a hash function chain is constructed involving each piece of code to be executed being hashed, combined with the previous hash value in the chain. This result is stored in a TPM register and is ‘extended’ with each subsequent ‘measurement’ (hash) taken. The final value represents all of the code that has been executed in booting and can be digitally signed by the TPM and sent to a remote

\(^6\)Formerly known as the Trusted Computing Platform Alliance (TCPA).
party for verification. The objective of this approach is to provide surety for remote parties as to the software executing on a given system. In this context, as Abadi [176] has argued, the TPM effectively acts as a built-in trusted third party for use in certain protocols. However, Sadeghi & Stüble [177] have criticised this approach, arguing that assurance as to the security properties provided by a remote system is of greater importance than the specific software it is running. Indeed, these authors suggest that attestation based upon the specifics of an individual configuration could be used to exclude systems running software belonging to a competitor. The use of the TPM architecture to potentially enable draconian digital rights management (DRM) [168,178] has also been heavily criticised by some, notably Anderson [179]. However, a scheme that uses it more generally for code verification will be discussed in Section 2.7.3.

2.7 Cryptographic File Integrity Verification

There has been a variety of data integrity schemes proposed [180-184] and cryptography is widely used to authenticate code in distributed systems [185-191]. However, the following discussion will focus on cryptographic mechanisms for verifying object integrity within a single system.

2.7.1 Tripwire

Kim & Spafford [25,192] developed the Tripwire filesystem integrity checker designed for Unix-based systems [25,192]. It is intended to detect changes to critical system files in the event the system is compromised and the attacker gains superuser access. For example, an attacker may install a ‘root kit’ that modifies system programs to disguise his or her presence and to facilitate easy re-entry, even if the original vulnerability is closed. Tripwire works by creating a database of ‘signatures’ (hash values) of administrator-specified system files and their attributes, with this being stored on a read-only device such as read-only floppy disk, CD-
ROM or NFS mount. Tripwire checks are then run on a regular basis, usually in an automated manner through the standard Unix `cron` facility and the output is manually inspected to identify problematic changes. A variety of hashing algorithms are supported and an arbitrary number of these functions can be used to generate signatures for each file.

**Weaknesses and Limitations**

The designers of Tripwire state that “security tools should be completely self-contained, needing no auxiliary programs to run” [25, p. 23]. However, in practice, Tripwire has numerous external dependencies that undermine its security. For example, removing or corrupting the `cron` job used to automatically execute Tripwire would conceal an intruder’s presence (at least temporarily) in most configurations. Although Tripwire’s signature database and configuration file may reside on read-only filesystem, an attacker can also trivially redirect Tripwire to use ineffective, alternative files. A more sophisticated attack involves replacing the Tripwire binary with a corrupted version that does not report specified modifications. This circular set of dependencies is analogous to the subverted compiler attack famously described by Thompson [193]. This task is made easier by the availability of the Tripwire source code and would be very difficult to uncover. Finally an attacker can target the layer below the Tripwire program, namely the operating system kernel and shared libraries it depends upon for correct operation\(^7\). This is entirely feasible for an attacker with superuser privilege and, in fact, allows them to subvert the behaviour of any program on the system—this being the very problem Tripwire was designed to prevent. These attacks demonstrate that Tripwire can only provide effective security against an attacker who is unaware of its existence.

The causes of Tripwire’s limitations are twofold. First, Tripwire is a passive mech-

\(^7\)The current state-of-the-art in terms of root kit technology involves the use of hardware-based virtualisation where the root kit software actually places itself between the operating system and hardware, thereby becoming completely undetectable. Examples of such mechanisms include the ‘SubVirt’ system developed by King et al. [194] and Ruthkowska’s [195, 196] reported ‘Blue Pill’ mechanism.
anism that can simply detect modifications but not prevent them or actively alert users to the presence of the modified code they may be executing. This problem is magnified by the significant detection latency involved, which increases an attacker’s options in bypassing detection. The second reason is Tripwire’s dependence upon the underlying infrastructure in terms of the filesystem, system services, shared libraries and operating system kernel. This gives the attacker numerous opportunities to bypass detection by attacking Tripwire at a lower layer and subverting the infrastructure it depends upon to operate correctly. This weakness could be remedied by closer integration with the kernel, thereby allowing proactive, automated responses to integrity violations. Use of an alternative cryptographic primitive, such as digital signatures, could also reduce an attacker’s options. The CryptoMark scheme, which incorporates both of these approaches, will now be examined.

2.7.2 CryptoMark

Beattie, Black, Cowan, Pu & Yang’s [21] CryptoMark scheme is a code integrity and authentication mechanism implemented on Linux that deals with some of the weaknesses inherent in Tripwire. It involves generating digital signatures for important system programs and embedding them into the binary itself. The kernel is modified to verify the signatures before the program is executed and, if the code has changed, then the system will either abort execution or allow it to continue and report the incident to an intrusion detection system. Thus CryptoMark is capable of proactive responses in a way that Tripwire is not. The scheme operates in one of two modes where either all system binaries are required to be signed in order to be executed or where all programs executing with superuser privilege (either by virtue of the set UID bit or simply because the program is executed by root) must be signed.

The scheme’s implementation depends heavily on the Executable Linkable Format (ELF) used for executable binaries. An MD5 hash is calculated across the executable portions of such a file (other segments are not included) and the final sig-
nature is stored in the Note segment of the file. Executable files such as scripts and non-ELF binaries (for example, programs in the a.out format) are not protected, although the designers suggest that an auxiliary table file could be constructed to hold signatures for them.

**Comparison with Tripwire**

While the objectives are similar, the architectures used in Tripwire and CryptoMark are markedly different. Tripwire is entirely user-space and requires no kernel support at all, while CryptoMark is tightly integrated with both the kernel and the binary executable format used. This makes CryptoMark more resilient to low-level attacks but simultaneously reduces its portability. The use of digital signatures in CryptoMark, rather than hash values in Tripwire, eliminates the need to protect the signature database file. However, use of signatures also means that CryptoMark must keep the private signing key secret and prevent substitution of the public key. CryptoMark also has a much greater impact on performance due to the slower signature verification process, which must be performed upon execution of every signed binary. Finally, as CryptoMark can prevent execution of compromised code, it can potentially actively prevent intruder re-entry and also protect users from inadvertently revealing passwords and cryptographic keys. These properties make CryptoMark superior to Tripwire from a security perspective.

**Attacks and Limitations**

The design of Beattie *et al.*’s CryptoMark, which limits verification to code in ELF executables, leads to incomplete mediation of code and other objects that influence the code’s behaviour. This design weakness is contrary to established secure design principles [15] and can be exploited in a number of ways. One set of attacks involves modifying external configuration files relied upon by trusted code. Because only the executable code itself is signed, an attacker can modify other static inputs to the program, such as configuration files, to alter this code’s behaviour. Examples
of these include modifying user database files to create new and privileged users and subverting firewall rules to open up a specific port where they could install a shell process executing superuser privileges.

Numerous other similar attacks exist depending upon system configuration and features. A significant class of these attacks involves non-ELF executable formats such as the scripting languages that are widely available on modern systems. While the interpreter executable may be signed, it is the unsigned scripts it executes that primarily determine its behaviour when executed. This allows the attacker to write practically as much malicious code as desired. CryptoMark would not detect these attacks because no signed code would actually have been modified. A secure code verification scheme must recognize the dependency of trusted code upon parameters and other external objects for correct execution.

**Improvements to CryptoMark**

The weaknesses described in the previous section stem from the reliance of CryptoMark on embedding signatures in ELF executable files. As CryptoMark’s developers suggest, one solution would be a separate table storing signature information for other non-ELF files. However, this approach makes it problematic to identify which objects need to be verified and which do not. While CryptoMark’s previous policy for determining where verification is required avoided the circular integrity dependencies exhibited by Tripwire, placing this information in a separate table simply makes the table the target for attack. Given that an attacker with superuser privilege can potentially modify the table, effectively no additional security is gained by its existence, thus the same vulnerabilities already discussed still apply. Furthermore, creating an arbitrary distinction between programs that are verified and those that are not provides scope for redirection attacks where a user is tricked into executing a different program to that intended. This form of attack means that the reliability of matching objects to entries in the verification table is critical.

8Examples of scripting languages that could be used in this kind of attack include Java, Perl, Tcl, Python and the various shell scripting languages (Bourne, C and Korn shells, for example).
One way of avoiding this problem might be to check signatures for every object on the system. However, as Reid & Caelli [168] have observed, such an approach is highly impractical. For example, a universal verification would impose an enormous performance penalty, and is an especially inefficient approach as the majority of these objects would likely have little or no security significance. Finally, it is unclear how such an approach would deal with the creation of new files or modification of existing ones as these would be unsigned and, therefore, presumably inaccessible unless automatically signed by the system. Aside from the questionable security semantics of automatic signing, which would need to be examined very carefully, this approach would require that the private signing key be kept on the system securely and would therefore create another avenue of attack.

### 2.7.3 A TCG-Based Approach

Sailer, Zhang, Jaeger & van Doorn [22] present a scheme that uses the measurement capabilities of the TPM chip found in TCG-compliant computer systems (discussed in Section 2.6.4) to verify the integrity of general objects beyond the operating system. Verification can include objects such as configuration files, scripts and other data that can influence program execution. While the scheme could be used for remote attestation, this would depend on the challenger being able to determine the validity of a hash value for any given object. However, as Reid & Caelli [168] have identified, this seems certain to be problematic given the inherent diversity of such values. A more practical alternative is to use this functionality to verify the integrity of programs before they execute by caching known-good measurements and aborting execution if the hashes do not match. This approach achieves a similar outcome to CryptoMark.

The limitation of Sailer et al.’s [22] scheme is that it is essentially a binary system, as either a program’s measurement matches and the program runs, or the measurement is different and execution is denied. Given that many data objects will change frequently, these objects could not be included in the cache otherwise...
execution would frequently fail [168]. However, as the contents of these objects could subvert the behaviour of otherwise verified programs, this introduces a vulnerability. Such a weakness is especially problematic as all code executes with the same privilege level, regardless of whether integrity has been verified or not, and the issue is exacerbated by the lack of a convenient mechanism for updating object measurements. Furthermore, interactions between verified code and unverified code or data are not addressed. Therefore, while Sailer et al.’s approach is an interesting alternative application of the TPM chip, its practical and security limitations make it unsuitable as a general-purpose integrity verification mechanism.

2.7.4 I³FS

I³FS is a filesystem integrity mechanism, presented by Patil, Kashyap, Sivathanu & Zadok [23], that follows the Tripwire paradigm of storing caches of system files but avoids many of the problems with this, as well as those of CryptoMark. This success is achieved by storing policy and hash databases in encrypted form and integrating the mechanism with the filesystem. Because of this, the filesystem must be manually mounted and the appropriate passphrase input. Upon access to a file, the system checks for the existence of a hash and, if found, verifies that the file data has not changed. If a change is detected, access may be denied and/or an appropriate message written to the log. The scheme includes a number of means for trading off security in favour of performance, such as extensive caching and a configuration option for reducing the frequency of checks. However, the security implications of employing these options are not analysed.

Security Issues with I³FS

I³FS has a number of security problems, some specific to its design and others more general in nature; for example, privileged attackers can divert or delete log messages recording verification failure by accessing the filesystem directly. There are also serious questions regarding the approach taken to trading-off security to optimise
performance. For example, if metadata-only verification is employed rather than file contents, an attacker can again avoid detection by bypassing the filesystem and writing directly to the disk. Similarly, the checking frequency mechanism would need to be used judiciously and analysed carefully for security implications in practice. Finally, as the file data of previously verified objects is not cached, a privileged attacker writing directly to the disk can insert malicious code in place of that which has been previously verified.

The necessity for the filesystem to be manually mounted and its databases decrypted means that the boot process itself cannot be verified and creates the possibility that an attacker may insert their own malicious code to subvert integrity checking. Alternatively, the attacker could steal the I^3FS database encryption passphrase by, for example, replacing the legitimate version of the mount program. Such an attack could be easily achieved on System V Unix by adding code to the appropriate /etc/rc.d directory and this would not trip integrity checks in I^3FS prior to reboot, since it involves the addition of new code rather than modification of existing code. The use of a passphrase to encrypt hash and policy databases also exposes them to the threat of dictionary attack.

However, the easiest way to attack the I^3FS scheme, as described by Patil et al. [23], is by subverting the matching of the file being accessed to a corresponding entry in the database that determines whether the file must be verified or not. This matching is achieved by comparing the inode number for the object accessed against those contained in the database, thereby avoiding “unnecessary string comparisons” [23, p. 70]. However, this design decision allows an attacker to trivially substitute malicious code for trusted code that should be verified. This substitution is achieved by renaming the legitimate object to an alternative name under which it is unlikely to be recognized and substituting the malicious object in its place with the previous path name. The legitimate object will retain its previous inode and, if executed, will be successfully verified. However, the malicious replacement will have a new inode number with no corresponding database entry and therefore I^3FS will not
attempt verification. The user will remain oblivious to this redirection attack as, from their perspective, they are simply executing a program with its normal path.

2.7.5 General File Integrity Protection Problems

Of all problems described, it is the inode redirection attack just described that highlights the most serious limitation of schemes such as I³FS. If an object is found not to have a checksum, then access is allowed and the program or data concerned is treated in exactly the same way as if the object had its integrity cryptographically verified. This problem could be avoided if all objects on the system are verified. However, universal verification is typically not feasible since many system objects will change frequently; the integrity of newly created objects may be uncertain, and because of the significant performance overheads involved—especially as many of these objects will have no impact on security. Therefore, what is needed is a mechanism to manage the privileges assigned to a process based upon whether its integrity has been verified. Once the trust level of a process is established, modifications that the process makes to trusted objects can be assessed allowing for the automatic updating of verification data. The lack of a key management infrastructure is also problematic in terms of the requirement to manually input passphrases and a similar issue exists with the storage of the private key in CryptoMark. Further discussion of these issues and a model for dealing with them will be discussed in Chapter 5.

2.8 Summary

Of the security models developed to date, mandatory approaches provide strong security but are complex, unintuitive and often do not match users’ requirements. Widespread discretionary-based models are simpler to understand and use, but concentrate too much privilege with the administrative superuser account. Such a design is the principal security weakness of systems such as Unix. An alternative
approach is needed that provides an intuitive and conceptually simple paradigm, stronger security (including limits on the superuser) and is consistent with user expectations and requirements.

Cryptography provides a convenient mechanism for implementing a locks and keys access control model. In addition to enhancing security generally, a cryptographic approach can also be effective at constraining superuser privileges. However, existing uses of cryptography in operating systems have been rather limited. A number of file encryption mechanisms have been discussed in this chapter but all have practical and security limitations and none provides all the properties expected of a generic access control scheme. This is because these schemes have largely been developed as stand-alone security tools with little or no reference to existing access control theory.

A cryptographic access control model is significantly dependent upon an underlying cryptographic infrastructure, particularly the existence of a mechanism for securely storing keys. While both hardware and software secure storage mechanisms exist and have been discussed in this chapter, they lack the means to determine whether a key should be released when a given program requests it. As a result, keys can be inadvertently released to malicious or subverted code, which completely undermines the security the mechanism is intended to provide. Avoiding this problem requires a means for authenticating code. However, while code integrity verification schemes exist, none has been integrated with a key storage facility. These code integrity verification schemes also have significant practical and security weaknesses, particularly in relation to identifying which code should be verified. These problems can be significantly mitigated through the provision of a means for limiting the privilege with which unverified code executes. However, this requires a supporting access control and privilege model, something that all current integrity verification schemes lack.

In summary, therefore, all of the different cryptographic security features that currently exist in operating systems are, in fact, interdependent. This notion is illus-
Figure 2.1: Cryptographic security feature interdependencies

trated in Figure 2.1. However, the *ad hoc* and fragmentary nature of their design means that, so far, these interrelationships have not been recognized. Consequently these mechanisms remain limited from both a security and practical point of view, with the result being that they have not been widely deployed and their potential benefits not fully realised. An integrated, holistic approach that recognizes the mutual benefits and synergies possible through combining these different types of mechanisms allows for significant security improvements. Such an integrated approach also provides an enhanced security model for discretionary-based systems without the additional complexity of mandatory designs. Vaults, a security model that meets all of these criteria, will be described and analysed in the remainder of this dissertation.
Chapter 3

Vaults Infrastructure

3.1 Introduction

This chapter describes a number of elements that form the infrastructure on which the Vaults architecture is based. The chapter begins by delineating the different types of secure repositories or ‘vaults’ that the model provides. It then describes the Vaults Public Key Infrastructure (VPKI) that allows users to cryptographically authenticate one another when exchanging privileges. Also discussed are the mechanisms for storing and retrieving keys belonging to specific applications in the user’s vault, a model for administrative privileges and the management of Vaults-related credentials associated with processes, including the trust levels assigned to them. The prominent security features of the model, namely cryptographic file access control and code integrity verification, that are described in Chapters 4 and 5 depend upon this infrastructure for their operation.

3.2 Vault Types and Properties

The centrepiece of the newly-developed architecture is the concept of a vault which is a repository used to securely store a variety of items. There are five different types of vaults: User, Global Private, Global Public, Escrow and Fundamental.
The Global Private, Global Public and Escrow Vaults are referred to collectively as the ‘system vaults’. All of the vaults are stored on disk in encrypted form and loaded into secure memory when needed. The encryption used is required to be authenticated such that modifications to the ciphertext are detected upon decryption. This may involve use of a message authentication code (MAC) [17] in addition to encryption or an authenticated encryption mode such as OCB [197].

Each vault type plays a different role in the overall model and has different rules for access to stored items. These rules are enforced primarily through cryptographic means, with supporting enforcement logic provided by the security kernel. Details of all types will now be described.

3.2.1 User Vaults

All users of the Vaults architecture have their own, individual vault, known collectively as user vaults. User vaults hold sensitive data items belonging to that specific user and these contents determine that user’s privileges on the system. The three main categories of items held in a user’s vault are application keys, fingerprints and tickets. The symbols used to represent these items in future diagrams are given in Figure 3.1. The general layout and relationships of items stored in a user’s vault are illustrated in Figure 3.2.

Application keys are secret values pertaining to specific applications; for example, an e-mail account password used by a mail client. Application keys are bound to the program or programs that require them and cannot be accessed by other non-administrative software. Application keys are distinct from the cryptographic keys used by Vaults to protect files, which are not stored in the user’s vault but rather in the Global Private vault controlled by the kernel. Instead, the user’s vault stores file access tickets which are linked with these keys. ‘Fingerprints’ stored in
Figure 3.2: Overview of items stored in a user's vault and their relationships

A user's vault are generated for programs and their dependencies (such as shared libraries) that the user has identified as relevant to their security requirements. These programs are verified upon execution to ensure they have not been modified, which protects the user against malicious code. Application keys, fingerprints and tickets are discussed further in Section 3.5, Chapter 4 and Chapter 5, respectively.

3.2.2 Global Private Vault

The purpose of the Global Private vault (GPRIV) is to store values for which confidentiality and integrity must be maintained. No users, including the privileged prime user, are able to directly access items held in GPRIV. All interactions are indirect and strictly mediated by the security kernel.

A common type of item held in GPRIV is file protection keys. When a user initiates protection (either read or write) of a particular file, a key is generated by the kernel. This key is placed into GPRIV and therefore users never gain direct access to these keys. Instead, a ticket for accessing the key is generated at the same time as the key and it is the ticket that is kept in the user's vault. The ticket is uniquely cryptographically linked with the protection key and access to the protected file therefore depends upon possession of the ticket. Consequently, GPRIV plays a pivotal role in the secure operation of the Vaults model. Some of the different aspects of its operation are illustrated in Figures 3.2 and 3.3 as discussed in the related sections.
3.2.3 Global Public Vault

The Global Public vault (GPUB) contains information that is intended to be publicly accessible in a reliably authentic way to all users on the system. For example, the vault stores globally-applicable fingerprints, users' public keys and encrypted file protection tickets that are being exchanged between users. The role of this vault is to act as an authoritative source that allows the authenticity and integrity of information it makes available to be relied upon. Ordinary users can only read data from GPUB and cannot directly modify the values it stores. All access is therefore mediated by the security kernel.

3.2.4 Escrow Vault

The purpose of the Escrow Vault is to hold copies of escrowed tickets for protected file objects. The Escrow Vault is kept encrypted with a key held in the Fundamental Vault and only the prime user has a ticket to access this key. This means that only the prime user can access the Escrow Vault, which represents one of the special privileges held by the prime user. Note that even with access to the Escrow Vault, the prime user does not gain direct access to the file protection keys themselves but rather to the file access tickets that relate to them. Details of the key escrow mechanism are discussed further in Section 4.8.3.

3.2.5 Fundamental Vault

The Escrow, GPUB and GPRIV vaults must be stored encrypted on the disk to prevent unauthorised access. However, their contents must be accessible by the security kernel when the system boots up. To facilitate this, the keys used to encrypt these vaults are stored in the Fundamental Vault and are this vault’s only contents. This relationship is shown in Figure 3.3. These keys are therefore known as ‘Fundamental Keys’. Unlike ordinary file protection keys, the only tickets that exist for access to Fundamental Keys are held by the prime user. This design is discussed further in Section 3.6.
At boot time, the Fundamental Vault must be decrypted and so the key ('Fundamental Vault key') with which it is encrypted must be securely stored. There are a number of ways that this could be achieved, depending upon how the system is used and its security requirements, and each approach has different advantages and trade-offs. These options are detailed below and summarised in Table 3.1.

**Manual Passphrase** The security administrator or machine operator can supply the appropriate passphrase at boot time. In this scenario, the system will not boot until the appropriate key has been provided. This approach is appropriate for scenarios where the machine is a personal workstation that is only ever used interactively and the administrator (user) is always present at boot time.

**Smartcard or Key Disk** Another approach involves storing the decryption key on a physical device such as a smartcard or, more cheaply, a ‘key disk’. The latter involves storing the key on a floppy, CD-ROM or USB flash drive and could facilitate unattended bootup if physical access to the machine itself is secured. Such a device could also be used for attended bootup for machines that lack physical security, as a more secure and user-friendly alternative to a passphrase.

**TPM Chip** An ideal solution in many scenarios is the use of the TPM chip avail-
<table>
<thead>
<tr>
<th>Property</th>
<th>Manual</th>
<th>Smartcard</th>
<th>Key Disk</th>
<th>TPM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unattended Boot</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hardware Expense</td>
<td>none</td>
<td>moderate</td>
<td>v. low</td>
<td>none</td>
</tr>
<tr>
<td>Req. Physical Security</td>
<td>low</td>
<td>high</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Security Afforded</td>
<td>moderate/high</td>
<td>low/moderate</td>
<td>low/moderate</td>
<td>high</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison between Fundamental Vault decryption key storage mechanisms

able in TCG-compliant systems (Section 2.6.4), which are virtually ubiquitous [198]. This chip incorporates a small amount of on-board secure storage suitable for the Fundamental Vault key. The authenticated boot process facilitated by the chip ensures the integrity of the kernel software before the key is released. This approach allows a high security unattended boot without requiring extensive physical security and with no additional hardware costs for TCG-compliant systems.

3.3 Vaults Public Key Infrastructure

3.3.1 The Need for a PKI

Public key cryptography can facilitate secure key exchange over an untrusted channel and authentication of messages through digital signatures. In the Vaults security model, public key cryptography is used in an analogous way for exchanging privileges between users. However, public key cryptography is problematic in the need to be able to authenticate the public keys themselves in order to prevent man-in-middle attacks [118, pp. 48, 49]. The most common solution for this problem is the use of a public key infrastructure (PKI), where a centrally trusted authority issues certificates binding public keys to specific identities by virtue of the trusted certification authority's (CA) digital signature [199, pp. 270, 271]. Although subject to numerous practical problems and much criticism (cf. [200]), in a decentralised distributed architecture, such as the Internet, few alternative options currently exist.

An equivalent need exists in Vaults as it is necessary for users to be able to crypto-
graphically authenticate one another. Without such cryptographic authentication, an attacker with superuser privileges can potentially impersonate users on the system and thereby obtain unauthorised access to file tickets being exchanged between these parties in what is effectively a man-in-the-middle attack. As these tickets are exchanged using public key cryptography, a PKI allows this exchange to be authenticated.

3.3.2 VPKI Architecture Overview

There are a variety of issues relating to the notion of user identity in computer systems [18, Ch. 14]. The identity of the user as a person typically dictates the rights and privileges assigned to their online identity within the context of the computer system. Also, while the system may use an internal identifier, such as a user ID, in making access control decisions, interactions between different users on the system still depend primarily on that person’s name. This is important because, when one user shares a file with another, they are primarily granting access on the basis of the person’s name and real-world identity, rather than an impersonal user ID. Therefore, in considering interactions and privilege sharing between users, it is important to achieve an authenticative link between the user’s real-world identity and the privileges being shared.

The Vaults Public Key Infrastructure (VPKI) addresses this problem by binding together the user’s identity and their privileges on the system in a cryptographically authenticated manner as represented in Figure 3.4. This allows users to ascertain the real-world identity of the party to whom they are granting privileges, rather than simply dealing with a non-descriptive user ID.

In a more concrete representation, the VPKI links the name of the person with their public key pair (which is the purpose of traditional PKIs), but then further connects the key pair with the vault belonging to that user (Figure 3.5). This association
reflects the fact that it is the contents of a subject’s vault that determines their privileges when attempting to access an object, rather than their identity. However, as indicated above, the user’s identity remains relevant in determining their authorisations, such as when exchanging privileges with other parties. Cryptographically linking these elements together therefore secures the exchange of privileges.

3.3.3 VPKI Certificates

As with most PKI schemes, VPKI uses certificates as constructs to create a verifiable binding between an identity and a public key. Being more specific in purpose, VPKI certificates have a simpler structure than, for example, X.509 certificates [17,120,201]. There are also a number of practical requirements for certificates, such as specification of algorithms and related parameters, but these details are tangential to the model and will not be considered here.

The following describes the structure of the certificate:

**Version** The VPKI version to which the certificate adheres. The structure described here is considered to be “1.0” and the existence of this field allows for future enhancements.

**Identifier** The name of the principal. This should be represented in a manner that is unambiguous and unique within the scope that the certificate is to be used; for example, their full name and title.

**Username** The label used to identify the user on the system. This will be unique only to an individual system.

**User ID** A non-negative integer number uniquely identifying the user’s account on the system.
**Vault ID**  This is a number that uniquely identifies a vault. The construction, properties and semantics of the Vault ID (VID) will be discussed in the next section.

**Certifying**  This field can have the values ‘yes’ or ‘no’ to indicate whether the owner of the key is considered to be a CA and will only be set to ‘yes’ for the prime user. The default value is ‘no’ and certificates cannot be loaded into GPUB with any other value. In order to switch this flag on, the certificate must already be stored in GPUB, which effectively restricts the setting of this flag to the prime user as only they have the privilege necessary to modify the contents of GPUB. Furthermore, modifying this field invalidates any certifying signatures previously appended to the certificate.

**Public Key**  The representation of the user’s public key in an appropriate format.

**Creation Timestamp**  A timestamp indicating when the certificate was created.

**Certifying Signatures**  The digital signatures belonging to those who have verified the ownership of this certificate. Unlike the previous fields, which are static, this field can be changed (appended to) during the life of the certificate and so is not included in the authenticated part of the certificate.

### 3.3.4 User and Vault Creation

The process for creating a new user account, the instantiation of their vault and the initialisation of credentials is important for ensuring system security. However, the best approach is one whereby these three processes are decoupled and do not necessarily happen at the same time. The reasons for this are explained below.

**User Creation**

Before a vault can be created, the new account must first be established by assigning a user name and user ID etc., along with any other system-specific tasks. This
general process will be the same as on systems that do not use Vaults. Separating this process from that of instantiating a vault maintains backward compatibility. This is because it facilitates migration to Vaults without requiring that all user accounts be recreated, and also allows user accounts to exist that do not have a vault. This design also aids security as it reduces the trust that must be placed in the administrator creating the account and, consequently, this step does not need to be performed by the prime user. Once the account is created, the user should then be supplied with the VID belonging to the prime user, which is the only account creation step not independent of Vaults.

**Vault Creation**

After the user’s account has been created, they may log into the system. However, as they do not yet have a vault, the system will behave largely like a non-Vaults aware system. The user may then proceed to create a new vault. The process of creating the vault involves three main stages and an L3 program will be used for automating these tasks.

1. **Key Generation**

   This step involves generating the user’s public/private key pair that will be used for authenticating the transfer of tickets.

2. **Certificate Construction**

   From the public key, the user can generate their VID (as detailed in the next section) and they will now have all the information necessary for the construction of their certificate as described previously. The user then adds their own signature as the first of the certifying signatures.

3. **Vault Instantiation**

   This step involves the user submitting their new certificate to the kernel along with the corresponding private key (proving their ownership of the certificate) and a suitable vault encryption key. On many systems, this is likely to
be a passphrase. However, another approach is to use a smartcard or USB flash drive, which facilitates the use of a high entropy key and two-factor authentication if this is further encrypted with a password. The kernel then verifies the relationship between the private key and certificate and between the public key and the VID. It also performs a number of basic consistency checks, namely that the certificate does not already exist in GPUB, that the certifying flag is not set on the certificate, and that there are no collisions between other items such as username and VID. If the certificate passes these tests, a new vault will be instantiated containing the private key and the calling process labelled as being associated with this vault (Section 3.4.1). A copy of the new vault is written to disk, encrypted with the supplied key. The user's certificate is then added to GPUB.

Finally, the user issues a request to the prime user to sign their certificate. While this request itself can be made through the system, the authentication of the user and verification of their possession of the appropriate private key must be done out-of-band.

### 3.3.5 Vault IDs

The Vault ID (VID) is similar to a User ID (UID) under Unix as it determines the privileges held by the subject. However, the VID is cryptographically derived and also establishes a verifiable link between the user's identity and their credentials as stored in their vault.

The VID is calculated by applying a one-way hash function to the user's public key. This establishes an irreversible binding between the user's identity (as stated on their certificate) and their vault, as the VID is now verifiably linked with both (cf. Figure 3.5). This design is used to control and manage the assignment of privileges to different users on the system by linking tickets directly to a specific vault as identified by its VID (Section 4.3.1). This design also simplifies key verification when certifying public keys. Assuming the user's name is known (a prerequisite...
for the granting of privileges), all that needs to be exchanged is the VID, as this
intrinsically and uniquely specifies both a particular vault on the system and the
user to whom that vault belongs. The VID can then be exchanged between users of
the system via an out-of-band means, allowing the recipient to verify that a specific
public key belongs to the other user and that the vault involved belongs to them.
Therefore, in addition to connecting a user’s identity with their vault, the VID
serves a similar purpose to a key fingerprint in PKIs (such as PGP [202]) as a more
convenient means for specifying and identifying a certificate.

While recent one-way hash functions [107] provide outputs of 256 bits or more, in the
context of the VID such length is largely unnecessary and is an inconvenient number
of characters for users to exchange out-of-band. However, significant attacks have
been reported against earlier, shorter hash functions and there is general movement
towards alternative algorithms [203,204]. In practice, this problem could be resolved
by discarding excess output or using an XOR operation to fold the hash in on itself
to arrive at a suitable, standardised length.

3.3.6 Prime User Certification of Key

After a new VPKI certificate is lodged in GPUB and the user’s vault created, the
user will issue a key certification request to the prime user. Although the exact pro-
cedure involved will vary depending upon the organisation’s structure and security
requirements, the new user must present themselves to the security administrator
(or their delegate) with their VID and suitable documentation necessary to prove
their identity. After retrieving their unsigned certificate from GPUB, the prime
user will verify the details it specifies. The prime user will verify the certificate’s
self-signature, confirm that the name and identity details on the certificate match
the authenticating documentation presented, and verify that the public key hashes
to the VID. If verification of these details is successful, the certificate is signed by
the prime user. This signature is appended to the Certifying Signatures field
and allows others to authenticate the new user.
3.3.7 User Certificate Verification

When users exchange tickets they must authenticate one another. Authentication is achieved by virtue of the prime user's signature on their respective certificates. Each user retrieves the other's certificate from GPUB, along with the prime user's certificate, and verifies the prime user's signature on the certificate. Assuming the prime user account remains secure and the user's identity has been properly verified prior to their certificate having been signed, this represents a cryptographically strong basis for mutual authentication. The only additional requirement in the process of authenticating another user's certificate is to access the certificate revocation list held in GPUB and confirm the user's certificate has not been revoked. Revocation is discussed further in Section 3.3.8.

Certificate Caching

The first time that a user authenticates another, they will need to verify the prime user's signature on the other party's certificate as described above. However, on subsequent occasions when tickets are exchanged between these two parties, it is not necessary to repeat the cryptographic verification of this signature. This is because the other party's certificate can optionally be cached in the authenticating user's vault after the initial verification. An additional timestamp indicating the date that it was cached is attached to the certificate and, within a user-specified timeframe, the certificate will be retained and the public key used for exchange of tickets. This approach improves efficiency by streamlining the authenticated ticket exchange process and also creates an additional barrier inhibiting the subversion of the process. However, it remains critical that the revocation status of the certificate is still checked upon each use.

The Role of Inter-User Certification

While certificate caching streamlines the re-authentication process, it does not create additional trust relationships. However, the VPKI certificate structure allows
for additional certifying signatures to be added, which could be achieved by users
signing one another’s certificates. This process would require users to authenticate
one another out-of-band using their respective VIDs in the same way as is required
for prime user certification. However, inter-user certification in this manner is not
equivalent to certification by the prime user. This is because the model assumes
the security of the prime user account cannot be violated and hence the prime user
is implicitly trusted as a CA. This is not the case with ordinary users whose ac-
counts and vaults are not as well protected and also cannot necessarily be relied
upon to verify the other party’s identity as carefully before signing their certificate.
Therefore a mechanism would be required for users to rate the trust they place in
another’s certifying signature similar to the ‘owner trust’ label in PGP [202]. Such
a mechanism would introduce significant additional complexity and would only be
advantageous if the prime user’s key has been compromised, which would mean
that both inter-user authentication and global system security would have already
substantially failed. Because the advantage of such a feature would be so marginal,
it has not been included in this version of the VPKI although the OpenPGP stan-
dard provides a clear blueprint for how this could be implemented if a need became
apparent in the future.

3.3.8 Revocation of VPKI Certificates

As with most authentication systems, there is a need for the VPKI to deal with
scenarios where an authentication token (specifically a user’s key pair) has been
compromised. In such an event, steps must be taken to limit the impact of this
compromise by publishing the fact that the key can no longer be used securely.
One approach for dealing with such a scenario—sometimes used in other PKIs such
as X.509—is a ‘certificate revocation list’ or CRL [199, pp. 276, 277]. While this
mechanism can be problematic in distributed systems where it can be difficult to
ensure that an up-to-date list is available to large numbers of clients spread over a
wide area, this is not the case in the context of the Vaults PKI where a ready-made,
central repository exists in the form of GPUB. This design ensures that up-to-date revocation information is available from a secure and reliable source to all users on the system.

The VPKI revocation list must be checked upon every action where certificates are either used directly to authenticate another user or where previous authentication is relied upon. The principal example of this is where one user grants access to a file to another by creating a secondary ticket and encrypting it with the recipient’s public key. However, as indicated above, this applies regardless of whether it is the initial authentication for using the certificate or the use of a cached, previously authenticated certificate. The CRL is also checked when a user attempts to access a protected file using a ticket, to ensure that the user’s vault and privileges have not been compromised in this way.

**Implementing VPKI Revocation**

As indicated, a certificate revocation list for the VPKI is stored in GPUB as it has the properties required for such a mechanism. Since certificates are uniquely identifiable by their VID, the CRL simply consists of a list of revoked VIDs. The list is maintained by the prime user but any user can request that the kernel revoke their certificate at any time by virtue of their possession of the corresponding private key. After the revoked VID is added to the list, the security kernel will then remove any unclaimed secondary tickets held in GPUB that have been granted to the revoked VID. All secondary tickets issued to this VID are also revoked. Further details of this process are discussed in Section 4.8.1.

The primary scenario where a user’s certificate needs to be revoked is where their private key has been compromised, possibly as a result of their vault having been compromised also. Revocation prevents an attacker from using their knowledge of a user’s private key to obtain secondary tickets granted to the key’s owner. However, revocation may also be appropriate if the user’s account is being downgraded to one that does not use Vaults for security or if their account is being removed alto-
gether. This ensures they are not inadvertently granted access to cryptographically protected files that they should not be able to access.

3.4 Process Credentials and Vault Session Initiation

3.4.1 Process Credentials and Metadata

In order to determine the privileges and other relevant parameters of any given process, the security kernel maintains a collection of metadata associated with each process currently executing. These new credentials associated with the Vaults model are stored in addition to existing credentials such as user and group IDs. All of these new items are either derived cryptographically or obtained from cryptographically secured sources. The new process credentials are described below.

**VID** The vault ID with which the process is associated. If the process is considered untrusted (L0), or the user has not yet initiated their vault session, this will be blank.

**Parent VID** The VID associated with the process’s parent. When one trusted process executes another, the VID and parent VID (PVID) fields will be the same.

**Specified Trust Level** The STL is the trust level (L0-L3) with which the program will normally execute as specified in either local TCB (LTCB) or global TCB (GTCB).

**Effective Trust Level** The ETL is the actual trust level that the process currently has. There are a variety of reasons why processes may have their trust level set lower at runtime than their STL states and these are discussed further in Section 3.4.4.

**Parent ETL** The ETL of this process’s parent. Abbreviated to PETL.
**Fingerprint** The fingerprint of the verified process or blank if the process has a trust level of L0.

**Runtime Digraph Entry Reference** This field references an entry in the runtime digraph data structure that specifies which other objects on the system this process depends upon for its security. Further details concerning these concepts are discussed in Chapter 5 on Trusted Fingerprinting.

**Protection Status Index** The PSI of the process is used to cryptographically verify the protection status of the executable file and is described in Section 4.5.

**Path** The filesystem path of the executable program.

### 3.4.2 Process Trust Levels

When a new process begins execution, it is assigned a trust level. The actual trust level assigned reflects a number of things, including whether the code has been cryptographically authenticated through Trusted Fingerprinting; the level of privilege the process is intended to have in the Vaults security model; and the trust level of the process’s origins. These issues will be discussed in more detail in the remainder of this section and in Chapter 5. Four hierarchical trust levels apply, labelled L0, L1, L2 and L3. Additional vault access privileges are assigned to each higher level, as illustrated in Figure 3.6.

**Level 0: Untrusted**

Processes at Level 0 (L0) are those with no privileges under the Vaults security model. Their code is not verified by Trusted Fingerprinting and they cannot access application keys or cryptographically protected files. L0 processes have no access to the user’s vault at all. In practice, these processes will usually include newly obtained software, software being tested or developed, and any other minor utilities or tools that require little or no interaction with other software and are not required
to have any access to important data. Any software of uncertain origin (for example, downloaded from an unfamiliar website) or where there are any other doubts regarding the potential behaviour of the program, should not be fingerprinted and will therefore execute as an L0 process. L0 processes therefore have no special access to trusted parts of the system.

**Level 1: Confined Trust**

Level 1 processes (L1) are those that must have some access to security-sensitive data or programs, but whose access is restricted to specifically identified items. For example, many applications maintain their own data and configuration files and have limited need for general access to other protected objects on the system. Particularly if an application is also interacting with data from an untrusted source (for example, from over the Internet) then the application can be confined to its own domain [26, p. 202]. Examples of potential L1 applications include web browsers, electronic mail clients and other non-management related network software. More general application and productivity software, such as word processors, may also be configured as L1 programs, and this may be beneficial where these programs are dealing with externally-obtained unauthenticated or untrusted data.

The confinement of L1 programs is implemented by granting them limited access to specifically identified protected objects, in addition to the unprotected files that
L0 processes may access. L1 programs can only access the application keys and file protection tickets that have been specifically bound to that application. This allows L1 programs to execute with the minimum of privileges required. L1 processes must be fingerprinted and this trust level will be specified by the user in the program’s dependency record. Note that L1 trust levels are designated on a per-user basis and are not set for globally fingerprinted objects.

Level 2: General Trust

Users may also assign a program a more general level of trust by designating it as Level 2 (L2). Such programs can access any protected file for which the user has the required ticket in their vault. However, they may only access application keys that are specifically bound to them. L2 programs must have a trusted fingerprint and again the trust level for the program is indicated in its dependency record. The trust level L2 can be applied to programs both by individual users within their own vault and by the prime user globally. However, the trust level specified by a user for a program will override that set globally. Software that needs general access to a variety of potentially protected files (rather than being limited to a specific subset of files that it is used to access) are designated as L2. Examples of candidate L2 programs include those with a management or administrative role and some general application software that cannot be confined at L1. The setting of the L2 trust level depends upon the user’s practical and security needs, as well as general system security configuration, policies and practices.

Level 3: Unmediated Trust

Level 3 (L3) is a trust level reserved for programs that require administrative access to a vault. While L2 processes may access all protected files for which the user has a file protection ticket, they do not have direct access to these tickets and other contents of the user’s vault. Instead, the selection and usage of the ticket is performed by the security kernel in response to the program’s access request.
However, an L3 process executed by a user has unmediated access to that user’s loaded vault as confirmed by matching the VID of the process with that of the vault.

Because of their security-sensitive nature, L3 programs must be carefully designed and implemented to ensure they can only be controlled interactively by the user and not manipulated by inputs through channels such as command line parameters or environmental variables. L3 applications are therefore designed specifically for administrative tasks within the Vaults security model. In practice, this means there will be relatively few such applications on a system and the L3 trust level can only be assigned by the prime user through setting this value in their dependency record in GPUB.

3.4.3 Session Initiation Process

Prior to Session Initiation

Users log into the system in the same way as on the systems that are not Vaults-enabled. This reflects the backwards compatibility of the architecture as not all users will have vaults. However, some aspects of Vaults will still be in effect as programs specified as being globally trusted (in the GTCB) will still have their fingerprints verified when executed. Therefore the initial login program is globally trusted and executes as an L2 process. However, if the user does not yet have a Vaults session running (or does not have a vault), then there is no significance to this beyond the program having been verified prior to execution. User interface shell processes also execute at L2\(^1\), although other programs spawned from the shell will execute at their globally specified trust levels, or L0 if they have no fingerprint.

However, while there is no Vaults session initialised, there will be no L1 processes as they can only be specified on a per-user basis. Regardless, it is expected that

\(^1\)Specifying these programs as globally trusted at L2 is necessary to allow the subsequent execution of the L3 vault login program and ensures that there is a chain of trusted programs leading up to it. Further details concerning this and related issues are discussed in Section 5.6.
users who have a vault will attempt to decrypt it and initiate a new Vaults session soon or immediately after logging in.

**Vault Session Initiation**

To initiate a new Vaults session, the user executes an L3 vault login application that prompts them for their passphrase or vault decryption key. Note, it is not necessary to prompt for their VID as this can be obtained automatically by searching certificates held in GPUB for their current UID. While the use of a passphrase may be appropriate for low-security scenarios, a superior approach is the use of a randomly generated key stored on an external device such as a magnetic stripe card, smartcard or USB flash drive. Further encrypting the key with a user-selected passphrase is also advisable to obtain two-factor authentication. A physical trusted path mechanism for the input of any passphrases must also be employed where appropriate.

The vault verification and instantiation protocol then proceeds as follows:

1. The kernel locates the vault stored on disk corresponding to the VID associated with the requesting user’s UID. As shown in Table 3.2, vaults are stored on disk with the VID in a plaintext header to facilitate this.

2. After obtaining the vault decryption key $K_V$ from the user, the kernel decrypts the vault that has been located. Note that user vaults are encrypted using an authenticated encryption mode so this step also involves confirming that no tampering with vault data has taken place.

3. The kernel hashes the user’s public key $P$, obtained from their certificate, and confirms that $H(P) = VID$. This step verifies the link between the public key on the certificate and the VID of the vault to be instantiated.

---

2If a passphrase is used then this should not be the same as the user’s main login password due to the risk of dictionary attack against the system password file.
Table 3.2: Overview of Vault structure on disk

<table>
<thead>
<tr>
<th>VID</th>
<th>plaintext</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private Key $p$</td>
<td>(remainder of vault)</td>
</tr>
<tr>
<td>...</td>
<td>ciphertext</td>
</tr>
</tbody>
</table>

4. Next, the kernel calculates $P(z)$; a random number encrypted with the user’s public key $P$. It then decrypts $P(z)$ with the private key $p$ found in the first record of the decrypted vault and confirms the value of $z$. This step verifies that the vault actually contains the private keys corresponding to the VID.

If the vault decryption process is unsuccessful then a message is displayed, the vault login process terminates, and the user is returned to their previous shell. However, on successful authentication, a new shell is executed with its VID set to that belonging to the user’s vault. This VID is then inherited by all subsequent trusted processes and process trust levels set by the user now take precedence over global settings.

3.4.4 Process Credentials Maintenance

When a new process is executed, the security kernel must ensure that the process credentials are correctly maintained and adjusted according to the parameters associated with the new program. Initially the VID, the process’s fingerprint and STL are blank as this is the default for untrusted (L0) programs. The ETL is also set to L0 and the PVID and PETL fields are set to the parent’s VID and ETL respectively. These defaults are set prior to the creation of every new process and apply uniformly.

The fingerprint matching process then occurs to identify whether a fingerprint pertaining to the program concerned exists in either the LTCB or GTCB. Fingerprint matching is described in more detail in Section 5.3. If no fingerprint exists, the program will execute untrusted and its credentials do not need to be modified from the previously set defaults. If the fingerprint matching process identifies a trusted
fingerprint for the program and it is successfully verified, the credentials of the process will be modified. The fingerprint and STL values are set based upon those contained in the verified fingerprint record and the VID is copied from the PVID field. The ETL is then set according to an algorithm designed to prevent a trusted program from being misused by a malicious parent process. Details of this algorithm are described in Section 5.6.3. However, the typical outcome is that the ETL is set to the lower of the STL and the PETL.

**Trust Level Variations**

There are three scenarios where the setting of a process’s ETL may vary at runtime from that just described. If a program accesses one of its data file dependencies where this object has a trust level lower than the program’s current ETL, then its ETL will be lowered to match it. This situation applies, for example, where the program is a command interpreter opening a script file for execution. Here the script may execute at L1 even though the interpreter normally executes at L2. This design ensures that the trust level of scripts reflect the trust level the user intends them to have. Alternatively, a program can voluntarily lower its ETL at any point. While this requires applications to be aware of the existence of the Vaults model, in this scenario programs can effectively reduce their privileges if, for example, they no longer need it.

Finally, users can require that a particular trusted program have its ETL reset to L0 at runtime if it accesses an object that is not explicitly specified as one of the program’s dependencies. This involves setting the **Require Trusted Data** flag on the program’s fingerprint record and is useful for dealing with interpreters which may execute both trusted and untrusted scripts. This flag also mitigates the need for applications to manually request that their trust level be lowered at runtime, which enhances the backward compatibility of the Vaults scheme.

All issues associated with the different scenarios where a program’s trust level will vary at runtime are discussed extensively in Chapter 5. Also note that, in all cases
where a program has its trust level modified at runtime, all files opened by this
process are automatically closed by the kernel. If this does not occur, the process
may retain privileges in relation to the files it can access that are not appropriate
to its new trust level.

3.5 Application Keys

Users can store sensitive values such as passwords or program-specific cryptographic
keys in their vault. These are known as ‘application keys’. Each application key
can be bound to a specific set of trusted applications and only these applications
(along with L3 applications, which are able to access the vault directly) are able to
access that key. The application keys mechanism therefore provides a way for users
to store secrets of various kinds in their vault and have these securely accessed in a
transparent manner by designated applications, thereby improving both usability
and security.

3.5.1 Application Key Entry Format

Each application key entry record in the user’s vault consists of the following fields:

**Application Label** This identifies the application or applications to which the key
is bound and also uniquely indicates the purpose of the key. This field can
have multiple values in the event that the key is used by multiple applications
or by the same application in different contexts. The structure of each value
is described below.

**Fingerprint** This is the fingerprint of the trusted application to which the key
is bound and, as with the label, this field may consist of more than one
value. However, there must be the same number of label and fingerprint
values forming corresponding pairs as determined by their order.
**Key Value**  The value of the key that may be accessed by the specified application. This is stored as provided, with no specific encoding to allow applications to store any value, binary or text, as required. The application is responsible for any encoding or decoding required for this value.

**Description**  A textual description of the purpose of the key for the benefit of the user.

**Flags**  A small number of configuration flags may be used to adjust the way the key access process is managed.

**Label Structure and Default Keys**

The structure of each application label value is shown in Figure 3.7. Essentially the label consists of three parts, separated by colons. These parts are the application’s name, its version, and a free-form descriptor that allows the application to specifically distinguish between multiple keys it may need to access. Note that, as described above, labels and fingerprint values must exist in pairs, as this allows a key to be bound to multiple applications if desired.

As many applications will only need to access a single key, applications may specify a default key. Default keys are indicated by an empty string as the label value. This allows an application to access the default key by requesting a key from the kernel without specifying any label. This design simplifies the task of adapting an application to retrieve keys from a user’s vault. The kernel then searches the user’s vault for a key with an empty label value and a corresponding fingerprint matching that of the application making the request. Note that this means subsequent keys added for that application are required to have a non-empty label.

It is important to note that labels exist only to allow applications to distinguish
between different keys they store in a user’s vault and they have no impact on security. The right of an application to access a given key is determined solely by the fingerprint stored in its process credentials and whether this matches with the fingerprint associated with that key. Therefore, the ability to access an application key is determined on a cryptographic basis.

**Application Key Flags**

Two flags are defined that can be set for application keys.

**Release Trust Fail** If the ETL and STL values are not equal, this indicates that the trust level of the process has been downgraded. In this case, the process’s behaviour may be uncertain and, if this flag is switched off—as is the default—then application keys will not be released to this instance of the program. However, this behaviour can be overridden by the user switching on the flag.

**Confirm Release** Switching this flag on requires that the user be prompted and specifically confirm the release of this key to the application. For keys with particularly high sensitivity, the user may switch the flag, but this is unlikely to be necessary for most keys and therefore the flag defaults to Off.

### 3.5.2 Creating Application Keys

The process of creating application keys may be either driven by the user or by the application itself. Both approaches have similar security properties but apply in different scenarios, depending upon whether the key originates with the user or with the application. In either case, the application key creation process can only proceed where the application process is executing at its STL and has not had this downgraded. Also, application keys can only be bound to natively executable programs and not to non-natively executable programs such as scripts.
User-Driven Key Creation

In some cases, the key to be bound to an application will be provided by the user. Examples of when this applies include secrets such as e-mail or remote login passwords. The procedure for a user-driven application key creation is as follows:

1. The user executes the application to which the key is to be bound and this program registers with the kernel, via a system call interface, one or more label and description pairs for the keys it can accept, according to the formats described previously.

2. Using an L3 vault administration program, the user selects an option to create a new application key.

3. The kernel compiles a list of suitable processes that have registered labels. It searches the fingerprints held in the user’s vault (LTCB) for those which correspond to that specified in the processes’ credentials and, from the fingerprint dependency record, obtains those programs’ execution paths. If a suitable entry is not found, then GTCB fingerprints are also searched. Note, it is not possible to bind application keys to programs that do not have a fingerprint.

4. Using the L3 program from step 2, the user selects the application to which the key is to be bound from a list of eligible processes. The user should confirm that the program’s execution path is correct.

5. The user supplies the key or keys as required for the label and descriptions indicated. They can also customise the Description field as desired.

6. Once this is completed, the kernel instantiates the new application key entry.

Application-Driven Key Creation

In other scenarios, the application key creation process will be driven by the program itself. This will apply where the program has generated the key itself and
subsequently wishes to store the value securely; for example, cryptographic software such as PGP when generating a new key pair. However, it is also common for applications to receive keys such as passphrases directly from the user, which is likely to make the application-driven key creation process more practical and user-friendly in many circumstances. For example, a web browser may wish to establish new authentication credentials when a user first visits a web site that requires them to sign in. In these cases, the following procedure for creating application keys applies:

1. The application creates or obtains the key and submits a key creation request directly to the kernel via a system call. This request includes the label, description and key value itself.

2. The kernel obtains the path for the application by searching on its fingerprint in LTCB and GTCB in the same way as for the user-driven key creation process.

3. The kernel executes an L3 vault administration application and passes it the parameters provided previously and the execution path obtained in the previous step.

4. The user verifies that the parameters of the key creation process are correct and then confirms that the process should proceed, possibly adjusting parameters such as the Description field as desired. If the parameters are incorrect, or the user does not wish the key creation to occur for some reason, they may abort the process at this point.

3.5.3 Application Key Release

The process by which an application gains access to a key is straightforward and is as follows:

1. The application requests the key from the kernel, specifying its label. If the label is blank, this indicates the default key for that application.
2. The kernel obtains the fingerprint, ETL and STL of the process from its credentials. If the process is an L0 process and no fingerprint exists, the key release request is denied.

3. The kernel searches for a fingerprint corresponding to the value in the process’s credentials in the list of application keys stored in the user's vault.

4. Once matching fingerprints have been found, if the ETL and STL of the process are equal, the key corresponding to this label is released to the application. However, if the trust levels are not the same (indicating that the trust level of the process has been downgraded), the key will only be released if the Release Trust Fail flag is set on the application key concerned. Otherwise the key is not released and the request is denied.

3.6 Administrative Privileges

One of the main objectives in developing an alternative, cryptographically-based security model is to correct deficiencies in current models by limiting the unrestricted powers of the superuser and to reduce the vulnerability that this omnipotence causes. Consequently, under Vaults, the superuser cannot, for example, access other user’s protected files without possession of the required ticket. Therefore the model is designed to be able to continue to fulfil its security goals, even if the superuser account has been compromised.

However, confining the superuser in this way means that there are certain administrative tasks that this account can no longer perform without recreating the vulnerabilities that existed before. Therefore, there is a need for a new administrative account, which is known as the prime user.

3.6.1 The Design of the Prime User Account

In creating a new privileged account, it is important that it not become effectively a surrogate for the existing superuser, which inherits all of the same deficiencies
and vulnerabilities. With this in mind, the prime user account has been carefully
designed to deliberately avoid these problems.

A key design goal is that the prime user is strictly an administrative account that
need only be used in a limited number of situations. Although the prime user has
the privileges to potentially subvert system security, the account is not trivially
omnipotent in the manner of the traditional superuser. In addition, the account
is not required for frequent or general usage. As a result, the prime user account
has far fewer privilege boundaries compared with the superuser. Its privileges are
clearly defined, which makes the security properties of the account simple to con-
ceptualise and model. Finally, access to the prime user account is cryptographically
determined, consistent with the stated design principles of the Vaults architecture
given in Section 1.2.2.

3.6.2 Prime User Privileges

The prime user has four special privileges that differentiate it from the traditional
superuser and these are:

1. GPUB Maintenance The prime user has the ability to modify the data stored
   in GPUB. Amongst other things, the prime user can create, delete, update and
   modify global fingerprint (GTCB) dependency records that apply to all users.
   They are also responsible for removing user certificates, deleting uncollections
   secondary tickets and maintaining the CRL.

2. VPKI Certification Authority The prime user is the CA for the system and
   is therefore responsible for ensuring that users can exchange privileges in
   an authenticated manner. The designation of the prime user as the CA is
   indicated by the certifying flag on their certificate held in GPUB (Section
   3.3.3) and therefore this privilege is essentially a derivative of the previous
   one.

3. Kernel and L3 Program Maintenance The prime user is responsible for the
maintenance of trusted system components; namely the security kernel and all L3 programs. This task cannot be delegated to the superuser as subversion of these programs could allow that account to compromise the system. This privilege is implemented by the prime user holding write tickets for these objects allowing them to be modified as required; for example, when upgrading this software.

4. Escrowed Ticket Recovery The prime user has the ability to obtain tickets via the Escrow Vault in order to bypass cryptographic access controls and thereby access protected files belonging to other users. This is equivalent to the superuser’s ability to bypass discretionary access controls, although far more controlled. Further details of key escrow are described in Section 4.8.3.

Note that, as with all users, the prime user does not have any access to GPRIV.

3.6.3 Fundamental Privileges and Tickets

Unlike superuser privileges under discretionary access control models that are based upon the identity associated with the process, all of the prime user privileges are cryptographically determined. The kernel maintenance privilege (3) depends upon the prime user possessing the required write tickets to modify the relevant objects. It is the prime user’s possession of these tickets, rather than anything intrinsic about this user, that results in this privilege. The VPKI CA privilege (2) can only be exercised by a user who possesses the private key corresponding to the public key flagged as a CA’s key as stored in GPUB. However, the ability to flag public keys in this way requires the GPUB maintenance privilege (1) and this, along with the escrowed key recovery privilege (4), are referred to as ‘fundamental privileges’ as they do not derive from any other privilege.

Fundamental privileges are granted to the prime user by their possession of ‘fundamental tickets’. These are different from ordinary tickets as they grant access to one of the system vaults rather than cryptographically-protected filesystem objects.
Fundamental tickets are therefore associated with keys held in the Fundamental Vault (see Section 3.2.5 and Figure 3.3 on p. 61) rather than file protection keys held in GPRIV. Also, unlike file access tickets, fundamental tickets may only be used by processes executing with an L3 trust level. The structure of a fundamental ticket is given as

\[ T_V^F := K_V^F(H(K_V^F, VID_p)), \]  

(3.1)

where the fundamental ticket \( T_V^F \) to access system vault \( V \) is constructed by encrypting a hash of the Fundamental Key \( K_V^F \) for vault \( V \) and the VID of the prime user, \( VID_p \). Unlike ordinary primary tickets used to access protected files (which will be described in Chapter 4), fundamental tickets contain no timestamp. Also, unlike file access tickets, fundamental tickets can only be accessed by highly-trusted L3 processes and cannot have additional or ‘secondary tickets’ generated for them.

Two fundamental tickets are defined corresponding to the two fundamental privileges just described. The first references the key for the GPUB and authorises the prime user to make modifications to this vault, which can already can be read by all users on the system. The second fundamental ticket authorises the prime user for limited use of the Escrow Vault (discussed further in Section 4.8.3).

Since both fundamental privileges are granted based upon possession of their respective fundamental tickets, they could potentially be granted independently to different users. In this respect, the collection of privileges associated with the prime user is not strictly atomic in the manner of traditional superuser privileges. However, it is not clear that a clean division can be made between these privileges and further analysis would be required to determine the security of this arrangement. Therefore, it is assumed in the remainder of this thesis that both fundamental tickets are held in the same vault and that there is only a single prime user account.
3.6.4 Prime User Account Creation

The prime user account is created as part of the process of installing the Vaults architecture. It involves the following steps:

1. **Obtain or generate an encryption key for the prime user’s vault.**
   
   On systems with low security requirements, this key could simply be a passphrase. However, given the significance of the prime user account, a randomly generated key stored on a removable device encrypted with a passphrase for two-factor authentication is a far better approach. Another alternative could be to store the key in the TPM chip and, based upon a secure system state as determined by measured code, biometrics could be used to locally verify the identity of the prime user and subsequently authorise the release of the vault decryption key.

2. **Create vault and key pair.**

   A new vault for the prime user must be instantiated, followed by the generation of a public/private key pair and calculation of the VID, in the same manner as for non-privileged users. This vault is then written to disk encrypted with the key obtained in the first step.

3. **Fundamental Vault construction.**

   The Fundamental Keys used to encrypt the system vaults are generated and the Fundamental Vault created. These keys are then stored in this vault.

4. **System vault construction.**

   The system vaults (GPUB, GPRIV and Escrow) are created. These vaults are then encrypted with the Fundamental Keys.

5. **Fundamental ticket construction.**

   The fundamental tickets to access the GPUB and Escrow Vaults are constructed and stored in the prime user’s vault.
6. **Kernel protection.**

A list is compiled of the filesystem objects making up the security kernel and its dependencies. Write protection tickets are generated for them and stored in the prime user’s vault. Specialised vault administration programs (L3 applications) are also write protected in the same way.

7. **Prime user certification.**

An unsigned certificate is constructed for the prime user based upon the public key generated previously and is then loaded into GPUB. The prime user uses their fundamental ticket for GPUB to modify the contents of this vault to set the `Certifying` flag for this certificate and append a self signature to the certificate copy held in GPUB. The private key is then stored in the prime user’s vault and bound to the L3 application used for signing the user’s certificates, limiting access to the key to this and other highly trusted L3 programs.

### 3.6.5 Prime User Security

A comparison between the traditional superuser and prime user accounts demonstrates the security advantages of the latter.

Superuser privileges are granted to a process through an identity-based label that is assigned, whereas prime user privileges require the possession of the relevant tokens in the form of the fundamental and kernel write access tickets. This property reflects Vaults design as a locks and keys access control scheme as discussed in Section 2.5.2. In traditional models, the full set of superuser privileges are automatically held by all processes executing with this label and typically include system and network servers, set UID programs, and programs executed directly by the superuser. In contrast, prime user privileges can only be utilised by highly trusted L3 processes spawned from one that has decrypted the prime user’s vault and therefore prime user privileges can only be held as part of an interactive login session. There are also
a number of in-built mechanisms that guard against malicious processes obtaining access to these tokens. For example, the ability to access the prime user’s vault is automatically lost if that user executes an L0 program (one which has no fingerprint and has not had its integrity cryptographically verified), as these processes cannot access any user’s vault. Even L2 programs, which have been cryptographically verified prior to execution, are not able to use the fundamental tickets that underpin the prime user’s privileges. In this way, there is extremely limited scope for use of these tokens by malicious code.

However, the exposure of the prime user account is also limited by the relatively few situations where it must be used. In contrast, superuser privilege is required for any privileged operation, regardless of how trivial. Only a small number of these tasks involve code being directly and interactively executed by the superuser with the majority being either non-interactive (such as with network servers) or interactive but executed by non-privileged users (such as set UID programs). In both of these scenarios, highly privileged superuser code is interacting with non-privileged subjects and is therefore highly exposed to numerous untrusted inputs. As a result, there are many privilege boundaries and increased opportunity for attackers to discover and exploit vulnerabilities. By comparison, prime user privileges are only required for a very small number of administrative tasks that arise infrequently. All of these tasks are interactive in nature and have very few, if any, privilege boundaries through which prime user privileges are exposed to attack.

**Object- and Subject-Based Higher Privilege Models**

Indeed, the most obvious means of attacking the prime user account involves targeting their encrypted vault stored on disk as this encapsulates all of these privileges. However, if a sufficiently secure encryption key has been used, this will be an extremely difficult, and very likely impossible, task. This result highlights a fundamental reason why the prime user account is more secure than the traditional superuser approach. Superuser privileges are held by a wide variety of subjects on
the system and it is extremely difficult, in practice, to isolate them from untrusted inputs. Furthermore, superuser privileges may be granted to a subject without requiring any authentication, as in the example of set UID programs. In contrast, the bulk of the time, prime user privileges exist as a static object and therefore have very few privilege boundaries as avenues for attack. Furthermore, access to prime user privileges always requires cryptographic authentication and the application key and Trusted Fingerprinting mechanisms effectively limit privileges to specifically designated, cryptographically authenticated code.

Interestingly, superuser privileges also are encapsulated as static objects, such as the password file. However, the superuser account is rarely directly compromised in such a way, as this requires the attacker to have already successfully elevated their privileges. This observation demonstrates how objects typically have fewer privilege boundaries than subjects and it follows that it is easier to develop a secure model for managing higher privileges if they depend upon possession of tokens rather than identity-based labels. This observation further highlights the advantages that cryptographically-based security models have over traditional approaches.

### 3.7 Summary

This chapter has described a number of mechanisms that exist to support the Vaults security model. The most fundamental element of this supporting infrastructure are the different types of vaults themselves; namely, user vaults, the Global Public vault (GPUB), the Global Private vault (GPRIV), the Escrow Vault and the Fundamental Vault. The basic Vaults PKI, which provides authentication to support secure exchanges of privileges between users, is also critical to facilitate secure operation of the model. Other underlying elements, such as process trust levels and application keys, have also been described. Finally, the design for managing administrative privileges within the Vaults model has been detailed. Privilege management is particularly vital to ensure that past mistakes concerning management of higher privileges are not repeated with Vaults.
The next two chapters in this thesis describe the two major security features provided by the Vaults model; namely, cryptographic file access controls and the Trusted Fingerprinting mechanism for authenticating trusted code. While these features play a prominent role in providing the enhanced security of the model, they remain heavily dependent upon the infrastructure described in this chapter. Consequently, this chapter has provided important background prior to elucidating these other major security mechanisms.
Chapter 4

Cryptographic Enhanced Access Controls

4.1 Overview

This chapter describes the cryptographically enhanced access controls that form a major component of the Vaults model. These access controls are designed to meet the security goals of confidentiality and verifiable integrity for read and write protected objects (respectively), even when faced with a privileged attacker. Furthermore, the access controls allow the independent setting of read and write protection on designated objects, controlled sharing of access between users, secure revocation, and non-transferability of privileges between unauthorised users.

Vaults cryptographically enhanced access controls are not dependent on identity like traditional models and are conceptually straightforward to use. The access control model uses a dual locks and keys approach with a file access ticket held by the user in their vault that is cryptographically linked with a key stored in the global private vault (GPRIV). This key, in turn, is used to secure the file and the security kernel will not grant access to the file without a ticket that references this key. In order to achieve of the stated security goals against a privileged attacker, the scheme incorporates a mechanism for authenticating the metadata that stores
the protection status of any given file on the system and thereby satisfies the overall security goal of verifiable integrity. Further details concerning the design and operation of the Vaults cryptographic enhanced access controls will be described in the remainder of this chapter.

### 4.1.1 File Protection Modes

Users may protect their files on the system, both in terms of confidentiality (controlling read access) and integrity (controlling write access). A read-protected file is encrypted with a symmetric cipher using a randomly generated key $K_{fr}$ and this key is held in GPRIV. A read ticket $T_{fr}^1$ referencing this key is then stored in the user’s vault. Once the file’s contents are encrypted, accessing it requires a valid ticket that references the correct decryption key held in GPRIV. If the user does not have such a ticket, the kernel will deny access. Even if the user is sufficiently privileged as to be able to bypass the kernel and directly access the contents of the disk, this will only give them access to the ciphertext.

A different process applies to write-protected files. A message authentication code (MAC) is generated for the data in that file $MAC(f)$ and this, together with the randomly generated MAC key used $K_{fw}$, is stored in GPRIV. A write ticket $T_{fw}^1$ referencing this MAC is then stored in the user’s vault. If a user attempts to open the file for writing, the security kernel will recalculate the MAC based upon the key referenced by the user’s ticket. Access is only granted if the MAC verification is successful.

While file protection modes may be enforced independently of one another, access via one mode may interact with the other protection mode. This property may impact on how objects are accessed. For example, writing to a read protected file will result in the kernel transparently encrypting this data, even if the user does not have read access. Alternatively, if a write-protected file is opened for reading, the kernel will transparently verify the MAC prior to granting access. This behaviour ensures that the verifiable integrity property, as defined in Section
1.2.2, is maintained. Also note that, for files that are both read and write protected, possession of the ticket for one mode of access does not in any way permit access for the other mode. This means, for example, a user with only write access to a file that is both read and write-protected may make modifications but not view the results.

4.1.2 File Protection Key Binding

For a process to access a cryptographically protected filesystem object, the program must have been assigned a fingerprint and have had this verified upon execution. Therefore, applications without fingerprints cannot access protected files. Further details of the Trusted Fingerprinting mechanisms are described in Chapter 5.

However, possessing a fingerprint does not guarantee that an application will be able to access any given protected file. While programs assigned a trust level of L2 and above can access all protected files, those designated as L1 can only access those protected files whose keys have been specifically bound to that application. This feature can be used for confining applications so that they can only access those protected files that are necessary and thereby reduces the amount of trust that needs to be placed in that program. Owners of protected files can bind their protection keys to any application with a fingerprint. This can be done either when the file is first protected or at any point while it remains protected. There is no limit to the number of protection keys that can be bound to a given application; protection keys can be bound to multiple applications as required. Further details on file protection key binding are given in Section 4.7.1.

4.2 General Filesystem Issues

4.2.1 Relationship to Identity-Based Access Controls

Vaults cryptographic enhanced access controls are designed as an additional layer on top of existing identity-based access controls. Being independent from them,
and being based upon the contents of the user's vault rather than their identity, limits the power of the traditional superuser and mitigates the numerous security problems that stem from this. Therefore, in order for a user to access a protected file they must have both the necessary privileges according to its identity-based file permissions and also the required ticket. This allows the enhanced access controls to be used to enhance security for important files according to users' requirements. Finally, the enhanced access controls are designed to be semantically similar to existing identity-based controls. In general, the same rules and principles apply to the behaviour of these access controls compared with existing mechanisms, which minimises the additional complexity users must deal with.

4.2.2 File Ownership

Only a file's owner can initiate cryptographic protection of it and, once the file has been protected, the kernel will not permit its ownership to be changed. This applies even if the party attempting to change ownership is the current owner or the superuser. Therefore, if a protected file's ownership must be transferred to another user, the file must be first unprotected. At this point, the traditional rules regarding ownership changes apply again. Note that ownership credentials associated with a given file are not cryptographically protected and so a privileged attacker can directly modify these values on the disk. However, such tampering has no impact on the security of the file as it remains protected and only accessible to authorised parties who possess the required ticket.

4.2.3 Protecting Directories and Their Contents

Protecting a directory involves abstracting it to be a file containing a list of directory entries and corresponding inode numbers [205, pp. 107-111]. Therefore, read protecting a directory means that it is not possible to view a list of the contents of a directory without the required read-access ticket. Conversely, a write-protected directory cannot be modified such that files cannot be created, deleted or renamed
within it without possession of the appropriate ticket. Having directories that are write protected is particularly important to ensure that an attacker cannot remove directory contents and replace selected files. These semantics are consistent with those used by existing Unix-based systems.

However, protecting a directory normally has additional implications in the context of Vaults. As directories contain a collection of files related to the same purpose, protecting a directory implies that these objects should receive the same protection. These semantics are applied under the Vaults model such that the specified protection mode is applied recursively to the protected directories contents. Furthermore, newly created objects in protected directories automatically inherit the protection mode of their parent directory. This means that the user does not need to continually manually protect files they create while working. Also, files created by applications without the user’s knowledge will be automatically protected if this occurs within protected directories. Naturally, users can override these default protections for specific files as required.

Aside from the security implications of protecting the directory itself, and the automatic application of a directory’s protection modes to newly created files within it, protecting a directory’s contents is largely a user interface issue. Even if an entire directory’s contents are protected simultaneously as part of the directory itself being protected, the outcome is identical to as if each file had been individually and manually protected by the user. Therefore, protecting directory contents involves creation of separate protection keys and access tickets for each file. This design avoids the significant negative performance implications of sharing a MAC key among a large number of files that may be modified independently.

4.2.4 Moving and Deleting Files

The tasks of moving and deleting cryptographically protected files are straightforward except for the determination of whether a given file is currently protected or not. Details of a mechanism and protocols used to verify the protection status of
a given file are described in detail in Section 4.5, with protocols for moving and deleting files given in Section 4.5.3. Beyond these steps, all that is required are those system-specific tasks that normally apply for the deletion of the file, such as unlinking the directory entry and removing inode metadata if no more links to the data exist for Unix-like filesystems [205, pp. 95, 96]. As discussed in the previous section, deleting a file requires write access to the directory in which it resides.

Other than the determination of protection status, the procedure for renaming or moving files is also not affected at all by cryptographic file protection except for updating path metadata values associated with that object. Details of these metadata are described in the following sections. Finally, the user interface should warn the user if they attempt to move a file to a filesystem that does not support cryptographic access controls as this will result in the loss of that file’s protection.

4.3 File Access Tickets

Tickets are the tokens used in Vaults enhanced access controls to determine whether a subject will be granted access to a specific protected object. Although files are cryptographically protected by keys, the keys are stored in GPRIV and no subjects (including the prime user) have direct access to them. Instead, it is their possession of a ticket that references a specific key which determines whether they may access a particular protected file. This relationship is illustrated in Figure 4.1. The layer of indirection provides greater flexibility and allows the security kernel to control the way users access files. This design allows Vaults to provide the properties of traditional, non-cryptographic access control schemes, namely: independent protection modes, control data sharing, secure revocation, universality and non-transferability of privilege, as detailed in Section 1.3.2.

Each ticket grants either read or write access to a specific protected object. Therefore, in order to have both read and write access to an object protected with both modes, one ticket for each mode is required. There are also two different classes
of ticket. Primary tickets are those received by the owner of an object when it is initially cryptographically protected. In addition to granting access, possession of a primary ticket authorises the holder to issue additional tickets of the same mode for that object to other users, to revoke access and to remove the object’s protection. Additional tickets issued to other users are secondary tickets. Unlike primary tickets, these only confer the right to access the object in the specified mode and cannot be used to construct additional tickets.

### 4.3.1 Ticket Security

Even though keys secure protected objects, it is the possession of a particular ticket that determines whether a subject can access an object. Security of file access tickets is therefore critical and there are several significant security issues that need to be addressed in designing tickets.

In particular, it is important that the owner of an object (the holder of the primary ticket) is able to revoke secondary tickets from other users when desired. As recipients of these tickets will hold them in their vaults, there are few intrinsic restrictions on what they can do with them. Consequently, another critical issue is ensuring users do not duplicate secondary tickets they receive and pass these privileges on to other parties. Preventing unauthorised transference of tickets therefore requires linking a secondary ticket unequivocally and irrevocably with its specific recipient.

One means of achieving these goals is to link the ticket with the recipient’s identity, which has the advantage that it reflects the way users grant privileges to others based upon the intended recipient’s identity. However, this approach con-
flicts with Vaults design principle that security decisions must be made based upon
cryptographically-derived parameters rather than identity-based labels like those
that play a role in many of the security deficiencies of existing systems.

The Vaults PKI provides an ideal alternative. The PKI mechanism binds a user’s
authenticated identity to a specific vault as identified by the Vault ID or VID
(Section 3.3.4). Therefore, linking a secondary ticket with a specific VID allows
a user to both grant access based upon their notion of the recipient’s identity (as
described on their certificate) and have the access token bound to a cryptographic
representation of the identity. Subsequent access control decisions are therefore
made based upon the contents of a user’s vault and not on the identity labels
assigned to a specific process. Furthermore, access tickets are cryptographically
bound to a specific vault and attempts to duplicate and transfer them to other
vaults will render them useless, regardless of any claimed non-cryptographically
derived identity labels of processes attempting to access that vault.

This approach also partially addresses the issue of revocation, as revocation of a
specific secondary ticket can be performed on the basis of the VID to which the
ticket is bound. However, this is not a complete solution and further details of the
revocation mechanism will be described in Section 4.8.1.

4.3.2 Primary Tickets

Primary tickets are those issued to the owner of an object when they initiate cryp-
tographic protection of that object. Possession of a primary ticket grants the user
the following privileges:

- access to the specific protected object to which the ticket relates according to
  the ticket’s access mode;

- create secondary tickets of the same mode that can be issued to other users;

- revoke secondary tickets belonging to specified users so that they may no
  longer access the protected object in that way; and

101
• remove protection from the object for the specified mode.

Primary tickets have a number of important properties. Like all tickets, they are associated with a specific cryptographic key used to protect an object in a specific mode. Therefore, a ticket must uniquely reference this key and no other. Primary tickets are also constructed in a manner that clearly demonstrates their status as a primary ticket. Therefore, it is not possible for a secondary ticket to be mistaken for a primary ticket and vice versa. Furthermore, it should not be possible for a secondary ticket to be transformed into a primary ticket. Finally, unlike secondary tickets, primary tickets do not have the same security issues as they are stored in the vault of the object’s owner. However, they are still bound to this specific VID and therefore are useless if an enterprising attacker manages to steal them.

A primary ticket for a given file is constructed as:

\[ T^1_f := K_f(K_f, H(VID_o, K_f, t_1)), \tag{4.1} \]

where \( T^1_f \) is made up of the protection key \( K_f \) to which it links and a hash of the VID of the user who protected the file \( V ID_o \), the protection key \( K_f \) and the time the original protection key was created \( t_1 \). These values are encrypted with \( K_f \) using an authenticated encryption method. Note that the small size of the ticket contents means that the cipher used should be carefully selected for maximum key agility, meaning that some ciphers will be more appropriate than others [12,100,206].

### 4.3.3 Secondary Tickets

Secondary tickets are those granted to another user by a user who possesses a primary ticket for that object. Note that, although the user holding the primary ticket will normally be the owner of the object in question, the ability to issue secondary tickets for an object is predicated entirely on the possession of the primary ticket. As with primary tickets, secondary tickets can be either for read or write access.
As discussed in Section 4.3.1, a number of security issues exist with secondary tickets because they represent limited access privileges granted to a user for a certain amount of time. Therefore, secondary tickets must have certain security properties. In particular, secondary tickets must:

- reference a specific key uniquely and not be able to be modified to reference any other key;
- be bound to a specific vault and not be usable if transferred to a third party;
- not reveal any information about \( K_f \);
- not be able to be converted into or mistaken for a primary ticket;
- be atomic in that the ticket holder cannot modify or manipulate individual parts of the ticket;
- not be able to be constructed by any unauthorised party; and
- not facilitate or assist in any way in preventing the effective revocation by the holder of the primary ticket of the access granted by the secondary ticket.

In order to achieve these properties and withstand a number of attacks that were identified during the design process, a separate key is used for encrypting the contents of secondary tickets. This is known as the ‘ticket key’ and is given as:

\[
K_T^f := H(VID_r, K_f, t_1),
\]

where the ticket key is designated \( K_T^f \) and is constructed by hashing the VID of the intended recipient \( VID_r \), the protection key \( K_f \) and the protection timestamp \( t_1 \). This design ensures that the ticket key is unique within the context of these three parameters.

The structure of secondary tickets is constructed as:

\[
T^f _2 := K_T^f(t_2, t_E, H(VID_r, K_f, t_1)),
\]
where \( t_2 \) represents the timestamp of the ticket’s creation and \( t_E \) is a second timestamp specifying an optional expiry time. The remainder of the secondary ticket consists of a hash of the VID of the intended recipient \( VID_r \), the protection key \( K_f \) and the protection timestamp \( t_1 \). These values are all encrypted with the ticket key \( K_Tf \) using an authenticated encryption method. This design is intended to be extremely robust and resist a variety of attacks by a highly privileged user. Further details of a number of attacks on secondary tickets modelled as part of the analysis of the Vaults scheme, and the subsequent results that were used to refine this design, are given in Section 7.3.

The ability for users to set an expiry date on secondary tickets reduces the management burden on the owner of the object to review the tickets they have granted to other parties, and manually revoke those no longer required. This is a feature of the Vault's enhanced access controls that does not exist on most traditional access control list schemes. Alternatively, the timestamp can be set to null, indicating that the ticket never expires.

The ability of a recipient to use a secondary ticket may also be terminated if the ticket is revoked by the user who issued it. The kernel is able to check if a ticket is valid by searching a list of ‘Ticket IDs’ (TIDs) in GPRIV. A TID is constructed by hashing the ‘Key ID’ (KID—explained in Section 4.4.1) for the protected object \( KID_f \), the VID of the recipient \( VID_r \) and the timestamp of the secondary ticket \( t_2 \).

The equation for the construction of the TID is therefore:

\[
TID'_f := H(KID_f, VID_r, t_2).
\]

(4.4)

Further details on the revocation process are described in Section 4.8.1.

### 4.4 Ticket and Key Metadata

File access tickets and protection keys have a set of data associated with them that supports the functioning of the access control scheme. Metadata relating to a
specific ticket are stored alongside that ticket in the user’s vault. While the user may modify these values, the scheme is designed so that such modifications do not enable the user to elevate their privileges in any way. However, some data associated with keys must be protected against modification so these parameters are held in GPRIV. A diagrammatic overview of the metadata associated with tickets and keys is shown in Figure 4.2. Details concerning the elements presented in the diagram are explained in the following sections.

4.4.1 The Key ID

A critically important part of ticket metadata is the key ID (KID). This is used to link a ticket, the specific key that it refers to, and the file protected with that key. As separate file protection keys are used for read and write access modes, separate KID values apply for each mode. These are designated $KID_{fr}$ and $KID_{fw}$, where relevant. The KID for a given file $KID_f$ is constructed by hashing the protection key for that file $K_f$ and the timestamp when the key was created $t_1$:

$$KID_f := H(K_f, t_1).$$ (4.5)

Protected files have a KID associated with them to identify which key or keys protect a given file. However, this association uses an authenticated mechanism to prevent the metadata from being manipulated by a privileged attacker writing directly to the disk. Details of the authentication mechanism used are discussed in Section 4.5. As shown in Figure 4.2, KIDs are also used to label keys in GPRIV and tickets in users’ vaults. The data stored in GPRIV cannot be modified by any user and is therefore safe from manipulation. However, while users can modify the contents of their vaults, altering the KID associated with a ticket does not make the ticket effective for accessing any other file since any such action does not change the key with which the file and ticket are encrypted.
Figure 4.2: Ticket and key metadata overview
4.4.2 Primary and Secondary Ticket Metadata

For both primary and secondary tickets, the following metadata items are held in the vault of the user who possesses that ticket:

**Ticket Class** A flag to indicate whether this is a primary or secondary ticket.

**Ticket Mode** The access mode of the ticket, either read or write.

**Key ID** A unique identifier of the key that the ticket references.

**Label** An optional description identifying the path of the file.

The purpose of the label is to remind the user to which file a ticket relates. By default, this is the path by which the user accesses the file. However, users may set this field to any value they like as it has no impact on system behaviour.

Again, note that changing the ticket class, KID and mode values in ticket metadata does not assist a would-be attacker, as this does not alter the encrypted contents of the ticket. Secondary and primary tickets are easily distinguishable by their different lengths and structures and their contents cannot be changed without knowledge of the file protection key. Similarly, ticket mode metadata values do not change which key the ticket references and—as read and write keys are separate and not interchangeable—if a ticket were to be used for the wrong access mode, access would simply fail.

4.4.3 GPRIV Protected Data

File protection keys and metadata related to them are held securely in GPRIV as shown in Figure 4.2. Therefore, unlike the metadata stored in the user’s vault, these values are securely maintained by the kernel. Also, unlike ticket metadata, not all fields are required to exist for every key. The following data are required for all protection keys:

**Key** The value of the cryptographic key $K_f$ protecting the file.
**KID** The unique identifier that allows keys to be referenced by the kernel.

**Timestamp** The time \( t_1 \) that the key was created and this particular instance of protection for the file initiated.

**PSI** The protection status index for this file. This value allows unique identification of the file object within a given filesystem and is used as part of the scheme for authenticating the protection status of a file. Further details are given in Section 4.5.

The optional data that may be present for a given key are:

**MAC Value** The value of the MAC for a write-protected file \( MAC(f) \). This value will be present if the key pertains to write access.

**Bound Applications** A list of hash values of programs to which this particular key is bound. Applications with an L1 trust level must have their fingerprint listed in this field in order to access a given protected file in the specified mode. This will be discussed in more detail in Section 4.7.1.

\(<VID_i, TID_j>\> A list of VIDs and corresponding TIDs representing secondary tickets that have not been revoked. This metadata only applies to secondary tickets.

### 4.5 Verifying File Protection Status

The question of which metadata is stored in the filesystem to link filesystem objects with specific keys and tickets was not addressed in the previous section but is critically important to ensure the security goals of Vaults are achieved. The naive approach of storing KIDs in filesystem metadata is vulnerable to illicit modification by a privileged attacker. If this metadata cannot be trusted, it becomes impossible to ascertain a file’s protection status with certainty and this, in turn, enables attacks on both the confidentiality and integrity of protected data.
The simplest attack on integrity would involve tampering with KID values and then re-writing file contents. As the protection status of the file cannot be determined, the file appears unprotected and the modifications cannot be cryptographically detected. Consequently, the verifiable integrity requirement cannot be met. A similar attack on read protection also exists. By removing the indication of read protection and re-writing file contents, future legitimate modifications or additions to the file will not be encrypted. If the user does not identify this, the attacker can gain access to these later, unprotected modifications. Furthermore, a race condition is likely to exist where an attacker removes the read protection designation of an encrypted file after it has been read into memory in plaintext but prior to the data being rewritten to disk after modification. If the kernel cannot, at this point, identify that the file is read protected, the data may be inadvertently written in plaintext.

Further analysis identified the existence of other more sophisticated substitution and rollback attacks, and a robust, cryptographic approach to ascertain protection status is therefore required. The security goal of this mechanism is to allow the protection status of a given file to be verified in such a way that an attacker cannot undetectably tamper with this information. The mechanism must be able to detect attacks such as creation of false protection status information, substitution of protection status information with that belonging to another file, and rolling back a file’s protection status to a previous, legitimate state. This goal is a challenging one as filesystem metadata is effectively untrusted in the presence of a privileged attacker, as is assumed. Protection of this data must therefore be passive in nature, according to Gifford’s [127] notions of active and passive enforcement mechanisms, as discussed in Section 2.5.2.

4.5.1 Filesystem Metadata

The filesystem metadata for each object must store information allowing the file’s protection status to be verified. This is in addition to existing metadata required
by the system. While the principles are general, the construction of the protection status verification mechanism requires consideration of some low-level details of some properties of the target filesystem. The description given here assumes a Unix-like filesystem involving an inode structure for each individual object but with the possibility of multiple directory entries or ‘links’ referencing that inode [205, pp. 92–95]. However, the design of the mechanism is not dependent upon such a structure; in fact the design would be slightly simpler when implemented on top of a filesystem where there is a one-to-one relationship between file objects and directory entries.

Table 4.1 summarises the additional filesystem metadata relating to Vaults enhanced access controls and protection status verification that is stored in each inode. The majority of these are value pairs containing the protection status authenticator (PSA) $PSA_{f_x}$ and the filesystem path $p_{f_x}$. Each pair relates to a different directory entry or link corresponding to a specific inode on a given filesystem with the number of pairs matching the link count. The PSA value is cryptographically generated and its construction is described in the next section. Path details relating to different directory entries are not normally stored in Unix inode metadata; however, this is necessary in order to efficiently determine all of the paths corresponding to a specific inode without requiring an intensive search of the filesystem. As the vast majority of files only have a single directory entry\(^1\), files with more than one PSA-path pair will be relatively uncommon in practice. The inode metadata also stores the time $t_{PSI}$ when the protection status of the file last changed. If the file is unprotected and has never been protected, this will be the time when the Vaults architecture was

\(^1\)An informal experiment conducted on Linux workstation using the `find(1)` program found that out of 248,639 files, a total of 248,203 or 99.8% had only a single link.
installed and the PSA values initially generated. Finally, the number of directory entries (links) \( n_f \) corresponding to this inode is also stored.

### 4.5.2 The Process Status Authenticator

The PSA is used to derive a Protection Status Index (PSI) that references an entry in the Protection Status Table (PST) held in GPRIV, as illustrated in Figure 4.3. The mechanism is secured using the protection status key \( K_{PS} \), which is also stored in GPRIV.

The PSA is constructed as given in Equation 4.6. A hash is calculated of the protection status key \( K_{PS} \), the access path \( p_f \), the inode number for the file in question \( i_f \), the number of links (directory entries) for this file \( n_f \) and the time its protection status last changed \( t_{PSI} \). This is XOR'd against the PSI for this file (padded with leading zeros as required) and the result encrypted with \( K_{PS} \):

\[
PSA_f := K_{PS}(PSI_f \oplus H(K_{PS}, p_f, i_f, n_f, t_{PSI})).
\]  

(4.6)
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PSI_f$</td>
<td>The protection status index (PSI) for that file.</td>
</tr>
<tr>
<td>$KID_{fr}$</td>
<td>The Key ID for the file's read protection key.</td>
</tr>
<tr>
<td>$KID_{fw}$</td>
<td>The Key ID for the file's write protection key.</td>
</tr>
<tr>
<td>$t_{PSL}$</td>
<td>The timestamp of the last protection status change.</td>
</tr>
</tbody>
</table>

Table 4.2: List and summary of Protection Status Table fields

Note that as each link for a given file results in an additional path, a different PSA must be generated for each. However, the PSI value remains identical so the same entry in the PST is referenced regardless of the path by which a file is accessed.

The construction of the PSA is designed to resist a variety of attacks by a privileged user who can write directly to the disk, bypassing the kernel. Without knowing $K_{PS}$, an attacker cannot either decrypt the PSA or recalculate its hash component in order to modify parameters such as path values or timestamps. However, both of these tasks must be accomplished in order to change the PSI value relating to a given file or modify the parameters associated with that file. This design cryptographically protects these parameters, which are also stored in plaintext in the inode metadata. Finally, XORing the hash component with the PSI value combines them together to ensure the atomicity of the PSA value. This prevents the substitution of selected parts of the PSA ciphertext for one file with that belonging to another in order to mis-apply the latter’s protection status to the former.

Upon access to a file, the PSA and file parameters ($i_f$, $n_f$, $t_{PSL}$) are obtained from the inode metadata. Note that the path name $p_f$ used in this calculation is that received by the kernel in the user’s access request. Using this value means that the PSA calculated will correspond with the file the user seeks to access. In conjunction with the protection status key $K_{PS}$ held in GPRIV, the hash component $H(K_{PS}, p_f, i_f, n_f, t_{PSL})$ is calculated. The PSA value is then decrypted using $K_{PS}$ and this is XOR’d against the previous result, giving the PSI value for that file according to the equation:
\[ PSI_f := (PSI_f \oplus H(K_{PS}, p_f, i_f, n_f, t_{PSi})) \oplus H(K_{PS}, p_f, i_f, n_f, t_{PSi}). \] (4.7)

Once the PSI for a given file has been obtained, this references a specific entry within the PST as shown in Table 4.2. Each entry specifies the KIDs, both read and write (\( KID_{fr} \) and \( KID_{fw} \)), belonging to that file, allowing its protection status to be reliably determined. The timestamp of the last protection status change is also stored here and this is cross-checked against the value from the inode metadata that was used to generate the PSA to ensure that a rollback attack has not occurred.

Assuming a secure hash function and sufficiently large output length, the probability is vanishingly small that any manipulation of filesystem metadata parameters will lead to the calculation of a valid PSI. Consequently, if no entry is found in the protection status table for a calculated PSI, this reveals tampering with filesystem parameters and access to the specified file is halted.

### 4.5.3 Protection Status Protocols

The following are specific scenarios where the protection status of a file is relevant and must be verified before proceeding. Descriptions are given of the steps required for each scenario in terms of a set of preliminary protection status verification routines, which are outlined in Appendix D. Names of routines are distinguished by the use of a monospaced font, for example AddPSA. Low-level filesystem issues are not considered; however, the use of journaling to ensure reliability, as found in many contemporary filesystems, is anticipated. Note that the design of these protocols is such that race conditions and other attacks on security by interfering with the completion of these protocols will leave the system in an inconsistent state. As a result, the attack will be detected upon next access to the file concerned and the goal of verifiable integrity is thereby achieved.
Vaults Installation

The filesystem is recursively scanned to identify all directory entries and inodes. New entries in the PST are created for each inode with PSI values being obtained incrementally, taking care to ensure they do not repeat, and the timestamp being initialised as the current time. As no files will be protected at installation time, all KID values held in the PST are initially 0. PSA values are then calculated for all directory entries and inode metadata extended to include all PSA-path pairs for that inode and the corresponding timestamp for the object held in the PST using the AddPSA routine. This process is then repeated for all local, compatible filesystems with separate PSTs used for each.

File Creation and Access

Creating a new file also requires creating the appropriate protection status metadata. This involves obtaining the next available PSI value incrementally\(^2\) and creating a new PST entry. After this is done, the PSA for the new file can be generated using the available PSI, path, inode, link count and timestamp parameters and these are written to the inode metadata (AddPSA). The protection status must also be checked when accessing a file, however, this simply involves calling the GetPS routine.

Link Creation and Deletion

Creating a hard link to an already existing file requires checking the protection status of all existing paths, accomplished by calling the PVerifyAll routine. Once this is completed, a new PSA-path pair is created using AddPSA. At this point the new directory entry can be created.

Link deletion requires firstly verifying all paths (PVerifyAll). Assuming this check succeeds, if the verified link count is greater than 1 then the PSA-path pair

\(^2\)Note that PSI values must never be reused to avoid collisions.
for the specified link is deleted (DeletePSA) and the process status updated. The corresponding directory entry can then be removed. If this is the only remaining link, the PST entry, inode metadata and file contents are deleted (DeletePSI). If the file is protected then the key entry in GPRIV is also deleted.

Moving/Renaming a File

Moving or renaming a file simply involves changing a path value relating to this inode. However, this requires a number of changes to protection status meta-
data. Once existing protection status data is verified (PSVerifyAll), a PSA-path pair corresponding to the new name is created (AddPSA) and the old value deleted (DeletePSA). The directory entry corresponding to the old name is then unlinked and the new one created.

Adding or Removing Cryptographic Protection from a File

When initiating the protection of a file, it is important to confirm that it is not already protected. This is achieved using PSVerifyAll and confirming the value of the relevant KID field in the PST. For unprotected files, this field will be 0. Once the file has been protected, the KID field in the PST is updated to reflect this and UpdatePS is used to ensure protection status metadata is consistent. Removing cryptographic protection similarly requires using PSVerifyAll first to confirm the file’s status. The relevant KID field in the PST is set to 0, the protection status metadata updated (UpdatePS), and protection can then be removed.

4.6 Ticket Creation Protocols

4.6.1 Read Protecting an Object

The procedure for read protecting an object involves the following steps.
1. User sends a request to protect a named file and specifies the protection mode as \texttt{read}. Initiating protection can only be performed using an L3 program.

2. The kernel checks the VID of the requesting process against the CRL and verifies that the owner of the file is the same as the user making the request. The instantiation of read protection is terminated if either of these checks fails.

3. The kernel checks the file’s current protection status using the file protection status verification mechanism described in Section 4.5. If the file is already read-protected, the attempt to re-protect it will fail, otherwise the kernel temporarily locks out any changes to the file’s protection status.

4. The kernel generates a key $K_{f^r}$ and encrypts the file with it.

5. A new primary ticket $T^1_{f^r}$ is constructed from this key as described in Equation 4.1 (p. 102) and is bound to the VID associated with the process making the request $VID_o$.

6. The file protection key is stored in GPRIV with the associated metadata; namely, the Key ID $KID_{f^r}$ (Equation 4.5, p. 105), timestamp $t_1$, protection mode (\texttt{read}) and the file’s protection status index $PSI_f$.

7. The kernel updates the read KID $KID_{f^r}$ and timestamp $t_{PSf}$ in the PST. After this is complete, the kernel can remove the lock previously held on changes to the file’s protection status.

8. A copy of the new primary ticket and its KID is placed in the Escrow Vault.

9. The kernel returns the newly generated primary ticket and its associated user vault metadata which the L3 process adds to the user’s vault.

At this point the user may now gain read access to the protected file.
4.6.2 Write Protecting an Object

The process involved in write protecting an object is similar to that for reading, with the principal difference being the existence of the MAC key used to cryptographically determine whether to allow or deny write access.

The procedure for write protecting an object involves the following major steps.

1. The owner sends a request to protect a named file and specifies the protection type, in this case write. Again this request is made using an L3 program.

2. The kernel checks the VID of the requesting process against the CRL and verifies that the owner of the file is the same user making the request.

3. The kernel checks to see that the file is not already write protected using the protocols from Section 4.5. If the file is already write protected, the attempt to re-protect it will fail. Otherwise, the kernel temporarily locks out any changes to the file’s protection status.

4. The kernel generates a MAC key $K_{fw}$ and a MAC value for the file is calculated $MAC(f)$. Note that the MAC is calculated based upon the plaintext contents of the file. If the file is currently read protected, this must first be decrypted in order to calculate the MAC.

5. A primary ticket $T_{fw}^0$ is constructed from this key, as described previously in Equation 4.1 (p. 102), and is bound to the VID associated with the process making the request, $VID_o$.

6. The key $K_{fw}$ and associated metadata, including the KID $KID_{fw}$ (Equation 4.5, p. 105), timestamp $t_1$, protection mode (write), the file’s PSI $PSI_f$, and MAC value $MAC(f)$, are stored in GPRIV.

7. The kernel updates the write KID $KID_{fw}$ and timestamp $t_{PSI}$ in the PST. After this is complete, the kernel can remove the lock previously held on changes to the file’s protection status.
8. A copy of the new primary ticket and $KID_{fw}$ are placed in the Escrow Vault.

9. The kernel returns the newly generated primary ticket and its associated user vault metadata which the L3 process adds to the user’s vault.

At this point the user may now gain write access to the protected file.

4.6.3 Creating and Distributing Secondary Tickets

The process for creating and distributing secondary tickets is straightforward.

1. The primary ticket holder for the object submits the following items to the kernel:

   - Primary ticket $K_f(K_f, H(VID_o, K_f, t_1))$.
   - Claimed KID, $KID_f$.
   - Path name for file in question $p_f$.
   - VID of user whom they are granting access $VID_r$. This can be securely obtained based on the user information found on their VPKI certificate stored in GPUB.
   - Expiry time/date $t_E$ or null if it should never expire.

This process is performed via an L3 vault administration program as only these programs have direct access to the user’s vault.

2. The kernel verifies that the ticket is a primary ticket and is valid for the file in question:

   (a) Kernel looks up the KIDs for the file in the PST using the GetPS routine as described in Appendix D. By identifying which KID matches with the KID claimed in Step 1, the kernel can determine whether the protection mode is intended to be read or write. If neither KID matches, the ticket issuing process aborts.
(b) The appropriate key entry in GPRIV is retrieved based on this KID. Note that the KID used to identify the key is that from the user’s vault and is therefore untrusted data. However, if this KID is incorrect then the wrong key will be obtained from GPRIV and verification of the primary ticket will fail.

(c) Verification of the primary ticket is performed as described in Section 4.7.4.

3. The kernel constructs the ticket key $K_T^r$ as $H(VID_r, K_f, t_1)$ using the parameters obtained in the previous steps.

4. Using the hash calculated in the previous step, the expiry timestamp specified in Step 1 and the current time, the kernel constructs the secondary ticket $T_f^2$ and encrypts it with the ticket key, giving $K_T^r(t_2, t_E, H(VID_r, K_f, t_1))$.

5. In order for the secondary ticket to be usable by the recipient, it must have a ticket ID (TID). As per Equation 4.4 (p. 104), the kernel constructs the TID $TID_f^r$ by hashing the KID of the file, the VID of the intended recipient, and the secondary ticket timestamp. The resulting value $H(KID_f, VID_r, t_2)$ is paired with the recipient’s VID, giving $<VID_r, TID_f^r>$ and this is added to the list of the VID-TID pairs associated with the key in GPRIV as shown in Figure 4.2.

6. The new secondary ticket is now distributed to the recipient:

   (a) The kernel locates the public key for the recipient user in GPUB using $VID_r$ and retrieves it.

   (b) The kernel encrypts the new secondary ticket with this public key.

   (c) The recipient’s VID is attached to the encrypted ticket as a plaintext header and the kernel lodges the result in GPUB for collection.

7. The recipient can then request collection of tickets from GPUB by searching for their VID, decrypting the ticket and storing it in their vault.
In practice, users will often want to distribute secondary tickets to groups of other users at a time rather than creating each ticket individually. However, the above steps will still need to be completed, albeit in an automated and transparent way, making this primarily a user interface issue. In practice, this could be implemented by the user creating lists of users via an L3 program that could either be stored in a separate write-protected file or directly in the vault. The construction of this list requires the user to verify the identity of the intended recipients, as specified on their certificate, with the list then being made up of these recipient’s VIDs. However, the revocation status of these certificates must still be verified upon the creation of each ticket.

4.7 File Access Protocols and Related Issues

4.7.1 Binding File Keys to Applications

Access control models involve limiting the objects that a user may access. However, actual access is performed by programs executing on behalf of the user. Therefore the security of a file also depends on the security of the programs that can access it. The Vaults Trusted Fingerprinting mechanism (Chapter 5) partially addresses this issue by preventing the applications that a user trusts from being illicitly modified without the user’s knowledge. However, Vaults also uses information from Trusted Fingerprinting to assign a trust level to applications when they execute and this determines the degree of access these processes will have to the user’s vault. Programs with a trust level set to L1 are restricted to a specified set of keys, regardless of the tickets contained in the user’s vault. Setting an L1 trust level can be used to confine an application to the minimum number of protected files that it needs to access to complete the work required of it by the user. Such an approach is particularly appropriate for applications that must deal with a large quantity of data from untrusted sources, and allows them to execute with minimal privileges. This confinement is accomplished by binding the protection keys of the specific
objects to the L1 application concerned, in a manner notionally similar to how application keys can be bound to specific applications. Although keys are bound to an application, and not vice versa, a key may be bound to many applications and a particular application may have many keys bound to it. Also, it is keys rather than tickets that are bound to an application. Therefore only the owner of a protected file who possesses the appropriate primary ticket can bind its key to a particular application. File protection keys may only be bound to natively executable programs and not to interpreted scripts etc. as otherwise determining which application is actually accessing the file becomes problematic. Finally, note that binding a file protection key to an application does not mean that only that application can access the file protected with that key; the set of programs that can access this file includes bound L1 applications as well as all L3 and L2 programs. The purpose of binding file protection keys therefore is to confine specific applications, rather than as a mechanism for further protecting specific files.

**The File Key Binding Process**

Binding a file protection key to an L1 trusted application involves using a specific L3 program to identify both the protected file (or files) and the application to which these are to be bound. A request is then made to the security kernel to instantiate this binding, submitting the primary ticket (or tickets) for the relevant files and the fingerprint for the application. After verifying the primary ticket (discussed in Section 4.7.4), the kernel then adds the fingerprint value for the application to the list of bound applications for that key stored in GPRIV. Note that file protection keys can only be bound to L1 applications that are part of the user’s LTCB, so any programs that only have fingerprint records in GPUB must first be duplicated inside the user’s vault before having keys bound to them.
<table>
<thead>
<tr>
<th>Protection</th>
<th>Access</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>r</td>
<td>Use key associated with ticket.</td>
</tr>
<tr>
<td>w</td>
<td>w</td>
<td>Verify MAC using key associated with ticket. Recalculate when complete.</td>
</tr>
<tr>
<td>r</td>
<td>w</td>
<td>No ticket required. Data is encrypted as written using $K_{fn}$.</td>
</tr>
<tr>
<td>w</td>
<td>r</td>
<td>No ticket required. MAC verified prior to granting access.</td>
</tr>
<tr>
<td>rw</td>
<td>r</td>
<td>Use respective keys to decrypt contents and verify MAC.</td>
</tr>
<tr>
<td>rw</td>
<td>w</td>
<td>Use respective keys to decrypt contents and verify MAC. Recalculate when complete.</td>
</tr>
</tbody>
</table>

Table 4.3: Protection mode and access request permutations and behaviour

### 4.7.2 Accessing Protected Files with Tickets

When attempting to open a file for either read or write, the protection status must first be determined using the GetPS routine (Appendix D). If the file is completely unprotected, the file is opened in the same way as in a non-Vaults environment with no ticket required. However, protected files can also be accessed without needing a ticket if the file’s protection mode does not intersect with the access mode being attempted. The applicability of this scenario can be determined based upon the KID fields in the PST as reported by the GetPS routine. Six scenarios exist, as summarised in Table 4.3, and are explained below. For those where a ticket is required, the correct ticket can be identified by matching the KID field for the designated access mode against the corresponding field in the ticket metadata in the user’s vault. Similarly, where the kernel must perform transparent encryption or decryption, the authenticated KID values from the PST allows selection of the correct key data in GPRIV.

When read access to a read-protected file is requested, the KID obtained from the PST is used to identify the relevant ticket in the user’s vault and the corresponding key in GPRIV is retrieved and the ticket verified as described in Section 4.7.4. If this succeeds, the kernel grants read access to the file. Writing to a file that is only write protected also requires the user to possess a suitable ticket. The $KID_{fw}$ is retrieved from the PST and the corresponding key in GPRIV is obtained for verification of that ticket. After the ticket has been verified, the MAC is verified against the file’s contents and, if successful, write access to the file is granted and the new MAC value is recalculated when the file is closed.
While write access to a file that is only read protected does not require a ticket, as the file’s contents are encrypted, the data written to the file must also be encrypted with the read protection key. The kernel therefore uses the $KID_{RB}$ from the PST to identify the correct encryption key. On the other hand, when reading from a file that is only write protected, the contents of the file must be read into memory and the MAC verified, again using the write protection key referenced by the PST entry. Note that read access must then be granted to the verified copy of the file held in memory in order to eliminate race condition attacks.

When accessing a file that is both read and write protected, the requirements of both protection modes must be considered. When reading from such a file, file contents must be both decrypted and the MAC verified prior to access. The same applies when writing to a file with both protection modes; however, when writing is complete, the MAC must then be recalculated.

4.7.3 Issues Relating to Trust Levels and Bound Files

The above assumes that the application attempting to access the file is operating at a trust level of L2 (or above). If the application is an L0 process, its access attempt will fail as such processes cannot access the user’s vault. However, if the application is an L1 process, its ability to access the file is dependent upon whether the key that protects the file is bound to this application.

When an L1 process requests access to a protected file and the user concerned possesses the corresponding primary ticket, the security kernel ascertains whether the Fingerprint value stored in the process’s metadata is present in the list of fingerprints of Bound Applications field in the GPRIV metadata for that key. If this value is found, the L1 process has permission to access the specified protected file in the relevant mode and access proceeds. However, if the fingerprint is not found in the list, the key has not been bound to this application and access is denied.

Although file protection keys can be bound to L1 applications in order to confine
them to the specified objects, this does not restrict which applications secondary
ticket holders may use to access these objects. As all L2 and L3 applications
can access all protected files and trust levels are determined on a per user basis,
the binding of file protection keys to specific applications has no impact on the
recipients of secondary tickets. As it is not a goal of Vaults to establish a digital
rights management (DRM) scheme, it is difficult to justify the additional complexity
involved in enforcing such restrictions on secondary ticket holders. Nonetheless,
there may be opportunities here for further research.

4.7.4 Primary Ticket Verification

Ticket verification is performed whenever a ticket is used; for example, when accessing
a protected file with either a primary or secondary ticket. However, for primary
tickets, verification is also required when creating or revoking secondary tickets and
when removing protection from an object.

Prior to verification, the KID for the ticket concerned is obtained from an authen-
ticated source, specifically an entry in the PST. Using the KID, the protection key
$K_f$ for the object is retrieved from GPRIV. The verification protocols described
below assume that a key length of $k$ bits, a hash output of $h$ bits and timestamp
length of $n$ bits.

Primary ticket verification involves the following steps and all of these must be
successful for the ticket to be considered valid and usable.

1. Verify that the VID associated with the requesting process is not on the
   CRL. If this process has no VID as it is executing at a trust level L0, access
   is automatically denied. Alternatively, if the trust level is L1, the kernel must
   additionally confirm that the process's fingerprint value is present on the list
   of bound applications associated with the key in GPRIV.

2. Decrypt the ticket $T^1_f$ received using the protection key $K_f$ obtained from
   GPRIV giving $[K_f, H(VID_o, K_f, t_i)]$. 
3. Verify that the first $k$ bits of plaintext from the previous step are equal to the value of $K_f$ used.

4. Using the VID of the requesting process and the key and protection timestamp values obtained from GPRIV, calculate $H(VID_r, K_f, t_1)$.

5. Verify that the next $h$ bits from the decrypted ticket are equal to the value calculated on the previous step.

4.7.5 Secondary Ticket Verification

Similarly for secondary ticket verification, the following steps apply and again all of these must be successful for the ticket to be accepted.

1. Verify that the VID associated with the requesting process is not on the CRL.
   As with primary tickets, if the process has no VID as it is executing at trust level L0, access is automatically denied. Alternatively, if the trust level is L1, the kernel must additionally confirm that the process fingerprint value is present on the list of bound applications associated with the key in GPRIV.

2. Construct the ticket key $K_{T_f}$ by hashing the VID of the requesting process, the protection key and protection timestamp $H(VID_r, K_f, t_1)$.

3. Decrypt the ticket $T_f^2$ with the ticket key calculated in the previous step giving $[t_2, t_E, H(VID_r, K_f, t_1)]$

4. Verify that the first $n$ bits ($t_2$) are prior to the current system time.

5. Verify that the next $n$ bits ($t_E$) are after the current system time.

6. Using the VID of the requesting process and the key and protection timestamp values obtained from GPRIV, calculate\textsuperscript{3} $H(VID_r, K_f, t_1)$.

\textsuperscript{3}Note that the hash construction here is the same as for the ticket key; however, the ticket key value will likely need to be truncated from the full length of the hash. This step therefore can be optimised by not discarding the full hash value calculated in Step 2 prior to truncation.
7. Calculate the TID by hashing the KID, VID of the process and secondary
ticket timestamp from Step 4, \( H(KID_f, VID_r, t_2) \).

8. Finally, the kernel searches the list of TIDs associated with the key in GPRIV
using the requesting process’s VID and confirms that the TID calculated in
the previous step is present. If the TID is not present, this indicates that the
ticket has been revoked and access is denied.

4.7.6 Fundamental Ticket Verification

Fundamental tickets are those associated with a key held in the Fundamental Vault
that encrypts one of the system vaults. Fundamental tickets are used by the prime
user when either modifying an entry in GPUB or obtaining a secondary ticket to
bypass cryptographic access controls (Section 3.6.3). Verifying a fundamental ticket
involves the following steps.

1. Confirm the trust level of the requesting process is L3. Only L3 programs are
   able to use fundamental tickets.

2. Decrypt supplied fundamental ticket \( T^F_V \) with the key \( K^F_V \) from the Funda-
   mental Vault for the system vault to which access is being requested, giving
   \( H(K^F_V, VID_r) \).

3. Hash \( K^F_V \) and the VID of the requesting process, and compare it with the
   result of Step 2.

4. Verify that the VID of the process making the request is not present in the
   CRL and has the \textit{Certifying} flag set on the corresponding certificate in
   GPUB.

If all of the above is completed satisfactorily, the ticket has been successfully verified
and access is granted.
4.8 Other File Protection Protocols and Issues

4.8.1 Revocation

Revocation is critical because users who have granted others access to their objects must be able to terminate this access if and when desired. Note that the design of Vaults, which does not grant users direct access to file protection keys, ensures that revocation is both efficient and final.

The possession of a primary ticket allows the owner of an object to both grant and revoke access to that object for the specified mode. As previously described in Section 4.4.3 and Figure 4.2, each key in GPRIV has associated with it a list of zero or more TIDs, each with a corresponding VID label. During the verification of a secondary ticket, the TID for that ticket and user is calculated and checked against this list. If the TID is not found, access is denied, effectively rendering the secondary ticket unusable. The primary ticket holder can therefore revoke use of a secondary ticket by requesting the removal of a corresponding TID from the list in GPRIV. The protocol for this is:

1. The primary ticket holder uses an L3 program and submits a request to revoke access, specifying:
   - Primary ticket $T^i_j$
   - File path $p_f$
   - User whose access is to be revoked $VID_r$
   - Access mode (read/write).

2. From the file path $p_f$, the kernel looks up $KID_f$ for the object in the PST and locates the relevant key $K_f$ in GPRIV.

3. The kernel then verifies the primary ticket as described in Section 4.7.4.

4. Using the VID of the revoked user, the kernel locates the TID in the list and deletes the $<VID_r, TID^r_f>$ pair.
Note that in the case of a user’s certificate being revoked, it is not necessary for those other users who have granted them secondary tickets to manually revoke them in order to maintain security. Not only is the revocation status of the user’s certificate checked when verifying both primary and secondary tickets, but revoking a user’s certificate involves the automatic revocation of all secondary tickets belonging to that user by deleting their VID from all protection key entries in GPRIV as described in Section 3.3.8.

4.8.2 Removing Cryptographic Protection from a File

The process of removing cryptographic protection from files differs depending upon the mode being removed and the file’s current protection status. In all cases an L3 application is used and the subject presents the relevant primary ticket and specifies the file path and protection mode. Using the path, the kernel obtains the relevant KID from the PST and verifies the primary ticket as described in Section 4.7.4. If the file is only read protected and this protection is being removed, the kernel will decrypt the file and write the plaintext to disk. Alternatively, if removing write protection, this requires verifying the MAC and warning the user if the verification fails. Note that if the file having write protection removed from it is both read and write protected, the file must first be decrypted in order to perform verification. However, this plaintext is not written to disk. For files that have both protection modes currently enabled but read protection is being removed, the data must be decrypted, verified against the MAC and then the plaintext written to disk. Finally, in all four of the scenarios described, the key record and associated data in GPRIV must be deleted and the relevant KID in the PST be reset to null.

4.8.3 Ticket Escrow

When a new primary ticket is generated, a copy is placed in the Escrow Vault and labelled with its KID. This allows the prime user to exercise one of their fundamental privileges, namely to bypass the cryptographic enhanced access controls provided by
Figure 4.4: Overview of Escrow vault operation

Vaults. Bypassing these controls in this way requires possession of the fundamental ticket for the Escrow Vault $T_E^F$. Figure 4.4 provides an overview of the operation of the ticket escrow mechanism.

However, consistent with the goal of limiting the prime user’s privileges, access to the Escrow Vault is controlled in a number of ways. Most importantly, the prime user does not gain access to file protection keys themselves, which remain secure in GPRIV and are not stored by the Escrow Vault. Also, as the primary tickets held in the Escrow Vault are bound to the VID of the file’s owner, they are not usable by the prime user. Instead, upon being provided with the correct fundamental ticket, the security kernel retrieves the relevant primary ticket and generates from this a secondary ticket bound to the prime user’s VID using the procedure described in Section 4.6.3. The expiry time set on these recovered secondary tickets is set to be relatively limited with selection of a default value being dependent upon the organisation’s security requirements. In this way, access to cryptographically protected objects is only for a limited amount of time, although the prime user can request additional secondary tickets at a later point if required.

The process of requesting a secondary ticket for a protected object requires the use of a specifically designed L3 program, which must be both interactive and require explicit selection of the files for which tickets are to be obtained. These requirements
ensure that the process cannot be subverted in an automated way by malicious code. The following steps apply to this process.

1. The prime user executes the appropriate L3 program and selects the path of the file $p_f$ for which they wish to retrieve a secondary ticket and the access mode required.

2. The L3 program submits these details to the kernel, along with the fundamental ticket $T_E^F$ for the Escrow Vault.

3. The kernel verifies the fundamental ticket as described in Section 4.7.6.

4. The kernel retrieves the KID for the mode and file requested from the PST.

5. The kernel then searches the Escrow Vault for the KID of the selected file and retrieves the corresponding primary ticket.

6. The kernel constructs a secondary ticket from this primary ticket, bound to the prime user’s VID $VID_p$, and with an expiry timestamp $t_E$ set as described above.

7. The new ticket is returned to the L3 program, which places it in the prime user’s vault for subsequent use.

### 4.9 Summary

This chapter has described the mechanism within Vaults for providing cryptographic enhanced access controls. Users can cryptographically protect a file in either read or write mode and this provides them with a primary ticket for accessing the file, issuing secondary tickets to other users and subsequently revoking this access if and when desired. All tickets are bound to a specific vault and are unusable if accessed via any other vault. This restriction prevents recipients of secondary tickets from duplicating them and transferring them to other, unauthorised users. Primary ticket holders can also bind the file protection key to specific L1 applications, which
serves as a mechanism for limiting the trust that must be placed in a specific program. Finally, Vaults provides a way for cryptographically verifying filesystem metadata and this, combined with the cryptographic nature of the access control model, ensures that the specified security goals are achieved even in the face of an attacker who is able to bypass the security kernel and write directly to the secondary storage device. The next chapter will examine the Trusted Fingerprinting mechanism, which protects specific programs and various objects that they depend upon for secure operation against illicit modification. This mechanism is also used to assign trust levels to programs and these are used to manage the program’s privileges on the system.
Chapter 5

Trusted Fingerprinting

5.1 Introduction

This chapter describes in detail Trusted Fingerprinting, an important security feature of the Vaults model that is facilitated by its cryptographic infrastructure. Trusted Fingerprinting enables users, both privileged and otherwise, to select programs that are important for their security requirements and have these cryptographically checked for unauthorised modifications prior to execution. If the program has been modified, execution will be denied and users thereby shielded from potentially malicious code. As a means for users to obtain authenticated access to trusted code, Trusted Fingerprinting is therefore a virtual implementation of the traditional notion of a physical trusted path to the trusted computing base (TCB). However the mechanism also allows for programs to be assigned different trust levels and this determines their privileges in relation to the cryptographic access control model described in the previous chapter.

5.1.1 Goals of Trusted Fingerprinting

The goals of Trusted Fingerprinting are as follows:

活跃地保护受信任软件的修改

The primary goal
is to detect modifications to trusted code executed via a specific path. These may be programs that are trusted by an individual user or relevant to the security of all users on the system. However, Trusted Fingerprinting recognizes that, in many cases, code behaviour can be manipulated by modification of external dependencies, such as libraries or configuration files. Consequently, these objects are also checked for unauthorised modifications. Finally, if modifications are detected, execution is denied. Therefore, Trusted Fingerprinting is not simply a passive detection mechanism but actively prevents further compromise due to execution of malicious code.

**Authentication of code for privilege determination** By verifying the integrity of code, Trusted Fingerprinting allows for code that is authorised to access cryptographically protected files to be distinguished from that which is not. It also facilitates the confinement of a program to a specific set of protected objects. In this way, Trusted Fingerprinting is tightly integrated with the access control model described in the previous chapter and plays an important role in determining a process's privileges.

**Authentication of code for application key release** This code authentication mechanism also serves as a means for ensuring that application keys (Section 3.5) are only released to the applications specifically authorised to access them. As discussed in Section 2.6, this resolves a significant limitation of previous secure storage mechanisms.

**Preventing mis-use of trusted code** In addition to limiting the privilege of untrusted code by keeping track of the trust relationships between parent and child processes, Trusted Fingerprinting also includes measures to prevent trusted programs from being used in undesirable ways by malicious code beyond simply direct modification.
Figure 5.1: Overview of process execution and fingerprinting

5.1.2 Overview of Process Execution and Fingerprinting

The checking of a program’s fingerprint occurs prior to a new process executing and an overview is given in Figure 5.1. The first phase involves searching to identify whether a fingerprint exists for the program being executed. This process is known as ‘matching’ and involves considering fingerprints in both GPUB and the user’s vault, if available.

If the matching process fails to identify a corresponding fingerprint, execution of the program proceeds. However, the process will have a trust level of L0 and is unable to utilise items held in the user’s vault. If matching succeeds, the program’s
fingerprint, and those of any shared libraries upon which it depends, are verified. If verification is successful, the program is executed with its trust level as specified by the fingerprint dependency record. However, if verification fails, execution is denied.

5.2 Trusted Fingerprint Dependencies

In order to specify that a particular program should be verified upon execution, a new fingerprint record must be created. This record specifies both the actual fingerprint (hash) value, along with some other parameters, and also identifies the objects upon which the program depends for secure execution. Similar records for these objects are also subsequently created. Due to their interconnected nature, all fingerprinted objects are referred to as ‘dependencies’.

New dependencies can be created by both the prime user and other users on the system. In the former case, these dependency records may be placed in GPUB by the prime user\(^1\) and therefore constitute part of the system’s global TCB (GTCB). However, for non-privileged users, these records are placed in their own vault and only verified when that specific user executes the program. While the prime user may create fingerprints for any program on the system, users can only create fingerprints for programs they are able to read and execute. Dependencies may be created for any object to which the user will have access when the main fingerprinted program itself is run. For example, set user ID programs will generally run with privileges different to those of the user who executed them [5, pp. 145–151].

5.2.1 Dependency Data Structures

Each dependency created stores several parameters associated with that object and its fingerprint. These are described below:

\(^1\)The prime user also has their own user vault that belongs just to them in addition to GPUB.
Value  The fingerprint value representing the output of the one-way hash function for the contents of the specified target.

Targets  The list of one or more fully qualified paths for the directory entries corresponding to this dependency.

PSI  The protection status index of the file.

Dependencies  The list of zero or more fingerprint values representing dependencies for the object. This information effectively links the dependency record with the records for the objects on which it, in turn, depends.

Type  The type of dependency specified as Program, Library or Data File.

Trust Level  The trust level assigned to this dependency that determines its access to the user's vault and, consequently, the program's privileges. This field must be set for Program dependencies and is optional for those of type Data File. However, it cannot be set for Library dependencies.

Flags  A list of zero or more flags that can be used to fine-tune the operation of the fingerprint verification process.

As described above, each fingerprint record has a list of dependencies that result in a graph data structure [207]. Specifically this is a directed graph (digraph) but with vertices (nodes) that are irreflexive (do not link with themselves). Each dependency relationship is only one-way since one fingerprint being a dependency for another does not automatically imply any reciprocal relationship. Indeed, two dependency records will not normally be mutually dependent, although this is not impossible. Thus the resulting digraph will normally be acyclic in nature.

5.2.2  Dependency Types and Flags

The type of each dependency must be specified as a Program, Library or Data File and this value determines how the entry is processed. Program dependencies
<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>Program terminated if specified dependency is inaccessible.</td>
<td>1:49, 150, 155</td>
</tr>
<tr>
<td>Require Trusted Child</td>
<td>Child processes require parent's trust level to be executed.</td>
<td>150</td>
</tr>
<tr>
<td>Ignore Data Fail</td>
<td>Program continues if verification of Data File fails.</td>
<td>155</td>
</tr>
<tr>
<td>No Cache</td>
<td>Always reload and re-verify dependency.</td>
<td>157</td>
</tr>
<tr>
<td>Require Trusted Data</td>
<td>ETL set to 1.0 if non-dependency object opened for reading.</td>
<td>157</td>
</tr>
<tr>
<td>No Auto Update</td>
<td>Do not auto update fingerprint if dependency modified.</td>
<td>162</td>
</tr>
<tr>
<td>Installer</td>
<td>Allow updates to fingerprints of non-Data File children.</td>
<td>163</td>
</tr>
<tr>
<td>Require STL</td>
<td>Active node may only execute at its STL.</td>
<td>168</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of dependency flags

are those relating to natively executable code. For example, on a Unix system, this would typically apply to processes executed via one of the `exec()` system calls [205, pp. 207–212]. Programs in the dependency digraph will often be 'sources' in that there will be no edges that reach them from other nodes. However, this property is not required, as some programs exist primarily as 'helper applications' that can be launched by another program to complete some specialised task. For shared libraries that are linked with an executable program, the LIBRARY type is set. Finally, DATA File dependencies are those that are not natively executable and are read by running processes via a POSIX `open()` system call [205, pp. 48–50] or equivalent. Examples of such objects include non-executable objects such as configuration files that have the potential to significantly determine a program's behaviour at runtime and non-natively executable code such as scripts. Full details on the effect that the type of dependency has on the verification process are described in Section 5.4.

The way a dependency is processed can be fine tuned by the setting of specific configuration flags in the Flags field of the dependency record. These are discussed in more detail at the relevant points in the following sections. However, an overview is given in Table 5.1 along with a reference to where these details can be found. Default values for these flags are given in Table 5.2.

5.2.3 Dependency Trust Levels

The trust level of a dependency determines how that program may access the user’s vault and therefore its ability to access protected objects. The trust levels are
Table 5.2: Dependency flag details and defaults

labelled L0 to L3 as described in Section 3.4.2. Of the different types of dependencies, trust levels must be set for PROGRAM dependencies as this information is maintained in each process’s credentials. However, as LIBRARY dependencies are typically shared between a large number of programs, trust levels cannot be set for this type of dependency and the field will remain blank. Instead, when a library that is a dependency of a program is linked at runtime, the process is assigned the trust level of that program. However, a trusted program executing at L1 or above cannot be linked with a library that is not specified as one of its dependencies. If such linking is attempted, execution is aborted.

A trust level may be set for a DATA_FILE dependency, depending upon the type of file involved. If the file represents non-executable data, such as a configuration file, the field should be left blank as non-executable data is not considered to have a level of trust\(^2\). However, if the file represents a non-natively executable program such as a script, the user should set its specified trust level (STL) in the same manner as for a native executable. The only restriction that applies here is that the trust level of the non-native executable should not exceed that of the program that is its parent dependency (i.e., its interpreter or virtual machine). Otherwise, upon opening a DATA_FILE dependency, the effective trust level (ETL) of the process is set to the lower of its current ETL and the STL given in the DATA_FILE dependency record.

\(^2\)While any data input can affect the behaviour of a program, attempting to label the trust level of all possible inputs is infeasible for any non-trivial program and the lack of precision involved in such an assignment would make it of limited security value at the cost of significant complexity. While Trusted Fingerprinting therefore allows for non-executable data files to be specified as dependencies for programs and verified upon access, the model does not intend for trust levels to be set for them.
5.2.4 Identifying the Program Target and Library Dependencies

The process of creating a fingerprint begins by the user specifying the path of the program they wish to fingerprint using a designated L3 application. This path is known as a ‘target’. The kernel then identifies any other paths by which the dependency may be accessed. This information can be conveniently obtained from the file’s authenticated metadata, as described in Section 4.5. These paths then become additional targets for that particular record and will also cause the program to be verified if accessed from that path. If required, users may also manually add additional targets that cannot be detected, such as symbolic links.

The dependencies of the specified program are then identified. These dependencies may include dynamically linked shared libraries, interpreters, helper applications, configuration files and any other relevant data files that may affect the program’s execution in a security-relevant way. Many of these dependencies potentially can be detected automatically by the system by analysing the selected program’s code. For example, all relevant shared library dependencies can be identified through a recursive consideration of the program and its libraries, and this task can be easily automated. However, the user should review the generated list to remove any that are unnecessary and add any that have been missed; for example, any helper applications. Ensuring the completeness of this list is particularly important in the case of GTCB dependencies, which must be comprehensively identified by the security administrator to ensure sufficient verification occurs. However, some analysis may also be necessary when making the decision to exclude certain potential dependencies; for example, if these frequently change independently of trusted programs. Again, in the case of the GTCB, the security administrator is well-placed to perform such an analysis.

5.2.5 Data File Dependencies

Many programs do not exist as native code and are instead converted into an executable format immediately or shortly before execution. Examples include Java
programs and those written in various scripting languages such as Perl and Python [208–210]. In these cases, the non-native code file will be specified as a data file dependency of the natively executable interpreter. Typically it is likely that the interpreter can be identified automatically by examining the first line or header of the source code and so the burden on the user to perform this manually is minimal (cf. [211, p. 11]).

A program’s configuration files often have a significant effect on its behaviour and therefore could be manipulated for malicious purposes; for example, by downgrading security settings. It is therefore important to identify data files that may contain important configuration parameters for the given program and include these files as dependencies. However, as the impact of the configuration file on a program’s behaviour is highly specific to the programs and files in question, they require more consideration and analysis by the user than other data file dependencies. The user should identify the configuration files involved and possibly evaluate the potential impact they may have on security, relative to how frequently the files will be changed by programs for which they are not dependencies. For programs with a relatively static configuration that, once established, rarely needs to be changed, these should be included even if their impact on security appears limited. Similarly, if a program’s configuration is only ever changed from within that program or another for which it is also a dependency, these updates can be automatically captured by the Trusted Fingerprinting mechanism, meaning there is no disadvantage to including these as a dependency. Therefore, in the majority of cases, the burden on the user is limited to the identification of the appropriate files. In practice, this burden could be significantly mitigated, or eliminated entirely, by the use of vendor-supplied profiles describing the dependencies for a specific trusted application.

5.2.6 Creating Dependency Records

Once the targets for the new dependencies are identified and collated, the user sets trust levels and dependency flags for each new dependency as desired. The kernel
then calculates the fingerprint for each dependency and instantiates new dependency records for each, creating a digraph structure by specifying the Dependency field for each record as previously described. This new record is then integrated into the larger digraph stored in the relevant vault (either the user’s or GPUB), with each new dependency record representing a new node in the structure. If there are no shared dependencies between the newly created digraph and the existing one, separate digraphs may be stored. However, this scenario is likely to be rare due to extensive overlap of dependencies such as shared libraries.

Modelling the dependency relationships between programs in this way has the advantage that, where two or more programs share a dependency, an update to this dependency is automatically inherited by all programs affected. However, the nature of runtime dependency verification means that newly added or modified dependencies require a program to first be restarted before these apply. This and related issues are discussed further in Section 5.4.

5.3 Fingerprint Matching

Matching is the process by which the security kernel attempts to identify whether a dependency record exists for an object that is about to be executed or accessed. If the matching process identifies a target, the object’s fingerprint is verified against the value stored in the fingerprint record and other parameters, such as trust levels and dependency flags, will be applied. Dependencies of the target are also identified through the digraph and these are also verified as required. However, if a matching target cannot be found for a program that is about to be executed, the program will instead execute as untrusted (L0). Such programs have no access to the user’s vault or cryptographically protected files. Restricting the program’s privileges in this way significantly mitigates the potential negative impact of a substitution or redirection attack involving malicious code.
5.3.1 Matching Techniques

Restricting a program to the lowest privilege level possible when matching fails to identify an appropriate target limits the scope for negative security outcomes. However, one remaining issue is that of the user’s expectations. If the user executes a program that they believe to be fingerprinted and yet are redirected to malicious code, they may still expect that the program has been verified and can be trusted. This expectation could influence their interactions with the program. For example, if the user executes their usual, fingerprinted, web browser to perform Internet banking, but is instead redirected to a compromised version that is not verified, they may unwittingly reveal sensitive information to an attacker\(^3\). Therefore the matching process must be performed in order to both maximise the likelihood of identifying the appropriate target and to be consistent with the user’s understanding and expectations of the system. This relates to the secure design principle of psychological acceptability as identified by Saltzer & Schroeder [15].

This second requirement excludes several otherwise ideal matching techniques. For example, matching by inode number is vulnerable to a trivial attack where legitimate, fingerprinted code is renamed or deleted and malicious code substituted with the original’s name. As this is the name by which the user executes the program, the malicious code will be executed and the substitution not detected. This is the same vulnerability that applies to the I\(^3\)FS scheme discussed in Section 2.7.4. Although under Trusted Fingerprinting the malicious program will execute with limited privilege, from the user’s perspective they have executed the same program they always do and will expect it to have its usual security properties. Matching based on a file’s protection status identifier (PSI) suffers from essentially the same problem. Although the PSI value can be relied upon to be unique and authoritative, it will not detect a substitution attack where the malicious code has the same name as the intended program, and nor is this its intended purpose.

\(^3\)Normally banking passwords and the like would be stored as application keys and bound to the legitimate, fingerprinted version of the web browser so the inability of the malicious version to access these keys transparently could alert the user to the attack.
5.3.2 Path Matching

Consequently, matching based upon the target’s path is most consistent with the user’s expectations. Therefore all paths by which a program may be executed are specified as targets when creating a new dependency. As noted, these can be securely identified by the kernel using the cryptographically verifiable list stored in the filesystem metadata. This list does not include symbolic links; however, these will be resolved to the actual path by the kernel prior to execution. Although new links to fingerprinted files may be created after its dependency record has been instantiated, this has limited impact on security for a number of reasons. First, the user is unlikely to execute a program via a new pathname of which they were unaware and, if aware, they can update the dependency record. Second, the security kernel can also periodically search and update the target fields of dependency records as required by crosschecking them against secure filesystem metadata. Finally, any fingerprinted program inadvertently executed through a link that is not specified as a target will default to an L0 trust level and thus mitigate the impact of any attacks based upon it.

A detailed algorithm for path matching is given in Appendix E. In summary, the algorithm works by iterating through the list of dependency records, searching for a target path amongst those matching that of the program being executed. If such a path is found, matching has been successful and the search is halted. The routine then returns the matched dependency record for verification. However, if all dependencies are searched without a match being found, the routine returns 0 to indicate this and the program will execute as untrusted.

5.3.3 Multi-digraph Matching

The matching algorithm from Appendix E applies to a single dependency digraph. However, matching may often need to be performed against a number of different digraphs. When one program executes another, initially the matching algorithm is applied to a digraph specifying the dependencies for the program that is currently
executing. This digraph is known as a ‘runtime digraph’ and is discussed in the next section. If a match is not found, matching is performed against the digraph structures in both the user’s vault and then GPUB. This process has three possible outcomes. If a match is found either locally or globally, but not in both, the verification process proceeds for the single matching target. If no match is found in either vault, matching fails and the program executes as untrusted as previously described. However, if matching targets are found in both the user and GPUB vaults, a ‘double match’ has occurred and this affects the construction of the runtime digraph as described in Section 5.4.

Matching may also need to be performed against multiple digraphs within a single vault. While generally there will be significant inter-relationships between dependencies due to shared libraries, any fingerprinted programs that are statically linked are likely to exist in separate digraphs, which must also be searched during matching. However, this requirement does not complicate the matching process with the algorithm described applied to all local and global digraphs (in that order). Also, unlike local and global matching, there is no possibility of a double match when scanning multiple digraphs within a single vault if the digraphs themselves are consistent, as otherwise this would suggest that they should be merged.

5.4 The Runtime Dependency Verification Model

5.4.1 Runtime Digraphs

As just described, applying the matching process to both local and global digraphs can result in a double match where the target is found in both vaults. A program being fingerprinted both locally and globally indicates a significant degree of security importance. As a result, dependency information from both sources is used for verification. In particular, any differences in the dependencies specified locally and globally must be identified. For example, an application may have a global file specifying settings applying to all users but also allow individual users to customise
their settings in a separate per-user configuration file. As both of these may affect
the program at runtime, both are considered to be dependencies and may require
verification.

To facilitate comprehensive verification, a runtime digraph (RTD) for the program
is constructed. The RTD specifies the dependencies for the matched target to ensure
that they are all verified at runtime and remains in effect for the execution lifetime
of the program\(^4\). It also facilitates tracking of the cache and verification status of
each dependency as detailed later in this section. For this purpose, each dependency
record in the RTD contains two additional fields not listed in Section 5.2.1. The
first is the verification status of the dependency and the second is a reference to its
cached image in secure memory. The dependency can only be used if it has been
loaded and its image previously verified against the specified fingerprint. Where
there has only been a single match found, either locally or globally, then the RTD
is a subset of the larger digraph where the match was found and consists only of the
dependencies relevant to the matched target. The nodes of the RTD are identified
by traversing the larger digraph beginning from the matched target.

5.4.2 Merged Runtime Digraphs

In the case of a double match, local and global digraphs must be merged to form
an RTD for the program. This is an additive process, while also eliminating du-
lication. It involves simultaneously traversing both local and global digraphs,
beginning with the matched target. After crosschecking to confirm that fingerprint
values are the same in both, this dependency becomes the source node in a new
RTD. Traversal then proceeds to the child dependencies for each node in the local
and global digraphs. Fingerprints are cross-checked and, where dependencies from
different digraphs have intersecting target names, they are considered to be the
same dependency and a single instance is added to the RTD. This process ensures
that all relevant dependencies that are in both the local and global digraphs are

\(^4\)This is the reason why dependencies added for a given program will not apply to the program
until it is restarted.
identified and included in the RTD for verification.

An abstracted but nonetheless representative example is given in Figures 5.2 and 5.3. The first diagram shows the two separate global and local diagrams prior to merging and the second shows the merged result. Such an RTD is created for each successfully matched application prior to verification and execution. It is then used to direct the verification of dependencies at runtime.

5.4.3 Runtime Digraph Construction Conflicts

If, during the RTD construction process, a dependency is found that has different local and global fingerprints for the same target, then this strongly indicates that a security failure has occurred and the verification process is considered to have failed. While this event may be the result of data in either the local or global vault being out of date, it may also indicate that the contents of one of those vaults has been subverted. Aborting the verification process at this stage is therefore critical in order to maintain the defence-in-depth afforded by the two-tier trust system.

Another potential conflict is that the flags set for related dependency records in the two vaults may differ. Table 5.2 specifies the default values for each flag and also the result if there is a conflict when merging to create the RTD. Note that the merged conflict result is always the safest possible outcome.

The two other possible differences between local and global dependency records are actually not conflicts at all. First, trust levels for dependencies may differ. However, this reflects the nature of trust level specifications as an indication of a user’s security policy with respect to that application. Therefore the globally-assigned trust levels are only defaults and user-specified values always override this setting. Second, the list of targets in related dependency records will not necessarily be identical. Again, this does not indicate a conflict and is consistent with the additive nature of the RTD construction process which identifies intersections between target names in both local and global digraphs.
Figure 5.2: Example of separate global and local digraph before merging

Figure 5.3: Merged runtime digraph
As dependency data in both GPUB and the user's vault is regarded as trusted, constructing the RTD aims to include as large a list of potential dependencies as possible. Therefore, the list of targets included in a dependency record in the final RTD represents the union of those targets specified for that matching dependency in both GPUB and the user's vault.

5.4.4 Practical Fingerprint Verification Issues

Certain practical issues influence the way fingerprint verification is performed at runtime. One significant issue is preventing race condition-based attacks surrounding the verification process. For example, it is critical that the program image that is verified be the same as that which is actually executed so an attacker cannot substitute unverified code between verification and execution time. Similar issues apply to non-PROGRAM dependencies; for example, the shared library image actually linked with the application on execution must consist of the same data that was verified against its fingerprint in the dependency record. This issues is even more significant for DATA FILE dependencies that may be accessed some time after the program has begun executing, making the window for potential attacks much wider. Therefore, secure verification requires that the dependency image hashed for verification against the stored fingerprint be the same as that which is actually used. To this end, all verified objects held in memory, regardless of their type, must be protected against modification by the security kernel. This requirement includes ensuring that these objects cannot be modified if written out to swap, either through the use of a virtual memory encryption scheme such as that described by Provos [136], or through a mechanism equivalent to the mlock() POSIX system call to disable paging for the memory concerned [212].

A similar issue relates to the availability of dependencies at runtime. For example, a dependency may not be available when execution begins if it is located on an external storage device or network filesystem. As long as the dependency is available for verification and use at the required time, this does not impact on fingerprint
verification. In particular, it is an important requirement that an attacker cannot avoid verification of any dependency by making it temporarily inaccessible at a certain time and, again, this highlights the need for verification to be performed on the same image that is subsequently used.

Alternatively, sometimes a specified dependency may not be available for verification and use. For many programs this unavailability will not impact on security and will simply be handled internally. However, in some cases, it may be important for a program that a specific dependency be available, as its unavailability may impact negatively on security. For example, a program may default to an insecure state if its configuration file is not available. In this case the *REQUIRED* flag should be set. The flag may be set on any type of dependency and if, for any reason, the object flagged in the program’s RTD is inaccessible, the kernel will deny or terminate the program’s execution. As indicated in Table 5.2, this flag is switched on by default.

5.4.5 Runtime Verification of Programs

Execution of a native program by the creation of a child process through the POSIX *fork()* system call [205, pp. 188–193], followed by a call to one of the *exec()* family of system calls [205, pp. 207–212], involves one of two possible relationships between the parent and new child process. In one case, the parent is a user interface shell process responsible for executing other programs and there is no dependency relationship between parent and child. Alternatively, the new child process may be a program that performs a specific task on behalf of the parent. In this case, the newly executed program is the dependency of the parent and is considered to be a ‘helper application’. Both of these outcomes, and their impact on matching and verification, are outlined in Figure 5.4 and described in detail below.

Upon execution of any new program through an *exec()* call, an attempt is made to find a matching *PROGRAM* target in the parent’s RTD. If it is found, this indicates the new program is a helper application and there is no need to attempt matching on local and global digraphs. If no match is found in the RTD, the new program has
no dependency relationship with its parent but may still exist in one of the other larger digraphs, most likely as a graph source. Matching is therefore performed on local and global digraphs and, if this succeeds, a new RTD is constructed for the new program based on the results. This new RTD is retained in memory throughout the lifetime of the program's execution and used to direct the runtime verification process. Alternatively, if no match is found, the program will execute as untrusted. If either match is found, verification proceeds. The program's image is loaded off the disk, hashed and the result is compared against the fingerprint stored in the RTD. If verification fails, execution is aborted. Otherwise, the program's record in the RTD is flagged as having been verified. After successful verification of and linking with dependent shared libraries, the new process's credentials are initialised, as described in Section 3.4.4, and the verified image referenced by the RTD is executed. By caching the program's image in memory, the race condition attacks described in the previous section are excluded.

5.4.6 Impact of Dependency Flags on Execution

With the general exception of user interface shell programs used to launch other applications, when one program executes another there is the potential for the child process to impact on the parent’s security. This issue can be addressed by setting the Require Trusted Child flag in a program's dependency record, meaning that any processes it spawns cannot execute at a lower trust level\(^5\). Execution of any child processes, whether matched as a dependency in the RTD or not, will be denied if its trust level is lower and the flag is set. Similarly, if the trust level of the child process is lowered at a later point after its execution is initiated, the flag also comes into effect. The flag is set by default but must be switched off when creating dependencies for user interface shell programs. However, the flag should not be switched off for L3 programs.

\(^5\)In practice this means parent and child must have the same trust level as it is not generally permitted for a child process to execute at a higher trust level than its parent (Section 5.6).
Figure 5.4: Matching and verification for POSIX `exec()` system calls
set for a program's dependency as specified in the RTD, and this dependency is
not accessible when accessed, then the parent program will be terminated. The flag
applies to Data File dependencies but can also affect helper applications that are
specified as dependencies of another program; if the application cannot execute for
some reason, the execution of the parent program is terminated. Also note that
the Require Trusted Child and Required flags can potentially interact. If
the former is set on a parent program and the latter set on a helper dependency,
and if the helper application would not execute at the required trust level and is
therefore prevented from launching, this will lead to the parent being terminated
also. However, this scenario strongly indicates a misconfiguration, suggesting the
trust level of the helper application should be raised or at least one of the flags
altered. This condition is easily checked for and detected during configuration so
that the user may select the appropriate solution. However, even if this checking is
not done, the fail safe defaults are designed to limit any possible security failure.

5.4.7 Runtime Verification of Libraries

A related process occurs when linking a program with its shared libraries and is
summarised in Figure 5.5. The RTD is traversed and all dependencies with the
type Library are loaded and, if verified successfully, linked with the program.
However, libraries are typically shared between many programs. Therefore, if a
particular library has already been loaded and verified for linking with another
trusted program for which it is also a dependency, this instance may be reused
for the new program so re-verification is not required. This approach creates links
between RTDs of different executing programs, allowing the verification status of
library dependencies to be tracked. However, it also requires that the kernel confirm
that the apparently mutually shared libraries are in fact the same by confirming
both the intersection of targets and identical Value fields.

If a verified library is later unloaded from memory (i.e., no longer part of any RTD)
and subsequently reloaded for linking with another program, the library must be
Figure 5.5: Linking and verification process for shared libraries
reverified before execution can proceed. Any loaded and unverified libraries linked
with untrusted (L0) programs must be reloaded and verified before linking with any
trusted program executing at L1 or above. Finally, trusted programs can only be
linked to shared libraries specified in their RTD and attempts to link with other,
possibly unverified, libraries causes execution to be aborted. In practice, though,
this scenario would be unlikely to occur due to the automated detection of library
dependencies for a given program.

5.4.8 Runtime Verification of Data Files

Dependencies of type Data File are those accessed with a read or write request via
the open() system call. However, the logic applied differs depending on the mode
of access requested. An overview of how read requests are processed is given in
Figure 5.6; write requests are considered in Section 159. Note that, for the sake of
simplicity, the overview presented in the diagram deliberately does not describe the
impact of the Require Trusted Child and Require STL flags. Upon receiving
a read request, the security kernel searches the RTD for a target matching that
specified. Note that matching is performed recursively and all nodes in the RTD
are searched. If no match is found, the file being accessed is not a dependency of the
program. Access then proceeds with no verification required, although the ETL of
the process may be lowered to L0 if the Require Trusted Data flag is set on the
program’s fingerprint record. Alternatively, if a matching dependency is located,
the file’s contents must be verified prior to granting access to the process. If a cached
and verified copy is available, this may be used. Further details concerning caching
are discussed in the next section. Otherwise, the system attempts to load the file’s
contents into memory and the image is verified against the target’s fingerprint. If
loading and verification are successful, data read from the file is taken directly from
the verified image cached in memory, thereby preventing race condition attacks.
However, if the Data File dependency has a trust level specified and it is lower
than the ETL of the program, the program’s ETL will be lowered to match it prior
to being allowed to read the file's contents. The reasons for this behaviour are discussed in Section 5.4.10.

Verification of a Data File dependency may fail in two primary ways. The first is when the file is not available for verification. In this case, the reading of the file clearly cannot proceed. However, if the required flag is set, this will also cause the requesting process to be terminated since the unavailability of a dependency may have a detrimental security-related impact on the program's behaviour. Alternatively, verification may fail due to a modification being detected. As a potentially malicious modification to the dependency has occurred, the program will not be granted read access to the file. Since the program's response to the unavailability of this dependency is unknown, by default the program's execution will be terminated. However, if the Ignore Data Fail flag is set for the dependency being accessed, the program may be allowed to continue to execute, although under no circumstances is read access to the modified dependency granted. Note that, if set, the Required flag takes precedence over this other flag and the program will still be terminated due to the unavailability of the dependency.

5.4.9 Impact of Flags on the Runtime Data File Verification

Caching of Data File images

When a file's contents are first read into memory and verified, they are held there while the file is opened and data is read directly from the verified image, rather than the disk. If the file is then closed, a cached copy of the verified file may be maintained in memory, space permitting, for the duration of the program's execution, allowing subsequent accesses without re-verification. This approach is feasible because configuration information and other data files that impact on a program's security-related behaviour are likely to be relatively small, whereas larger files containing images, audio and video typically would not represent security dependencies for most programs.
Figure 5.6: Matching and verification for open() system calls for reading
When re-opening a cached file, the security kernel checks its modification timestamp to determine whether the cached image must be refreshed. If the timestamp is more recent than the cached copy, the image is reloaded and re-verified; otherwise, the existing cached image is used. Note that illicit modifications that do not update the modification timestamp may be safely ignored as the previously cached and verified copy is not affected by them. However, this means such illicit modifications will not be detected as promptly and, if it is considered to be important for a particular dependency or if memory capacity or image size makes caching problematic, the No CACHE flag can be set for that specific dependency.

**The Require Trusted Data flag**

There is a need for some programs to behave in a specific way so as to maintain their expected secure behaviour. For example, programs such as command interpreters will generally run with an L2 trust level to allow them to execute non-native code with trust levels up to this value. However, if such programs execute a non-trusted script, it is important that they do not run with their normal trust level. The Require Trusted Data flag addresses this and equivalent problems with other types of programs.

The flag operates by automatically and transparently lowering the trust level of the flagged program to L0 in the event that the program reads from a file that is not specified as a dependency. This design ensures that untrusted scripts may be safely executed by command interpreters where the flag is set on these programs. It also limits the need for a program to manually and voluntarily request that its trust level be lowered, thereby increasing the backward compatibility of the Trusted Fingerprinting mechanism and avoiding the need for modifications to existing programs.

There is scope for the Require Trusted Data flag to interact with other flags. In particular, if the Require Trusted Child flag is set on the parent of a process that has the Require Trusted Data flag set, it will cause the child process to be
terminated if it accesses untrusted data and has its trust level lowered. Similarly, the Require Trusted Data and Require STL flags may also interact as the effect of the former will cause the requirements of the latter to be violated, with the result that the process will again be terminated. These interactions are not described in Figure 5.6 in order to maintain the diagram's simplicity.

5.4.10 Runtime Verification of Non-Native Code

Execution of non-native programs, such as scripts, represents a special case of Data File dependency verification. The natively executable runtime environment or command interpreter program that executes scripts is specified as a Program dependency, with all scripts that require verification specified as separate Data File dependencies. These dependencies in turn may have their own Data File and Program dependencies representing, for example, configuration and helper applications, respectively. This is why the matching process applied to the RTD for Data File dependencies is performed recursively when opening a file for reading, thus ensuring dependencies of other dependencies are checked. Execution of such scripts typically involves executing the interpreter and supplying the name of the script to be executed as a parameter. Verification is therefore performed first on the natively executable interpreter and then on the file containing the script when it is loaded. In this way, complete and secure verification of non-native code is achieved.

Non-natively executable objects can be differentiated from non-executable data as they will have a trust level specified in their fingerprint record. Consequently, the trust level of the natively executing process is lowered to meet this value where necessary. Since a natively executable interpreter running a non-native script is effectively under the control of the script, adjusting the trust level in this way is necessary in order to maintain sensible and secure semantics.

The kernel also differentiates between the roles played by the different nodes in the RTD. In particular, it distinguishes between the ‘executing process’ (EP), which represents the native code, and the ‘active node’ (AN), which refers to the actual
program notionally being executed. While for natively executable code, the EP and AN will be the same, for non-native code the EP is considered to be the interpreter and the AN refers to the non-native script currently being run. These distinctions do not affect the scheme’s behaviour when reading from Data File dependencies. However, they become important when considering modifications to these objects. This issue is further developed in Section 5.5.2.

5.5 Fingerprint Updating

5.5.1 Manual Updating

If a fingerprinted object changes, the fingerprint information in the Value field for this dependency record will need to be updated. If this change occurs independently of any program for which the object is a dependency (for example, when manually editing a configuration file using a text editor), the fingerprint value must be manually updated in the vault where it is stored. In the case of globally fingerprinted objects, only the prime user can update the values stored in GPUB. However, individual users may update the fingerprint values stored in their own vaults at any time. Manual updating involves using an L3 application and specifying the target name of the object to be updated. The new fingerprint value is then calculated and the user confirms whether to replace it in the dependency digraph. All dependency digraphs in the vault are then fully traversed searching for other occurrences of the old fingerprint value in the Dependencies field of other records, as these too will need to be updated.

Race conditions represent a serious potential security threat and manual fingerprint updating has been identified as the only significant means for potentially violating the security properties of the Vaults model. This issue is discussed extensively in Chapter 8 and a number of mechanisms for reducing, and potentially eliminating, the need to perform manual updates are discussed in the next section. Nonetheless, if performing a manual update becomes necessary, it is important that great care is
taken with this procedure. Wherever possible, users should calculate the fingerprint for an object’s new state from a trusted source, such as a digitally signed copy or data held on securely distributed read-only media. These precautions are particularly important when the prime user makes manual updates to global fingerprints.

5.5.2 Automatic Updating

Maintaining the correctness of fingerprint hash values is important to avoid false-negative verification errors that may inconvenience users. Therefore fingerprint values must be kept up to date. Similarly, as discussed in the previous section, manual updates should be avoided wherever possible as they represent a potential opportunity for subverting security. Automatic updating serves to achieve both of these goals.

In general, programs and libraries are likely to change relatively rarely, for example when software is periodically upgraded, and manual updating can be a suitable mechanism for managing these changes to fingerprint values. Manual updating may also be suitable for configuration files that, after being set initially, require only very infrequent modifications. However, some DATA File dependencies may change more frequently. Typically, however, many configuration files will be regularly modified by the application itself; manually updating these fingerprints is both inconvenient for users and increases the risk of race condition attacks. Another practical issue is that it may be difficult for the user to identify which configuration files contain security-relevant settings, and which contain general preferences, making the safest option to include all such files as dependencies. However, this approach makes manual updating even more impractical and risky as fingerprint values would need to be updated more frequently and for a greater number of objects.

Trusted Fingerprinting avoids these problems with a transparent and secure mechanism for automatically updating fingerprint values where a DATA File dependency is modified by the application that depends upon it. A summary of the process is shown in Figure 5.7. When a trusted program makes an open() system call
Figure 5.7: Matching and verification for open() system calls for writing

to open a file for writing, the kernel performs matching on that program's RTD.
However, unlike when opening a file for reading, this matching process is normally
only applied to DATA_FILE dependencies that are immediate descendants of the
active node (AN) and that do not have an STL value. This approach ensures that
automatic updates are only applied to non-executable data that are actual direct
dependencies of AN and not dependencies of secondary helper applications.

The request to open the file for writing succeeds regardless of whether a match is
found. However, if the file being modified was identified as a dependency of the
program performing the modification, the kernel captures these writes and initiates the calculation of a new hash value from them. All subsequent modifications are added to the context for the hash calculation and, once writing is complete and the file closed, the hash calculation is finalised and the fingerprint of the modified file is updated in the program’s RTD and the user’s vault. This design exploits the iterative nature of existing hash functions to avoid the need to buffer the entire file [17, pp. 86, 87].

5.5.3 Automatic Update Restrictions

There are situations where automatic updating is undesirable. To address this, the 
No Auto Update flag can be set on specific dependencies to disable automatic updating of their fingerprints. When set on a given program’s dependency, the program may make modifications to this file but fingerprints will not be automatically updated. As a result, the next time the file is opened for reading by the program (or any other for which it is a dependency), verification will fail and access is denied. This flag is intended to be used where any updates to a dependency ought to be user-verified before allowing access to a dependent program. Alternatively, the flag may be set to express the semantics that the dependency should not be modified by the program. For example, although non-natively executable source code may be a dependency of a specific interpreter, the interpreter software generally should not modify this source code.

Automatic updating is also intrinsically restricted to dependency digraphs belonging to the user concerned. The automatic updating of a fingerprint for a dependency in one user’s digraph does not cause fingerprint values to be updated in the digraphs of any other users who may have fingerprinted this object. As a result, this modification will be detected by these other users upon access. Similarly, fingerprint values in GPUB are never updated automatically. However, neither of these situations is likely as users will not generally depend on the integrity of objects that can be legitimately modified by others and non-privileged users should not be able to
modify objects that are globally trusted.

5.5.4 Installer Applications

As described in Section 5.5.2, automatic updating normally only applies to the immediate Data File dependencies of the AN where they do not have an STL. However, to further reduce the need to perform manual updating, the Installer flag may be set on any dependency with an STL. When set, this flag reduces the restrictions on automatic updates so that they may be performed on all descendants of the AN, regardless of type.

The purpose of the flag is to allow trusted installers to update installed programs and their related objects in a secure way, where automatic updates are used to eliminate the potential race conditions that exist with manual updating. This further requires that all programs that are to be updated by the installation software must be specified as dependencies of the program. However, the flag should be used with caution and only set on programs that may be trusted to take on the critical role of updating applications on the system. Note that the setting of the Installer flag does not take precedence over the effect of the No Auto Update flag.

5.6 Managing Process Trust

5.6.1 Trusted Fingerprinting and Trusted Path

As discussed in Section 2.3.1, trusted path is a mechanism found in some traditional trusted systems that allows users to gain assured access to the TCB such that they may be certain that they are not interacting with malicious code. Trusted path is traditionally implemented through some physical mechanism, such as Windows NT’s ‘Secure Attention Sequence’ where the pressing of the Control, Alt and Delete keys together is trapped by the kernel and is never passed to an application (Section 2.3.2). Through a physical trusted path mechanism, users can perform a set of specifically defined security-related operations. For example, under Windows NT,
the user can change their password, log off or shut down the system and initiate a task manager to kill selected processes. However, the operations that can be performed are limited to these predefined tasks hardcoded into the TCB and there is no flexibility for them to be customised to local or individual requirements.

The Trusted Fingerprinting scheme in Vaults also provides an authenticated means by which users can interact with secure code. However, instead of relying on a physical mechanism for ensuring communication with the security kernel, Trusted Fingerprinting utilises cryptography to verify the integrity of the code with which the user is interacting. Trusted fingerprinting therefore represents a ‘virtual trusted path’ mechanism. However, unlike traditional physical mechanisms, Trusted Fingerprinting is highly flexible, being able to be applied to any program deemed relevant to security requirements, both globally by the security administrator and locally by each individual user.

Beyond this additional flexibility, the principal difference between existing physical trusted path schemes and the virtual approach of Trusted Fingerprinting is the lack of a physical mechanism for the latter. However, the security of such a physical mechanism depends upon two assumptions. First, that the integrity of the kernel remains intact and is able to trap the requests it receives; and second, that the physical link between user and kernel has also not been compromised. In fact, the security of a virtual trusted path scheme such as Vaults Trusted Fingerprinting also depends on these same assumptions, although it lacks the psychological reassurance that stems from a physical mechanism. A user securely launching an application that, along with any external objects it depends upon, has been cryptographically verified can place a similar amount of trust in this program as if they had performed it through a physical trusted path mechanism. Indeed, the cryptographically verified code provides additional assurance in the event that the task performed is implemented in a separate program and not kernel code. Furthermore, if trusted processes that have been verified can be visually differentiated from those executing at L0 (for example, by colour coding their window border), then users are
able to identify the trust level of a program without needing to physically invoke communication with the TCB. Trusted Fingerprinting therefore provides most of the security properties of a physical trusted path mechanism but with significantly enhanced flexibility. In fact, if implemented with a mechanism to allow the kernel to clearly flag to the user the trust level of given applications (such as labelled all colour-coded title bars or window borders), then it achieves equivalent or superior security, while maintaining its flexibility advantage.

5.6.2 The Trust Elevation Problem

An issue that applies to both virtual and physical trusted path schemes is the direction of the trust involved. Trusted fingerprinting provides the user with assurance that they are interacting with trusted code. However, it is also implicitly assumed that the requests this code receives come from the user, although this is not guaranteed to be the case. For example, when one program executes another, the parent can often exercise a significant degree of influence over the child’s behaviour. Significant aspects of this include controlling its environment, command line parameters and standard input. While this is normal system behaviour, it becomes a potential security issue when the parent process is of lower privilege than the child, creating a privilege boundary between them (Section 2.2.3). This situation can occur with Vaults Trusted Fingerprinting where the parent process is untrusted and does not have access to the user’s vault whereas the program it executes has a fingerprint. In this scenario, it is important to prevent the untrusted parent from manipulating the behaviour of its privileged child to violate security policy. This issue is referred to as the ‘trust elevation’ problem.

Examples of this problem are highly dependent upon the interface of the trusted program and particularly its responses to command line parameters and environmental variables. However, finding examples of such programs is nonetheless trivial. The man program under Unix is used to remove a directory entry for a file, which

\[ ^6\text{This is the converse of the issue that the Required Trusted Child flag resolves, where a helper application is executed by a trusted parent process (Section 5.4.6).} \]
in most cases means that the data in the file will be deleted. As this program will need to access users’ directories, it must be designated at trust level L2. However, this means that malicious code executing at L0 can effectively elevate its privileges to trust level L2 and delete all files for which the user has the requisite privileges by executing the `rm` program with the parameters “-Rf /”. While such a vulnerability is easy to identify, related problems in other programs are likely to be less obvious and require careful analysis of that program’s external interface. Guaranteeing that all such problems have been identified is likely to be intractable. Therefore a more general solution is needed and this will now be described.

### 5.6.3 Limiting Process Trust Elevation

The trust elevation problem must be resolved in a general way that avoids requiring the analysis of the specific behaviour of all trusted applications in response to potential manipulation by a malicious parent. The solution is achieved by applying the general principle that a child process will not execute at a higher trust level than its parent. This principle is the origin of the distinction between the STL, which is that given in the dependency record for the program, and the ETL, which is stored in a process’s credentials that determines its current vault access privileges (Section 3.4.1).

A specific algorithm to determine where the trust level of a new process needs to be restricted is given in Figure 5.8. In the general case, the new process inherits the lower of its STL and its parent’s ETL. However, beyond this, there are two special cases to consider. One of these applies when the parent is executing at trust level L1. In this case, the new child process is downgraded to trust level L0 unless the parent and the child are the same program. This response is necessary because two processes executing at trust level L1 are likely to have different sets of privileges unless they are the same program. Note that this applies even when the child’s ETL is normally L2, as a latent set of protection key bindings may still exist if this
if parent.ETL == L1 AND parent.fpr != child.fpr
    child.ETL = L0
else
    if parent.ETL >= L2 AND child.STL == L3
        child.ETL = L3
    else
        if parent.ETL < child.STL
            child.ETL = parent.ETL
        else
            child.ETL = child.STL

Figure 5.8: Trust elevation restriction algorithm

program was previously set to run at L1\(^7\).

The other exception allowed by the algorithm is where an L2 process attempts to spawn a program with an STL of L3. In this case, it is necessary to allow the new process to elevate its ETL to L3. If this were not permitted, either L3 applications would not be able to be executed or otherwise many programs that do not require L3 privileges would have to be set with this trust level simply to allow legitimate L3 programs to be spawned. However, this exception is not problematic since, as described in Section 3.4.2, L3 applications are designed specifically with resistance to trust elevation attacks in mind and must be used interactively. In order to minimise exposure, this exception is not applied where an L0 or L1 process attempts to spawn an L3 program and, in this scenario, the child’s trust level will be confined to that of the parent. Note that in this case, while the L3 process will still execute, it will not execute at its usual privilege level and will not have the level of vault access required. As a result, the program will be unable to function and will terminate.

5.6.4 Impacts of Variable Effective Trust Levels

As a result of the algorithm described in Figure 5.8, programs may execute with an ETL lower than that specified in their fingerprint record. This will naturally impact

\(^7\)A this subtle scenario was identified as part of the attack tree analysis and is discussed in more detail in Section 2.30.
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent.ETL</td>
<td>ETL field in the data structure associated with the parent process.</td>
</tr>
<tr>
<td>parent.fpr</td>
<td>Fingerprint field in the data structure associated with the parent process.</td>
</tr>
<tr>
<td>child.fpr</td>
<td>Fingerprint field in the data structure associated with the child process.</td>
</tr>
<tr>
<td>child.STL</td>
<td>STL field in the data structure associated with the child process.</td>
</tr>
<tr>
<td>child.ETL</td>
<td>ETL field in the data structure associated with the child process.</td>
</tr>
<tr>
<td>Lo-3</td>
<td>Trust level symbolic constant.</td>
</tr>
</tbody>
</table>

Table 5.3: Key to identifiers in the trust elevation restriction algorithm

on what the program is able to do at runtime; for example, possibly preventing it from accessing objects that it would normally be able to access if executing at its STL. The exact outcomes of this will differ depending the specific program involved. Furthermore, the expectations of the user are also relevant as the user may anticipate that the program will always execute at its normal trust level. This is a more subtle variation of the issue considered when discussing matching techniques in Section 5.3.1.

If a new process cannot execute at its STL due to its ancestry, the system may either terminate it or let it continue at the lower ETL. Terminating the process has the advantage that there is no possibility of the user incorrectly believing that the program is executing at its normal trust level; this is therefore the most secure outcome. However, in many cases, this is likely to be a heavy-handed and impractical response as many programs may execute successfully even under these circumstances. For example, a program capable of displaying a given file format may be executed automatically by another process when data is encountered of this type that the parent process cannot display itself. Examples of this scenario range from simple software displaying certain types of media, to more complex applications such as web browsers and word processors. However, all of these programs may also be executed directly from the shell by the user. Therefore, such common programs could well end up being executed with an ETL lower than their STL.

To address this issue, the REQUIRE STL flag may optionally be set for all dependencies that have an STL. If the flag is set and the program cannot execute at its STL, it will not be permitted to execute at all. Also, if a flagged program has its
ETL lowered at runtime, this will also result in it being terminated. As indicated in Table 5.2, the flag defaults to \texttt{on}. However, such policy may be inappropriate for some systems and so the system-wide default should be configurable to allow administrators to trade-off security for increased usability in this instance. For programs that are likely to be executed with a variety of parents—and therefore may often need to execute at a lower trust level—it may be more appropriate to set this flag to \texttt{off}. As long as the user is aware that the program may not execute with its stated trust level, this approach has minimal impact on security. This issue further highlights the importance of user interface support in flagging the trust levels of specific applications by visual means such as a colour-coded title bar.

5.6.5 Trust Levels of Non-Program Dependencies

As previously described, libraries are not assigned a trust level in a dependency record as they may be linked with a wide variety of programs. However, a trusted program may only be linked with libraries that have been cryptographically verified, and execution fails if linking is attempted with a library that does not have a Library dependency record in the RTD. Therefore a library notionally takes on the trust level of each program with which it is linked.

The situation is more variable in relation to Data File dependencies. If a program accesses a Data File dependency that has a trust level specified, the process’s ETL is set to the lower of its current ETL and the STL of the dependency. This scenario applies primarily where the data object represents non-natively executable code that is to be interpreted by the program for execution. This approach means that the interpreter program must be assigned a trust level equal to or greater than the most trusted non-native program that it needs to execute. However, in general, trust levels should not be set for data files; opening a file for reading that does not have a dependency record, and therefore cannot be verified, does not affect a program’s ETL. Many trusted programs need to deal with untrusted data (i.e., data that comes from a source other than a file indicated as a dependency for that
program at the same trust level) and this does not necessitate them forgoing all of
their assigned privileges. However, there are situations where it is desirable for a
program dealing with data that has no fingerprint (notionally L0) to lower its ETL.
An example is the case of command line interpreters and, as discussed in Section
5.4.9, the Require Trusted Data flag addresses this issue. However, it is also
possible that a process which is Vaults-aware may voluntarily reduce its ETL for
the duration of its current execution instance by making a request to the security
kernel. However, regardless of the cause, once a process’s trust level is lowered, it
cannot revert to its previous level.

5.7 Summary

This chapter has described Trusted Fingerprinting, a cryptographic file integrity
verification mechanism that is made possible by the Vaults security model and
also enhances other Vaults security features by allowing the privileges of executing
processes to be determined based upon their authenticated identity and a user-
specified trust level. Trusted Fingerprinting allows for verification of both native
and non-native executable program code, along with other objects upon which they
depend for secure behaviour at runtime such as shared libraries and configuration
files. These dependency relationships are modelled using digraph structures that
allow the verification status of each object to be monitored at runtime. Leveraging
the information provided by this data structure, verification can be performed in
such a way as to prevent race condition attacks based on the time between verify-
ing an object and loading it into memory for use. This mechanism also facilitates
automatic updating of fingerprints for data files that are dependencies of a trusted
process. The trust level assigned to such processes is also tracked at runtime and
used to prevent untrusted malicious code from manipulating trusted programs in
order to obtain elevated privileges. Finally, the objects that are to be fingerprinted
and the trust levels assigned to these programs can be specified both globally by the
security administrator for software that affects system-wide security and locally on
a per-user basis in relation to each individual’s security requirements. This makes Trusted Fingerprinting a highly flexible and secure mechanism for code authentication and integrity verification.
Chapter 6

Security Analysis Background and Methodology

6.1 Introduction

This thesis presents a new operating system security model that maintains a similar interface to existing approaches while differing significantly through its use of cryptography to obtain a higher level of security. In particular, the Vaults model is designed to maintain its stated security properties, even in the face of an attacker who has obtained a high level of privilege on the system. It is therefore necessary to analyse the model with regards to its use of cryptography and to test the validity of this assertion. In particular, to consider the question, “is the security model able to resist penetration by a privileged attacker?” In order to do this, potential attacks available to a privileged attacker need to be identified and evaluated as to whether these enable the attacker to defeat the defined security properties. In order to identify a suitable methodology for this analysis, literature on security analysis and assessment techniques will now be reviewed. This review focuses particularly on threat-oriented techniques useful for defining valid attacks by a privileged attacker and modelling the possible outcomes of these. Following on from this, the methodology adopted for the analysis will be described in detail.
6.2 Review of Security Analysis and Assessment Literature

6.2.1 Petri Net-Based Approaches

Petri nets are a widely used tool for graphically representing and modelling systems [213]. A Petri net model consists of places, transitions and connecting arcs. Places contain tokens that collectively described the overall state of the system. Transitions are activities that occur (‘fire’) and cause tokens to move to a new place, thereby indicating a change in the system’s state. Figure 6.1 gives an example of a trivial Petri net for the purposes of illustration. In the example, transition $t_1$ fires, causing the token to move from place $p_1$ to $p_2$. According to Murata [214], Petri nets are especially suited to analyses of distributed, parallel, nondeterministic and/or stochastic systems, as well as those involving concurrency. These properties suggest the suitability of Petri nets for security analysis.

McDermott [215] first proposed the use of Petri nets for modelling security and referred to them as ‘attack nets’. Attack nets are Petri nets where the places represent security-relevant states or modes of a system, and transitions the events that alter these states. McDermott’s attack nets are disjunctive such that transitions are able to fire as long as at least one of the incoming places has a token. Depending upon the nature of the system being modelled, typically transitions represent commands or inputs that impact on the system’s security. The movement of tokens from place

![Diagram of Petri net example](image)

Figure 6.1: Petri net example
to place within the attack net as a result of these inputs shows the progress of the attack and these places model both intermediate and final objectives.

McDermott [215] suggests that attack nets are particularly useful for investigating combinations of flaws, especially those involving concurrency and multiple stages. This attribute allows attack nets to potentially show where the combined impact of a number of different flaws may be greater than the sum of their individual effects. The technique is also well-suited for describing attacks involving a sequence of attack stages, particularly where there are interdependencies between these steps. However, according to McDermott, attack nets are most suited where a bottom-up analysis is appropriate. McDermott suggests that the technique would be more difficult to apply where a top-down approach is required due to the low-level details of the system not being available; for example, where documentation is not available or when analysing a high level security model, as applies in this case.

A further limitation of attack nets is that, while the technique may reveal additional information about the vulnerabilities being analysed, it is largely unsuitable for identifying them in the first place. Instead, to facilitate vulnerability identification, McDermott recommends pairing attack nets with the Flaw Hypothesis Methodology (FHM) described by Weissman [216]. FHM is a four-stage approach for identifying, and ultimately eliminating, flaws in the system being studied. A structured approach is used for generating potential flaws that may exist. These flaws are subsequently ordered according to priority, investigated to confirm their presence, generalised to identify if the same problem re-occurs elsewhere in the system and, finally, eliminated by developing an appropriate solution [217]. The FHM approach for generating hypothetical flaws involves conducting a detailed background study using resources such as documentation and source code, followed by use of the Delphi technique and brainstorming by the team of analysts involved [218–220]. Therefore, while the coupling of the FHM and attack net techniques may be useful for a team of penetration testers analysing a new product, the approach is unsuitable for a single researcher evaluating a new security model.
The attack net approach is extended by Steffan & Schumacher [221] who recognize both the limited scope for descriptive detail in diagrammatic modelling techniques and the need to further facilitate collaboration between engineers and security analysts. To address these issues, Steffan & Schumacher combine the attack net method with a web-based ‘wiki’ collaboration tool. However, while both of these approaches can be used to model existing vulnerabilities and possibly reveal additional information about them, neither assist with identifying unknown vulnerabilities in a specific system, which limits their usefulness in this instance.

A more extensive approach employing Petri nets is presented by Xu & Nygard [222] and specifically uses Predicate/Transition nets, extended to support aspect-oriented modelling. The technique involves defining the security model (intended functions) from a detailed specification, modelling the threats by formalising anomalies that violate these goals and then specifying appropriate mitigation aspects. This approach is then applied iteratively until the model has been verified as being secure. The principal weakness of the technique is that it verifies how the threat model applies to the specified behaviour model, but has no scope for demonstrating that this model is complete. Therefore, the absence of threats in the mitigated behaviour model may be verified, but only for the specific set of threats already identified. As with McDermott’s attack nets, Xu & Nygard’s scheme requires a detailed, low-level specification in order to be applied, and its iterative, time-consuming nature makes it more suitable as a complement to informal threat modelling such as FHM rather than a stand-alone technique. In another application of Petri nets to security modelling, Fukuzawa & Saeki [223] propose the use of coloured Petri nets to evaluate the architectural design of software systems for a number of properties including security. However, the focus of this approach on software architectures makes it unsuitable for analysing security models or more general systems.
6.2.2 Graph-Based Multistage Attack Modelling

Philips & Swiler [224] propose the idea of an ‘attack graph’ for describing the interactions between a series of ‘atomic attacks’ that may be combined to achieve some ultimate outcome. By analysing these interactions between component vulnerabilities, new attacks may be discovered. While not specifically identifying the result as a graph, Templeton & Levitt [225] describe a similar concept where interrelationships between atomic attacks can be constructed by specifying their requirements (effectively pre-conditions) and the outcomes they provide (post-conditions) using an attack specification language. The authors suggest that the benefit of the technique is in the discovery of new attacks in complex systems where a number of existing attacks are already known. Sheyner Haines, Jha, Lippmann & Wing [226] improve upon this earlier work by describing an algorithm for automated generation and analysis of attack graphs rather than requiring this labourious task to be completed by the security analysts. More recently, Ou, Boyer & McQueen [227] have described a logic-based approach to improving severe scalability issues with these earlier schemes.

Unlike the Petri net-based approaches discussed previously, attack graph-based methodologies have the potential to facilitate discovery of previously unidentified attacks. However, these methodologies are restricted to attacks that may be synthesised through a combination of existing atomic attacks as there is no means for discovery of new atomic attacks. These techniques are therefore unsuitable for analysing new proposed models where no or few attacks are currently known. They are also not well-suited to analysing abstract systems or models. Instead, as Lippmann & Ingolds [228] observe, attack graph-based approaches are likely to be most effective in complex, multi-component systems, such as networks, where suggested applications include network security analysis and intrusion detection systems.
6.2.3 Attack Patterns and Profiles

Moore, Ellison & Linger [229] describe an approach for documenting specific attacks in a structured and reusable form. A major component of this documentation is ‘attack trees’, which are used for both identifying new attacks and describing existing generalised attack methodologies. Attack trees are an important analysis tool in their own right and are described in detail in Section 6.2.7. However, Moore et al. also introduce the notion of an ‘attack pattern’ as “a generic representation of a deliberate, malicious attack that commonly occurs in specific contexts” (p. 8). They propose that attack patterns and corresponding generalised attack trees can be organised into related groups known as ‘attack profiles’. These represent reusable attack groupings that may be used to analyse the security of a given system. From this foundation, the authors suggest that further work on documenting and identifying commonly occurring sets of attack patterns could lead to collections of reusable profiles.

Gegick & Williams [230] extend and formalise the textual descriptions used by Moore et al. by using regular expressions to represent attack characteristics. They also present the results of experimental studies showing that regular expression-based attack patterns can be successfully applied even by those with a relatively limited security background. More recently, the same authors have provided a detailed taxonomy of 30 regular expression-based attack patterns and profiles [231] and have related these to existing taxonomies, such as those by Landwehr et al. [232], Krsul [233] and Hoglund & McGraw [234].

The idea of reusable patterns for vulnerability identification is a useful one and Gegick & Williams’ results are encouraging. However, the principal weakness of this technique is that it is dependent upon a comprehensive database of existing flaws and their corresponding patterns. In the experiments conducted, the systems analysed were deliberately seeded with vulnerabilities specifically corresponding to the patterns that were to be used. The results show that the attack patterns methodology was reasonably successful in detecting these vulnerabilities. However,
there is currently no evidence to demonstrate their effectiveness in a real-world system with unintended and unknown vulnerabilities not inserted specifically for the purpose of being detected. Further work therefore needs to be conducted to determine whether attack patterns can reliably detect subtle vulnerabilities caused by interaction between multiple components in complex contemporary systems. Attack patterns also cannot be used for identifying entirely new vulnerabilities or those unique to individual architectures and systems. More specifically, the technique is unsuitable in this work as attack patterns do not yet exist for the newly devised security model being studied. This is reinforced by the focus on implementation-related vulnerabilities in the attack patterns described by Gegick & Williams [231].

6.2.4 Security Metrics and Assessment Techniques

Numerous metrics for security assessment exist, although many of these are focused along organisational lines [235]. Wang & Wulf [236] describe a preliminary framework for quantifying security utilising a tree-based decompositional approach, while Alves-Foss & Barbosa [237] take the opposite approach by attempting to quantify the susceptibility of a computer system to attack, based on a number of generic factors influencing security.

A more specific and precise approach to security measurement is given by Manadhata & Wing [47]. This metric is based on the notion of an attack surface, which represents externally visible system resources and the actions associated with them. Intuitively the more actions and resources exposed, the more vulnerable the system will be to attack. A state machine is used to formally model both the system and the threats involved. A subsequent paper by the same authors [48] adds a framework to assist identification of resources that contribute to the attack surface and which therefore must be included in the measurement. Weightings can be assigned to these resources based on the number of times they are influenced by system actions, the number of reported vulnerabilities associated with this resource, and
the damage resulting if successfully compromised. These weightings may then be used to compare the systems being studied; an example is presented in that paper comparing the security of four different Linux distributions.

Unfortunately, the technique has a number of limitations. For example, the process is particularly dependent upon the expert domain knowledge of the analyst. There are also a large number of choices involved in precisely how the metric is applied. As a consequence, while the results of the technique are quantitative, they generally support only relative comparisons between the systems being analysed. These results are therefore not externally valid as they cannot then be compared to systems studied separately, unless the metric has been applied in precisely the same way in all cases. Further, the technique cannot be applied to a single system in isolation. In its current form, attack surface-based methodologies are therefore not suitable for assessing abstract security models, particularly in isolation, and therefore are not suitable for analysis of the Vaults scheme.

6.2.5 General Threat Modelling Methodologies

A variety of security analysis and modelling methodologies have been developed that typically take a risk or threat-driven approach [238, 239]. The CORAS methodology is typical of these and specifies a seven stage process for conducting a security analysis. Den Braber, Hogganvik, Lund, Stolen & Vraalen [240] present a description of CORAS, including a detailed example. Step 4 of CORAS involves risk identification and is therefore relevant to the problem at hand. The suggested approach is referred to as ‘structured brainstorming’, where ideas for potential attacks are workshopped by the analysts in a manner that somewhat resembles the flaw hypothesis methodology [216] described previously in Section 6.2.1. The results are documented using the diagrammatic ‘CORAS security risk modelling language’ to construct threat scenario diagrams. These are graph-like but involve a rich symbol set and show the connection of a threat source to a set of one or more vulnerabilities to specific threat scenarios and ultimately to the assets affected. These diagrams
are expanded during brainstorming sessions and different kinds of threats are modelled in separate diagrams. Although den Braber et al. suggest that the CORAS diagrams are inspired by tools such as attack trees, they lack the latter's ability to assist in the generation of new vulnerabilities and, instead, rely heavily on the group brainstorming process. As a result, the technique cannot be used for this research.

6.2.6 Argument Trees

Kienzle [1] introduces a technique called Methodically Organised Argument Trees (MOAT)\(^1\) where a tree-based structure is used to develop assurance that a given system is able to provide specified security properties. Although Kienzle’s work pre-dates attack trees (described in the next section), he acknowledges the influence of the earlier technique of fault tree analysis on his methodology [242]. Kienzle’s argument trees are less threat-orientated than these other techniques and have a desired security property as their root node. This property is then decomposed into its detailed requirements in a hierarchical, top-down fashion. This approach allows assumptions made about security properties and their provision to be identified and documented at multiple levels. However, it also facilitates verification of these properties by assigning justifications to each node in the tree. Internal nodes contain arguments that this property does in fact decompose to its child nodes, while leaf nodes contain a justification as to why the property they assert is true. The actual form of the justification is flexible and depends upon the goals of the analysis, the level of threat associated with the node, and the level of rigour required for a sufficiently convincing argument.

Argument tree nodes may be conjunctive (AND) or disjunctive (OR), with the former reflecting dependence on multiple sub-properties and the latter either design alternatives or defence in depth. An example argument tree for maintaining the confidentiality of a user’s e-mail is given in Figure 6.2. A small icon beneath the

\(^1\)A summary of the technique by Kienzle & Wulf is also available [241].
node box indicates whether the node is conjunctive (convex top with a flat base) or disjunctive (angled top and concave base). For example, the node “User’s e-mail kept private in transmission” is conjunctive and requires that the e-mail is both encrypted and that the encryption key is known only to the sender and receiver. By comparison, the node “User’s e-mail kept private on disk” is disjunctive and indicates that either encryption or OS-based security measures may be used to satisfy this requirement.

Kienzle demonstrates the practicality and effectiveness of the methodology by applying it to the ‘Legion’ security model [243]. Kienzle observes that this facilitated discovery of the security properties required to meet its security objectives, and that employing the methodology as part of the design and development process helped to inform the compromise between design criteria such as performance and the established security goals. The results of these analyses aided communication between developers and proved to be easily accessible to those involved in the project. Other more general benefits were also reported. The approach is flexible in that it does not provide a fixed notion or level of evaluation and may be integrated with other additional supplementary validation techniques as required. Furthermore, identifying the underlying assumptions behind various security mechanisms
enabled these designs to be potentially reused in the future in other scenarios where the same assumptions apply. A similar approach has been successfully employed more recently by Dojen & Coffey [244] to support logical verification of cryptographic protocols. However, a limitation of the approach is its dependence upon other verification methodologies in order to obtain a high level of assurance. If these supplementary methodologies must be applied at multiple levels in the tree to verify the security property arguments made at these points, this would result in the amount of analysis needing to be performed rapidly escalating. Consequently, this may necessitate some of the analysis work being omitted, with the result that the argument tree structure primarily serves to highlight the areas where the analysis fails to adequately support the claimed security properties.

6.2.7 Attack Trees

Attack trees have their origins in fault tree analysis [245], which has been used for decades in the domain of safety-critical systems such as those in the nuclear and aerospace industries [242, 246]. Fault trees have also recently been applied to security. Foster [247] describes a model for security protocol development that includes an enhanced fault tree analysis technique for determining protocol security requirements. Brooke & Paige [248] also show how fault trees may be used to analyse security-critical systems, while Helmer et al. [249] apply the technique to requirements analysis for intrusion detection systems, with coloured Petri nets later employed to aid design of detector mechanisms [250].

However, attack trees dominate fault trees in the security literature and are discussed in relation to a wide range of security analysis, verification and modelling topics [215, 221, 222, 227, 229–231, 238, 240, 250–254]. Schneier [255] is generally credited with being the first to introduce the term ‘attack trees’2 and he describes them as “a formal methodology for analysing the security of systems” (p. 21). The tech-

---

2Some authors refer to these as ‘threat trees’ [49, 253, 256]. However, these should not to be confused with the threat trees described by Amoroso, Kleppinger & Majette [257] which, Foster [247] argues, use a more restrictive threat categorisation scheme.
nique is also widely used by practitioners for threat modelling and vulnerability analysis in systems and software [2, 3, 28, 49].

**Construction and Use**

The tree and associated analysis are highly threat-centric, with the root node being the overall goal and leaves being different specific ways of achieving this goal. However, trees will typically also have a number of internal nodes that are incomplete in the sense that they represent a general means of achieving the overall goal but still lack the required detail and specificity. In this way, attack trees are a highly top-down approach facilitating refinement and decomposition of generic classes of attack into more specific attack modes. As a result, Moore et al. [229] refer to attack trees as being ‘referentially transparent’ such that the lower-level details of an attack are abstracted in the higher level nodes. This property in particular assists in the management of complexity associated with analysing detailed attacks of large systems and facilitates convenient navigation of these trees. It is this design that also makes attack trees so useful as a structured and systematic means of deriving previously unidentified vulnerabilities in the system being analysed [215, 229]. As with argument trees, internal nodes are characterised as either disjunctive (OR) or conjunctive (AND), with OR being the default in most notations. Disjunctive nodes allow a choice so that the attack goal is attained if any of the sub-goals is satisfied, whereas conjunctive modes require satisfaction of all sub-goals. In a qualitative sense therefore, a large proportion of OR nodes indicates greater opportunities for achieving the overall attack goal whereas AND nodes suggest greater constraints on the attacker.

Figure 6.3 gives an example of a simplified attack tree for learning a user’s Unix password. The notation used in the diagram is one of several commonly used and represents leaf nodes as a square, OR nodes with an angled top and concave base, and AND nodes with a flat base and convex top. These symbols are similar to those used as part of Kienzle’s [1] argument trees. At the root of the tree is the stated
Figure 6.3: Example attack tree for learning a user’s Unix password

overall goal and, in this example, there are four different methods of achieving the
goal given at the top level. In all but one instance, these methods are subsequently
decomposed further to indicate more specific ways of achieving the sub-goal. De-
composition continues until a sufficient level of detail appropriate for the analysis
is achieved. The example in the diagram is highly simplified as decomposition is
halted very early. Note that, in all instances, tree branches terminate in a leaf
node. Also observe how OR nodes describe options available to the attacker. For
example, the attacker has the choice of attempting to convince the user to reveal
their password either by threatening them or tricking them through ‘social engi-
neering’. Depending on the analysis, OR nodes may also reflect different scenarios
that might arise. For example, the work that is required for the attacker to obtain
the password hash is different depending upon whether these hashes are stored in a
world readable password file or ‘shadowed’. Finally, AND nodes describe multiple
constraints placed on the attacker. In the example, as Unix only stores one-way
hashes of passwords, in order to recover the plaintext password the attacker must
both obtain the hash and then successfully mount a dictionary attack on it. This
requirement is modelled through the AND node in the diagram.

Values may also be assigned to leaf nodes to facilitate further analysis beyond sim-
ply vulnerability enumeration [256]. Schneier [255] suggests that the use of simple
Boolean attributes such as possible/impossible or expensive/inexpensive may be appropriate, depending on the intended purpose. With appropriate application tool support, these attributes allow the tree to be ‘pruned’ to remove branches that are logically impossible according to the determined criteria. Alternatively, use of continuous values, like cost or probability, facilitates analyses such as removing attacks with the cost over a certain value or retaining attacks that are more likely than not. Assigning multiple attributes enables more sophisticated querying of attacks with different characteristics, increasing the flexibility and reusability of the information captured in the tree. Although trees are often represented in diagrammatic form, which is the most intuitive approach, complex trees can also be written in textual ‘outline’ form for convenience [3, 255].

Enhancements and Applications

As noted earlier, attack trees have received considerable attention in the security literature and this has resulted in numerous enhancements and extensions. As detailed previously in Section 6.2.3, Moore et al. [229] incorporate attack trees into an approach for documenting attacks in a structured and reusable form. Salter, Saydjari, Schneier & Wallner [45] also use attack trees as a key component in a broader secure design methodology. This methodology uses the results of modelling likely adversary behaviour and objectives and combines them with attack trees to inform application of countermeasures. Tidwell, Larson, Fitch & Hale [258] augment attack trees by adding multiple parameters, pre- and post-condition assertions, and a complementary specification language to allow “systematic visualisation, vulnerability assessment and attack prediction” (p. 54) in the context of a network intrusion detection system. Buldas, Laud, Priisalu, Saarepera & Willemsen [256] also employ multi-parameter attack trees but select them according to game theory in an attempt to improve the quality of risk estimations. However, the authors acknowledge the difficulties in quantitatively estimating risk for unknown parameters and it is consequently unclear whether the level of precision obtained in the results of
this specific approach is fully justified.

Swiderski & Snyder [49] present their version of attack trees, which include both unmitigated and mitigated threat conditions in the same tree. Dowd, McDonald & Schuh [2] also suggest including mitigation nodes in attack trees. These are drawn with a different (circular) symbol so as to clearly differentiate them from attack nodes. However, Dowd, McDonald & Schuh also urge caution on the addition of mitigation nodes to an attack tree as this may prematurely curtail exploration of seemingly unlikely attack vectors. Dowd et al. therefore recommend some form of validation of these countermeasures before they are included in the threat analysis. McDermott [254] describes a similar process suitable as a lightweight assurance methodology centred around abuse cases. Finally, Yager [259] introduces the concept of ordered weight averaging (OWA) trees that support node types beyond the simple disjunctive and conjunctive types commonly used. Non-leaf nodes in Yager’s OWA trees may have weightings assigned to them to support modelling in cases where there is some probabilistic uncertainty concerning how many of that node’s children must be satisfied for the branch as a whole to be evaluated as being successful. While potentially valuable for certain types of analysis, Yager’s extension does not appear to have been otherwise adopted.

There has also been some work on developing a more fundamental understanding of attack trees. Mauw & Oostdijk [260] describe a formalisation of attack trees that uses the notion of attack suites to abstract the internal structure of the tree. The aim of this work is to clarify the semantics of trees and the transformations that may be performed on them in order to assist work on automated support tools for attack tree analysis. An early example of such a tool is that by Opel [261], although this tool appears to no longer be available. However, sophisticated commercial tools such as SECURITree [262] have appeared to fill the gap.
Advantages and Limitations

Attack trees have numerous advantages. Schneier [255] argues that they provide structure to the security analysis, taking an ad hoc brainstorming process, such as that involved in the CORAS and flaw hypothesis-based approaches [216, 240], and replacing it with “a repeatable methodology” [29, pp. 332]. According to Tidwell et al. [258], attack trees can also support expression of multi-stage attacks. Once constructed, trees with labelled nodes can be further analysed according to probability of attack, cost constraints or other parameters as appropriate [256]. This process is supported through the use of software tools [261, 262]. The hierarchical structure also allows multiple experts from within a single group to work on different branches of the tree concurrently [221]. The methodology is both intuitive and flexible and the literature contains examples of its application in scenarios ranging from e-mail message security [29, 247, 255] to network attacks [258] and even, less seriously, how to get a free meal at a restaurant [260]. McDermott [215] suggests that the top down, decompositional structure of the trees facilitates the ability to analyse systems for vulnerabilities, even where low-level details are not available. This makes attack trees more suitable for the current project than bottom-up analysis techniques, such as attack nets described in Section 6.2.1. Attack trees therefore provide “a standardized approach for identifying and documenting potential attack vectors” [2, p. 59].

Although they have been widely adopted, some limitations of attack trees have been observed. Steffan & Schumacher [221] found that they lacked the ability for modelling preconditions and the graphical format restricted the capacity for explanatory detail. However, the combination of attack trees and attack patterns devised by Moore et al. [229], and extended by Gegick & Williams [231], avoids this problem. Furthermore, software tools such as SECUITREE have extensive support for incorporating and categorising detailed notes for each and every node in the tree being analysed [262]. The other principal deficiency of the technique is that it is highly dependent upon the expertise of the analysts who construct it [3]. However,
it has been observed that this is essentially unavoidable and is common to other equivalent analysis and assessment techniques [215,221,263].

6.2.8 Summary and Conclusion

Of all the analysis methodologies considered, attack trees are most suited to the current work. Unlike attack nets and attack graphs, attack trees can be applied to a high-level security model where some level of abstraction is involved and do not require a detailed low-level specification in order to produce worthwhile results. Significantly, attack trees are also useful as an independent technique for identifying unknown flaws in a new system. On the other hand, McDermott’s [215] attack nets are more useful for gaining additional information about known vulnerabilities, while Xu & Nygard’s [222] application of Predicate/Transition nets serve to demonstrate the absence of a set of specified threats. Other techniques for the elucidation of new vulnerabilities, such as FHM and CORAS [217,240], are heavily dependent upon brainstorming-like processes within a team. As Schneier [255] argues, these somewhat ad hoc processes lack the methodical approach of attack trees to vulnerability discovery. Attack patterns are also unsuitable for identifying new vulnerabilities, especially those that are specific to a particular system. Manadhata & Wing’s [47] attack surface measurement technique is useful for notionally comparing the relative security afforded by two or more systems but cannot be applied to a single system in isolation. Finally, Kienzle’s[1] argument trees provide valuable structure to other non-threat orientated analysis techniques but do not necessarily provide sufficient rigour on their own.

Since an entirely new system is being analysed, the identification of potential vulnerabilities is of particular importance. The existence and nature of potential vulnerabilities is a critical indicator of both the overall security of the model and of the validity of the proposition that use of cryptography within an operating system security model provides significant security benefits. Furthermore, the ability of attack trees to manage complexity supports a rigorous analysis by a single researcher.
in a way not feasible with other approaches. Consequently, in this study, an analysis methodology based around attack trees was employed for assessing the security of the Vaults model.

6.3 Analysis Background and Methodology

6.3.1 Overview

This thesis argues that the use of cryptography in operating system security models has the potential to significantly improve security over conventional approaches. In particular, it is claimed that certain security properties can be achieved, even in the presence of an attacker who has attained a high level of privilege on the system such as would be sufficient to bypass security entirely under conventional models.

To test this claim, attack trees were developed to model potential attacks on each of the Vaults scheme’s stated security properties. Specifically, three trees were constructed to analyse the confidentiality property, the verifiable integrity property and the Trusted Fingerprinting mechanism. The assumptions of the analysis (detailed in Section 6.3.6) were set to allow for the possibility that the attacker had obtained a privilege level equivalent to superuser access under traditional Unix systems. This meant that the attacker could bypass the active enforcement of security properties by the kernel and directly access and modify data stored on disk. The attack trees sought to identify the different possible ways that the system might be attacked under these assumptions in order to ascertain to what degree the Vaults security model is able to resist such attacks by a highly privileged adversary. These attack trees also served more generally to validate the model’s security and present evidence as to its robustness. Details of the assumptions and methodology employed in this analysis will now be described.
6.3.2 Trees and Security Properties

As just stated, three attack trees were constructed, one for each of the three principal security properties provided by the Vaults model. These properties were defined as follows.

**Confidentiality Property** Within the limits of the relevant underlying cryptographic primitives and algorithms, Vaults guarantees the confidentiality of objects designated as being read-protected. This property is defined as the plaintext content of designated objects can only be read by those with a valid access ticket and only those authorised users will be able to use such a ticket.

**Verifiable Integrity Property** Vaults guarantees the verifiable integrity of objects designated as being write protected. This property is defined such that only authorised users in possession of a valid and usable ticket may make legitimate modifications to write protected objects and that any other modifications will be detected upon subsequent legitimate access to the object, regardless of mode.

**Trusted Fingerprinting Mechanism** This mechanism within the Vaults model encompasses a number of specific security properties, namely:

- Trusted code and its dependencies accessed via a specified filesystem path are cryptographically verified on execution or access (as appropriate) to confirm that these have not been modified.

- Privileges of trusted code determining access to cryptographically protected objects are assigned based on whether this code has been successfully verified and the code’s corresponding specified trust level. A corollary of this property is that unverified code is considered untrusted, has no access to the user’s vault and therefore has effectively no privileges under the Vaults security model.

- Access to application keys is dependent upon the cryptographically verified identity of a process.
• The privileges of authenticated trusted code are constrained such that this code may not be used indirectly by malicious code to obtain unauthorised privileges.

Attack trees were constructed to address each of these properties. In the case of both the confidentiality and verifiable integrity properties, each property was analysed with a single tree. However, the attack tree analysing the Trusted Fingerprinting mechanism incorporates a separate top-level branch to consider each of the mechanism’s sub-properties.

6.3.3 Tree Construction Methodology

The attack trees were generally constructed by following the top-down decompositional approach as described in the literature. However, as the work progressed, a refinement of this approach was developed and was employed to make the attack tree construction process even more methodical. This improved methodology involves dividing a high-level node into a set of logically complementary attack scenarios. Particularly at the higher levels there are often only two mutually exclusive alternatives, which leads to this portion of the structure resembling a binary tree. However, the principle is also often applicable where three or more branches are required.

The technique is best applied by beginning with the highest node in the tree that represents an attack scenario. In most attack trees, this will be the root node. This was true for all of the trees in this analysis except for that analysing Trusted Fingerprinting, which included attacks on four distinct but closely related security properties. From the high-level attack scenario node, the analyst should identify a specific high-level property that may apply to a subset of the attacks within this general scenario. If it is a binary property such that it either applies to the specific attack scenario or does not, the analyst should create two branches from the high-level node to reflect each of these cases. Where there are three or more attack scenarios that collectively address all possible outcomes, a separate branch
is made for each of these. The critical feature of this approach is that, collectively, all branches logically encompass all possible scenarios. Typically the easiest way of ensuring this is to develop the analysis using binary properties of the attack where possible. The process is subsequently repeated for each of the new branches in the tree with another attack property of the refined scenario identified and another set of logically complementary attack scenarios created from this point. An example of how the process was applied in this analysis is given in the next section.

Clearly the order in which the decomposition is applied is important. If a sub-property is found to apply in both branches of its parent, this strongly implies that it should appear further up in the hierarchy. As a result, the tree may require some minor rearrangement while it is being developed and this rearrangement should occur as early as possible.

It was found during the analysis that the technique described was most beneficial higher up in the tree. Iteratively categorising scenarios according to whether hierarchical attack properties were applicable to a specific attack helped to ensure that less obvious scenarios were still identified. This outcome is particularly important at the upper levels of the tree, as neglecting an entire branch of high level attacks at this point as a result of an implicit assumption has the potential to lead to a large number of scenarios being unintentionally omitted from the analysis. As the process for applying the technique became clear, branches in the tree that had been constructed previously were reviewed and restructured explicitly according to the improved methodology. This process proved to be beneficial as several additional attack scenarios were subsequently identified and the overall structure of the tree became clearer and more logical.

6.3.4 Logically Complementary Attack Scenarios Example

Figure 6.4 gives an example of the application of the logically complementary attack scenarios technique. Specifically, the diagram shows a relatively high level
view of the “Defeat Trusted Fingerprinting” tree. At this high level, virtually all of the branches demonstrate the technique being described. For example, in the leftmost branch considering attack scenarios where an object is modified and the modification is not detected, the scenario is subdivided into those scenarios where the modified object is stand-alone trusted code and those where it is a dependency of some other program. These two cases cover all possible scenarios as, under the model, they are the only two possibilities. Furthermore, if the modified object is a dependency, it can either be a helper application, a shared library or a data file dependency. As these are the only three possible types of dependency, again collectively these are comprehensive. Note this is an example of a case where the attack property being considered cannot be treated as binary as three branches are required.

The second top-level branch of the tree analyzes attacks where privileges are incorrectly assigned. This node is divided into scenarios where privileges are incorrectly assigned to a trusted program and when they are incorrectly assigned to an untrusted program. As programs cannot be anything except either trusted or untrusted, again this incorporates all possibilities. The same process is also applied to this leftmost branch, as the only two ways that a trusted program can be assigned privileges incorrectly are if it executes at its specified trust level (STL) when it should not or if it does not execute at its STL when, in fact, it should. While many of the subtrees in the diagram are rolled up, a similar process has been repeated within these subtrees where appropriate.

In some cases the methodology suggests the inclusion of nodes within the tree that are trivially invalid. For example, in the right most top-level branch in the diagram considering where a trusted program is subverted by its malicious parent, the child of a malicious program executing at trust level L1 may either be the same program or a different program. However, if the parent and child are the same then there is no difference in privileges and no ability for the parent to manipulate the child. As a result, the subtree describing scenarios where the child program is the same as its

---

3Refer to Section 206 for details on the attack tree notation used in this diagram.
Figure 6.4: Example of logically complementary attack scenarios methodology
parent is simply written as an impossible leaf node. This is an example of where the methodology may lead to a small amount of overhead in terms of additional nodes in the tree. However, this redundancy does not add to the overall complexity of the tree and, in fact, may simplify the process of navigating the tree due to the more methodical and logical way in which the attack scenarios are structured.

6.3.5 Trees and Libraries

The attack trees in this analysis were constructed with the SecurITree software package. As already stated, three separate trees for each aspect of the Vaults model were created. However, early on in the analysis, it was identified that the details of lower level attacks were often the same and that these attacks frequently re-occurred as specific ways of achieving different high-level goals. One of the features of the SecurITree software is the ability to create ‘attack tree libraries’. These are reusable subtrees that can be incorporated within other trees as required. Libraries can also include other libraries, which creates a ‘meta-hierarchy’ beyond the structure of the actual trees themselves. If a modification is made to any library, loading or reloading a tree which uses that library results in all instances of the library being updated to reflect the latest changes. This feature is a powerful tool for the construction of sophisticated and detailed attack tree models and was pivotal in enabling the degree of comprehensiveness achieved in the analysis.

As the tree analysing the confidentiality property was developed first, libraries were constructed to reflect attacks on this property. Each library was assigned an alphabetic identifier which was assigned as a prefix to its descriptive label. For example, the first library developed for the tree analysing the confidentiality property was named: \textit{(A)Modify GPUB without Fundamental Ticket}.

When the analysis progressed to considering the verifiable integrity property, it was identified that these existing libraries could not be applied in their original form to the new property. In some cases this was due to often subtle differences in the

\footnote{Use of the software was very generously provided by Amenaza Technologies Limited.}
way specific attacks applied to the different security properties. Alternatively, while the general attack structure was the same, sometimes the labels and descriptions of the specifics of the attack needed to be altered to reflect the different context. However, the most significant change involved interdependencies between the two security properties. Specifically, a number of attacks described in the confidentiality property tree and its libraries required violation of the verifiable integrity property. To reflect this interdependency between the two trees, the labels for these nodes were prefixed with the flag $TREE$. When considering the confidentiality property, these nodes were initially assumed to be possible in order to identify the impact of the interdependence between the two properties. However, if an attack on the verifiable integrity property requires defeating the same property to succeed, this attack has becomes circular and therefore should be excluded from the analysis. Therefore, when constructing the verifiable integrity property tree, all of the libraries developed for analysing the confidentiality property were adapted and revised with separate versions being created. To clearly distinguish between these new versions, all attack tree libraries corresponding specifically to the verifiable integrity property were flagged with the prefix [/]. When analysing the Trusted Fingerprinting mechanism these libraries were again reviewed. However, for this tree, only one of the subtree libraries needed to be revised and modified. When the analysis was completed, a total of 41 attack tree libraries were created in addition to the three primary trees.

**Construction Optimisation**

The use of attack tree libraries significantly assists the analyst in managing the complexity associated with larger attack trees. However, modelling the security properties at such a level of detail resulted in very large attack trees being developed. Details of trees are described in Chapter 8. As the trees grew in size during the development, there was a concomitant reduction in performance and the time

---

5In some rarer cases, an attack on the confidentiality property required violation of only a subset of the verifiable integrity property. In these cases, the node was flagged with the prefix $PROP$ and again was treated as being possible.
taken to perform basic tasks, such as creating and editing nodes, increased substantially. In particular, typically when the tree reached a size of between 5,000 and 10,000 nodes, this slowdown would begin to significantly impact the speed at which construction could be performed. Although the SecurITree tool remained extremely reliable, to improve general responsiveness, one of the most common and largest attack tree libraries, $\langle H \rangle$, was compressed down into a single node stub which was used instead of the complete library while the construction work was being performed. This technique significantly increased the speed at which the trees could be constructed. As the stub node was stored as a library, overwriting this file with the complete version of the library and reloading all of the trees allowed the full tree to be conveniently reconstructed when required for analysis.

### 6.3.6 Assumptions

A number of assumptions were made to define the scope of the analysis. This was important both to ensure all necessary attacks were included and to exclude attacks not relevant to the questions being considered. As discussed in Section 6.3.7, assumptions were used in determining the value to assign to the leaf nodes in the tree. If the actions required by a leaf node violated one of these assumptions, the value of the node was set to impossible. Otherwise the node was considered possible for the purposes of the analysis. Each of these assumptions will now be discussed.

**Assumption 1** Keys, including passwords, contain sufficient entropy to be secure against any guessing or cryptanalytic attacks within the capabilities of the attacker.

Attacks targeting low entropy keys are often successful in compromising the security of systems that depend on these keys. As Vaults employs cryptography as the underlying means for attaining superior security, the strength of such keys becomes critically important. However, this is a well understood
issue and can be avoided in practice through the generation of cryptographically secure keys, selection of high entropy passphrases and/or use of additional authentication technologies such as smartcards. Further, the problem is not specific to Vaults and applies in the same way to other systems. Therefore, attacks involving this do not reflect a fault with the Vaults model but rather a more general issue affecting systems that utilise passwords. As a result, these problems are excluded from the analysis under this assumption with the acknowledgement that they are an important issue that must be addressed in practice.

Assumption 2 Users will adhere to appropriate practices required to keep password and key values secure, including, but not limited to: not writing passwords down; not revealing passwords over unsecured remote login sessions; and making use of physical trusted path mechanisms where appropriate and available. Furthermore, users will not reveal their password or other confidential values directly to an attacker, although they will supply these to the computer system where it appears appropriate.

There are a myriad of ways in which users could reveal sensitive values, such as passwords, to an attacker. However, again these are not specific to the Vaults model and therefore were not included in the analysis.

Assumption 3 Prior to an attack commencing, the initial integrity and correctness of the security kernel and trusted computing base (TCB) files and configuration are assumed. This includes the following:

- The kernel correctly implements the specified security model.
- Other TCB components are correctly implemented according to their specified requirements under the model.
- The integrity of all TCB software (including L3 applications) is intact at the point at which it is received from an authorised distribution point. This also includes the integrity and correctness of all subsequent author-
risid updates.

- The configuration and use of the security mechanisms is correct. For example, L2 and L3 kernel files, applications and relevant dependencies are cryptographically write protected.

However, this assumption does not exclude the general prospect of malicious code or attacks on the kernel and TCB integrity at runtime pursuant to Assumption 5.

This assumption is also intended to ensure that the focus of the analysis is the design of the model and not a range of external and largely universal factors. For example, including attacks requiring the model to be implemented incorrectly introduces an effectively infinite number of ways that the model might fail, the bulk of which are not specific to the model’s design. Problems caused by misconfiguration are also excluded as they are the user-level equivalent of programming errors. Similarly, attacks involving subversion of the software prior to it being received by the user are also excluded as they are unconnected with the model itself and could equally affect any scheme.

**Assumption 4** Users will not mistakenly give away their privileges or otherwise explicitly undermine their own security through any overt mechanism or series of steps. For example, it is assumed that the user will not give away their privileges by issuing secondary tickets or granting arbitrary code and its dependencies a trust level of L2 or L3, regardless of how convincing the attacker’s social engineering is, as these actions involve explicitly giving away the user’s privileges. However, covert attacks, such as inducing the execution of malicious code, are not excluded by this assumption. Indeed, the existence of malicious code is explicitly assumed.

As with any discretionary system, users can compromise their security by giving away privileges in an almost unlimited number of ways. In the cases where the leakage of privilege involves some explicit privilege-granting action on the part of the user, these actions are excluded from the analysis. Otherwise a
large set of spurious vulnerabilities would be identified that result solely from
the user’s behaviour and have nothing to do with the security model being
analysed. However, the analysis also acknowledges that more subtle attacks
exist that do not involve the user directly assigning privileges to the attacker.
Such attacks are highly relevant to the question of the model’s security and
are therefore included in the analysis.

**Assumption 5**  *It is assumed that the attacker already has some form of access on
the system such as a local user account that allows them to log in and work in-
teractively. This assumption also encompasses the possibility that the attacker
may have attained a high level of privilege equivalent to superuser access that
allows them to bypass the security kernel and directly access secondary storage
devices on the system.*

Although in reality the bulk of attackers may not initially have access to the
system at all, the purpose of the analysis is to consider the security prop-
erties provided by Vaults and not the difficulty of overall penetration of the
system by an outsider. Therefore, in practice, this is likely to be a generous
assumption in favour of the attacker. However, this assumption is critical
to validating the hypothesis that the Vaults security model can continue to
provide its stated security properties even against highly privileged attackers.

**Assumption 6**  *The isolation of the kernel and its data is assumed such that an at-
tacker cannot directly manipulate or subvert its behaviour at runtime. There-
fore, the kernel is able to maintain itself and data securely in memory, inac-
cessible to all system processes.*

Hardware-based support for restricting user process access to designated areas
of memory is available on a typical computer system and this is required for
the kernel to operate and enforce the security model correctly.

**Assumption 7**  *All cryptographic algorithms and primitives used by the system are
assumed to be sufficiently secure in design and implementation to resist any*
and all cryptanalytic attacks at the attacker’s disposal.

While the Vaults model is highly dependent upon cryptography for achieving its stated security properties, the security of specific algorithms or primitives is not relevant to this analysis. As a result, all forms of cryptanalytic attack on these algorithms and primitives are excluded by this assumption.

**Assumption 8** It is assumed that the user will not trivially leak plaintext from read-protected objects to which they have legitimate access; for example, by copying this text into an unprotected file. This assumption also excludes a similar incidental leaking of plaintext by legitimate (non-malicious) applications in the kind of failure described by Czeskis et al. [264].

As with Assumption 4, this assumption excludes behaviour by both the user and their applications that is not malicious but nonetheless involves explicitly and unavoidably causing security to fail. While such problems could be mitigated by extending the model to monitor and restrict information flow, mandatory security policies of this type are not within the scope of the model and were therefore excluded from the analysis.

**Assumption 9** The security of unprotected objects is not considered in the analysis.

The defined security properties only apply for designated objects and this assumption makes it explicit for the purposes of the analysis. Therefore, attacks that affect unprotected objects, but do not negatively impact the security properties that apply to those objects designated as being protected, are not considered to be successful attacks within the scope of this analysis.

**Assumption 10** It is assumed that the hardware on the system provides the necessary security features, implements these correctly and cannot be subverted by the attacker. Specifically these features include memory protection, the ability of system hardware to provide a secure bootstrap process to ensure the integrity of the system kernel prior to booting and the presence of trusted platform mod-
ule (TPM) chips as required.

The Vaults model only requires basic hardware security features as are typically found in commodity PC systems. However, this assumption requires that these features work correctly and are sufficiently resistant to attack. It also addresses the need for the hardware to facilitate a secure boot sequence to ensure that the attacker cannot tamper with or replace the kernel with a malicious version as a means of bypassing the security of the model. Secure booting may be achieved through use of a TPM chip or by booting from physically secured read-only media such as a CD-ROM.

**Assumption 11** Administrative practices and protocols undertaken by the prime user are assumed to be conducted with sufficient care and rigour as to not result in any reduction of the security provided by the Vaults model. This assumption includes both online and offline procedures and particularly relates to Vaults PKI administrative tasks, such as verifying user identity and signing of certificates. It also involves correct configuration of the system where required to avoid or mitigate well-known avenues of potential attack. In general, this assumption addresses the integrity of procedures that are necessary to support the Vaults security model but are not central to the model itself and cannot be guaranteed by it.

This assumption is an extension of those previously described, addressing the need for secure configuration and excluding user error but specifically applying to the actions taken by the prime user. If the system is not maintained correctly and the prime user does not properly authenticate users’ identities before issuing certificates, this has significant potential to result in a failure to achieve the stated security objectives but not as a result of failure in the model itself. For this reason, these events are excluded from the analysis.

**Assumption 12** The Vaults model specifies that certain encrypted items used within the model must also be integrity protected. However, the specific means of protection (for example, a MAC or authenticating block cipher mode) may be
selected by the implementer. Since the precise means of attacking this mechanism in order to change an integrity protected parameter depends upon the mechanism used, for this analysis it is assumed that if the attacker learns the encryption key they may also violate the ciphertext authentication.

This assumption models the scenario where an authenticating block cipher mode is employed and only a single key is used for encryption and authentication. However, if a MAC were used instead, this might require the attacker to obtain an additional key. Therefore, this assumption establishes the scenario that is most favourable to the attacker. This issue is further discussed in Section 7.2.

Other Excluded Scenarios

In most cases where an attack was excluded from the analysis, this was because it had violated one of the above assumptions. However, in some cases, attacks were excluded due to their requirements having become circular. For example, in order to steal a user’s private key from their vault to allow the decryption of a secondary ticket available to this user in GPUB, the attacker would need the victim to execute malicious code that has been designated as L3. In order to assign their code trust level L3, the attacker must be able to modify GPUB, and one way of achieving this is to steal the fundamental ticket using malicious code executed by the prime user. However, this in turn requires that the second piece of malicious code execute at L3, making the attack cyclical. In cases such as these, a leaf node is included in the tree to show that this attack methodology has been considered but was nonetheless considered to be impossible due to this circularity.

Some scenarios constructed by combining existing scenarios were also excluded from the analysis unless there was some advantage to the attacker in combining these. For example, scenarios concerning an attacker stealing a ticket belonging to another user and then attempting to use this themselves were considered separately to scenarios where an attacker attempts to revive an expired secondary ticket that
was previously issued to them. While it is possible that an attacker could steal an expired ticket belonging to another user, this only unnecessarily complicates both attacks and therefore combinations such as this were not considered.

6.3.7 Node Attributes

In attack tree analyses generally, leaf nodes may be assigned a variety of different attributes. The tree can then be queried in order to identify the properties of the attack based on these inputs and the logical structure of the tree. For example, assigning values to leaf nodes to indicate the cost to the attacker of that component of the attack can be used to reveal the cheapest way that security may be compromised. Alternatively, the analyst could identify all attacks that may be completed for under a certain figure.

However, in this analysis, parameters such as cost or difficulty are of little relevance. The objective is to identify any and all attacks that allow violation of the specified security properties, irrespective of their cost or difficulty. If the attack is at all achievable, then it is of interest. As a result, all leaf nodes have an attribute labelled IMPOSSIBLE that is assigned a Boolean value. If the node does not clearly violate one or more of the assumptions, or there is not some other compelling reason for excluding it from the analysis, then this attribute is assigned the value FALSE and this value is also the default.

While the possibility or otherwise of specific attack scenarios is of the greatest interest in this analysis, a second attribute was also created. This attribute is labelled DIFFICULT and also stores Boolean value. The purpose of the attribute is to indicate the existence of some significant obstacle or constraint that could prevent the attacker from successfully completing this component of the attack. For example, the obstacle may reflect a scenario that could occur but cannot be controlled or otherwise influenced by the attacker. However, it can not be definitively stated that the attacker will be unable to achieve this or the constraint will not otherwise be satisfied simply by chance. As a result, a number of elaborate and relatively
unlikely attack scenarios were identified and included in the analysis. As these scenarios did not strictly violate any of the assumptions they could not be considered impossible and were therefore flagged as **Difficult**. The value of this attribute also defaulted to FALSE.

**6.3.8 Pruning Analysis and Presentation of Results**

Once the tree analysing a specific property was completed and all leaf nodes assigned appropriate values, the tree was then ‘pruned’ by removing all attack scenarios modelled as being impossible according to the logic defined by the structure of the tree. This process is performed by the SecurITree software after specifying the relevant criteria. The result is a pruned tree containing only those attacks that are actually possible. While a great deal of additional information is available in the tree, in this analysis the principal question concerns whether an attacker with a sufficiently high level of privilege is able to violate the stated security properties. Therefore, any attacks remaining after pruning are of great interest.

Results of the attack tree analysis are presented in three parts. In Chapter 7, a number of attacks identified while the attack trees were being constructed are described in detail. The identification of these attacks through the attack tree technique demonstrates the effectiveness of the attack tree approach for locating vulnerabilities in security models. Where these vulnerabilities were discovered, the model was revised to eliminate the problem and the improved version subsequently re-analysed by updating the attack tree. As a result, these vulnerabilities are not present in the pruned trees. In Chapter 8, the remainder of the results from the analysis are presented. For each tree, a high-level summary is given of the attacks modelled in the tree and then the results of pruning are presented and any remaining attacks discussed. Full details of the annotations of each node describing the attack logic and reasons for leaf node indicator values are given in the accompanying CD. Refer to Appendix 325 for further information.
6.3.9 Attack Tree Notation

In the following chapters attack trees are presented diagrammatically, which is generally superior to textual form in terms of intuitiveness when examining a tree. A brief overview of a typical attack tree notation was given previously on page 184. However, Figure 6.5 provides another example with some additional details specific to the diagrams presented from this analysis. The nodes in this tree are labelled and include the root node, which is an OR node, as is usually the case. There are two top-level nodes; one an OR node and the other an AND node. The top-level OR node includes two child OR nodes, one of which has been ‘rolled up’. This is a feature provided by the SECURITree software to hide the details of a subtree in order to conserve page and screen space when another part of the tree is being considered. In some of the diagrams presented here of complete unpruned trees, some subtrees have been rolled up where the low-level details of the attack are not central to the discussion and where it would not be feasible to present a complete tree due to the number of nodes involved and space constraints imposed by the printed page. In these cases, an indication of the complexity of the abstracted subtree will be given in terms of the number of nodes. In some diagrams of pruned trees, some subtrees are also rolled up in order to fit the diagram on a single printed page. However, in these cases full details of the low-level attack will be given and discussed as required.

The square notation for leaf nodes is also shown in the diagram. Leaf nodes have attributes that are assigned values and are shown on the right-hand side adjacent to the node. For example, the node ‘Default leaf node’ has the default values of FALSE set for both the Impossible and Difficult attributes. In the case of ‘Impossible leaf node’, the Impossible attribute is set to TRUE while Difficult remains FALSE. The converse applies for ‘Difficult leaf node’ and a node is also presented that is both Impossible and Difficult. Therefore, as is shown in the diagram, the value of the Difficult attribute is stated on the top and the Impossible attribute below this. Finally, SECURITree provides support for internal links within the tree.
This feature allows a specified subtree to exist at multiple locations within a single tree and changes to any of these instances are automatically inherited by the others. Internal links are shown with a “chain” icon immediately above the node as can be seen in the node ‘Linked leaf node’ that occurs twice within the example diagram. Note that ‘Default leaf node’ also exists twice; however, these nodes are not linked and any changes made to one do not automatically affect the other.

6.4 Conclusion

This chapter has reviewed the security analysis literature, focusing on threat-oriented methodologies that may be suitable for both identifying vulnerabilities in the newly developed model and answering the question as to whether a privileged attacker may be able to violate the claimed security properties. The attack tree technique was selected as being the most suitable for identifying whether the model is capable of resisting attack by a privileged attacker and the details of the specific methodology developed for the analysis have been described. This methodology involved developing trees by decomposing a high-level attack into logically complementary attack scenarios. Leaf nodes in these trees were then assigned values according to whether this stage in the attack could be accomplished under the stated assumptions. From this, trees were pruned to remove impossible attacks,
thereby revealing valid attacks on the studied security properties. In the next chapter, a discussion is presented of vulnerabilities discovered in the Vaults model while the attack trees were being constructed and Chapter 8 reports on the results of the analysis.
Chapter 7

Attacks Identified during Tree Construction

7.1 Introduction

Attack trees reveal information about vulnerabilities in two different ways. The results of the pruning analysis identify which of the enumerated potential attacks are actually possible for a given set of criteria, and these results are presented in the next chapter. However, the tree construction process itself also assists in elucidating security problems through the methodical formulation and organisation of attack scenarios and their properties. As hypothetical high-level attacks were identified, and these attacks refined and further details added, in some cases potentially feasible vulnerabilities became apparent. As a result, a number of potentially problematic issues in the design of the security model were identified prior to pruning, while the trees were still being constructed. These issues are discussed in this chapter. The issues discovered vary as to the danger they pose. The less serious are best described as ambiguities in the model’s original specification, which could potentially lead to undesirable behaviour in certain circumstances. Others were simply undesirable properties in certain aspects of the design, which were not sufficient in themselves to lead to a violation of the security properties but nonetheless were best eliminated.
However, in a small number of cases, complete or nearly complete attack scenarios were identified. All of these issues were subsequently rectified by altering the design of the model based upon the information revealed by the analysis. Therefore, these attacks do not appear in the pruning trees given in the next chapter. However, these now-eliminated vulnerabilities remain of interest as they demonstrate the success of the attack tree methodology in identifying what were generally extremely obscure (but nonetheless potentially dangerous) vulnerabilities resulting from subtle combinations of factors. Details of these attacks will now be discussed.

### 7.2 Integrity Checking on Encrypted Parameters

When the Vaults model was first being developed, the need to protect encrypted data against modification of the ciphertext was identified with respect to some of the tokens involved where these were specifically exposed to tampering by an attacker. Examples of these values included secondary tickets and PSA values. However, as the attack tree analysis proceeded, it was identified that the lack of authenticated encryption of the various other tokens and parameters used throughout the model created further possibilities for attackers to tamper with these pieces of ciphertext. Although no specific attacks were identified initially, it became apparent that there was a need to clarify the security properties of all the various pieces of ciphertext used in the model and to provide defence-in-depth to mitigate partial attacks that might be discovered in other components of the scheme\(^1\).

Integrity checking of encrypted values can be achieved several ways, such as through the use of a MAC applied in addition to the encryption [17]. However, while robust, use of a MAC has the disadvantages of additional computational overhead and storage of a separate key. A technically superior alternative is the use of a cipher mode that provides both authentication and encryption, such as OCB mode proposed by

\(^1\)As will be discussed in the following sections, as the analysis progressed it was subsequently discovered that the use of authenticating encryption prevented at least one complete attack against the confidentiality property under the model’s original design and also played a role in preventing another related attack.
Rogaway, Bellare & Black [197]. Such an approach imposes minimal computational overhead and does not require a separate key. At present, the principal limitation of OCB mode appears to be its patent-encumbered status. However, this is not relevant from the strictly technical perspective of the analysis. Furthermore, alternative authentication and encryption modes exist, although these are not as computationally efficient as OCB mode [265,266].

Therefore, regardless of whether a MAC or an authenticated encryption mode is used, the same required security property is achieved. The only difference with respect to the analysis is precisely what an attacker must do in order to undetectably modify an encrypted parameter in some way. In the case of a MAC, the attacker must obtain or otherwise cryptanalytically compromise the key used for authentication, whereas in the case of an authenticated encryption mode, and for the modes discussed, this will be the same as the decryption key. Therefore, for the purposes of the analysis, it is assumed that defeating the integrity check requires learning the decryption key, or some equivalent attack, and this is stated as Assumption 12 in Section 6.3.6. If a specific implementation employs a MAC rather than an authenticated encryption mode, this would make the attacker’s job more difficult as an additional key would need to be recovered.

The following values within the Vaults model are therefore both authenticated and encrypted:

- Primary tickets
- Secondary tickets
- Fundamental tickets
- User vaults
- All system vaults (GPUB, GPRIV, Escrow)
- Fundamental Vault
- PSA values
Note that the use of authenticated encryption for PSA values is not believed to
normally be strictly necessary. However, the precise effect of modification of una-
thenticated PSA ciphertext on PSI generation depends on the block cipher mode
employed. Therefore, the use of an authenticated encryption mode ensures that
the security of the PSA scheme remains intact. The original motivation for not
employing authenticated encryption for PSA values was to minimise the perform-
ance impact of these as they must be checked upon each access to every file, even
if the file is not protected. However, experimental results reported by Rogaway
et al. [197] indicate that OCB mode imposes an overhead of only 6.4% compared
with CBC encryption. Therefore it is held that the performance impact of us-
ing an authenticating encryption mode is significantly outweighed by the security
and defence-in-depth advantages. The use of authenticating encryption has there-
fore been assumed in the attack tree modelling and included as part of the Vaults
model specification.

7.3 Secondary Ticket Known Plaintext and Key Reuse
Properties

The attack tree analysis of the confidentiality property identified two properties of
the original design of secondary tickets that were undesirable and have subsequently
been corrected. Collectively these two properties represented an almost complete
attack. Originally, secondary tickets were constructed as:

\[ T_f^2 := K_f(H(K_f, \ t_1), \ t_2, \ t_E, \ VID_r), \]  \quad (7.1)

where a hash of the file protection key \( K_f \) and protection timestamp \( t_1 \) is encrypted,
using the protection key, along with the secondary ticket timestamp \( t_2 \), expiry time
\( t_E \) and recipient’s VID \( VID_r \). The revised structure for a secondary ticket involves
the use of the unique ticket key \( K_{T_f} \):
\[ K_{T_f} := H(VID_r, K_f, t_1), \quad (7.2) \]

which, as previously described in Section 4.3.3, is used to encrypt the secondary ticket constructed for each user:

\[ T_f^2 := K_{T_f}(t_2, t_E, H(VID_r, K_f, t_1)). \quad (7.3) \]

The first undesirable property found was that all of the individual plaintext components of the original ticket design from Equation 7.1 are either known, selectable or can otherwise potentially be learned by an attacker. This property therefore allows an attacker to construct the plaintext, although not the required ciphertext, of a secondary ticket they do not possess. Specifically, while the KID of the ticket, constructed as \( H(K_f, t_1) \), is not a public value, it is also not regarded as a secret. After applying the assumptions of the analysis, it was therefore identified that the possibility that an attacker might learn the value of the KID could not be definitively excluded. Appropriate timestamp and VID values could also be chosen by the attacker.

This known plaintext property by itself does not facilitate the attacker creating their own secondary tickets. However, the attack tree analysis also identified that the key \( K_f \) used to encrypt secondary ticket plaintext was also used for encrypting the data written into the protected file. If an attacker could inject their selected secondary ticket plaintext into the file protected with this key (for example, if they had write permission to the file), then this data would be transparently encrypted by the kernel. The subsequent ciphertext could then be retrieved and would be extremely close to representing a valid secondary ticket for read access\(^2\). Despite this, no means for actually exploiting this combination of undesirable properties was identified. To actually use a secondary ticket constructed in this way, the

\[^2\text{It should be noted that this attack only applies to the confidentiality property. No equivalent attack was found for the verifiable integrity property as write protection keys are not used for encrypting file contents or any other user-modifiable parameter.}\]
attacker would first have to add their VID to the list of Secondary VIDs associated with the key in GPRIV and also generate the required authentication information for the constructed ticket ciphertext. No attacks for accomplishing either of these objectives were found. Nonetheless both of these properties are clearly undesirable and the structure of secondary tickets was consequently modified to make the design more robust.

The first change involved removing the KID value from the ticket. While the KID may be difficult for an attacker to learn, the model also does not treat it as a secret and its use for associating protection keys with specific files increases its risk of being exposed. This problem was resolved by replacing the KID with a hash of the ticket recipient’s VID prepended to the KID pre-image as shown in Equation 7.3. Unlike the KID, this new value is not used for any other purpose and the attacker would need to learn the file protection key in order to calculate the value. Furthermore, placing the VID at the start of the pre-image prevents an attacker potentially using the length extension properties of common iterative hash functions [17] to extend the known KID value. The order of the parameters that make up the secondary ticket was also changed with the two timestamps (secondary ticket creation time and expiry time) repositioned to before the hash value. While the model requires that ciphertext be authenticated, even without this requirement, changes to the ciphertext for the timestamps will likely lead to errors propagating through to the second ciphertext block and corrupting the hash value when encrypted with most common block cipher modes such as cipher block chaining (CBC) mode. This placement improves the robustness of the ticket without incurring additional cost.

The final improvement was to change the key used to encrypt the secondary ticket from the file protection key \( K_f \) to a unique key specific to the ticket. This is referred to as the ticket key \( K_{T_f} \) (Equation 7.2) and is constructed in the same way as the hash value contained within the ticket itself. This design means that the value does not need to be recalculated when conducting ticket verification. Encrypting the secondary ticket with a value derived from the recipient’s VID makes the ticket
key unique for each combination of user and file protection key. This approach eliminates the property where files and secondary tickets were previously encrypted with the same key and therefore resolves the weaknesses described above. It also has the added advantage of more tightly binding a ticket to its intended recipient. Further partitioning the privileges assigned to each user increases the difficulty of attacks involving substitution of ciphertext acquired from tickets belonging to different users.

7.4 Improvements to Primary and Fundamental Ticket Structure

The attack tree modelling also identified an issue equivalent to that just described affecting primary tickets. The original structure of a primary ticket involved the protection key $K_f$, owner’s VID $VID_o$ and timestamp $t_1$, all encrypted with the protection key $K_f$:

$$T_1^1 := K_f(K_f, VID_o, t_1).$$ (7.4)

However, the analysis showed that if an attacker could steal a primary ticket issued to another user, they could learn from this the ciphertext corresponding to the encrypted protection key and timestamp. If the attacker could then subsequently write their VID into a read-protected file encrypted with the same key, it would give them the ciphertext for a complete primary ticket issued to themselves. As with the attacks on secondary tickets, this synthesised primary ticket would be valid but not usable by the attacker. However, in the case of primary tickets it was only the mandated use of authenticated encryption discussed in Section 7.2 that would have prevented an attacker from using tickets created this way\(^3\). Therefore, in light of the improvements made to secondary tickets to deal with a similar problem, the design of primary tickets was also revised to achieve equivalent robustness. The

\(^3\)In practice, the choice of cipher mode could also affect the success of this attack methodology.
new design involved replacing the VID and protection timestamp in the primary ticket with the ticket key hash used in the improved secondary ticket design. This new design, previously described in Section 4.3.2, eliminates the known plaintext property:

\[ T^1_f := K_f(K_f, H(VID_o, K_f, t_1)). \] (7.5)

The plaintext structure of fundamental tickets was similarly modified from the Fundamental Key and prime user’s VID to the hash of these two values. The original design involved encrypting the prime user’s VID \( VID_p \) and Fundamental Key \( K^F_V \) with that key:

\[ T^F_V := K^F_V(K^F_V, VID_p), \] (7.6)

while the revised structure hashes this plaintext prior to encryption:

\[ T^F_V := K^F_V(H(K^F_V, VID_p)). \] (7.7)

However, this change was done solely to give fundamental tickets the same conservative design as the other types of tickets and not because of any specific exploitable attack. In fact, the attack tree analysis indicated that the partial attacks affecting primary and secondary tickets could not be applied in the case of fundamental tickets due primarily to the different exposure of the internal parameters used.

## 7.5 Discussion of Improved Ticket Design

After the discovery of the partial weaknesses and the development of improved ticket designs as discussed in the previous sections, the relevant attack trees were revised to reflect the new structures. The results of this revision confirmed the benefits of the improved designs. For example, revisions to the subtree covering theft of
a secondary ticket indicated that the new secondary ticket structure resulted in stolen secondary tickets being essentially useless to the attacker. This result was demonstrated by the subtrees describing modification of the stolen tickets to refer to the attacker’s VID being identical to those for creating a secondary ticket from scratch. In other words, the improved design means that a stolen secondary ticket does not assist the attacker in any way.

Similarly the new design for primary tickets resulted in attacks involving modification of a primary ticket to access a different object being significantly inhibited. If the attacker opted to alter only a single part of the primary ticket ciphertext then, even ignoring the impact of ciphertext authentication, this modification would still automatically invalidate the ticket as the two parts of the ticket would no longer correspond when verification was subsequently performed. Alternatively, if the attacker attempted to modify all parts of the ticket then, once again, this attack became equivalent to creating an entire ticket from scratch.

Furthermore, under the new design, primary and secondary tickets are now encrypted with different keys and this applies even if the tickets refer to the same object. This design therefore inhibits potential attacks that attempt to convert a secondary ticket into a primary. In fact, the attack tree model again indicated the possession of a secondary ticket was of no assistance in the construction of a primary ticket for the same object, again forcing the attacker to begin their task from scratch.

7.6 Protection Instantiation Race Condition

During the course of the attack tree analysis, it was identified that it might be possible for an attacker to instantiate protection of an object while the process of protection instantiation by the legitimate owner was already in progress. If the attacker were able to make this request prior to the protection status information being updated, the request would succeed. While clearly an undesirable outcome,
beyond this, no actual security implications were identified. Specifically, even if the attacker were able to successfully mount such a race condition attack and receive a ticket for the object, this ticket would not allow them to, for example, read the contents of the file encrypted by the legitimate owner as this would involve the use of a different protection key. However, despite this conclusion, the protection instantiation protocol was modified to require the kernel to lock out any changes to the specific object’s protection status prior to protection instantiation being completed. This approach eliminates any such race conditions.

### 7.7 VID Assignment Vulnerability

In protocols involving public key cryptography, certificates are typically used to link the identity of a principal to a specific public key. As the public and private keys are intrinsically cryptographically linked, if knowledge of the private key is demonstrated through the running of the protocol, this also constitutes proof of identity. However, the original Vaults protocols for verifying a user’s identity were not completely consistent with the above approach. While the certificate associates an identity with a given public key and a cryptographic link exists between the public key, private key and VID, in certain cases the protocols did not always require the user to demonstrate knowledge of the private key. While the private key is required in order to decrypt and receive secondary tickets issued by other users, the initial protocol for assigning a VID to a process was based purely upon the user supplying the necessary passphrase to decrypt the vault associated with this VID. Also, the original specification of vault structure meant that there was not necessarily a cryptographically strong association between a VID and a given vault stored on disk. These issues and their implications were identified during the attack tree analysis.

The absence of a strong link between a VID and the user’s vault created the potential for an attacker to tamper with this association. For example, assigning the attacker’s VID to another user’s vault provides one means for performing an online
guessing attack against the user's vault encryption key. Far more seriously, the attacker could attempt to associate the user's VID with their own vault for which they already know the passphrase. If successful, this false association would allow the attacker to assume that user's VID and subsequently use tickets stolen from them. Such an attack would be particularly serious and was made possible by the assignment of a VID to a process when instantiating a new vault session being based purely on knowledge of the vault key (passphrase) and not on the private key associated with the VID. In other words, the original protocols only verified knowledge of the passphrase and did not confirm that the private key corresponding to the VID was actually present in the vault being accessed.

In light of these results from the attack tree modelling, the protocols for vault instantiation were modified. The new protocols verify that the public key on the certificate matches the VID and that the private key stored in the vault being accessed also corresponds to this public key. The use of authenticated encryption also prevents ciphertext tampering; for example, substitution of the ciphertext corresponding to another user’s private key. However, without knowledge of the appropriate vault decryption key, the attacker still cannot recover another user's private key and is ultimately prevented from assuming that user's VID.

7.8 Secondary Ticket Revocation Semantics

It was identified prior to beginning the attack tree analysis that the semantics for secondary ticket revocation were somewhat anomalous. This issue stemmed from secondary tickets not being uniquely identifiable and, as a result, revocation was previously performed by removing the recipient’s VID from a list stored in GPRIV. While separate lists were maintained for each protection key, this arrangement meant that, for a given object, it was the user’s ability to use the ticket rather than the ticket itself that was actually revoked. The effect of these semantics in practice was that, if the user had a secondary ticket that was revoked and a subsequent ticket issued for the same object, the original ticket would become usable.
again. While somewhat irregular, it was unclear prior to performing the attack tree analysis whether this property could lead to any undesirable security outcomes, as possession of two separate tickets for the same object did not appear to provide any advantage. However, while constructing the attack tree for analysing the confidentiality property, it was identified that these semantics could be problematic. Specifically, if a user were reissued a second ticket for an object after their original ticket had been revoked, the original ticket would not only become usable again but also might allow them to retain access to the object beyond the time intended if the first ticket had a later expiry timestamp than the second. In such circumstances, the owner of the object could not reasonably be expected to remember to revoke the original ticket a second time.

This problem was resolved by creating a unique identifier for secondary tickets known as the Ticket ID (TID). The TID is constructed as a hash of the KID, the recipient’s VID and the creation timestamp of the secondary ticket:

\[
TID'_r := H(KID_f, VID_r, t_2). \tag{7.8}
\]

Prior to permitting use of a secondary ticket, the kernel calculates its TID and confirms that this value is present in a list of valid TIDs stored in GPRIV. Since reissuing a secondary ticket at a later point in time results in a different TID, any revoked secondary tickets previously issued to the user for the object remain unusable. Furthermore, this approach significantly increases the difficulty of certain potential attacks on secondary tickets. For example, modification of secondary ticket ciphertext now requires that the attacker not simply produce timestamps that fall within the correct ranges, but also be the exact value of the secondary ticket creation timestamp used in the construction of the original valid TID. This new design therefore both resolves the original problem and further increases the robustness of the security model.
7.9 Overriding Fingerprint Value Conflicts

A number of attacks were also discovered when analysing the Trusted Fingerprinting mechanism within the Vaults security model. When the analysis began, the model included a fingerprint flag called {	extsc{Priority Override}} that could be set on dependency records in either the user’s vault or GPUB. The default behaviour, if the fingerprint values for the same target are different, is for verification to automatically fail and execution be denied. Previously, however, setting the {	extsc{Priority Override}} flag on only one of these records caused the conflict to be ignored and the fingerprint on the flagged record to be used for verification. The flag existed to address situations where a program or dependency with both local and global fingerprint records had been modified but the fingerprint value had not been updated in both vaults.

However, the attack tree analysis identified that the existence of the flag could allow an attacker to negate the defence-in-depth benefits that the two-tier trust architecture is intended to provide. Specifically, if an attacker were able to modify a fingerprint in one vault but not the other, they could likely also set the {	extsc{Priority Override}} flag on the modified fingerprint record. The setting of this flag would leave the attacker free to substitute malicious code for the trusted program, as the presence of the flag on the corrupted fingerprint record would cause the legitimate value to be ignored.

This attack was identified when considering attacks concerning execution of trusted programs and the problem was subsequently corrected. However, when later analysing attacks on verification of shared libraries, it was realised that this vulnerability would have been particularly dangerous with respect to these, as libraries are not assigned a trust level and effectively assume the trust level of the program with which they are linked at runtime. If the {	extsc{Priority Override}} flag were set on a maliciously modified library, this would have the potential of subverting all programs linked with the library concerned across all of their specified trust levels. The success of the attack trees in detecting this vulnerability also highlights the value of
threat-oriented analysis, as the behaviour of the flag in benign circumstances is as expected. It is only by adopting the point of view of an attacker that the potential to subvert this seemingly innocuous behaviour becomes apparent. It should also be noted that, for this effect to apply, the attacker would need to be able to modify the fingerprint record in the first place, and no valid attacks were found that allowed this. Nonetheless, this potential behaviour is clearly undesirable.

After the detection of this vulnerability, the potential benefits of the Priority Override flag were further scrutinised and determined to be minimal. For example, if an administrator were to upgrade software on the system, they would also update the fingerprint at this time. While this modification might temporarily lead to users who have created their own fingerprints finding execution of the software being denied, this outcome is arguably the correct behaviour as the user has specifically indicated that they wish to make sure that the program has not been modified prior to their executing it. On the other hand, if the user has installed a piece of software for their exclusive use, it is unlikely this target will have a global fingerprint record. As a result, and considering the potential vulnerability that it creates, the Priority Override flag was removed from the model.

7.10 Caching of Mutually Trusted Verified Shared Libraries

A potential vulnerability was discovered during the attack tree analysis concerning the caching of previously verified shared libraries that are dependencies of multiple programs. In particular, it was determined that using a file name to match cached libraries in the runtime digraph was potentially insecure. To exploit this vulnerability, the attacker would create a malicious version of an existing legitimate library with the same name and then link this library with a separate program that they also constructed. The program itself need not do anything malicious but, by creating fingerprint records for both the library and program and then executing the
program, the attacker would be able to inject the malicious library into the cache of verified objects. If a trusted program linked with the legitimate version of the library were subsequently executed, this vulnerable behaviour could have led to this code being linked with the cached and verified malicious library. To resolve this problem, the specification of the model was clarified to ensure that both fully qualified pathnames and fingerprint values of cached objects match those in the runtime digraph. In particular, if the fingerprint values match then, barring cryptographic failure, the two objects must be the same and any vulnerability is thereby avoided.

7.11 Runtime Digraphs of the Non-Natively Executable Code

A runtime digraph (RTD) is constructed for a program when its execution begins and is used to identify dependencies that require verification at runtime. In particular, a program’s RTD specifies which DATA File dependencies will be verified when opened for reading and may potentially have their fingerprints automatically updated when opened for writing. In the case of non-natively executed code, the root of the RTD will be the natively executable interpreter or runtime environment and the non-native code will then be specified as a DATA File dependency. These in turn may have their own PROGRAM (helper applications) and DATA File dependencies.

In the original specification for processing of these runtime digraphs, when searching the RTD prior to opening a file for reading, the search would be performed recursively and all dependencies in the digraph would be searched until one matching the target was found. However, the attack tree analysis revealed a serious problem with this approach. In particular, it was identified that the RTD for natively executable interpreter programs will contain all of the non-natively executable programs as DATA File dependencies. Therefore when one of these non-native programs is executing, all of the other non-native programs that share the same interpreter
will exist in the runtime digraph. This situation is illustrated in Figure 7.1 where programs P1 and P2 share the same interpreter.

This approach had a number of negative consequences of varying degrees of seriousness that were uncovered while constructing the attack trees. For example, there is potential inefficiency as unnecessary verification of objects may occur when a script such as P1 is currently executing and opens file f3 which, as a dependency of P2, requires that the fingerprint of f3 be verified even though f3 is not a dependency of P1. The dependency flags may also have a negative effect and, in particular, the REQUIRED flag could potentially cause P1 to be terminated if f3 were not available to it at runtime. This behaviour is undesirable as the policy is intended to apply to P2 rather than P1. However, there is an even more serious problem whereby, if P1 were to open f3 for writing, this situation would normally result in these modifications automatically updating the fingerprint value for f3 even though this file is not a dependency of P1. This behaviour is particularly problematic if, for example, P1 happens to execute at a lower trust level than P2 as indicated in the diagram.

These problems were resolved by clarifying the roles played by the different nodes in the RTD. The model now distinguishes between the executing process (EP), which is the native code running, and the active node (AN), which is the actual program that is notionally being executed. In the case of native code, the AN and EP will be the same. However, if a non-natively executable program is running, the
DATA FILE dependency containing this code will be the AN while the EP will be its interpreter or runtime environment. Under the modified version of the model, all files in the runtime digraph will still be verified if opened for reading. However, automatic updating of fingerprints has been significantly restricted. Updates are now normally only performed if the No AUTO UPDATE is not set and the dependency is both an immediate DATA FILE child of the active node and does not have a trust level specified in its fingerprint record. This restriction is relaxed slightly if the INSTALLER flag is set on the active node and, in this case, automatic updating applies to all descendants of the active node, again subject to the No AUTO UPDATE flag not being set. This design was subsequently modelled in the attack trees with no vulnerabilities being discovered.

7.12 Trust Level Changes and Open Objects

Another issue identified through the attack tree analysis concerns the effects of changing trust levels at runtime and the implications of trust level changes for objects that are already open. Vaults previously employed the Unix paradigm of determining whether a subject has sufficient privileges to access an object at the time of the request and, if the privileges of the subject subsequently change, this change will not affect access to objects that are already open. However, this approach created problems where the trust level of a process is lowered at runtime. This change in trust level can occur because the process voluntarily requests it, if the process accesses a dependency object with an STL lower than its ETL or if the process accesses a non-dependency object and has the REQUIRE TRUSTED DATA flag set. This conflict between the new trust level of the process and its latent privileges represents a particular threat if the program is a script interpreter and the reason for its trust level being lowered is that it has been given an untrusted script to execute. While this problem could be dealt in a number of ways, it was decided that the safest and cleanest approach from the point of view of compatibility was for the system to automatically close all open files that would not be accessible
Figure 7.2: Original and revised trust level elevation restriction algorithms

to the program at its new trust level prior to allowing execution of the process to
proceed at the new trust level. The model has now been updated to reflect this
new approach.

7.13 Process Trust Elevation Restrictions and L1 Pro-
cesses

As discussed in Section 5.6.2, it is necessary for the model to prevent malicious pro-
grams from manipulating more trusted children in order to elevate their privileges.

When this aspect of the Trusted Fingerprinting mechanism was analysed using the
attack trees, it was identified that, in certain circumstances, the original algorithm
for restricting process trust had flaws.

The original algorithm, given in Figure 7.2, would restrict the child process to the
lower of its specified trust level (STL) and its parents effective trust level (ETL),
except in the case of highly trusted L3 processes. However, while constructing the
attack tree to analyse this behaviour, it became clear that this algorithm did not
adequately confine L1 processes. An L1 program is granted limited indirect access to
designated protected objects only. The attack tree analysis showed that, unlike the
other trust levels, two different L1 programs would therefore usually have different sets of privileges. Consequently, it is not secure for an L1 program to be executed by a different L1 parent as the parent may be able to manipulate the child in order to indirectly gain privileges it is not intended to have. Similarly, if an L1 program executes a more trusted program with an STL of L2 or L3 then, under the original algorithm, the trust level of the new process would be confined to L1. However, if the nominally more privileged program had previously been configured to run at trust level L1, a variation of the previous scenario applies as the program may have a latent set of protection key bindings for objects to which it had previously been granted access. Such bindings would normally have no impact when executing at the higher trust level but, if confined to trust level L1 after being executed by an L1 parent, this situation would once again create a privilege differential between parent and child.

This vulnerability was resolved by modifying the algorithm to confine the children of L1 processes to L0 unless the fingerprints of the parent and child are the same. In this case, as the parent and child have been cryptographically verified to be the same program, execution at L1 is permitted. Figure 7.2 contains both the original and revised versions of the algorithm for comparison. Refer to Table 5.3 on page 168 for information about the specific identifiers used in the algorithm.

7.14 Summary

This chapter has summarised those weaknesses found in the Vaults model while the attack trees were being constructed and also discussed details of how the model was strengthened to eliminate these problems. These results demonstrate the value of the attack tree methodology for identifying undiscovered flaws, as the problems found were typically obscure and often relied upon subtle interactions between various elements of the model’s design. The threat-oriented nature of the analysis helped to uncover these properties where otherwise they were likely to have remained undiscovered. In particular, considering the system from the perspective
of an attacker was found to be beneficial in revealing the way seemingly innocuous behaviour could be leveraged to potentially violate the intended security properties. The attack tree analysis also indicated how the design of the model could be adapted to eliminate or mitigate the weaknesses discovered. However, merely discovering these problems does not provide substantive evidence that the refined model was now secure. Therefore, the revised specification was then re-analysed to validate the success of these changes and verified that new problems had not been introduced. As the weaknesses that were discovered have since been resolved, they do not appear in the final pruned trees. Details of the pruned trees, as well as summaries of the complete unpruned trees, are given in the next chapter.
Chapter 8

Attack Tree Results

8.1 Introduction

To analyse the Vaults security model, three attack trees were constructed. These complete trees contained a large number of hypothetical attacks devised by considering only the design of the model and not the assumptions and constraints described in Section 6.3.6. Boolean values were then assigned to the DIFFICULT and IMPRESSIBLE attributes of these nodes according to the assumptions of the analysis. After this process was complete, pruning criteria were applied to the trees. Pruning resulted in removal of all of the hypothetical attack scenarios except those identified as valid according to the logic modelled by the structure of the tree, the value of the node attributes and the specified criteria.

This chapter provides a summary of each of the unpruned trees, giving an overview of the hypothetical attacks considered. Due to the extremely large size of these complete trees, it is not practical to describe details of the low-level attacks. Instead, copies of the complete trees and node descriptions are available on the provided CD-ROM (refer to Appendix C for details). However, detailed descriptions are given here of the pruned trees and the results, in terms of the attacks that both were eliminated and remained after pruning, are also analysed and discussed extensively. These results showed that, when considered in total, for the two principal security
properties provided by the Vaults model, no valid attacks remained after pruning. This result supports the claim that the Vaults model is able to maintain the properties of confidentiality and verifiable integrity, even when faced with an attacker who has gained effective superuser privileges on the system. The Trusted Fingerprinting mechanism also proved highly resilient under these assumptions, although a small number of minor and severely constrained attacks were identified. Details of these are discussed in Section 8.4.4.

8.2 The Confidentiality Property Attack Tree

8.2.1 Overview

An attack tree was constructed to analyse whether an attacker could defeat the confidentiality property of the Vaults security model as defined in Section 6.3.2. The final complete tree consisted of a total of 160,509 nodes representing the unpruned, hypothetical attacks. The total number of attack scenarios in the unpruned tree could not be calculated by the SEcurITree software as this number apparently exceeded the maximum positive integer able to be stored in the data type used. Assuming the data type to be a 32-bit value and, given the figure reported by the program of -899,728,224, this suggests the number of attack scenarios considered to be at least 2.2 billion. However, given the number of attack scenarios reported by far smaller trees, the actual figure is likely to be in the tens of billions\(^1\). Due to this limitation in the software, the number of reported attack scenarios for large subtrees must be treated with caution and will not be stated in the following discussions unless of specific interest. A screen capture showing the attack tree properties reported by the SEcurITree software is given in Figure 8.1.

This complete tree then had pruning criteria applied to it to identify which of these hypothetical attacks were valid when considering node attribute values and tree logic. The results showed that the confidentiality property could be violated

\(^1\)For example, one tree, which contained a subset of 7752 of these nodes, was reported to have just over 2 billion attack scenarios.
only if the attacker were also able to defeat the verifiable integrity property and adequate care was not taken to avoid race conditions surrounding manual updating of trusted program fingerprints. These attacks, and the issues associated with them, are discussed at length in Sections 8.2.4 and 8.2.5.

8.2.2 Summary of Unpruned Confidentiality Attack Scenarios

Access plaintext contents without ticket

This section provides an overview of the hypothetical attack scenarios\(^2\) contained in the confidentiality attack tree, with the next section summarising the results of applying pruning criteria to this complete tree. Potential attacks on the confidentiality property were divided into two categories according to the logically complementary attack scenarios technique described in Section 6.3.3. The categories were those where the attacker utilises a usable and valid ticket to access the contents of a protected file and those where access to plaintext was gained without such a ticket. Figure 8.2 gives a top-level view of the confidentiality property attack tree with an expanded leftmost branch showing the category of attacks that do not involve the attacker acquiring a ticket.

As shown in the diagram, for these attacks, four different high-level scenarios were identified and evaluated. First, the attacker might acquire the ciphertext and then

\(^2\)Note that in the discussions of attack scenarios in this and the following sections, the hypothetical attacks are often described as if they are, in fact, possible. However, it is important to bear in mind that the validity of a given attack scenario is dependent upon it remaining in the tree after pruning criteria have been applied.

231
Figure 8.2: Top-level view of the “Defeat confidentiality property” attack tree with expanded left-most branch
obtain the protection key with which to decrypt it. Details of these attacks are abstracted in the diagram and are contained in attack tree library (M), which includes 16 nodes and 9 attack scenarios. Since the analysis assumes that the attacker has the ability to bypass the kernel and access data directly on the disk, the majority of the complexity involved in this attack is in obtaining the protection key. Second, by compromising the integrity of the kernel code, the attacker may cause the protection key or plaintext to be revealed directly. Third, scenarios were explored where the attacker sought to access the plaintext after it had been decrypted by the kernel for some legitimate purpose. The complete subtree for this category of attacks is given in the diagram. Finally, hypothetical attack scenarios were analysed where the attacker was able to trick the kernel into believing that a protected object is not protected. In this event, data subsequently written to the file by an authorised user would not be encrypted and the confidentiality of the data thereby compromised. The critical element of this attack is changing the protection status of the file and this is analysed in attack tree library (I), which contains 3,375 nodes and 6,768,882 attack scenarios.

**Obtain usable valid ticket**

If the attacker cannot otherwise obtain the plaintext, they would need a usable and valid ticket in order to gain access to the read-protected object. The subtree modelling attacks that fall into this category is given in Figure 8.3, expanded from the right-hand top-level branch in Figure 8.2. Note that the analysis distinguishes between a *valid* ticket, which is correctly formed and can be processed by the kernel, and a ticket that is actually *usable* and will result in access being granted. For example, a ticket issued to another user or one that has been issued to the attacker legitimately, but has since been revoked or expired, is valid but not usable. Therefore, while the tree extensively considers scenarios involving theft of a ticket held by an authorised user, in many cases these also include subtrees relating to the attacker satisfying the requirements for the use of such a ticket that has not been
issued to their VID.

In Figure 8.3, the leftmost branches are partially expanded and show hypothetical attacks where the ticket is created from scratch. A total of 47,372 nodes exist in this subtree. This may involve the attacker creating either a primary ticket or a secondary ticket\(^3\). As shown in the diagram, in both cases these tickets potentially may be constructed in one of two ways. For example, the attacker may build the ticket by obtaining its plaintext components and then encrypting it with the appropriate key. Alternatively, they may attempt to obtain from different sources the various pieces of ciphertext that make up the ticket and then concatenate them without learning the encryption key. Note that, although these two general approaches apply in both cases, the specifics of the attacks involved are quite different depending on whether a primary or secondary ticket is sought due to the different structures of both ticket classes.

Alternatively, if the attacker cannot create the required ticket from scratch, they must use an existing valid ticket as shown in Figure 8.4, where the leftmost branch covering scenarios where the ticket is issued to the attacker has been expanded. The first set of hypothetical scenarios analysed are those where the attacker is able to utilise a ticket previously issued after the attacker’s certificate, and therefore their vault ID (VID) has been revoked. If the user’s VID is revoked, then all tickets associated with it should become unusable; continuing to be able to use these is a technical violation of the security property. A total of 14,500 nodes are contained in this rolled up subtree.

The next branch analyses where a ticket has been issued to an attacker but was actually intended for another user. This covers a number of scenarios but focuses on the attacker attempting to obtain a usable primary ticket for an object that is already protected. It also covers possible means for tricking the owner of a protected object into issuing a secondary ticket for the attacker’s identity rather than that of

\(^3\)Since possession of a primary ticket confers greater privileges, this is likely to be preferred by the attacker if achievable. However, possession of a valid and usable ticket of either class is sufficient to violate the confidentiality property.
Figure 8.3: Subtree of the “Defeat confidentiality property” attack tree showing attacks where the attacker obtains a usable valid ticket
Figure 8.4: Subtree of the “Defeat confidentiality property” attack tree showing attacks where the attacker uses an existing valid ticket
some legitimate user. This subtree includes 6,693 nodes.

The third branch of the subtree contains 20,004 nodes and examines the various ways an attacker may attempt to modify an existing ticket to violate the confidentiality property. Scenarios modelled include how the attacker might modify their own primary ticket to make this refer to a different object that they do not own, modifying a secondary ticket to facilitate access to a different object, and converting a secondary ticket into a primary. Note that this branch does not consider attacks associated with modification of stolen tickets as these are addressed in a different subtree (see Figure 8.5).

Attacks that involve reviving expired and revoked secondary tickets are also analysed in Figure 8.4. Reviving an expired secondary ticket requires modifying the expiry timestamp stored encrypted within this token and 6,691 nodes are contained in this subtree. Alternatively, revoked tickets may be used again if the required ticket ID (TID) can be added to the list associated with the protection key in GPRIV or if the VID and timestamp of the ticket can be modified to match an existing valid TID. This branch consists of 24,118 nodes. Note that in all scenarios involving both expired and revoked secondary tickets, it is assumed that the attacker already has such a ticket and the ticket has been issued to them.

Finally, Figure 8.5 shows the expanded right-most branch of Figure 8.4 and covers potential scenarios where the ticket used by the attacker was not issued to them. Here a distinction is drawn between scenarios where the attacker utilises another user’s ticket to access the protected data without actually taking possession of this ticket, and those scenarios where the attacker takes possession of the ticket by importing it into their vault. In the former case, the attacker may gain use of the ticket through the use of mcode and three subtrees have been developed, one for each of the relevant trust levels (L3, L2 and L1). These subtrees consist of 3,265, 2,179 and 2,199 nodes, respectively. Alternatively, the attacker may attempt to gain control over the victim user’s vault and the question here is whether the attacker can learn that user’s vault key and thereby effectively assume their identity. This
Figure 8.5: Subtree of the “Defeat confidentiality property” attack tree showing attacks where the attacker uses a ticket not issued to them
is evaluated in attack tree library (N) which consists of 11 nodes.

The right-hand branch shown in Figure 8.5 examines how the attacker may take possession of a ticket to facilitate access to the target object. One means for achieving this is by stealing the ticket from an authorised user and subtrees are included considering scenarios for both primary and secondary tickets (10,820 and 17,397 nodes, respectively). However, acquiring the ticket is not enough and these subtrees also include scenarios considering how the attacker can actually use the ticket by ensuring that the VID of their process matches that specified by the ticket. This can be achieved either by changing the VID specified in the ticket or changing the VID of the process. Alternatively, tickets may be acquired by the attacker through the escrow mechanism. This can be achieved by gaining possession of the fundamental ticket for the Escrow Vault, which is normally used by the prime user to obtain a secondary ticket to allow access to a protected object or by attempting to directly recover the primary tickets stored in the Escrow Vault. Attacks relating to this scenario are contained in a library (T), which consists of 1,809 nodes.

8.2.3 Overview of Pruning Results

Pruning was applied to the complete tree just described to remove impossible scenarios according to the assumptions described in Section 6.3.6. After removing all nodes evaluated as IMPOSSIBLE, the final pruned tree had 1,057 nodes with 110 different attack scenarios with 99.54% of nodes being eliminated. For the minimum possible number of attack scenarios in the original tree, this means at least 99.99999% were eliminated. No means were identified enabling attackers to create their own tickets or modify existing tickets for unintended purposes. Expired and revoked secondary tickets also remain unusable by an attacker and no means for taking possession of another user’s vault was identified. Finally, the remaining pruned tree also evaluated as DIFFICULT, indicating that the ability of an attacker to successfully complete the attacks it described is constrained in some way beyond the attacker’s control as described in Section 6.3.7. In particular, it was found
that defeating the confidentiality property required specific insecure user behaviour that facilitated the attacker’s subversion of designated trusted code. Even more significantly, the attack tree modelling demonstrated that attacks on the confidentiality property are dependent upon successfully violating the verifiable integrity property. While the initial attack tree model for the confidentiality property, as described here, assumed that this could be achieved, the results of the attack tree analysis of the verifiable integrity property indicate that it cannot. Consequently, the 110 attack scenarios that remained after pruning are only valid if these constraints are ignored. Further details of this are discussed in Sections 8.2.6 and 8.3.3.

Nonetheless, the non-empty pruned tree is an interesting result and is indicative of potential weaknesses in the confidentiality property. The ability of the attack tree methodology to model the preconditions and structure of attacks therefore allows for the nature of the attacks identified to be further explored and this proved to be informative.

The pruned tree (Figure 8.6) showed that, within the constraints indicated previously, the confidentiality property may be violated by an attacker without obtaining a ticket by removing the read protection designation from a file resulting in subsequent legitimate writes to this file not being encrypted (6 scenarios). Alternatively, the attacker could violate the property by: using a valid ticket after their certificate had been revoked (4 scenarios); tricking a user into issuing them with a ticket using a false certificate (6 scenarios); using malicious code to indirectly obtain access to encrypted data (54 scenarios); using a stolen primary or secondary ticket (36 scenarios); and obtaining a ticket via the escrow mechanism (4 scenarios). The underlying causes of these theoretical attacks will now be discussed.

8.2.4 The Manual Update Race Condition Attack

An advantage of the attack tree methodology is that its hierarchical nature facilitates exploration of the prerequisites of a given attack and the reasons why the
Figure 8.6: Top-level view of the pruned “Defeat confidentiality property” attack tree
attack occurs. Drilling down into the pruned attack tree showed that the same subtree occurred in all of the remaining attack scenarios. In particular, two specific leaf nodes were identified as preconditions for all of the 110 attack scenarios. Examining the tree structure indicated that removal of either one of these leaf nodes resulted in the entire tree becoming IMPOSSIBLE. This was tested experimentally by setting the values of each of these nodes in turn to IMPOSSIBLE and then recalculating the pruned tree. In both cases an empty tree resulted, indicating no valid attacks.

The cause of these attacks is the ability of malicious code to execute with an elevated trust level of either L2 or L3. Subtrees representing this type of attack exist in all branches of the pruned tree. Twenty of these subtrees involve malicious code being able to execute at L3 and the relevant subtree is shown in Figure 8.7. Only in two cases did malicious code executing at L2 lead to a successful attack and Figure 8.8 gives the subtree that applies in this case. Note that two instances of the subtree from Figure 8.7 were removed from Figure 8.8 for purposes of clarity. Also note that the two subtrees are extremely similar, with the only non-superficial difference being that, in the case of L2 programs, the modification of the hash value can occur in either the user’s vault or the global public vault. Significantly though, in both subtrees the same leaf nodes and tree structures are responsible for making the attack possible. In particular, the subtree present in both Figures 8.7 and 8.8, beginning from the AND node “Perform manual update race condition attack”, is pivotal in the success of the overall attack. This subtree is given in Figure 8.9, and Figure 8.10 provides a high-level view with all instances of the subtree in the overall pruned tree circled. While the labels of the individual nodes in Figure 8.10 are not readable due to the scaling necessary to present the entire tree on a single page, the diagram highlights the role that these nodes play in violating the confidentiality property. In all the scenarios, this set of nodes occurs at the leaf-end of the subtree and, as discussed before, changing the value of either of the leaf nodes to IMPOSSIBLE prevents all of the attack scenarios identified.

The first of the leaf nodes in Figure 8.9 represents what has been termed the ‘man-
Figure 8.7: Pruned subtree showing attacks where malware executed at L3
Figure 8.8: Pruned subtree showing attacks where malware executed at L2 program.
Figure 8.9: ‘Perform manual update race condition attack’ subtree

manual update race condition attack’ (MURCA). This attack applies when a trusted program on the system has been legitimately changed; for example, being upgraded to a new version. Such a change requires that its fingerprint must be updated. The attack tree analysis identified that any time elapsing between the modification of the program and the recalculation of its trusted fingerprint potentially allows an attacker to substitute malicious code for the trusted program. This requires the attacker to be able to bypass the system kernel and directly access the storage device. However, this ability is specifically allowed in the analysis under Assumption 5, as stated in Section 6.3.6. As a result of these modifications, when the fingerprint is subsequently updated, it will now reflect the attacker’s code rather than the trusted program. When the attacker’s code is then executed, it will run with the trust level of the trusted program it is impersonating. If this attack can be performed for a fingerprint update of an L3 program, then the attacker gains the ability to arbitrarily access the contents of the vaults belonging to any users who execute this code, including the prime user. An equivalent attack applies for L2 programs. However, this attack is less severe as these programs cannot directly access the contents of user vaults. Nonetheless it still permits violation of the confidentiality property.

The leaf node representing the MURCA can be found in Figure 8.9. This node is flagged as being DIFFICULT (but not IMPOSSIBLE), as the attacker’s ability to successfully perform it is constrained by the requirement that an L3 program is
Figure 8.10: Pruned “Defeat confidentiality property” attack tree with MURCA subtrees highlighted
legitimately updated in an insecure way that allows for substitution to occur and the attacker is unlikely to be able to influence the occurrence of this event. Note that, if the node becomes Impossible, then so do all four AND nodes above it in the subtrees in Figures 8.7 and 8.8. As noted above, this causes all identified attacks against the model’s confidentiality to become impossible and highlights their dependence upon this specific problem.

However, the MURCA alone is not sufficient for the attacker to substitute malicious code for a legitimate program; violating the verifiable integrity property of the model is also required. This is because programs with trust levels of L2 or L3 are considered to be of sufficient importance that they will be cryptographically write protected, which is stated in the analysis under Assumption 3. Therefore the failure to successfully verify the object’s MAC will reveal the illicit modification, even if its fingerprint does not. Defeating the verifiable integrity property of the Vault’s model is therefore the second prerequisite in order to violate the confidentiality property, and this is indicated in the second leaf node in Figure 8.9.

When modelling the confidentiality property, nodes relating to the defeat of the verifiable integrity of the system were treated as being possible. This was partly because an attack tree for the verifiable integrity property had not yet been constructed, but principally to highlight any interdependence between the confidentiality and verifiable integrity properties.

Nodes relating to the verifiable integrity property appear three times in the subtree given in Figure 8.7 and four times in Figure 8.8. These nodes appear multiple times to reflect the need to defeat the verifiable integrity property in relation to different requirements of the attack. However, these requirements can all be satisfied with a single successful illicit modification. The uppermost occurrence (two levels below the root on the left-hand side of the tree) relates to an attacker modifying a legitimate trusted program via the kernel interface. This action would require a violation of only one part of the verifiable integrity property and is therefore prefixed with the label PROP. This node has no impact on security, as that branch of the subtree’s
root AND node can be completed via an alternative means as modelled by the leaf node’s parent OR node. However, if either of the other two occurrences (eight and two levels down below the root on the right-hand side) become IMPOSSIBLE, the entire subtree will also evaluate in this way, again eliminating all attacks in the pruned tree.

8.2.5 Specific Attack Instances

Plaintext Data Written to Encrypted File

The attack tree analysis did not identify any way in which an attacker could directly cause encrypted data to be decrypted for them without possession of a valid and usable ticket. However, one attack methodology was found for accessing plaintext without a ticket and this involved removing the read-protected designation from an encrypted file. This was a somewhat limited attack as it would not cause the file to be decrypted and would prevent legitimate users reading the encrypted data already in the file⁴. However, data subsequently written to the file by legitimate users would no longer be transparently encrypted, which is sufficient to violate the confidentiality property.

The most difficult aspect of this attack is the removal of the read protection designation from the file. Although no means was identified for defeating the cryptographic mechanism for verifying file protection status, the analysis identified an attack that used malicious code to remove the designation. A high-level view of the attack is given in the leftmost branch of Figure 8.6. This branch shows that changing the protection status of a file requires a process to have an L3 trust level, which could ultimately be achieved by the MURCA. The MURCA requires that the malicious code be evaluated by the kernel as an L3 program and scenarios for achieving this are modelled in library (F). The simplest means involves the subtree shown previously in Figure 8.7.

⁴A potential race condition exists whereby, if an attacker could remove the read-protected status from a file after the contents of this file had been buffered to memory but prior to the contents being rewritten, then this attack could be extended to decrypt existing plaintext.
Figure 8.11: Pruned subtree showing attacks for malicious code executing at trust level L3
An attack was also identified involving the creation of the dependency record for the malicious code by the attacker stealing and using a fundamental ticket. Ultimately though, this attack also depends upon the MURCA. Further details of the latter attack are given in Figure 8.11, which follows on from the left-most branch of Figure 8.6, and shows how the fundamental ticket can be stolen using malicious code and subsequently used by the attacker assuming the prime user’s VID. In order to assume the prime user’s VID, the attacker must either take control of the prime user’s vault or associate the prime user’s VID with the attacker’s vault, which is shown in Figure 8.12. No possible scenarios were identified for the former attack but the latter could be achieved by learning the prime user’s private key. This in turn was achieved using malicious code executing with trust level L3 to access the prime user’s vault and represents another application of the MURCA.

**Ticket Used with Revoked Certificate**

Once a user’s certificate has been revoked, the kernel should not permit the use of any file access tickets assigned to that user; to do otherwise is considered a violation of the security properties of the model. Certificate revocation is implemented through a certificate revocation list (CRL) stored in GPUB and only the prime user is able to modify the contents of this vault. However, if an attacker is able to steal the fundamental ticket permitting modification of GPUB, they can remove their VID from the CRL. The attack for the theft of the fundamental ticket from the prime user’s vault is the same as that described in the previous section and, once again, the attack is enabled by the MURCA by allowing malicious code to execute with an L3 trust level and therefore to gain direct access to users’ vaults. A high-level view of the attack is given in Figure 8.6 which extends down to the beginning of attack tree library (B). Further details continuing on from (B) are contained in Figures 8.11 and 8.12.
Figure 8.12: Pruned subtree showing attacks for subverting VID assignment
Secondary Ticket Mistakenly Issued

Figure 8.13 shows an attack where the user is tricked into issuing a secondary ticket for an object. This is ultimately achieved by the attacker creating a fraudulent certificate for themselves containing the intended recipient’s name and signing it with the prime user’s private key stolen from their vault. Once again, this attack is enabled by the presence of the subtree from Figure 8.7 containing the MURCA. The central AND node in Figure 8.13 highlights the highly constrained nature of this attack and three of its four children evaluate as DIFFICULT.

To be successful, the user issuing the ticket must select the attacker’s certificate over that of the intended recipient and must not already have a cached copy of the intended recipient’s certificate from a previous ticket exchange. Note that as the attacker and intended recipient’s certificates will have the same name, the issuing user may notice this if, for example, the names are adjacent in an alphabetically sorted list presented to them. This duplication would likely arouse their suspicion and perhaps cause the attack to fail in practice. Consequently, the attacker might construct the certificate’s **Identifier** field so as to subvert the normal alphabetical sorting; for example, by prepending non-printable characters. The practicality and effectiveness of such an approach is implementation-dependent but was regarded as possible for the purposes of the analysis. However, most significantly, the user must be able to verify the prime user’s signature on the attacker’s fake certificate. It was found that the attacker could create this signature by learning the prime user’s private key, having stolen the key from the prime user’s vault using malicious code executing at trust level L3 via the MURCA.

Leak Contents using Malcode

Figure 8.6 contains a high-level attack involving the use of malicious code to leak the contents of read-protected objects and this attack is expanded further in Figure 8.14. As the diagrams show, the malicious code that leaks the plaintext may be executing at either L3, L2 or L1. In the case of L3 malicious code, this attack is
Figure 8.13: Pruned subtree for when an attacker has a ticket mistakenly issued to them
Figure 8.14: Pruned subtree showing use of malcode to leak read-protected file contents

contained in attack tree library (C) and was illustrated in Figures 8.11 and 8.12. Attack scenarios involving L2 malicious code are structurally almost identical to those utilising code with trust level L3. These are included largely for completeness as the same underlying attack in the form of the MURCA applies to both and it seems unlikely in practice that an attacker would have their code execute at a lower trust level given the choice. This applies even more with respect to L1 code and the attack is further complicated by the need to have the protection key specifically bound to the L1 malicious program. Note that, as shown in Figure 8.14, the only attacks identified for instantiating this binding were if the user chose to, or was otherwise tricked into, this situation. These scenarios seem relatively unlikely but could not be definitively excluded under the assumptions of the analysis.
Figure 8.15: Pruned subtree for stealing a usable primary ticket

Steal Usable Ticket

The subtree for stealing a usable primary ticket is shown in Figure 8.15, which is the continuation of a high-level tree from Figure 8.6. This attack requires the attacker to both steal the ticket and then be able to use it by ensuring that the VID of the ticket matches that of the process making the access request. The left branch of the tree shows how the attacker may steal the primary ticket from the user’s vault by having the user execute malicious code. However, in order for the code to steal the ticket, it must have direct access to the user’s vault which requires that it executes with a trust level of L3 (Figure 8.11). As with the previous attack scenarios, this could be achieved through the MURCA.

In order to use the primary ticket once it has been acquired, the attacker must ensure that the VID specified by the ticket matches that of the process attempting access. No means for altering the VID specified by a ticket was found. However, the attacker may assume the user’s VID by using L3 malicious code to steal the
prime user’s private key (Figure 8.12). This attack again utilises the MURCA in order to have the malicious code execute with this high trust level.

The subtree for stealing a usable secondary ticket is shown in Figure 8.16. This is very similar to that for stealing a primary ticket except that secondary tickets can also potentially be stolen from GPUB. This occurs when a secondary ticket is distributed to a recipient by encrypting it with their public key and storing the result in GPUB. If the attacker is able to retrieve the recipient’s private key from their vault via the MURCA, then they may retrieve and decrypt the secondary ticket while it is awaiting collection. Note that this latter attack is no more difficult than the former as the recipient’s private key must be acquired in order to use the ticket in both cases. Use of the secondary ticket is therefore similar to that for primary tickets except that the attacker must know the VID to which the ticket was issued. However, this is unlikely to be a significant impediment.

Obtain Ticket through Escrow Mechanism

An equivalent attack to that just described allows an attacker to obtain a secondary ticket through the escrow mechanism, which is shown in Figure 8.17. Normally the prime user may use their possession of the fundamental ticket for the Escrow Vault to obtain a secondary ticket for a protected object. However, the attacker may steal and use this fundamental ticket in the same way as previously described using the MURCA.

8.2.6 Discussion

As indicated, all the attack scenarios in the pruned tree contain at least one instance of the subtree from Figure 8.9 and depend upon the attacker being able to violate the verifiable integrity property and perform the MURCA. The results confirm that removing either leaf node leads to the entire tree being evaluated as IMPOSSIBLE.

Significantly, therefore, the results show that the security of the confidentiality property of the Vaults model is not dependent upon the verifiable integrity property.
Figure 8.16: Pruned subtree for stealing a usable secondary ticket

Defeating the verifiable integrity property does not, by itself, allow the attacker to defeat the confidentiality property as successful completion of the MURCA is still required. However, defeating the verifiable integrity property remains a prerequisite for bypassing confidentiality — even if the MURCA is possible. That the interdependence of the two properties is only in regards to the attacks that may be performed on them, and not with respect to their security, is therefore an ideal result.

The existence and occurrence of the MURCA is therefore of particular importance. The attack highlights the importance of code integrity and reaffirms that, if the integrity of highly trusted code can be violated, even strong security models are very likely to fail. On this basis, it is positive that the failure was limited to only six
Figure 8.17: Pruned subtree for obtaining a ticket through the escrow mechanism
high-level attack outcomes and demonstrates resilience in extreme circumstances.

The MURCA is also interesting as it is not an attack on the model itself but rather on the user’s actions in support of the model. It is therefore more of a vulnerability in a practical aspect of administration rather than a theoretical flaw in the model itself. Consequently, the vulnerability can be avoided through procedural changes and does not necessarily require alteration of the underlying model. For example, precalculating fingerprints for updated L3 programs obtained from a trusted source prior to writing these programs to disk avoids the issue entirely. Indeed, one of the most valuable outcomes of this part of the attack tree analysis is to highlight the need for greater care when updating the fingerprints of trusted programs; the analysis has shed light on the extent of the impact of not doing so.

Another approach is to use an L3 program to act as a trusted installer. This program would obtain the new version of the object to be updated from a trusted source (for example, read-only optical media), calculate the new fingerprint value and update this in GPUB prior to installing the new program. This approach eliminates the need to perform manual updates and therefore prevents the MURCA. To facilitate this, the Installer fingerprint flag (Section 5.5.4) was introduced. This flag allows a program to have the fingerprint values of all of its dependencies to be automatically updated and therefore avoids the MURCA entirely. Repeating the attack tree analysis after adjusting the assumptions to require that trusted installers are used to upgrade all L3 software, prevents the MURCA from happening and thereby eliminates all attacks on the confidentiality property, irrespective of whether the attacker can violate the verifiable integrity property. Indeed, although not explicitly stated in the analysis assumptions made at the outset and given in Section 6.3.6, the spirit of Assumption 3 (relating to the security of system configuration) and Assumption 11 (concerning secure administrative practices) are consistent with the use of trusted installers to avoid race condition-related problems. However, the extent of these problems was not anticipated when the assumptions were laid down and therefore this requirement was not explicitly included. Consequently, it is
Figure 8.18: Properties of the “Defeat verifiable integrity property” attack tree appropriate that the attack tree analysis should identify the MURCA as this result serves to highlight the need for users and administrators to protect systems against this threat. While mandating the use of trusted installers would avoid the problem, such an approach is arguably too prescriptive and the flexibility to allow manual updating, where performed securely, is likely to be appropriate in practice for many systems.

8.3 The Verifiable Integrity Property Attack Tree

8.3.1 Overview

After the completion of the attack tree to evaluate the security of the confidentiality property, a second tree was constructed to analyse whether an attacker could defeat the verifiable integrity property. When complete, this second tree consisted of a total of 163,816 nodes. As with the confidentiality tree, the size of the tree meant that the total number of attack scenarios could not be calculated by the ScurITree software as the result exceeded the maximum storable positive integer. Hence a value of -1,434,440,612 was reported, as indicated in Figure 8.18.

As with the confidentiality tree, pruning criteria were then applied to remove all impossible attacks from the set of hypothetical attacks represented by the complete tree. Again, the subtree relating to the MURCA also existed in the unpruned tree. However, unlike with the confidentiality tree, when considering attacks on the verifiable integrity the MURCA was no longer possible. This was because
the MURCA subtree from Figure 8.9 requires violation of the verifiable integrity property to complete successfully, leading to a set of circular attack requirements. Consequently, pruning led to all of the nodes from the complete tree being removed and the resulting empty tree indicates that no valid attacks were identified on the verifiable integrity property. As discussed in Section 8.2.6, as defeat of the confidentiality property was found to depend upon successful violation of verifiable integrity, the absence of attacks in this latter security property means that no valid attacks were identified in either of the two main security properties of the Vaults model. Details of the analysis will now be discussed.

8.3.2 Summary of Unpruned Verifiable Integrity Attack Scenarios

Figure 8.19 gives a high-level view of the complete, unpruned tree and shows that the hypothetical attacks it describes fall into one of two categories. First, there are those attacks considered in the left-most branch of the tree where the attacker makes an illicit modification through a request sent via the kernel interface. This requires the attacker to take some action to ensure that the request succeeds despite being in contravention of the verifiable integrity property. The second category of attacks, described in the right-hand branch of the tree, apply where an attacker with sufficient privileges bypasses the kernel interface and directly modifies objects on the disk, as permitted by Assumption 5 (Section 6.3.6). However, subsequent to this, the attacker must take further action to prevent the kernel from identifying that this illicit change has occurred.

Attacks Via the Kernel Interface

In Figure 8.19, the left-hand branch of the tree is expanded showing details of the attacks that take place via the kernel interface. The tree shows that the attacker can potentially perform modifications via the kernel interface as a result of three different scenarios. The first of these is due to kernel error, which is specifically excluded by Assumption 3. Alternatively, the attacker can make a modification to
Figure 8.19: Top-level view of the “Defeat verifiable integrity property” attack tree with expanded left-most branch
the object if they have access to a usable and valid ticket. The methodologies for this subset of attacks are almost identical to the equivalent attacks considered in the attack tree analysing the confidentiality property. The tree structure shows that these attacks involving the use of a ticket may be as a result of the attacker either creating the ticket from scratch or using an existing valid ticket. Where the ticket is created from scratch, it may be either primary or secondary, and scenarios involving the construction of the ticket from both plaintext and ciphertext are considered in the tree. A total of 19,898 nodes are contained in the subtree for creating a primary ticket, 13,244 concerning the creation of a primary ticket from plaintext and 6,654 for the creation of the ticket from ciphertext components. By comparison, the subtree for creating a secondary ticket contains 27,440 nodes, 24,087 of these for the creation of the secondary ticket from plaintext and 3,353 for ciphertext attacks. Alternatively, when the attacker uses an existing valid ticket, this ticket may be one that has been issued either to the attacker or to some other party. Scenarios where a ticket has been issued to an attacker include where the attacker attempts to revive an expired or revoked secondary ticket (6,691 and 24,118 nodes, respectively), where a ticket is mistakenly issued to the attacker (6,693 nodes), and where the attacker attempts to use a ticket after their certificate has previously been revoked (14,473 nodes). An attacker may also modify a primary or secondary ticket to refer to a different object (6,618 and 10,050 nodes respectively) or modify a secondary ticket to attempt to convert it into a primary ticket (3,358 nodes).

The other branch of the subtree considering use of existing valid tickets is where the ticket was not issued to the attacker. This can be achieved by gaining use of an authorised user’s ticket through malicious code (7,644 nodes) or by gaining access to the user’s vault (15 nodes). Alternatively, the attacker may take possession of a ticket by placing it in their vault, either by obtaining the ticket through the escrow mechanism (1,809 nodes) or by stealing it from another user (10,793 nodes for the theft of a primary ticket and 17,397 nodes for a secondary). Finally, another high-level attack on the verifiable integrity property via the kernel interface exists where the attacker can alter the file’s apparent protection status such that
the kernel believes the file is not protected and will therefore permit write access.
Attack scenarios for this are addressed in the integrity version of the subtree library
‘(I)Remove protection from file’ and this comprises 3,375 nodes.

**Attacks Bypassing the Kernel Interface**

Figure 8.20 shows an expanded version of the right-hand branch of the top-level
tree rolled up in Figure 8.19, and describes scenarios where the attacker illicitly
modifies an object through bypassing the kernel interface. When the attacker seeks
to accomplish this attack, they must first be able to make the modification and,
second take some action to cause the kernel to not identify the modification on
subsequent access to the affected object. This constraint is reflected in Figure 8.20
using an AND node with two corresponding children at the root of this subtree. As
the analysis assumes that the attacker will have sufficient privileges on the system to
potentially bypass the kernel and directly access the data on the disk, the principal
question is whether the attacker can prevent this modification from being detected.
The two scenarios where this may apply are when the kernel checks to see if the file
has been illicitly modified and still fails to detect the change, and when the attacker
can cause the kernel to not perform this check.

In the case of the former scenario, the result may simply be due to kernel error or
because the attacker has been able to cause the hashes to match. This latter cir-
cumstance would occur in the event of a cryptographic failure in the one-way hash
function causing a collision (excluded by Assumption 7 requiring secure crypto-
graphy) or if the MAC value stored in GPRIV were to be modified. The modification
of this value could be performed by the attacker recalculating and updating the
MAC. However, being able to update the MAC requires learning both the file pro-
tection key (modelled in subtree library (M) containing 16 nodes) and also gaining
access to the value in GPRIV (subtree library (G) containing 14 nodes). Alterna-
tively, the MAC could be updated either by another user modifying the file or by
the kernel acting independently. In the case of a user modifying the file, the model
Figure 8.20: Subtree of the “Defeat verifiable integrity property” attack tree showing attacks where the modification bypasses the kernel interface requires that the kernel verify the MAC value of write-protected files before permitting write access. For this not to occur, either the kernel has behaved incorrectly (excluded by Assumption 3) or the attack has become circular. The kernel acting independently to incorrectly modify the MAC value also violates Assumption 3 that requires the kernel to implement the model correctly. Both of these scenarios are therefore excluded by assumption.

Excluding these scenarios leaves the right-hand branch of the tree from Figure 8.20 addressing hypothetical scenarios where the attacker takes some action to prevent the kernel from checking for illicit modifications. This scenario will apply if the attacker is able to cause the kernel to no longer be able to identify that the file is designated as write protected, which is addressed in the subtree library (I) (3,375 nodes). Although not visible in Figure 8.20 due to being abstracted within the library, this subtree library considers attacks targeting the protection
status verification scheme employed by Vaults and also reflects the importance of verifying the protection status of an object in a security model intended to be secure against privileged attackers.

Alternatively, the kernel may fail to check whether a write-protected file has been modified if the kernel code itself has been compromised, either by the attacker modifying the code in memory or on disk. Modification of kernel code in memory is excluded by Assumption 6, which concerns the isolation of the kernel and hardware enforcement of secure memory. Modification of the code on disk is also not possible, as booting of code modified directly by the attacker will fail due to the requirement for a secure bootstrap sequence (Assumption 10). Any attempt to modify the code by the kernel interface represents a circular attack. Finally, the kernel could fail to perform the check due to internal error but this is also excluded under Assumption 3.

### 8.3.3 Pruning Results

It was initially anticipated that the MURCA, identified when analysing the confidentiality property, would also be problematic when modelling attacks on the verifiable integrity property. However, the pruning results of the confidentiality property tree identified that MURCA-related scenarios were dependent upon the attacker being able to defeat the verifiable integrity property. Therefore, as the prospective attacks performed via the kernel interface using a ticket are largely the same for the verifiable integrity property as for the confidentiality property, this leads to circularity whereby one of the requirements for an attack on the verifiable integrity property is defeating the same. This is highlighted in Figure 8.9 from the confidentiality analysis that contains nodes representing both the MURCA and defeat of the verifiable integrity property (prefixed with the label TREE) with a parent AND node. This tree structure models that the attacker must overcome the verifiable integrity property in order to successfully exploit the MURCA by replacing trusted code on the disk with malicious code.
In the analysis of the confidentiality property, nodes referencing defeat of the verifiable integrity property were considered to be possible. One reason for this was that, as the confidentiality property was modelled first, the result of the verifiable integrity property tree was not known. More importantly, setting this node’s IMPOSSIBLE attribute to FALSE also ensured that subtrees describing valid attacks were not prematurely eliminated, and assisted with identifying dependency relationships between the two trees.

However, in the context of attacking the verifiable integrity property, the attack requirements for the nodes referencing this same property became circular and consequently the MURCA-based scenarios became no longer possible. That is, if an attack scenario for defeating the verifiable integrity property contains the node that requires defeat of this same property, the attack has become circular and therefore impossible. Consequently, all 5,107 occurrences of these nodes in the verifiable integrity tree and its subtree library versions were changed to IMPOSSIBLE.

Therefore, although the underlying MURCA still exists in the verifiable integrity tree, the attack is not exploitable in this new context. The single underlying problem that caused all the attacks identified on the confidentiality property in the pruned tree therefore does not apply when analysing the integrity property.

As a result, pruning of the verifiable integrity tree to remove impossible nodes leads to removal of all subtrees relating to the MURCA. Since no additional attacks specific to the verifiable integrity property were identified, an empty pruning tree containing no nodes resulted. This result indicates that no means for compromising the verifiable integrity of the model were identified. However, significantly, the absence of attacks on the verifiable integrity property also means that the security of the confidentiality property remains intact, as all of the attacks on confidentiality also depend upon successful exploitation of the MURCA and thus violation of the verifiable integrity. Therefore no successful attacks were identified on the two major security properties of the model.

Beyond this, it should also be noted that a large number of nodes exist in both the
confidentiality and verifiable integrity property trees that reference only a specific subset of attacks on the verifiable integrity property. The label of these nodes is prefixed with PROP. In the confidentiality property tree, these nodes are designated as being possible in the same way as those that refer to the full tree. However, in the verifiable integrity property tree, some of these nodes are logically impossible where they occur in the subtree modelling attacks on the same subset of the property. Despite this impossibility, to avoid the complexity involved in making multiple versions of these nodes, the same node for each property subset was used in all subtrees of the verifiable integrity property tree and was treated as being possible. In total, this affected 1,276 nodes. Note that including these nodes does not have any deleterious impact on the outcome of the pruning process as this approach could only result in the inclusion of invalid attacks rather than excluding potentially legitimate ones. The lack of impact of including these logically impossible nodes on the pruned tree results for the verifiable integrity property therefore further suggests a significant degree of defence-in-depth in relation to the design of the overall security model.

8.4 Trusted Fingerprinting Attack Tree

8.4.1 Overview

The Trusted Fingerprinting feature of the Vaults security model plays a significant role in supporting both the confidentiality and verifiable integrity properties. Consequently, some of the aspects of Trusted Fingerprinting have been evaluated in the previous two attack trees. However, as the attacks considered in these other trees were restricted to those that specifically supported violation of the security property being considered, it was identified that certain aspects of the Trusted Fingerprinting feature had not been fully studied. In particular, issues relating to the user’s expectations in relation to the privileges held by certain programs had not been considered in detail, and the behaviour of the Trusted Fingerprinting mechanism in
Figure 8.21: Properties of the “Defeat Trusted Fingerprinting” attack tree

scenarios not specifically pertaining to the violation of the other two security properties required further evaluation. As a result, a third attack tree was constructed to evaluate the security properties of Trusted Fingerprinting, namely:

- Trusted code and its dependencies accessed via a specified filesystem path are cryptographically verified on execution or access (as appropriate) to confirm that they have not been modified.

- Privileges of trusted code determining access to cryptographically protected objects are assigned based on whether this code has been successfully verified and its determined trust level. A corollary of this property is that unverified code is considered untrusted, has no access to the user’s vault, and therefore has effectively no privileges under the Vaults security model.

- Access to application keys is dependent upon the cryptographically verified identity of a process.

- The privileges of authenticated trusted code are constrained such that this code may not be executed by a malicious program to indirectly obtain these privileges.

The attack tree for analysing Trusted Fingerprinting has four top-level branches corresponding to the security properties just described. The entire attack tree consisted of 63,124 nodes. Once again, the number of attack scenarios present in
this tree was greater than the largest positive integer able to be stored and was reported by the software to be -759,101,058, as stated in Figure 8.21.

8.4.2 Summary of Unpruned Trusted Fingerprinting Attack Scenarios

Undetected Modifications

Figure 8.22 gives a high-level view of the complete, unpruned attack tree analysing the Trusted Fingerprinting mechanism with each of the four top-level branches corresponding to the four security goals. The diagram shows the leftmost branch expanded and covers those hypothetical attacks where an object is modified and the modification is not detected. This leftmost branch, in turn, is split into two branches, one considering the potential for undetected modifications to trusted code and the other considering undetected modifications to dependencies of the trusted code. The former branches are fully expanded down to the low-level attack libraries and show that, if the code can be modified (as permitted by Assumption 5, which allows the attacker to bypass kernel and modify objects directly on the disk), this modification may not be detected if the code is either not verified or the verification is performed but the change is not detected regardless. Code may not be verified if matching does not identify the specified target due to, for example, the fingerprint record having been removed (4,328 nodes). Alternatively, matching may succeed but verification still not be performed for some reason (2 nodes). On the other hand, hypothetical attacks where verification does occur, but the modification is still not detected, are addressed in the right-hand branch of this subtree labelled ‘Verification performed but change not detected’. Attacks considered in this branch include alteration of the stored fingerprint value to make this match malicious code (4,326 nodes), and attempts to make the code being verified match the stored fingerprint (9 nodes); for example, through possible race conditions where legitimate code is verified but malicious code substituted at execution time.

The second-level subtree considering detection of modifications to dependencies is
Figure 8.22: Top-level view of the “Defeat Trusted Fingerprinting” attack tree with undetected modification branch partially expanded
similar to that for trusted code but is split into subtrees considering helper applications (8,683 nodes), shared libraries (8,695 nodes) and data file dependencies (8,715 nodes). For example, helper applications have largely the same characteristics as stand-alone trusted code with the only significant difference being that matching is performed on the runtime digraph of the parent program rather than on the various digraphs stored in vaults. Libraries are similar again. However, as libraries may be shared between multiple applications, the analysis also considered potential attacks that attempt to exploit the caching of verified and unverified libraries in memory. Finally, attacks involving modifications to data file dependencies have much in common with the other subtrees. However, these scenarios also include attacks involving automatic updates as the model allows fingerprint values for data file dependencies to be automatically updated by the programs that depend on these files, which potentially creates additional avenues for attack.

**Incorrect Assignment of Privileges**

An important aspect of the Trusted Fingerprinting mechanism is the role it plays in assigning privileges to a process within the Vaults model. Figure 8.23 gives an overview of the attack tree branch describing hypothetical attacks on this privilege assignment, whereby the attacker attempts to have an identity or trust level assigned to a process contrary to the specifications of the model. The tree consists of two branches considering incorrect assignment of privileges to both trusted and untrusted programs, respectively. One subtree considers scenarios where a trusted program executes at its specified trust level (STL). These scenarios should not occur due to the program accessing an object with a lower trust level, which should result in the process’s trust level being downgraded (4,338 nodes). There are also situations where trusted programs should voluntarily lower their effective trust level (ETL), but for some reason do not, and these have also been considered (23 nodes). A third scenario involves the Require Trusted Data fingerprint flag that, if set on a trusted program, should result in the process’s ETL being set to L0 if the
process accesses any unverified data (8,671 nodes). Hypothetical attack scenarios, where this flag is set but not enforced for some reason, were therefore modelled. Finally, scenarios where a trusted program executes at a trust level higher than its STL were also analysed (4,330 nodes).

The second branch of the tree in Figure 8.23 considers where a program, which should execute at L0 (untrusted), for some reason executes at a higher level of privilege. This could be because the attacker has created a new dependency record to correspond to the untrusted, and likely malicious, program (4,324 nodes). Alternatively, the attacker may be using an existing dependency record and attempting to have the untrusted program somehow impersonate this trusted code (36 nodes). These attack scenarios were also evaluated, although were ultimately found to be largely similar to those relating to undetected modifications.

**Unauthorized Access to Application Keys**

In some respects, the application keys mechanism is logically and functionally distinct from Trusted Fingerprinting. However, the security of application keys is dependent upon that of Trusted Fingerprinting. Therefore, potential scenarios involving unauthorized access to application keys were included in the Trusted Fingerprinting attack tree. Figure 8.24 contains the resulting subtree. The leftmost branch considers how an attacker’s malicious code might access another user’s application key, which is the principal attack methodology. For this to successfully occur, the malicious code would need to be considered trusted, such that it has a fingerprint in either GPUB or the user’s vault. More specifically, the fingerprint of the malicious code must match that associated with the application key in order for the kernel to grant access. This situation potentially may be achieved either through the malicious code impersonating a legitimate process (10 nodes) or by the attacker changing the application key’s fingerprint value to match their own code (3,282 nodes). Finally, access to application keys is only permitted where either the program is executing with its usual privileges (its ETL is equal to its
Figure 8.23: Subtree from the “Defeat Trusted Fingerprinting” attack tree showing attacks where privileges incorrectly assigned to an executing program
Figure 8.24: Subtree from the "Defeat Trusted Fingerprinting" attack tree showing attacks where unauthorised code accesses an application key

STL) or the Release Trust Fail flag has been set on the application key to indicate that the key may be released, even if the requesting process is not executing at its normal trust level. Note that, in the attack tree analysis, it was assumed that this final requirement would be met and either or both of these conditions would be satisfied. This assumption was made because none of the assumptions of the analysis definitively excluded these conditions and it ultimately represented an unlikely, but possible, scenario. Finally, the case where a non-malicious but otherwise unauthorised program accessed an application key was also considered. However, very limited scenarios were identified meeting this criteria and essentially no hypothetical attacks resulted.
Figure 8.25: Subtree from the "Defeat Trusted Fingerprinting" attack tree showing attacks where a trusted program is subverted by its parent

**Trusted Program Subverted by Parent**

Figure 8.25 presents a complete subtree modelling all of the potential attacks on the aspect of the Trusted Fingerprinting mechanism that aims to prevent malicious code from gaining additional privileges by executing a trusted program. Vaults imposes trust elevation restrictions in order to prevent these attacks. Improvements to the algorithm for enforcing these restrictions resulting from the attack tree analysis were described in Section 7.13. In this part of the analysis, attacks involving manipulation of the victim user's vault data were excluded since, if the attacker were able to alter the trust level of their malicious program (for example, by creating a trusted fingerprint record for it in the victim user's vault), then they have no reason to need to manipulate a trusted program that already has such a record. Attacks involving manipulation of vault data are considered extensively in each of the other attack trees.

The analysis proceeded by identifying all of the permutations of trust levels for
both the malicious parent and the trusted child. It was identified that the following scenarios could be safely excluded:

- All scenarios where the malicious code has a trust level of L3, as this code already has unrestricted access to the user’s vault.

- Scenarios where the malicious code has a trust level of L2 and the trusted code has a trust level other than L3 as, in this case, the malicious code already has either equal or greater privileges than the trusted code.

- Scenarios where the child program’s trust level is L0, as such programs are untrusted and therefore will execute at the same or lower trust level to a malicious parent.

Excluding the above cases left potential scenarios where the malicious code was currently executing at L0, L1 and L2. In the case of the malicious code executing at L0, the attacks simply reduce to either changing the trust level of the malicious code itself (rather than subversion of a trusted child process) or a kernel error in the application of the trust elevation restriction algorithm. Similarly, the only additional privileges that malicious code executing at trust level L2 may obtain are from an L3 program and, when considering the assumptions of the analysis, this scenario subsequently reduces to a leaf node in the tree.

In the case of the malicious code executing at L1, the attack scenarios become more complex. Since two different programs executing at L1 are likely to also have different privileges (as identified in the attack tree analysis and discussed in Section 7.13), unless the parent and child have the same fingerprint, the child should execute at trust level L0. The attack tree therefore describes attacks for scenarios where the fingerprints for the parent and child are either the same or different. These attacks are shown in the central branch of the tree in Figure 8.25.
8.4.3 Pruning Overview

The pruning analysis of the Trusted Fingerprinting attack tree was performed in two stages. The first stage was performed on the basis that the attacker could not defeat the verifiable integrity property and considers only those attacks specific to Trusted Fingerprinting. This takes into account the results from the attack tree analysing the verifiable integrity property, which found no valid attacks after pruning was performed. In the second stage, the analysis was broadened and the effect on Trusted Fingerprinting of the verifiable integrity property failing was simulated using the attack tree model. This served to analyse the interdependence of the two properties.

8.4.4 Trusted Fingerprinting-Specific Attacks

Excluding attacks dependent upon defeat of the verifiable integrity property resulted in a pruned tree with 37 nodes remaining, representing the elimination of 99.94% of the nodes in the unpruned tree. More significantly, pruning left only five attack scenarios remaining from the billions in the original tree.

The first of these attack scenarios applies to the detection of illicit modifications (Figure 8.26) and specifically the detection of modifications to a data file dependency. The attack represents a limited case where a data file object can be modified and the modification will not be detected, despite verification being performed, as the fingerprint value in the legitimate fingerprint record is incorrectly modified through an automatic update. The attack relates to a very specific and apparently highly unlikely scenario with numerous constraints, but is nonetheless worthy of discussion. When the INSTALLER flag is not set on such a program's fingerprint record, automatic updates will only be performed for data file objects that are direct dependencies of this program. However, the success of this update also depends upon the No AUTO UPDATE flag not being set on these dependencies, which is the first constraint applying to the attack. A second and very significant constraint is that the malicious code is considered trusted and therefore executes at a trust level
Figure 8.26: Pruned subtree for attacks involving undetected modification of a fingerprint object
above L0. Assumption 4 specifically states that users will not configure the system to allow arbitrary code to execute at trust levels of L2 and above, as this would essentially give away all of that user’s privileges. However, this assumption still allows for arbitrary and potentially malicious code to possibly be granted trust level L1. This might be done, for example, if the user wishes to ensure the integrity of the program and the objects it depends upon, even if they do not entirely trust the program. A third constraint is that the malicious code executing at trust level L1 must have sufficient privileges to write to the object in question. Since L1 programs are restricted in their access to protected objects to those specific objects and access modes designated by the user, this severely restricts the attacks that can apply. For example, even if an L1 program specifies an important system configuration file as a dependency, it still would not be granted write access to the object. However, it was identified with further analysis that, if a malicious program executing with trust level L1 shared a common dependency with another L1 program, and if all the above constraints were met, then the malicious program could update the fingerprint values of the second program’s dependency through automatic updates. This would apply even if the malicious program did not ostensibly have a legitimate need to modify the shared dependency and is considered a possible attack within the scope of the analysis, as Assumption 4 does not explicitly require L1 dependencies to be write protected. Nonetheless, this represents an extremely limited avenue of attack. The attack applies only to L1 programs and requires the user to explicitly designate the object as a dependency of the partially trusted program, which thereby authorises the program to perform such automatic updates. This is documented and expected behaviour and therefore, arguably, a somewhat dubious vulnerability. Indeed, the scenario arguably violates the spirit, although not the actual letter, of Assumption 4. Furthermore, the analysis also demonstrated that problems can be easily prevented by cryptographically write protecting objects that are dependencies of L1 programs but should not be able to be modified by these programs.

The other four attack scenarios apply to programs being assigned incorrect privi-
leges and are given in Figure 8.27. All of these attacks apply in the case where a trusted program should voluntarily lower its effective trust level (ETL) at runtime but, for some reason, does not do this. The first attack applies where the attacker modifies the program's code on disk to prevent it lowering its trust level. This is, arguably, an unrealistic attack as, if the attacker has the ability to modify a trusted program's code, they also have the ability to make it do anything beyond simply neglecting to lower its trust level. Furthermore, successfully completing this attack would require violation of the Trusted Fingerprinting security property pertaining to detection of illicit modifications and, as just discussed, no attacks applying to modification of code were found in this branch of the attack tree. Therefore, despite being found in the pruned tree, the attack tree results indicate this attack clearly would not actually be possible in practice. Finally, the attack only applies in the case of L1 programs, as otherwise this would require that the verifiable integrity property be defeated, since L2 and L3 programs are assumed to be cryptographically protected under Assumption 4. Nonetheless, this attack is included for the sake of completeness.

The remaining three attack scenarios all involve a trusted program failing to lower its privileges due to an error in that program. Whether this is the result of actions taken by the attacker to cause the failure, or something that happens spontaneously in a given set of circumstances, the possibility of such errors is not excluded by the assumptions of the analysis. However, the REQUIRE TRUSTED DATA flag significantly mitigates such problems as it reduces the need for programs to voluntarily lower their trust levels at runtime. As discussed in Section 5.4.9, the flag operates by automatically lowering a program's ETL to L0 if that program opens a non-dependency object for reading. Beyond this, universally preventing general malfunctions in programs is outside of the scope of protection afforded by Trusted Fingerprinting. Nonetheless, its inclusion in the analysis recognizes that this represents an unavoidable problem in practice.
Figure 8.27: Pruned subtree for attacks involving privileges being incorrectly assigned
Malicious Code Injection Attacks

A special case of such programmatic errors are malicious code injection vulnerabilities, such as buffer overflows, where an attacker is able to insert malicious code into a running program so that the malicious code is executed with that program’s privileges. Such attacks represent a powerful way for an attacker to potentially alter a program’s behaviour. However, despite this, these attacks do not strictly violate the first of the Trusted Fingerprinting security properties, as the mechanism verifies the integrity of the object at execution or access time and provides no explicit guarantees as to the maintenance of that integrity beyond this. As a result, these attacks are not included in the branch of the attack tree analysing this property.

However, these attacks represent a legitimate way of preventing a program from lowering its trust level and, more generally, represent a serious problem in practice that is worthy of further discussion. While eliminating these errors in the program code itself is the ideal way of mitigating the problem, this task has proven to be difficult in practice [41]. Hardware and software mechanisms exist that have the ability to limit the scope of these attacks and are also beneficial [267]. However, the Vaults architecture itself also assists in the mitigation of this class of vulnerability with respect to the security goals of the model.

Normally, the amount of code that may be injected into a vulnerable program is extremely limited and is used to execute a second program that either carries out a series of tasks on behalf of the attacker or provides them with a generic interface to the system (such as a shell), thereby granting them all of the privileges of the vulnerable program [36]. The design of the Vaults security model serves to stymie both of these attack methodologies.

In the first case, if the injected code executes a second malicious program then, assuming this program does not have a trusted fingerprint, it will not execute with the privileges of its parent. Similarly, if the attacker uses the injected code to execute a shell (which would normally have a trust level of L2), any code subsequently executed by this shell will only retain its privileges if it too has a trusted fingerprint.
Finally, code injection attacks can only be exploited where a privilege boundary exists. Examples of such scenarios include where the vulnerable program is a network server, fixed privilege code (e.g., the set UID mechanism in Unix), and code interactively executed by another user. In the case of network servers and similar service programs, as these types of programs do not execute interactively and do not have access to a user vault, they have no additional privileges under the Vaults model that the attacker can obtain. Similarly, there is no equivalent of Unix’s set UID programs in the Vaults model and therefore these privilege boundaries do not exist. Vulnerable code executed interactively by another user could potentially be exploited by an attacker to obtain its privileges if the attacker were able to identify a suitable vector for injection of their malicious code into the vulnerable program at runtime. However, as discussed above, this attack would be limited to the amount of malicious code that was able to be injected, as Vaults Trusted Fingerprinting would curtail the privileges of any malicious programs executed subsequently. Therefore, while not representing a complete solution, the Vaults security model significantly limits the impact from this dangerous class of attack.

8.4.5 Verifiable Integrity Property Dependence

To assess the dependence of the Trusted Fingerprinting mechanism on the verifiable integrity property, the attack tree model was adjusted to assume that defeating the verifiable integrity property is possible. The pruning analysis was then repeated with a number of attack scenarios increasing from five to 176, with the new pruned tree having 2,059 nodes. Of these scenarios, five were the attacks just discussed. In addition to these, allowing the attacker to violate the verifiable integrity property meant that L2 and L3 programs could be modified in order to prevent them dropping their privileges when required. This is a direct extension of the attack on L1 programs described in the previous section. It is important to note that, as with the modification of L1 programs, for this new attack to be successful, the attacker would also have to compromise the Trusted Fingerprinting mechanism for verifying
code integrity. As has already been discussed, the attack tree analysis identified no means for achieving this goal. The remaining 170 attack scenarios were a direct result of the MURCA previously discussed and were of the same form as described in relation to the confidentiality property in Section 8.2.4.

8.5 Observations on Attack Trees

As the attack tree analysis was being performed, a number of observations were made concerning the technique and its practical applications. Significantly, it was found that the tree structure assisted greatly in managing the complexity of the attack properties being analysed. Without this benefit, it would have been significantly more difficult, if not infeasible, to obtain the level of comprehensiveness that was achieved with this analysis. However, as the complexity of a given tree increased, there was a corresponding reduction in the manageability of the tree. Over time, this effect was partially mitigated as familiarity with the attack tree analysis technique increased and a set of standard practices for constructing and managing trees was adopted. However, it was always found that the process of manually ‘walking the tree’ to review or evaluate a particular attack remained manageable, even for the most complex tree structures. This was particularly true when considering pruned trees where extraneous and impossible attack scenarios had been removed.

8.5.1 Subtree Libraries

The subtree libraries feature provided by the SecurITree software, described in Section 6.3.5, assisted significantly in managing tree complexity during the construction phase and into the pruning analysis. It was frequently found that attacks on different aspects of the model shared common attack scenarios and the libraries feature reduced the amount of manual work involved by allowing reuse of existing subtrees. An added benefit of this approach was that subtrees assisted in abstract-
ing complexity when analysing patterns in various attack requirements, and often allowed conclusions to be drawn between the similarities of different attack methodologies. For example, one subtree related to the subversion of the assignment of the VID to a process in order for the attacker to execute a process with another user’s VID. The two methodologies identified for achieving this goal involved either the attacker learning the the user’s vault key in order to gain access to the victim’s vault, or otherwise associating the victim’s VID with the attacker’s vault for which the attacker would already know the key. The latter attack methodology led to a subtree of 762 nodes. However, upon reviewing this subtree, it became clear that all of the attacks it described, which were not trivially impossible, were dependent upon the same subtree library concerning the attacker learning the user’s vault key as applied in the alternative high-level attack. From these observations made when reviewing the tree, it can be concluded that attacks allowing the attacker to execute a process with another user’s VID are effectively equivalent to the attacker learning that user’s vault key — which is as it should be according to the security semantics of the model. These kinds of observations were often repeated and the abstraction of attack components into subtree libraries assisted greatly with the recognition of these patterns.

However, an outstanding problem that limits the benefits of subtree reuse involves identifying at exactly what level a subtree should be separated and placed in a separate library. If the separation occurs at too high a level, there is an increased likelihood of potential problems occurring, such as a circularity of attack requirements or a disconnect between the goals of the attack at the higher level of the tree and those specific methods identified in the lower levels. Further consideration of general strategies for identifying the appropriate level for decomposition of the tree may be a valuable future research area leading to improve methodologies and perhaps enhanced tool support.
8.5.2 Tree Properties and Characteristics

During tree construction, the different impacts on security of the occurrence of OR and AND nodes was observed. OR nodes were more common and effectively played the roles of a delimiter between the various leaf nodes and, at the higher levels of the tree, a mechanism to organise and categorise attack methodologies as per the logically complementary attack scenarios technique (Section 6.3.3). Therefore, while OR nodes represent different options available to the attacker, in a subtree containing no AND nodes it is the number of leaf nodes, rather than OR nodes, that is the more significant indicator of the vulnerability of the mechanism being analysed.

However, the impact of AND nodes on the success or otherwise of attack scenarios was significantly more dramatic. As was discussed in Section 6.2.7, AND nodes specify prerequisites that must be satisfied in order for an attack methodology to succeed. As the trees were constructed, more details became clear concerning the impact of AND nodes on the likelihood of the success of an attack scenario. Intuitively, the greater the total number of AND nodes, the greater the number of constraints placed on the attacker with respect to the relevant attack scenarios. Therefore, the number of AND nodes found in a tree corresponds to the degree of defence-in-depth of the system being analysed [15]. More specifically, the higher the proportion of subtrees of a given AND node that evaluate as IMPOSSIBLE according to the relevant criteria, the more secure a design will be in this respect.

The impact of AND nodes can apply in different ways. For example, a number of AND nodes found horizontally immediately below the root node of the tree would indicate that each attack option was subject to at least two constraints. On the other hand, a series of nested AND nodes indicates an attack methodology subject to multilayered constraints. These observations open the possibility for further research into quantitative interpretation of attack tree structures. That is, by analysing the number and placement of AND nodes in a subtree, it may be possible to obtain a quantitative metric of the security of the design. In particular, factors
relating to AND nodes and their impact on security include the total number of AND nodes (the greater the quantity, the lower the likelihood of an attack succeeding), the depth at which these AND nodes appear in the tree below the root node (the closer to the top of the tree, the lower the likelihood of attack success), and the number of descendant nodes of a given AND node. That is, the more descendants of an AND node, the greater the probability that these attack scenarios will not be possible. In particular, the greater the number of AND nodes that are descendants of another AND node, and the greater the number of immediate descendants of the parent AND node, the lower the likelihood of success of these attack scenarios.

While AND nodes were observed to have a far greater impact on security than other nodes, such a metric would need to also consider the role of OR and leaf nodes. For example, a more general scheme for quantifying security using attack tree structures could consider the average number of nodes above the leaf node where a particular subtree becomes impossible. The closer this value is to zero, the greater the security of the design. Such an approach takes into account the impact of AND nodes to some degree as subtrees often become impossible at this point. In any case, this and other approaches for quantitatively measuring security using attack trees represent a significant avenue for potential future research.

8.5.3 Limitations of Pruning

The processing of the tree using pruning analysis to remove invalid attack scenarios is critically important as it simplifies the tree by leaving only those attacks that are possible according to the set criteria. However, a disadvantage of this approach is that it may obscure results that are eliminated during pruning and yet are still of interest and worthy of consideration. This effect primarily occurs at AND nodes, where all subtrees of this node must be successful in order to remain in the pruned tree. However, if one of these subtrees becomes impossible, the other will also be pruned. A hypothetical example derived from this analysis would be an AND node concerning theft of a usable and valid ticket broken into subtrees covering both
theft of the ticket and its successful use, despite the thief having a different VID. If either of these subtrees were to be evaluated as IMPOSSIBLE, both would be removed from the analysis at pruning. However, if the other subtree remains possible, this is an important result that may represent an opportunity for further securing the model. This observation indicates that further analysis of AND nodes identifying scenarios such as this can sometimes yield valuable results concerning the security of the design being analysed and ways of further improving it.

8.6 Discussion and Summary of Attack Tree Results

8.6.1 Summary of Attacks

The results from the attack tree analysis come in two forms. The first set of results is the vulnerabilities identified during tree construction, which were discussed in Chapter 7, and informed improvements to the model to avoid these security problems. The second set of results is those from the pruning analysis described in this chapter that provide evidence as to the security of the Vaults model.

The pruning results identified attacks on the confidentiality property through a race condition in manual updates — the manual update race condition attack or MURCA. This attack involved substituting malicious code for a recently upgraded program prior to its fingerprint being manually updated. However, the attack tree model also identified that these attacks were dependent upon defeat of the verifiable integrity property and, when this latter property was analysed, no valid attacks were found. Therefore, while the MURCA subtrees also exist in the tree analysing the verifiable integrity property, the prerequisite of defeating this same property led to the requirements of these attacks becoming circular and therefore impossible. Consequently, an empty pruning tree resulted indicating no possible attacks on the verifiable integrity property. As compromising verifiable integrity is required to violate the confidentiality property, under the assumptions of the analysis, no valid attacks were found on both of the major security properties of the Vaults model.
Despite this, the MURCA represents a serious practical issue that the attack tree analysis brought to light. To further mitigate this problem, the INSTALLER flag was developed. This flag provides a convenient mechanism for automatically updating fingerprints when installing new or updated software, thereby eliminating the need to perform potentially vulnerable manual fingerprint updates.

A highly limited set of attacks were also identified in the Trusted Fingerprinting mechanism. One of these attacks involved an L1 program automatically updating the fingerprint of a dependency shared with another trusted program. While somewhat unlikely and subject to numerous constraints, this attack remains technically possible under the stated assumptions of the analysis. The attack demonstrates that L1 programs are largely untrusted and need to be treated with caution. It also highlights that assigning an object as a dependency of a program has the effect of granting that program a very limited set of additional privileges in relation to the object; namely, the ability to potentially automatically update that object’s fingerprint if it also has write access. While potentially problematic in a very rare set of circumstances, this is intended and otherwise correct behaviour. The identification of these ‘attacks’ is therefore a valuable result that can serve to inform user behaviour and system configuration. For example, the problem is easily prevented if dependencies of L1 programs are cryptographically write protected and, if the object concerned is security-sensitive, this should apply regardless. The other identified attacks on the Trusted Fingerprinting mechanism all involve a trusted program failing to pre-emptively lower its trust level as a result of some form of error in the program when circumstance indicates that it should. These attack scenarios delineate the limits of the model in being unable to eliminate programmatic errors in user software and therefore, arguably, are more indicative of flaws in the analysis assumptions than in in the security model itself. However, again the model provides a mechanism for avoiding these problems in practice through use of the REQUIRE TRUSTED DATA flag, which supports automatic lowering of a process’s trust level if this program accesses untrusted data.
<table>
<thead>
<tr>
<th>Tree</th>
<th>Total Leaf</th>
<th>Total Possible Leaf</th>
<th>% Leaf Possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>80698</td>
<td>30897</td>
<td>26.64%</td>
</tr>
<tr>
<td>Integrity</td>
<td>88484</td>
<td>26445</td>
<td>33.89%</td>
</tr>
<tr>
<td>Fingerprinting</td>
<td>34075</td>
<td>12177</td>
<td>28.57%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tree</th>
<th>A5 Leaf Possible</th>
<th>% A5 Total</th>
<th>% A5 Possible</th>
<th>A5 Unpruned</th>
<th>% A5 Eliminated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>7017</td>
<td>8.09%</td>
<td>22.71%</td>
<td>63</td>
<td>99.10%</td>
</tr>
<tr>
<td>Integrity</td>
<td>7161</td>
<td>8.09%</td>
<td>27.08%</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Fingerprinting</td>
<td>2768</td>
<td>8.12%</td>
<td>22.73%</td>
<td>1</td>
<td>99.99%</td>
</tr>
</tbody>
</table>

Table 8.1: Statistics on leaf node possibility and Assumption 5

8.6.2 Security against Highly Privileged Attackers

Assumption 5, detailed in Section 6.3.6, permits the attacker to have a high level of privilege on the system; namely, the attacker has the equivalent of Unix superuser privileges and is able to bypass the operating system kernel and access secondary storage devices directly. This assumption is important as its impact on the analysis results reflects the success of the model in achieving its security goals, even against a highly privileged attacker. To determine the impact of the assumption, leaf nodes were extracted from the unpruned trees using the SECUITREE software and exported as CSV files. A Perl script was developed to extract the nodes that were possible from these results, along with information identifying which assumptions were involved in determining the node’s value. Statistics were then collected on these results and are presented in Table 8.1. This information showed that, across all three attack trees, only a single attack scenario remained in the unpruned trees that was dependent upon the attacker having a high level of privilege. Details of these results will now be discussed.

The columns in the first part of Table 8.1 show the total number of leaf nodes in each of the unpruned trees, the total number of these nodes that were possible, and this value as a percentage. While the number of leaf nodes is approximately proportional to the size of the tree, the proportion of possible leaf nodes is notably smaller in the verifiable integrity tree compared to the other two trees, which are close to identical. The similarities in the tree statistics demonstrates the underlying
commonality of attack structures. While the security properties considered by the
trees are quite different, attacks on the Vaults model were typically found to reduce
to a common set of basic attacks, such as obtaining a system vault key or learning
the user’s vault passphrase. These underlying ‘attack primitives’ were contained
within the attack tree libraries, shared between the main trees. Consequently, the
size and frequent recurrence of these libraries means the properties of the primitives
overwhelm the high-level differences in attack methodologies in the statistics pre-
presented in the table. However, while the attacks modelled in the verifiable integrity
tree depend upon these same primitives, fewer leaf nodes were possible due to the
circularity in these shared attack methodologies that depend upon violating the
same property as discussed in Section 8.3.3.

The second part of the table reports the number of leaf nodes that were dependent,
either entirely or partly, on Assumption 5. These represented proportionally about
8% of all leaf nodes and 22.7% of possible leaf nodes in the confidentiality and
Trusted Fingerprinting trees. However, in the verifiable integrity tree, this propor-
tion increased to 27.08% as a result of their being fewer possible leaf nodes overall
for the reasons just described.

Finally, the second part of Table 8.1 also includes information about the pruning of
Assumption 5-dependent leaf nodes. Since all of the attack scenarios in the verifiable
integrity tree were removed during pruning, this shows that, according to the attack
tree analysis, a highly privileged attacker cannot violate this security property.
Similarly, while 63 Assumption 5-dependent leaf nodes remained in the interim
pruned confidentiality tree, as discussed in Section 8.3.3, these were all dependent on
the attacker also being able to defeat verifiable integrity. Since this was not possible,
all of these nodes were ultimately pruned. Consequently, the attack tree analysis
ultimately showed that the Vaults model was able to maintain the confidentiality
of protected data when under attack by a highly privileged attacker. These results
with respect to the two central security properties of the Vaults model support the
argument that cryptographically-based security models are better able to maintain
their security properties against a higher level of threat.

In the Trusted Fingerprinting tree, only one Assumption 5-dependent leaf node remained after pruning. This represented a 99.96% elimination of such nodes. This figure is a quantitative indication of the security of Trusted Fingerprinting against a highly privileged attacker. The single remaining attack is that described in Section 8.4.4, where a malicious program assigned L1 privileges becomes able to automatically update the fingerprint of its dependencies when the relevant flags are set correctly, and the malicious program is allowed to modify the object in question. As noted, this ‘vulnerability’ is actually correct and intended behaviour, and only becomes problematic where the dependency concerned is shared with one or more non-malicious programs. In this particular case, the presence of this attack scenario primarily serves to highlight this potentially undesirable behaviour so that it can be avoided in practice. This, therefore, is the only valid attack scenario from the billions considered in the attack trees that is in any way assisted by the attacker having a higher level of privileges and demonstrates the resilience achievable by cryptographically-based models.

8.7 Conclusion

This chapter has presented the results from the attack tree analysis of the Vaults security model. Summaries have been given of the complete unpruned trees containing hypothetical attacks against the design of the model. Detailed results have then been described for applying pruning criteria to those complete trees in order to remove those attacks not possible according to the assumptions of the analysis. These results showed that no attacks could be identified against the two principal security properties, confidentiality and verifiable integrity. However, the attack tree modelling also identified the seriousness of the threat posed by potential attacks involving race conditions when manually updating the fingerprints of trusted programs. While these attacks were not sufficient in themselves to violate either of the two main security properties, the INSTALLER flag was added to the model to facil-
iterate the use of trusted installers to perform automatic updating, thereby limiting the need for manual updating in practice.

The attack trees analysing the Trusted Fingerprinting mechanism also showed it to be very secure. However, an attack was identified involving the ability of potentially-malicious L1 programs to automatically update the fingerprint values of dependencies that may be shared with other trusted programs. Closer inspection showed that this attack was, in fact, correct behaviour and its presence in the pruned tree is therefore valuable in serving to highlight the potential for this undesirable behaviour when the user configures and uses the system in a certain way. Attacks on Trusted Fingerprinting were also found where trusted programs might fail to voluntarily lower their trust levels when accessing program-specific types of untrusted data. However, such issues are entirely indicative of the general unavoidability of undesirable behaviour by trusted programs, rather than a deficiency in the underlying security model, and possibly should have been excluded from the analysis by assumption. In any case, this problem can be mitigated through the use of the \texttt{REQUIRE TRUSTED DATA} flag, which causes the program to lose its privileges if it accesses any untrusted data.

Finally, the effect of the assumed ability of an attacker to obtain a higher level of privilege on the system and potentially bypass the security kernel in accessing objects on secondary storage was also considered. A statistical analysis of the presence of this assumption in the attack tree nodes showed that it had no effect on either the confidentiality or verifiable integrity properties, and played only a minor role in the malicious L1 dependency automatic update attack identified associated with Trusted Fingerprinting. These results therefore support the claim that the cryptographically-based Vaults model is secure, both in general and when defending against a highly privileged attacker.
Chapter 9

Discussion and Conclusions

9.1 Summary of Research Results

This thesis has presented the Vaults scheme, a novel security model that utilises cryptographic features in order to achieve superior security compared with existing non-cryptographic mechanisms. However, the Vaults model also maintains an access control interface that is notionally similar to existing mainstream schemes, rather than the often complex and unfamiliar approaches used by other high security alternatives. The Vaults model includes mechanisms to cryptographically protect the security of files, verify and assign trust levels to processes and securely manage the use of user passwords and other keys. However, unlike previous uses of cryptography, these mechanisms are employed in an integrated manner that obtains maximum benefit by recognising the interdependence between these features.

Finally, the complete model was analysed using a threat modelling-based methodology in order to evaluate its overall security and, in particular, assess its ability to resist penetration by highly privileged attackers.

Achieving the goal of maintaining its specified security properties against a privileged attacker requires addressing the weaknesses of widely used security models, where all administrative privileges are concentrated in a single user identity. Under such schemes, the concentration of privileges makes this account the primary target
for attackers. Furthermore, the account is also highly exposed to attack as it must be used whenever any administrative task needs to be performed. If an attacker is able to compromise the superuser account, they are generally able to bypass any and all security restrictions enforced by the system.

Vaults addresses this problem in a number of ways, such as by limiting the degree to which identity determines privilege. Under the new scheme, the ability to access a protected file depends primarily on the contents of the vault to which the process has access, rather than the identity of the user nominally associated with the process. Consequently, gaining access to the superuser account does not, in itself, confer the privileges necessary to access protected files. Even more significantly, the use of cryptography facilitates a passive protection architecture, as defined by Gifford [127] and discussed in Section 2.5.2. As a result, protection remains in effect even if a privileged user bypasses the active protection mechanism in the form of the security kernel and accesses objects directly on the secondary storage device.

This property gives the Vaults model the ability to maintain its specified security properties, even if an attacker obtains complete control over the superuser account. Finally, while Vaults includes a highly privileged account (the prime user) that is able to bypass its security controls, the design of this account specifically avoids the problems of privileged identities in existing mechanisms. In particular, the account is designed to be rarely needed, unlike the ubiquitous superuser. Similarly, these privileges only apply to interactively executed processes that have access to the necessary cryptographic tokens held in the prime user’s vault, and then only to cryptographically-verified processes that have been specifically authorised to do this. Consequently, the opportunities to attack these privileged processes are extremely limited, at best.

### 9.1.1 Vaults Security Features

The titular component of the Vaults security model are the secure repositories used for storing sensitive values and access tokens. In the case of user vaults, they store
parameters specific to a particular user and these tokens effectively determine the
privilege of the user on the system. A set of system vaults also exist, which hold
data relevant to the system as a whole, and should not be under the control of any
individual user.

The Vaults model includes the Trusted Fingerprinting feature that allows security-
relevant programs, and other objects these depend on, to be cryptographically ver-
ified prior to execution or access. Trusted Fingerprinting applies on both a global
and per-user basis. Therefore, the security administrator can create fingerprints
for programs that are relevant to the security requirements of all users or the sys-
tem in general. These fingerprints are stored in the Global Public vault (GPUB)
and verified whenever any user executes one of the specified programs. However,
users may also instantiate fingerprints for programs relevant to their own individ-
ual security requirements. Fingerprints for such programs are stored in the user’s
vault and similarly verified upon execution. In the case of both global and local
fingerprints, verification is performed, not just for the executable program, but
also for a variety of objects on which the program depends for secure and correct
operation. These additional dependencies may include shared libraries, as well as
data files containing information such as configuration parameters. Fingerprints
for library dependencies are verified prior to linking, while data file fingerprints are
verified when the designated program opens the object for reading. The Trusted
Fingerprinting mechanism is also capable of tracking modifications to data file de-
pendencies and automatically updating fingerprint values where these modifications
are performed by the relevant verified program. This feature means that the finger-
prints of regularly modified security-sensitive files are kept up-to-date and avoids
the security risks and inconvenience associated with performing manual updates.

Trusted Fingerprinting also facilitates the assignment of privileges to processes.
Program fingerprint records specify one of three trust levels that will then be as-
signed upon successful verification. The trust level of a process dictates the nature
and extent of its access to cryptographically protected objects on the system. Pro-
grams that do not have a fingerprint, and have not been cryptographically verified, are regarded as untrusted. Untrusted programs have no access to protected objects and are therefore unprivileged. This design resolves a significant weakness in existing code verification schemes that do not differentiate between verified and unverified code with respect to their privileges. Failing to make this distinction opens the door to a class of attacks where verification is suppressed and malicious code substituted for otherwise trusted code.

Vaults also incorporates a cryptographically-based access control model. Users may designate specific files to be read protected, write protected or both. Read protection results in the file’s contents being encrypted, and this data will only be decrypted if the process attempting to read the object has access to the necessary token. Write protection involves calculating a message authentication code (MAC) of the file’s contents. Subsequent attempts to open the file require that the MAC be verified to ensure that the file has not been modified and, again, the correct token is required for write access. Upon legitimate modifications to the file, the MAC value is recalculated and securely stored for later verification. Vaults cryptographically enhanced access controls are designed to achieve their confidentiality and verified integrity goals, even against an attacker who is able to bypass the security kernel and gain unrestricted access to objects stored on the disk.

The access control model also supports sharing and revocation of privileges. When a user first protects a file that they own, they receive a primary ticket. Possession of this token permits subsequent access to the file and also allows the user to grant access to others by the creation of secondary tickets. These tickets are securely exchanged using a lightweight PKI, and are irreversibly and cryptographically linked with the intended recipient’s vault. This design prevents recipients of secondary tickets from duplicating and distributing them to unauthorised users, and makes the use of stolen tickets impossible. The primary ticket holder may also revoke access to the protected object, thereby rendering the specified secondary tickets unusable by their recipients. Revocation is computationally inexpensive and does
not require re-encryption of existing data. Therefore, unlike most cryptographic filesystems, the Vaults approach maintains the properties expected of generic access control schemes. Finally, the design includes a mechanism to facilitate the secure verification of the protection status of system files. This is necessary because naive storage of metadata in the filesystem can be trivially subverted by an attacker who is able to bypass the security kernel and tamper with the values directly.

However, all of these features of the Vaults security model are not merely discrete, independent mechanisms but are designed to be complementary and synergistic. While the vaults themselves underpin the other two mechanisms by securely storing keys, tickets and fingerprints, the Trusted Fingerprinting mechanism and associated trust levels are used to authenticate the release and use of these parameters. This resolves a serious weakness in many previous secure storage architectures, where ascertaining whether to release sensitive values to unauthenticated processes remains problematic. Similarly, the integration between Trusted Fingerprinting and the cryptographic access controls allows for verified and unverified code to be distinguished from one another in terms of access privileges to protected objects. Therefore, unlike previous uses of cryptography to improve operating system security that exist as ad hoc, independent mechanisms designed to achieve a single security function, the design of Vaults recognizes the interdependencies between these mechanisms and integrates them, leveraging their individual strengths to provide maximum security benefits.

### 9.1.2 Security Analysis

To examine the security of the Vaults model, and evaluate its success in utilising cryptography to improve security, a threat modelling-based analysis was performed using attack trees. The analysis sought to collect evidence, not only as to the overall security of the model, but particularly with respect to its principal security objective of maintaining the properties of confidentiality and verifiable integrity when faced with a highly privileged attacker. After considering a number of security analysis
techniques, the attack tree methodology was selected. This approach allowed the elucidation of the potential attacks on the model that might be available to a highly privileged attacker and provided a mechanism to identify which of these attacks would actually be achievable.

Attack trees were constructed for the model’s major security features and properties; namely, the confidentiality and verifiable integrity properties, and the Trusted Fingerprinting mechanism. In addition to the natural features of the attack tree technique, a refined and even more methodical approach involving the identification of logically complementary attack scenarios was developed and employed in order to maximise the completeness of the analysis. All scenarios that could bring the attacker closer to achieving their goal were included. Attack tree libraries were also developed, collecting together subtrees containing reusable, generic scenarios. While libraries often needed to be adapted to the specific tree in which they were to be used, this approach assisted significantly in managing the complexity of the large trees that were developed, as new attacks incorporated into a library after being discovered would automatically be inherited by all trees referencing that library. The complete unpruned trees, containing all hypothetical attacks identified, constituted a total of 160,509 nodes for the confidentiality property tree, 163,816 nodes for the verifiable integrity property tree, and 63,124 nodes for the Trusted Fingerprinting tree.

Prior to beginning the building of the attack trees, a list of 12 assumptions was compiled. During tree construction, leaf nodes were assigned a Boolean value indicating whether the particular stage or component of the overall attack described by the node could be achieved by an attacker. The value of this attribute was principally determined by whether or not the action described by the node violated one or more of the stated assumptions. If the action did not violate any of the assumptions, it was considered to be possible even if the action was very difficult or required a significant degree of luck. A small number of leaf nodes were also designated as being IMPOSSIBLE where the attack requirements of the scenario had
become circular, and therefore logically impossible.

Once construction of the tree was complete, the tree was then ‘pruned’ by removing all attack scenarios deemed impossible according to the values of the leaf nodes and the logic defined by the structure of the tree. This process removed all hypothetical attacks, leaving only those possible according to the analysis assumptions. These results initially identified a number of attacks on the confidentiality property, involving a potential race condition when manually updating the fingerprint value for a trusted program. However, the analysis also identified that these attacks required defeating the verifiable integrity property in order to be successful. Since no attacks were found on verifiable integrity, when the pruning analysis was repeated on the confidentiality tree taking this result into account, no attacks on confidentiality were ultimately found to be possible. Consequently, the analysis demonstrated that the Vaults model is able to maintain security, even in the face of a highly privileged attacker.

A small number of highly limited attacks on Trusted Fingerprinting were found to remain after pruning. However, closer inspection showed that they either reflected the conservative nature of the analysis assumptions or were, in fact, correct behaviour and therefore served to more clearly delineate the bounds of the security enforced by the model. A number of issues with the model were also identified during tree construction. In the small number of cases these reflected potential attacks, but more often highlighted ambiguities in the specification of the model that could potentially lead to undesirable behaviour. As a result, the model was adapted to take into account these issues and the attack trees correspondingly modified. These results show the benefits of the attack tree approach in helping to clarify and refine the design of a security model being analysed.
9.2 Limitations of the Research

9.2.1 Lack of Availability Protection

The Vaults model has specific security objectives that must be met with respect to confidentiality and verifiable integrity. In particular, the model is designed to meet these objectives even if an attacker is able to bypass the security kernel and access objects directly on the disk. However, there are no stated security goals with respect to availability. Therefore, while an attacker would not be able to view or undetectably modify designated protected objects, the scheme has no mechanism to prevent them from deleting or corrupting such data. This represents a deliberate limitation of the model’s scope that reflects both typical user requirements and the practical reality that it is very difficult, if not completely infeasible, to prevent such actions by a privileged attacker. In particular, an attacker who can bypass the security kernel cannot be actively prevented from tampering with protected data without taking significant additional measures that go well beyond the scope of the Vaults scheme. This is therefore an acknowledged limitation of the model.

9.2.2 Limitations of the Analysis

As described in Chapter 6, the attack tree technique was selected as being most suited to the task of analysing the Vaults security model. In particular, attack trees can be applied to analyse schemes involving various levels of abstraction and do not require any specific level of detail. The technique was also well-suited for delineating the capabilities of a highly privileged attacker with respect to the restrictions enforced by the model. Consequently, the results demonstrate the ability of Vaults to maintain its specified properties when faced with an attacker who can modify objects stored directly on the disk. It is believed that the comprehensive nature of the attack trees produced constitutes significant evidence regarding the degree of security the model achieves.

However, as with virtually all analysis techniques and methodologies, success is
significantly dependent upon the ability of the analyst to correctly apply it. It is always possible for implicit assumptions to be applied by the analyst without these being recognized. Such unconscious assumptions are generally hard to identify. As a result, certain valid attacks may be unconsciously excluded when this is not justified. More generic errors can also creep in, particularly when a high level of complexity is involved.

Fortunately, the methodical unstructured nature of attack trees reduces the dangers stemming from unrecognized assumptions. The logically complementary attack scenarios technique employed (Section 6.3.3) further minimises this risk. Finally, to the degree allowed by the large size of the trees involved, trees were checked (and often rechecked) after construction to verify correctness. The logic expressed in tree libraries, was also examined and reviewed before incorporating them within another tree. These practices therefore sought to minimise the risk of errors in the tree. However, while all efforts have been made to reduce the risk of these problems, the possibility of errors in the analysis cannot be definitively excluded and remains an acknowledged limitation of the research.

9.2.3 Absence of an Implementation

Many of the schemes that employ cryptography discussed in Chapter 2 have sample implementations available. However, these implementations are typically little more than proofs-of-concept and very few have seen deployment beyond their use by the researchers involved. Furthermore, these implementations provide only limited information concerning the security of the schemes. The principal objective of this research was to demonstrate how cryptographic models can be used to improve security and a prototype or sample implementation would have done little to achieve this goal.

An implementation would have demonstrated the practicality of the Vaults model. However, Vaults does not have any fundamental requirements that go beyond the low-level features typically found in contemporary hardware and widely available
operating systems such as Linux. Indeed, Vaults utilises the same basic cryptographic mechanisms employed by previous schemes, albeit in an integrated and holistic way. Further, given the number of practical but highly vulnerable systems in widespread use, the development of a secure system appears to be a bigger and more important challenge than simply getting a system working.

One practical use of implementations of earlier security schemes has been to obtain benchmarks in order to assess the impact of the use of cryptography on overall performance. However, the acceptable performance of these schemes—even as non-optimised prototypical implementations—along with the continuous improvement in hardware performance, indicate that Vaults performance would be acceptable in practice\textsuperscript{1}. While the use of cryptography naturally imposes an overhead on performance, the potential security benefits are believed to outweigh these additional costs. As Schneier & Ferguson [17] have stated: “We have enough fast, insecure systems. We don’t need another one.”

9.3 Directions for Further Research

9.3.1 Distributed Applications of Vaults

The Vaults scheme was conceived from the beginning as a security model applicable to a single autonomous system. This focus was significantly influenced by the view put forward by Loscocco et al. [4], that application and network-based security mechanisms are fundamentally unable to sufficiently mitigate increasingly critical security risks and instead stronger operating system-level security features are required. Therefore, a central goal of the research was to explore the unique advantages that the application of cryptography was expected to have in the specific context of operating system security.

\textsuperscript{1}It is interesting to note that in the paper describing his scheme, Gifford [127] entirely discounts the impact of cryptography on real-world performance. Given this paper was published in 1982 and, considering the progress with respect to hardware and cipher performance since this time, it seems likely that fears over the computational costs of cryptography are largely unwarranted.
However, the cryptographic nature of the Vault model suggests a natural applicability to distributed systems. For example, the assumption that an attacker can potentially access and manipulate data on a secondary storage device implies significant commonality with the threat model of a distributed system. Indeed, an initial inspection suggests that, under the model, a transparently mounted network filesystem would inherit similar, if not identical, security properties that apply to local storage. While scenarios involving interactions between multiple autonomous systems are likely to involve more complexity, the scheme appears amenable to the use of secondary tickets held on a different system. This could be done, for example, by including both the ticket and certificate in the request, along with a suitable protocol to authenticate the request and ensure freshness and transport security. With the rise of cloud computing and questions regarding its security [268], the relevance of cryptographically-based access control schemes, such as Vaults, seems likely to increase.

The central issue where multiple systems, and potentially multiple users, are involved is ensuring adequate authentication. While the lightweight PKI presented here is adequate for inter-user authentication within a single system, more sophisticated features are likely to be required if applied to the distributed context. For example, mutual authentication between users from two autonomous systems would require a certificate chaining mechanism and either mutual authentication between the certification authorities (CAs) of both systems or a mutually trusted root CA. Alternatively, an external PKI such as X.509, which already has these features, could be integrated with the scheme. In any case, with some work, it seems likely that the Vaults model could be extended and applied within the distributed context.

9.3.2 Assurance Advantages of Cryptography

An ongoing problem in the development of sophisticated modern systems and security models is verifying that both the design and implementation are correct. The difficulty in obtaining such assurance is directly related to the complexity of the
code, and the broad consensus is that, the larger and more complex a piece of code, the harder it will be to audit and the more likely to contain vulnerabilities [3, 15]. A hypothesis arrived at during the design of the Vaults model was that the use of cryptography itself could have additional advantages with respect to the assurance level of a system. The first aspect of this is a natural extension of Gifford’s [127] notion of a passive protection system. That is, if a protection system does not need to constantly actively enforce security policy, then it is likely to be both simpler and more reliable. For example, intuitively, there are less ways for an attacker to breach the confidentiality of a file encrypted with a secure cipher than an unencrypted file, assuming legitimate access is obtained via an equivalent active security kernel in both instances.

However, the complexity of the kernel required to control access to encrypted objects may also be effectively less than that used in a non-cryptographic scheme. Consider two abstract security kernel implementations, one that employs cryptography and the other that does not. Assuming the total complexity of both in terms of code size is essentially the same, a critical part of the cryptographic implementation will be the routines that perform the various cryptographic algorithms required. While these routines may be complex, they are also easily verified for correctness through the application of standardised test vectors for which the corrects output are known. In effect, therefore, the use of cryptography may allow a significant part of the overall complexity of the code for enforcing security policy to be moved into a well-defined and easily tested set of routines. While additional routines will usually still be required to enforce the logic necessary to support the model, in many cases they may be both smaller and simpler than those in non-cryptographic implementations. Furthermore, the use of cryptography may mitigate the effects of any failure in the supporting logic by virtue of the inherently passive nature of the cryptographic protection system as described. While the limited scope of this study precluded exploring the validity of these hypotheses, it is believed that the potential assurance benefits of cryptographic security mechanisms warrant further examination.
9.3.3 Further Research on Attack Trees

During the use of the attack tree methodology to analyse the security of Vaults, a number of observations were made concerning potential research opportunities to extend the capabilities of this technique. For example, during the analysis, the library feature of the SECUrITree software was used extensively to create reusable collections of subtrees reflecting shared attack components. Since attacks on different aspects of the model often exhibit significant commonality, use of libraries can streamline the process of tree construction. However, a number of limitations of subtree reuse were identified during this work. In particular, there is a trade-off between making the nodes in a library sufficiently generic to facilitate reuse in a wide range of attack scenarios and ensuring that the low-level details of a specific attack are properly described. In many cases, a subtree must therefore be amended to take into account its low-level context. These changes can range from modifications to node labels, annotations and attribute values, to the addition or removal of nodes. This can lead to multiple versions of what is essentially the same subtree applied in different contexts, thereby limiting the principal benefit of library use. Future work in this area could involve the parameterisation of attack subtree libraries, analogous to the parameterisation of code modules, to maximise reusability. Such an approach would increase library reusability in a flexible manner, while incurring only a small cost in terms of additional complexity.

Another area for possible future research relating to libraries involves identifying at exactly what level in a tree a subtree should be separated out and placed in a library. It was identified during attack tree construction that, if a subtree were moved into a library too early, this increased the likelihood of problems such as undetected circularity in tree logic or a disconnect between the goals of the attack at the higher levels of the tree compared with the specific methods described at the lower levels. Therefore, consideration of general strategies and methodologies for identification of the optimal level for tree decomposition may be a valuable future research area, perhaps leading to enhanced tool support.

307
Section 8.5.2 discussed a number of ways that quantitative metrics of security could be developed based upon attack tree characteristics, which represent an important opportunity for further research. For example, intuitively, the sooner any given attack scenario becomes impossible, the more secure the overall design. By averaging the height above each leaf node that the subtree becomes impossible, it may be possible to obtain a quantitative picture of overall security. Alternatively, the security of the model in relation to specific high- and mid-level attacks could also be quantified by considering the calculated values at this point. As discussed in Section 8.5.2, it may also be possible to obtain a quantitative measure of the degree of defence-in-depth provided by the system by considering the number and distribution of AND nodes within a tree. Quantitative analysis of attack tree structure and characteristics therefore present valuable opportunities for additional research.

Finally, there may be opportunities to combine attack trees with the argument trees technique described in Section 6.2.6. While both techniques employ a tree structure for modelling security-related information, the methodologies are otherwise effectively opposite in their focus. An attack tree presents a high-level attack goal and provides a structure for elucidating the detailed methods that may be used to achieve this goal. Conversely, an argument tree describes an overall security property and supports the modelling of the requirements that must be met in order to provide this property. The two techniques are therefore complementary and further research to examine the possible benefits of combining them may be worthwhile.

9.4 Conclusion

The Vault's security model presented here leverages cryptography to provide security properties superior to those available in conventional models, while maintaining a simpler and more intuitive interface than the alternative mandatory security-based schemes. The integrated use of the cryptographic features contained within Vault's is designed to provide maximal leverage from the benefits of employing cryptography. Consequently, a design objective of the model is to maintain its
security properties, even if an attacker is able to obtain a high level of privileges on the system. To assess the ability of Vaults to achieve these goals, a large-scale attack tree analysis was performed involving the construction of trees describing hypothetical attacks available to a highly privileged attacker. After analysing these trees and pruning them by removing branches considered to be impossible, the results indicated that the Vaults model is, in fact, able to meet these objectives. Vaults therefore demonstrates the advantages in the use of cryptography as a means for achieving the enhanced levels of security required in contemporary computing environments.
Appendix A

Glossary

**active node** The node in the runtime digraph for a process that actually determines that process's behaviour.

**AN** See *active node*.

**anonymity** The security requirement that one, some or all of the identities of parties involved in a given transaction not be able to be determined by one, some or all of the other parties.

**application keys** Keys held within a vault that have a single, primary purpose and are therefore used by a specific application.

**attack scenario** Informally, a specific situation that may lead to an attack. More formally, with reference to attack trees, an attack scenario is a minimal subset of nodes from a complete tree that constitutes a valid attack according to the top-level attack goal.

**authentication** A process whereby the identity of the parties involved in a given transaction may be reliably verified.

**availability** The security requirement that the allocation of, and access to, computing resources will be ‘reasonable’ and in accordance with security
policy and expected service quality levels. It is regarded as one of the primary dimensions of security.

**CBC**  See *cipher block chaining mode*.

**CFS**  Cryptographic File System (see Section 2.5.1).

**cipher block chaining mode**  A block cipher encryption mode where the plaintext of each block is XORed against the previous ciphertext block prior to encryption. The first block in the message is XORed against a randomly selected initialisation vector, which is then prepended to the message in plaintext.

**complete tree**  An attack tree that has not been pruned. See also *pruned tree*.

**confidentiality**  The security requirement that data, either in whole or in part, may not be viewed by any unauthorised parties. Often this extends to protecting not only the data itself but any additional metadata that may reveal any information about the data itself. Sometimes referred to as secrecy, *confidentiality* is one of the dimensions of security.

**DAC**  Discretionary Access Controls. See *discretionary security*.

**dependency**  Notionally an object on which another object (generally a program or program component about to be executed) depends. More specifically it is a record in a *dependency digraph*.

**discretionary security**  Security models where users decide the access to others have to their data.

**digital rights management**  Technology used to control the use and distribution of copyrighted digital content.

**digraph**  A directed graph data structure. See also *dependency digraph*.

**digraph source**  In general, a vertex or node in a directed graph (digraph) is normally only reachable by itself and therefore has an indegree of 1 (only
a single edge leads to that vertex). In the case of dependency digraphs, a source has an indegree of 0 since these dependency digraph edges are irreflexive and do not link with themselves. Sources in dependency digraphs will generally represent an executable program since they are not a dependency of any other object; however, not all executable programs will be digraphs sources since many helper and utility applications will represent dependencies of other programs.

dependency digraph A directional graph structure used to store dependency data and to show the interrelationships between individual dependencies necessary for correct verification (see Section 5.2.1, especially p. 136).

dependency record See fingerprint record.

double match Where the matching process identifies a match in both GPUB and the user’s vault (see p. 144).

DRM See digital rights management.

ECB See electronic code book mode.

effective trust level The actual trust level assigned to a process. Note that this will normally be the process’s specified trust level (STL) but may be reduced as a result of a number of factors.

EFS Encrypting File System (see Section 2.5.1).

electronic code book mode A block cipher encryption mode where each block is individually encrypted.

EP See executing process.

ETL See effective trust level.

executing process The node in a runtime digraph corresponding to the actual native code currently executing.
FSO  See File System Object.

fingerprint  A unique representation of a specific piece of data that can be used as a method of confirming its integrity. Normally implemented through a cryptographic one-way hash function.

fingerprint record  The hash value and other associated metadata stored with regards to a specific program or object on the system that is to be verified. When discussed in the context of interrelationships between these objects, a fingerprint record may be referred to as a dependency record.

fingerprint verification  See verification.

File System Object  An object residing on secondary storage. Primarily an ordinary file but can also refer to directories and various filesystem esoterica such as sockets, FIFOs and symbolic links. The term may be used within the context of the new model to refer to any object to be protected by enhanced access controls.

Global Trusted Computing Base  Under the new model, the trusted computing base is divided into two parts. The global part pertains to those parts of the system upon whose integrity the security of the entire system depends. For example, if the kernel, login software or vault management software is compromised and replaced by an attacker with a modified version, the security of the entire system will very likely fail. See also Local Trusted Computing Base.

GPRIV  See global private vault.

GPUB  See global public vault.

global private vault  A system vault, maintained by the kernel and used to store values for which the confidentiality and integrity must be ensured. No users, including the privileged prime user, are able to directly access
items held in GPRIV with all interactions being indirect and strictly mediated by the security kernel (see Section 3.2.2 for details).

**global public vault** A system vault that stores information intended to be publicly accessible in a reliable and authentic way to all users on the system. Ordinary users may only read data from GPUB and cannot directly modify the values it stores. All access to the vault is therefore mediated by the security kernel.

**GTCB** See *Global Trust Computing Base*.

**integrity** The security requirement that data cannot be modified by unauthorised parties or be modified in an unauthorised way by those who are otherwise authorised to modify it. This also implies the result of any modification will also be internally consistent and correct. It is one of the principal dimensions of security.

**key** In general, a secret value that provides access to something; for example, a decryption key. Specifically pertaining to the model, a key refers to an access code or token kept inside a vault that may be provided to specified applications and/or utilised internally by the system kernel. It can also refer to generic sensitive items held within a vault.

**L0** The trust level assigned to a process that has not been cryptographically verified and has no access to cryptographically protected objects. This trust level may also be assigned at runtime to cryptographically verified programs if certain criteria are met that require the process’s trust level to be reduced.

**L1** The trust level assigned to a program that is to be cryptographically verified but may only access cryptographically protected objects for which it has been specifically authorised.

**L2** The trust level assigned to a program that is to be cryptographically
verified and may access all cryptographically protected objects for which the user is authorised.

L3 The trust level assigned to a program that is granted direct access to the contents of the user’s vault for administrative purposes.

Local Trusted Computing Base Under the new model, the trusted computing base is divided into two parts. The local part refers to programs that the user depends on for their own security and correspond to their own security requirements. It does not include parts of the Global Trusted Computing Base on which all users’ security depends. For example, the user may have keys in their vault that are bound to specific applications, such as their PGP passphrase, which is bound to the PGP program they use. Should the PGP program be modified to leak user’s keys to an attacker, the user’s security may be compromised. Hence the user may fingerprint this application, thus including it in their own LTCB.

LTCB See Local Trusted Computing Base.

MAC Mandatory Access Controls. See mandatory security.

mandatory security A property of some security models where access to data is decided based on a systemwide policy, rather than individual user’s discretion.

manual update race condition attack An attack on the integrity of fingerprints for trusted programs identified in the attack tree analysis. The attack involves substituting malicious code for a trusted program that has recently been upgraded prior to the fingerprint of the new program being calculated and manually updated.

MLS See multi-level security.

matching The process of attempting to identify whether a program about to be executed has a fingerprint that corresponds to it (see Section 5.1.2).
MUA  Mail User Agent. The piece of software that provides the end-user with access to an electronic mail system. That is, the software allows the user to receive, read, write and send email.

**multi-level security** The ability for data with different sensitivity labels to coexist securely on a single computer system.

**MURCA** See *manual update race condition attack*.

**network file system** A protocol for accessing files from a different computer system across a network in a largely transparent way.

**NFS** See *network file system*.

**non-repudiation** The technical ability and goal of preventing the sender of a message from later claiming the message did not originate from them.

**parent ETL** In the credentials metadata associated with each process, this field specifies the ETL of the parent of that process.

**parent VID** In the credentials metadata associated with each process, this field specifies the VID of the parent of that process.

**partial match** A possible outcome of the matching process that occurs when there is evidence to suggest a fingerprinted program exists corresponding to that being executed, but it is not clear that the fingerprint is intended for this program. This essentially corresponds to a situation where the name of the file being executed matches that of a fingerprint, however, the fully-qualified path names do not match and the normal response to this is to deny execution (see Figure 5.1).

**PETL** See *parent ETL*.

**PKI** See *Public Key Infrastructure*.

**primary ticket** A ticket possessed by the subject who has protected a given object. Possession of a primary ticket allows the subject to grant access
to other subjects (via a secondary ticket), to revoke access previously granted and to remove protection from an object.

**principal** An entity participating in an authentication protocol. Typically the protocol will result in the principals involved having mutually authenticated.

**process trust level** Under Vaults, each process is assigned a trust level labelled from L0 to L3, depending upon the information that relates to this program as stored in the vaults. To some degree, a process’s trust level dictates what it may do in regards to its interaction with the user’s vault and the protected items managed within it (see Section 3.4.2 for more details).

**protected** A term pertaining to the new model designating the use of an enhanced access control on an object for either read protection through encryption or write protection using a MAC.

**protection status** The status of a file with respect to whether or not the file has been cryptographically protected by its owner (see Section 4.5 for further details).

**protection status authenticator** A piece of filesystem metadata, stored for every file on the system, that is used by the kernel to securely determine the protection status of that file (see Section 4.5 for further details).

**protection status index** A sequential number in the protection status table used to identify and look up protection status information for each file on the system (see Section 4.5 for further details).

**protection status table** The tabular data structure held securely by the kernel in GPRIV that contains protection status information for each file on the system (see Section 4.5 for further details).
**protection mode** Whether a filesystem object is granted read or write protection.

**pruned tree** An attack tree that has had a set of pruning criteria applied to it to remove all nodes not valid according to those criteria. See also *complete tree*.

**pruning** The process of removing impossible attack scenarios from an attack tree to leave only those valid for the specified criteria.

**PSA** See *protection status authenticator*.

**PSI** See *protection status index*.

**PST** See *protection status table*.

**Public Key Infrastructure** A system whereby public keys may be authenticated as actually belonging to the party named on the key.

**PVID** See *parent VID*.

**revocation** The annulment of a specific privilege in an access control model. It may also refer to the cancellation of a token such as a certificate used to authenticate a party.

**RDVM** See *runtime dependency verification model*.

**Runtime Dependency Verification Model** The process by which dependencies of a program are verified essentially in an on-demand manner at runtime, taking into account differences between dependency types (see Section 5.4).

**runtime digraph** The merged digraph consisting of dependencies from both the user’s vault and also GPUB (local and global TCBs) that applies to an application throughout its execution lifetime for the verification of its dependencies.
**recipient** In an exchange of messages, the recipient receives the first message.

**scenario** See *attack scenario*.

**secondary ticket** The class of ticket granted to a subject by the owner of a protected object who wishes to grant that subject access to that object.

**secrecy** See *confidentiality*.

**sender** In an exchange of messages, the sender initiates the exchange.

**set UID** In Unix, a program that always executes with a specific, fixed UID regardless of the UID of the user who runs it.

**specified trust level** The trust level specified for a program in its fingerprint record.

**STL** See *specified trust level*.

**TCB** See *trusted computing base*.

**TCFS** Transparent Cryptographic File System (see Section 2.5.1).

**target** A fully-qualified path name, including file name, that completely describes of filesystem entry to which a specific dependency record applies. Since a single filesystem object may be referred to by several different names, each dependency record may have more than one target to which it applies. The term is also used in reference to the process of matching an object against a specific dependency record (see Sections 5.2.1 and 5.2.4).

**ticket** An unforgeable token that grants the user access to a specific item.

**ticket ID** A unique identifier of a secondary ticket used to indicate whether the ticket is still valid or has been revoked (see Section 4.3.3).

**ticket class** Whether the ticket concerned is a primary or secondary ticket. The ticket class therefore reflects the relationship that the ticket holder has with the designated protected object.
ticket mode The mode of access granted by possession of a ticket, being either read or write access.

TID See ticket ID.

TPM See trusted platform module.

Trojan horse A program that appears to one that performs some legitimate function but in fact performs some additional, covert and malicious action.

trusted computing base The collection of system components responsible for enforcing security policy that must be protected and remain unaltered for policy to be enforced correctly.

trusted fingerprint A one-way hash of a file or application believed to be in some legitimate and unaltered form. The fingerprint is then stored in a vault.

trust level See process trust level.

trusted path A guaranteed means through which a user can interact directly with a system’s trusted computing base such that they may be sure they are not interacting with some malicious program such as a Trojan Horse.

trusted platform module A hardware component of commodity computer systems that serves as the foundational component of the Trusted Computing Group (TCG) security architecture (see Section 2.6.4).

trusted process A term that may be generally used to refer to some process that has been subject to some form of validation or assurance and/or to one that plays a critical part in some aspect of security. However, the term is used to refer specifically to a process which is executing at a trust level of L1 or above and therefore has access to the user’s vault. See also untrusted process.

trust elevation A term developed to refer to the trust relationships between a process and objects that it deals with, most particularly in relation to
the trust relationship between a parent and child process. This becomes problematic when a parent process spawns a child that will execute at a higher privilege level; the trust elevation problem (see Section 5.6.2).

**UID** See *user ID*.

**untrusted process** A process with a trust level of L0 and which therefore does not have access to the user’s vault. See Section 3.4.2. See also *trusted process*.

**user ID** Identity-based label associated with a user and their processes.

**vault** A structure used by the Vaults model for the storage of sensitive and secret data. Access to a given repository is carefully restricted.

**vault ID** A value constructed by hashing a user’s public key and which is used to identify a particular user vault and consequently the set of privileges associated with this vault.

**verification** The process of ensuring that the data corresponding to a previously matched program and its dependencies has not been modified based upon comparing trusted fingerprint values with those values that apply to the data in its current state (see Section 5.1.2).

**VID** See *vault ID*.

**VPKI** Vaults Public Key Infrastructure. The PKI utilised by the Vaults architecture.
Appendix B

List of Symbols Used in Equations

\( H(x) \)  The application of a one-way hash function to some value \( x \).

\( i_f \)  The inode value corresponding to file \( f \).

\( K(x) \)  The symmetric encryption of some value \( x \) with a given key \( K \).

\( K_f \)  The protection key corresponding to file \( f \).

\( K_{fr} \)  The read protection key for file \( f \).

\( K_{fw} \)  The write protection key for file \( f \).

\( KID_f \)  The Key ID corresponding to file \( f \).

\( KID_{fr} \)  The Key ID corresponding to the read protection key for file \( f \).

\( KID_{fw} \)  The Key ID corresponding to the write protection key for file \( f \).

\( K_{ps} \)  The protection status key.

\( K_{T_f} \)  The ticket key used to construct a secondary ticket for recipient \( r \) to access file \( f \).

\( K_V \)  A key used by a specific user for encryption and decryption of their vault \( V \).

\( K_V^F \)  The Fundamental Key for system vault \( V \).
\( MAC(x) \) The application of a message authentication code (MAC) to some value \( x \).

\( n_f \) The number of links corresponding to the inode for file \( f \).

\( P \) The public key belonging to a specific user.

\( P(x) \) The encryption of some value \( x \) with public key \( P \).

\( p \) The private key belonging to a specific user.

\( p_f \) The path of a specific directory entry for file \( f \).

\( p_f^1 \) The path of the first directory entry for file \( f \).

\( p_f^n \) The path of the \( n^{th} \) directory entry for file \( f \).

\( PSA_f \) A Protection Status Authenticator (PSA) value corresponding to a given path for file \( f \).

\( PSA_f^1 \) The PSA value corresponding to the path of the first directory entry for file \( f \).

\( PSA_f^n \) The PSA value corresponding to the path of the \( n^{th} \) directory entry for file \( f \).

\( PSI_f \) The Protection Status Index (PSI) for file \( f \).

\( t_1 \) The protection timestamp for a given file.

\( t_2 \) The timestamp for the creation of a specific secondary ticket for a given file.

\( t_E \) The time at which a specific secondary ticket expires.

\( T_f^1 \) The primary ticket for access to file \( f \).

\( T_f^2 \) The secondary ticket for access to file \( f \).

\( T_{fn}^i \) The primary ticket granting read access to file \( f \).
$T_{fw}^1$ The primary ticket granting write access to file $f$.

$TID_f^r$ The ticket ID corresponding to the secondary ticket constructed for recipient $r$ to access file $f$.

t$_{PSf}$ The protection timestamp corresponding to file $f$.

$T_V^f$ A fundamental ticket to access vault $V$.

$V$ A given vault.

$VID$ The vault ID of a given user.

$VID_o$ The vault ID of the owner of a given file.

$VID_p$ The vault ID of the prime user.

$VID_r$ The vault ID of the recipient of a secondary ticket for a given file.

$z$ A randomly selected value.

$\oplus$ The exclusive-OR (XOR) operation.
Appendix C

Attack Tree CD-ROM Guide

C.1 Directories

Confidentiality Property Attack tree for analysing the Confidentiality property.

Integrity Property Attack tree for analysing the Verifiable Integrity property.

Trusted Fingerprinting Attack trees for analysing the Trusted Fingerprinting mechanism.

Libraries Complete set of attack tree libraries used by the three trees.

C.2 File Formats

Trees are presented in four different file formats; some that are native to the Security software and others that may be viewed in freely available programs. The file formats used are summarised in Table C.1.
<table>
<thead>
<tr>
<th>Extension</th>
<th>Name</th>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.rit</td>
<td>Risk Tree</td>
<td>SECUROTREE</td>
<td>Native attack tree file format of SECUROTREE.</td>
</tr>
<tr>
<td>.ril</td>
<td>Risk Library</td>
<td>SECUROTREE</td>
<td>Native attack library file format of SECUROTREE.</td>
</tr>
<tr>
<td>.svg</td>
<td>Scalable Vector Graphics</td>
<td>Inkscape</td>
<td>Platform-independent vector graphics format.</td>
</tr>
<tr>
<td>.txt</td>
<td>ASCII Text</td>
<td>Notepad/vim</td>
<td>Textual representation of the attack tree.</td>
</tr>
<tr>
<td>.csv</td>
<td>Comma Separated Values</td>
<td>Excel</td>
<td>Format showing full details for each node, including notes.</td>
</tr>
</tbody>
</table>

Table C.1: File formats of attack tree presented on the CD-ROM
Appendix D

Protection Status Verification Routines

The following are routines used internally by the security kernel as part of the protection status verification process when performing various operations on files.

CalcPSI Calculates the PSI value for the file being accessed according to the PSA and other parameters stored in the inode metadata as described in the Section 4.5. This routine underpins all the others involved in protection status verification tasks.

CalcPSA Generates a new PSA value from supplied parameters as described in Equation 4.6.

GetPS Used when accessing a file in order to obtain its protection status. This routine signals the verified protection status of the file or otherwise flags an integrity violation and aborts the access process. The routine involves the following steps:

1. Obtain the PSI value from CalcPSI.

2. Retrieve protection status from the PST using the PSI. If no PST entry is found, this indicates an integrity violation and the process is aborted.
3. Confirm timestamp values $t_{PSI}$ from PST and inode metadata are the same. If this is not the case, flag an integrity violation and abort.

4. Return KID values (read and write) to indicate protection status.

**PSVerifyAll** Performs the same steps as **GetPS** but performs **CalcPSI** for all PSA-path pairs for that object, verifying that the PSI values for each are identical and that there are the correct number of PSA-path pairs. Note that for the majority of files that have only one link, the effect is the same as **GetPS**.

**UpdatePS** Uses **CalcPSA** to regenerate all PSA values for a given inode with specified parameters, and updates the timestamp value $t_{PSI}$ in the PST and inode metadata.

**AddPSA** Generates a new PSA value for a new path using **CalcPSA** and then writes it to inode metadata along with the new path, updated link count and new protection status timestamp. All PSA values are also updated to reflect the new parameters and a new value for $t_{PSI}$ is changed in the PST.

**DeletePSA** Deletes the specified PSA-path pair, decrements the link count in the inode metadata for a given file, updates all remaining PSA values to reflect the new parameters, and also updates $t_{PSI}$ in the PST.

**DeletePSI** Deletes the entry for a specified PSI in the PST.
Appendix E

Matching Algorithm

The algorithm in Figure E.1 describes the process of matching a program to be executed against the dependency records in a specific digraph. For reasons of clarity, low-level details of traversing the digraph are deliberately abstracted as they are not central to the matching process. Instead, each dependency record is represented as a node in a linked list with the list of targets for each represented as an array. In practice, these nodes would be extracted by a separate routine that maximises performance by returning source node PROGRAM dependencies prior to other types when executing a program. Descriptions of the identifiers used in the algorithm are given in Table E.1.

The algorithm works by iterating through the list of dependency records, searching for a target path amongst those matching that of the program being executed. If

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thisdep</td>
<td>Pointer to current dependency being processed.</td>
</tr>
<tr>
<td>firstdep</td>
<td>Pointer to first dependency in the list to be processed.</td>
</tr>
<tr>
<td>targets</td>
<td>List of pathnames to which the dependency applies.</td>
</tr>
<tr>
<td>next</td>
<td>Pointer to the next dependency to be processed.</td>
</tr>
<tr>
<td>tindex</td>
<td>Counter variable for iterating through the list of targets.</td>
</tr>
<tr>
<td>length</td>
<td>Length of the list/array.</td>
</tr>
<tr>
<td>found</td>
<td>Boolean value indicating whether a match has occurred yet.</td>
</tr>
<tr>
<td>run</td>
<td>Data structure holding information relating to the program about to be run.</td>
</tr>
<tr>
<td>pathname</td>
<td>Field in the run data structure representing a fully qualified path and filename.</td>
</tr>
</tbody>
</table>

Table E.1: Key to identifiers in the matching algorithm

329
thisdep = firstdep
found = false
while thisdep != NULL AND NOT found
    for tindex = 0 to thisdep->targets.length
        if thisfpr->targets[tindex] == run.pathname
            found = true
            break
    if NOT found
        thisdep = thisdep->next

if found
    return thisdep
else
    return 0

Figure E.1: Fingerprint matching algorithm

such a path is found, matching has been successful and the search is halted. The routine then returns the matched dependency record for verification. However, if all dependencies are searched without a match being found, the routine returns 0 to indicate this and the program will execute as untrusted.
Bibliography


337


339


