Department of Computer Science
The University of Auckland
New Zealand

Towards an Open Trusted Computing Framework

Matthew Frederick Barrett

February 2005

Supervisor: *Clark Thomborson*

A THESIS SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

The University of Auckland

Thesis Consent Form

This thesis may be consulted for the purpose of research or private study provided that due acknowledgement is made where appropriate and that the author's permission is obtained before any material from the thesis is published.

I agree that the University of Auckland Library may make a copy of this thesis for supply to the collection of another prescribed library on request from that Library; and

1. I agree that this thesis may be photocopied for supply to any person in accordance with the provisions of Section 56 of the Copyright Act 1994.

   Or

2. This thesis may not be photocopied other than to supply a copy for the collection of another prescribed library.

   (*Strike out 1 or 2*)

Signed: . . . .

Date: . . . .

Created: 5 July 2001

Last updated: 9 August 2001

Abstract

A trusted computing framework attempts to provide high levels of assurance for general purpose computation. Trusted computing, still a maturing research field, currently provides four security primitives — attestation, sealed storage, curtained memory and secure I/O. To provide high assurance levels amongst distributed, autonomous systems, trusted computing frameworks treat a machine owner as a potential attacker.

Trusted computing frameworks are characterised by a need for their software to be closed-source. Ken Thompson's famous subverted-compiler shows that a user's trust in software tools may be considered lower when their source is not examinable.

This thesis proposes required characteristics of a community-developed trusted computing framework that enables trust in the framework through examination of the source code, while retaining assurances of security. The functionalities of a general purpose computing platform are defined, and we propose that a trusted computing framework should not restrict the usability or functionality of the general purpose platform to which it is added. Formal definitions of trusted computing primitives are given, and open problems in trusted computing research are outlined.

Trusted computing implementations are surveyed, and compared against the definitions proposed earlier. Difficulties in establishing trusted measurements of software are outlined, as well as enabling the use of shared libraries while making a meaningful statement about an application's functionality.

A security analysis of framework implementations of the Trusted Computing Group and Microsoft are given. Vulnerabilities caused by the implementation of curtained memory outside the Trusted Computing Base are discussed, and a novel attack is proposed.

We propose modifications to the Trusted Computing Group specification to enable curtained execution through integration with an architecture intended to prevent unauthorised software execution. This integration enables virtualisation of the Trusted Platform Module, and the benefits this gives are discussed.

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Clark Thomborson. I could not have imagined having a better advisor and mentor for my thesis. I gratefully thank him for his time, effort, expert help and guidance, and of course his friendship.

I would also like to thank Ellen Cram, from Microsoft, and David Safford, from IBM Research. Their correspondence with me throughout the year has been of great benefit.

I am also grateful to Richard Clayton for supplying photos of the IBM 4758.

Contents

List of Figures

List of Tables

Abbreviations

AES	Advanced Encryption Standard
AIK	Attestation Identity Key
API	Application Program Interface
AV	Attestation Vector
BOBE	Break-Once Break-Everywhere
CA	Certificate Authority
CRTM	Core Root of Trust for Measurement
DDT	Domain Definition Table
DES	Data Encryption Standard
DIT	Domain Interaction Table
DRM	Digital Rights Management
EK	Endorsement Key
ESS	Embedded Security Subsystem
F/OSS	Free/Open Source Software
IPC	Inter-Process Communication
LHS	Left-Hand Side
LILO	Linux Loader
LPC	Low-Pin Count
LSM	Linux Security Module
MAC	Message Authentication Code
NCA	Nexus Computing Agent
NGSCB	Next Generation Secure Computing Base
OSI	Open Source Initiative

OSS	Open Source Software
OTP	One-Time Pad
PCR	Platform Configuration Register
PKI	Public Key Infrastructure
POST	Power-On Self-Test
RHS	Right-Hand Side
RTM	Root of Trust for Measurement
RTR	Root of Trust for Reporting
RTS	Root of Trust for Storage
SEM	Secure Execution Mode
SK	Storage Key
SML	Stored Measurement Log
SRK	Storage Root Key
SSC	Security Support Component
TBB	Trusted Building Blocks
TC	Trusted Computing
TCB	Trusted Computing Base
TCF	Trusted Computing Framework
TCG	Trusted Computing Group
TOCTOU	Time of Check to Time of Use
TPM	Trusted Platform Module
TPM/XOM	Trusted Platform Module/Execute Only Memory
TSP	Trusted Service Provider
TSS	Trusted Software Stack
TTP	Trusted Third Party
XOM	Execute Only Memory

"Begin at the beginning," the King said, gravely, "and go on till you come to the end: then stop."

Lewis Carroll

1 Introduction

1.1 Background

Trusted computing is a relatively new approach to computer security that allows secure applications to be built on standard hardware and operating system architectures. It is intended to allow specific applications to be given increased security properties, without requiring significant modifications to the underlying hardware and operating system currently in use. Trusted computing adds some hardware-enforced immutable functionality and secure storage to implement a number of security primitives. This additional functionality is enabled through the addition of a chip to a standard PC motherboard.

Trusted computing aims to provide *assured* operation of applications both locally, and on remote platforms, under the control of a possibly malicious administrator. A limited set of functionality contained within a *Trusted Computing Base* (TCB) is assumed to operate correctly. This functionality is used to assure the state of a computer to a remote party, enabling them to trust arbitrary computation performed on that computer. It also enforces a number of security features locally, to protect an application and its data from a wide range of software attacks.

We consider a *Trusted Computing Framework* (TCF) to be a collection of both software and hardware that implements and enforces trusted computing primitives. It is assured by the assumption in the correctness of the Trusted Computing Base, and trusted through the correctness of software which has those restrictions and controls enforced upon it. The Trusted Computing Base is the technical *root of trust* from which trust in the correct operation of the TCF flows.

1.2 Trusted Computing Threat Model

Before continuing further, this section outlines the threat model of trusted computing discussed in this thesis.

The trusted computing primitives outlined in section 2.3 are not intended to protect against any form of physical attack against the platform on which they run. Safford explains this when discussing the Trusted Computing Group's implementation (Section 3.2) [50]:

> [The] chip sits on... [an] easily monitored [bus]. The chip is not defended against

Class	*Layer*	*Explanation*
1.	l_2	A malicious application or normal user.
2.	l_1	A malicious operating system or administrative user.
3.	l_0	A malicious hardware device or hardware-modifying or hardware-snooping user.

Table 1.1: Classes of attackers.

> power analysis, RF analysis or timing analysis. The bottom line is that the physical owner of the machine could easily recover any ... secrets from the chip.
>
> ...we simply are not concerned with threats based on the user attacking the chip...

Trusted computing is intended to secure consumer and corporate desktop PCs. The threat model is not intended to guarantee any form of availability of data or service. This is because the most naive attacker can easily succeed in preventing a PC from operating. Safford states that physically owning the machine enables a user to “easily” recover any secrets from the chip. Recovering secrets from the chip does require an attacker to physically read the secrets from the bus on the motherboard. It is not clear how “easy” this form of attack is. Certainly it would require a technically advanced user. Less advanced attacks, such as resetting the BIOS by removing its backup battery, should not allow an attacker to cause the trusted computing chip to leak any secrets.

The trusted computing threat model is primarily concerned with protecting against software attacks from malicious users and malicious system administrators, as well as applications and operating systems. There are three main classes of attackers. These are shown in Table 1.1. The layer indicates the level that the software or user executes in the system model introduced in Section 2.3.1 on page 25. Each class of attacker includes both a user and the associated software attack they are able to perform. A normal user is one who can only install user-level applications such as Adobe Acrobat. An administrative user is one who is able to install arbitrary kernel drivers or modify or affect the configuration of the operating system in critical ways. A hardware-modifying attacker represents a technically advanced user that trusted computing is not intended to protect against.

One of the contentious issues surrounding trusted computing is the inclusion of the system administrator, or owner, in the threat model. As will be discussed in Section 2.3 on page 23, trusted computing primitives are intended to protect against attacks by a malicious system administrator in order to ensure the confidentiality and integrity of certain data. Schoen of the Electronic Frontier Foundation proposes a change in the threat model of trusted computing [58]. His suggestion, known as *owner-override*, would allow the system administrator to force the trusted computing framework to generate false statements when they are not prepared for a truthful statement to be presented. These statements, known as attestations, are a critical part of the security of a trusted computing framework. Attestation is introduced in Section 2.3.5, and reasoning given there makes clear why Shoen's owner-override feature is unworkable in the context of trusted computing primitives.

As mentioned above, the trusted computing threat model is not concerned with guaranteeing availability. It is required, however, to guarantee the confidentiality and integrity in the face of attackers in class 1 and 2 in Table 1.1. The threat model can be considered to require trusted computing frameworks to *fail-safe*. That is, when presented with a given attack, data protected by a trusted computing primitive can fail to be available, but must never be released. The threat model is this way primarily due to cost considerations. Availability is typically far more expensive to secure than confidentiality and integrity, and trusted computing is intended to operate on consumer and corporate desktops where availability may be impossible to guarantee.

1.3 Motivation

As outlined in Section 1.2, a user is forced to rely on a Trusted Computing Framework to control access restrictions to her data and applications in a manner that she cannot influence. Given the assumption of the immutability of the security primitives it provides, this thesis examines various trusted computing frameworks for their ability to enable a user to properly consider the framework as a root of trust, through enabling examination of the source code by herself or another party she trusts, while still securely enforcing the assured primitives. In general, we propose a definition of *openness* that allows a trusted computing framework to be developed and examined by an open community. It is this community that then forms the social *root of* *trust* which allows a user to trust the operation of the Trusted Computing Framework will be correct.

Trusted computing frameworks have a technical *root of trust* that is implemented in hardware for a number of reasons. Firstly, functionality that implemented in hardware is more difficult to modify than the equivalent functionality implemented in software. Indeed, given the architecture of a standard PC, there are difficulties in guaranteeing the immutability of some piece of software without requiring substantial changes to that underlying hardware and software design that motivated trusted computing initially.

Secondly is the difficultly, or impossibility [19], of hiding secrets in software. Trusted computing frameworks have, at the heart, some cryptographic secret used to assure remote parties of their validity, as well as to keep local data secure from modification.

This thesis also examines the ability of proposed trusted computing frameworks to properly enforce their security primitives, yet still retain the usability and functionality of the operating system and platform to which they are added. As trusted computing research is still a developing field, we propose definitions of the security primitives that it aims to provide, and compare existing implementations against these definitions.

Given the relative immaturity of trusted computing research, this thesis attempts to outline some of the open problems that must be solved before an open, general purpose, trusted computing framework can be implemented.

1.4 Organisation

Chapter 2 proposes definitions of the terms *open* and *general purpose*, as well as giving technical definitions for the security primitives that trusted computing intends to provide. Chapter 3 surveys a number of trusted computing implementations, describing frameworks specified by industry, as well as academic research. Chapter 4 discusses the frameworks outlined in Chapter 3, and compares them against the definitions given in Chapter 2. Chapter 5 proposes some architectural changes to improve the security and usability of the trusted computing framework discussed in Chapter 3. Chapter 6 summarises our conclusions, and discusses future work.

For secrets are edged tools,

And must be kept from children and from fools.

John Dryden

2 Defining an Open, General Purpose, Trusted Computing Platform

2.1 Open

Of the three attributes this thesis aims to describe, *openness* is the most nebulous, and the most contentious. To ascertain if a trusted computing framework is open, the term must first be defined for our specific purpose. As discussed in Chapter 1, this thesis concerns itself with the development of an open trusted computing platform. Current trusted computing platforms, and why they do not meet the definition of open, described below, are discussed in Chapter 3. The relative merits of an open or closed security primitive are beyond the scope of this thesis. This thesis considers an open security framework to be worth investigating. Various properties required for open, community-developed code to be able to engender trust are proposed. The ability of naive users to rely on a closed proprietary entity, as opposed to an open community, is considered.

Section 2.1.1 surveys a number of sources to find requirements for openness, especially as it affects security. Section 2.1.2 surveys the processes through which open source software is developed and examined, improving the security and correctness of the code. Section 2.1.3 surveys literature regarding the make-up and motivations of a community that is able to function as a root of trust. Section 2.1.4 proposes some ways in which the community must operate to facilitate trust in the framework. Section 2.1.5 compares the open framework that results from our requirements to one developed in a closed, proprietary manner.

2.1.1 Requirements of Openness

In order to limit the scope of our definition of open, we concern ourselves primarily with those attributes which influence the assertion made by Thompson in his 1983 Turing Award Lecture [65]:

> You can’t trust code that you did not totally create yourself... No amount of source-level verification or scrutiny will protect you from using untrusted code.

Thompson’s assertion states that unless you had a hand in the writing of every part of your software environment, you cannot trust any code you compile inside that environment. He proves his case by subverting the C compiler to insert a back door into every copy of the Unix `login` program that it compiles. Examination of the `login` code itself will not reveal the back door.

While Thompson's assertion is demonstrably true, it is equally true that developing a useful software environment yourself is entirely impractical, if not impossible, over the course of a lifetime. An argument against such an enterprise could be that, for the majority of computing tasks, the level of trust that would be obtained is not required. A relatively lower level of trust can be obtained by examining the source code, in its entirety, of a given software environment.

Bruce Schneier, a noted computer security expert, is of the opinion that computer security comes from transparency, allowing public examination of source code [57]:

> Security in computer systems comes from transparency – open systems that pass public scrutiny – and not secrecy.

He also asserts that those responsible for engineering security products, and who wish to develop strong security products, should require transparency and availability of code [56]:

> ...the only way to tell a good security protocol from a broken one is to have experts evaluate it...
>
> The exact same reasoning leads any smart security engineer to demand open source code for anything related to security.

The source and binary representations of software are legally protected through copyright. Only the copyright owner is entitled to make further copies of a piece of software, and in order to sell a program, the end user is typically granted a *license*. The terms *open source* and *closed source* typically describe two opposing methods of software development. However, they are also used to classify the license under which a piece of software can be distributed.

The Open Source Initiative (OSI) [5] was set up to vet and approve licenses which could be referred to as 'open source,' or more specifically *OSI Certified Open Source Software*. Currently, the OSI website lists over 50 licenses which are able to refer to themselves as open source licenses. The ten criteria which a license must meet to be considered open source by the OSI

Requirement		*Explanation*
1.	Free Redistribution	The license shall not require a royalty or fee, not prevent the software being given away for free or being sold, as part of an aggregate software distribution.
2.	**Source Code**	The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicised means of obtaining the source code for no more than a reasonable reproduction cost, preferably downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.
3.	Derived Works	The license must allow modifications and derived works.
4.	**Integrity of The Author's Source Code**	The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code, for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.
5.	**No Discrimination Against Persons or Groups**	The license must not discriminate against any person or group of persons.
6.	**No Discrimination Against Fields of Endeavour**	The license must not restrict anyone from making use of the program in a specific field of endeavour.
7.	**Distribution of License**	The rights attached to the program must apply to all users.
8.	License Must Not Be Specific to a Product	The rights attached to the program must not depend on the program being part of a particular software distribution.
9.	**License Must Not Restrict Other Software**	The license must not place restrictions on other software that is distributed along with the licensed software.
10.	License Must Be Technology-Neutral	No provision of the license may be predicated on any individual technology or style of interface.

Table 2.1: Criteria of open source licenses, as specified by the Open Source Initiative, reproduced from [5]. Requirements in bold are proposed to be relevant to the security of the trusted computing framework, and are adopted by our definition.

are adapted and shown in Table 2.1. The OSI criteria require open source licenses to give all possible users (requirements 5 and 6) a wide range of rights. Requirement 1 stipulates that open source software can be sold, or given away for free. Requirement 3 stipulates that users of Open Source Software (OSS) must be able to modify the software, and distribute their modifications. These two requirements match the traditional concepts of *openness* when discussing software.

We are concerned with restricting a definition of *open* to those criteria which are relevant from the perspective of security, and especially a user's ability to obtain and view the source code. The majority of the ten criteria listed by the OSI do little to affect this goal.

To this end, we adopt requirement 2 in Table 2.1 as a requirement for an *open* trusted computing framework. Source-code of the framework must be available for inspection by all users. Additionally, we adopt requirement 4 in Table 2.1 as a requirement. This means software distributed with the trusted computing framework comes with a 'certificate of authenticity' as being the work of the stated author. Requirements for identifying authors are discussed in Section 2.1.3.

Requirement 1 is not included in our definition of open as it does not affect the availability of source code for viewing, nor its authenticity. Requirement 3 allows dilution of the appearance of there being an official, trusted version of the framework. It is not adopted for this reason. We discuss this issue at greater length in Section 2.1.4. Also, in Section 2.1.3 we discuss how trust can be developed for a single, official, version of an open source operating system.

Requirements 5, 6, and 7 allow the equal distribution and use of the software by any and all parties. These requirements are also adopted for our security-focused definition. They ensure that all groups are able to use and examine the framework, without restriction imposed by an authority who may be concerned with preventing the examination of the framework by some groups.

Requirement 8 stipulates that an individual program cannot be required to only be distributed with a specific distribution. Our trusted computing framework should always be distributed wholly, never in part, as this may result in weakened security.

Requirement 9 states that use of the trusted computing framework should not preclude or prevent the use of any other software. This prevents the security of the product from depending on the lack of certain software. For this reason, requirement 9 is adopted by our definition of open.

Requirement 10 prevents the software from being dependent on a specific technology or interface. For a trusted computing framework to give satisfactory security assurances, it may need to be implemented or run on specific implementations of hardware, as discussed in Section 1.3. For this reason, we specifically do not include requirement 10 in our definition — the security of the framework will depend on its use with specific hardware devices and technologies.

2.1.2 Examination of Source

Criteria 3 (Table 2.1), allowing derivations and modifications to be distributed, encourages the traditional development model of OSS. This model of development, in part, contributes to the contention about the security of OSS. The OSS development model is thoughtfully explained and justified by Eric Steven Raymond in his book *The Cathedral and the Bazaar* [49]. The cathedral and bazaar models of software development are discussed below, as well as the implications for security and trust in the code they generate.

Many of the central tenets of, and justifications for, the OSS development model are explained by Raymond in his book. One of them he dubs *Linus' Law*. Linus Torvalds is the creator of the Linux operating system, one of the most well-known and successful open source software projects. Linus' Law is defined as *Given enough eyeballs, all bugs are shallow*. More formally Raymond says that:

> Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.

Linus' Law is not immediately concerned with the security of a given piece of code. Instead, it refers to the overall number of bugs, and the speed with which they are found and resolved. The open source community does argue, however, that the *many-eyes* principle means security holes are found more quickly in an open source product than in a closed source product. Closed source advocates argue in return that open source code gives *black hats* the ability to study the source of a program to find vulnerabilities and exploit them, before *white hats* can find and fix them. The strength of these arguments is outside the scope of this thesis, and will not be discussed in depth here.

Linus' Law is given here to show where some of the perceived value of an open source product is derived from. The argument described here is best applied to a popular product, with relatively 'many-eyes'. For the purposes of discussion, the Linux kernel will be used as a case in point.

Just as it is impractical to develop an entire software environment on your own, it is also impractical to study and understand the source code of one developed by others. For a naive user, the Linux kernel is perceived to be secure, correct, and not contain malicious code based on the consensus of the community that develops and examines it.

A naive user may not have either the time or expertise to fully understand the source code in the Linux kernel. She instead relies on the combined technical expertise of the development community, and trusts them to be actively looking for, reporting, and fixing bugs.

2.1.3 Community as Root of Trust

Linus' Law shows why open source software gains value and quality. For an open trusted computing framework, it is envisioned that *trust*, as Thompson intended it to mean [65], would flow from the community that develops, examines, and vets the source code. This leverages the many-eyes principal discussed above, but also gives another significant advantage to the closed frameworks discussed in Chapter 3. If a user so chooses, they are able to move from relying on many-eyes to relying on their eyes. A naive user however, requires a successful *open* community to show that its interests are similarly aligned with their own. Such interests include developing and assuring a trusted computing framework that operates without malice or deception towards a user. Our inclusion of requirements 5—7 from Table 2.1 allows all groups and individuals with an interest, to examine the framework. Examination by a group whose motivations a naive user considers similar to her own allow her to avoid the infeasibility of examining the source code herself. This is an examination by proxy.

The title of Raymond's book, *The Cathedral and the Bazaar* [49], refers to two different models of open source development. Although it is common to attribute the cathedral to the closed, proprietary development model and the bazaar to the open source model, both Raymond's analogies refer to open source development. In the cathedral model, source code is available to the public only with official releases. Between releases, development goes on behind closed doors with only an exclusive, and approved, set of developers being involved.

The bazaar model differs in that the project is developed in full view of the public. Anyone is able to view and use the latest version of the source code, typically obtained over the Internet. Motivated by Thompson [65], we aim to maximise the possibility of source code scrutiny, and so require an open trusted computing framework to operate under the bazaar model of software development.

Additionally we require that the development community be composed of many groups, with diverse interests. They should also be security-conscious. Such a community is capable of serving as trustworthy 'other-eyes' for examination by proxy for a wide range of groups and users with differing security requirements.

For the naive user described above to be able to consider the open source development community as a suitable *root of trust* (see Section 1.3), that community, and the way in which it interacts with the project's code, must be defined. From the variables given in Table 2.2 on the following page, and their effect on the development community and processes, we will derive further requirements for our definitions of open.

There is no one open source development model. Each community comes with its own variations. The exact development model used by an open source project varies along a number of axes. A number of these are described by Gacek [27], and shown in an abbreviated form in Table 2.2 on the next page. As discussed above, the development model and administration processes can affect the security of the derived code in a number of subtle ways.

The starting point of the project is of concern to security if the community begins by adopting a code base used for another purpose. The security design of the adopted code must be examined carefully. The motivations and interests of the community members directly affects the security of the code. Lakhani and Wolf [33] say the following in regard to ascertaining the motivations of OSS developers:

Variable Characteristics
• Project starting points
• Motivation
• Size of community and code base
• Community
○ Balance of centralisation and decentralisation
○ How their meritocratic culture is implemented
○ Whether contributers are co-located or geographically distributed, and if so, to what degree
• Software development support
○ Modularity of the code
○ Visibility of the software architecture
○ Documentation and testing
○ Tool and operational support
• Process of accepting submissions
○ Choice of work area
○ Decision making process
○ Information submission and dissemination process
• Licensing

Table 2.2: Variable characteristics of open source software development models, adapted from [27].

> Another central issue in F/OSS [Free and Open Source Software] research has been the motivations of developers to participate and contribute to the creation of a public good. The effort expended is substantial. ... But there is no single dominant explanation for an individual software developer's decision to participate and contribute in a F/OSS project. Instead we have observed an interplay between extrinsic and intrinsic motivations: neither dominates or destroys the efficacy of the other. It may be that the autonomy afforded project participants in the choice of projects and roles one might play has "internalized" extrinsic motivations.

An associated requirement for a successful open trusted computing frame is that the community be of a sufficient size, and populated with a sufficient number of experts in the field. Quantifying these numbers is non-trivial and depends highly on the project field in question. It is currently an open problem in the field of open source software research.

Variable characteristics, pertaining to the community associated with an OSS project match-

Role Title	*Explanation*
Project Leader	Person who initiated the project, and responsible for vision and overall direction.
Core Member	Responsible for guiding and coordinating development. Involved for a relatively long time.
Active Developer	Regularly contribute new features and fix bugs.
Peripheral Developer	Occasionally contribute new functionality or features to the existing system.
Bug Fixer	Fix bugs discovered either by themselves or reported by other members.
Bug Reporter	Discover and report bugs, but do not fix them themselves.
Readers	Active users of the system, who invest time to understand its operation by reading source code.
Passive User	Use the system in the same way they use closed source systems.

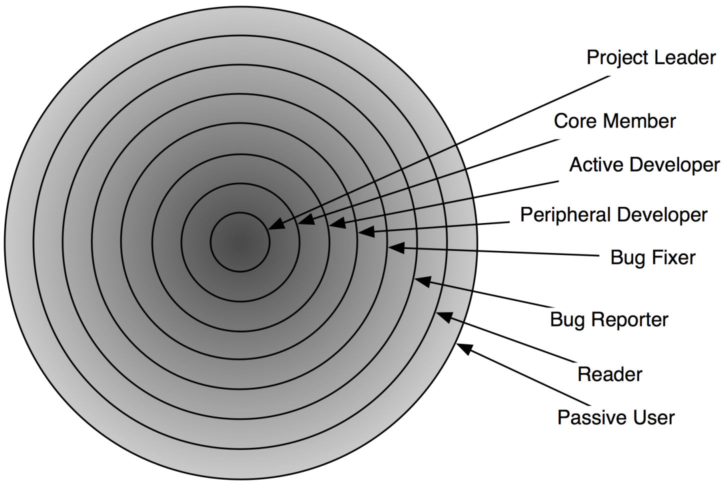
Table 2.3: Categories of open source software project contributors, adapted from [74].

ing our definition of open, involving centralisation and geographical distribution (Table 2.2) do not affect the security or trust in the derived code. The operation of the meritocratic culture however, is important. An open community relies on the contributions of its members, and an open trusted computing framework requires those members who are experts in the field to have more control over the project than others. Ye and Kishida [74] place contributors to an open source project into eight different categories, shown in Table 2.3. Not all eight types exist in all open source communities, and there are differing percentages of each type in each community. For example, they cite the Apache [1] community as consisting of 99% of passive users.

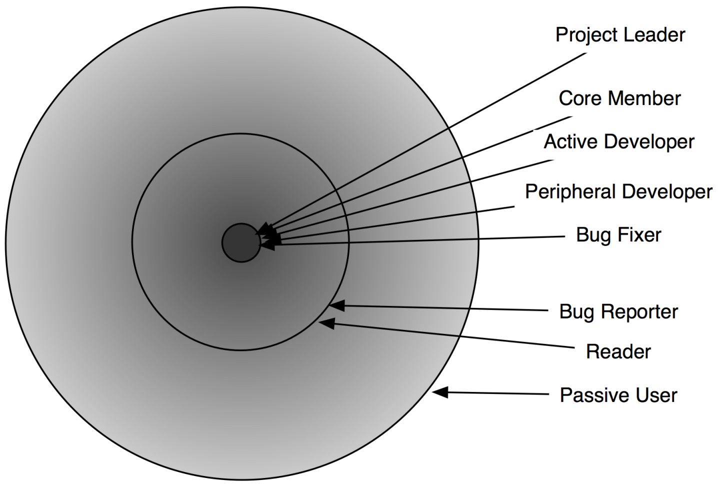
Figure 2.1, adapted from Ye and Kishida [74], shows the onion-like hierarchy of member types, and their influences. According to Ye and Kishida, the ability for an individual to effect change on the project decreases in relation to their distance from the centre.

> ...[a] role closer to the centre has a larger radius of influence... the activity of a Project Leader affects more members than that of a Core Member, who in turn has a larger influence than an Active Developer... Passive Users have the least influence...

Figure 2.1(a) shows the relationship diagram of a normalised open source community, where



(a) Normalised group sizes.



(b) Adjusted group sizes for open trusted computing framework.

Figure 2.1: Hierarchical structure of relative group sizes in open source communities, adapted from [74].

ability to effect change in the project decreases proportionally from the middle. There may be no open source project that actually fits this model of relative group sizes and influences, and it is shown here only to contrast with Figure 2.1(b).

Figure 2.1(b) show relationships for an open trusted computing framework community. Motivated by the many-eyes factor resulting in increased trust in the source code (see above), an increase is required in the roles and influences of bug reporters and readers. An individual reader, with a prior reputation as an expert in the field she is commenting on, is of disproportionate importance to the project. Such comments and bug reports from experts with little history inside the community should be given a disproportionate weighting in the project. This structure increases the ability of the passive user group to rely on the community as the root of trust described above.

Apart from the less formal description of a preferred community structure, there are also more formal criteria for an open trusted computing framework, related specifically to characteristics described by Gacek [27] (see Table 2.2 on page 14) as software development support and the processes of accepting submissions.

Requirement 4 in Table 2.1, included in our definition of open, is also intended to indicate the author of each piece of code in the framework. Anonymous check-ins of code are not allowed. While readers and bug reporters may need to be anonymous, depending on their circumstances, code must first be vetted, approved, and included in the code tree by identified and traceable individuals. Code must not be modified without clear indication of the author of the modification.

2.1.4 Operation of the Community

The project leaders and core members should be responsible for deciding when and how to issue official releases of the code base. Requirement 3 of Table 2.1 specifically requires OSS licenses to allow modification and distribution of software. Modification and distribution of the trusted framework by individuals outside of the community may result in vulnerabilities or bugs being introduced into the code. Additionally, use of such code may result in code with known vulnerabilities and exploits remaining in use, weakening the assurances given by the primitives described in Sections 2.3. An open framework that also specifies requirement 3 of Table 2.1 allows for the possibility of a fork, or unofficial versions of the framework created, to be distributed and used. To ensure the security and correctness of the framework, this should not be allowed. The attestation primitive, discussed in Section 2.3.5, allows a challenger to verify the state of a remote platform. Such verification could assure the challenger that the platform is running a correct, official version of the framework. However, other security primitives operate only locally, and are not subject to external verification. The use of a modified framework could result in those primitives being subverted by the user themselves, or by a third-party attacker.

These requirements result in the operation of our open framework community being different from traditional open source projects in a number of ways. The OSS philosophy that motivates the ten licensing criteria of the OSI (see Table 2.1) is given on their web page [5].

> When programmers can read, redistribute, and modify the source code for a piece of software, the software evolves. People improve it, people adapt it, people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing.

This philosophy encourages the modification, derivation, and distribution of open source software. The requirements for a root of trust to be formed from the community are in contrast to this philosophy. Once the trusted computing framework features (see Section 2.3) have been developed, the code base should remain static and resistant to change. Necessary changes to fix bugs and vulnerabilities must, of course, be done. But, to borrow a phrase from software engineering, a *feature freeze* must occur.

This allows naive or passive users to treat the static code base, and associated community, as the root of trust. It is difficult to imagine passive users being willing to trust an always moving code base. Additionally, from a purely technical standpoint, administering a constantly changing code base, while still giving the required security assurances, would be non-trivial. The number of different official versions in use must be kept minimal, as attestation uses *white-lists* of known code signatures. Attestation is discussed in Section 2.3.5 on page 42.

2.1.5 Comparison with Closed Development

The requirements of an open framework given here, are intended to satisfy the need for source to be available, be developed by an open community with many interests and motivations, and yet still act as the root of trust in a similar manner to closed, proprietary systems. A significant criticism of OSS is that there is no one to apportion blame or liability for faults in the code. One argument for proprietary, closed software is that the cost of the closed software includes the ability to acquire either support or financial recompense in case something goes wrong with it. It is this corporate entity which acts as the root of trust for the product. For example, IBM's secure coprocessor, discussed in Section 3.6, was developed primarily for the banking industry. IBM provides guarantees of the correctness of the manufacture and design of the device. But it is IBM's reputation, and its fiscal and legal responsibilities to its shareholders, that enable other corporate entities (banks) to have IBM as the root of trust for the device.

A corporate entity has no avenue for recompense against an open source community following some software failure. To date, however, little recompense has been obtained from major software vendors for failures in their products. The license of one major software vendor explicitly states they are not liable for any costs incurred due to failings in their software. A legal analysis of Microsoft's End-User License Agreement (EULA) is outside the scope of this thesis. It is interesting, however, to quote a relevant portion of their EULA for their Windows Server 2003, Enterprise Edition [41]:

> 15. DISCLAIMER OF WARRANTIES. ... Except for the Limited Warranty and to the maximum extent permitted by applicable law, Microsoft and its suppliers provide the Software and support services (if any) AS IS AND WITH ALL FAULTS, and hereby disclaim all other warranties and conditions, whether express, implied or statutory, including, but not limited to, any (if any) implied warranties, duties or conditions of merchantability, of fitness for a particular purpose, of reliability or availability, of accuracy or completeness of responses, of results, of workmanlike effort, of lack of viruses, and of lack of negligence, all with regard to the Software, and the provision of or failure to provide support or other services, information, software, and related content through the Software or otherwise arising out of the use of the Software.

Landwehr [36] elaborates on the arguments about software:

> It has a significant cost of design and implementation, yet very low cost of replication. Very minor changes in its physical realisation can cause major changes in system behavior. Though a great deal of money is spent on it, it is rarely sold. Usu-

ally it is licensed, customarily under terms that relieve the producer from nearly all responsibility for its correct functioning.

Depending on their situation, the fiscal and legal avenues for recompense provided by closed, proprietary systems may not be of any use to the user. Users in different countries, or with little financial resources, may be of little importance to a major software vendor unless there is some other contractual obligation between the two parties.

The value of an open community-developed framework for a passive, or naive, user is dependent on the similarity between their security requirements and those of the project leaders, core members, and developers. They may find their requirements are closer to those involved with an open framework than an alternative developed by a corporate entity. The security requirements of those in charge of the closed framework for their framework, may not dominate the development direction of that framework. Other considerations, such as a return on investment or possible liabilities, may be the motivating interests.

2.2 General Purpose

This section establishes some measures of functionality and usability that lead to a general purpose computing platform.

A general purpose computer is formally understood to be one which is Turing-complete. That is, it matches the definition of a universal Turing machine. This section discusses a less formal, higher-level, interpretation of the phrase. When declaring a given trusted computing framework to be general purpose, we are not making reference to its ability, or otherwise, to be used as a universal Turing machine. All the trusted computing frameworks discussed in Chapter 3 are implemented on hardware machines, and in software languages, that are Turing-complete.

A trusted computing framework is itself considered to be *general purpose* if it does not restrict the general purpose usability and functionality of the computing platform and operating system upon which it builds. As discussed in Section 1.3, trusted computing extends general purpose hardware and software architectures to provide assured computation for chosen applications. General purpose computer architectures in use today give the user a considerable range of functions and abilities with regards to administering their system, writing, compiling and distributing applications, and obtaining and executing arbitrary applications. It is measurements of usability and functionality like this that a trusted computing framework, when added to a platform that already enables such use, should not restrict in order to implement the security primitives discussed in Section 2.3.

Garfinkel et al [28] give examples of general purpose computing platforms such as "workstations, mainframes, PDAs and PCs." In contrast to this, they list "automated tellers, game consoles, and satellite receivers" as examples of *single,* or *special purpose* computing devices. Their simple taxonomy defines single purpose computing devices as platforms that are restricted in their usage through a limited hardware or software interface. The internal architecture and components of a single-purpose and general-purpose computing platform may not differ greatly. It is the interface, and functionality, that each presents to a user that separates them.

One of the functionalities of a general-purpose computing platform is the ability of users to write, compile, and run their own programs, listed in Table 2.3 as characteristic 1. Spinellis [61] cites the Xbox console [15] as an example of a special-purpose computing device:

> The Xbox... can be considered an instance of a special-purpose TC [Trusted Computing] platform. Although based on commodity hardware, the Xbox is designed in a way that allows only certified applications (games) to run...

The Xbox is not intended to allow a user to execute arbitrary code on it. Code must be inspected and approved by Microsoft before it can be sold to end users, and there is no avenue for the free distribution of programs to run legally on an Xbox. The inspection procedure can guarantee a certain standard in the final product, but it means the code that executes on an Xbox is not arbitrary. This restriction of arbitrary code is listed in Table 2.4, as required characteristic 2 of a general purpose computing platform.

Characteristic 3 in Table 2.4 is best understood by considering the measures taken to control a corporate computing environment. The installation of software on desktops and servers is under the control of an IT department. Users make specific requests to have software installed on their desktop. Additionally, versioning of a software application is tightly controlled.

	Characteristics
1.	Write, compile, and run own programs.
2.	Execute arbitrary programs.
3.	Control over software versioning.

Table 2.4: Capabilities of a general purpose computing platform.

Versioning is the process through which patches and updates are installed, over time incrementally keeping a software application up to date. Strict control over the software installed on a computing platform, and the ability to prevent the installation of malicious programs, or programs considered insecure, allows the corporate desktop to be made considerably secure without trusted computing primitives.

Software applications are typically upgraded through the release of patches and updates. A general purpose computing platform allows the user to manage the installation of these releases. A release denoted as a *patch* typically implies it is concerned with fixing a security vulnerability or bug. A release denoted as an *update* typically implies the addition or upgrading of features in the application. A general purpose computing platform allows the owner to apply patches and updates at their discretion. Most patches and updates are in the user's interests — they do not remove some aspect of the application the user previously found useful, but supply instead either improved security or usability.

The distinction between the *owner* and the *user* of the platform is important. A corporate desktop is owned by a different entity to its every day user. The interests of the user are only considered in relation to the ability of the user to do what the owner of the platform requires. For a home computer, the owner and the user are the same.

The requirements listed in Table 2.4 for a general purpose computing platform, enable wide and varied usage of the platform by many different groups of users. Applications that were not considered when the software and hardware platform was released can be developed, deployed, and tested. Hardware peripherals can be added to extend functionality. Upgrades can be performed to improve the performance, functionality, and usability of both software and hardware.

The security features provided by a trusted computing framework, discussed in Section 2.3, are intended to be used to secure specific applications in a manner not possible without the framework. As discussed in Section 1.3, trusted computing primitives are intended to secure certain applications where and when it is considered necessary, not improve the security of the platform as a whole.

One measure we propose of the usefulness, or general purposefulness, of a trusted computing framework is its ability to improve the security and assurance of specific applications without reducing the general purpose platform upon which it runs to a special or single-purpose one. A reduction in the general usability and functionality of a platform as a whole, to a level that would allow applications to be secured and assured to comparable levels through that reduction alone, is not especially useful.

It should be noted that trusted computing primitives do preclude the functions of a general purpose computing platform in order to provide assurances of confidentiality and integrity. For example, the sealed storage primitive (Section 2.3.5 on page 42) is intended to enforce strict limitations on programs that can be used to access specific data, and the attestation primitive may be used by a remote challenger to prevent a user from accessing a service with an arbitrary application. Reductions in functionality and usability required to assure confidentiality and integrity, are not considered reductions of the general purposefulness or usability of a platform.

2.3 Components of a Trusted Computing Framework

As discussed in Section 1.3, trusted computing uses a hardware-based component to assure a limited set of immutable functionality and a limited set of cryptographic secrets. These two functions are used as leverage to provide a considerably larger set of security functions. The four security primitives considered to make up a trusted computing framework are known as *attestation*, *curtained memory*, *sealed storage*, and*secure I/O*.

Attestation provides remote assurance of the state of the hardware and software stack running on a computer. Curtained memory provides memory separation of processes. Sealed storage provides access controls to data based on the executing software stack. Secure I/O provides assured input and output from the CPU to peripherals such as the keyboard or display.

Using a hardware device, embedded in the motherboard, to provide assurance is a relatively new approach to securing the personal computer. The motivation and reasoning for trusted computing was discussed in Chapter 1, and will not be repeated here. The four trusted computing primitives are not implementations of new concepts in computer security. They are generally evolutions of previous security features found in previous operating systems.

The term *trusted computing* used in this thesis is distinct to the term *secure computing* commonly used to describe development processes, standards, and verifications that must be applied to arrive at a 'secure' system. The United States Government Department of Defence's Orange Book [16] specifies four different levels (D–A) of what it refers to as 'trusted computer systems.' Common Criteria (CC) was introduced in 1999 to replace the ageing Orange Book specification. It specifies seven different Evaluation Assurance Levels (EAL1–7) against which products and systems can be tested.

The trusted computing primitives described in this section are not intended to meet any of the CC or Orange Book standards for secure computing. Their use by an application or system does not guarantee anything about the security of that application or system as a whole. However, trusted computing primitives can be used as part of systems or applications intending to be evaluated against CC.

For example, the attestation primitive of trusted computing allows a remote party to trust some statement made by a platform about itself. This statement is limited in its nature — "program A is (or programs A–D are) executing, under operating system X, and I am a device Y with capabilities Z, here are my credentials W to prove what I say is true."

The security of the entities named in the attestation statement is not assured in any way. The development processes, methods, and verifications applied to those programs, operating systems, and devices affect the security of those products, not the validity of the attestation statement itself.

In computer security, the method of guaranteeing the security of a software component, either through secure engineering practises using specific development methodologies throughout the software life cycle or through formal verification of the final state, is an open problem. Proving the correctness of code is non-trivial. Trusted computing does not attempt to assure the *security* of arbitrary code or a system. It only assures the correctness of certain functionality, and the validity of the contents of a limited statement made about a platform.

To the best of our knowledge, the primitives provided by a trusted computing framework have not been comprehensively formalised to date. This thesis makes the first steps toward an adequate formalisation of trusted computing. A fully developed and fully-argued formalisation is beyond the scope of this thesis. The definitions given here should be considered first drafts of further formalisations.

Section 2.3.1 will briefly introduce the system model used in this thesis. Section 2.3.2 introduces the concept of code identity. Sections 2.3.3, 2.3.4, 2.3.5, and 2.3.6 each introduce a trusted computing primitive, and place it in a historical context. Motivations for the primitive will be given, as well as some examples of the types of application-layer features that could be built with it. Classes of attacks possible against each primitive will also be discussed. Each primitive will be formally defined, and these formal definitions will be used to validate trusted computing frameworks discussed in Chapters 3 and 5.

2.3.1 Machine Model

A common model used for the discussion and development of computer security is the *access control model* [34]. Figure 2.2 shows the components of the access control model. Explanations for each of the components is given by Lampson et al [35], and reproduced below.

- *Principals* — sources for requests.
- *Request* — attempt to perform an operation with or on a resource.
- *Reference Monitor* — a guard for each object that examines each request for the object and decides whether to grant it.
- *Objects* — resources such as files, devices, or processes.

In a trusted computing framework, objects typically take the form of cryptographic keys to which the principal is requesting some form of access. This access could either require the disclosure of the key, or the decryption of some cipher text by the monitor on its behalf. The specific entities in the model will be described on a case by case basis throughout this thesis.

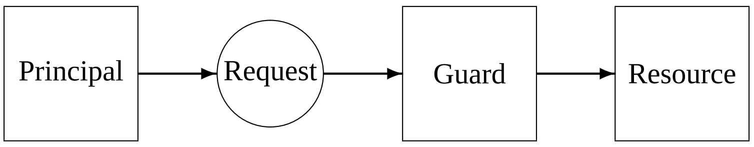


Figure 2.2: Components of the access control model, adapted from [35, 34].

A trusted computing framework operates both on the level of a single machine, and in a distributed environment. The threat model a trusted computing framework is designed to resist is discussed in Section 1.2. Trusted computing functions occur either on a single machine, or between two machines across some network. England and Peinado [26] give a usable definition of a *computing device*:

> ...the computing device is a programmable, state-based computer with a central processing unit (CPU) and memory. The memory is used to store state information (data) and transition rules (programs). The CPU executes instructions stored at a memory location, which is identified by an internal register (instruction pointer IP).

In addition to this definition, our computer device has a network interface through which it is able to communicate with other computing devices. The network channel itself is untrusted. Data that is not otherwise protected before it is transmitted, is able to be viewed or modified by attackers.

As discussed in Section 1.3, trusted computing builds its trust from the immutability property of some hardware-based device. This formation of trust leads us to consider the layers which make up a computing device, and build a model to introduce trusted computing with. England and Peinado [26] outline their model, and an adapted version is reproduced in Figure 2.3. The immutable hardware device is shown at layer l_0, the operating system kernel at layer l_1, and applications at layer l_2. Layer l_0 acts as a guard to some resource — a cryptographic function or secret. Layer l_1 acts as a principal to layer l_0, and as a guard to layer l_2. Layer l_2 acts as a principal to layer l_1. As in the access control model outlined above, a principal at layer l_i initiates requests to a guard at layer l_{i-1}. This guard/principal relation continues until layer l_0, where the resource resides, and the request is serviced.

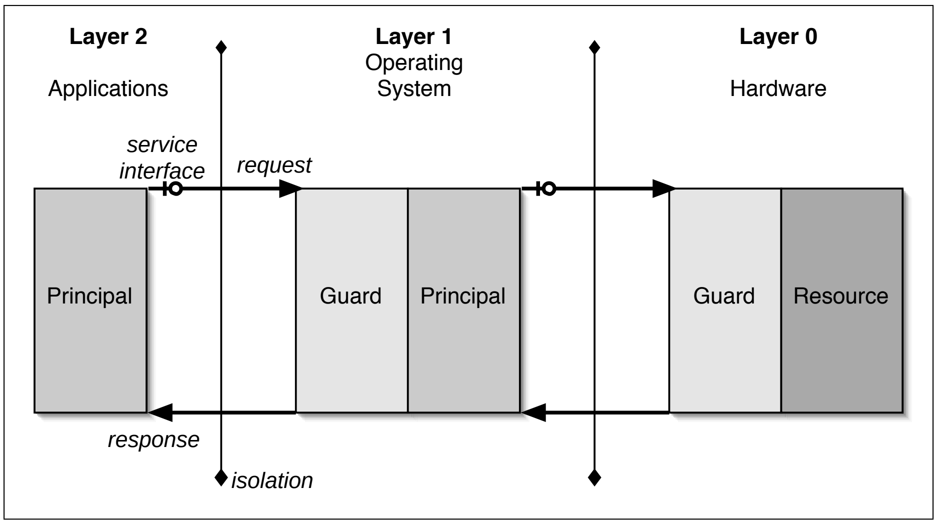


Figure 2.3: A layered computing system showing hardware, firmware, and software, adapted from [26]

The interaction between principals in differing layers is not restricted to the request and response access control model, discussed above. The relative layers in which two principals are executing also indicate their relative privileges. A principal P executing in a layer l_i has the ability to affect a principal Q in a layer l_j, where $i < j$, without requiring Q to initiate a request. Specifically, our system model requires an operating system P, executing in layer l_1 to have specific functions it can perform on a principal Q executing in layer l_2. An OS P must be able to *create* Q, by loading code from the file system into memory, marking it executable, and initiating its execution on the CPU. Once started, an OS P can *view* the state of Q throughout its execution, to ensure it operates correctly. It is also to *modify* that state arbitrarily, perhaps halting the execution of Q if it behaves maliciously or in a manner contrary to some system policy. An application Q is also capable of being *signalled*, in a well-defined manner, by P. This can inform Q of some important state-change in the system, or pass a message to Q from the user, or another application.

In our system model, such actions may occur from the point of view of Q, arbitrarily — that is without Q initiating a request. Additionally, two principals P and Q, both executing in layer l_i, are able to communicate through some mechanism set up or managed by a principal R in a layer l_j, where $j = i - 1$. This can be viewed, in our access control model, as a request from P in l_2 to access a resource provided by Q, also in layer l_2. The guard in this case is R in layer l_1.

2.3.2 Code Identity

Code identity is a pivotal concept in distributed computer systems. This is especially true from trusted computing frameworks. It is introduced here, before discussion of any specific trusted computing primitive, because it is relevant to all of them.

England et al [25] give a brief introduction to code identity when discussing Microsoft's Next-Generation Secure Computing Base. However, their introduction is not only relevant to Microsoft's product. Informally, they state that a code identity (code ID) is a "cryptographic digest or "hash" of the program executable code" [25, p 56]. England et al succinctly outline the motivation for a securely derived identity for program code [26]:

> If the operating system can guarantee file-system integrity, it can simply assume that the program "is who it says it is." However, in distributed systems or platforms that let mutually distrustful operating systems run, establishing a cryptographic identity for programs is necessary.

It is obvious that when used in a distributed system to make access control decisions, the code ID must be derived in exactly the same manner on all computing devices in the system. Additionally, there must be some way to prove to the remote platform that this is occurring. This leads us to definition 2.1 of code identity.

Definition 2.1. A trusted computing framework τ has a correct *code identity* mechanism if it derives identical cryptographic digests $ID(P)$ for a program P on all computing platforms ρ running the framework τ, and no other distinct program Q has $ID(Q) = ID(P)$ on any other ρ'.

Under definition 2.1, the use of $ID(P)$ to identify the functionality of P is only reasonable if P is not dependent on functionality provided by code not measured in $ID(P)$. This functional dependence is precisely the case when P makes use of shared, or dynamically-linked, libraries.

Shared libraries are not distributed with P, and are usually managed and updated by an entity other than the author of P. If the shared libraries called by an application are included in its code identity, updates to one shared library will change the code identities of all the applications that call it.

If the code identity of P does not include all the libraries that P calls, then the code ID $ID(P)$ is not a meaningful statement about the functionality of P. A third party cannot use this code identity to make decisions that depend on the identity and functionality of P. An identity $ID(P)$ fixed only to P does not indicate changes in functionality caused by updated shared libraries.

Generating a meaningful statement $ID(P)$ about the functionality of P, when P has a functional dependence on shared libraries or other code outside the code base of P, is an open problem in trusted computing frameworks. Various trusted computing frameworks attempt to solve the problem in different ways. These solutions are discussed in Sections 3.2.4, 3.3.3, and 3.7.3.

2.3.3 Sealed Storage

Sealed storage allows applications to persist data securely between executions, making use of traditional untrusted long term storage medium like hard drives, flash memory, or optical disks. Data is encrypted with some symmetric encryption algorithm before storing. The particular encryption scheme used depends on the implementation. Strong cryptography assures the confidentiality and integrity of data when stored on an insecure medium. The symmetric key used to encrypt and decrypt the plain and cipher text is derived from the code ID of the application requesting the cryptographic operation.

Sealed storage, also known as *secure storage*, is an evolution of traditional file system access control mechanisms. When implemented in a trusted computing framework, sealed storage allows an application to encrypt some secret of arbitrary size, and be assured only it will be able to decrypt it. To provide this assurance, the key that is used to encrypt the data cannot be specified or obtained by the application.

Early access control mechanism, such as those found in Unix-style operating systems, based decisions for allowing or restricting access to files solely on ownership and identity parameters. Which account *owned* a file was stored in a data structure associated with the file. Additional access control information specified the possible access permissions (read, write, or execute) pertaining to the owner of the file, users in the same group as the file, as well as all other users

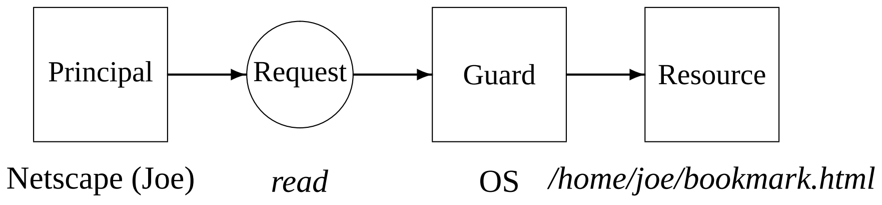


Figure 2.4: Access control model showing attempt to read a file from the file system with Unix-style access control mechanisms.

in the system. If an account was named as the owner of a file, that was given full control over the file. Full control includes the ability to specify another user account as the owner of the file, as well as change the specific access permissions associated with the file.

Each running process in the operating system runs with the *identity* of a user account on the system. User accounts are maintained by the system administrator, and most programs take the identity of the user that started the application. The system requires a user to authenticate herself to the system, establishing their user account name in the process.

When a user, or program running with that user's identity, attempts to read, write, or execute a specific file, the operating system makes an access decision based on the permissions stored with the file. This process is shown in terms of the access control model in Figure 2.4. Here, the web browser Netscape, running with Joe's privileges, is attempting to read the `bookmark.html` file from his home directory. This is a typical access request. Although not shown here, the access permissions set would be examined by the operating system. If Joe, as the owner of the file, had given himself `read` permission, the operating system would allow the request to continue.

This model shows the identity of the logged-in user, Joe, and the set of permissions on the relevant file being used as parameters to the access control decision. Programs that are intended to run without user interaction, or be accessed remotely over a network, can be run under their own user account. These programs, often network daemons, are started by the root (superuser) account, and set to execute under their own specific user account. This user account is deliberately configured to adhere to the principle of least privilege. In practise, this means that the user account is configured to allow the minimum access for the program to function as required.

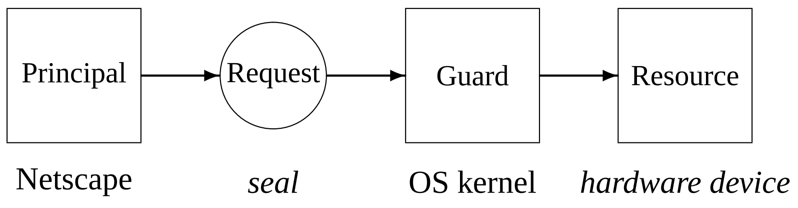


Figure 2.5: Access control model showing Netscape requesting some arbitrary data to be sealed.

An example of this configuration involves running the Unix e-mail daemon Sendmail [10] under a `sendmail` user account. This program has had a long history of security exploits which give a remote user shell-access to the server with the privileges of the user account Sendmail was running under. When run under its own user account, a successful attack against Sendmail by a remote attacker gives only a minimum privilege account. If the `sendmail` user account is configured correctly, the attacker is severely restricted in their ability to further compromise the system. If Sendmail had been run under the account it was started by (typically root), the attacker would have complete access to, and control over, the compromised system.

This technique conflates the two parameters of the traditional access control decision, account identity and file permission, into one logical parameter — the identity of the program itself. The sealed store primitive continues this trend, by binding access control decisions to the code ID of the application in question, without checking the identity of the user.

In the majority of trusted computing implementations, the sealed store function is carried out by some trusted entity at a lower layer in the system hierarchy. As explained in Section 1.3, trusted computing relies on the immutability property of some hardware device to be implicitly trusted by higher layers. From an application's perspective at layer l_2, the sealed store function is carried out by some lower layer, $l_{i<2}$. This layer simply passes on the request until it reaches layer l_0. In practise, the hardware device does not often contain all the required functionality for the primitive to be usable, and so all layers below the application work in conjunction to provide the required functionality. An equivalent effect can be also be shown to be obtained through different methods, as discussed in Sections 3.7 on page 92 and 3.4.1 on page 82. These methods do not directly map to the access control model, and operate without requiring a request to be issued to a lower layer.

An application makes use of the sealed storage primitive by making a `seal` call, passing one or two parameters. If the calling principal of `seal`, P_s, specifies only one parameter, the data to be sealed d; the lower layer generates the code identity of P_s, $ID(P_s)$, and uses it as the second parameter, referred to as the intended unsealing principal, P_u. The first case is a special subset of the second. It is equivalent to P_s calling `seal` and passing d, generating $ID(P_s)$, and including it as the second parameter. The second parameter is used to authenticate the caller of the `unseal` operation.

The `seal` function returns the data d symmetrically encrypted, denoted as c. An example is seen in Figure 2.5. Here Netscape, at layer l_2, initiates a request to some lower layer $l_{i<2}$, passing in password data, p (plain text), as a parameter to be sealed securely. Some lower layer l_i, depending on implementation, generates the code ID of Netscape and uses it as P_u. It should be noted that the guard in the model does not authenticate or authorise the requesting principal. Any principal P is able to call the `seal` function and logically the code ID, $ID(P_u)$, is used to generate the encryption key, ensuring that only P is able to decrypt the resulting cipher text c.

Layer l_0 uses $ID(P_u)$ of the intended unsealing application, to generate the symmetric key it will encrypt the passed data d with. Sealed storage assures the decryption of c will only occur on the same platform it was encrypted. The sealing platform is denoted as ρ_s and the unsealing platform denoted as ρ_u . To this end, a device-unique property is used to generate the symmetric key used to encrypt d. As discussed in Section 1.3, the hardware device at layer l_0 of the specific trusted computing framework contains a number of keys. These keys are never allowed to leave layer l_0. One of these is a device-unique symmetric key K_s. The key K_s and $ID(P)$ are concatenated together ($||$) to form a key of the form $\{K_s||ID(P)\}$, denoted as $K_{s||ID(P)}$. This key is constructed to be unique to both the specific platform and application. It is used to encrypt d. This encryption, $E_{K_{s||ID(P)}}(d)$, results in a cipher text c. This cipher text c is returned in response to the initial request from the principal, Netscape, at layer l_2. Netscape then stores c using the traditional insecure file system provided by the operating system.

In the example above, Netscape does not specify an intended unsealing application, and so its own code ID is used as $ID(P_u)$. Netscape could equally have specified another intended unsealing application, by passing its code ID as $ID(P_u)$. Netscape obtains this code ID through some traditional means of inter-process communication (IPC) — either a shared directory, or through a socket. It should be noted that there needs to be no pre-existing trust relationship between Netscape and any application it intends to be able to unseal d.

The `unseal` function takes only one parameter — the data c to be unsealed. A principal P at layer l_2 initiates a request, passing c to some lower layer $l_{i<2}$, depending on the implementation. Layer l_0 generates $ID(P)$ and uses it as $ID(P_u)$, along with the platforms symmetric key K_s, as when sealing data. The generated key, of the form $\{K_s || ID(P_u)\}$, denoted as $K_{s||ID(P)}$, is used to decrypt c. $D_{K_{s||ID(P)}}(c)$ gives d, which is returned to the calling principal P.

We have identified six cases for the operation of `unseal`.

1. The data d is returned; or

2. the decryption $D(c)$ fails, because c has been modified; or

3. the decryption $D(c)$ fails, because the calling principal of `unseal`, P_u identified by $ID(P_u)$, is not the same as the identified unsealing application specified by the calling principal of `seal`; or

4. the decryption $D(c)$ fails, because the device-unique property used to derive the symmetric key is different; or

5. data d' is returned, because the underlying cryptographic algorithms failed to operate as expected; or

6. if c has been deleted.

The occurrence of case 1 implies a number of things. The first is that c was sealed on the platform ρ_s and unsealed on ρ_u, and that $s = u$. The second is that $ID(P)$ of the unsealing principal equals $ID(P_u)$ specified as the identity of the intended unsealing application.

The occurrences of cases 2, 3, and 4 are assumed to be caused by the underlying cryptographic algorithms operating as expected. The decryption $D(c)$ should fail when the key used is not the same as used in $E(d)$ that gives c. Case 2 implies that c has been tampered with while stored on persistent storage medium. Case 3 implies that the unsealing application is not intended to unseal c. Case 4 implies that $\rho_s \neq \rho_u$.

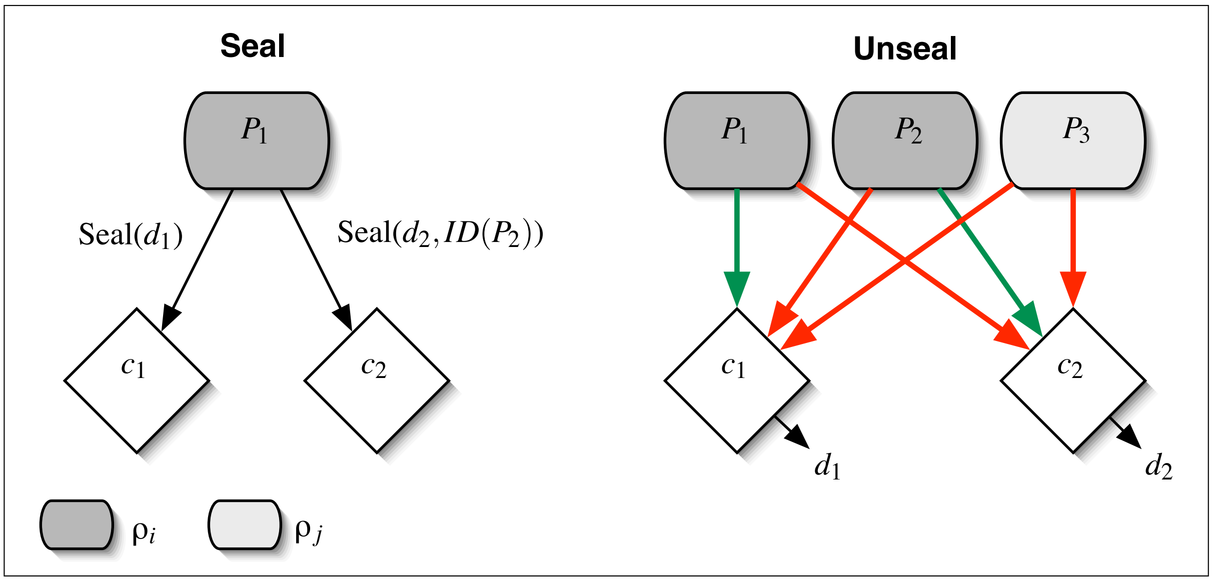


Figure 2.6: Operation of sealed store command, resultant sealed text and respective unseal command.

Case 5 occurs when a modified c' decrypts to d', through the failure of the underlying cryptography to detect a change in the integrity of d. It also occurs when $ID(P)$ of the unsealing principal does not equal ID_u, and this decryption yields a c' that is not detected as incorrect.

Case 6 is listed here for completeness; $D(c)$ will not so much fail as never begin. Cases 2 and 6 are the result of storing c on an untrusted medium. The sealed data is intended to be stored on a storage medium not under the control of either the sealing application or a trusted operating system enforcing policy restrictions set by the sealing application. Providing a secure trusted storage medium is expensive, as discussed in 1.3, but results in fewer failing cases for $D(c)$. A motivation for the design of the sealed storage primitive is to give no guarantee of data availability, and strong guarantees of confidentiality and integrity, at a greatly reduced cost compared with giving a strong guarantee of data availability.

Figure 2.6, adapted from [25], gives a graphical illustration of the seal and unseal primitives, and illustrates their typical uses. Below, we analyse this figure in terms of the six cases we identified above. Principal P_1 is shown, calling `seal` twice, both times on platform ρ_i. The first call specifies only one parameter, the data to be encrypted d_1, resulting in c_1. The second call specifies two parameters. The first is the data to be encrypted d_2, and the second is the code ID $ID(P_2)$ of the principal intended to unseal d_2. The `unseal` call is shown six times, called by each principal P_1, P_2, P_3 on each sealed data blob c_1, c_2. Principals P_1 and P_2 call unseal on

ρ_i, and principal P_3 calls unseal on ρ_j. Additionally, $ID(P_1) = ID(P_3)$. The unseal call by P_1 passing c_1 as the parameter results in $D(c_1)$ completing successfully, and d_1 being returned to P_1. The unseal call by P_1 passing c_2 results in $D(c_2)$ failing. This failure is an example of case 3. Principal P_2, calling unseal on c_1, results in a failure due to case 3, also. When principal P_2 calls unseal on c_2 however, $D(c_2)$ succeeds because P_1 specified the code ID of P_2 ($ID(P_2)$) when calling seal on c_2. Both calls by P_3 result in a failure of c_1 and c_2. These failures are examples of case 4.

Figure 2.6 shows that cases 1, 3, and 4 are the commonly-occurring cases, assuming that c is not deleted or modified. The modification or deletion of c occurs due to an attack or hardware failure. If c has been modified, all `unseal` operations in Figure 2.6 will fail due to case 2, and if c has been deleted, will fail due to case 6. The occurrence of case 5 implies a weakness being found in standard cryptographic theory.

Due to the lack of secure, persistent storage mechanisms, a number of attacks are possible under the threat model outlined in Section 1.2. The first of these is a denial of service attack (DoS), resulting in a failure due to case 1 or 4 occurring. An application relying on sealed storage to persist long-lived application state from execution to execution cannot rely on that state to be available when it next executes. A malicious operating system, application or user can delete or corrupt a sealed secret thereby forcing an application to return to its null, or initial, state on each execution.

A more subtle attack, taking advantage of the same vulnerabilities that lead to the above DoS attack, is known as a *replay attack*. To describe a replay attack, the notation c_T is introduced. Here T indicates a point in time, or an execution in an ordering of executions of P, that result in c being unsealed at P's initialisation, modified during execution, then sealed when P exits.

In a replay attack, an attacker replaces a sealed store c_n, with a copy c_i, where $i < n$, obtained from a previous execution cycle of the application. If the application relies on the sealed store c to save its state from execution to execution, that application will in effect *replay* some earlier behaviour executed when it first saw the state contained in c. A malicious attacker can perform a replay attack for a range of reasons.

One possible reason is that the attacker wishes to have P perform some action a number of times, which would result in a violation of a security policy of P. Alternatively, they may wish to obtain information, random bits of which are being leaked during each successive execution. Further more, they may wish to replay a certain execution that is dependent on some external factor they are not able to directly influence. This may involve some communication via network to a third party server, which is only rarely in the required or appropriate state.

Sealed storage enables restricted access to data to a selected set of applications. In the normal case, the selected set will have only one member — the application which created the data. But the additional case allows for interaction between and amongst sets of programs. In this case, an application P_i can generate and seal some data, and specify at the same time some other application P_j to be the only application which is able to unseal the data. At a later point, application P_j can unseal that same data, perform some other computation upon it, and then seal the data again, nominating some third application P_k as the application able to unseal the resultant data.

As long as no malicious attacker intervenes, and modifies or replaces the sealed data between P_i and P_j or P_j and P_k, the resultant data can be assured to have passed through the nominated applications in the prescribed order. This method of strictly ordering computations can be seen in strongly typed systems, and is explained further in Section 3.4.1 on page 82. It is one example of an assured application made possible by sealed storage.

Following on from the discussion of sealed storage above, Definition 2.2 is given below. This definition highlights the aspects of sealed storage discussed above. It states that an application P, in layer l_i, on platform ρ is only assured that sealed data d will remain confidential and unmodified until it is unsealed, not that it will be available to P at later execution. This is due to the failure cases outlined above. It only assures P that d is unmodified by, and kept confidential from, all other principals in layers l_i, where $i = j$, on ρ. Specifically, it is not kept confidential from the principal Q in layer l_{i-1} on ρ that performs the seal and unseal functions for P. The principal Q may or may not execute in layer l_0. It is however kept confidential from all principals in all layers not on ρ.

Definition 2.2. A trusted computing framework τ provides *sealed storage* if an application P in layer l_i with code ID $ID(P)$ on a computing device ρ implementing τ is assured of the confidentiality and integrity of data d, when sealed, from all other applications with $ID(P') \neq ID(P)$ in layer l_j for all $j \geq i$ on ρ, and all applications on all devices ρ', unless $ID(P) = ID(P_u)$, where $ID(P_u)$ is the intended unsealer specified by P, and $\rho = \rho'$.

The implementation of sealed storage may involve P interacting directly with the sealing resource in layer l_0. However, if principal Q in layer l_1 provides access to the resource in l_0 to P in l_2, Q can view d before it is encrypted and after it is decrypted. In addition, Q may be able to arbitrarily unseal d by generating $ID(P_u)$, if known to Q, without P_u issuing an unseal request.

Correctly assuring P of the correct behaviour of any Q used to implement sealed storage on τ, if Q is not in l_0, is an open problem in trusted computing frameworks.

2.3.4 Curtained Memory

Curtained memory provides, possibly hardware-enforced, assured memory space separation between processes. Curtained memory is also called *curtained execution* or *strong process isolation*. In this thesis, it will be known as curtained memory. It prevents processes from reading or modifying another process' memory space. Additional to this, it prevents a process from learning of another process' existence on the computing device.

In the 1950s, computer programs were typically batch-processed one after another. They were queued up by a system administrator, and then executed and left to run unattended. Programs ran until they generated an error, or until they completed. This meant that an executing program had exclusive access to all the resources of the computer it was running on. More pointedly, it meant that the entire address space of the computer was available for use by the executing program. There was no resource contention for memory, and programs were written to address memory directly.

This method of sharing a computer was not a particularly efficient use of the limited computing resources available, especially for users of the computer. Multi-tasking operating systems were developed to allow more than one program to run concurrently; programs were swapped out when they made a request to some peripheral, and a waiting program was swapped in and began executing. This meant the processor was not left idle, waiting for a slow peripheral to return some data. This also mean that an executing program was forced to share resources on the computer. A program was no longer able to address the entire memory range directly, as memory was now shared between executing programs.

This forced operating systems to take a more active role in memory management, especially enforcing some form of memory protection. Memory protection prevented programs from accessing memory addresses outside of their allocated range. If a program tried to access such an address, an exception would occur and the operating system would deal with the program, typically by stopping any further execution.

Multi-tasking (multiple programs) and multi-tasking (multiple users) operating systems were primarily advances in software. Hardware advances lead to virtual memory addresses, allowing the operating system to virtualise a programs address space, possibly allocating that program more RAM than the computer had. Slower storage, such as a hard drive, was used to store pages of memory when they weren't being accessed. A program's memory space was now managed completely by the operating system, considered to be executing with a higher privilege level. This resulted in significant usability and efficiency gains for programmers and users. Users were able to run programs that required more memory than was physically available on their machine, and programmers could leave memory management up to the operating system.

Operating system (privileged, or layer l_1) memory management code and virtualised memory went some way to keeping (unprivileged, or layer l_2) programs from accidentally or maliciously overwriting the memory of other programs. However, for an attacker it was trivial to write code that could be run with the same privilege level as the operating system kernel. On x86 architectures, device driver code, installed by the system administrator, is able to view and modify the entire memory space of the computing device, including virtual memory address ranges. Hardware architectures designed to run Unix variants fared better, as processes were able to mark pages of their address space as read, write, or execute only at the hardware level.

As a result of architectural changes since programs ran exclusively on a computer, an application's address space could no longer be trusted to remain confidential to, or unmodified by, other processes. With an untrusted operating system at layer l_1, an individual application at layer l_2 can be given no guarantee that its memory address space will not be read from or written to by code, either deliberately or accidentally, executing at either level l_1 or l_2. Despite these vulnerabilities, most applications are written on the presumption that their memory range will remain unmodified by another process, during its execution.

One noteworthy application that does not trust its memory space to remain unmodified during execution is the Google search engine. As is commonly known, Google pioneered the use of low-cost, low-quality components to build their massively parallel and distributed search engine. Instead of low-count, high-cost mainframes responding to a user's query, Google employ highly fault-tolerant software running on high-count, low-quality commodity-component computers prone to failure [45]. Multiple levels of redundancy ensure that if a computer fails while processing a query, that query is also being processed on another computer, which is able to step in and finish the transaction seamlessly.

In addition to using low quality CPUs, hard drives, and mother boards, Google also sources below-retail grade memory. This memory has failed manufacturer's tests and been declared unfit for sale at regular prices. Given the low quality of the memory, data read from an address x cannot be relied upon to be equal to what was last written to address x. Google has extended the memory I/O routines in their custom operating system. Memory operations involve the of use error-correcting codes to perform highly expensive checksum operations to verify all memory reads and writes at the operating system level.

Google is obviously an extreme edge-case of memory usage. However, the steps taken to verify data stored temporarily in volatile memory can be seen as equivalent to those required to protect a user-level application's address space from accidental modification by poorly written device driver code. Such driver code could be expected to write seemingly at random into a user-level application's address space. Google's error-detection motivated checksumming of memory I/O would not resist an intelligent attacker attempting to modify the memory space of an application. For a considerable increase in computational cost, the naive checksum could be replaced with a cryptographically secure hash verifying the integrity of data stored to volatile memory. Such a scheme would, of course, require a location in memory hidden from the attack in which to store the key that generates the hash.

Most applications written today do not perform any extra memory verification or protection

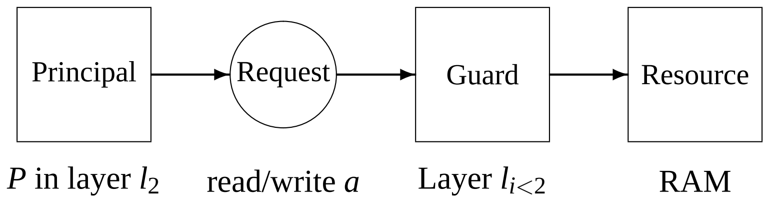


Figure 2.7: Access control model showing a layer l_2 principal issuing a read or write volatile memory request to an address μ_i.

than what is provided by the operating system and hardware. As mentioned above, for assured computation on multi-tasking and time-sharing computing devices, this is not good enough. Trusted computing aims to provide higher levels of assurance than the weak protection currently available in desktop personal computers.

Curtained memory is intended to keep the address space of a program executing at level l_2 protected from viewing and modification from all other untrusted processes on the machine. This includes kernel level code at level l_1, made up of (badly written) device drivers, as well as code maliciously inserted to attack user-level programs. As discussed in Section 1.2, trusted computing treats the user or owner of the machine as a malicious attacker. In line with this reasoning, curtained memory does not allow the local user or administrator to view or modify the address space of any application running behind the so called *curtain*.

Figure 2.7 shows the access control model introduced in Section 2.3.1 showing a program P accessing (read or write) memory at address μ_i. The guard here is any lower layer $l_{i<2}$. The request can be approved or denied by any principal in any lower layer in the computing device, depending upon the implementation of τ. However, for a given τ, every request by all principals in layer l_i must be serviced by the same principal in layer l_j, where $j < i$.

Each program P executes in an associated memory address range, denoted $\mu_{0...n}$. A specific memory address is denoted μ_i. This memory space contains, to keep our model simple, both code (instructions) and data. Instructions and static data is stored in the range $\mu_{0...j-1}$, and stack and heap data in the range $\mu_{j...k}$. As mentioned in Section 2.3.1, a principal at layer l_j runs with higher privileges than a principal at layer l_i, where $j > i$. As with most hierarchical operating system security models, a higher privilege implies the ability to signal, create, modify, and view processes with a lower privilege.

Definition 2.3 of curtained memory suffices under the assumption that no principal in layer l_j, where $j < i$, is incorrect or acts maliciously. This assumption leads to the current model of memory protection, outlined above. To provide curtained memory, a principal in layer l_i must have no principals considered untrusted, in a layer with higher privilege ($l_{j<i}$) — remembering that layer l_0 is axiomatically trusted by all principals in a layer l_i, where $0 < i$.

Definition 2.3. A trusted computing framework τ provides *curtained memory* if there exists a memory range $\mu_{0...n}$, associated with an application P in layer l_i, unaddressable to all other applications P' in layer l_j for all $j \geq i$.

A principal P in layer l_1 is able to view or modify the memory range $\mu_{0...n}$ of any principal in layer l_i, $i > 1$. In order to assure the confidentiality and integrity of the memory of a principal in layer l_i, $i = 2$, the axiomatic trust in layer l_0 must be extended from layer l_0 to layer l_{i-1}. This extension of trust must encompass all principal-guards servicing memory access requests for principals in each layer from l_0 to l_i (l_1 only in our model). Once trust has been extended from l_0 to a principal Q in l_1 in some manner, principal Q's memory range must share the same attributes as principal P's, as given by Definition 2.3, so that Q is not subverted by a malicious principal in the same layer l_1.

This observation leads to the conclusion that *strong* curtained memory, where Definition 2.3 holds true for each pair of principals P and P' from the set of all principals in a layer l_i, is difficult to achieve. A *weak* curtained memory definition, Definition 2.4, is easier to implement. Conceptually, it partitions the entire memory range of a computing device into two, one side of which is *curtained* from the other side. An application P's memory range behind the curtain is unaddressable to all applications P' not behind the curtain, regardless of their respective levels. However, P''s memory range is addressable by some principal P behind curtain, allowing communication between the two partitions.

Definition 2.4. A trusted computing framework τ provides weak *curtained memory* if there exists a memory range $\mu_{0...k}$, associated with a set of applications $P_{0...n}$ in levels $l_i, ..., l_j$, where $0 < i \leq j$, that to all principals not in the set $P_{0...n}$ is unaddressable.

Curtained memory prevents applications that are running 'uncurtained' from knowing of the curtained memory range. With strong curtained memory, programs behind the curtain are also curtained from each other. Weak curtained memory merely provides a single contiguous curtained address range, in which all curtained principals execute.

2.3.5 Attestation

Attestation enables a computing device to export a data structure verifying its identity and local state. This data structure, known as an *attestation vector* (AV), enables a *challenger* to perform remote verification of the device's state. This allows arbitrary computation, performed remotely on an otherwise untrusted device, to be trusted as though done locally. Of the three primary trusted computing primitives, attestation is the most significant, and most novel.

Early computers were typically shared amongst many users, connected to a single computer via a dumb terminal or similar. Programs and data that users relied upon typically resided on the same machine that a user logged in to, even if the user was not located at the console of the machine. A user, making use of applications and data residing in layer l_2, relied on the administrator of the system, located in layer l_1, to ensure the integrity of their workspace (see Section 1.2 for discussion of user classes).

An application, locally stored, could be trusted to remain unmodified as long as the proper access controls were set, and enforced correctly by the operating system. Users at layer l_2 were not intended to be able to make changes that would affect system integrity, and system administrators at layer l_1 were not considered to be malicious. The security boundary in this environment was limited to the one computer that all users logged in to.

Computing environments moved from this monolithic, single machine environment to a client/server architecture. Users had computers on their desktops for personal use. Depending on specific application characteristics, user data and applications could reside on the local machine, or the user would run a client application, consuming data and services provided by a server on the network. Users worked in layer l_2, and typically had no level l_1 access to their machine. All machines on the network were under the control of an administrator at level l_1. Operating systems capable of providing this form of computing environment were necessarily network-aware. The security boundary extended to cover the entirety of all the machines under the administrator of a single entity.

During the 1990s, the Internet increased the levels of communication between what were previously autonomous networks under different administrators. Corporations and individuals began to rely on applications and data provided by entities on different networks, all outside the security boundary from within which they traditionally used to work.

Attestation grew out of a desire to assure the executing image and environment of an application located on a remote host. Specifically, so that these assurances could be given for computing devices outside the security boundaries noted above. More formally, a single administrative principal P at layer l_1 in a set of computing devices $\{\rho_{0\ldots n}\}$ implies the security boundary covers all applications $\{Q_{0\ldots n}\}$ in layer l_2 on all devices $\{\rho_{0\ldots n}\}$. This security boundary typically results in an implicit trust relationship existing between all applications in the set $\{Q\}$.

There is no single administrative entity P at layer l_1 over the disparate computing devices in the 'network of networks' that users now interact with. Attestation attempts to provide a similar extension of security boundaries that a single administrative entity at layer l_1 provides, by providing a single administrative entity at level l_0.

Attestation can also be considered an evolution in integrity-checking in client-server software architectures. Cryptographically signed Java applets, in conjunction with some public key infrastructure (PKI), allow end users to verify the integrity and manufacturer of a Java application. In this technique, an applet distributor cryptographically signs the Java Application Runtime (JAR) file. The distributor uses the private key of an asymmetric key pair, the public part of which is distributed through some PKI. The resulting certificate is distributed with the applet, and can be used by the client to verify that the JAR has not been corrupted, either maliciously or during network transmission.

The guarantee of integrity provided by this mechanism is only available to the client. The trust relationship goes only one way. Should the applet perform some computation, acting as a client, and send the results back to the applet distributor server, the results are unable to be verified. The applet can be modified in some way before it is executed, so as to affect the results of the computation in some way. Alternatively, the applet could not be executed at all, and fabricated results returned. To use traditional client-server architecture parlance, the server is unable to rely upon any computation, performed by the applet at the client, to be performed correctly. Other methods of assuring the results of a remote computation are discussed briefly in Section 3.4.

From the motivations for the development of an attestation procedure outlined above, it can be seen that attestation vector must be capable of withstanding attacks from users at both levels l_1 and l_2 on a computing device ρ. The local administrator must not be able to modify the attestation procedure, or vector, in any way. The affect of this on a trusted computing framework's threat model is discussed in Section 1.2.

The attestation vector must contain sufficient information to allow a computing platform ρ_i to be assured that a platform ρ_j is also running a valid τ. This assurance allows ρ_j to trust that the attestation vector was generated correctly, and accurately reflects the state of ρ. It prevents a device ρ', not running the framework τ, from fabricating a valid attestation vector.

The attestation vector is required to travel over an untrusted network from ρ_i to ρ_j. The integrity of the AV must therefore be guaranteed by the attestation procedure itself; the primitive cannot rely on a secure transport layer provided by a principal in any layer l_i, where $i > 0$.

Attestation is intended to assure a remote platform ρ_i of the relevant state of ρ_j, in order to assure the correctness of the computation of a principal P on ρ_j. The *relevant state* of ρ_j that can affect the computation of P depends on the implementation of the framework τ. For example, in a traditional personal computer architecture, a principal P in layer l_2 can be affected by all principals in layer l_1. For an arbitrary operating system, this includes all executable code loaded into l_1, as well as configuration files, from the point in time when execution was passed from the BIOS in layer l_0 to software in layer l_1 to when the attestation vector is created.

The function which derives the attestation vector must be immutable, and located in level l_0. It may be that l_0 is not properly able to identify all principals $Q_{1\ldots n}$ in layers $l_{i>1}$ capable of affecting P, and include their identities $ID(Q_j)$. To increase the functionality of the attestation function in layer l_0, a principal R in layer l_1 can be used to find all $Q_{1\ldots n}$. If the attestation vector from ρ_i includes $ID(R)$, ρ_j can include R as a parameter in its decision to trust the computation performed by P.

Recall that an Attestation Vector (AV) is a statement about the present state of a platform ρ and a principal P. The information contained in the vector may not allow a remote party to infer the state of ρ and P for any point after the generation of the AV. In the traditional computer architecture mentioned above, an administrative user in layer l_2 can make arbitrary changes to principals in layers l_1 and l_2 at any point after an attestation vector is generated, either by elevating an application P in layer l_2 to layer l_1, or introducing a new application directly into layer l_1. The computation that a remote party is interested in can be delayed by the administrative user until they are able to compromise it.

Attacks based on this vulnerability are classed as Time Of Check to Time Of Use (TOCTOU) attacks. The mitigation of these sorts of attacks is an open problem in trusted computing frameworks. Various implementations, and the manner in which they approach TOCTOU attacks, are discussed in Chapter 3.

Table 2.5 proposes some attributes, derived from the discussion above, which we would require of an attestation procedure in order for it to assure a remote party of the state of a platform. The proposals are split into two categories — those of the attestation procedure or protocol, itself, including the generation of the attestation vector, and those of the generated attestation vector.

Requirement 1 comes from the trusted computing thread model discussed in Section 1.2. Requirement 2 prevents the AV from being modified undetected when being transmitted across an untrusted network. Requirement 3 prevents the disclosure of sensitive information, such as executing programs, disclosed in the attestation vector, discussed below. This is typically implemented through encryption of the AV between ρ and the challenger. Requirement 4 requires the attestation vector to protect against replay attacks, in which an attestation vector is reused by an attacker to fool a challenger. Requirement 5 indicates that the attestation protocol should make some attempt to reduce the vulnerability of the generated vector to TOCTOU attacks.

Requirement 6 assures a remote party that the generation of the AV was performed as specified by the trusted computing framework τ, and that τ is executing in layer l_0. This implies that ρ is under the control, in some sense, of τ, and not being emulated. Emulation by an attacker

	Requirements of attestation protocol
1.	Withstand software attacks from all levels l_i, where $i > 0$.
2.	Ensure the integrity of the Attestation Vector (AV) when transmitted from layer l_0 on ρ_i, to the challenger on ρ_j.
3.	Ensure the confidentiality of the AV when transmitted over an untrusted network between ρ' and ρ.
4.	Ensure freshness of the attestation vector.
5	Mitigate Time of Check to Time of Use class attacks.

	Requirements of attestation vector
6.	Assure the challenger that ρ is running τ in layer l_0.
7.	Attest to all state on ρ_j capable of affecting the computation of the principal P being attested.
8.	Attest to all principals used to derive the AV in layers $l_{i>0}$.
9.	Mitigate losses of privacy.

Table 2.5: Proposed requirements of trusted computing attestation protocol, as well as contents of attestation vector.

would allow the correct operation of τ to be subverted. Requirement 7 states that the AV should identify all principals capable of affecting the state of P. Requirement 8 states that any principal used by layer l_0 to obtain the code identities in the AV, should also be included in the AV. This formalises protection against an attacker subverting a trusted kernel in layer l_1 used to generate code identities for the attestation procedure. Requirement 9 points out that an attestation vector that includes all principals $Q_{1\ldots n}$ on ρ, when attesting the state of P, may result in privacy concerns, allowing a challenger to see all applications executing on ρ. These privacy concerns are further discussed below.

Definition 2.5 formally specifies the requirements of an attestation procedure discussed above.

Definition 2.5. A framework τ provides *attestation* if it can transmit an *attestation vector* for an application P from layer l_0 on device ρ_i to layer l_0 on device ρ_j, ensuring the AV's confidentiality and integrity, that

- assures ρ_j running τ that ρ_i is also running τ; and

- includes the identities $ID(Q_i)$ of all principals $Q_{1...n}$ able to affect the state of P (dependent on τ); and
- includes the identities $ID(R_i)$ of all principals $R_{1...n}$ used by l_0 to obtain $ID(Q_{1...n})$ (if any); and
- includes the identities $ID(R_i)$ of all principals $R_{1...n}$ used by l_0 to obtain $ID(P)$ (if any); and
- includes the identity $ID(P)$ of P.

The nature of the information contained in an attestation vector raises a number of privacy issues. An AV is required to include all principals capable of affecting the state of P, to enable the challenger to make a decision about trusting P. Attesting to all principals on ρ results in a significant loss of privacy for a user on ρ. Mitigating this loss of privacy, yet still generating a meaningful attestation vector, is an open problem in trusted computing frameworks.

In addition to mitigating the loss of privacy of ρ to the challenger described above, the contents of an attestation vector should be confidential to those two parties. This allows the attester ρ_i to be assured their platform state is being revealed only to the challenger.

The cryptographic keys located in layer l_0 uniquely identity that platform. These invariant keys pose a privacy issue similar to the per-processor serial number included in Intel's Pentium III CPUs [30]. The actual threats to privacy caused by the processor serial number were discounted by some commentators [59] because the serial number was never linked to the identity of a platform, nor intended to be used to strongly identify a platform. In contrast, the unique keys embedded in a layer l_0 device are intended to identity that platform, to assure challengers it is a valid τ. Those keys can be linked to an individual when they sign an AV sent to a challenger that collects personal information with the assured computation.

Anonymous attestation, where an attestation vector assures a challenger that rho_i is running a valid implementation of τ without revealing any information distinguishing ρ_i from ρ_k, also running a valid τ, is non-trivial to implement. An implementation has been developed [21] which allows *direct* anonymous attestation between two parties. Alternatively, a platform ρ can have a third party, trusted by them and the challenger, to verify τ on ρ and generate an *identity*. This identity is signed by the Trusted Third Party (TTP), and certifies that the identity belongs to a valid τ. A platform ρ can use the obtained identity to prove the validity of an attestation vector.

2.3.6 Secure I/O

Secure I/O allows applications to assure the end-points of input and output operations. This allows a program to be assured output intended for a specific peripheral, such as a printer or a video display, is actually consumed by that device. It also allows a principal to be assured that data received from a device, such as a keyboard or a mouse, is generated by that device.

Secure I/O is intended to assure the confidentiality and integrity of I/O between an application and any peripherals it communicates with. In addition, secure I/O also identifies two end-points of the communication. A user can be assured that their interaction with an application P is not intercepted by another application Q, nor that Q can spoof input from a user, so as to appear to P that it was typed on the keyboard.

Cryptographically assuring input and output from an application to a peripheral has not evolved from any comparable feature in present operating systems. Plug and Play (PnP,[7]) assists with the automatic configuration of peripherals when they are attached to a computer, most typically in Microsoft Windows. A platform and device-independent group also develops specifications for seamlessly identifying and configuring devices (Universal Plug and Play, [13]). This plug and play technology can be seen to have led to secure I/O only in that it uniquely identifies peripherals.

Identifying the privileges a user has when interacting with the system currently occurs in Unix-style operation systems. When a user is logged in as the root, or super user, the command-line interface typically indicates a change in privilege level through a change in the user interface. The standard command line prompt is indicated by a *$* symbol, whereas interaction as the root user is indicated by *#*. Although not resistant to spoofing, this change in interface is intended to inform the user they are interacting with a specific part of the system.

Definition 2.6. A trusted computing framework τ implements *secure I/O* if an application P can be assured that some specific

- input is obtained from a specific peripheral; or
- output is received by a specific peripheral.

Definition 2.6 gives a definition of secure I/O. A trusted computing framework may not implement secure I/O for all peripherals. Typically it is provided to assure input, in the form of passwords or mouse movements, is from the user and has not been spoofed by another application. This allows, for example, electronic banking applications to be assured that a user is present and is initiating a transaction.

Who would not rather trust and be deceived?

Eliza Cook

3 Survey of Trusted Computing Frameworks

3.1 Introduction

This chapter surveys the field of trusted computing frameworks, including industry implementations and academic research projects. Chapter 4 discusses aspects of the framework implementations, and compares them against the requirements proposed in Chapter 2.

The Trusted Computing Group's Trusted Platform Module is discussed in Section 3.2. This specification deals with a layer l_0 entity exclusively. Software implementations in layers l_1 and l_2 are introduced. Section 3.3 discusses Microsoft's Next-Generation Secure Computing Base, built upon the layer l_0 TPM device. NGSCB specifies extra modifications to layer l_0, as well as a software kernel executing in layer l_1 providing services to specially written secure applications in layer l_2. Some relevant implementations providing assured computation in software are discussed in Section 3.4.Sections 3.5, 3.6, and 3.7 discuss trusted computing frameworks that do not build upon the Trusted Platform Module, but which are based upon other immutable hardware devices.

3.2 Trusted Computing Group's Trusted Platform Module

The Trusted Computing Group (TCG) [12], previously known as the Trusted Computing Platform Alliance (TCPA), is an industry group made up of over 90 members with interests in computer security. The seven major members are AMD, Hewlett-Packard, IBM, Intel, Microsoft, Sony, and Sun Microsystems. The TCG describe [70] their purpose and role as:

> ... to develop, define, and promote open, vendor-neutral industry specifications for trusted computing. These include hardware building block and software interface specifications across multiple platforms and operating environments.

They have published the lengthy Trusted Platform Module (TPM) Specification version 1.2 [72, 68, 69], three documents that together run for over six hundred pages. Working from this specification, selected TCG members have developed their own chips. Intel has developed what they refer to as *LaGrande* technology. AMD calls their own development Secure Execution Mode (SEM) [62]. IBM's own TCG specification-compatible chips are called Embedded Se-

Capabilities
Asymmetric key generation
Asymmetric encryption and decryption
Hashing
Random number generation

Table 3.1: Specified capabilities of the cryptographic coprocessor in a Trusted Platform Module, as specified by the Trusted Computing Group in [72].

curity Subsystem (ESS). IBM have been shipping these ESS chips in some desktop and laptop computers since 2002 [51].

An in-depth discussion of the TPM 1.2 specification document is outside the scope of this thesis. However, as mentioned the TPM chip specification is used as the base of Microsoft's NGSCB (3.3) trusted computing framework. IBM's own TPM chip is also used as the basis of their Global Security Analysis Lab's (GSAL) research and analysis of the specification and its capabilities, as well as other researcher's attempts to develop practical trusted computing frameworks. This section will first discuss the TCG's technical specification for the Trusted Platform Module v1.2, limited to layer l_0. Layer l_1 and l_2 functions, implemented in software, developed by IBM's GSAL and other researchers will then be discussed.

3.2.1 Concepts

Safford describes the TPM chip as having three primary functions [50]. Listed below, item 1 refers to both asymmetric and symmetric encryption routines. Item 2 allows the TPM to attest to a specific software state. Item 3 refers to features that allow the chip to be 'owned' by an entity, not necessarily the primary user of that specific TPM.

1. Cryptographic functions
2. Trusted boot functions
3. Initialisation and management functions

Cryptographic functions are enabled by three algorithms that each TPM must support: RSA, SHA-1 and HMAC [72]. The cryptographic processor inside the TPM must be able to perform the operations listed in Table 2.4. Symmetric encryption is only is only used within the TPM itself. Specifically, the specification states the TPM is not allow to "expose any symmetric algorithm functions to general users of the TPM" [72, p.12]. The specification also stipulates a lower bound on key strength. Storage and Attestation Identity Keys (AIK), explained below, must both be at least as strong as 2048 bit RSA keys. The hashing function, implemented with SHA-1, serves to generate platform integrity measurements, which are stored in Platform Configuration Registers (PCR). PCRs are used by the Trusted Platform Module to securely store, record, and report the state of the platform itself. Each TPM must provide at least 16 PCRs.

When the system is booted, each PCR is zeroed. Before the CPU begins the execution of some piece of software or firmware, the code is first *measured* by the Core Root of Trust for Measurement (CRTM), and the results are stored in one of the PCR registers. Measurement is the term used by the TCG to describe the generated of the code ID $ID(P)$ for an application P, as described in Section 2.3.2 on page 28. This measurement procedure is defined below. The first code to execute on a TPM device, the BIOS, is assumed to reliably measure itself. Pashalidis and Mitchell [47] give a succinct description of the process used to measure and record a platform's state, adapted below. We define the notation $M[x](P)$ as the steps listed below, where x specifies the appropriate PCR.

1. Software or firmware code, P, about to be executed is hashed giving a digest SHA-1(P).
2. A specific PCR is selected, and its existing value concatenated with the calculated digest giving $\{\text{SHA-1}(P)\|\text{PCR}[x]\}$.
3. The concatenated string is hashed again, and stored in the specific PCR register, giving $\text{PCR}[x] = \text{SHA-1}(\{\text{SHA-1}(P)\|\text{PCR}[x]\})$.
4. The Stored Measurement Log (SML) is updated with an entry containing the identity of P, and the PCR register used in step 3.

Step 3 in the above is referred to as *extending* a PCR digest. This process of extending a PCR digest has a number of important properties. The first is its ability to record an arbitrary number of program measurements in a limited amount of secure storage space. An alternative method, storing all individual measurements of P separately and securely, results in expensive space requirements. Using the Stored Measurement Log, it is possible to obtain any earlier PCR value by simply repeating all the measurements that occur before it in the SML. The only constraint is that all values of P must still be available to the TPM. The PCR values are stored inside the TPM, and only modifiable in two ways. The first is when the platform is reset, or turned on, resulting in all PCR registers being zeroed. The second is through the use of the measure command, defined above.

Additionally, the non-commutative nature of the extend operation, step 3 in the definition of $M[x](P)$ above, captures the changing state of a computing platform. The execution of a program P followed by the execution of a program Q does not result in the same state as the execution of Q followed by P. Equation (3.1) shows this important property.

$$M[i](A \text{ then } B) \neq M[i](B \text{ then } A) \tag{3.1}$$

This measurement procedure is used by the TPM in its implementation of attestation. The first few steps of booting a TPM-enabled device are described by Marchesini et al. [39] below:

> During boot time, the BIOS measures itself and reports that to the TPM ... The BIOS feeds the Master Boot Record (MBR) to the TPM to hash before passing control to it. Subsequent software components are expected to hash their successors before loading them, and the hashes are stored in PCRs.

As mentioned above, the BIOS is assumed to measure itself, and then the MBR, before passing execution to the MBR. The Root of Trust for Measurement is one of three roots of trust in the TPM. A function or entity considered to be a root of trust in the TCG specification is one which is "trusted to function correctly without external oversight" [71]. The TCG intend for trust in the correct operation of these entities to be generated by the design and implementation procedures, as well as by inspection. The specification states [71, p.6]:

> Trusting "roots of trust" may be achieved through a variety of ways but is anticipated to include technical evaluation by competent experts.

The other two roots of trust in the TPM are the Root of Trust for Storage (RTS) and the Root of Trust for Reporting (RTR). The RTM is the procedure outlined above, and is implemented through a number of TPM commands, shown in Table 3.2. The RTS is a logical entity capable of maintaining values generated by the RTM for as long as necessary. In the TPM, the PCR registers fill this role. The TCG specify that only four commands are able to alter the value of the PCR registers.

`TPM_Extend` Used to extend a PCR value with a 160 bit field not calculated by the TPM itself.

`TPM_SHA1CompleteExtend` Used to extend a PCR value with a SHA-1 digest calculated by the TPM.

`TPM_Startup` When called with a *clear* flag, resets all PCR registers to default (zeroed) state.

`TPM_PCRReset` Resets specified PCRs to default (zeroed) state, if they are marked as 'resettable.' PCR registers reserved for system use (0–7) are marked as not resettable.

These four commands prevent an attacker from resetting specific PCR registers, or from 'rolling back' a PCR to an earlier value. The platform must be rebooted, thereby resetting all data measured and stored in the PCR registers themselves, to reset the PCR registers.

The RTR is a mechanism for correctly exporting the values held in the RTS to interested parties. In the TPM, this function is implemented through commands: `TPM_PCRRead` and `TPM_Quote`. These commands are used to implement the attestation primitive, discussed below.

The TCG specification uses a concept of *transport sessions* to ensure the confidentiality and integrity of communication between a calling principal *P* and the TPM chip inside the TCB [72, p.71]:

> Session establishment creates a shared secret ... [and uses the] shared secret to authorize and protect commands sent to the TPM using the session.

The exact implementation of the session encryption layer is outside the scope of this thesis. Two protocols are specified: Keyed-Hashed for Message Authentication (HMAC) [32] and a

Command	*Explanation*
`TPM_SHA1Start`	Prepares TPM for subsequent *Update* or *Complete* commands. Initialises a thread in the TPM to calculate a SHA-1 digest.
`TPM_SHA1Update`	Adds an integral number of 64 byte blocks to the current SHA-1 digest being calculated.
`TPM_SHA1Complete`	Adds a partial $i < 64$ or complete $i = 64$ byte block to the current SHA-1 digest. Finishes the calculation, and returns the completed SHA-1 hash.
`TPM_SHA1CompleteExtend`	Adds a partial $i < 64$ or complete $i = 64$ byte block to the current SHA-1 digest. Finishes the calculation, extends a specified PCR register, and returns the extended value.
`TPM_Extend`	Extends the specified PCR register with an arbitrary 160 bit value.
`TPM_PCRRead`	Returns the 160 bit value of 1 specified PCR.
`TPM_Quote`	Returns the 160 bit value of 1 or more specified PCRs, a nonce value (arbitrary 160 bit field) specified as a parameter, all signed with a specified key.
`TPM_PCRReset`	Resets all specified PCRs, if all those PCRs are able to be reset.
`TPM_Init`	Command sent by hardware platform to inform TPM that platform is starting the boot process. Unable to be issued via software, and must occur only during platform power up.
`TPM_Startup`	Preceded by *Init* command above, and called with one of three flags to indicate to the TPM its initial state: clear, save or deactivate.
`TPM_ExecuteTransport`	Delivers a wrapped TPM command, ensuring its confidentiality and integrity, from a caller to the TPM, which unwraps and executes it.
`TPM_ReleaseTransportSigned`	Completes a transport session. If logging is enabled for the session, the log is returned. If logging is not, an error is returned.
`TPM_Seal`	Seals data so that the returned blob can only be unsealed when PCR registers are in the state specified.
`TPM_Unseal`	Unseals a blob. Unsealing will only succeed if the PCR registers match those specified by the seal command that generated the blob.
`TPM_CreateWrapKey`	Creates a secure storage bundle for asymmetric keys. The newly created key can be locked to a specific PCR value by specifying a set of PCR registers.

Table 3.2: Commands present in the Trusted Computing Group's Trusted Platform Module v1.2 specification [69].

Mask Generation Function (MGF1) [31, §10.2]. Together they are intended to authenticate commands and parameters, and prevent replay and man-in-the-middle attacks. They are not intended to provide long-term confidentiality guarantees for authentication data made up of passwords or other low-entropy data [72, p.48].

The session is managed and enforced by the TCG Software Stack (TSS). The TSS is intended to alleviate interface shortcomings of the TPM that arise due to its limited resources. The TSS is responsible for resource management of the TPM, ensuring synchronised access and proving a single entry point for applications to use the TPM. In regards to session management, it is specifically required to [72, p.89]:

> ...ensure that only commands using the session reach the TPM. ...the TSS should control access to the TPM and prevent any other uses of the TPM.

Session management, done by the TSS, occurs outside the TPM. The TSS is considered untrusted, just like the operating system and applications on the platform [67, p.14].

> In the TCG architecture, the boundary around the TPM is the TCB [Trusted Computing Base]. All components outside the TPM (i.e., the TCB) are untrusted such as the TSS, OS, and applications.

The TCG specification places a number of restrictions on session management. The TPM is able to support only one session at a time. Any command, other than `TPM_ExecuteTransport` or `TPM_ReleaseTransportSigned` (see Table 3.2), being issued when a session exists results in that session being invalidated.

3.2.2 Code Identity

The `TPM_SHA1*` group of commands are used to generate a SHA-1 digest of arbitrary data. The `TPM_SHA1CompleteExtend` command allows the resulting digest to be used to extend a specific PCR value, without requiring the computed digest to leave the TPM. It should be noted that the TPM command specification does not enable a SHA-1 digest to be computed and used to extend a PCR atomically. To compute a SHA-1 digest of data over 64 bytes in length, at least *n* commands must be issued.

1^{st}	`TPM_SHA1Start`
$2 \ldots (n-1)^{th}$	`TPM_SHA1Update`
n^{th}	`TPM_SHA1CompleteExtend`

The `TPM_SHA1Start` command returns a value specifying the maximum number of bytes that can be specified in a `TPM_SHA1Update` command. The maximum size of the update command is limited to 2^{32} bytes, by the use of a 32 bit field to specify the size of the command. No available information specifies the maximum number of bytes that can, in practise, be processed by a `TPM_SHA1Update` command.

To briefly summarise, the SHA-1 digest generated by the TPM is used as the code identity (Section 2.3.2) of an application. The SHA-1 algorithm is considered to meet Definition 2.1 on page 28 for a code identity function. Specifically, that there is no collision for any two distinct programs P and Q. Additionally, the mechanism used to generate the digest on ρ is contained within the TPM and trusted implicitly by all other platforms ρ'.

3.2.3 Sealed Storage

Another of the cryptographic capabilities of the TPM is protected storage. This is implemented through two TPM commands, listed in Table 3.2 on page 56, `TPM_Seal` and `TPM_Unseal`. Marchesini et al. [39] describe the sealing and unsealing process from a programmer's point of view:

> ...one can ask the TPM to seal data, and specify a subset of PCRs and target values. The TPM returns an encrypted blob (with an internal hash, for integrity checking). One can also give an encrypted blob to the TPM, and ask it to unseal it. The TPM will release the data only if the PCRs specified at sealing now have the same values they had when the object was sealed (and if the blob passes its integrity check).

The TPM device itself makes use of both asymmetric and symmetric encryption routines when *sealing* data. A set of PCR digest values, the PCR register indexes they are found in, and the symmetric key K_s used to encrypt the data, are asymmetrically encrypted with a key

K_{pub}. The set of PCR digest values are 160 bit SHA-1 digests. The set of them is denoted by $PCR\{X\}$, where X is a set of 2-tuples (i, v), in which i specifies a PCR index from 0–15, and v is the SHA-1 digest.

It is important to note that the "data" referred to by Marchesini et al. is restricted in size to a few kilobytes. An application P is intended to use the seal primitive to store a cryptographic key. An application P must use this key to ensure the confidentiality and integrity of data by encrypting and decrypting it outside the TPM.

Equation (3.2) shows the encryption performed by the `TPM_Seal` command, resulting in an encrypted blob β. Equation (3.3) shows the decryption performed by the `TPM_Unseal` command on the resultant β.

Here, the notation $E(k, d)$ indicates the encryption of d, limited in size to a few kilobytes, with the key k. When called, it returns the cipher text of d. Similarly, the notation $D(k, d)$ indicates the decryption of d with the key k. The encryption and decryption method is dependent upon the key type of k.

$$E\left(K_{pub}, \left\{ \{X\} \parallel K_s \right\}\right) = \beta \tag{3.2}$$

$$D\left(K_{prv}, \beta\right) = \left\{PCR\{X\} \parallel K_s\right\} \tag{3.3}$$

The key K_{prv} is the decrypting key of the asymmetric encrypting key K_{prv} selected in the seal command. The set of register indexes and values specified by the `TPM_Seal` command are not the set of values in the PCR registers when the command is called. They are intended to be specified separately so that a future configuration, when unsealing the returned blob β is intended, can be specified.

When the `TPM_Unseal` call succeeds, the unencrypted data is returned to the caller. Additionally, the call returns the full set of PCR register values at the time the `TPM_Seal` was called. The TCG specification explains the motivation for this [69, p.44].

> ...suppose an OS [Operating System] contains an encrypted database of users allowed to log on to the platform. The OS uses a sealed blob to store the encryption

Index	*Usage*
0	CRTM, BIOS and platform extensions, limited to executable code only.
1	Motherboard configuration including hardware components and their configuration.
2	Optional ROM executable code contained on executable peripherals.
3	Option ROM configuration and data that may influence ROM execution.
4	Initial Program Loader (IPL) code, usually the Master Boot Record (MBR).
5	IPL code configuration and data that may influence MBR execution, such as configuration data selecting the partition to be booted.
6	Power events such as sleep and wake cycles.
7	Reserved for future usage.

Table 3.3: Defined Platform Configuration Register Usage for 32 bit PC architecture, adapted from [66].

> key for the user-database. However, the nature of seal is that any software stack can seal a blob for any other software stack. Hence the OS can be attacked by a second OS replacing both the sealed-blob encryption key, and the user database itself, allowing untrusted parties access to the services of the OS. To thwart such attacks, sealed blobs include the past software configuration.

For the OS to properly trust the contents of a decrypted blob, it must verify the state, as defined in PCR registers, of the platform when it was encrypted. That state must be known and trusted by the OS. The PC Specific Specification v1.1 [66] discusses implementation specific details for the 32 bit PC architecture. It specifies the usage of the 8 PCR registers that are reserved for the system. These are shown in Table 3.3. This set of PCR registers identifies all code in layer l_0, as well as the initial portions of the operating system code loaded into layer l_1.

When a principal P calls `TPM_Seal` they may, in addition to the parameters described above, be required to specify a 20 byte (160 bit SHA-1 digest) authorisation code. This authorisation code proves to the TPM that the caller is authorised to use the symmetric key K_s, used to encrypt the blob, seen in Equation (3.2). An authorisation code is first established for the TPM owner during an *ownership-initiation* phase. This ownership-initiation procedure is discussed in more detail below, in regards to the TCG’s specification for secure I/O.

The ownership-initiation phase establishes a shared secret between the TPM and the *TPM Owner*. This shared secret takes the form of a SHA-1 160 bit digest, typically obtained by hashing a password obtained from the user. Once a TPM Owner has been established, they are able to create the Storage Root Key (SRK). This is a type of storage key, shown in Table 3.4. There is only one SRK key in the TPM, and it is used to encrypt other keys outside the TPM, enabling vastly increased storage space for keys, yet ensuring their confidentiality and integrity [71, p.17]. The SRK is considered part of the Root of Trust for Storage (RTS) and is assured to be unable to be removed from the TPM.

In addition to creating the SRK, the TPM owner can create other storage keys. These keys can be created with an associated authorisation code, requiring a pass phrase from the user to be used. Alternatively, they can be associated with a set of PCR registers, limiting their use to a specific platform configuration. It is this authorisation code that, if set when the asymmetric key pair $K_{prv,pub}$ (Equation (3.3)) is created, must be specified by the caller of `TPM_Seal` and `TPM_Unseal`. The symmetric key used to encrypt the data is generated by the Random Number Generator (RNG) inside the TPM, and is stored only inside the returned blob β.

Creating a storage key, and associating it with a set of PCR values, including those in Table 3.3, as well as PCR values indicating the identity of the calling principal P, satisfies the definition of sealed storage given in Section 2.3.3 on page 29. The use of the SRK, tied to the platform on which it was created, prevents a sealed blob β, sealed on a platform ρ from being unsealed on any other platform ρ'. The TCG specify a command, `TPM_CreateWrapKey`, for this specific task. Marchesini et al. [40] describe the command:

> [The command `TPM_CreateWrapKey` makes it] possible to create keys which are bound to a specific machine configuration ... This alleviates the need to create a key and then seal it, allowing both events to be performed by one atomic operation.

3.2.4 Attestation

The other keys listed in Table 3.4 are used by the TPM to implement the attestation primitive described in Section 2.3.5 on page 42. They work in conjunction with the *credentials* listed in Table 3.5 to prove that the attestation vector (AV) is from a platform ρ with a valid TPM implementation.

Key	*Usage*
Signing keys	Asymmetric, general purpose. Used to sign application data and messages.
Storage keys	Asymmetric, general purpose. Used to encrypt data or other keys.
Identity keys	Asymmetric, non-general purpose. Used to sign data exclusively generated by the TPM (e.g. PCR values).
Endorsement Key (EK)	Asymmetric, single key, with public (PUBEK) and private (PRVEK) components. Used to establish platform ownership and when generating Attestation Identity Keys (AIK). Inserted by platform manufacturer.
Bind keys	Asymmetric, used to encrypt small amounts of data for transference between TPM platforms.
Legacy keys	Asymmetric, general purpose. Generated outside TPM and imported in, to be used for signing and encryption.
Authentication keys	Symmetric, used to protect transport sessions.

Table 3.4: Key types and uses in the Trusted Computing Group's Trusted Platform Specification v1.2 [71].

Credential	*Usage*
Endorsement credential	Issued by the manufacturer inserting the EK, inserted into the TPM by the manufacturer before shipping. Signed by the *TPM Manufacturer.*
Conformance credential	Issued by an entity with sufficient expertise to verify the correctness of the TPM and the platform, either the manufacturer, vendor, or third party. Signed by a *Conformance Entity.*
Platform credential	Identifies platform manufacturer and capabilities. References the endorsement credential and any conformance credentials. Signed by the *Platform Manufacturer.*
Validation credential	Issued by an entity with sufficient expertise to take and attest to measurement values that validate the correct operation of certain devices which may pose a security threat, such as disk storage or video adaptors. A validation is performed in a clean-room environment, and a measurement is produced for comparison with measurements taken during normal use. Intended to detect tampering of hardware and firmware. Signed by a *Validation Entity.*
Attestation Identity credential	Identifies the AIK private key used to sign PCR values. Contains the AIK public key. A Trusted Third Party (TTP) issues an AIK in return for proof that the AIK is owned by a TPM with valid Endorsement, Platform, and Conformance credentials. Signed by a *Trusted Third Party.*

Table 3.5: Credentials supplied with a Trusted Platform Module, as specified by the Trusted Computing Group [71].

During the manufacturing process, the Endorsement Key (EK) (Table 3.4) is generated and inserted into the TPM. A TPM can only have one EK inserted over its life time. Subsequent attempts to insert another EK will fail. The EK is a 2048 bit RSA key, with public (PUBEK) and private (PRIVEK) parts. The Endorsement Credential (Table 3.5) includes PUBEK. This has the effect of certifying the corresponding PRIVEK as belonging to a valid TPM platform. If the PRIVEK can be removed from the TPM, it can be used to impersonate a TPM device, thereby invalidating all assurances of security.

The EK is unique to each TPM, and is considered to be *personally identifiable information* [72, p.25]. While the PUBEK key alone does not contain personal information, it does uniquely identify the TPM platform. These unique invariant keys inside the TPM pose a privacy issue as described in Section 2.3.5.

The TCG specify the creation of an Attestation Identity Key (AIK), certified by a TTP (Table 3.5) to provide a layer of indirection and reduce the risk of loss of privacy. A TPM can generate any number of AIKs. This allows a TPM user to present a different identity to all services that require them. An AIK is a symmetric key used exclusively for signing purposes, the private component of which never leaves the TPM. Initialising an AIK involves 5 steps, discussed by Pashalidis and Mitchell [47] and Marchesini et al. [40].

1. The TPM Owner calls `TPM_MakeIdentity`. This generates a new asymmetric key pair $AIK_{pub,prv}$, and an *identity-binding*. This structure contains the public key of the key pair, an arbitrary text name t, and the public key of the TTP which will be used to generate the AIK TTP_{pub}. The final structure is:

$$\left\{AIK_{pub}||t||TTP_{pub}\right\}$$

2. The TPM packages the identity-binding described above, along with the Endorsement $Cred_E$, Platform $Cred_P$, and Conformance $Cred_C$ credentials (Table 3.5) into a structure, encrypted with the TTP's public key.

$$E\left(TTP_{pub}, \left\{AIK_{pub}||t||TTP_{pub}\right|\left|Cred_E||Cred_P||Cred_C\}\right) \quad = \quad \beta$$

3. The above structure is sent to the TTP, which decrypts and examines it. The TTP verifies the signatures of the issuing authorities of the supplied credentials. If satisfied the TPM is genuine, the TTP generates the Identity credential. The Identity credential, signed by the TTP, binds the public key of the key pair generated in step 1 to the arbitrary text name, along with a statement about the capabilities of the platform, obtained from the Platform and Conformance credentials.

$$\begin{aligned} D\Big(TTP_{prv}, \beta\Big) &\Longrightarrow \Big\{AIK_{pub} \| t \| TTP_{pub} \| Cred_E \| \\ &\qquad Cred_P \| Cred_C\Big\} \\ S\Big(TTP_{prv}, \{AIK_{pub} \| t \| Cred'_P \| Cred'_C\Big\}\Big) &\Longrightarrow Cred_I \end{aligned}$$

4. The TTP encrypts an arbitrary symmetric session key K_s (see Table 3.4), used to encrypt the Identity credential, with the PUBEK of the TPM obtained from the Endorsement credential. This ensures that only the TPM named by the Endorsement credential is able to decrypt and use the Identity credential. It also ensures that the IDK keys were generated by the TPM in question. The encrypted Identity credential and the encrypted symmetric key are returned to the TPM.

$$\begin{aligned} E\Big(PUBEK, K_s\Big) &\Longrightarrow \beta_1 \\ E\Big(K_s, Cred_I\Big) &\Longrightarrow \beta_2 \end{aligned}$$

5. The TPM owner calls `TPM_ActivateIdentity`, specifying the two encrypted blobs received from the TTP. The TPM uses PRIVEK to decrypt the symmetric session key, which is then used to decrypt the Identity credential.

$$\begin{aligned} D\Big(PRIVEK, \beta_1\Big) &\Longrightarrow K_s \\ D\Big(K_s, \beta_2\Big) &\Longrightarrow Cred_I \end{aligned}$$

At the end of this procedure, the TPM owner has an Identity credential $Cred_I$, signed by a

TTP with a root signing key TTP_{prv}, that names a public key AIK_{pub} of a key pair. The private key AIK_{prv} of the pair cannot be extracted from the TPM. The TPM owner uses this private key in `TPM_Quote` and `TPM_CertifyKey` commands.

The attestation vector (AV) discussed in Section 2.3.5 is created by calling `TPM_Quote`. The caller specifies both an AIK to be used for signing, and a set of PCR registers $PCR\{X\}$ to be signed. In addition to these two values, the caller can also specify 160 bits of external data, d. This means that the signing key can be used to sign arbitrary data, not just data generated by the TPM (see Table 3.4).

The attestation vector AV of a TPM platform ρ is made up of two parts. The first is the attestation itself. This is created by a call to `TPM_Quote`, seen in equation (3.4), denoted as $Q(signingkey, PCRvalues)$. The second is the identity credential associated with the specified AIK. These two components are concatenated together. The result is encrypted with the public key of the receiver ρ' of the AV to ensure its confidentiality and integrity during transmission, seen in Equation (3.5).

$$Q\Big(AIK_{prv}, PCR\{X\}\Big) \implies AV \tag{3.4}$$

$$E\left(\rho'_{prv}, \left\{AV \| Cred_I\right\}\right) \implies \beta \tag{3.5}$$

The construction of this attestation vector and its contents satisfies the requirements given in Table 2.5 on page 46. The attestation procedure itself is resistant to all software attacks from all levels l_i, where > 0, as all the information attesting to the state of the platform ρ is generated inside the TCB. The confidentiality and integrity of AV is assured through the use of encryption with an asymmetric key of ρ'. The attesting platform ρ does not required an assurance that the AV can only be read on another platform with a valid implementation of the TCF. Additionally, the arbitrary data in AV allows the challenger ρ' to specify a nonce to guarantee the freshness of the AV it receives.

The AIK-creation procedure discussed above requires both platforms to trust a third party before ρ' can be assured ρ is a valid platform. The construction of the AIK helps mitigate privacy losses, assuming the non-collusion of ρ' and the TTP, as described in Section 4.5.1.

The discussion thus far of the TCG's TPM has focused on the device itself, executing in layer l_0, and the interface it presents to principals in layers $l_{i>0}$. Discussion that extends from the specifications put forward by the TCG alone [72, 68, 69, 67, 71] lacks implementation experience of those upper layers. Given the specification for layer l_0, it is difficult to assess the functionality and features of layers $l_{i>0}$, used to implement a trusted computing framework with the properties discussed in Chapter 2.

The definitions of attestation (Definition 2.5 on page 47) and of sealed storage (Definition 2.2 on page 37) require an understanding and verification of all layers $l_{i>0}$, in addition to the axiomatically trusted TCB in layer l_0. As an example, for sealed storage it is important to ensure that a principal Q cannot impersonate a principal P, so that it appears $ID(P) = ID(Q)$.

3.2.5 Secure I/O

The TCG TPM v1.2 implements only a severely limited for of secure I/O. It is not considered to be one of the core priorities for the TCG platform [69].

Secure I/O is used to establish the *physical presence* of the user through the depression of a hardware switch, or a key sequence depressed during the Power-On Self-Test (POST). Such physical presence is used to initially authenticate as the owner of a Trusted Platform Module. Once authenticated as being physically present, the user can establish a shared secret, in the form of a password, between themselves and the TPM.

3.2.6 Curtained Memory

The TCG TPM v1.2 specification does specify any implementation of curtained memory. The consequences of this are discussed in Section 4.5.1.

3.2.7 Framework Implementations

An examination of a trusted computing framework implies not just a discussion of the trusted device in layer l_0. An examination of the software layer l_1 that can be built upon it, and the features layer l_1 provides to applications in layer l_2, as well as the features both layers provide to the user, is required. Literature [40, 39, 53, 52] discussing implementations of software in layer l_1 and l_2 built on top of the TCG's TPM specification forms the basis for further discussion and critique.

Marchesini et al. [40] were motivated to attempt to solve the canonical problem that motivates Trusted Computing (see Section 1.1), given an additional two constraints:

> ...Alice needs to trust that certain properties hold for a program running on Bob's machine, even though Alice may have little reason to trust Bob.
>
> To be effective, a solution to this problem must satisfy several constraints:
>
> - It must be *real*...
> - It must be *practical*...

They considered the TCG's TPM v1.1b, shipping in a number of IBM's laptops and desktops, to satisfy their constraint of the solution being real, by which they mean currently available. Their constraint of practicality requires it to work with standard protocols, and not be a "significant departure from the standard software base" [40].

Their instantiation of the general problem takes three forms, two of which are relevant to this discussion. The first is that of securing an SSL web server run by Bob offering a service with some security-critical properties to Alice. Marchesini et al. state that this would give Bob a marketing advantage: "You can trust my service, because you don't have to trust me [40, p.4]". The second is a more general case, attempting to solve the problems arising with the specification's lack of curtained memory, mentioned in Section 3.2.6 on the preceding page.

To solve the first problem, Marchesini et al. [39] use a TPM device to bind a private key to a specific software configuration, as described in Section 3.2.3. This private key, SSL_{prv}, is used to decrypt messages intended for the SSL web server by users of the system, encrypted with SSL_{pub}. Their first attempt bound this private key to the code identities of the entire server configuration. A Certificate Authority (CA) inspects this configuration and binds it to an AIK used for signing messages from the server, and a Storage Key (SK, see Table 3.4) used to store SSL_{prv}. The authors quickly realise the error of this approach.

> The naivete of this approach is obvious to anyone who has ever tried to deploy a system or a Web site in the real world. For one thing, the software will not be static. For bug fixes and security patches alone, various elements of the suite will have to be upgraded (and perhaps sometimes downgraded) over time. *The promise of responsibly maintaining a secure site requires that the executable suite, considered as a whole, be dynamic.*

Their second attempt partitions the system software based on how often they are expected to change. A *long-lived* system core consisted of the BIOS, MBR, kernel, and the *Enforcer* kernel module. The TPM boot process, previously described, ensures that the kernel and Enforcer are properly measured and loaded. Any change in their structure will result in differing PCR values being recorded, and the TPM denying access to the AIK and SK secrets of the SSL web server. Their Enforcer kernel module is used to verify the correctness of the *medium-lived* software, consisting of the Apache web server and configuration data.

The correct medium-lived software configuration code identity is generated by a third party, a *Security Admin.* The security admin creates a certificate including the code IDs of the medium-lived software. The Enforcer module was responsible for verifying the signature of the security admin on the certificate, by using the security admin's public key stored in the long-lived core. It also ensured that, once loaded, the PCR register values of the Apache web server, and its configuration data, were the same as the certificate. The Enforcer would then release the AIK and SK secrets to the Apache web server.

The system was built on a Debian "unstable" distribution, with the Enforcer implemented as a Linux Security Module (LSM). The first-stage of the LILO boot loader was modified to measure the second-stage, which was in turn modified to measure the Linux kernel itself.

The second instantiation of their motivating problem confronts the issues raised by the TPM's lack of curtained memory. Marchesini et al. [40] describe this as *compartmented attestation.* They conclude that the problems discussed in Section 3.2.6 are not solvable by the TCG's TPM specification itself [39, §3.3]:

> ...consider the case where Bob is a consumer [of some content], running a program

> *P* whose authenticity and integrity is of concern to a remote stakeholder Alice... Alice would like to ensure that Bob uses [a valid and correct *P*] that makes illicit use sufficiently difficult for her tastes.
>
> ...Bob may have to expose *everything* on his machine to Alice — even programs and data that have little to do with the application in question. Alice might even choose to deny services ... to Bob, if he has a competitor's product installed.
>
> ...TCPA/TCG itself appears insufficient to solve Alice's problems... Even [version] 1.2 TPM's attempts to localize PCR contexts appear to suffer from this problem.

To address these issues, Marchesini et al. merged the National Security Agency's (NSA) Security Enhanced Linux (SELinux) [9] with their Enforcer LSM. SELinux is mandatory access control architecture similar in many respects to that implemented by the LOCK system, discussed in Section 4.2.1 on page 102. SELinux will only be discussed in the context of the merger with the Enforcer kernel module here. SELinux enables what Marchesini et al. refer to as *software compartments*. These confine applications to their own compartment, prescribed by policy set by the system administrator. Specifically, the merger of SELinux with the Enforcer module gives *finer granularity, a restricted root, and a central access-control checking module* [39, §3.3]. Finer granularity refers to the ability of SELinux to differentiate between more privilege levels than the two, root (layer l_1) and normal (layer l_2) , that traditional operating systems provide. Restricted root refers to its ability to prevent the superuser from modifying the memory range of other applications. A central access-control checking module allows the concentration of all system security policy in one place, instead of it being spread around the operating system and file system in configuration and meta-data. The SELinux integration itself provides a number of specific features:

> The Enforcer module can use a key pair to testify about (and certify a key pair for) the contents of just one compartment... Alice can have assurance that the attestation she receives really pertains to the compartment in question, and that the Enforcer with SELinux will confine her data to just that compartment; Bob can have confidence that nothing outside of that compartment and above Enforce/SELinux will

be communicated to Alice.

Sailer et al. [53] did not attempt to solve the privacy concerns voiced by Marchesini et al. when instrumenting the Linux operating system to provide integrity measurement and reporting for layers l_1 and l_2. With a TCG/TPM system, they implement a form of *trusted boot*. Trusted boot refers to a boot process that measures and records all code and data relevant to the integrity of the system, before it is executed. The generated log can be trusted, via external mechanisms, by a remote party to have been correctly maintained, and not be tampered with. At the time of attestation, the log accurately reflects the state of the platform. This is in contrast to *secure boot*, implemented by the Aegis system (Section 3.5), which compares measured values with stored values to insure integrity of a program before it is allowed to execute.

The motivation of Sailer et al. [53] is the same general problem as that which drove the work of Marchesini et al. described above.

> Our goal is to enable a remote system (the *challenger*) to prove that a program on another system (the *attesting system* owned by the *attester*) is of sufficient integrity to use.

They split the implementation of their system into three areas:

- The *Measurement Mechanism*, responsible for determining what and when to measure, and storing the results securely.
- An *Integrity Challenge Mechanism*, enabling a remote platform to request an attestation vector, and be guaranteed of its freshness and completeness.
- An *Integrity Validation Mechanism*, capable of being run by the remote platform to verify all aspects of the attestation vector.

The implementation of the measurement mechanism was done on a Redhat 9.0 desktop box with a TPM module supplied by IBM. Like Marchesini et al. the software component responsible for taking measurements was implemented as a LSM. Similarly, the example application used throughout the discussion was the Apache web server, in this case also running Tomcat.

Building from the TPM boot process discussed in Section 3.2.1, the LSM extends this measurement process to include all executable content loaded by the kernel. This LSM is the measurement mechanism described above. The measured code IDs of all executable content are kept as an ordered list inside the kernel, filling the role of the Stored Measurement Log (SML) described earlier. A specific PCR register inside the TPM is updated along with SML. If the aggregate value of the code IDs found in the SML matches the value of the PCR register, the SML log is assumed to be untampered and correct. This enables the attestation vector to be formed, as described in Section 3.2.4, and accompanied with the SML. This integrity challenge mechanism, and its protocol, is given in Table 3.7 and described below. The integrity validation mechanism includes the decision process the requesting platform goes through to arrive at a decision in regard to the integrity of the attesting platform.

They consider the integrity of each program P in a system to be a binary property. Given the lack of curtained execution acting as coarse-grained mandatory access control, the integrity of the system as a whole is dependent upon the set of all integrity measurements of programs on the system. Specifically, Sailer et al. state that [53, §2.3]:

> Unless information flows among processes are under a mandatory restriction, the integrity of all processes must be measured.

The most interesting aspect of the system proposed by Sailer et al. is that encapsulated by the measurement mechanism. The determination of *what* to measure, and *when* to measure it, is crucial. For the initial execution of a program P, measurement occurs before execution begins, and $ID(P)$ is added to the SML, as well as the PCR register inside the TPM. For repeated executions of P, where P is unmodified, no record is made in the SML. For this reason, the SML is not an ordered log of all executions of principals $P_{1..n}$ on ρ, but an ordered log of the initial executions of principals $P_{1...n}$ only.

In addition to the executable content itself, Sailer et al. give two distinct categories of data that may affect the integrity properties of a program [53, §2.4]. The first is *structured data*, defined as data that has an *identifiable integrity semantics*. For the Apache web server under discussion, a number of files meet this definition:

- Apache configuration file (httpd.conf)
- Java virtual machine security configuration files
- Servlets and web services libraries

The second type of data is *unstructured data*, defined as data that affects the program, but whose integrity semantics cannot be measured. In the case of the Apache web server, this form of data includes the various kinds of requests received from remote users. In addition to the impracticality of measuring all requests from users, such measurement would be useless as it is impossible to predict values that would result in a loss of integrity. In effect, it is necessary to trust an application to resist attacks that may arrive in unstructured data. For the Apache web server, this may come in the form of malformed HTTP requests.

The first category of data can easily be measured in the same way as executable content is measured. However, it is not so trivial to ensure that an application *P* will either:

- correctly report to the LSM which configuration files will be interpreted at this execution and have identifed integrity semantics, so that the LSM may measure then and add them to the SML; or
- correctly measure and report to the LSM the code IDs of those configuration files that it will interpret and that have identified integrity semantics, so that it may add them to the SML.

Applications that have so-called structured integrity semantic data in configuration files must be re-written for the system proposed by Sailer et al.

One major class of program relies almost entirely on unstructured code — script interpreters. The interpreter itself is measured and loaded as per a standard executable. However, the call to execute a script interpreter may not include all the information necessary to identity all integrity semantic data. Modification of these programs to do one of either case described above is not sufficient. Pointedly, the authors instrumented the *bash* shell as an example of a script interpreter that cannot identify all integrity semantic data when it is first loaded [53, §5.1]:

Vulnerability	*Definition*
TOCTOU race condition	A Time of Check to Time of Use (TOCTOU) condition where file contents are changed after measurement, and before execution. Involves tracking number of open file descriptors pointing to the file, and invalidating the log if suspicious activity is detected.
Bypass user-level measurements	A redirection of user-level calls to the `/sys/security/measure` node requesting measurement of a file by preventing the unmounting of the system file system by root.
Bypass dirty flagging	Attempts by the root user to access the storage interface `/dev/hda` and the memory interface `/dev/kmem`, and invalidates integrity measurement log.
Run-time errors	Any error during the recording of measurements, such as an out-of-memory or other exception preventing the correct logging of measurement data, results in the invalidation of the measurement log.

Table 3.6: Vulnerabilities that lead to possible subversions of the integrity measurement log and cause a deliberate invalidation, adapted from [53].

> ...we have instrumented the *bash* shell to measure any interpreted script and configuration files before loading and interpreting them. This includes all service startup scripts into the measurement list [sic]... Instrumenting other programs (Perl, Java) is straightforward, but we anticipate the need for more support from application programmers.

One of the gains of the system proposed by Marchesini et al. was that, however cumbersome, the root or superuser was restricted from changing the memory addresses of other processes with a debugger. The system proposed by Sailer et al. does not actively seek to prohibit the root user from doing this. Instead, the system invalidates the SML log and aggregate PCR register, causing all future attestation attempts to fail, at step 5c of the integrity challenge protocol seen in Table 3.7, until the system is reboot. The cases that lead to an invalidation of the integrity measurement system are listed in Table 3.6.

Invalidation of the measurement log results in a system which is unable to perform a successful run of the integrity challenge protocol until the system is rebooted. Sailer et al. admit that the detection of the vulnerabilities cases given in Table 3.6 that result in an invalidation

Step	*Action*
1.	ρ': create non-predictable 160 bit nonce
2.	$\rho' \rightarrow \rho$: `ChReq(nonce)`
3a.	ρ: load protected AIK_{prv} into TPM
3b.	ρ: obtain $Q(AIK_{prv}, PCR \| nonce)$ from the TPM
3c.	ρ: retrieve SML
4.	$\rho \rightarrow \rho'$: `ChRes(Q′, SML)`
5a.	ρ': determine trusted certificate for AIK_{pub}
5b.	ρ': validate signature generated in step 3b by AIK_{prv}
5c.	ρ': validate nonce and SML with PCR value

Table 3.7: Integrity challenge protocol implemented on a TPM, adapted from [53].

occurring, may not be the result of an actual attack, nor that the attack would be successful. They describe the system as being necessarily *pessimistic* about such occurrences.

The integrity challenge mechanism, a protocol shown in Table 3.7, begins with a requesting platform ρ' creating a non-predictable non-repeating nonce. This nonce ensures the freshness of the response, preventing replay attacks, as required in Table 2.5 on page 46. This nonce is packaged into a suitable data structure, and sent to ρ. Sailer et al. assume communication between ρ and ρ' occurs over a SSL-authenticated transmission link. The attesting platform ρ uses an AIK key pair, as discussed in Section 3.2.4, to have the TPM sign (`TPM_Quote`) the aggregate PCR register, as well as the nonce. The attesting platform packages the signed data structured returned by `TPM_Quote`, denoted Q' in Table 3.7, with the SML, and returns both to ρ'.

Steps 5a through to 5c involve ρ' verifying the response to the attestation challenge. In step 5a ρ' checks to see the Trusted Third Party that signed AIK_{pub} is one of the TTPs trusted by ρ'. This step assures ρ' that ρ is a valid TPM implementing the same trusted computing framework τ. Assuming that ρ' does trust the TTP used by ρ, step 5b verifies that AIK_{prv}, stored securely in the TPM, was used to sign the aggregate PCR register and nonce. This step assures ρ' that the response originated from the TPM named in AIK_{pub}. It also ensures the quoted values have not been tampered with. Step 5c involves the verification of the SML with the aggregate PCR register value, ensuring that the state of ρ described by the SML is correct. Sailer et al.

succinctly describe this process [53, §4.3]:

> Tampering ... is made visible ... by walking through the measurement list *ML* [SML] and re-computing the TPM aggregate (simulating the TPM extend operations [Section 2.3.2] ...) and comparing the result with the TPM aggregate *PCR* that is included in the signed *Quote*...

If the two values match, the state of ρ when it generated the attestation response is assured to be what is stated by the SML.

Once ρ' has verified the state of ρ, a decision on whether or not to trust that state must be made. For the system proposed by Sailer et al. this involves matching each individual code ID in the SML with a previously computed one obtained from some trusted source, and reasoning about its affect on the integrity of the system, and more specifically, the application *P* that is of interest. The Redhat 9.0 system that Sailer et al. modified was measured by them to generate a trusted repository of "known-good" application identities.

> ...the workstation used... which runs Redhat 9 and whose workload consists of writing this paper, compiling programs, and browsing the web does not accumulate more than 500 measurement entries. On a typical web server the accumulated measurements are about 250.

Once each entry in the SML log has been identified, the remote platform ρ' must make a decision on whether to trust the integrity of ρ. The policy used to describe the decision process may be arbitrarily complex. The presence of any malicious programs may result in a decision to not trust the integrity of ρ. More complex decision processes, dependent on the required integrity and importance of *P*, would be required when programs classified as being remotely or locally vulnerable.

In response to the possibility of Time of Use to Time of Check (TOCTOU) attacks, Sailer et al. require the attestation procedure to occur twice — once prior to the transaction taking place, and once after, for ρ' to be assured that the transaction occurred in a manner suitable to it [53, §4.4]:

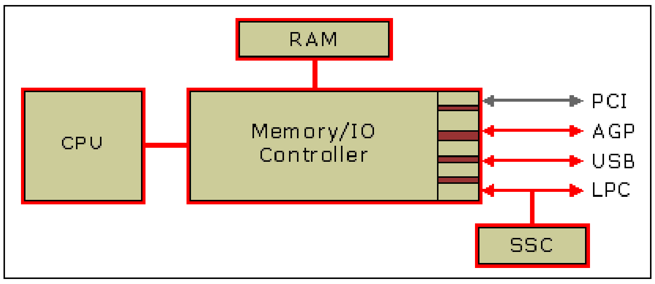


Figure 3.1: NGSCB hardware modifications, adapted from [42].

> To verify the integrity of a transaction that is taking place ... the challenging party can challenge the integrity of the attesting system before and after the transaction was processed ... If the attesting system is trusted both times, then — so it seems — the transaction can be trusted, too.

3.3 Next-Generation Secure Computing Base

Microsoft's Next-Generation Secure Computing Base (NGSCB, pronounced *"ing-scub"*) appeared to be, for much of 2003 and early 2004, the most advanced and complete trusted computing platform in development. In May of 2004, amidst confusing statements and press releases Microsoft first dropped completely, then reinstated, their NGSCB program. News reports [14] and correspondence with NGSCB team members [22] throughout the latter part of 2004 indicate that NGSCB would undergo considerable revision before being released with Longhorn. No information on the intended shipping version of NGSCB is available at the time of writing. With this in mind, this section will discuss the version of NGSCB released to attendees at Microsoft's Professional Developers Conference in Los Angeles, in November 2003.

3.3.1 Curtained Memory

NGSCB is based around the TCG's Trusted Platform Module v1.2 device. This device has been described in Section 3.2 on page 51, and will not be expounded upon here. In addition to making use of the TPM chip, Microsoft also make significant changes to other parts of the PC architecture. A typical logical view of a PC motherboard is shown in Figure 3.1, with changes

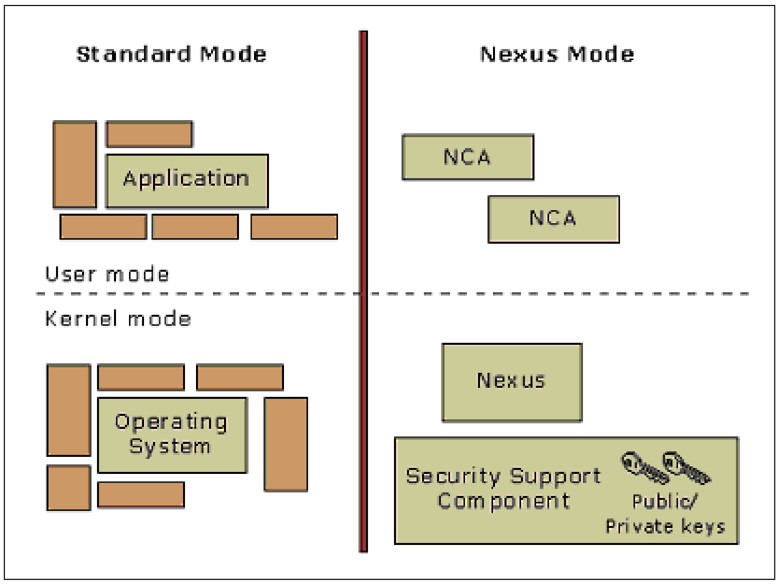


Figure 3.2: Logical software layout of standard and secure operating systems in NGSCB, taken from [44].

due to NGSCB shown in red.

A new mode flag is added to the CPU to differentiate between standard execution mode, and *privileged* execution mode. When this mode flag is active, the CPU is able to address what is referred to as *curtained memory*. Curtained memory is standard memory that has been marked in the hardware page table with a new addressing-mode bit. Unless the CPU is in privileged execution mode, the CPU is unable to address or access any pages marked as being curtained. There are additional architectural changes to the USB and AGP buses, to allow the exchange of encrypted data between the CPU and devices on the bus. Additionally, the Low-Pin Count (LPC) bus permits communications between the CPU and the Security Support Component (SSC). This SSC is Microsoft's name for a device that fills the role of the TCG's Trusted Platform Module.

Seen in Figure 3.2 is the logical partitioning of the software architecture in NGSCB. Microsoft refers to programs executing in standard, or unprivileged mode, as executing on the *left-hand side* (LHS) of the four-quadrant diagram. Privileged execution takes place on the *right-hand side* (RHS). The bottom quadrants show the operating system executing in standard mode on the LHS, and the *nexus* executing on the RHS. The nexus is the secure kernel that hosts and provides services to *nexus computing agents* (NCA), which are directly analogous to standard applications on the LHS. These NCAs interact with the nexus to obtain cryptographic services supplied by the SSC.

Software executing on the RHS (in nexus mode) is stored in curtained memory. This gives it hardware-level memory protection from the standard Windows operating system and userland applications. The standard Windows operating system, and the applications that run on it, will remain largely unchanged on the NGSCB platform. The insecure LHS will provide some limited services to the nexus and NCAs. These will include I/O operations such as those required to read and write files to disk, as well as communicate over a network. Such streams will, of course, not be protected from the rest of the LHS environment. All data leaving the RHS is intended to be encrypted, to protect its confidentiality and integrity.

Microsoft's NGSCB implements all four trusted computing framework primitives — attestation, sealed storage, strong process isolation, and secure I/O. NGSCB refers to secure I/O as *secure paths*. As expected, Microsoft's NGSCB makes heavy use of the cryptographic primitives exposed by the underlying TPM device. Its implementations of attestation and sealed storage are heavily influenced by the design decisions made by the TCG. There are a number of subtle differences however. These stem from Microsoft's decision to implement a secure kernel, the nexus, as well as partitioning the software layer into an insecure LHS and a secure RHS.

As outlined above, the NGSCB platform modifies the PC hardware architecture to allow for pages of memory to be marked as curtained. Such curtained memory can only be accessed when the processor is in privileged mode. It is in this hardware-mandated privileged mode that the nexus kernel executes. The hardware architecture prevents the unprivileged LHS from observing any page that has been marked as curtained. Shown in Figure 3.2, the LHS is unable to view the memory of any program on the RHS. However, the same is not true for the RHS, which is not prevented in hardware from viewing pages not marked as curtained. It should also be noted that individual NCAs on the RHS are granted no more memory protection from other NCAs, than normal applications on the LHS are from each other.

This hardware-mandated separation of memory pages marked as privileged and unprivileged meets our formal definition of weak curtained execution, defined in Section 2.3.4, definition 2.4 on page 41.

3.3.2 Sealed Storage

The sealed storage primitive implements access control to sealed data, based on the code identity of an NCA, and the nexus. This code identity is the hash of a *manifest* describing the NCA. This manifest names all the code libraries, including specifying versions, that an NCA can load, as well as data and configuration files. Changing a manifest, for example, by specifying a different DLL version results in a distinct NCA identity being generated.

To seal some arbitrary data, an executing NCA calls the `Seal` function, exposed by the nexus. This function takes two parameters, d and $ID(P_u)$. The parameter d is the text to be sealed, and $ID(P_u)$ is the code identity of the NCA that is able to unseal the returned cipher text. In the general case, the calling NCA specifies its own code identity, ensuring the confidentiality and integrity of the cipher text until it is unsealed at a later date. As mentioned previously, the nexus relies on the insecure LHS operating system for various I/O services, including file reads and writes. The encrypted cipher text is written to the standard file system, with no more protection than is available now.

The alternative case is one in which an NCA specifies some other code identity which is allowed to unseal the cipher text. This resultant cipher text can, however, only be unsealed on the same NGSCB device it was sealed on. This is due to the use of the unique symmetric key stored in the SSC (see Section 3.2) to encrypt the plain text. This use of code identity to name the intended unsealer of the cipher text allows chains of computation to be designed, as in the LOCK (see Section 3.4.1) and XOM (see Section 3.7) systems.

The `Unseal` function takes only one parameter — the cipher text to be unsealed. The success of the function is dependent on three identities. The first two, the NCA and the SSC chip, are explained above. The third identity is that of the nexus kernel itself. Microsoft has said that it is possible for anyone to write and distribute their own nexus [43], although doing so legally may present some licensing issues [23]. If the unseal function depended on the identities of only the NCA and the SSC, an attacker could modify the nexus before it is loaded. Such modifications, made to the compiled binary of the nexus before it is loaded, could result in the nexus leaking sealed data to the insecure LHS.

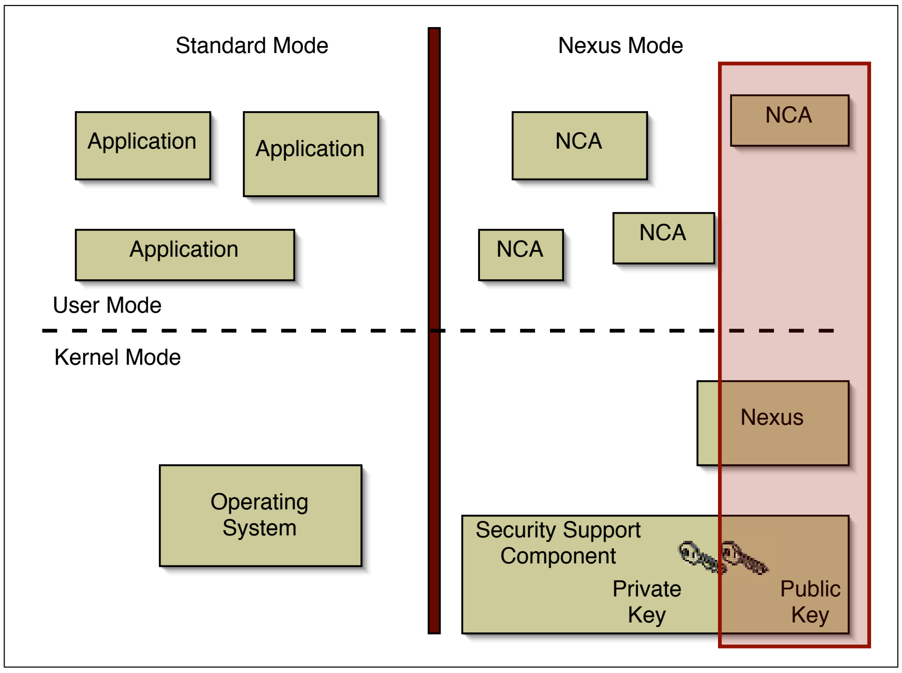


Figure 3.3: Hardware and software stack identified in an NGSCB attestation operation, shown in red.

3.3.3 Attestation

Microsoft's implementation of attestation is built on top of the architecture of the Trusted Platform Module. It is described in various technical white papers published by Microsoft [43, 44], as well as by England et al. [25].

Logically, it uses the *Quote* operation, as well as various certificates, to identify the hardware and software stack, seen in Figure 3.3, running on the NGSCB platform. The stack identified consists of the TPM device, the executing nexus, and a specific NCA. The quote operation is described by England et al. in [25]:

> The Quote operation concatenates an input string from the program wishing to authenticate itself with the program's code ID, signs the resulting data structure with the platform's privacy quoting key, and returns the result to the caller. The requesting program can send this signed data structure to a remote party, typically along with platform certificates that support use of the platform-quoting key.

In operation, the executing nexus calls the TPM's `Quote` function, passing in a data structure containing the code ID of the nexus requesting the attestation, NCA_{id}. The TPM concatenates the code ID of the nexus with this data structure, forming $(NEXUS_{id}||NCA_{ID})$. It then signs with the appropriate private key named in the manufacturer's certificate, and returns to the nexus $(NEXUS_{id} \| NCA_{ID})_{k'}$. k' signifies the private key embedded in the TPM, as described in Section 3.2.

3.3.4 Secure I/O

Microsoft's NGSCB treats secure I/O as one of the core components of its trusted computing framework. It allows secure, assured communication between NCAs and various peripherals connected to the device. Microsoft intends to secure those devices which are typically found outside the computer case. These include things like printers, speakers, keyboards, and mice. One exception to this is a secured video card, which resides inside the computer case.

Microsoft does not intend for NGSCB's secure I/O be used, at least initially, for DRM-style applications [48]. Typically, secure I/O reduces throughput of the device below acceptable limits for media playback. They are more concerned about securing the input from the keyboard and mouse to an enabled application. In the same manner, they intend to secure certain output from an enabled application so that it reaches the local user. Common attacks which Microsoft's secure I/O aims to prevent include user spoofing and screen scraping. User spoofing occurs when an attacker sends keystrokes to a program so as to appear as if they came from the local user. Screen scraping involves an attacker capturing an application's screen output and storing it for later use, either by replaying (in the case of video) or by capturing textual information from the data. For example, other work by the author [20] investigated using Microsoft's NGSCB to assure high-value legal documents were delivered to a valid printer, and not printed to a file.

Microsoft intends to demarcate user interface windows used to display a secure I/O-enabled application. This demarcation is intended to elevate the level of trust a user can have when interacting with that window. The typical use-case of this technology is to properly secure and assure communication with an online banking service provider. Such assurance prevents local attackers from monitoring the keystrokes of the user (to obtain passwords), and the information displayed in the window (bank account numbers, etc). Secure I/O also prevents local attacks from sending keystrokes to initiate bank account transfers. The demarcated window is intended to indicate to the user that when interacting with such an application, more care should be taken, as the information is considered to be security-relevant.

3.4 Trusted Computing in Software

As discussion in Section 1.3, it is considered most appropriate to implement a trusted computing base with trust rooted in a hardware device. However, the manufacture of these hardware devices, such as the Trusted Platform Module (Section 3.2), has only recently begun. Prior to this, many attempts were made to provide what is now the level of assurance that hardware-driven trusted computing aims to provide, through software-only means. This section will discuss and outline some of these attempts, specifically focusing on methods which attempted to arrive at the three security primitives outlined in Section 2.3.

3.4.1 LOCK

Section 2.3.3 discussed sealed storage, and illustrated a method of using the `seal` primitive to form cryptographically secured computation paths. This concept of using the identity of an entity to evaluate and allow access to data is not new. The concept of using *object type* to set and enforce access restrictions to resources has been an area of research in computer science for some time.

The Logical Coprocessing Kernel (LOCK) system, developed in the late 1980s, is able to enforce security restrictions derived from strongly-typed data domains. LOCK is considered a strongly-typed Unix-like operating system. The LOCK system itself is introduced in [54], and further developed in [55]. Rogers and O'Brien in [46] describe the use of LOCK to design and build a LOCK-enabled application. As described in their paper, LOCK is:

> ...a highly assured INFOSEC [information security] system that can be used as a platform to develop countermeasures to current and future security threats. The system is based on a trusted computing base (TCB) that satisfies the security requirements defined for the A1 level in the *Trusted Computer System Evaluation Criteria*

> [16, *Orange Book*]... [and] uses a security coprocessor, called the SIDEARM, that makes access decisions ... [along with] LOCK's unique type enforcement mechanism.

The LOCK system provides security primitives equivalent to a current trusted computing system's sealed storage and curtained execution. It also goes some way to implementing secure I/O. The system described in [46] does not consider network operations, and so does not implement remote attestation.

Entities on a LOCK system are divided into two groups; *subjects*, which represent processes (programs), and *objects* which represent data and other resources on the system including hardware. Each subject is placed into a *domain*, and each object is given a *type*. The type of access permitted for each domain to each type is defined in a *Domain Definition Table* (DDT). A sample DDT, modified from [46, p 149], can be seen in Table 3.8. Capabilities that can be specified in the DDT are: r — read, w — write, a — append, e — execute, c — create, and d — destroy.

Table 3.8 shows how a subject in the domain *F1* is allowed `r`, read, access to objects of type *UnF1 Data*. A subject in the *DB* domain is able to read, write, create, and destroy objects of type *DB Data*, and can execute data of type *DB Code*.

A detailed explanation of how the LOCK system ensures that these access restrictions remain mandatory, and cannot be circumvented, will not not be given here. For the purposes of illustration, we will assume the access restrictions are mandatory. The purpose of this discussion is to highlight, with a (trusted) mandatory type enforcement mechanism, how trusted computing security primitives can be built.

Sealed Storage

As discussed in Section 2.3.3, sealed storage allows applications to store some arbitrary data and be assured that no other entity on the system is able to modify or view that data. The file system, a hardware resource type T_{fs}, is used to persist the arbitrary data. Each domain given access to T_{fs} is done so through intermediary access via a unique domain D_{int} into which an application in domain D_{sched} has only write-access. The file system, in its own domain, is able

Type / Domain	…	UnF1 Data	F1 Data	F1 Code	TrP1 Data	TrP1 Code	…	DB Data	DB Code
Pre F1		r, w c, d	-	-	-	-		-	-
F1	-	r	r, w, c, d	e	-	-	-	-	-
TrP1	-	-	r	-	r, w, tw, c, tc	e	-	-	-
…			-	-	r, d	-		-	-
DB	-	-	-	-	-	-	-	r, w, c, d	e
…			-	-	r, d	-		-	-

Table 3.8: Sample Domain Definition Table adapted from [46], listing domains on the vertical and types on the horizontal. A cell (i, j) contains the types of access a subject in domain i is permitted to make on an object of type j.

to only read from D_{int}. The file system on a LOCK system is constructed to enforce access restrictions based on the DDT table. To a LOCK-enabled application, it appears as though they have their own private file system. This construction of sealed storage in LOCK meets the definition as given in Section 2.3.3.

Curtained Execution

The *Domain Interaction Table* (DIT) defines allowed interactions between domains. The capabilities that can be specified in the DIT differ from those that can be specified in the DDT, to reflect the difference between types and domains. The DIT capabilities are observe, signal, create, and destroy. In this case, for an application in domain D_j to be able to observe an application executing in domain D_k, the cell (j,k) in the DDT must be explicitly marked with the `observe` capability. Interaction between domains must occur with through well-defined signals. For two domains, D_j and D_k, to communicate with each other, the cells (j,k) and (k, j) must be marked with the `signal` capability.

Secure I/O

From the two previous implementations described, it can be seen that implementing secure I/O in the LOCK system is trivial. Each hardware resource is designated with its own type, and placed in its own domain. Access to that domain is controlled through an intermediary arrangement of domains, as with sealed storage.

3.5 Aegis

AEGIS provides an assured, trusted environment to host a general purpose operating system. Execution begins with power-on, and will not pass to the next level unless that level passes cryptographic integrity checks. This process is referred to by Arbaugh et al. [18] as a *guaranteed secure* boot process.

Integrity verification in the AEGIS system is based upon level l_n being trusted to perform integrity verification of level l_{n+1}, before passing execution to that level. Level l_0 is presumed to be trusted, and does not have its integrity verified, in a meaningful way, by any other entity.

AEGIS makes some modifications to a normal IBM PC's boot process. These are comprised of partitioning the standard BIOS into two. The first, level l_0, contains the *trusted* components required for integrity verification of level l_1, including level l_1's precomputed hash. It also contains rudimentary code to replace code in level l_1, if it is found to be corrupted, from some external trusted source. This could be an AEGIS expansion card added into the machine, or some remote network host.

The AEGIS concept and threat model is relatively simplistic compared with more advanced trusted computing models. Arbaugh assumes [18, p 66]:

> ...the motherboard, processor, and a portion of the system ROM (BIOS) are not compromised, *i.e.*, the adversary is unable or unwilling to replace the motherboard or BIOS.

Although not explicitly stated, it seems reasonable to assume that the AEGIS architecture is intended to operate in a corporate environment, where hardware can be protected from user replacement or modification through traditional methods, and the administrator is not an adversary, but interested in securing the system. The AEGIS system is designed to protect a computing environment from a *malicious user*, not a *malicious administrator* (see Section 1.2).

AEGIS is one of the first attempts to enable the creation of a known-secure environment on commodity hardware, without major modifications to the underlying architecture. The system is very brittle — any modification to the components in the measurement chain results in the system failing to boot. The executing operating system is able to presuppose that, because it is executing, it can trust the lower layers.

It is useful to include in our discussion the formal definition of AEGIS' trust model. The AEGIS n to $n+1$ method of inducing trust from lower layers to higher layers is analogous to many other trusted computing models, which do not state their assumptions or reliance on a trusted root so explicitly. Arbaugh et al. give two recurrence equations showing the *chaining* of trust, from level l_0 [18, p 69].

$$\begin{aligned} I_0 &= True, \\ I_{i+1} &= \left\{ I_i \bigwedge V_i(L_{i+1}) \quad \text{for} \quad 0 < i \leq 4 \right. \end{aligned} \tag{3.6}$$

Here, l_i is a boolean value giving the integrity of level i, $\bigwedge$ is the boolean *and* operation. V_i is the verification function associated with the i^{th} level, taking the level to verify as its argument, and returning a boolean. As described earlier, AEGIS performs a cryptographic hash of the level being verified, and compares the resultant hash with a precomputed value. This value is retrieved from a previously verified level; either the currently executing level, or some other lower level.

This recurrence equation (3.6) explicitly states the axiomatic trust placed in level l_0. During operation, level l_0 is trusted to not allow execution to continue onto level l_1 if it fails to pass a verification check. This behaviour is also expected of all other levels. Assuming each level is functioning correctly, level 5, consisting of user-level applications, will be hosted in a well-known environment of levels 0–4.

The AEGIS system is not intended to directly provide any of the three primitives of a trusted computing framework. However, it does restrict the boot process to only allow well-known software to begin executing on the machine. It could be used to boot a software system like LOCK (see Section 3.4.1 on page 82). The LOCK operating system would then be guaranteed to be executing on benign hardware with unmodified software configured to properly enforce the primitives.

3.6 IBM 4758

Work by IBM in the late 1990s lead to the development of the IBM 4758 Secure Coprocessor [4]. It was developed to provide IBM's Common Cryptographic Architecture product group with a *tamper-responsive*, general purpose, secure coprocessor. A tamper-responsive device performs some action when it is tampered with. A tamper-resistant device merely makes attempts to subvert the device difficult. The design goals and challenges are discussed by Dyer et al. in [24]. The 4758 takes the form of a PCI-bus expansion card, that can be added to an unmodified IBM PC.

While the implementation and design of all aspects of the 4758 is not immediately relevant, it does implement an interesting form of attestation. Called *outbound authentication*, it enables a 4758 card to export information about its current internal state. The manufacturing process of the card is also similar to that of trusted platform modules, used as the trusted root in other trusted computing frameworks (see Sections 3.3 and 3.2). The 4758 can be seen as the precursor device to the trusted platform module.

Shown in Figure 3.4, the IBM 4758 is heavily protected from physical tampering. Upon detecting a physical attack, the IBM 4758 is designed to clear its internal secrets and destroy certificates embedded during manufacture. The 4758 is sensitive to *"physical penetration, power sequencing, temperature, and radiation [attacks]"*[3].

The design goals, decisions, and processes were described by Dyer et al. [24], and show their consideration of the same design principals used to construct the LOCK software system. The logical design structure consists of three major components; the hardware, the firmware, and the end-user supplied software. These are shown in Table 3.9, including their further cate-

Supplier	*Category*	*Components*	*Relative Trust*
Users	Software	Applications Operating System Environment Kernel Loaders	Low
IBM	Firmware	Post Miniboot	Medium
IBM	Hardware	Processor Flash, RAM and ROM Locks Tamper-responding unit Crypto functions	High

Table 3.9: Components of the IBM 4758, logically split into the familiar categories of hardware, firmware, and software, including their logical layers. The relative level of trust in each layer decreases from hardware, to firmware, to software.

gorisations.

Similar to the AEGIS system, trust in any layer above the first is inherited from the lower layer. The hardware layer is manufactured by IBM, as is the firmware layer. The software layer is supplied by the card owner, and is designed to run on the embedded 486-class processor.

As mentioned above, the IBM 4758 is capable of outbound authentication, which is analogous to attestation. It also implements a form of secure boot, similar to AEGIS. Although not intended to be part of a trusted computing framework, the 4758 does provide persistent storage. The 4758's method of implementing attestation is illustrative, as it is similar in method to that chosen by IBM researchers when working on the TCG's Trusted Platform Module (see sections 3.2 on page 51, and 4.4.2 on page 111).

During the manufacturing process a root certificate, used for attestation, is embedded in the device. This root certificate is signed by keys derived from IBM's root key, and attests that the named public key is associated with a private key also embedded in the device during manufacture. It is from this certificate that trust in the identity of the device is obtained, as explained in Section 2.3.5. Trust in attestation statements made by the device are dependent on IBM's root keys remaining uncompromised.

Figure 3.4: Photograph of the IBM 4758 secure coprocessor.

The IBM 4758's firmware and software stack is delineated into *layers*. The initial layer, consisting of an asymmetric key pair, manufacturers certificate, and minimal operating code is embedded in ROM during this manufacturing process. It is the only layer which is inserted into the device without a public key authentication operation taking place.

Layers are used to break up software on the 4758 into separate pieces, along functional and trustworthy lines, as in LOCK. Each software load operation results in a new logical layer in the device, and each operation must be permitted by the *owner* of the previous software layer. This *authentication chaining* of ownership, most of which takes place during manufacturing, layers the *miniboot* (layer 0) and *boot* (layer 1) firmware layers, seen in Table 3.9, directly above the ROM code. Ownership of layer N allows that owner to name and install code into layer $N+1$.

Each of the layers (1 and 2) programmable in the field have various information associated with them. Each layer has an ownership id; the layer 2 owner ID is assigned by IBM, and the layer 3 owner id is assigned by the owner of layer 2. The content of each layer also has data stored about it. This consists of an image name and revision field, specified by the owner of that layer, and most importantly a *hash field*. This hash field is derived, by layers 0 and 1 inside the device, from the entire contents of that layer's code.

When delivered to the purchaser, the 4758 is configured to allow only one further software

Component	*Explanation*
A nonce	This nonce is originally specified by the requester, so they can be assured of the vector's freshness.
The Vital Product Data (VPD)	This specifies the exact model of the IBM 4758.
Whether layer 2 and layer 3 are owned and have reliable contents	Indicates if layers 2 and 3 have become owned since being shipped from IBM, and that the physical tamper-responsiveness has not activated.
The layer 2 OwnerID and layer 3 OwnerID	A two byte field, indicating owning entity of layer N, set by the owner of the $N-1$ layer.
If layer 2 is owned and reliable, the name of the image it contains	The human-readable name of the installed software.
If layer 3 is owned and reliable, the name of the image it contains	The human-readable name of the installed software.

Table 3.10: Components of an IBM 4758 attestation (remote authentication) vector, adapted from [60].

load — that of the end-user's custom application. With only one user application executing on the device, it is unable to launch an attack against any other application on the device, or be attacked itself from another user-level application. Only attacks against lower layers are possible, attempting to subvert some given operation of the device. The code for these lower layers has, by the time execution passes to layers 2 and 3, been protected in hardware by advancing *hardware ratchets*. These ratchets protect the firmware from modification until the device is next reset, and the ratchets released. This 'single-application' design is a major restriction in the use of the IBM 4758 as a general-purpose trusted computing platform.

The operating system, a variant of the CP/Q operating system known as CP/Q++ [29], is loaded into layer 2, and is usually included when shipped from IBM. The end user application is loaded into layer 3. The attestation procedure involves a signed vector being transmitted from the device, in response to a user query. IBM outlines the procedure in a white paper [60], as well as the information found in the response. This is reproduced, with modifications and explanations, in Table 3.10.

This vector is signed with the unique key of layer 0 in that device, which in turn is named in a certificate chain that ends with IBM's root key. By trusting the tamper-proof hardware, and the code in layers 0 and 1, the user can trust that the information contained in the vector is correct. They are then able to make an informed decision about trusting the entities named as owners of layers 2 and 3. It should be noted that the attestation vector described here does not include the hash fields of layers 2 and 3. This data is available through another API call, and is used in the Reusable Proof of Work (RPOW) concept server, described in Section 3.6.1.

The IBM 4758 device also provides a form of sealed storage. The 4758 is able to store data across power cycles in flash memory. Flash memory chips are also found in the ubiquitous USB key-chain storage devices. Application data is encrypted when stored in flash memory. The device is unable to guarantee the wiping of data stored in flash memory if the device is physically attacked. Therefore it is encrypted with a symmetric cipher; the key is stored in battery-backed RAM (BBRAM). BBRAM is guaranteed to be wiped in response to physical attack, thereby rendering the data encrypted in flash unrecoverable.

The 4758's implementation of secure storage illustrates a common pattern in trusted computing frameworks. A small, secure, and expensive portion of memory can be used to satisfactorily secure data stored on larger, less secure, and cheaper memory. This leveraging of trusted storage could have been used to provide persistent storage on the host device; denial of service attacks would then become far easier.

3.6.1 Reusable Proof of Work

The Reusable Proof of Work [8] (RPOW) server is a proof-of-concept implementation on an IBM 4758 device. It makes excellent usage of the outbound authentication capability of the 4758 card. The system itself allows *hashcash* (also known as Proof of Work) tokens [2], instead of being disposed after a single use, to be exchanged for another of equal value, and addressed to another recipient. In essence, this allows one hashcash instance to be used multiple times by two communicating individuals. Additionally, users are able to trade in one POW token, with a given value x, for two POW tokens both of value $x/2$. The RPOW server acts as a virtual bank, enabling the exchange of POW tokens to take place in a similar manner to a real bank.

All software used in the system is available through an open source license, allowing end users to verify and compile the client they use to interact with the server. They are also able to verify the source code for the software that runs on the 4758 device itself, as well as the host software running on the server PC. The client software, during each interaction with the RPOW server, requests the generated hash of the card's running software. This value is compared against a precomputed value distributed with the client application. This computed hash value is signed and delivered in a data structure similar to the one outlined in Section 3.6 on page 87. Taken together, the two attestation vectors assure the client of the validity of, firstly the IBM 4758 device itself, and secondly the hashed identity of its executing code.

3.7 Execute Only Memory

Execute Only Memory (XOM) was developed at Stanford University [64] in 2000. XOM (pronounced *"zom"*) was originally intended to prevent unauthorised software *execution*. Later research by a number of the same authors [37] led to the implementation of an untrusted operating system running on XOM hardware. One section of their paper described how components of XOM Operating System (XOMOS) were able to implement the three parts of a trusted computing platform (see Section 2.3 on page 23). While the XOM architecture was not developed to form part of a Trusted Computing Framework (TCF), it is a simple, elegant architecture.

The rest of this section will briefly describe the hardware architecture of XOM, and how the trusted computing aspects of XOMOS operate.

3.7.1 Concepts and Hardware Architecture

The XOM architecture introduces a number of concepts similar to those found in most trusted computing platforms, such as an axiomatically trusted hardware layer. While no functioning XOM system has been built in hardware, it has been modelled in software on a MIPS with IRIX system [37].

The XOM architecture, as described by Lie et al. in [64], relies on a modified processor that provides three key operational extensions. The first operation allows for asymmetric decryption of a data block using a private key embedded securely within the processor. The second operation allows symmetric decryption of an instruction stream, with what is known as a *session key*. The third and final operation provides, and enforces, compartmentalised storage that prevents one process from viewing the contents of any register not marked as belonging to its own compartment. Additional hardware modifications include a number of XOM-specific state bits on register and cache lines. Lie et al. [37] briefly describe the benefits of compartmented execution:

> XOM uses its master secret to protect programs by supporting *compartments*. A compartment is a logical container that prevents information from flowing into or out of it. A process in a compartment is immune to both *modification* and *observation*.

The XOM chip contains, at its heart, an asymmetric key pair which is created or embedded during manufacture. As in other trusted computing architectures the private key of this key pair should never be transmitted outside the chip. While not specified in the original literature describing XOM, the public key should be signed by the manufacturer. This asserts that the certified public key belongs to an actual XOM device, and allows it to verify its identity.

3.7.2 Application Distribution and Execution

The distribution process, as described by Lie et al. [64], is seen in Table 3.11. The process begins with a user on a XOM platform χ requesting an application P from the distributor, D. The XOM chip, as with the TCG's TPM chip, contains a unique key pair $\chi_{pub,prv}$, embedded during manufacture. The distributor D, upon receiving a request from χ, encrypts their application P with a unique symmetric key K_s. The encryption, $E(K_s,P)$ results in a blob β. The symmetric key K_s is encrypted with the public key χ_{pub} of the requesting platform. The result of $E(\chi_{pub},K_s)$ is an encrypted blob α. These two blobs are concatenated into a data structure, $\{\alpha||\beta\}$, which is returned to χ. This data structure allows the execution of P on χ, analogous to the execution of P normally. At this point, χ has been supplied with a customised version of P that only it is able to decrypt and execute.

The distribution and execution scheme described above ensures the integrity and confiden-

Step	*Action*
1.	$\chi \rightarrow D$: `AppReq(`χ_{pub}`)`
2a.	D: verify χ_{pub}
2b.	D: generate K_s
2c.	D: perform $E(K_s, P) \Rightarrow \beta$
2d.	D: perform $E(\chi_{pub}, K_s) \Rightarrow \alpha$
3.	$D \rightarrow \chi$: `AppRep(`$\{\alpha \| \beta\}$`)`
	...
4a.	χ: load $\{\alpha \| \beta\}$
4b.	χ: perform $D(\chi_{prv}, \alpha) \Rightarrow K_s$
5a.	χ: perform $D(K_s, \beta) \Rightarrow P$
5b.	χ: execute P

Table 3.11: Packaging and distribution protocol of XOM, used to ensure an application's integrity and confidentiality throughout its lifetime.

tiality of P from its packaging by D until it is decrypted inside χ. The application P exists in unencrypted form only in the L1 and L2 caches and registers of χ. The XOM architecture implements compartments to prevent another application Q from viewing or modifying P while it is unencrypted, and stored in the CPU registers. This compartmentalisation, discussed below, satisfies the requirements of curtained memory defined in Section 2.3.4. Specifically, it satisfies the requirements of *strong* curtained memory, given in Definition 2.3 on page 41.

At some later point in time, χ is able to execute P without further interaction with D. To begin with, the XOM chip in χ decrypts α, $D(\chi_{prv}, \alpha)$. The resulting symmetric key K_s is not released outside of the XOM chip. During the execution of P, β is stored in main memory, and is decrypted with K_s inside the XOM CPU, and stored decrypted in the L2 memory and L1 instruction cache. Data created and used by P during execution is stored in memory using K_s.

During the loading and decryption of P, a new entry for the program is made in the *XOM key table* containing, amongst other information, the symmetric key just obtained. A *compartment ID* is also generated, and this is used to reference the table to obtain the symmetric key when required. Entries in the key table represent what are known as *principals*, analogous to executing processes on a normal processor. The currently executing principal on the XOM chip is known as the *active principal*, and its tag is stored in the *XOM ID register*.

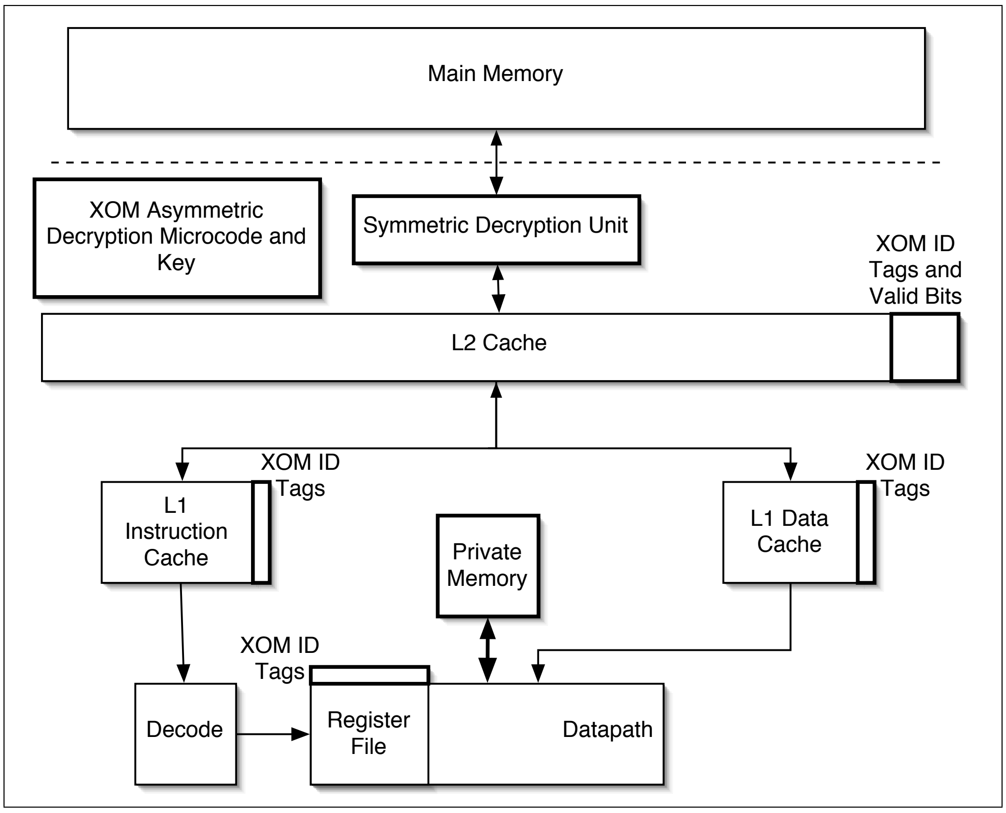


Figure 3.5: Overview of XOM architecture, adapted from [64]. Bold black borders indicate new components to a traditional CPU architecture.

As shown in Figure 3.5, the instruction stream is brought in from main memory, then decrypted and stored in the L1 instruction cache, on each cache miss. Also shown in Figure 3.5 is the state required in the XOM chip. There must be an appropriately sized *private memory* to allow the XOM processor to maintain the key table. The maximum key table size limits the number of concurrently executing principals. The minimum table size is 2 lines — one for the *null* compartment, and one for an active principal. The null compartment is active when the XOM processor is executing normal (non-XOM encrypted) code.

The XOM key table itself is made up of two sub-tables, the *Register Key Table* and the *Compartment Key Table*. The *XOM ID Register* contains the compartment ID of the currently active principal. This compartment ID is used, in the Register Key Table, to denote the compartment the data in a register belongs to. If the register contains data belonging to a process executing in the null compartment, that is also recorded here. Every time the active principal tries to read from a register, the ownership of that register is checked against the Register Key Table. If the register tag does not match that in the XOM ID Register, an exception is thrown by the XOM hardware. As data is stored unencrypted in registers on the XOM chip, it is this mechanism, implemented in hardware, that prevents principals from viewing another's memory space.

Table 3.12 shows the two commands, `xsd` and `xld`, used by an application executing in a compartment to save and load data securely. As an executing principal generates data, it is stored in memory with the `xsd` command. Initially, the data is stored unencrypted in an L1 or L2 cache line. The specific line is tagged with the compartment ID of the active principal. The XOM CPU prevents principals from reading cache lines that are tagged with a different compartment ID to its own. These tags are present on all caches inside the XOM CPU, and are shown in Figure 3.5.

The process described above ensures the confidentiality and integrity of instructions and data when it is unencrypted in the L2 and L1 caches, and in the register files. Data generated by a program *P* executing in a compartment is also secured when it is evicted from the L2 cache, and stored in main memory. Figure 3.5 shows the encryption and decryption of data occurring at the boundary between the L2 cache and main memory.

When a cache line is evicted, the XOM CPU retrieves the symmetric compartment key from the compartment table, and uses it to encrypt the data before it is stored in memory. This ensures the integrity of the data. To ensure that data is not tampered with when stored in main memory, the data is hashed. The hash used is a keyed cryptographic hash, or message authentication code (MAC) [32, p.181]. Lie et al. describe the process of storing and retrieving data [37]:

> Each time a cache line is written to memory, a hash of it is generated, and both the hash and the cache line are encrypted. The hash pre-image contains both the virtual address and the value of the cache line. When decrypting the cache line, a matching hash must also be loaded before the XOM processor will accept the encrypted value as valid.

In order to implement a properly multi-tasking operating system with the XOM chip, the untrusted operating system must be able to save the state of the active principal when it is interrupted. The operating system, in layer l_1, is intended to execute in the null compartment. The XOM architecture allows an untrusted operating system in layer l_1 to schedule and interrupt

Instruction	*Explanation*
`xsd $rt, offset($base)`	Stores register `$rt` to memory, tagging the cache line with the active principal's compartment ID.
`xld $rt, offset($base)`	Loads the specified memory address into register `$rt`. If the load results in a cache miss, the data is retrieved from main memory, its associated hash is validated, and the value decrypted and stored in the specified register.
`xgetid $rt, $rd`	Get the register tag value of `$rt` and store it in `$rd`.
`xenc $rt, $rd`	Check that the ownership tag of `$rt` matches that of `$rd`. If so, use the contents of `$rd` to index the Register Key Table to locate the corresponding register key. Encrypt the contents of `$rt` with this key and place it in XOM private memory registers `$0...$3`.
`xsave $rt, offset($base)`	Store `$rt`, one of the private memory registers `$0...$3`, to the memory address specified. The cache line is tagged with the `null` compartment.
`xrstr $rt offset($base)`	Fill `$rt`, one of the private memory registers `$0...$3`, with the encrypted value to be restored, located at the specified memory address.
`xdec $rt, $rd`	Use the contents of `$rd` to index the Register Key Table to locate a register key. Decrypt the 256 bit value set by `xrstr`, validate the result and restore to register `$rt`. Set the ownership tag of `$rt` using the contents of `$rd`.
`xmvtn $rt`	Sets the compartment tag of `$rt` to null, as long as the register `$rt` belongs to the active principal.
`xmvfn $rt`	Sets the compartment tag of `$rt` to that of the active principal, as long as the register `$rt` is in the null compartment.

Table 3.12: Adapted from [37], additional processor operations implemented in hardware to support XOM. Original hardware is described in [64], while the new hardware is described in [37]

an active principal P executing in a compartment in layer l_2, without resulting in a loss of confidentiality or integrity for P. XOM implements an on-chip method of securely saving and restoring the state of an active principal when it is interrupted.

Shown in Table 3.12, the commands `xgetid`, `xenc`, and `xsave`, allow the operating system, executing in layer l_1, to securely save the state information of an application executing in layer l_2 to main memory, without being able to view or modify that state itself. The commands `xrstr` and `xdec` are used to restore the context of a principal when it is scheduled by the operating system.

The `xgetid` command retrieves the compartment ID of the specified register `$rt` and stores it in `$rd`. The `$xenc` command ensures that the ownership tag of `$rt` is the same as the tag stored in `$rd`. If it is, `$xenc` continues and encrypts the contents of with the correct register key. The encryption process is outlined below.

The 64 bit register is encrypted with the register key, and stored in the first of four reserved XOM register files, `$0`. The register ownership tag and register number are stored in `$1`. A 128 bit MAC is generated over registers `$0` and `$1`, and stored in `$2` and `$3`.

This 256 bit word is then stored in main memory at a location specified by an argument to the `xsave` command. It is first cached in the L2 cache line, where it is tagged with the null compartment. This allows the untrusted operating system to manage the encrypted context. This process is repeated for all other registers in the ownership of the active principal.

To restore the context of a principal, the encryption process is reserved. The `xrstr` command retrieves the 256 bit word, and stores it in the private registers `$0...$3`. The integrity of the registers `$0` and `$1` are checked by calculating their hash, and comparing it with the hash stored in `$2` and `$3`. If the integrity check succeeds, the register key specified in `$1` is retrieved, and used to decrypt the value in `$0`. The decrypted value is stored back in the appropriate register, again retrieved from register `$1`.

The register key used in the above process is distinct from the symmetric key used to decrypt the instruction stream and encrypt and decrypt data stored in main memory. The register key is generated by the XOM CPU, and is regenerated for each compartment each time a process switch occurs. This prevents an attacker from replaying encrypted register values from previous context switches. The register key is changed after a compartment's context is restored.

The processes described above enable strong curtained execution for all principals $P_1,...,P_n$ that execute in a compartment, by assuring the confidentiality and integrity of their individual memory ranges. In addition, a principal in layer l_2 is protected during execution from a malicious operating system in layer l_1.

To enable principals executing in a compartment to communicate with other applications, and process I/O, the XOM CPU provides two commands, seen in Table 3.12. These are the move-to-null command `xmvtn`, and the move-from-null command `xmvfn`. These commands simply change the associated compartment tag of a register, allowing data to be brought into or pushed out of a principal's secure memory space.

3.7.3 Aspects of Trusted Computing

The XOM architecture, originally intended to prevent unauthorised execution, also provides a form of attestation and sealed storage. Using the XOM architecture as the base of a trusted computing framework is discussed only tangentially by Lie et al. in [37, p.190].

A XOM application attests to a third party by way of them both knowing some shared secret, say K_{shd}. The method of preparing an application for distribution is outlined in Section 3.7.2 on page 93. During this preparation, a secret key K_{shd} is embedded in the application image. This can either be personalised for each customer, or be the same key for all copies of the application. If the key is the same for all distributed copies of the application, two copies of that application may attest to each other in a straight forward manner. To communicate securely, the two copies of the XOM application simply encrypt their messages with the shared secret Sh_k they both have. As long as the symmetric keys which encrypt the application images are not decrypted outside a XOM chip, the shared secret remains secure.

The model of attestation for XOM has one shared secret amongst all copies of the application. This is vulnerable to a *break once, break everywhere* (BOBE) style failure. To prevent one compromised application invalidating the integrity of all others, each distributed application must be customised, by embedding unique key pairs and certificates, before distribution to the user's machine. Specifically, more customisation than just differing headers for each XOM CPU is required.

The XOM architecture implements sealed storage by making use of cryptographic keys embedded in the encrypted binary image, similar to attestation. Given that an application P must be unmodified in order to successfully execute on a platform, P can use a symmetric key embedded in its own image to encrypt data before moving it to the null compartment and storing it on untrusted, long-term medium. As with attestation, each application image must be customised with a unique key to prevent a BOBE-class vulnerability, and to ensure that data encrypted on one platform cannot be encrypted on another.

The security primitives provided by the XOM architecture rely on the correct operation of the software distributor *D*. The distributor *D* is required to customise a XOM application *P* by embedding keys in the image for use with attestation and sealed storage, and encrypt the whole of *P* with a unique symmetric key as described in Table 3.11. These keys should be unique to *P*, and no record of them should be kept by *D*. A XOM-enabled application relies on the correct operation of the XOM CPU, but is also dependent on the integrity of the distributor to properly customise *P*.

He who is firmly seated in authority soon learns to think security, and not progress, the highest lesson of statecraft.

James Russell Lowell

4 Discussion

4.1 Introduction

This chapter explores a number of issues with the implementations discussed in Chapter 3.1, and describes some of the open problems in trusted computing that they highlight.

Section 4.2 discusses frameworks that only implement part of a trusted computing framework. Section 4.3 discusses the difficulties with generating trusted code identities (measurements) for software. Section 4.4 discusses the difficulties with implementing shared libraries, and including them in statements of a programs functionality. Section 4.5 discusses resistance to software attacks.

4.2 Partial Framework Implementations

4.2.1 LOCK

It is instructive to observe the differences between modern trusted computing platforms and the LOCK system as a whole. Current trusted computing platforms are strongly tied to the identity of the application. LOCK, on the other hand, concerns itself more with information flow inside a specific application. Moving an existing application to the LOCK platform requires considerable thought during design to correctly modularise the application. Rogers and O'Brien describe this in [46]:

> Rather than just decomposing the application along functional lines, it must also be partitioned along security and integrity lines. The application designer must identify the components of the application that require added security or integrity, and modularise the application to isolate those components in separate subjects...
>
> The design goal is to put each different security or integrity relevant task into its own subject that runs in a distinct domain, and to isolate the data that these subjects must handle into special types.

Discussed in Section 3.3, Microsoft intends NGSCB to be used in much the same way. Those parts of an application which perform security-relevant computations should be imple-

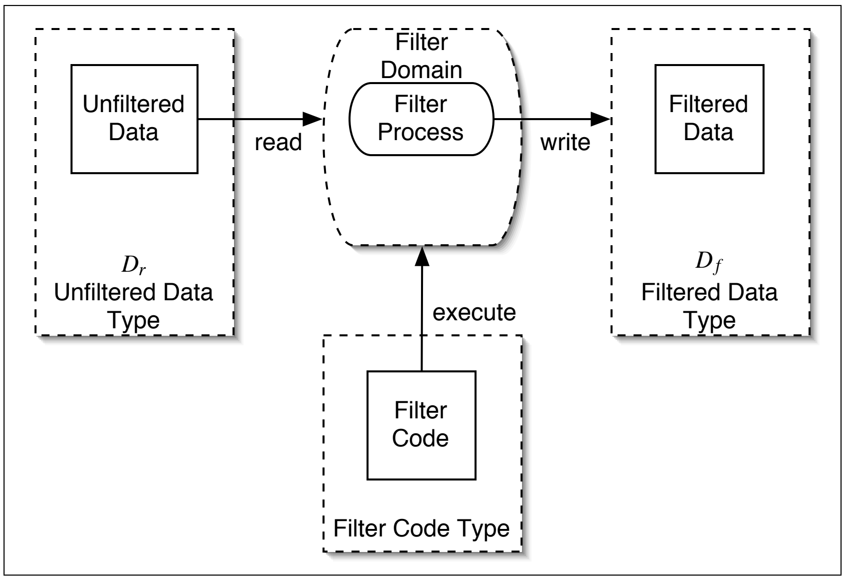


Figure 4.1: A filter process illustrating an assured pipeline as implemented in the LOCK system, adapted from [46].

mented separately, and partitioned accordingly, so as to be implemented in the secure Right Hand Side (RHS). Each module should perform its service or computation for the remainder of the application, with interaction occurring through well-defined interfaces. This method of system design and construction is well known, and is similar to object-orientated software design.

Mentioned in Section 2.3.3 was the use of sealed storage, based on the identity of the application, to create *assured pipelines* of computation. The type enforcement architecture of LOCK lends itself ideally to the construction of these assured pipelines in software. An example of such an assured pipeline is given in [46], through the construction of a filter to process data. This example is shown in Figure 4.1. Shown is unfiltered (raw) data, D_r, being read by a filter process, P_f. The filter domain is only able to read data of the type D_r, and only able to write filtered data of the type D_f. Also shown is the filtering code, marked in the system's DDT as being only able to execute in the filtering domain. As only unfiltered data of type D_r can be read into that domain, the filter is assured of operating correctly, filtering only appropriate data, and writing it, once processed, to only the correct type.

As can be seen, the LOCK system is highly reliant on type definitions. During system and application design, the designer must define named domains or types for all entities. This designated name is relevant only on the local system itself.

This is in contrast to modern trusted computing systems, where the identity of the application is given by a secure hash derived from the executing code, and is invariant for all platforms. The trust in the invariance of the derived hash is dependent on trusting layer l_0 to operate correctly. On a LOCK system, layer l_0 is a secure coprocessor trusted to enforce domain interactions correctly. However domain, type, subject and role information is created by a system administrator and stored in layer l_1. This is trusted axiomatically, like layer l_0, because the LOCK system was not designed to withstand attacks from a local administrator, only a local user. A LOCK system is capable of providing and enforcing all of the trusted computing primitives except attestation.

A LOCK system is intended to be designed, developed, and verified as a whole, in order to ensure that the relevant security policies are correctly enforced. Once this process has been completed, a LOCK system remains relatively static. Code intended to run on a LOCK system must be compiled elsewhere and introduced to the system by an administrator, who ensures that the required additions and changes to the LOCK Domain Definition and Domain Interaction Tables to not compromise any security policy.

It should be obvious that the complicated development procedure required to use a system such as LOCK precludes the use of the platform as a usable general purpose computing device (Section 2.2).

4.2.2 Hardware Assured Security

The AEGIS, IBM 4758, and XOM architectures all assure their security primitives with an axiomatically-trusted layer l_0 hardware device.

The IBM 4758 and XOM architectures are capable of withstanding attacks on the confidentiality and integrity of their assured programs and data from a malicious system administrator. Both rely on an untrusted host in order to ensure availability. A malicious system user or administrator is able to perform a naive denial of service attack by simply turning the untrusted host off. A more advanced denial of service attack, that does not reduce the usability of the system for the attacker, can be accomplished by a system administrator for both systems. A malicious operating system, running on the XOM architecture, is capable of preventing XOM-enabled applications from executing by either starving their scheduled execution time, or modifying the encrypted application image. An IBM 4758, able to be administered by a distinct entity to that of the host platform it runs in, is able to ensure the long term availability of a limited amount of data. However, it is limited in internal capacity and in practise will need to rely on the untrusted host for storage of arbitrarily sized data. These two examples serve to show that assuring availability is considerably harder, and more expensive, than assuring confidentiality and integrity.

The AEGIS system is not intended to provide secure booting in the face of a malicious system administrator. To be of practical use, the integrity measurements AEGIS validates to ensure layer l_{n+1} has not been tampered with must be able to be upgraded. Software environments, as discussed in Section 2.2, require the installation of patches and upgrades in order to remain secure in the face of newly-discovered vulnerabilities and bugs. This process requires a system administrator capable of generating the required integrity measurements and installing them in layers l_0, l_{n-1} where n is the last layer measured for integrity.

In order to prevent naive denial of service attacks, and guarantee availability in the face of non-malicious corruption of a layer l_i, where $i > 0$, the AEGIS architecture is capable of retrieving valid copies of upper layer software. For this, a layer l_i must be able to contact a trusted, secure host to retrieve a copy of the software component in layer l_{i+1}. This retrieval method is only viable in the face of naive DoS attacks through modification and non-malicious corruption of layer l_{i+1}. Performing a denial of service attack on a typical network transport layer is trivial. This guarantee of availability can be strengthened through appropriate securing of the transport layer used to connect with the trusted host. Again, the AEGIS system shows the difficulty and cost in assuring availability when compared with integrity and confidentiality.

The XOM implementations of sealed storage and attestation are unsatisfactory, as described in Section 3.7.3 on page 99, and rely heavily on the behaviour of an application distributor D. Even if D is trusted to behave correctly, the distribution method is necessarily inefficient, requiring the individual customisation of each application copy. It is possible to imagine the use of XOM, as initially described by Lie et al. [37], to be appropriate for only a few classes of applications. An application distributor D would be required to implement the procedure outlined above for each copy of their application downloaded over the Internet. Applications could be customised, encrypted and stored on CD/DVD media for later distribution, but a user is still required to interact with the distributor to obtain the unique symmetric key used to encrypt their copy.

4.3 Generating Trusted Measurements of Applications

Section 2.1 on page 7 discusses the motivations for an open source trusted computing framework. An open source framework allows a user to trust the code based on her own examination, or to make a decision to trust the code on the basis of its examination by a group, or groups, of people that they trust. Section 2.3.2 on page 28 discusses the importance of a cryptographically secure code identity $ID(P)$ to strongly identify the functionality of a program P.

With a software environment capable of compiling P, a user is able to obtain the source code of P, and compile it herself. She can then generate $ID(P)$ and obtain the code ID of a P compiled from known source, in a known environment. Recall the assertion of Thompson [65]: a user cannot trust the compiled P any more than she trusts the software environment that was used to compile it (assembler, linker, compiler, etc). This implies that a user who highly depends on the integrity of P, measured by $ID(P)$, must only trust a measurement $ID(P)$ she has generated herself.

A measurement $ID(P)$ of P that a user does not generate herself does not guarantee the integrity or correctness of P itself. Instead, repeated presentations of $ID(P)$ in an attestation vector only guarantee the non-modification of P from challenge to challenge.

Let a platform ρ, used to compile P from source, generate a P^{ρ} that gives a measurement $ID(P^{\rho})$. Two platforms ρ and ρ', used to compile identical copies of the source code of P, are unlikely to produce identical binaries of P. This results in distinct measurements of P, $ID(P^{\rho}) \neq ID(P^{\rho'})$. Differences in the compiled versions of P, P', as measured by comparing $ID(P) \neq ID(P')$, result from the compilation of $P \neq P'$. These differences cannot be distinguished from $ID(P) \neq ID(P')$ resulting from differences in the software environments used for the compilation of $P = P'$ on ρ and ρ'.

Reconciling the disconnection between the source of P and a compiled version of P identified by $ID(P)$ in general is an open problem in trusted computing. The RPOW architecture (Section 3.6.1 on page 91) has solved this problem for one application through publishing the specific software environment used to compile the software that executes on the IBM 4758.

Recall the Reusable Proof-of-Work (RPOW) server discussed in Section 3.6.1 on page 91. The RPOW system administrator [8] makes available the complete compilation environment on the platform ρ' used to generate the binary running on the IBM 4758 server. The specifications given include the GCC version number, as well as other tools that affect the final binary. A user is able to reproduce this software environment on their own machine ρ, and generate an identical P, such that $ID(P^{\rho}) = ID(P^{\rho'})$, compiled from source. This solution cannot practically be repeated for every piece of software that a user wishes to trust. Below we propose a number of possible approaches to solving this problem in general.

1. An open trusted computing framework community could prescribe an approved, official (and necessarily open) software environment to be used to compile applications.
2. Attestation vectors could include compilation environments for each P, along with the generated $ID(P)$.
3. Distributed compilations by members of the community lead to many differing versions of P^i, resulting in many $ID(P^i)$ that are considered correct compilations of P from source.

Proposal 1 would allow a user to recreate the software environment used to compile a specific application P, and to generate an identical measurement $ID(P)$ for P as presented in an attestation vector by another party. The difficulties in managing prescribed software environments for arbitrarily large numbers of programs would be non-trivial. This approach has the benefit of allowing a naive user to trust a group, or groups, to maintain their own software environment, as prescribed by the framework community. These groups would check the integrity of a compiled P, such that $ID(P) = ID(P')$, and publish matches and discrepancies as required.

Proposal 2 allows software authors to control the software environment used to compile P themselves. The inclusion of a statement about the compilation environment with all applications would require a complicated PKI to ensure the integrity and validity of the statements. This is equally non-trivial.

Proposal 3 relies on a secure server to store multiple computed measurements $ID(P)$ of P generated by individual members of the community. An attestation protocol would contact one of these servers and submit $ID(P)$ for each application in the vector as required. The server would return information about P such as the application name, version, and author as well as the compilation environment used to generate P^i. It would also return the number of times the measurement $ID(P^i)$ had been submitted for P, as well as other measurements of P generated in different compilation environments.

A user could make a decision to trust P based on a statistical calculation of the number of people that generated $ID(P^i)$ for the P in question, compared with the number of people who generated a distinct $ID(P^j)$. For example, a single measurement $ID(P^i)$ with a significantly greater number of measurements $ID(P^j)$, could lead a user to conclude that P had been maliciously modified $P \rightarrow P'$ before compilation, and the measurement $ID(P')$ submitted to the server in an attempt to fool her.

This method relies on the assumption that, as the number of people n performing non-malicious compilations and measurements of P increases, the frequency of distinct measurements $ID(P)$ for P being generated decreases, i.e. that there is a finite number of compilation environments that generate distinct $ID(P) \neq ID(P)$, for $P = P$. This assumption may not be correct.

4.4 Shared Libraries

It is important to note the differences between a XOM attestation and an attestation of the NGSCB and the TCG frameworks (see Sections 3.3 on page 76 and 3.2 on page 51 respectively). In XOM, the validity of an application's attestation is based on the assumption that the symmetric key which encrypts the application binary is never exposed outside a XOM chip.

Also, standard cryptographic techniques are employed to ensure that a modified application image cannot be executed successfully. Contrasted with NGSCB and TCG models, the end-user's trust in the validity of an attestation requires trust in the distributor of that application to properly ensure the confidentiality of the shared secret, as well as trust in the manufacturer of the XOM CPU. An NGSCB or TCG attestation is considered valid if the manufacturer of the TPM device ensures the key used to sign the attestation vector is kept secret. These differing attestation procedures result in differing methods of properly measuring shared, or dynamically-linked, libraries as discussed in Section 2.3.2.

The XOM architecture, designed by Lie et al. [37] allows an untrusted software agent, the operating system in layer l_1, to manage a trusted resource (the XOM CPU), and allow software in layer l_2 to execute on that resource, while assuring its integrity and confidentiality. With the trend in modern day software application architectures away from monolithic application code, and towards using dynamically shared libraries executing in either layer l_1 or l_2, it is expected that software in layer l_2 will wish to execute both securely as well as be reliant on shared libraries for functional integrity.

Third party software libraries can either be statically linked during compilation, or linked at run time by the operating system. For the XOM architecture, the former case allows easy inclusion of third party libraries, at the cost of an increase in size in the final application. Such an increase in size comes with the benefit of a decrease in the required number of context switches, resulting in a slight performance improvement. In the latter case, the compartmental operation of the XOM architecture prevents code not encrypted with the same key from reading a principal's registers. Lie et al. address this issue [37, pp 185-186]. It is their intention, in line with the modularity principal discussed in Section 3.4.1, that third party code critical to the security of the application, i.e. OpenSSH [6], should be statically linked during compilation. The counter example, of standard I/O routines, can execute insecurely through dynamic linking at runtime.

An active principal in the XOM architecture calls dynamically-linked code in a specific manner. It must first copy parameters intended for the called library to the null compartment. It must then call the unencrypted, dynamically-linked code to run in the null compartment. When the linked code finishes executing, the XOM application can copy the results from the null compartment. As discussed in Section 3.2.2, this results in the principal relying on possibly modified shared libraries for functional purposes.

4.4.1 NGSCB

Microsoft's NGSCB framework precludes the use of third party shared libraries. Additionally, code intended to execute as an NCA on the secure Right Hand Side (RHS) must be modified to execute under the secure kernel known as the nexus. NCAs can be full applications, or a limited security-relevant component of an application or a service. This is similar to the modularity suggested by the designers of the LOCK system (Section 3.4.1 on page 82). During the NGSCB presentation at Microsoft's 2003 PDC [23], the concept of Trusted Service Providers (TSP) was introduced. The difference between a TSP and a standard NCA is only logical. It is intended by Microsoft that as the NGSCB environment matures, TSPs will be developed that provide standard services and capabilities in secure, trusted code. They will function similarly to public shared libraries on the LHS. They will provide libraries to NCAs, typically like an RPC call, and will work through Inter-Process Communication (IPC). The attestation vector of an NCA, seen in Figure 3.3 on page 80, will be modified to include all TSPs that the NCA calls on.

Previous work by the author [20] showed that it is possible to refactor an existing application *P* into secure and insecure modules in order to solve existing vulnerabilities in *P*:

> [Our] integration illustrates that it is possible to redesign an existing application to make use of the new security primitives provided by NGSCB, without being forced to redesign completely, discarding the existing usability and strengths of an application.

This research showed that, for certain application classes, it is possible to implement security-relevant functionality separately, without being forced to redesign the entire application.

4.4.2 Trusted Platform Module

The measurement of shared libraries in the layer l_1 and l_2 systems discussed for the TCG's Trusted Platform Module differs due to the implementations of attestation built by Marchesini et al. and Sailer et al.

Our definition of attestation (definition 2.5 on page 47) requires the attestation vector to include the identities of all principals $Q_{1...n}$ able to affect the state of P. The Trusted Computing Group's Trusted Platform Module v1.2 specification does not provide curtained memory. This means that, as discussed in Section 2.3.4, any principal Q in layer l_1 is capable of affecting the state of a principal P in layer l_2 or layer l_1. Any attestation vector AV for a principal P on ρ in layer l_2 must include all principals $Q_{1...n}$ that are able to affect the state of P. This is dependent on the operating system controls and code loaded into layer l_1, and their use of the functions provided by the TPM in layer l_0.

Implementation of Sailer et al.

The implementation work carried out by Sailer et al. [53] (see Section 3.2.7 on page 66) was done on a Redhat 9.0 system. Recall Table 3.7 on page 74 that shows the integrity challenge protocol. Step 5c. involves the verification of the Stored Measurement Log (SML). Each measurement $ID(P)$ in the SML must be classified by the verifier, and a decision then made about the integrity of the attesting platform. This implementation enables the use of shared libraries, as all code used on by P is identified in the SML. Any decision about the integrity of a P that depends on shared libraries for functional importance can be made knowing the integrity of those shared libraries.

Work by Sailer et al. [52] classified applications into five classes depending on their ability to affect the integrity of the system, shown in Table 4.1. Sailer et al. classify only those programs in the *known* set, and did not include in their discussion classification of measured structured data with identifiable security semantics. Table 4.1 has been extended with these extra classifications. Structured data, typically a configuration file, known to enable possible vulnerabilities in an associated program, reclassifies that program as either remotely or locally

Class	*Explanation*
Malicious	Programs known to be malicious, such as versions of system tools found in root kits, or those designed to attack the measuring system directly.
Remote Vulnerabilities	Programs that rely on unstructured data from the network, such as e-mail or web browsers, with known vulnerabilities. Acceptable programs configured with structured data known to cause remote vulnerabilities.
Local Vulnerabilities	Programs that rely on unstructured data from local input, and are known to have vulnerabilities. Acceptable programs configured with structured data known to cause local vulnerabilities.
Uncontrolled	Programs that change the software stack, but have not been instrumented to measure and record the changes.
Acceptable	Programs with no known vulnerabilities or malicious code that do not enable the user to circumvent measurement system. Also structured (configuration) data associated with an acceptable program that maintains that program's integrity.
Unknown	Unidentified programs and structured data.

Table 4.1: Integrity classes of software, adapted from [52].

vulnerable. Additionally, structured data known to configure an acceptable program P in a manner ensures the integrity of P is classified as acceptable.

For the framework implementation described by Sailer et al. the known set of measurements included "all Redhat 9.0 programs and libraries including updates, the fingerprints of our own extensions for client policy control, acceptable kernels, and boot configurations" [52, §4.1]. This measurement process is known as an *enrolment scheme*. Sailer et al. enrolled the Redhat 9.0 platform into their classification by generating measurements $ID(P)$ for all programs and structured data. Each measurement was classified as *acceptable*, as defined in Table 4.1.

Establishing and implementing an enrolment scheme for a single, well-defined platform (Redhat 9.0) and environment is trivial. The problem quickly becomes non-trivial as the enrolment scheme is scaled to include other platforms and software, and enrolment performed by other parties. The issue of establishing a link between a code identity $ID(P)$ and a specified application P are discussed in Section 4.3 on page 106. The implementation of attestation proposed by Sailer et al. mitigates those problems through defining a compiled set of applications that are considered trusted.

We introduce the notation T_q and $T_{q'}$ to indicate a point in time at which an application Q begins executing (T_q) and finishes executing ($T_{q'}$). Consider an application Q that starts executing at time T_q and finishes executing at time $T_{q'}$. Consider also an application P that starts executing at time T_p, where $<$ indicates 'before' and $T_q < T_p < T_{q'}$. Recall the code ID measurements of applications P and Q are taken immediately before T_p and T_q respectively. An attestation vector AV prepared for application P, denoted $AV(P)$, attests to the identity of P at time T_p. However, the attestation vector itself is prepared after $T_{q'}$. No assurance that Q has not affected the state of P during the period $T_p \rightarrow T_{q'}$ can be given without P executing in assured curtained memory. The attestation vector $AV(P)$, to comply with definition 2.5 on page 47, must contain the code ID of all applications $Q_{1\ldots n}$ able to affect the state of P. This set of applications includes all that executed at any point in time after T_p, including those that finished executing before $AV(P)$ is created.

This means that any attestation vector for P from ρ is required to include the identities of all other principals $Q_{1\ldots n}$ on ρ. This obviously leads to a significant loss of privacy for any user of ρ, forcing them to include the identities (through the Stored Measurement Log) of all applications on their computer. It also greatly increases the complexity of a decision by another platform ρ' to trust the software configuration of ρ. Given the abilities of a general purpose platform, as described in Section 2.2, making a decision to trust an AV that contains code IDs of unknown applications may be impossible.

Consider a measurement verification policy of ρ' that considers the integrity of ρ to be unsatisfactory if any application not considered acceptable is found in the SML. This results in a service not being supplied by ρ', or in ρ' not using a service provided by ρ. Recall that the measurement system on ρ will invalidate the SML if it detects one of the cases in Table 3.6 on page 73, requiring a reboot before any attestation will succeed. If ρ executes any *unacceptable* or *unknown* application before attempting an attestation it will fail, and require a reboot of ρ before the attestation will succeed.

Given the prevalence of applications with known local or remote vulnerabilities, regular reboots may be required. With the increased stability and functionality of modern operating systems, this sort of user experience has long been considered unacceptable. Alternatively, a

Issues
Establishing suitable enrolment scheme to move digests from *unknown* to *known*.
Managing movement of measurements from *acceptable* to *vulnerable*.
Reliance on PKI to distribute measurement lists.
Timing attacks used to exploit different versions of the classification list.
Classification of a program as *acceptable* may not reflect its actual integrity or security properties.
Closed source programs can have measurements performed and signed by authors.
User compilation of open source program *P* results in many distinct measurements of *P*.

Table 4.2: Issues with enrolling and managing application measurements.

weaker policy could allow some quantity of vulnerable programs to be present in the SML. However, user knowledge of these programs could lead to their use and installation purely to exploit their vulnerabilities.

The implementation of a large scale enrolment scheme to generate and maintain correct classifications of programs and structured data raises a number of issues distinct to those discussed in Section 4.3. These issues are summarised in Table 4.2, and discussed briefly below.

The enrolment scheme must ensure that only acceptable programs, i.e. those without any known vulnerability, are classified as acceptable. Doing this in a distributed environment requires trust in third parties to correctly analyse a program and make a statement about its acceptability, as well as a PKI to distribute those statements securely. An application *P* that has been enrolled and classified, in some manner, as *acceptable* may become considered vulnerable at some later point in time. Both the initial enrolment and any later reclassification of an acceptable program as vulnerable must ensure that programs are truly vulnerable. An attacker should not be able to force an acceptable program to be wrongly classified as vulnerable. Any decision about the integrity of a platform will require an evaluation of the cost of false rejections or false acceptances. Preventing malicious classification of programs as vulnerable requires careful administration of the enrolment scheme, and it may not be possible to achieve a completely correct classification.

All verifications of an SML log must be performed against the most recent classification list, to prevent attackers from exploiting the time period between updates to a locally-cached list from a central repository. The classification of an application as *acceptable* merely indicates that it is *thought* to not be vulnerable to attacks that allow the subversion of the integrity of the platform it executes on. An operating system distributor, such as Redhat, can perform and distribute measurements of their product for inclusion in the classification lists. For individual software applications released as binaries, measurements may be obtained from the author. The compilation of open source software by users may result in considerable numbers of programs in an SML to be considered *unknown* (see Section 4.3 on page 106). A solution to the issues listed in Table 4.2, and discussed above, that result from a classification scheme (Table 4.1) being used to make decisions about the integrity of a remote platform, is an open problem.

There are two other important points to note about the implementation described by Sailer et al. One of these is the partially-ordered nature of the SML, and the other is the vulnerability of the system to Time Of Check to Time Of Use (TOCTOU) class attacks.

Recall that repeated executions of a program P do not result in repeated additions of $ID(P)$ to the Stored Measurement Log (SML, Section 3.2.7 on page 66). Consider an interaction between two distinct programs P and Q, that exposes a vulnerability when P is executed before Q. In notation introduced above, this can be stated as $T_p < T_q$, where T_p is the point in time where P is executed. Consider an SML log indicating the execution of Q before P ($T_q < T_p$). Given that repeated executions of Q are not recorded, this statement shows that an execution sequence $T_q < T_p < T_q$, resulting in the vulnerability exposed by $T_p < T_q$, may have occurred. The measurement system, as currently designed, is not intended to expose such execution sequences on ρ. The measurement system was designed like this to keep the size of the SML to a minimum, so that it could be easily transmitted to ρ'. While every execution could be recorded in the SML, long-lived execution of ρ may result in the size of the SML increasing beyond the storage capacity of ρ.

The attestation vector does not assure the integrity of ρ at any point in time after it has been generated, so Sailer et al. [53] intend attestation to occur twice for any transaction Θ — immediately before Θ and immediately after Θ. We propose the following attack, where a platform ρ is compromised after a valid attestation vector AV has been generated and transmitted to ρ', who verifies the integrity of ρ as suitable.

Consider a platform ρ responding to an attestation challenge, intended to assure ρ' the integrity of ρ is suitable for some transaction Θ to occur. There is some non-trivial time Δ between the generation of the attestation vector AV on ρ and the beginning of the transaction Θ on ρ. This time Δ is made up of a number of components:

1. The transmission of AV from ρ to ρ'.
2. The verification of AV on ρ'
3. The transmission of Θ from ρ' to ρ.

Of the three components of Δ, ρ' is only able to affect the first and third. The attacker on ρ wishes to delay the start of the transaction Θ until he can compromise ρ. He must therefore not act suspiciously until ρ' has verified the SML and issued Θ. There are many possible ways this could be done. As an example, ρ could compromise the integrity of the network link causing repeated re-transmission of Θ by ρ'.

In general, the attesting platform ρ can delay the start of Θ on ρ, by maliciously extending Δ, until they are able to compromise the integrity of ρ. The delayed Θ would then be processed on ρ in a state that would cause an attestation challenge to fail. The second attestation challenge from ρ', intended to occur immediately after the completion of Θ, could be ignored by ρ. The challenging party ρ' would be left uncertain as to whether ρ had been subverted by a malicious attacker, or simply suffered from hardware or network failure.

This observation implies that an attestation vector must guarantee integrity of ρ for some non-trivial period of time into the future. Otherwise, any conclusion drawn by ρ' about the integrity of ρ for a transaction Θ is reduced to simply trusting the local administrator to not delay Θ and compromise ρ as described here.

Implementation of Marchesini et al.

The attestation protocol implemented by Marchesini et al. [40] (see Section 3.2.7 on page 66) exposed limitations of both their implementation and the TCG specification. One of these was the ability of a root or superuser, known as an administrative user in our threat model (see Section 1.2), to modify the memory space of other applications [39, p.11]:

> ... a simple test on Linux shows that, without further countermeasures, the root user can manipulate the memory space of other processes with a debugger.

This vulnerability, well documented when discussing curtained memory in Section 2.3.4, leads to a possible TOCTOU attack on their system. The code IDs of the long-lived and medium-lived software are generated directly before they are loaded. The inability to secure the code base from the root user leads to the possibility of the AIK and SK keys being released to the SSL web server, after which a malicious administrative user uses a debugger to modify the executing code. Marchesini et al. also consider the possibility of hardware based attacks, possible even if the root user can be prevented from modifying the memory address space of other programs. Such an attack involve some internal bus, perhaps an IEEE1394 OHCI Firewire bus, being used to write directly via DMA to some process P's address space, after $ID(P)$ has been generated.

As discussed in Section 3.2.7 on page 66, Marchesini et al. integrated the NSA's LinuxSE mandatory-access control system into their framework to allow *compartmented attestation*. While the integration itself was successful, defining a usable security policy was non-trivial, and in one instance resulted in a system that required a complete re-installation. In addition, it is not clear that the information about the defined SELinux security policy on ρ required to be sent to ρ' does not result in equivalent privacy concerns as an attestation vector attesting to all programs on ρ. Marchesini et al. describe the usability of SELinux as less than satisfactory:

> The policy language is robust and expressive, but is also cumbersome to learn and use. It is not always clear how to state the security goals of the system and then build a policy which accomplishes those goals.

Their attempted implementation concerned only one program, with only one associated security goal. The difficulty for specifying a security policy in SELinux for a general purpose computing platform, with many programs, perhaps with conflicting security goals, would be

Requirements of attestation protocol	*Marchesini et al.*	*Sailer et al.*
1. Withstand software attacks.	·	⋆
2. Mitigate TOCTAU-class attacks.	⋆	·
Requirements of attestation vector		
3. Mitigate losses of privacy.	·	·

Table 4.3: Features of attestation implementations by Marchesini et al. and Sailer et al. A star (⋆) indicates the feature is present.

exponentially more difficult. An implementation of the framework proposed by Marchesini et al. [40] for a general purpose computing platform (Section 2.2 on page 20) would severely restrict its usability. Integrating shared libraries into the access policy could, in theory, be done securely. An attestation vector from the framework proposed by Marchesini et al. could include those software compartments that contained the shared libraries.

The design of a powerful, robust mandatory access policy language that allows developers and end users to succinctly specify their security goals is still an open problem in the field of trusted and secure computing research.

Summary

The two attestation procedures discussed here both fail to fulfil all of the requirements of an attestation procedure discussed in Section 2.3.5, Table 2.5. Table 4.2 summarises those elements of Table 2.5 that either implementation fails to provide. Marchesini et al. fail to secure ρ from an administrative user. Their integration of SELinux results in a difficult and cumbersome to manage system that is non-trivial to reconcile with the requirements of a general purpose computing platform, discussed in Section 2.2. However, it does allow an attestation vector to give some guarantee of the integrity of ρ into the future. It is not clear that attesting the presence of a SELinux-enabled kernel, along with only a specific software compartment's security policy, properly assures ρ' of the integrity of that compartment. The entire security policy of ρ may need to be attested, resulting in privacy concerns.

Sailer et al. do succeed in protecting from the administrative user, by non-intrusively causing future attestations to fail if a possibly malicious situation occurs. However, they do nothing to reduce the privacy implications of the SML. Protection from TOCTOU attacks is limited, only succeeding in assuring a challenging platform ρ' that a platform ρ could be trusted throughout a transaction when it is not compromised. A compromised system can fail to report its status, leaving ρ' unsure of the final status of the transaction. In addition, with heterogeneous software distributions on ρ in layers l_1 and l_2, enrolling and maintaining the list of acceptable and unacceptable programs is non-trivial. It also appears to preclude a user compiling their own software, as such software appear as *unknown*. Any reasonable security policy on ρ' would require an unknown entry in the SML to cause ρ' to distrust the integrity of ρ.

4.5 Resistance to Software Attacks

The threat model for trusted computing, discussed in Section 1.2, requires a framework to be resistant to all forms of software attack. This section analyses both the Trusted Computing Group's TPM specification and Microsoft's NGSCB from this perspective.

4.5.1 Trusted Platform Module

Recall that the Trusted Software Stack (TSS) in layer l_1 (see Section 3.2.1 on page 52) enforces synchronised access to the Trusted Platform Module in layer l_0. The TSS works in conjunction with a TPM Device Driver [67, p.16]:

> The TPM Device driver is typically provide by the TPM manufacture and incorporates code that has understanding of the specific behavior of the TPM. This code is expected to be loaded and function in Kernel Mode... The TSS exclusively opens the TPM device driver; the driver does not allow any applications to have an additional connection to the TPM device besides the TSS.

Given the location of the TSS outside the Trusted Computing Base, it is vulnerable to software attacks, and cannot be assured to operate correctly. It executes as a kernel-mode driver, operating in layer l_1. It is vulnerable to attacks by other code in layer l_1. A simple attack involves another principal Q in layer l_1 issuing commands to the TPM, thereby invalidating an existing session through which P is issuing commands to the TPM. A principal P, executing in layer l_1 or above is unable to be guaranteed exclusive access to the TPM in layer l_0. Our analysis concludes that this construction, coupled with the TPM's lack of curtained memory, allows a possible *insertion attack* to occur, shown in Figure 4.2, as well as a denial of service vulnerability.

As discussed in Section 3.2.2 on page 57, a measurement (see definition of $M[x](R)$ on 53) of a code ID $ID(R)$ is taken of an application R before it begins executing in layer l_2. If R is of a length greater than 64 bytes, the measurement requires at least 3 commands to be issued by the measuring principal, in this case P. The computation of a SHA-1 hash of R of length $R.length > 64$ is an n-tuple set of commands $(1,2,3,...,n)$, where $n > 2$ and approximates $R.length$ modulo the maximum data size specified by `TPM_SHA1Start`. The first command, $i = 1$, is `TPM_SHA1Start`. The last command, $i = n$, is `TPM_SHA1Complete` or `TPM_SHA1CompleteExtend`. The middle commands, $1 < i < n$, are `TPM_SHA1Update`.

Our attack, proposed below, assumes that a principal Q in layer l_1 is able to monitor the commands P in layer l_1 issues to the TPM. It can do this in at least two ways. The first is by preventing P from using a session to ensure the confidentiality and integrity of its commands, by consuming all available sessions with the TPM itself. It is then able to issue commands, as discussed below, without invalidating P's interaction with the TPM.

The alternative is through monitoring P's execution. The Trusted Computing Group's v1.2 specification discussed here does not include provisions for assured curtained memory. This means that a principal at layer l_1 is able to view and modify the memory space of another principal in layer l_2, and in layer l_1. When the caller P sets up the session, a shared secret is established between P and the TPM. This shared secret is used to provide confidentiality and integrity. A rolling nonce is used to prevent replay and man-in-the-middle attacks. With no assurance of memory confidentiality or integrity, the secret values shared between P in layer l_2 or l_1 and the TPM are able to be viewed and modified by any principal Q in layer l_1. The following attack assumes either of the two methods for inserting TPM commands described here are possible.

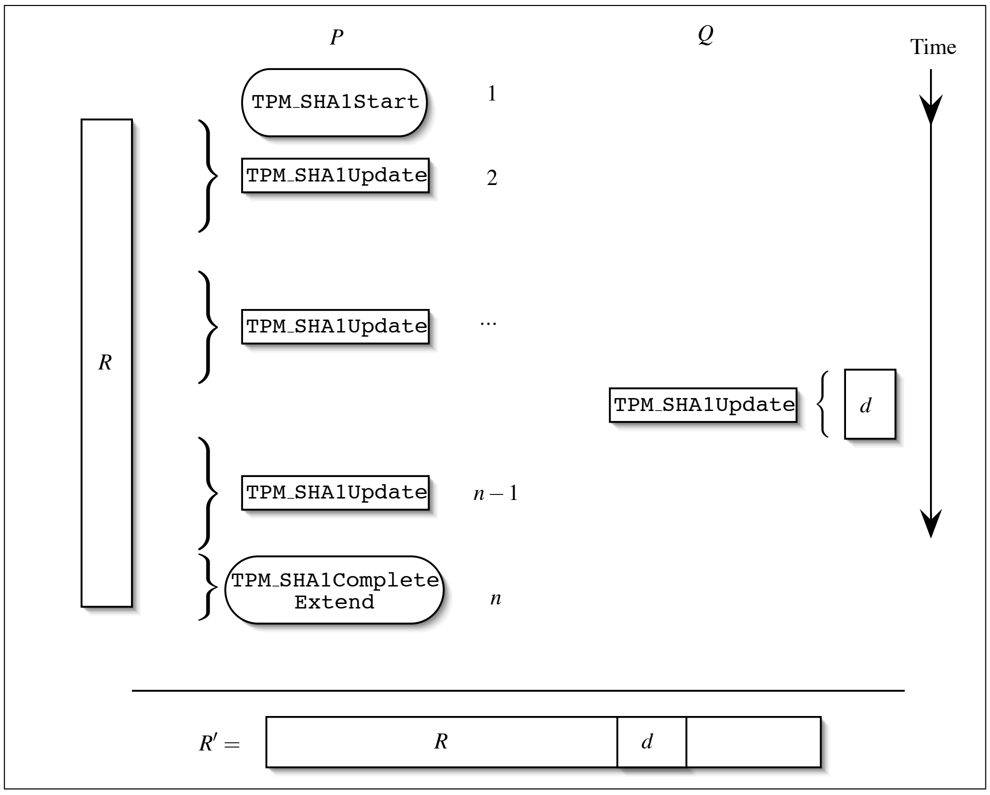


Figure 4.2: An insertion attack being used by Q to force P to generate a measurement of an application R before R begins executing. The program R' is shown with d inserted into its measured image.

Figure 4.2 shows this attack. Consider the issuing of an n-tuple command sequence intended to measure an application R, before R begins executing in layer l_2. The operating system P on the left calculates a measurement of an application R, through issuing multiple `TPM_SHA1Update` commands, each one passing the TPM a range of bytes from R. The operating system P cannot be assured that another principal Q in layer l_1 will not maliciously issue a `TPM_SHA1Update` sometime between P's issuing of the first command and the last.

An attacker Q, on the right, issues an `TPM_SHA1Update` command, specifying some arbitrary data d. The value of d is not important, as it is only used to cause the TPM to generate an incorrect measurement $ID(R')$, such that $ID(R) \neq ID(R')$. The result, when P finishes issuing all its `TPM_SHA1Update` commands and finally calls `TPM_SHA1CompleteExtend`, is an incorrect calculation of code identity (measurement) of the application R. The operating system P then allows R to start executing, with an incorrect $ID(R')$ stored in the Stored Measurement Log and PCR register. There is no way for P to test the validity of the returned measurement, except for re-computing it either using the TPM, or through an implementation in software. If Q is able to control the ordering of the malicious `TPM_SHA1Update` call, relative to *P*'s `TPM_SHA1Update` calls, the same incorrect measurement will be obtained for all of *P*'s repeated computations of $ID(R)$.

The specific attack described above can be generalised. We refer to it as an *insertion attack*. An attacker Q in layer l_1 is able to insert arbitrary TPM commands into a TPM command sequence issued by application P, in layer l_1 or l_2. We refer to TPM commands that require the issuing of an n-tuple command sequence to use as stateful commands. An attacker Q is able to modify the output of stateful commands by inserting TPM commands, as appropriate, into the n-tuple sequence issued by P.

A general solution to this vulnerability involves virtualising the TPM at layer l_0 inside the TCB, so that principals P and Q are assured of interacting with logically distinct TPMs. A limited solution is modifying the function signatures of the `TPM_SHA1Update`, `TPM_SHA1Complete`, and `TPM_SHA1CompleteExtend` commands. Their function return signatures could either include the number of bytes processed so far, or the number of times `TPM_SHA1Update` has been called since `TPM_SHA1Start`. Either would allow a principal P to discover if another principal Q had issued malicious `TPM_SHA1Update` calls as described above. This solution would prevent an attack based on the second of the two possible methods for maliciously inserting TPM commands described above. These solutions are discussed further in Chapter 5.

The complaint of Marchesini et al. about the complexity of SELinux (Section 4.4.2 on page 111) seems equally applicable to the TCG's TPM specification, and the difficulties in assuring correct usage of the security primitives it provides. Specifically, the complexity of the AIK creation procedure through a Trusted Third Party (Section 3.2.4 on page 61), caused concern. When coupled with the additional requirements of creating a storage bound to the identity specified in the attestation identity key Marchesini et al. stated [39, p.14]:

> The complexity of this process troubles us. In security, one should be careful about trusting something that is too big to fit into one's head. It takes a long time to find the right combination of commands and properties from the specification. What *other* combinations are present? Are there any combinations that enable function-

ality that the designers did not intend?

Their implementation experience was done with v1.1b of the TPM specification. The specification discussed in this thesis is version 1.2. It should be noted that the latest version of the TPM specification discussed here does include support for Direct Anonymous Attestation [21]. With this, a Trusted Platform Module is able to anonymously attest to a challenger, without requiring the indirection of a trusted third party.

However, no other complexity has been removed between versions. This issue does not affect the functionality or features provided by the TPM, but it does impact the ability of application developers to interact with the device, correctly leveraging the security primitives it provides. Without a trusted, standardised software kernel to interface with, developers writing code to run in layer l_2 may face considerable difficulties in validating the security properties of their code's interaction with the TPM. The TCG specification requires lengthy sequences of commands to be issued by an application P to implement some security function. This requirement, coupled with the insertion attack described above, leads us to conclude that the SHA-1 sequence of commands may not be the only sequence that allows an attacker to successfully subvert an application P.

Standardising an interface for the TPM, inside the Trusted Computing Base, that allows applications to atomically execute commands yet also retain the power and flexibility of a comprehensive API, is an open problem in trusted computing research.

4.5.2 NGSCB

Recall that Microsoft's Next-Generation Secure Computing Base (Section 3.3.1 on page 76) implements weak curtained memory, as defined in definition 2.4 on page 41. This partitions system memory into two, with secure Right Hand Side (RHS) software running with increased privileges in order to address the curtained memory.

NGSCB uses a secure kernel, called the Nexus, to manage applications that execute on the RHS. Figure 4.3 shows a Nexus Computing Agent (NCA) in layer l_2 requesting the *unseal* operation be performed on some data by the Nexus in layer l_1 on the TPM in layer l_0. An NCA

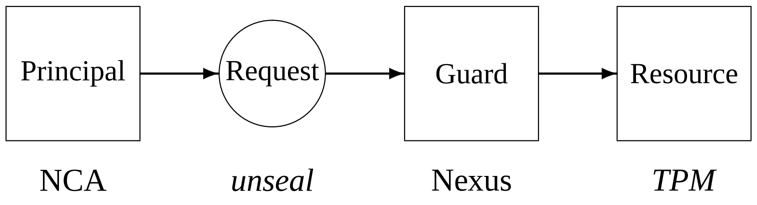


Figure 4.3: Access control model showing Nexus Computing Agent (NCA) attempt to unseal a file through the Nexus secure kernel.

relies on the Nexus to service all its access requests. The assurance that an NCA executing on the RHS executes correctly relies on the nexus itself to be correct. Also, an NCA relies on the nexus to correctly implement the trusted computing primitives sealed storage and curtained memory. The Nexus enforces memory separation, in software only, between differing NCAs on the RHS.

An NCA is unable to verify the identity of the Nexus under which it is running, and so must trust it axiomatically. Microsoft will include with the NGSCB system an official Nexus, distributed as a compiled binary. The source code of Microsoft's nexus kernel will be kept closed. It may eventually be made available to certified individuals and corporations for code review — an attempt by Microsoft to engender trust in what is otherwise a closed system [48]. This process of only allowing certain individuals and groups to view the source code does not meet the definition of *open* given in Section 2.1 on page 7.

As discussed in Section 2.3.1 on page 25, in general any principal P in layer l_2 relies on the operating system or kernel Q in layer l_1 to be non-malicious to ensure the correct operation of P. This equally applies to an NCA in layer l_2 and the Nexus in layer l_1.

The software and hardware stack that is included in the attestation vector is shown in Figure 3.3 on page 80. The code IDs of the Nexus and the NCA, as well as credentials proving the validity of NGSCB hardware platform itself are used to attest to a remote challenger. An important consideration is that there is no similar concept for 'local' attestation. Specifically, although the nexus is able to generate a code ID for any NCAs it is hosting, through use of the `Quote` function, an NCA is not able to securely identify the nexus within which it is executing.

This implies that an NCA P *must* perform an attestation to be assured of the validity of the platform ρ and Nexus Q it is operating on and under. After an attestation procedure, in which a remote party trusted by P verifies the identities of ρ and Q to be suitable for P, P can be supplied with some data d expected to be kept confidential. It is able to seal d, as described in Section 3.3.2 on page 79, under the $ID(P)$, $ID(Q)$ and ρ.

This holds for any principal P in layer l_2 that relies on the integrity of Q in layer l_1 for correct operation. Before P generates or obtains data intended to be kept secure and confidential from other principals in layer l_2 or l_1, as well as the local user or administrator, it must first perform an attestation with some trusted remote party to verify the integrity of ρ and Q. Assuring P of the correctness of ρ and Q, without relying on a trusted third party, is currently an open problem in trusted computing research.

There is one safeguard known generally to the wise, which is an advantage and security to all, but especially to democracies as against despots. What is it?
Distrust.

Demosthenes

5 Architectural Improvements

5.1 Introduction

This chapter proposes additions and modifications to the Trusted Computing Group's Trusted Platform Module v1.2 specification, through modifying and integrating the XOM CPU architecture.

Section 5.2 discusses the proposed modifications and additions. Section 5.3 summarises some motivations for using the TCG specification as a base, as well as the issues and threats our proposed design attempts to mitigate.

5.2 Modifications

The modification of the XOM architecture discussed here is referred to as TPM/XOM. Briefly, the compartmented execution model of XOM is modified and added to the TCG specification, enabling selected principals on τ to execute in curtained memory. The distribution and packaging model of XOM, discussed in Section 3.7.2 on page 93, is removed. Intended to prevent the unauthorised execution of applications, it is not required to obtain curtained execution. The TPM is modified to include compartment-specific state, allowing each compartment to interact with a logically separate TPM.

Section 5.2.1 discusses modifications to the application loading mechanism of the XOM and TPM architectures. Section 5.2.2 discusses compartment-specific state inside the TPM. Section 5.2.3 discusses hardware modifications. Section 5.2.4 discusses modifications to the cryptographic algorithms proposed by Lie et al. [37] to improve performance. Section 5.2.5 discusses the use of shared libraries.

5.2.1 XOM Compartments

The XOM architecture ensures the integrity of P from its packaging by a distributor D through encrypting P. One of the characteristics of the attestation method of the trusted computing frameworks built on top of the TCG's TPM (Sections 3.2 on page 51 and 3.3 on page 76) is the importance of the integrity of P only *after* $ID(P)$ has been measured and recorded. The decision

Step	*Action*
1.	χ: `LoadX`(P)
2a.	χ: generate $ID(P)$
2b.	χ: perform $E(K_s, P)$
3a.	χ: perform $D(K_s, P)$
3b.	χ: execute P

Table 5.1: Modified load and measure procedure of TPM/XOM.

of whether to trust the program P identified by $ID(P)$ is left up to the challenging party in the attestation protocol.

The TPM/XOM architecture leverages the curtained execution provided by the XOM CPU to ensure that P, identified by $ID(P)$ remains unmodified throughout its execution. Table 3.11 gives the outline of the steps involved with loading an application P that will execute in a compartment. The abstract command `LoadX`(P) instructs the TPM/XOM CPU to load P. The `LoadX` command is distinct from a normal `Load` command used to begin the execution of a non-compartmented program. The command `LoadX` signifies that the program to be loaded is to execute in its own compartment. It is loaded into memory, and its code identity $ID(P)$ is generated and stored in a XOM compartment table.

The initial generation of the code ID $ID(P)$ of P is vulnerable to the types of attacks described in Section 4.5.1, Figure 4.2 on page 121. Under the threat model established in Section 1.2, all classes of software-only attacks are possible. Our TPM/XOM model therefore *does not* rely on a trusted software stack, executing outside the TCB, to correctly load and measure P. The TPM's Trusted Software Stack (TSS) specification, in comparison, is required to ensure exclusive, controlled access to the TPM itself.

The TPM/XOM architecture does not prevent these insertion attacks, but renders them ineffectual. The load and measure of P, done by the untrusted operating system in layer l_1, results in two distinct cryptographic operations occurring in the TPM for each `TPM_SHA1Update` call. The first is the generation of the SHA1 digest of P, as in the unmodified TPM. The second is the encryption of P, and the storage of the result $E(k,P)$ into main memory for later execution. Initialisation of the required state in the XOM CPU also occurs at this point. Encryption and protection against replay, spoofing, and splicing attacks are discussed in more detail in Section 5.2.4.

We introduce the notation $P \rightarrow P'$ to denote the modification ($\rightarrow$) of P to P'. Our attack, outlined in Section 4.5.1 on page 119, performs $P \rightarrow P'$ during the measurement of P. This is denoted $ID(P \rightarrow P')$, which results in a measurement equal to $ID(P')$.

Recall that P is loaded from untrusted storage. The integrity and confidentiality of P before it is loaded is not assured in any way. An attack that performs $ID(P \rightarrow P')$ for the generation of $ID(P)$ (step 2a, Table 5.1) is equivalent to an attack $P \rightarrow P'$ before `LoadX` is called. The TPM/XOM architecture duplicates any malicious modification $P \rightarrow P'$ during the generation of $ID(P)$, to the executing image of P. The use of $ID(P)$ in the access control decision of sealed storage, and its inclusion in the attestation vector generated during attestation, remains the same in TPM/XOM.

Recall that the XOM CPU architecture [37] performs the decryption of code retrieved from main memory at the L2/main memory boundary. As code is brought in for execution, it is decrypted from main memory into an L2 cache line. For this reason, the data block size used by TPM/XOM to encrypt P is the same as the L2 cache line. The payload of each `TPM_SHA1Update` command must be encrypted in blocks of this size. The update command itself is specified to take multiples of 64 byte blocks; the L2 cache line in the original XOM architecture is 128 bytes long. The TPM/XOM `SHA1Update` command must be modified to take multiples of the L2 cache line size, specific to the platform it is on.

A code block of P, denoted $P_{[i]}$, is encrypted by the TPM. It must then be stored in main memory for later retrieval, decryption, and execution by the XOM CPU. This storage can occur in two ways. The code block $P_{[i]}$ can be:

- Returned from the TPM to the XOM CPU, where the untrusted operating system in layer l_1 copies it to the appropriate location in memory; or
- copied directly into main memory by the TPM through Direct-Memory Access (DMA).

Copying from the TPM to the XOM CPU results in considerable wasted processing; data must be decrypted by the XOM CPU, stored in the cache, then ejected and encrypted again

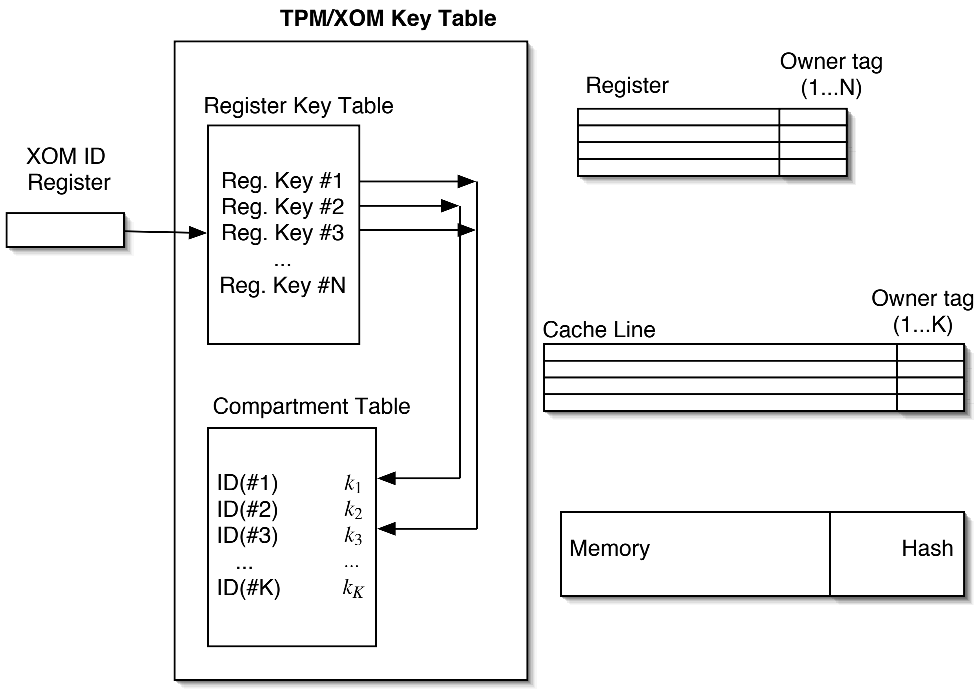


Figure 5.1: Contents and design of TPM/XOM key tables, maintained inside the XOM CPU, adapted from [37].

before storage in main memory. Copying directly from the TPM to main memory via Direct-Memory Access (DMA) is also possible, and decreases the time taken to load an application considerably. As in the original XOM architecture (Section 3.7 on page 92), despite being encrypted, the untrusted operating system in layer l_1 still manages virtual memory for P in layer l_2.

A malicious operating system can give the TPM an incorrect memory address range to copy encrypted blocks of P to. This does not result in the loss of integrity or confidentiality of P, but does allow a loss of availability — meaning such behaviour by the operating system results in a Denial of Service (DoS) attack.

The original XOM architecture [37] described two key tables stored inside the CPU. The first of these was the *compartment key table*. It is used to hold the symmetric key used to:

- decrypt the application P, and
- encrypt and decrypt P's reads and writes to memory.

The second key table was the *register key table*, used to encrypt the register contents of an executing principal when it is interrupted by the operating system. The TPM/XOM table architecture can be seen in Figure 5.1. This key table design is adapted from work by Lie et al. [37]. The XOM ID Register indexes into the register key table, and specifies the register key of the currently executing principal. The register key table in turn indexes into the compartment table, specifying the compartment and $ID(P)$ of the currently executing principal. Each register key in the register key table points to one compartment in the compartment table.

The manner in which TPM/XOM enforces the separation of principals executing in separate compartments is the same as the original XOM architecture. This procedure is described in Section 3.7 on page 92 and will not be repeated here.

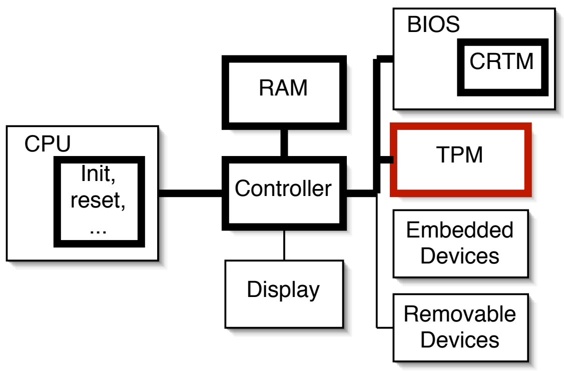
5.2.2 TPM Virtualisation

Recall our attack discussed in Section 4.5.1, Figure 4.2 on page 121. In general, in the TCG's TPM specification, a principal P cannot be assured that a set of TPM commands $(C_1, C_2, ..., C_n)$ will not be interleaved with a command, or set of commands, from a malicious principal Q. Such interleaving of commands renders the result of P's commands incorrect.

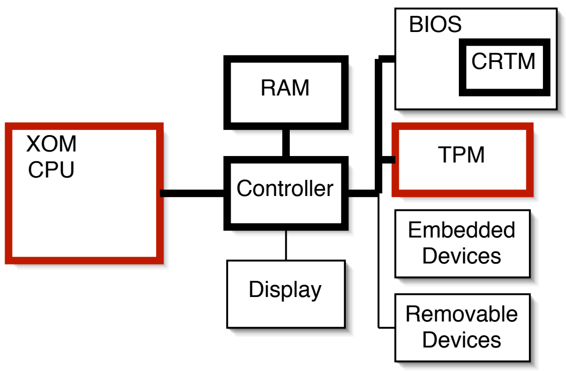
Our proposed TPM/XOM architecture prevents attacks of this type. Commands issued by a principal P executing in a compartment i on χ are handled by a virtualised TPM, through the use of *compartment-specific state* inside the TPM. The notation ${}^{i}P$ is used to indicate a principal P executing in compartment i, where $i > 0$. The notation ${}^{\emptyset}P$ indicates that P is executing in the null compartment.

Each command that a principal ${}^{i}P$ issues to the TPM acts on state inside the TPM exclusive to the compartment i. It should be noted that some TPM commands (see Table 3.2 on page 56) are stateless, i.e. they do not affect state in the TPM. There are a number of stateful commands, however. For example, the commands used to generate a SHA-1 digest of data involve an initialisation command and a finalisation command, both of which affect state in the TPM.

When a principal ${}^{i}P$ issues a TPM command, the TPM queries the XOM CPU for the compartment ID of the active principal. The XOM CPU responds with the compartment number i, where $i > 0$, or $i = \emptyset$ to indicate P is in the null compartment. The TPM device maintains the required state for each compartment i separately. This assures a principal ${}^{i}P$ that no other



(a) TBB components and extent of the TCB in the TCG's TPM architecture.



(b) TBB components and extend of the TCB in the TPM/XOM architecture.

Figure 5.2: Thick black borders indicate devices expected to not act maliciously, known as Trusted Building Blocks (TBB), adapted from [71]. Thick red borders indicate devices inside the Trusted Computing Base (TCB), containing shielded storage and protected functionalities.

principal ${}^{j}Q$, where $i \neq j$, is capable of inserting malicious commands into a stateful command sequence of ${}^{i}P$. This attack was described in Figure 4.2 on page 121. The query and response of the active compartment ID is carried out entirely in layer l_0.

The loading procedure of P, shown in Table 5.1, resets the compartment state i in the TPM. This allows the re-use of compartment IDs, and ensures that no information is left behind by ${}^{i}P$ when it finishes, for ${}^{i}Q$ to obtain.

Principals that execute in the null ($\emptyset$) compartment all share the same state. The TPM does not make any attempt to prevent the interleaving of stateful commands from principals in the null compartment. Additionally, no principal Q in layer l_1 is required to enforce synchronised access to the TPM, as is the case with the TCG's TPM specification.

5.2.3 Hardware Modifications

The original XOM design made significant changes to the internal architecture of the CPU. The CPU was required to be able to perform asymmetric decryption of the program header with a unique key pair, as well as perform symmetric encryption and decryption with that key. XOM key tables were maintained inside the CPU. Cryptographic hash operations were also required to ensure the integrity of data stored to main memory. In the XOM architecture, only the CPU was considered to be part of the Trusted Computing Base (TCB).

The Trusted Platform Module was the only entity considered to be within the TCB in the TCG specification. It supports advanced cryptographic functionality and secure internal storage as describe in Section 3.2. The TCB of the TPM/XOM architecture is seen in Figure 5.2, outlined in red. The Trusted Building Blocks (TBB) of the TPM and TPM/XOM architectures are shown outlined in black. A TBB is a hardware component not considered to be inside the TCB itself, but is relied upon to perform correctly in order for the TPM to operate as expected. It is these devices that are not designed or expected to with stand hardware attacks as described in Section 1.2.

The TPM/XOM architecture keeps the Trusted Platform Module v1.2, and includes the XOM CPU inside the TCB, as seen in Figure 5.2(b). The TPM/XOM CPU is still required to perform asymmetric cryptography, and its symmetric encryption routines are simplified as described in Section 5.2.4. It is not required to contain a unique asymmetric key pair. The XOM CPU and the TPM communicate over a shared bus, which is not protected from hardware snooping attacks.

5.2.4 Cryptographic Algorithms

Initial discussion of the XOM architecture by Lie et al. [37, 64] encrypted P with a symmetric block cipher such as Triple DES, with the key generated by the distributor D. The execution of a principal P in our TPM/XOM system begins with P being loaded, measured, and encrypted by the TPM. Our encryption routine must ensure both the confidentiality and integrity of P when it is stored in main memory. Work by Suh, Clarke, Gassend, van Dijk and Devadas [63] propose modifications to the XOM architecture to use One-Time Pad (OTP) encryption to speed up memory read and write accesses. Yang, Zhang and Gao [73] propose the use of OTP encryption to encrypt the instruction stream as well.

Suh et al. [63] propose the use of the AES decryption function (AES^{-1}) to generate a suitable OTP key, where as Yang et al. [73] implement their OTP encryption scheme with the DES encryption function. Adapted from Yang et al. the encryption and decryption of data to and from memory appears as equations (5.1) and (5.2) respectively. Here, p denotes plain text, c denotes cipher text, s denotes the seed, k denotes a random key, and $\oplus$ denotes the exclusive-or (XOR) operation.

$$p \oplus \mathrm{AES}_k^{-1}(s) \longrightarrow c \tag{5.1}$$

$$c \oplus \mathrm{AES}_k^{-1}(s) \longrightarrow p \tag{5.2}$$

A cryptanalysis of this scheme is outside the scope of this thesis. Suh et al. claim schemes of this general form to be secure [63, p.6]:

> The conventional one-time-pad scheme is proven to be secure [17]. Our scheme is an instantiation of a counter-mode encrypt scheme, and can easily be proven to be secure, given a good encryption algorithm that is non-malleable [38].

Our one-time pad encryption scheme is considered to be secure as long as the seed s is never repeated with the same key k.

We propose two possible OTP seed generation functions appropriate for the TPM/XOM architecture. One is for the once-only encryption of P itself, and another generates unique OTPs to allow repeated encryption of data to the same virtual address.

The Random Number Generator (RNG) in the TPM is used to obtain a 128 bit key k, equivalent to the symmetric key generated by the distributor D. However, a different key is generated and used for each execution of P.

We require a 128 bit seed s. Suh et al. [63] state that for performance reasons the XOM CPU should be able to generate s, so as to compute $\mathrm{AES}_k^{-1}(s)$, before c is retrieved from main memory. This allows a 1-cycle XOR operation to obtain p from c, when c arrives. Equation (5.3) shows the seed generated for the once-only encryption of P. A is the 64 bit virtual address of the 128 bit block being encrypted. It is concatenated with the low-order 64 bits of $ID(P)$, denoted $ID(P)_{[0,\ldots,63]}$.

$$\begin{aligned} A \,\|\, ID(P)_{[0,\ldots,63]} &= s \qquad (5.3) \\ |s| &= 128 \end{aligned}$$

Equation (5.4) shows the seed generated for the encryption of data generated by P, adapted from the one proposed by Suh et al. [63]. The random vector V is generated by the TPM during the initial encryption of P. The time stamp TS ensures that each cache line stored in main memory is encrypted with a unique OTP. The time stamp TS is a monotonic counter, increased each time a cache line is evicted from memory. Suh et al. require the re-encryption of all blocks in memory with a newly generated key k when their time stamp reaches its maximum value. They use a time stamp of 32 bits. A 64 bit time stamp allows 2^{64} cache evictions before a new key would need to be generated. However, if TS is initialised equal to zero, an attacker may guess its value through monitoring cache eviction counts. If there is no random vector V used, the seed depends on a non-random TS and an easily obtained address A. It should be noted that the time stamp TS used to encrypt a cache line must be stored in main memory for later decryption of that cache line.

$$\begin{aligned} A \parallel V \parallel TS &= s \qquad (5.4) \\ |s| &= 128 \end{aligned}$$

Recall the initial encryption of P is performed inside the TPM, and that all decryption of P occurs inside the XOM CPU. The TPM must send the key k and the $ID(P)$ to the XOM CPU for storage in the compartment table. It should be noted that the transmission of k from the TPM to the XOM CPU occurs unencrypted along a bus vulnerable to hardware snooping. The general trusted computing threat model does not include hardware attacks. However the generation of a new key k for every principal that executes in a compartment means a compromised key is only valid for one execution of P. For simplicity in the design of the XOM CPU, the random vector V is also generated inside the TPM and sent to the CPU. However, this is not required and could be generated inside the XOM CPU, preventing a hardware attack from revealing the vector V.

The above OTP encryption scheme ensures the confidentiality of P, and the data it generates, but not the integrity. Lie et al. propose the use of a reversible hash to lower the computational cost associated with each store and load to and from main memory [37, p.175]:

> We can exploit the fact that a MAC provides much more functionality than we re-

> quire. A MAC is able to provide authentication for messages that are not encrypted, by using a hash that is difficult to reverse. Since the cache lines are encrypted, we are free to use a reversible hash for redundancy.

A reversible hash, such as a CRC, is generated from the L2 cache line before it is encrypted. The generated hash is stored in a separate page to the encrypted data.

Suh et al. [63] proposed a novel *Log Hash* algorithm. A discussion of its implementation is outside the scope of this thesis. However, it ensures integrity of data generated by P with an increase in the required storage space of only 6.25%.

Our proposed cryptographic algorithms, briefly discussed here, provide protection against replay, splicing, and spoof attacks for data generated by P and stored in memory. Replay attacks on data generated by P during a single execution are prevented through the use of the monotonic time stamp TS in the seed. A replay attack with data generated from a previous execution of P, where TS could be expected to repeat in value, is prevented through the use of a different key for each execution. Splicing attacks, where encrypted code or data are copied from elsewhere in memory to form new contiguous blocks, are prevented through the use of the address in the seed. And spoofing attacks, where data is generated by an attacker and copied over memory of P, are prevented through the use of the key k, and the random vector V, being kept from software-class attacks.

5.2.5 Shared Libraries

Recall in the original XOM architecture, shared libraries had to be statically compiled into the application by the distributor. Layer l_1 and l_2 implementations on top of the TCG's TPM specification took two general approaches. They either required the entire software stack of the platform ρ to be included in the attestation vector (Sailer et al. [53]), or they implemented software compartmentalisation to provide curtained memory (Marchesini et al. [40]). The former design allows the use of dynamically linked libraries, as each library used was present in the Stored Measurement Log. The latter design did not properly consider the use of shared libraries. Microsoft's NGSCB platform precluded the use of shared libraries, requiring code that wished to run on the secure RHS to be rewritten.

Our proposed TPM/XOM architecture allows principals that execute in the null compartment to use shared libraries as in a normal operating system. Programs that execute in a compartment communicate, as in the original XOM architecture (Section 3.7 on page 92), with the operating system and other applications through the null compartment. The original XOM architecture requires the use of a *caller-save* calling convention [37, p.185]:

> ..recall that in a callee-save calling convention, the dynamic library subroutines are expected to push the caller's registers on the stack. However, since the subroutine is not in the same compartment as the XOM code calling it, it will not have the ability to access those values. Thus, the caller, rather than the callee, must save all secure registers. In addition, before calling the subroutine, the calling XOM code must first move, as necessary, [required] register values... to the null compartment so that the callee can access them.

A principal P can call shared libraries that it does not functionally depend on (see Section 2.3.2 on page 28), such as certain I/O routines, by making an insecure call through the null compartment, with a caller-save convention. This functionality is described in Section 3.7.2 on page 93. A principal P can also make use of shared libraries that it does functionally depend on, and still have the platform ρ generate an attestation vector that meaningfully identifies the functionality of P (see Section 2.3.2 on page 28).

Shared libraries, specified by P, are dynamically linked while the operating system is loading P. The TPM/XOM architecture enables applications to specify two classes of shared libraries: *unidentified* and *identified*. Unidentified shared libraries are linked as in a standard operating system.

Consider a principal P that wishes to call an identified shared library L. During the loading of P, the operating system performs a `LoadX(`L`)`, as described in Section 5.2.1, for each shared library specified by P as identified. This results in L being loaded, measured, and encrypted with the same key k and seed s as P, i.e. the seed used to generate the OTP for L does not come in part from $ID(L)$. The code ID $ID(L)$ is stored, along with $ID(P)$, in the XOM compartment table in the XOM CPU.

In the TPM/XOM architecture, a principal P is protected during execution through the use of compartments, as described above. For this reason, an attestation vector AV need only include the identity of P itself, and the identities of any identified libraries $L_1, L_2, ..., L_n$, that P considers itself to be functionally dependent on.

Consider a principal P executing in compartment i, denoted ${}^{i}P$, on a platform ρ, where $i \neq \emptyset$. The attestation protocol has the same structure as described in Table 3.7 on page 74. It is the generation and contents of the attestation vector AV that differ in our proposed TPM/XOM architecture.

For ${}^{i}P$, the attestation vector includes a statement by the TPM indicating that P is executing in a compartment $i \neq \emptyset$. This allows a challenging party ρ' to be assured of the curtained memory protection implemented on ρ. The code ID $ID(P)$, generated during the loading of P, is retrieved from the compartment table in the XOM CPU and included in AV. The code IDs of any identified libraries $ID(L_1), ID(L_2), ..., ID(L_n)$, are also retrieved from the XOM CPU. Additionally, because P executes in a compartment and interacts with a virtualised TPM, ρ' can be assured that, if properly instrumented, P's measurement of any structured data with integrity-semantics (Section 4.3 on page 106) has not been subverted or modified by an attacker.

Access control decisions made by the TPM, such as when performing a seal or unseal function for P, that were previously dependent on $ID(P)$, now depend on a concatenated code ID, denoted $ID(P) \,||\, \sum ID(L)$, shown in equation (5.5).

$$ID(P) \,||\, \sum ID(L) \quad = \quad \{\, ID(P) \,||ID(L_1) \,||ID(L_2)||\, ... \,||ID(L_n)\} \tag{5.5}$$

The TPM/XOM architecture described above allows a principal P to rely on shared libraries L, and have them included in a statement made about its functionality. However, those shared libraries may still be upgraded or modified by another principal Q on ρ or by the user. Such modification $\sum ID(L \rightarrow L')$ may not be malicious, but will result in any data sealed under $ID(P) \,||\, \sum ID(L)$ no longer being unsealable unless separate copies of L are retained on ρ. It also means that, during execution of P, any shared libraries used by P are stored in memory

Issue
Privacy concerns.
Non-intrusive architecture easier to implement, but requires invalidation of the SML log.
An intrusive, or prohibitive, system is non-trivial to implement.
Difficulties with establishing a suitable enrolment scheme for application identities.
Making trust decisions based on SML with unknown or known-vulnerable applications non-trivial.
Invalidation of SML prohibits long-lived execution scenarios.
Attestation vector has no meaning for any duration of time after its generation.
Shared libraries are easy to implement.

Table 5.2: Issues arising from attestation of entire platform.

for the exclusive use of P. Multiple principals, that both specify a shared library L as being identified, result in two copies of L being stored in memory.

5.3 Motivation and Benefits

We propose additions and changes to the TCG specification specifically for a number of reasons. It is the most complete Trusted Computing Framework specification available, and is currently shipping in a number of products. Our analysis of implementations based upon it revealed a number of open problems and vulnerabilities that an integration with the XOM architecture could possibly solve.

The lack of curtained memory in the TCG specification means that either an attestation vector is required to include the identities of all principals on the system, or a secure kernel in layer l_1 is required it implement software compartments, through mandatory access control.

A number of issues that arise from attesting the state of the entire platform ρ are summarised in Table 5.2. The non-intrusive architecture proposed by Sailer et al. [53] does not intend to prevent an administrative user from carrying out actions that result in compromises to the integrity of the platform. Instead, it merely invalidates the Stored Measurement Log (SML), preventing any future attestation from succeeding. This allows peculiar usage by developers

Issues
Mandatory access systems implemented with strongly typed systems are difficult to implement correctly.
Restricts general usability of the platform.
Entire security policy may need to included in attestation vector.
Shared libraries require complex security policies to work correctly.
Attestation vector makes some statement about application state for a non-trivial period of time into the future.

Table 5.3: Issues arising from curtained memory being implemented in software.

and hackers to occur, not restricting or limiting the usability or functionality of the system for them. Indeed, designing and implementing an intrusive system that prohibits a user from taking actions that compromise the integrity of the system is an open problem in operating system security research.

Issues that arise from implementing curtained memory in software, in a security kernel in layer l_1, are summarised in Table 5.3. The LinuxSE system was noted by Marchesini et al. [40] as being cumbersome to work with, and a security policy that properly protects an application from interference may require a restriction in usability for the rest of the system. Additionally, the entire security policy may need to be included in the attestation vector, removing any privacy gains from compartmented attestation. A properly implemented mandatory access control system can prevent an administrative user from taking actions that would compromise the integrity of the platform. This allows an attestation vector to guarantee the state of the platform for some period of time into the future, mitigating Time of Check to Time of Use class attacks.

Both implementations, summarised above, also mean that an application P must attest to a third party to ensure the correctness of the secure kernel in layer l_1, as described in Section 4.5.2 on page 123. Our proposed TPM/XOM architecture attempts to mitigate these threats by implementing curtained memory in layer l_0, and enforcing curtained memory on specific principals in layer l_i, where $i > 0$, from layer l_0.

The insertion attack described in Section 4.5.1, Figure 4.2 on page 121, results from TPM commands issued by two principals P and Q working on the same state inside the TPM. Generally this can be considered virtualisation of the TPM *outside* the TCB. There are a number of issues that arise from this. As shown, insertion attacks are possible, resulting in incorrect results being produced for P. Additionally, naive denial of service attacks are possible. Our proposed TPM/XOM architecture implements virtualisation of the TPM *inside* the TCB, mitigating these sorts of issues.

I don't have all the answers, but I am beginning to ask the right questions.

Lee Lorenz

6 Conclusion and Future Work

6.1 Conclusion

This thesis investigated the requirements of an open, general purpose, trusted computing framework. In Chapter 2, we defined the terms *open* and *general purpose*. Our definition of open proposed characteristics of an open-source community-developed framework that would allow it to form a social root of trust for naive users. It was compared with a closed, proprietary trusted computing framework that was intended to act as a root of trust for those same users.

Our definition of general purpose defined certain aspects of usability and functionality that a general purpose computing platform allows. We proposed a measurement of the usefulness of a trusted computing framework to be its ability to assure certain security properties, without restricting a general purpose platform to a single or special-purpose one.

We gave formal definitions of each of the four security primitives of trusted computing: curtained memory, attestation, sealed storage, and secure I/O. Each was placed in a historical context, and shown to be an evolution of a previous security feature. We discussed the concept of a cryptographic code identity used to make statements about program functionality.

We outlined the difficulties in generating a meaningful statement about a program that depended on shared libraries for functionality. Including shared libraries in code identities, for all programs that rely on them, results in updates to any one shared library invaliding many different code identities at once. Alternatively, the code identity of a program that uses shared libraries, made without including those libraries, will not reflect changes in functionality that arise through changes to shared libraries. We highlighted the reliance that applications executing in layer l_2 have on a secure kernel, executing in layer l_1. Assurance about the correct enforcement and operation of any trusted computing primitives that the secure kernel manages rely on the kernel being valid. We showed that current implementations of trusted computing frameworks require an application to attest to a remote, trusted third party, to assure the operating system and platform that they were executing on was considered trustworthy.

Chapter 3 surveyed implementations of trusted computing frameworks. Complete framework implementations from the Trusted Computing Group and Microsoft were discussed, as well as implementations discussed in academic literature. The LOCK system, that assured all trusted computing primitives except attestation, was described. This illustrated that those primitives could be constructed purely in software, and had in fact been implemented in earlier research.

Chapter 4 discussed the framework implementations surveyed in Chapter 3. We showed that the implementation of Sailer et al. [53], discussed in Section 4.4.2 on page 116, requires an enrolment scheme to manage classifications of software integrity. We also showed that any reasonable security policy for their system would require attestations to fail when the remote platform had an unknown program executing on it.

The system proposed by Marchesini et al. [40], discussed in Section 4.4.2, implemented curtained memory in software, and protected those compartments from modification from the local administrator as well. This allowed their system to give some assurance about the integrity of a compartment for some non-trivial period of time after the generation of an attestation vector. The system proposed by Sailer et al. did not provide these assurances, remaining vulnerable to Time of Check to Time of Use (TOCTOU) class attacks. We concluded that any attempt to mitigate TOCTOU attacks would require curtained memory that ensured the integrity of an application.

We proposed an insertion attack on the Trusted Computing Group's Trusted Platform Module, made possible by the expected virtualisation of the Trusted Platform Module outside the Trusted Computing Base. Motivated by this observation, we proposed architectural changes to the Trusted Platform Module. We removed the software distribution process of Execute Only Memory (XOM), so that it ensured integrity and confidentiality of an application, through its use of compartments, only after it had started executing on a platform. We mitigated the vulnerability of a specific insertion attack on our system, by making the measurement and encryption of an application atomic. This ensured that there could be no differences between the application that was measured to derive its code identity, and the application that was loaded into memory and executed. We mitigated the vulnerability to general insertion attacks, by virtualising the Trusted Platform Module on a per-compartment basis.

Our proposed architecture enabled the selective use of shared libraries, including those libraries in any statement about the functionality of a program. These statements of functionality were used in the attestation and sealed storage primitives implement by our architecture.

6.2 Future Work

This thesis has outlined a number of open problems in trusted computing research. Many of these need to be solved before an open, general purpose, trusted computing platform, as defined in Chapter 2, is possible.

An examination of the statistical model described in Section 4.3 on page 106 is of interest. If our assumptions are correct, it would allow many different compilation environments, giving distinct code identities $ID(P)$ for the same source code of P, to be compared. If so, a community may able to generate statements of $ID(P)$, that are known to have been obtained from unmodified versions of P, without requiring a prescribed compilation environment to be used for all compilations of P.

The proposed integration of XOM and the Trusted Computing Group's Trusted Platform Module specification is another avenue for future research. The XOM architecture has been implemented in software, and recently a Trusted Platform Module software emulator [11] has been released. Integrating the TPM emulator and XOM system in software would allow for analysis of performance, as well as formal verification of the security model.

Research into the implementation of curtained memory in software is considered. Specifically, research that attempts to assure separation of programs during execution with a simple, expressive, mandatory access control language that does not severely restrict the usability of the platform as a whole. Additionally, the ability to attest that assurance without requiring a statement about the integrity of the platform as a whole would reduce privacy concerns.

Bibliography

[1] Apache HTTP server project. Available online. Cited December, 2004. http://httpd.apache.org/.

[2] Hashcash. Available online. Cited December, 2004. http://www.hashcash.org/.

[3] IBM hardware - IBM PCI cryptographic coprocessor. Available online. Cited December, 2004. http://www-3.ibm.com/security/cryptocards/overhardware.shtml.

[4] IBM PCI cryptographic coprocessor. Available online. Cited December, 2004. http://www-3.ibm.com/security/cryptocards/pcicc.shtml.

[5] Open Source Initiative. Available online. Cited December, 2004. http://www.opensource.org/.

[6] OpenSSH. Available online. Cited November, 2004. http://www.openssh.org/.

[7] Plug and Play - Architecture and Driver support. Available online. Cited December, 2004. http://www.microsoft.com/whdc/system/pnppwr/pnp/default.mspx.

[8] Reusable proof of work. Available online. Cited December, 2004. http://www.rpow.net/.

[9] Security-Enhanced Linux. Available online. Cited November, 2004. http://www.nsa.gov/selinux/.

[10] Sendmail. Available online. Cited December, 2004. http://www.sendmail.org/.

[11] Software-based TPM emulator. Available online. Cited February, 2005. https://developer.berlios.de/projects/tpm-emulator.

[12] Trusted Computing Group. Available online. Cited December, 2004. https://www.trustedcomputinggroup.org/home.

[13] Universal Plug and Play. Available online. Cited November, 2004. http://www.upnp.org.

[14] WinHEC: Microsoft revisits NGSCB security plan. Available online. Cited December, 2004. http://napps.nwfusion.com/news/2004/0505msngscb.html.

[15] XBox games console. Available online. Cited December, 2004. http://www.xbox.com/.

[16] *Trusted Computer System Evaluation Criteria*. Department of Defense, 1985. Department of Defense Standard, DoD 5200.28-STD.

[17] ANDERSON, R. J. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley and Sons, 2002.

[18] ARBAUGH, W., FARBER, D., AND SMITH, J. A secure and reliable bootstrap architecture. In *Proceedings of IEEE Symposium on Security and Privacy* (May 1997), pp. 65–71.

[19] BARAK, B., GOLDREICH, O., IMPAGLIAZZO, R., RUDICH, S., SAHAI, A., VADHAN, S., AND YANG, K. On the (im)possibility of obfuscating programs. In *Advances in Cryptology - CRYPTO '01* (Santa Barbara, California, August 2001), vol. 2139 of *Lecture Notes in Computer Science*, pp. 1–18.

[20] BARRETT, M., AND THOMBORSON, C. Using NGSCB to mitigate existing software threats. In *2nd International Workshop on Certification and Security in Inter-Organizational E-Services (CSES)* (2004), Kluwer Academic Press, to be published.

[21] CAMENISCH, J. Better privacy for trusted computing platforms. In *9th European Symposium On Research in Computer Security ESORICS* (2004), Springer-Verlag.

[22] CRAM, E. Status of NGSCB. Private correspondence, 9 November, 2004.

[23] CRAM, E., AND KAPLAN, K. Next-Generation Secure Computing Base - Overview and Drilldown. Presentation at Microsoft Professional Developers Conference, Los Angeles, 27 October 2003.

[24] DYER, J., LINDEMANN, M., PEREZ, R., SAILER, R., VAN DOORN, L., AND SMITH, S. Building the IBM 4758 secure coprocessor. *Computer 34*, 10 (2001), pp. 57–66.

[25] ENGLAND, P., LAMPSON, B., MANFERDELLI, J., AND WILLMAN, B. A trusted open platform. *Computer 36*, 7 (2003), pp. 55–62.

[26] ENGLAND, P., AND PEINADO, M. Authenticated operation of open computing devices. In *Proceedings of the 7th Australian Conference on Information Security and Privacy* (2002), Springer-Verlag, pp. 346–361.

[27] GACEK, C. An interdisciplinary perspective of dependability in open source software. In *Proceedings of the Building the Information Society: Proc. IFIP 18th World Computer Congress* (Toulouse, France, August 2004), R. Jacquart, Ed., Kluwer Academic Publishers, pp. 685–692.

[28] GARFINKEL, T., ROSENBLUM, M., AND BONEH, D. Flexible os support and applications for trusted computing. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (May 2003), pp. 145–150.

[29] IBM CORPORATION. *CP/Q Operating System Overview for OEMs*, June 1998. `ftp://www6.software.ibm.com/software/cryptocards/lsldesgd.pdf`.

[30] INTEL CORPORATION. *Processor Serial Number Questions and Answers*, 2003. `http://www.intel.com/support/processors/pentiumiii/psqa.htm`.

[31] Kaliski, B., and Staddon, J. *PKCS #1: RSA Cryptography Specifications Version 2.0*, October 1998. `http://www.faqs.org/rfcs/rfc2437.html`.

[32] Krawczyk, H., Bellare, M., and Canetti, R. *HMAC: Keyed-hashing for message authentication*, February 1997. `http://www.ietf.org/rfc/rfc2104.txt`.

[33] Lakhani, K., and Wolf, R. G. Why hackers do what they do: Understanding motivation and effort in free/open source software projects. Tech. Rep. 4425-03, MIT Sloan Working Paper, September 2003. `http://ssrn.com/abstract=443040`.

[34] Lampson, B. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems* (Princeton University, 1971), pp. 437–443.

[35] Lampson, B., Abadi, M., Burrows, M., and Wobber, E. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems 10*, 4 (1992), 265–310.

[36] Landwehr, C. E. Trusting strangers. In *Open-Source Software in Dependable Systems* (2004), IFIP, To be published.

[37] Lie, D., Thekkath, C. A., and Horowitz, M. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003), ACM Press, pp. 178–192.

[38] Lipmaa, H., Rogaway, P., and Wagner, D. Comments to NIST concerning AES-modes of operations: CTR-mode encryption. In *Symmetric Key Block Cipher Modes of Operation Workshop* (Baltimore, Maryland, USA, October 2000).

[39] Marchesini, J., Smith, S. W., Wild, O., and MacDonald, R. Experimenting with TCPA/TCG hardware, or: How I learned to stop worrying and love the bear. Technical Report TR2003-476, Department of Computer Science, Dartmouth College, December 2003.

[40] Marchesini, J., Smith, S. W., Wild, O., Stabiner, J., and Barsamian, A. Open-source applications of TCPA hardware. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)* (2004), IEEE Computer Society, pp. 294–303.

[41] Microsoft Corporation. *End-User License Agreement for Microsoft Software, Windows Server 2003 Enterprise Edition*, March 2003.

[42] Microsoft Corporation. *Hardware Platform for the Next-Generation Secure Computing Base*, December 2003. Available from `http://www.microsoft.com/resources/NGSCB/documents/NGSCBhardware.doc`.

[43] Microsoft Corporation. *NGSCB: Trusted Computing Base and Software Authentication*, November 2003. Available from `http://www.microsoft.com/resources/NGSCB/documents/NGSCB_tcb.doc`.

[44] Microsoft Corporation. *Security Model for the Next-Generation Secure Computing Base*, 2003. Available from `http://www.microsoft.com/resources/NGSCB/documents/NGSCB_Security_Model.doc`.

[45] NEVILL-MANNING, C. Finding needles in a 20 TB haystack, 200 million times a day. Presentation at the University of Auckland, June 2004.

[46] O'BRIEN, R., AND ROGERS, C. Developing applications on LOCK. In *Proceedings of the 14th National Computer Security Conference* (October 1991), NIST/NCSC, pp. 147–156.

[47] PASHALIDIS, A., AND MITCHELL, C. J. Single sign-on using trusted platforms. In *ISC* (2003), C. Boyd and W. Mao, Eds., vol. 2851 of *Lecture Notes in Computer Science*, Springer, pp. 54–68.

[48] RAY, K., AND CRAM, E. Interview at Microsoft Professional Developers Conference, 29 October 2003.

[49] RAYMOND, E. S. *The Cathedral and the Bazaar*. O'Reilly, 2001. `http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/`.

[50] SAFFORD, D. *Why TCPA*. IBM Research, October 2002. `http://www.research.ibm.com/gsal/tcpa/why_tcpa.pdf`.

[51] SAFFORD, D., KRAVITZ, J., AND VAN DOORN, L. Take control of TCPA. *Linux J. 2003*, p. 112 (2003), p. 2.

[52] SAILER, R., JAEGER, T., ZHANG, X., AND VAN DOORN, L. Attestation-based policy enforcement for remote access. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security* (2004), ACM Press, pp. 308–317.

[53] SAILER, R., ZHANG, X., JAEGER, T., , AND VAN DOORN, L. Design and implementation of a tcg-based integrity measurement architecture. In *Thirteenth Usenix Security Symposium* (August 2004), pp. 223–238.

[54] SAMI SAYDJARI, O., BECKMAN, J., AND LEAMAN, J. LOCKing computers securely. In *Proceedings of the 10th DoD/NBS Computer Security Conference* (1987), NBS, pp. 129–140.

[55] SAMI SAYDJARI, O., BECKMAN, J., AND LEAMAN, J. LOCK trek: Navigating uncharted space. In *Proceedings of the "IEEE" Symposium on Security and Privacy* (1989), pp. 167–175.

[56] SCHNEIER, B. Open source and security. Crypto-Gram Newsletter, September 1999. `http://www.schneier.com/crypto-gram-9909.html`.

[57] SCHNEIER, B. Getting out the vote — why is it so hard to run an honest election? San Francisco Chronicle, October 2004. `http://www.schneier.com/essay-067.html`.

[58] SCHOEN, S. Trusted computing: Promise and risk. Available online. Cited Nov, 2004, December 2003. `http://www.eff.org/Infrastructure/trusted_computing/20031001_tc.php`.

[59] SHERRIFF, P. Intel PSN no threat to personal freedom. Available online, November 1999. `http://www.theregister.co.uk/1999/11/29/intel_psn_no_threat/`.

[60] SMITH, S. W. *Verifying Type and Configuration of an IBM 4758 Device*. IBM T.J. Watson Resarch Center, February 2000. `ftp://www6.software.ibm.com/software/cryptocards/verify.pdf`.

[61] SPINELLIS, D. Reflections on trusting trust revisited. *Communications of the ACM 46*, 6 (2003), 112.

[62] STRONGIN, G. Platform Security Architect, Advanced Micro Devices. Interview at Microft Professional Developers Conference, Los Angeles, 28 October 2003.

[63] SUH, G. E., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. Efficient memory integrity verification and encryption for secure processors. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture* (2003), IEEE Computer Society.

[64] THEKKATH, D. L. C., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J., AND HOROWITZ, M. Architectural support for copy and tamper resistant software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (2000), ACM Press, pp. 168–177.

[65] THOMPSON, K. Reflections on trusting trust. *Commun. ACM 27*, 8 (1984), pp. 761–763.

[66] TRUSTED COMPUTING GROUP. *TCG PC Specific Implementation Specification*, v1.1 ed., August 2003. `https://www.trustedcomputinggroup.org/downloads/TCG_PCSpecificSpecification_v1_1.pdf`.

[67] TRUSTED COMPUTING GROUP. *TCG Software Stack Specification*, August 2003. `https://www.trustedcomputinggroup.org/downloads/TSS_Version__1.1.pdf`.

[68] TRUSTED COMPUTING GROUP. *TPM Main Part 2 TPM Structures*, v1.2 ed., October 2003. `https://www.trustedcomputinggroup.org/downloads/tpmwg-mainrev62_Part2%_TPM_Structures.pdf`.

[69] TRUSTED COMPUTING GROUP. *TPM Main Part 3 Commands*, v1.2 ed., October 2003. `https://www.trustedcomputinggroup.org/downloads/tpmwg-mainrev62_Part3%_Commands.pdf`.

[70] TRUSTED COMPUTING GROUP. *Backgrounder*, November 2004. `https://www.trustedcomputinggroup.org/downloads/background_docs/TCG_Backgrounder_November_2004.pdf`.

[71] TRUSTED COMPUTING GROUP. *TCG Specification Architecture Overview*, v1.2 ed., April 2004. `https://www.trustedcomputinggroup.org/downloads/TCG_1_0_Architecture_Overview.pdf`.

[72] TRUSTED COMPUTING GROUP. *TPM Main Part 1 Design Principles*, v1.2 ed., October 2004. `https://www.trustedcomputinggroup.org/downloads/tpmwg-mainrev62_Part1%_Design_Principles.pdf`.

[73] YANG, J., ZHANG, Y., AND GAO, L. Fast secure processor for inhibiting software piracy and tampering. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture* (2003), IEEE Computer Society, p. 351.

[74] YE, Y., AND KISHIDA, K. Toward an understanding of the motivation open source software developers. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* (2003), IEEE Computer Society, pp. 419–429.