

<http://researchcommons.waikato.ac.nz/>

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

Tamper-Evident Data Provenance

A thesis
submitted in fulfilment
of the requirements for the Degree
of
Master of Engineering
at the
University of Waikato
by
Mohammad Bany Taha



THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2016

Abstract

Data Provenance describes what has happened to a users data within a machine as a form of digital evidence. However this type of evidence is currently not admissible in courts of law, because the integrity of data provenance cannot be guaranteed. Tools which capture data provenance must either prevent, or be able to detect changes to the information they produce, i.e. tamper-proof or tamper-evident.

Most current tools aim to be tamper-evident, and capture data provenance at a kernel level or higher. However, these tools do not provide a secure mechanism for transferring data provenance to a centralised location, while providing data integrity and confidentiality.

In this thesis we propose a tamper-evident framework to fill this gap by using a widely-available hardware security chip: the Trusted Platform Module (TPM). We apply our framework to Progger, a cloud-based provenance logger, and demonstrate the completeness, confidentiality and admissibility requirements for data provenance, enabling the information to be used as digital evidence in courts of law.

Acknowledgements

I would like to thank my supervisor Dr. Ryan Ko and my co-supervisor Dr. Sivadon Chaisiri for their support and for the time they spent with me through this work. I would also like to thank all members of Cyber Security Researchers of Waikato (CROW) lab, in particular, Alan Tan, Baden Delamore, Jeff Garae and Mark Will for their help and the time that they spent with me for this work.

I am especially grateful to my father and mother, without whom this work would not have been possible.

Contents

List of Figures	5
List of Tables	6
1 Introduction	3
1.1 Motivation	3
1.2 Research Goal	6
1.3 The Objectives	6
1.4 The Scope	7
1.5 Key Contributions	7
1.6 Definition of Terms	8
1.7 The Outline	9
2 Literature Review	11
2.1 History of Tamper-Evidence	11
2.2 Related Work	13
2.2.1 System Management Facilities	14
2.2.2 Hash Operations	15
2.2.3 Audit Trails	16
2.2.4 Tools that Collect Provenance at Network Level and in Cloud Computing	17
2.2.5 Capture System Call	18
2.3 Background	19
2.3.1 AEGIS	19
2.4 Requirements for Digital Evidence	20
2.4.1 Reliability	21
2.4.2 Authenticity	21
2.4.3 Admissibility	21
2.4.4 Completeness	21
2.4.5 Believability	21
2.5 Tamper-Evidence	22
2.6 Summary of the Gaps in Tamper-Evidence Tools	24

2.6.1	Integrity and Confidentiality	24
2.6.2	Provide Tamper-Evident at Boot Stage	25
2.6.3	Remote Attestation	25
2.6.4	Collecting Provenance at Application Level	25
2.6.5	Detect any tampering for the application that generates provenance logs	25
3	Overview of the Trusted Platform Module	27
3.1	What a TPM Provides	27
3.2	TPM Architecture	28
3.2.1	Platform Configuration Register (PCR)s	32
3.2.2	Keys	32
3.2.2.1	Non-Migratable Keys	33
3.2.2.2	Migratable Keys	34
3.3	Integrity Measurement Architecture (IMA)	34
3.4	Intel-TXT	36
3.5	TrustedGRUB	39
4	Framework Design	42
4.1	Client Side	42
4.2	Provenance and Backup Server	45
5	Framework Implementation	48
5.1	Client machine	49
5.1.1	Provenance Generator and Provenance Logs Buffer	49
5.1.2	Create Chunk	51
5.1.3	Hash Chunk and Transfer the Chunk to the Provenance Server	54
5.2	Provenance Server	55
5.3	Backup Server	58
5.4	TPM-Quote	60
6	Framework Advantages	63
6.1	Guarantee that Data Provenance is created and transmitted in a secure environment	63
6.1.1	Tampered Chunk	64
6.1.2	Tampered Provenance Generator	64
6.1.3	Trusted BIOS Configuration	64
6.1.4	Tampered Bootloader	64
6.1.5	Change Kernel OS	65
6.2	Data Provenance with Integrity is Guaranteed	65

6.3	Data Provenance with Confidentiality is Guaranteed	65
6.4	Data Provenance with Availability is Guaranteed	66
6.5	Remote Attestation	66
7	Evaluation	67
7.1	Detecting tampering in machine components	67
7.2	Detecting Tampering at runtime	69
7.3	Results	71
8	Conclusion and Future Work	75
8.1	Conclusion	75
8.2	Future Work	77
9	List of Publications	79
10	References	80
	Appendix A	89
	Appendix B	91
	Appendix C	93

List of Figures

1.1	Stages from Boot Time Till Run Time	4
1.2	Summary of Chapter 1 and Chapter 2	6
2.1	Timeline of Evolutionary History of Tamper-Evidence	12
2.2	Tamper-Evidence Landscape	14
2.3	AEGIS Boot Control Flow	19
3.1	TPM Components	28
3.2	Root Of Trust	30
3.3	TPM Key Hierarchy	33
3.4	Measurement List	35
3.5	TPM Based Integrity Measurement	35
3.6	The mechanism of Intel-TXT	38
3.7	Chain of Trust, extended Boot Loader is in use (TrustedGRUB)	40
4.1	Framework Design	43
5.1	Framework Flowchart	48
5.2	Provenance Generator	50
5.3	Measuring the chunk file by IMA	52
5.4	Using IMA to detect tampering could happen for the chunk .	52
5.5	The mechanism of hash chunk and how the administrator re- motely check the status of client machine	54
5.6	The operations on Provenance Server	56
5.7	Intel-TXT technique and Tboot	58

5.8	Backup Server	59
5.9	Remote Attestation Using TPM-Quote	61
7.1	PCRs in TPM	68
7.2	Detect BootLoader Tampered	69
7.3	TPM keep the values of SRTM and DRTM inside PCRs . . .	70
7.4	Comparison Between Files Created By Progger	71
B.1	Sample of Chunk File	91
B.2	Sample list of IMA runtime measurements	92

List of Tables

2.1	Comparison Between Tamper-Evident Technologies	24
5.1	Progger's Log Format	51
7.1	Evaluation	74
A.1	TPM Locality	89
A.2	The Standard Usage of PCRs	90
C.1	Some of TPM Commands	94

Acronyms

ACM Authenticated Code Module

AIK Attestation Identity Key

AIK Attestation Identity Key

BIOS Basic Input/Output System

CMOS Complementary Metal-Oxide Semiconductor

CRTM Core Root of Trust for Measurement

DRTM Dynamic Root of Trust for Management

EK Endorsement Key

EK Endorsement Key

IMA Integrity Measurement Architecture

IPL Initial Program Loader

MBR Master Boot Record

MLE Measurement List Environment

OS Operating System

PCR Platform Configuration Register

POST Power-On Self Test

RAM Random Access Memory

RNG Random Number Generator

ROM Read Only Memory

SMM System Management Mode

SRK Storage Root Key

SRTM Static Root of Trust for Management

TCPA Trusted Computing Platform Alliance

TCP Transmission Control Protocol

TLS Transport Layer Security

TPM Trusted Platform Module

TXT Trusted Execution Technology

TXT Trusted eXecution Technology

UUID Universally Unique Identifier

Chapter 1

Introduction

1.1 Motivation

Tamper-evidence tools inform users that their machines have been tampered with. These tools must be accurate so that they can be used in courts of law to prove that tampering occurred. They must also provide secure storage for the evidence, and secure environments for the machines which host them. Finally, these tools must cover the entire process, from boot time to the time of inquiry.

Preservation of data integrity is critical in providing valid evidence. The tools used to provide tamper-evidence must be certified by the authorities and the court(if requested by the court).

Many technologies provide tamper-evidence. Most of these technologies are based on audit provenance by collecting the provenance logs, and then examining these logs in order to detect tampering and anomalies. Provenance logs in virtual or physical machines keep track of the operations in the system (providing answers to questions such as who executes the operation? and when did the event happen?). But provenance logs can be tampered with by malicious users, which renders them useless.

Cloud computing presents a new tamper-evidence challenge. Since a physical machine can contain a set of virtual machines that have a set of logs,

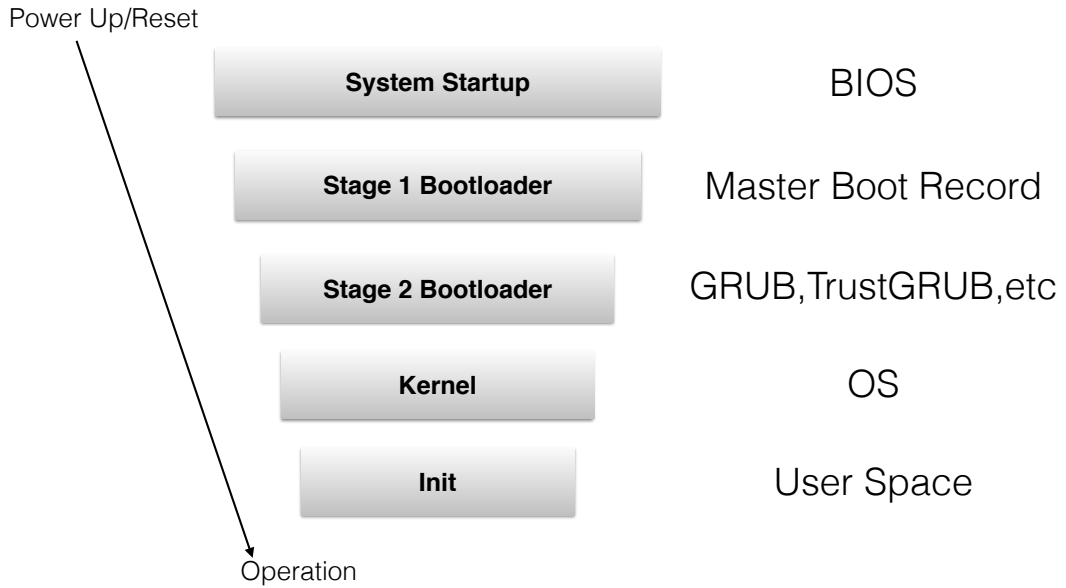


Figure 1.1: Stages from Boot Time Till Run Time

auditing the logs for every virtual machine is very difficult. Providing secure storage for the logs is another challenge. However, the main challenge still arises when an adversary tries to tamper with provenance logs, which means that we cannot trust the tampered logs. Therefore, it is impossible to know what has happened in a system if its provenance logs have been tampered with. It is therefore important to be able to guarantee that provenance logs have been *created*, *stored*, or *transmitted* securely and that no one can tamper with them. Malicious users, including insiders with high-level access, have the ability to access the logging system, and can perform unlogged activities or tamper with history provenance. This will result in uncertainty about whether this provenance was generated by the system itself or inserted by someone else. Therefore, the system must have the ability to monitor everything from boot time until run time.

Fig. 1.1 shows the stages of the computer system's boot process (from boot time until run time). This begins when the user switches on the machine and the system starts up (bootup). Booting is defined as a bootstrapping process that starts the Operating System (OS) when the user switches on a computer system [14]. The boot sequence is the set of operations the computer performs

when it is switched on. The processor executes code at a pre-defined location that is Basic Input/Output System (BIOS), which is stored in a flash memory on the motherboard (0xFFFF0), and must determine which devices are candidates during the boot process. The Master Boot Record (MBR) contains the primary bootloader in the stage 1 bootloader. After the MBR is loaded into Random Access Memory (RAM), the BIOS yields control to the MBR. Typically the job of the bootloader in MBR is to load the stage 2 bootloader. The bootloader is loaded into RAM and executed. The stage 2 bootloader is loaded and executed in RAM. After this, the stage 2 bootloader passes control to the kernel image, which checks the system components. Finally, the OS is loaded.

It is essential to know that the process stages (in Fig. 1.1) run as expected (e.g., the proper bootloader). Otherwise, any changes occurring in these stages cannot be detected and we cannot know what has really happened. This means that a tamper-evidence solution is not credible. For example, a hacker can change the bootloader (stage 2 in Fig. 1.1) and access the system without being detected due to the changes in the bootloader [27]. According to this scenario, tamper-evidence tools should detect any changes in the system from boot time until run time in order to know what has happened in the system.

To the best of our knowledge, few tamper-evidence solutions provide full tamper-evidence. This will be discussed further in Chapter 2. The meaning of the term full tamper-evidence tools is that these tools can detect any tampering that occurs from boot time until run time. Most of the tamper-evidence solutions so far proposed focus on collecting provenance logs at the application level. These solutions then analyse the collected provenance to determine what has happened in the machine at the application level only. These solutions are only partial tamper-evidence because as we mentioned in the previous paragraph, tamper-evidence tools should detect any changes in the system from boot time until run time (i.e., system level to application level).

Fig.1.2 provides a summarised overview of the content of chapters 1 and 2.

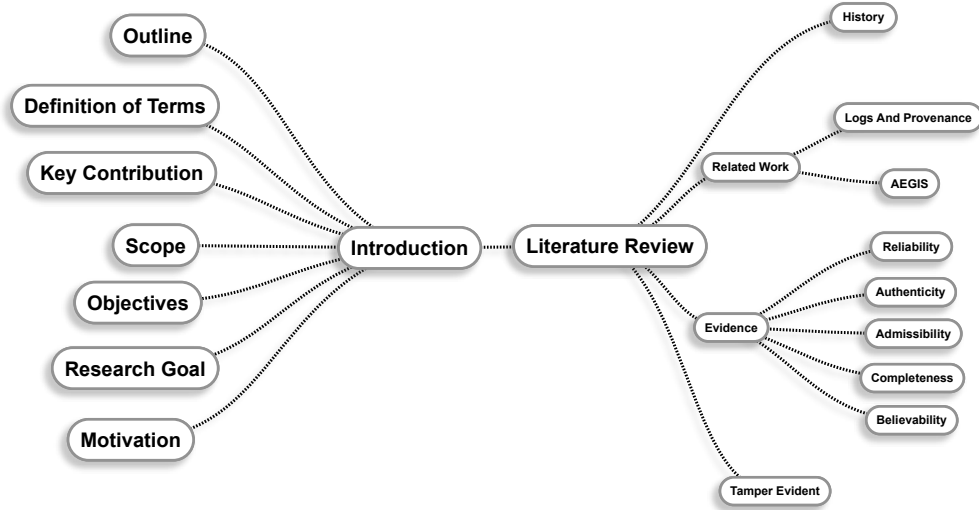


Figure 1.2: Summary of Chapter 1 and Chapter 2

1.2 Research Goal

In this thesis, we propose a framework to enable full tamper-evidence and preserve the confidentiality and integrity of data provenance using Trusted Platform Module (TPM) from boot time until run time (application level). We also focus on providing remote attestation for **all** types of data provenance logs (e.g., system provenance and application provenance) generated in distributed systems such as cloud computing.

1.3 The Objectives

We have the following objectives:

1. To preserve the confidentiality and integrity of data provenance using TPM. Our objective is also to store provenance logs in trusted backup servers to guarantee the availability of data provenance.
2. To provide remote attestation for client machines (physical or virtual) and backup servers. This will help us to check the status of a remote machine whether it is in well-known (this terminology is explained in section 1.6) status or not.

1.4 The Scope

The aim of this thesis is to provide tamper-evidence for data provenance logs in physical and virtual machines. In our framework, we collect provenance logs from dedicated client machines and store them in a provenance server. We later move the old provenance logs from the provenance server to the backup server for archival purposes. In this thesis we focus on the provenance logs in the system from boot time until run time. We do not focus on the network logs. We use special tools for remote attestation to guarantee that we provide a secure way to check the system status for the provenance logs in remote machines.

We do not provide tamper resistance solutions, but we use PCRs of TPM to store the hash values of our important programs (e.g. progger, backup software) and these PCRs provide tamper-resistance for the provenance that is stored inside them.

The TPM cannot prevent a cold boot attack [30] (explained in section 1.6), so we assume that this type of attack is not part of the threat model.

1.5 Key Contributions

The main contributions of this thesis are summarised as follows:

- We propose a framework to provide tamper-evidence and preserve the confidentiality and integrity of data provenance using the TPM.
- We store provenance logs in trusted and backup servers to guarantee the availability of data provenance. The framework also allows users to check the system status of client machines.
- We develop a framework that provides a secure environment for the provenance in different stages; *at-creation*, *at-rest*, and *in-transit*. This means the provenance logs collected are admissible, complete, and con-

fidential. Therefore, the provenance logs can be used as evidence in a court of law.

1.6 Definition of Terms

This section explains the terms used in this thesis.

- *Tamper-Evidence* is evidence that detects any tampering that could affect provenance logs.
- *Partial Tamper-evidence* provides evidence about what has happened in the machine at either system level or application level.
- *Full Tamper-evidence* is evidence of what has happened in the machine from boot time until run time.(i.e., system and application levels)
- *Chain of Trust*: is a validation of each component in a computer system from boot time by measuring and hashing these components and storing these hash values inside a TPM chip.
- *Root of Trust* is a set of functions in a trusted computing module that is always trusted by the computer's OS. The root of trust serves as separate computing engine controlling the trusted computing platform cryptographic processor on the machine.
- *TPM* is a hardware chip installed on a motherboard. This chip provides tamper-evidence for the machine from boot time until run time.
- *PCR* is a register inside TPM used to store bootloader or machine components hash values.
- *IMA* is a linux kernel module that maintains a list of file hash values and aggregates the integrity values over this list inside TPM.

- *Remote attestation* is a method used to allow users or administrators to check the system status for remote machines in a secure way (e.g., TPM-Quote).
- *Well-Known status* The status of the machine is stored inside PCRs, where PCRs store the hash value of computer components (such as BIOS or bootloader). Any changes in the PCRs values mean tampering has occurred, and that means the status of the system after this tampering will not be normal and thus not Well-Known.
- *Cold Boot* is a physical attack, in which an attacker is able to retrieve encryption keys from a running operating system after using a cold reboot to restart the machine [46].

1.7 The Outline

This thesis is organised as follows:

- Chapter 2 discusses the history of tamper-evidence, and related work. It also covers the background of our research area and indicates the gaps in past and existing solutions.
- Chapter 3 discusses the architecture of TPM, and the components of TPM.
- Chapter 4 discusses the design of our framework, and presents each component in the design.
- Chapter 5 discusses the details of our framework implementation.
- Chapter 6 addresses the advantages of our framework.
- Chapter 7 provides evaluation of our work, by explaining and testing some attacks that could occur.

- Chapter 8 presents the conclusion and discusses potential future research directions.

Chapter 2

Literature Review

This chapter presents the literature review. We will first discuss the history of tamper-evidence, to understand the flow of development in tamper-evidence tools. We also discuss related work in order to understand the technologies and methods that provide tamper-evidence by generating, collecting and storing provenance logs. We then discuss the requirements of legal evidence and how provenance logs should comply with these requirements to be admissible as evidence in courts of law. Finally, we discuss gaps in existing tamper-evidence technologies.

2.1 History of Tamper-Evidence

Fig. 2.1 shows the timeline of the evolution of tamper-evidence research.

In 1964, the committee of the IBM SHARE group was concerned about the problem of maintaining security of data, especially for those systems which allowed multiple program execution [28]. The committee's report discussed the monitoring of program instruction violations at the hardware level. The OS at that time, IBM OS/MVT, was not designed to prevent deliberate user-tampering with the OS [11]. The committee suggested that the first step in preventing impairment of system integrity was to isolate users from each other and from the OS. Isolation features included storage protection, program interrupts, tape/disk write protection, and privileged instructions. The second

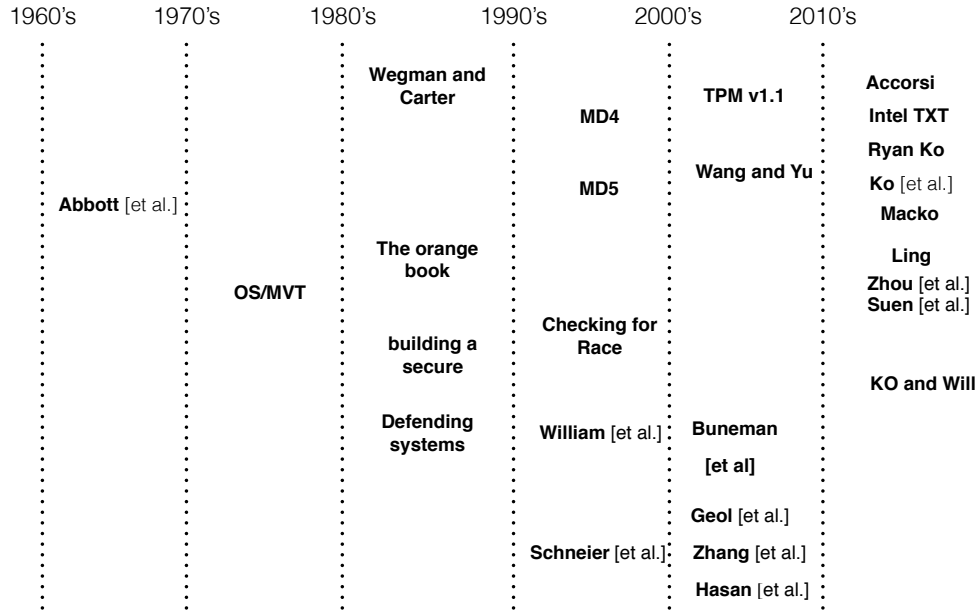


Figure 2.1: Timeline of Evolutionary History of Tamper-Evidence

suggestion was for a hardware monitor that could be attached to the existing hardware to record or trap execution actions. Monitoring how efficiently a system is being used can refer to computing resources e.g. I/O channels and disk drives. The committee also proposed a System Management Facility (SMF) to provide integrity monitoring; detecting user tampering in the operating system or files. We will further discuss SMF in section 2.2.1.

In 1981, Wegan and Carter presented a new authentication technique that could detect any modification or forged message [64]. This technique provided a secure authentication for messages sent over insecure lines.

In 1985, the United States Department of Defense [41] proposed their own definition of a trusted computer. The document described the concept of a trusted computing base; a combination of computer hardware and an OS that supports untrusted applications and users. The document described the concept of a trusted computing base; a combination of computer hardware and an OS that supports untrusted applications and users. Seven levels of trust were described, ranging from systems with minimal protection through

to those providing the highest level of security currently available. The aim was to provide objective guidelines for the evaluation of both commercial and military systems. This document was the first to point out that the boot components should be measured to provide a trusted computer.

In the 1990s, the MD4, MD5, and SHA1 [52] were used to detect tampering. In 1995 Wang and Yu [63] demonstrated a method which enabled attacks on MD5.

Bishop and Dilger in 1996 presented a tool that analysed programs for possible race conditions (undesirable situations that occur when a device or system attempts to perform two or more operations) by checking for race conditions from "time-of-check-to-time-of-use" (TOCTTOU) [17]. The results located five previously undiscovered potential race conditions in a very widely used program. In 1997, Arbaugh *et al.* presented the AEGIS architecture [16], which will be explained in detail in section 2.2. In 2001, Trusted Computing Platform Alliance (TCPA) announced the release of version 1.0 of its Trusted Computing Platform Specifications [34]. A TPM can provide a root of trust by measuring computer components (We will further discuss TPM in chapter 3).

In the 2000s, many tamper-evidence tools were developed using audit logs to detect tampering. These techniques require auditing and analysis of all the provenance logs in the machine. The hash chain is one of these technologies used for tamper-evidence. Like the Merkle tree data structure [21], a hash chain allows efficient and secure verification of the contents of large data structures. Accorsi presented BBOX in 2001 [13]. In 2014, Ko and Will proposed Progger [40], a kernel-level provenance logging tool which supports tamper-evidence.

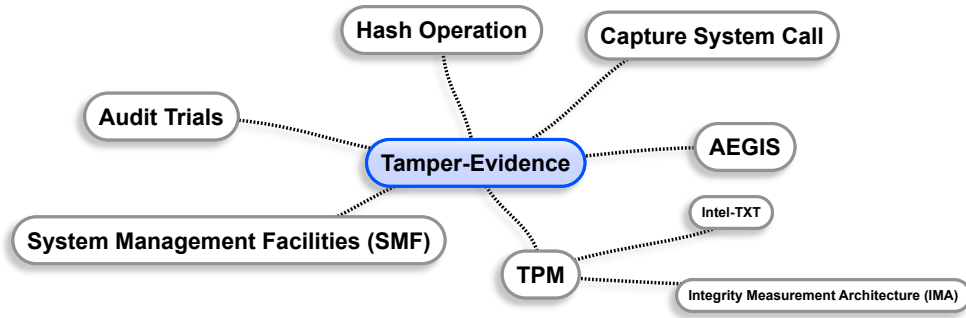


Figure 2.2: Tamper-Evidence Landscape

2.2 Related Work

This section begins by providing an overview and background of tamper-evident research. We then review past and existing work in the field of tamper-evidence for provenance logs. Based on these studies, we will outline the gaps and limitations in the current research. Fig.2.2 shows all the subsections which will be addressed in this section. These subsections are the technologies that provide tamper-evidence. We will mention briefly how these technologies work and discuss the weaknesses in each tool or technology. In section 2.6 we discuss the gap in the research on tamper-evidence tools.

2.2.1 System Management Facilities

IBM SHARE group suggested isolating users from each other and from the OS to prevent deliberate user-tampering with the OS [11]. The OS at that time was not designed to detect tampering in the file system or in any file in general. The committee pointed to System Management Facilities (SMFs) to detect user tampering in the system. An SMF is a IBM's z/OS-based that maintains records of information about system and jobs [20] using appropriate formatting. Each SMF record has a numbered type. System-related SMF records include information about the system configuration, paging activity, and workload. Job-related records contain information about the CPU time, SYSOUT activity, and data set activity of each job step. This information

identifies the resources that are repeated targets of detected unauthorised attempts to access them, and identifies the users who make detected unauthorised requests. However, these data are regularly cleared and saved in related log files (e.g. system log, SMF data records), and then these records can be tampered with by an attacker. In addition, a further possible method of attacking system logging is by preventing the generation of log data by means of appropriate tampering with the generating components. SMF data records are written, for example, in z/OS entered in a configuration member. By making changes to this member or by setting exits, it is possible to ensure that certain SMF data records are no longer written.

2.2.2 Hash Operations

Research by [32] and [68] developed application-level provenance tools for tracking the data writes of applications and validating the integrity of the provenance using checksum-based approaches. Schneier and Kelsey employed hash chains to protect audit trails [56]. They provided algorithms to create an audit trail and authenticate its entries to detect any tampering in provenance. Audit logs rarely require the selective confidentiality assurances needed for provenance. Schneier and Kelsey present secure logs as a whole, but do not allow authentication of individual modifications, so we cannot detect in which provenance record the tampering occurred. Accorsi [13] proposed BBOX, a secure logging system that covers both the transmission phase and the storage phase. BBOX allows for individual log entry verification and verification of the complete log file and uses hash chains to provide integrity for the logs. However, BBOX as a solution is not available yet, as it is undergoing a re-build to include a robust code for new kinds of crypto protocols.

Zhang *et al.* [68] considered a set of factors (e.g., users, processes, transactions), that contributed to one or more data objects through insertions, deletions, updates, and aggregations. Information about these modifications was collected and stored in the form of provenance records. The authors stated

that using a trusted solution (Hardware) is impractical for provenance collecting. However, collecting provenance in unsecured environment is useless, since if we cannot detect tampering, then we cannot guarantee that the provenance was not tampered with.

2.2.3 Audit Trails

Auditing and analysing provenance log files is one of the techniques that provide tamper-evidence for provenance logs. A provenance log file contains records of logs. These records refer to events or activities that have happened in the OS [24]; the logs describe the events. Provenance logs describe the origins and derivation of the data, starting from its original source [19] [69]. Data provenance, for example *provenance logs*, can be obtained at system, network, and application levels [69, 61]. In [42] Ling presented a solution using Linux tools (e.g., `lastlog` command, `history` command) based on system logging mechanisms to collect logs. Ling’s proposed framework uses a combination of audit logs and Linux tools to detect tampering. The framework allows users to find out what is happening in their system. However, using Linux tools is not enough to detect tampering, since these tools are limited to the application level and cannot provide a full overview of what has happened in the machine.

Accorsi [12] mentioned some logging tools that collected provenance logs. Most of these logging tools use Transmission Control Protocol (TCP) (e.g., `rsyslog`, `syslog-ng`, and `syslog-sign`) to transport logs from the devices to the collector. These are then analysed to detect tampering or unexpected behaviour. Logging tools collect provenance from different machines or from a local machine without providing security for the provenance while it is *at-rest*. In [15], Ansari *et al.* analysed and measured the performance of an efficient file system intrusion detection system, and established a complementary approach for existing access control mechanisms in the Linux kernel 2.6.x.y. They focused on preserving Modification, Access, and Creation Data and Time Stamp

(MAC DTS) of files. This mechanism can be used to maintain a log file (e.g., provenance log file) that records how the MAC DTS of the files is being accessed and changed in any underlying file system that is registered to the Virtual File System (VFS). The mechanism can trap and log activities from system calls and then hide the log from the file system. The administrator can unload the module from the system to use this log for tamper-evidence. However, the authors focus on preserving MAC DTS but not on recording every change in MAC DTS and the reasons for these changes. In addition, this mechanism does not record the type and amount of access. The provenance which is collected by this mechanism cannot be complete, and therefore cannot be used as legal evidence.

2.2.4 Tools that Collect Provenance at Network Level and in Cloud Computing

In [70] Zhou *et al.* presented Time-Aware Provenance (TAP), a provenance model that explicitly represents time, distributed state, and state changes. This mechanism helps in maintaining and querying provenance. The consistent and complete query results are guaranteed despite network variability. This mechanism captures the time, distribution, and causality of updates. This mechanism can explain why some data events exist, appear, disappear, or change. In TAP, some query language enables a declarative specification of time and changes. However, TAP still cannot answer some questions, particularly where nodes in a distribution system are compromised. In [71] Zhou *et al.* proposed Secure Network Provenance (SNP). SNP can securely construct network provenance graphs in untrusted environments. This technique can help the administrator to determine the causes and effects of specific system states (e.g., why a suspicious routing table entry is present on a certain router, or where a given cache entry originated). SNP provides capabilities for partial tamper-evidence (e.g., allowing the administrator to track down faulty or misbehaving nodes, and to assess the damage such nodes may have caused to the

rest of the system). In summary, the SNP system can help the administrator to provide tamper-evidence, but cannot determine the exact provenance of a given system status. In [29] Haeberlen *et al.* describes PeerReview, which is a system that provides accountability and fault detection for a distributed system. This system maintains a secure record of the messages sent and received by each node. When a node’s behaviour deviates from a given reference implementation, the record automatically detects the unexpected behaviour. In addition to that, nodes can sign messages, and each node is periodically checked by a correct node. In summary, this technique can detect violations of a single property (correctness of execution). It is not designed to check other properties of interest in the cloud, such as conformance to SLAs, protection of confidential data, or service availability.

In cloud computing, logs or provenance logs can be used to enhance cloud accountability and trust [39, 38, 35] as evidence for auditing, forensic, and data analysis purposes [36]. In general, using logs or provenance logs as evidence should comply with evidence rules (this will be further discussed in section 2.4).

2.2.5 Capture System Call

Ko and Will proposed Progger [40], a kernel level provenance tool to capture system calls. Progger can generate provenance logs which include data activities (e.g., reading, modifying a file) along with actors and related identifying entities (e.g., process ID, user ID, and timestamp). Progger can be deployed in cloud environments monitoring data activities within physical and virtual machines. In the past decade, several data provenance tools have been developed for cloud computing. However, Progger differs from other kernel-level monitoring tools such as Forensix [26], which do not provide tamper-evidence capabilities in their implementation. The Forensix tool generates data logs and collects them in database servers, after which, users can submit SQL queries to retrieve the events that occurred in the machine (e.g., PID, start-time, end-time). The authors did not mention any technology which provides secure,

integrity-preserving environments for the database server. This potentially makes it easy for malicious users to mask their tracks by modifying the logs in the database server or within the server which generates the logs. To the best of our knowledge, work on Forensix has been discontinued since 2011, with the last source code update not working as expected in terms of providing a secure environment.

Many other provenance tools have been proposed but have no tamper-evidence features in them. For example, in PASS [43] [47], mechanisms were developed to collect system-level provenance logs from virtual machines. Flogger and S2Logger logged file-level and block-level kernel-space system calls for cloud virtual machines and physical machines respectively [37, 60]

2.3 Background

2.3.1 AEGIS

In 1997, Arbaugh *et al.* proposed a secure bootstrap process, ensuring the integrity of the bootstrap code by constructing a chain of integrity checks beginning at power-on and extending to the last stage when control passes to the operating system [16]. The boot will abort if the hashes cannot be validated.

Fig. 2.3 shows the AEGIS boot control flow. Level 0 contains a small section of trusted software, digital signatures, public key certificate, and recovery code. This level contains the usual BIOS code, and the Complementary Metal-Oxide Semiconductor (CMOS). The second level contains all of the expansion cards and their associated Read Only Memory (ROM)s. The boot block in the third level resides in the bootable device and is responsible for loading the operating system kernel. The fourth level contains the OS. The fifth and final level contains user level programs and any network hosts.

Referring to Fig 2.3 the control passes from level to level if and only if the verification of the component in each level is successfully completed. For

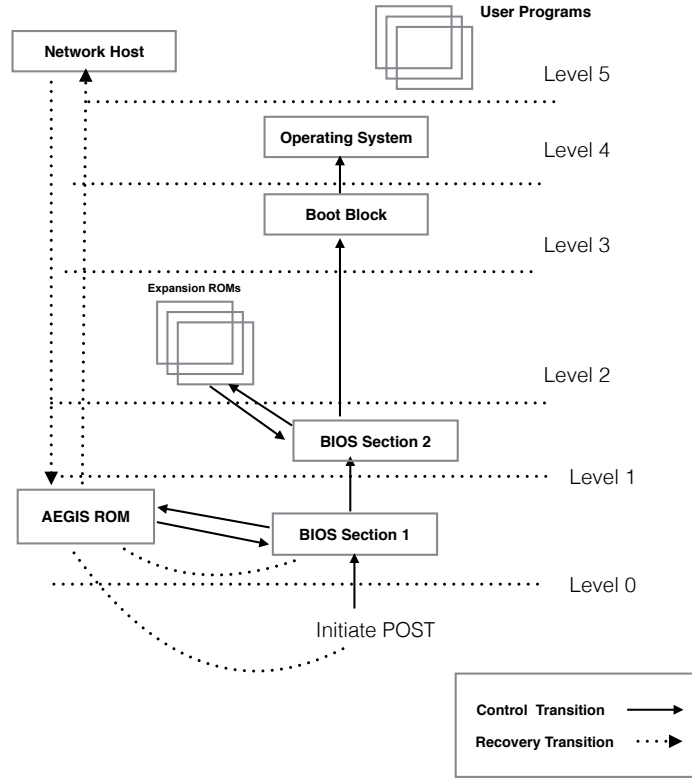


Figure 2.3: AEGIS Boot Control Flow

example, in level 0 section 1 carries out standard checksum calculations, and passes control to the next stage. Then the cryptographic hash at BIOS in section 2 is compared with the shared signature and control passes to the next level if the verification is successful. In the same way, control continues to pass to the next level after verification of each component is successfully completed.

In summary, this architecture can measure the components in hardware levels but cannot support the integrity of data at the user level. Additionally, this architecture does not support either Dynamic Root of Trust for Management (DRTM) or remote attestation. If the administrator wants to check system status remotely, then he cannot do so with the AEGIS implementation.

2.4 Requirements for Digital Evidence

In the previous section we mentioned some of the tools used to collect data provenance at different levels. While data provenance describes the origins

and derivation of the data, the integrity of provenance data is critical for the integrity of the forensic process. In [18] Braid presented **five** requirements that are essential if provenance is to be used as evidence in a court of law. Data provenance used as evidence should comply with the principles of *reliability, authenticity, admissibility, completeness, and believability* [18]. **All** these requirements should be applied to data provenance to be used as evidence.

2.4.1 Reliability

Provenance must be consistent to be admissible as evidence in a court of law. We must therefore be sure that the provenance was *created, transferred, and stored* in a trusted environment.

2.4.2 Authenticity

Evidence must be positively related to an actual incident and must be sufficient to support a finding that the item is what the proponent claims it is. If we cannot explain an event using specific evidence, we cannot use this evidence as proof of the event, and we cannot then use this evidence in court or for forensic purposes.

2.4.3 Admissibility

Evidence must be able to be used in court, and therefore must be relevant; this means that the evidence must be prove or disprove an important fact in a criminal case. If the evidence does not relate to a particular fact it will be considered irrelevant and inadmissible.

2.4.4 Completeness

Evidence must be available from boot time until runtime. While the provenance logs describe the origins and derivation of the data, starting from its original source, all these provenance logs must be available to describe any inci-

dent happening at any time (e.g., from creation time until runtime). Complete evidence means the story that the material purports to tell has no gaps in it.

2.4.5 Believability

The evidence that is presented must be clearly understandable and believable by a jury. For example, there is no point in presenting a binary dump of process memory if the jury has no idea what it all means.

2.5 Tamper-Evidence

A tamper-evidence tool is a device or mechanism that detects any tampering with provenance logs. Trusted provenance is admissible evidence. Trusted provenance must be applied to **all** the evidence requirements which are mentioned in section 2.4. Otherwise, the evidence cannot be used in court.

In section 2.2 we focused on tools that collect data provenance from different levels (system, network, and application). In this section we will focus on the tools that provide tamper-evidence. In [55, 56], the authors describe a computational method for making all log entries generated prior to the logging machine’s compromise impossible for an attacker to read, modify, or destroy without being detected. However, these solutions rely on a hash chain which requires auditors to examine every intermediate event between snapshots. In [44] a tamper-evident log is presented based on a skip list. It has logarithmic lookup times, which assumes the log is known to be internally consistent. However, proving internal consistency requires scanning the full contents of the log. In [51], Pavlou and Snodgrass showed how to integrate tamper-evidence into a relational database, and prove the existence of tampering, if suspected. Auditing these systems for consistency is expensive, requiring each auditor to visit each snapshot to confirm that any changes between snapshots are authorized. In [67] Yumerefendi *et al.* presented a network-storage service with strong accountability properties, such as taking

snapshots of the internal state, and which probabilistically detects tampering by auditing a subset of objects for correctness between snapshots. In [71], SNP provides a strong guarantee even in a system that is under attack. It makes concessions that limit its usability: SNP detects omissions and equivocations by checking inconsistencies between node logs and multiple faulty nodes might coordinate their lies in order to avoid detection. However, since SNP provides answers to queries about behaviour that is observable by at least one correct node, SNP can answer questions about network activity when one or more of the communicating nodes are un-compromised and fully functioning. (As the number of correct nodes in the network decreases, so too does the observable network state, reducing the network area the administrator can see when she issues a query.)

Network provenance used in distributed systems is recorded as a global dependency graph, where the vertices show states at a particular node. The edges show local processing or message movements across nodes. Such graphs can refer to queries about potential tampering. Tamper-evident logging could identify forgeries, omissions, and other types of tampering can be detected and used as evidence of malpractice.

In [68], Zhang *et al.* used hash chains to provide tamper-evident provenance in databases, and tackled the issue of providing audit logs of compound objects rather than just for a linear sequence of operations. The authors believe that trusted hardware is impractical due to the loosely-organised nature of provenance collection. In [32], the authors provided a thorough analysis of threats to provenance systems, and proposed a system using encryption and chained signatures to provide integrity protection. However, this research did not provide trusted solutions for data provenance. Hence, there is still the potential for attacks.

The previous mechanisms mentioned in section 2.2 [43, 47, 37, 60] are concerned with collecting data provenance in virtual machines and providing tamper-evidence.

Ko and Will developed Progger [40], a kernel-level provenance logging tool which supports tamper-evidence. Progger can capture major data activities such as kernel-level system calls relating to creation, reading, updating and deleting actions. To the best of our knowledge, few tools are able to fully satisfy these rules. The provenance must be trusted; in other words, provenance logs must be in a trusted environment *at-creation*, *at-rest* or *in-transit*.

The previously described tools for collecting provenance logs cannot fully meet the **five** requirements for the use of provenance as evidence. They do not provide a root of trust for the environment where the provenance is created and stored.

Table 2.1: Comparison Between Tamper-Evident Technologies

	VM	PM	APP	Boot	Run-Time	Tamper-Evidence
Ling <i>et al.</i> [42]	✓	✓	✓	✗	✓	NA
Zhou <i>et al.</i> [71], hasan <i>et al.</i> [32]	✓	✓	✗	✗	✗	Partial
Progger [40]	✓	✓	✗	✗	✓	Partial
Forensix, Ansari <i>et al.</i> [15]	✗	✓	✗	✗	✓	Partial
AEGIS [16]	✗	✓	✗	✓	✗	Partial
TPM + IMA + intel-TXT [9],	✓	✓	✓	✓	✓	Full

Table 2.1 shows several mechanisms used to collect data provenance at different levels, whether from a virtual or a physical machine. To the best of our knowledge, none of the previous mechanisms mentioned in this chapter can provide full tamper-evidence tools that satisfy **all** evidence requirements (discussed in section 2.4) for data provenance. To enable tamper-evidence and preserve the confidentiality and integrity of data provenance we must provide a root of trust so that we can know what has happened in the machine from boot time until run time. This will help us to know that our provenance has been *created*, *stored*, and *transported* in a secure and trusted environment.

2.6 Summary of the Gaps in Tamper-Evidence Tools

2.6.1 Integrity and Confidentiality

Provenance logs can provide evidence if they can be trusted. The integrity of the data logs is critical for the integrity of the forensic process. Also, the confidentiality of data logs is very important to make them admissible as evidence. Hence, the keys which we use to protect logs or provenance logs should be stored securely. Most tamper-evidence technologies mentioned in section 2.2 cannot provide these conditions for logs or provenance logs.

2.6.2 Provide Tamper-Evident at Boot Stage

As we mentioned in chapter 1, measuring all computer components from boot time is very important to provide full tamper-evidence. Some kind of attack could occur based on altering the BIOS version, changing the bootloader or using a boot-live attack [30]. If the administrator or the user cannot detect these changes then they cannot provide full tamper-evidence for the machine.

2.6.3 Remote Attestation

Remote attestation is a method used to help the administrator to check remote machines in a secure way [59]. The administrator or the user can check the system status to determine whether the machine is running securely. Basically, the servers are the main machines, which have a very important task to do, and the administrator will want to check the status of these remote machines rather than a local machine. This feature is not provided by most tamper-evidence tools.

2.6.4 Collecting Provenance at Application Level

Provenance logs can be generated at different levels. Most of the tamper-evidence technologies collect provenance logs or logs at the application level. The administrator in this case is not aware of the system status before that level (e.g., system level or boot time). To provide tamper-evidence we must collect provenance from boot time until run time.

2.6.5 Detect any tampering for the application that generates provenance logs

When we use provenance logs to detect tampering in the system, we need to make sure that the provenance generated by the program is correct. Malicious attack can change the code of the program and then change the details of the provenance logs. For this reason we need secure storage hash values of these applications to detect any tampering that may occur.

Chapter 3

Overview of the Trusted Platform Module

A key component of this thesis is the application of the Trusted Platform Module (TPM). Provenance logs cannot be used as evidence for forensic or auditing purposes, unless they are *created*, *transited* and *stored* in an environment that has a root of trust. To obtain a root of trust we use a (TPM) for provenance logs.

A TPM is a computer chip (microcontroller) on the motherboard that is used to securely store artifacts used in a trusted computer platform [9]. The term trusted platform means the platform always behaves in the expected manner for the intended purpose,” as defined by the Trusted Computing Group (TCG) [22].

3.1 What a TPM Provides

TPM is designed to:

1. Ensure the security of the TPM private keys. Private keys stored in TPM cannot be extracted from the chip in any form [65].
2. Detect malicious code at run time. This is done by using Intel-Trusted eXecution Technology (TXT).

3. Allow the administrator to check the system status for remote machines in a secure way (Remote Attestation).

3.2 TPM Architecture

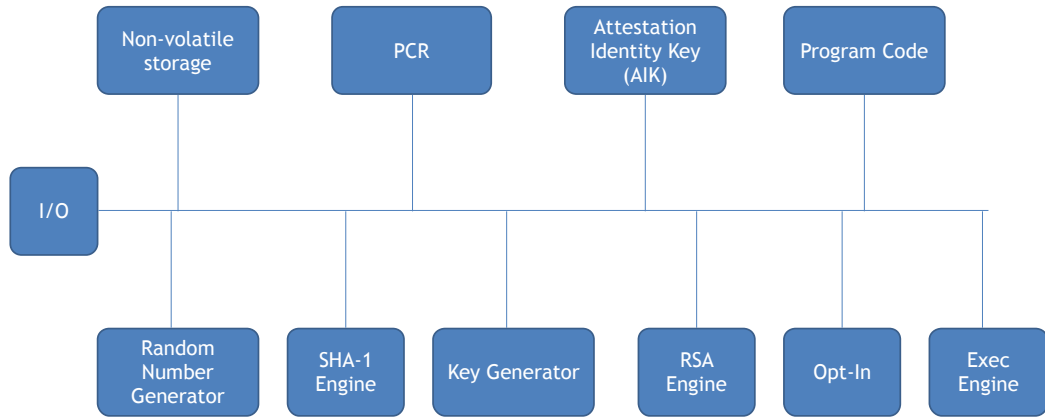


Figure 3.1: TPM Components

The components of the TPM are shown in Fig. 3.1 and are briefly explained as follows:

- *I/O* is a component that manages communication over the I/O bus.
- *Non-volatile storage* (NVRAM) is a component that holds persistent state information and identity information.
- *Random Number generator* is the source of randomness for nonce, key generator and signature in the TPM.
- *SHA-1 Engine* is an implementation of the SHA-1 algorithm and is a capability primarily used by the TPM.
- *Platform Configuration Register* (PCR) is a secure storage register that contains a 20-byte SHA-1 hash value of a specific computer component. Some PCRs are set to known values during the boot up process; for example, a PCR might contain the hash value of a BIOS, or a boot

loader. The use of different PCRs set to known values is discussed in section 3.2.1.

- *Key Generator* is a function that manages the generation of keys and nonces.
- *Attestation Identity Key* (AIK) is used to provide a cryptographic proof by signing the properties (i.e., signing PCR values such as TPM Quotes) of the non-migratable key (keys that never leave the TPM).
- *RSA Engine* is an RSA algorithm used for digital signatures and encryption.
- *Opt-In* is a component that allows the TPM to be disabled if necessary.
- *Exec Engine* is a component that runs the program code to execute a TPM command.

The I/O component manages the encoding/decoding of information flowing to and from the bus. Attestation Identity Key (AIK) is attached to the platform where the module is located. It works as an asymmetric key pair that can guarantee the integrity of the platform's identity and configuration. The RSA engine is an asymmetric algorithm used for digital signatures and encryption. The RSA Engine can also create one-time symmetric keys of up to 2048 bits. The TPM contains a 2048-bit RSA key pair called an Endorsement Key (EK), which is used during key wrapping operations, digital signing, and encrypting large blocks of data. The private parts of the EK and the RSA never leave the TPM. SHA-1 hash capability is primarily used by the TPM. The hash interfaces are exposed outside the TPM to support measurements taken during platform boot phases, and to allow environments that have limited capabilities access to a hash functions implementation of a hash algorithm. The functionality is not intended to provide an accelerated hash capability, and there are no specific performance requirements for TPM hash services. Therefore, this

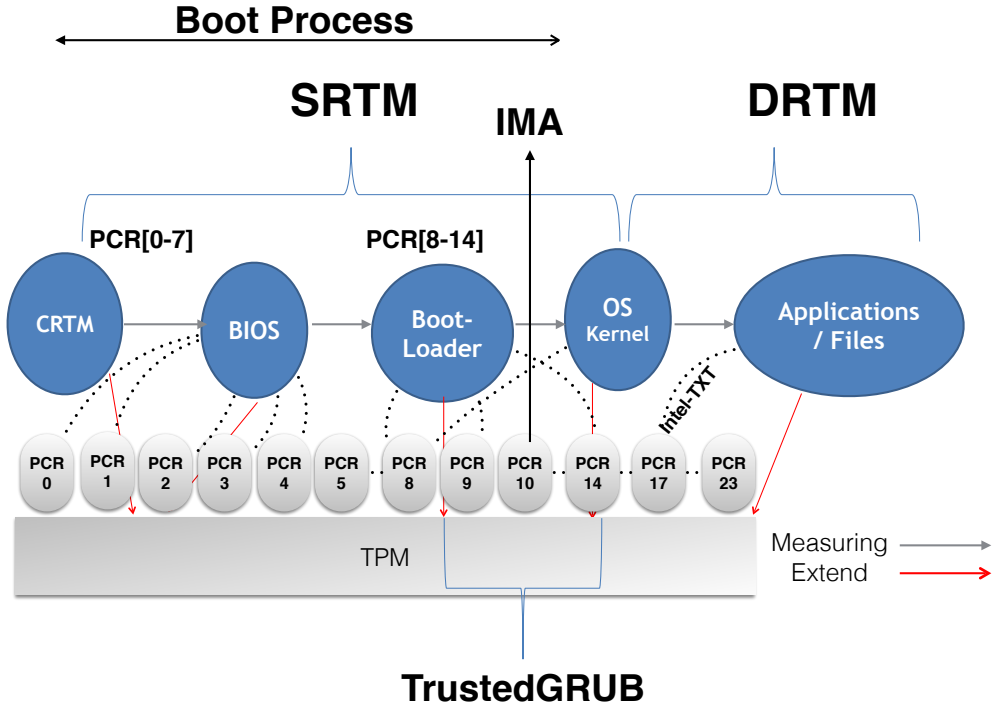


Figure 3.2: Root Of Trust

engine should only be used to compute hash values of small chunks of data. Larger chunks of data should be hashed outside the TPM if possible.

Opt-In allows the TPM to be enabled/disabled, or activated/deactivated in a secured manner. The program code operates inside the execution engine and processes the TPM commands streaming from the I/O port.

In Fig.3.2, the root of trust starts from the Core Root of Trust for Measurement (CRTM), which is a trusted code stored in the BIOS boot block. It reliably measures the integrity values of other entities, and stays unchanged during the lifetime of the platform. The CRTM is an extension of the normal BIOS, which first measures other parts of the BIOS and stores the hash values in PCR-0 and PCR-1 as shown in Fig.3.2, and then passes control to the BIOS [57].

The BIOS measures the bootloader and stores the hash value of the bootloader in PCR-4 and PCR-5 as shown in Fig. 3.2. Then the BIOS passes control to the bootloader. Next, the bootloader measures the OS kernel image

and passes control to the OS.

Each step of this boot process stores a hash value in the appropriate PCR (extend PCR) in the TPM with the measurements taken in the corresponding step in the boot process. The term *extend* in Fig. 3.2 means hashing the measurement value and saving it inside the PCR. The extend function uses the following method to compute the hash value that is to be stored in the PCR:

$$PCR_{n+1} = SHA1(PCR_n + SHA1(Component)) \quad (3.1)$$

where PCR_{n+1} denotes the new (expected) value of PCR and PCR_n denotes the current value of the PCR.

Static Root of Trust for Management (SRTM) depicted in Fig 3.2 contains a set of trusted code stored in the BIOS. This code is executed when the system is running. All the codes in the chain of the components in in Fig 3.2 are measured by the previous components. (e.g., CRTM measures the BIOS). Therefore, any changes in the codes of the components in the chain will be detected. If any change happens in the SRTM phase, the value of PCR_{n+1} will indicate the change because the value will be different from the value of PCR_n .

In Fig. 3.2, TCG defines the next phase, called the Dynamic Root of Trust for Measurement (DRTM), as the measuring of the platform at run time, which indicates the dynamic chain of trust starts on a request from the OS via a special processor instruction. Intel developed the Trusted Execution Technology (Trusted eXecution Technology (TXT)) [65] (further discussed in section 3.4), which provides the DRTM with the ability to check the secure mode of the environment at run time by checking the value of PCR-17 as shown in Fig. 3.2 (which will be discussed in subsection 3.4). Similarly, AMD implements equivalent technology called the Secure Virtual Machine (SVM) [4]. IBM provides the Integrity Measurement Architecture (IMA) to maintain a runtime measurement list (e.g., measurement for a list of sensitive files stored in PCR-10 as shown in Fig. 3.2) and test the runtime integrity of the platform

using a remote attestation feature [45].

3.2.1 PCRs

A PCR is a 160-bit storage location for discrete integrity measurements. The old version of TPM has 16 PCRs (old version TPM1.1). TPM 1.2 and the newest version have 24 PCRs, as shown in Fig. 3.2. All PCRs are shielded locations that will protect the hash data inside the TPM, preventing physical attacks. The decision about whether a PCR contains a standard measurement or is available for general use is deferred to the platform specific specification (e.g., PCR-0 is allocated for BIOS components, while PCR-23 is available for application). The PCR is designed to hold an unlimited number of measurements in the register, which it does by hashing all updates (see equation 3.1) using a cryptographic hash.

A TPM provides trusted space, so data (provenance logs) from it will be admissible and authentic as forensic evidence since the environment is monitored and measured by the TPM. Several of the tools normally used to collect machine data cannot guarantee the environment in which the data logs are created and stored. Therefore, a solution that can measure everything in the machine from boot time until run time is needed. This can be done through a TPM, because the TPM can store all hash values of the measured components in a secure and shielded location (PCR), thus guaranteeing that the hash value is related to the machine status (SRTM and DRTM) and cannot be tampered with even during a physical attack. Based on this, if an attacker uses any kind of attack at boot time (e.g., changing the bootloader) or at run time (e.g., malicious attack), this can be detected since the expected value of the PCRs will change. Then we can decide if our data logs have been tampered with or not. This will be further explained in chapter 5.

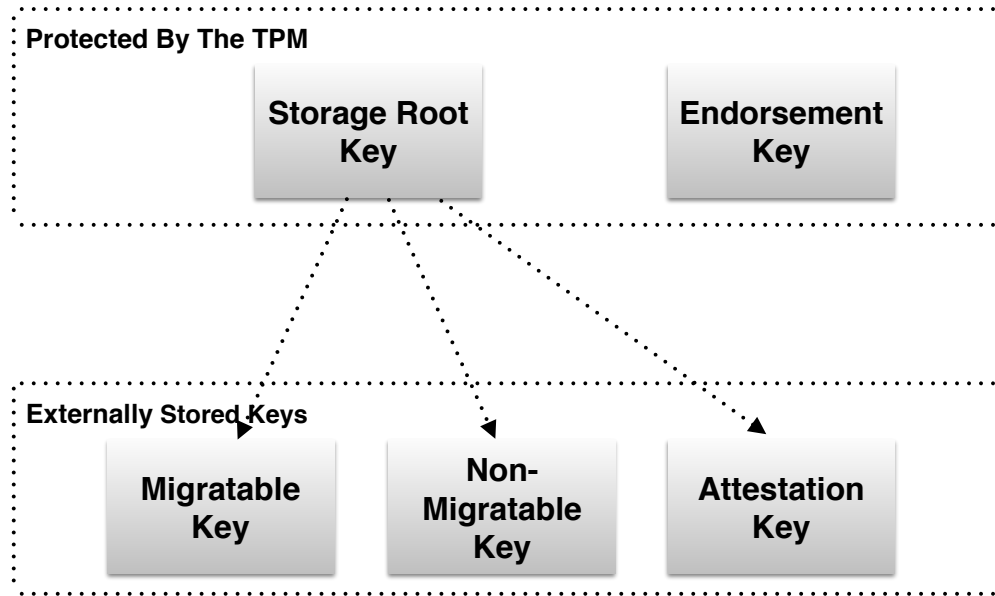


Figure 3.3: TPM Key Hierarchy

3.2.2 Keys

The TPM contains keys for different uses. Some of these keys never leave the TPM and some are migratable. Fig. 3.3 shows the hierarchy of TPM keys.

3.2.2.1 Non-Migratable Keys

Non-Migratable keys are bound to a single TPM. These keys are unique to a TPM and cannot be migrated or exported from the TPM. For example, the Endorsement Key (EK) is a public or private key pair, generated during manufacture [23]. The EK can attest to the authenticity of values produced by the TPM. This key is unique to each TPM. The EK is a non-migratable key and can be used to create identity keys. Attestation Identity Keys (AIKs) are non-migratable keypairs that are essentially aliases for the EK. The private key of an AIK never leaves the TPM in plaintext and is used only for signing data originated by the TPM.

Another example of a non-migratable TPM key is the Storage Root Key (SRK). The SRK is a keypair that is generated internally in the TPM and has a private key which never leaves the TPM. The SRK is created when the

owner of TPM takes ownership of the TPM. The SRK is used to encrypt data using the `TPM_sealed` command (which will be further explained in section 5.3).

3.2.2.2 Migratable Keys

Migratable keys can be moved and used outside a TPM. For example, the public part of AIK can be moved to a remote machine for platform authentication purposes.

3.3 Integrity Measurement Architecture (IMA)

An IMA, an open source trusted computing component can maintain a runtime measurement list and an combine integrity value for this list, in order to produce verifiable information about the software running on a Linux-based system [1]. Remote parties can use this information to assess the execution environments integrity. The measurement list is obtained by computing an SHA1 hash value for the files representing executable content and then storing all measurements since the booting of the system in a kernel-held measurement list. The IMA measures the execution environment of service (binaries, configurations, and libraries) [33].

The measurement list cannot be compromised by any software attack without the attack being detectable, since the hash value of the measurement list is stored inside PCR-10. In a trusted boot system, the IMA can be used to attest the system's runtime integrity.

In Fig. 3.4 the number '10' located at the beginning of each row refers to PCR-10, which is the default PCR for the IMA measurement. The second and fourth columns are hash values. The hash value in the fourth column is the hash value of the chunk file (i.e., *filedata-hash*). The hash value in the second column is called *template-hash* and is the result of concatenating the SHA1 hash value of PCR-0 to PCR-7 (called *boot-aggregation*) with the hash value


```

m@provenance:/home/m
File Edit View Search Terminal Help
[root@provenance m]#
[root@provenance m]# su -c 'head -5 /sys/kernel/security/ima/ascii_runtime_measurements'
10 09c04af7b11f833ce87a826c21db79c3e02df1a8 ima-7ccce77359e77cdf1e00b58f66a88fb6236cfbc boot aggregate
10 e12d5a82e4a72c4b8435fdeff3e084aa821cbb23 ima 808e809c3bb8b9adbeb9b50041bea096756cd3e9 /init
10 d36c46138fca4dcd091e2e0db07dbae2358b3853 ima 3ec3b76addd36e1b88fcd4c6fc3bc695a4d541f /init
10 bedcb1cded5eca88570df398c57c892bb147f04a ima a718a0de8b65ab58acd487f29c1dc94afc2c11da ld-2.12.so
10 33a6567fdc79b26510b574602f13f3e15cbd55ad ima db455be4bbe673e9d54a87a3b143628e2ed30c32 ld.so.cache
[root@provenance m]#

```

Figure 3.4: Measurement List

of the file. This template-hash is calculated by

$$Template - Hash = SHA1(\text{filedata-hash} || \text{boot-aggregation}) \quad (3.2)$$

The resulting hash is stored in PCR-10 with boot-aggregation.

Fig.3.5 shows how the IMA is applied for remote attestation. Measurement is initiated by the measurement agent, which induces a measurement of a file, stores the measurement in an ordered list in the kernel, and reports the extension of the measurement list to the TPM.

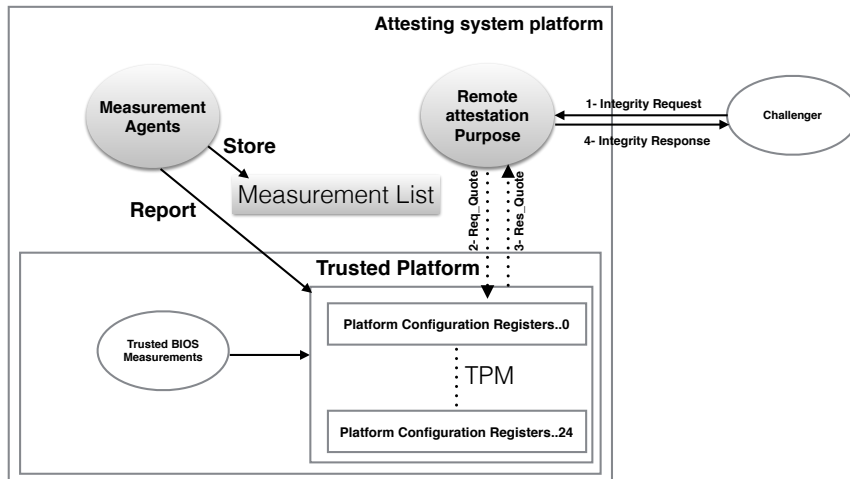


Figure 3.5: TPM Based Integrity Measurement

The integrity challenge mechanism allows a remote challenger to request

the measurement list together with the TPM signing aggregate of the measurement list (step 1 in Fig. 3.5). Upon receiving a result (steps 2 and 3 in Fig. 3.5), the attesting system first retrieves the signed aggregate from the TPM and then the measurement list from the kernel. Both (signed aggregate and measurement list from the kernel) are then returned to the challenger (step 4 Fig. 3.5). Finally, the challenger can validate the information and assess the trustworthiness of the attesting system's run-time integrity.

3.4 Intel-TXT

The TPM can measure the machine components at boot time until the kernel OS takes over (explained in section 3.2). Then we need a mechanism that can detect an attack at run time (after the OS is up and running). Intel-TXT is a hardware-based technology, designed to provide security of a computer platform [10]. Intel-TXT can be deployed in both physical and virtual environments.

Intel-TXT can ensure that no unauthorised changes can be made in critical parts of the code. This validation is performed each time the environment launches. The administrator should add security policies (e.g., to create non-migratable keys and tell the TPM to not allow anyone but the owner to evict it) to make sure that the machine runs in a trusted environment. Servers that implement Intel-TXT can demonstrate (attest) that they comply with a specific trust policy, and thus can be used to form pools of trusted servers based on the established policies. For example, Intel-TXT allows an operating system to launch if it knows the platform and system software are secure and trusted.

A successful measured launch has the following requirements:

- Authenticated Code Module (ACM) must be valid.
- Server Platform must pass the launch the control policy.

- Measurement List Environment (MLE) code measurement has passed the launch control policy.

We briefly outline the requirements for installing intel-TXT and then we explain how this technology works.

The requirements for installing Intel-TXT are as follows:

1. *Hardware Requirements:*

The TPM chip must be integrated with the chipset. Both of them work together to check the measurement and security of the system. Therefore, the hardware should support Intel-TXT [49].

2. *Software Requirements:*

Fig 3.6 shows the MLE component that verifies the software to guarantee that all components are secure.

The MLE includes:

- An ACM to perform the measured launch, starting the dynamic chain of trust.
- A Server Platform which includes the BIOS code, BIOS configuration, System Management Mode (SMM) code, option ROM code and configuration, system state, MBR and boot configuration.
- An Initial system software code (referred to as MLE code) that sets up the platform to protect the OS/hypervisor kernel code.

In Fig. 3.6 the dynamic chain of trust (which is carried out by Intel-TXT) starts following a request from the OS via a special processor instruction, which measures and verifies another ACM (the SINIT ACM), which will oversee the secure launch. SINIT is an acronym for Secure Initialization; it initializes the platform so the OS can enter a secure mode of operation. The SINIT ACM performs additional checks, which include making sure the BIOS has passed

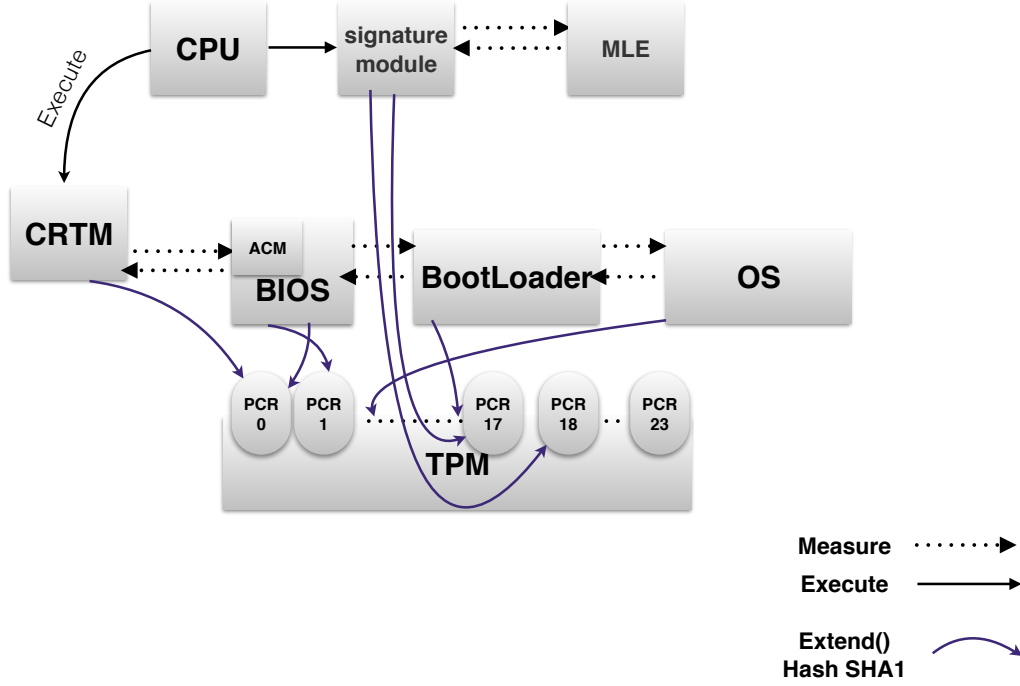


Figure 3.6: The mechanism of Intel-TXT

its security checks and has locked the platform configuration. The ACM then measures the OS (a portion of the OS referred to as the trusted OS) and invokes a launch control policy (LCP) engine which is stored within the TPM NVRAM (NVRAM is an area of flash storage inside TPM). This determines whether the platform configuration and OS can be trusted (as defined by the policy set by the system administrator).

After this, once a secure CPU-contained environment is created, the signature module is validated and its identity is sent to the TPM (PCR-17). Then the signature module measures the Measured Launch Environment (MLE) and sends the result to the TPM (PCR-18).

Enabling Intel- TXT technology in our framework is very important since we aim to provide security for the environment in which the provenance logs are *created* and *stored* at run time. We use this technology to provide a secure environment at run time, and enable awareness of what is going on in the machine at run time.

The features and advantages of using Intel- TXT are listed below.

1. Intel-TXT provides features as follows:
 - (a) Secure measurement
 - (b) Dynamic launch mechanisms via special instructions
 - (c) Configuration locking
 - (d) Sealed secrets
2. Intel-TXT helps detect and/or prevent software attacks such as:
 - (a) Attempts to insert nontrusted VMM (rootkit hypervisor)
 - (b) Reset attacks designed to compromise secrets in memory
 - (c) BIOS and firmware update attacks

Intel-TXT uses enhanced processor architecture, special hardware, and associated firmware that enables certain Intel processors to reduce the overheads associated with system virtualization and allow guest OSs and applications to run in their intended modes.

Trusted Boot (TBoot) is an open source, prekernel/VMM module that uses Intel-TXT to perform a measured and verified launch of an OS kernel/VMM [3]). TPM NVRAM stores the launch control policy (LCP). The SHA-1 hash of these components is compared with the hash value that has already been shared in NVRAM. If the comparison results match, the OS/hypervisor is running in a secure environment. The policy consists of the platform owner specifying the minimum version of ACM, the platform configuration as measured by (PCR-0 till PCR-7) in the TPM containing known good values, and the MLE measurement, which is a known good value.

3.5 TrustedGRUB

TrustedGRUB is an enhancement of the open-source bootloader GNU GRUB [5]. When BIOS measures the bootloader located in the master boot record, control is transferred to the loader (TrustedGRUB) (shown in Fig. 3.2). The

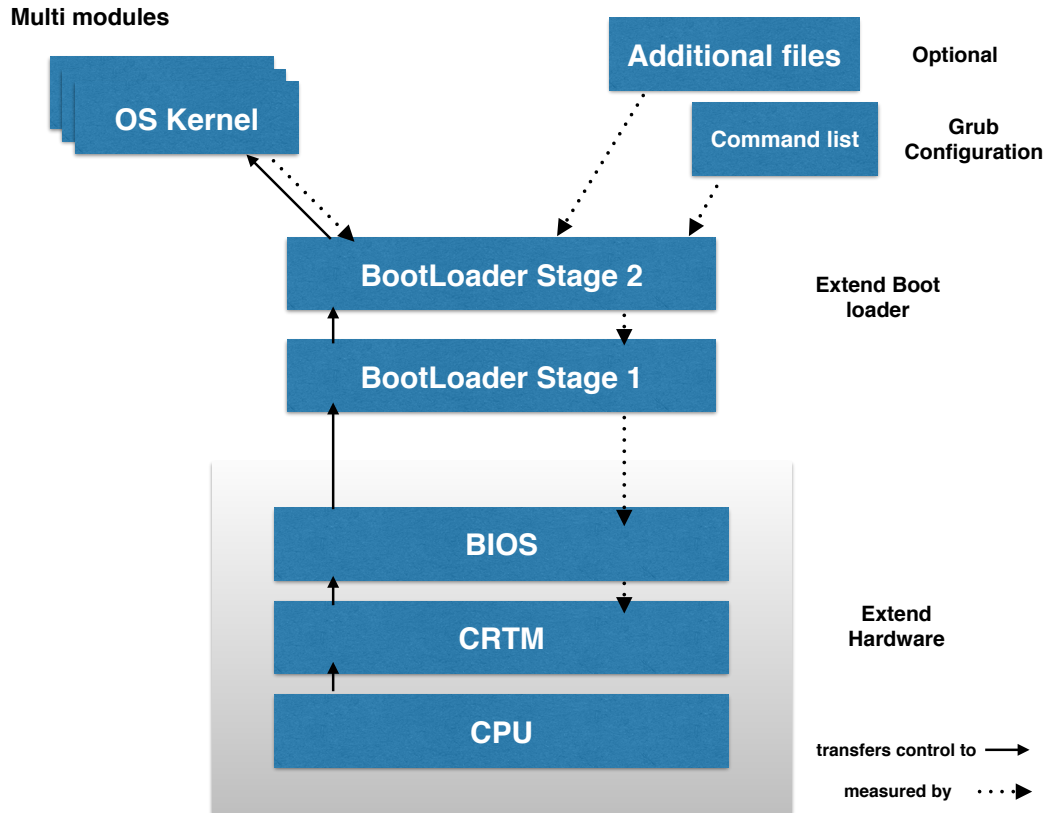


Figure 3.7: Chain of Trust, extended Boot Loader is in use (TrustedGRUB)

chain of trust is carried on by TrustedGRUB by measuring the integrity of the OS configured to load and extending the result into PCRs (i.e., PCR-8 to PCR-14). The hash values of the PCRs provide the evidence to attest system status at boot time. Fig 3.7 shows how the bootloader is measured and how the bootloader receives and passes control to finally achieve a chain of trust.

PCR-8 and PCR-9 contain information about the bootloader (TrustedGRUB). If an attacker tries to access the machine by changing the boot of the machine (e.g., boot attack[31]) then it is easy for machines that contain TPM to detect this kind of attack. Since the value of the current bootloader is secured in PCR-8 toPCR-14, changing the bootloader will change the hash values inside PCR-8 - PCR-14.

Furthermore, TrustedGRUB offers an important feature to enable verification of the integrity of an arbitrary file after the OS is loaded. With the help of this functionality, users can continue the chain of trust with the necessary

component (i.e., the OS) to enable integrity checking. This functionality is realised by providing a "check file" option, where TrustedGRUB will load and verify given files by comparing the SHA1- results with pre-calculated values stored in the check file. The integrity of all files listed in this check file is verified during the boot process by comparing the referenced hash values to the newly computed values. If some of these do not match, a warning is displayed. All check files verified are extended into PCR-13.

In addition, TPM encrypts the data using TPM keys. This encryption can also depend on certain PCR values. If the value of PCR is changed for any reason, there is then no way to decrypt the data [5]. (we will further discuss this in chapter 5).

Chapter 4

Framework Design

In this chapter, our design framework is discussed. The design focuses on the preservation of the confidentiality and integrity of data provenance, and on the tools that we use to generate these provenances. In our framework we also aim to ensure easy availability of the provenance logs to the system administrator at any time.

In the following sections we will explain our framework's design and the components that we used in constructing this framework.

4.1 Client Side

1. *Client* : In Fig. 4.1, the client machine could be a virtual or a physical machine. This client machine collects provenance logs and is thus the source of the provenance logs. Therefore, it is very important to provide a trusted environment to guarantee that the provenance logs *at-creation*, or *at-rest* are in a trusted environment. To achieve this, client machines must have a TPM chip. This chip must be enabled and active in order to work properly. As we mentioned in chapter 3, the TPM chip measures the machine components from boot time until the OS is up and running in the SRTM stages. The TPM then keeps the hash values for these components at SRTM stages inside the appropriate PCR in the TPM chip. Then the administrator can remotely read these values us-

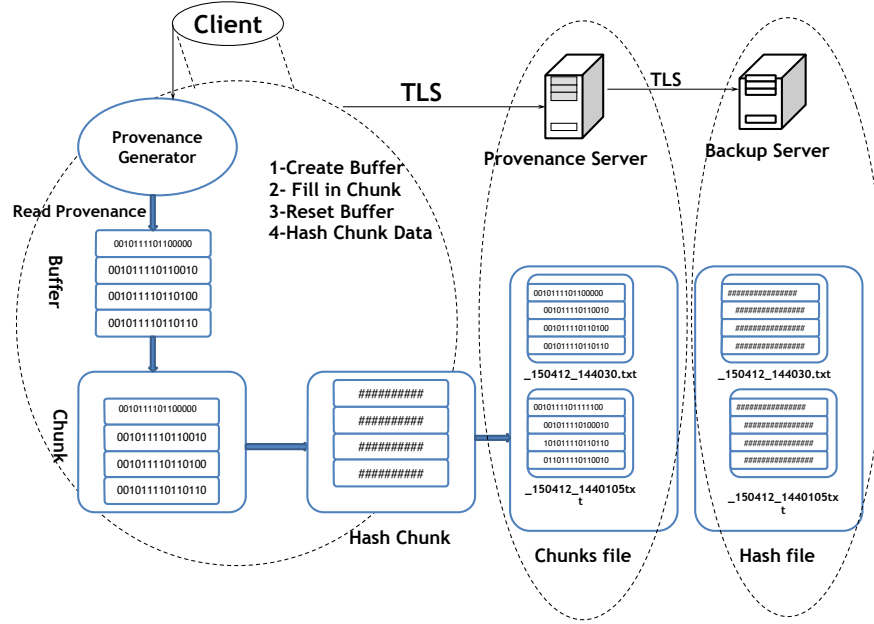


Figure 4.1: Framework Design

ing a special protocol for security reasons (this will be further discussed later in the chapter). The administrator is thus able to check the client system status frequently (e.g., to detect if an attacker has changed the bootloader, or for the presence of a malicious user).

2. *Provenance Generator* : In Fig. 4.1 the provenance generator is a kernel-space provenance logging tool. The reason for using a provenance generator is that this tool can record all kernel events; based on these provenance logs we can tell what operations have been executed in the machine. This tool must be compatible with virtual and physical machines and must be able to allow all cloud stakeholders to trace their data. The provenance generator should also collect provenance from the lowest possible atomic data actions. In addition, this tool must be accurate and have granular timestamp synchronisation across several machines. Finally, we provide integrity for this tool by extending (hashing the provenance generator code) and storing this hash value in PCR-23

which is allocated for this purpose. We can thus guarantee that any potential tampering with this tool code can be detected by comparing the new hash value of the code with the one that is stored in PCR-23 inside the TPM. The mechanism that the administrator applies to remotely check the hash value of PCR-23 for the client machine will be further discussed in chapter 5.

3. *Buffer* : The buffer is created in the client machine. This buffer temporarily receives provenance logs from the provenance generator. We reduce the probability of attack and can detect any tampering that may occur in this file. Our software checks the time and the size of the buffer. If the time limit is exceeded, or the buffer becomes full, then the buffer content will fill the chunk (the provenance log file).
4. *Chunk* : The chunk is created once the buffer becomes full or the time is exceeded. The chunk is then moved into storage. The data inside the chunk are the provenance logs, which were collected by the provenance generator (Progger). Storing the provenance logs inside the chunk file based on the time and the size of the buffer will help the administrator to check specific provenance logs at specific times, where the creation date and time of the chunk will be the name of the chunk and of the hash chunk file.
5. *TPM* : The TPM is a chip required to provide integrity, confidentiality and reliability for the data provenance logged. The hash value of the program of the provenance generator (e.g., Progger) is stored inside a PCR-23 to detect any change that may happen in the code. After this, IMA is used to detect both expected and unexpected events in the provenance log file. TPM stores the hash values of measurements for the CRTM, BIOS, bootloader, OS kernel, and OS. During runtime, Intel-TXT and IMA can detect attacks at the DRTM phase. We assume that the storage used to store data provenance applies the Opal TCG

technology to enable full disk encryption [2]. Intel-TXT can be configured to only allow the OS to launch if it knows the platform and system software are secure and trusted (as defined by the data centre’s policy). LCP guarantees that OS/Hypervisor runs if and only if the current policy matches with its (LCP) policy which consists of specific values of PCRs. This secure launch control policy allows the software to operate in a trusted OS, but only after validating that the platform configuration and system software meet the system administrator policy [65].

4.2 Provenance and Backup Server

In this section we will explain our design for Provenance and Backup servers.

- *Provenance server:* This server is used by the administrator to monitor the system status in client machines and backup servers. This server receives all data provenance (provenance logs) collected by client machines. In addition, the provenance logs consolidated in the provenance servers can be examined by the administrator in the short term. Because the files rapidly become very large, they are subsequently moved to the backup server. Thus, the idea behind using a provenance server (which is used by the administrator) in our design is to give the administrator the ability to frequently check the system status of client machines (PCRs); the administrator can detect if any tampering has happened in the client machines. Also, by temporarily storing the provenance logs it is possible to check the confidentiality of these provenance logs. This server will be a verifier server; the administrator can remotely check system status (check specific PCRs in client machines or backup servers). This feature is called remote attestation. In this system, we use a special protocol design from TCG to ensure that the connection between machines is secure. This server runs in a secure environment, as does the client server. (This server has a TPM, to allow the administrator to frequently check

the status of his provenance server.

- *Backup Server* : In the longer term, the backup server works as an archive server for the provenance logs when the number of provenance logs becomes too large to be stored in the provenance server. The large numbers of provenance logs in the backup server can also be used for data analysis purposes. The provenance logs in the backup server are sealed by a TPM sealed feature (i.e. they are encrypted). This feature adds confidentiality to our framework, since the key which is used to encrypt provenance logs is kept in a safe place inside the TPM. This key never leaves the TPM chip.

Both provenance and backup servers must have a TPM, Intel-TXT, and IMA. The provenance server is accessed by the administrator, who has the ability to examine the data provenance. The administrator must have the TPM-ownership password to gain access to the TPM features. In this framework, groups of provenance logs from a client machines provenance server are grouped in a file called the *chunk*. Every chunk, along with its hashed value, will be transferred via the secure Transport Layer Security (TLS) to the provenance server. The administrator can check the hash values from the client machines and compare them with the hash values stored in the provenance server. We assume that the hash function (used to hash chunks) applies a keyed-hash message authentication code (HMAC) (e.g., HMAC-SHA1). The secret key used for the HMAC is shared by all the machines with the support of the TPM.

We assume that the connections between clients, provenance and backup server are secure, using a TLS connection. However, as we will see in chapter 5, the administrator who works on the provenance server uses a special protocol (TPM-Quote [6] or TCG IF-M protocol [50]) for remote attestation. For the data provenance (provenance logs) we will explain (in chapter 5 and chapter 7 how we can detect any tampering that may occur in the data provenance,

even if the tampering occurs in the transmission stage.

Algorithm 4.1 Our Software in The Client machine

```

void generator (log)

if (buffer.full() || buffer.timeout()) then
    Chunk chunk = new Chunk(buffer.getData())
    chunkStore.add(chunk)
    hashStore.add(hash)
    buffer.flush()
    chunk.getbackup()
    hash.getbackup()
else
    buffer.add(log)
end if

```

This algorithm 4.1 summarises our framework's software in client machines, provenance and backup servers. In client machines, the provenance generator starts to generate the provenance logs and fills the buffer. If the buffer becomes full or the time limit is exceeded, the provenance is moved to the chunk and the buffer is flushed (reset). Then the chunk is hashed using HMAC. The chunk and hash file are transferred to the provenance server through a TLS connection.

We extended our software (Algorithm 4.1) to utilise an unused/ available PCR, (e.g., PCR-16) to detect any change that occurs due to an attack on the software of the Algorithm 4.1. Hence, any tampering affecting the program will be detected. Meanwhile, the administrator should frequently remotely check PCR-17 on the client machines to make sure that the provenance is created in secure mode. PCR-17 is related to the run-time environment used by Intel-TXT technology.

Chapter 5

Framework Implementation

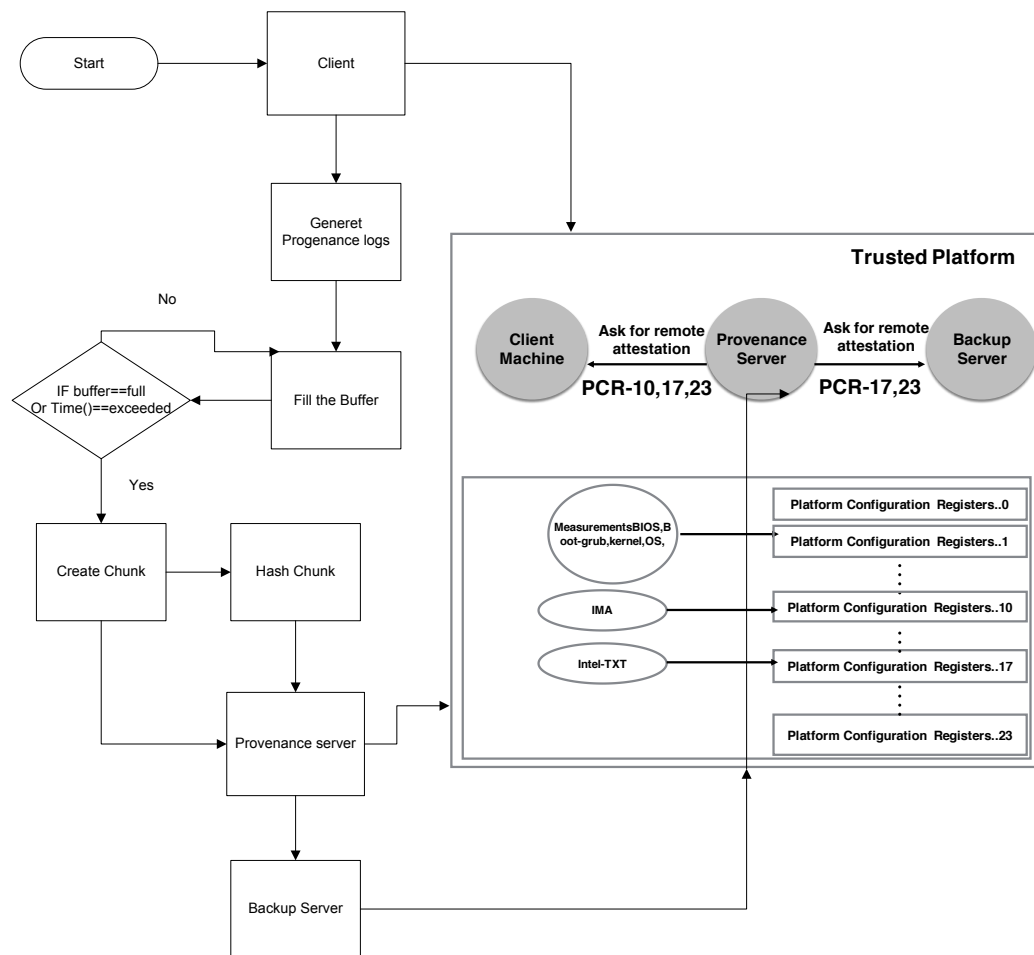


Figure 5.1: Framework Flowchart

In this chapter we will explain the implementation of our framework. We had client virtual machines and two physical servers as the provenance and backup servers. All tools used in preparing this framework were either kernel

modules or open sourced software in addition to the TPM chip, e.g., IMA, TPM-Quote, and Intel-TXT. The components of the framework and how it is based on the TPM are shown in Fig. 5.1. The client machines, provenance and backup server all had a TPM. Each PCR inside a TPM has its own responsibility (e.g., PCR-0 for BIOS).

In our implementation we used CentOS 6.4 with kernel 2.6.33 for all machines. Proggen (a provenance generator) was deployed in the client machines. The machines came with TPM v1.2; the vendor for this chip is ATMEL. Trusted Execution was activated from the BIOS to enable Intel-TXT. Software such as TrouSerS [8], TrouSerS-devel package, and TPM-tools were installed for communication with the TPM chip. TPM-Quote was also installed in the provenance server for remote attestation.

To enable Intel-TXT in a machine, a user must first activate the TPM from BIOS, after which Intel-TXT can be activated from BIOS. In our framework we used kernel 2.6.33 for the physical machines; for virtual machines we used Xen*3.4. Trusted Boot (TBoot) [3] is an open-source bootloader used to launch Intel-TXT, the SINIT command [7] and pre-kernel/hypervisor module that use Intel TXT to perform a measured and verified launch of an OS kernel/hypervisor as discussed in chapter 3.

Fig 5.1 are discussed as follows :

5.1 Client machine

5.1.1 Provenance Generator and Provenance Logs Buffer

The provenance generator is a kernel-space provenance logging tool. In our case, the provenance generator was Proggen (a cloud-based provenance logger). Table 5.1 shows examples of Proggen's log format. The first row shows Proggen's log format for an open system call. This system call request by a user generates Proggen's output as a provenance generator, which is; PID,PPID,

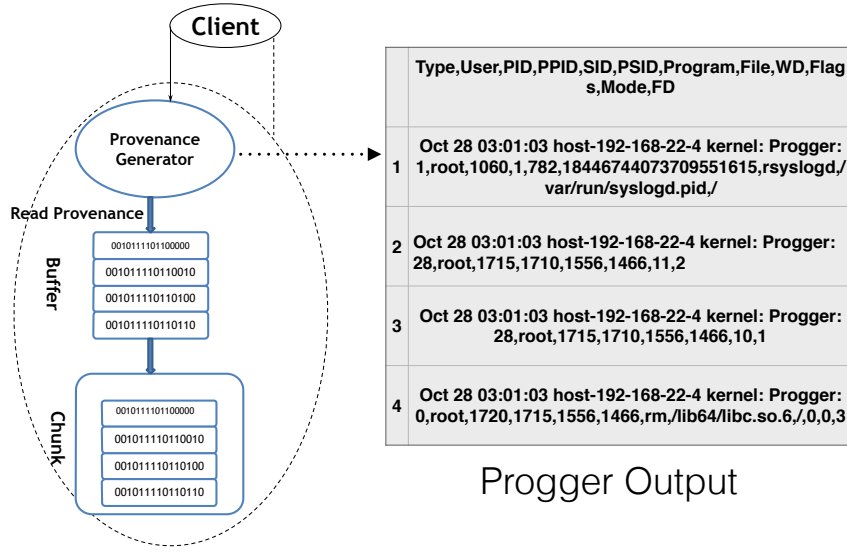


Figure 5.2: Provenance Generator

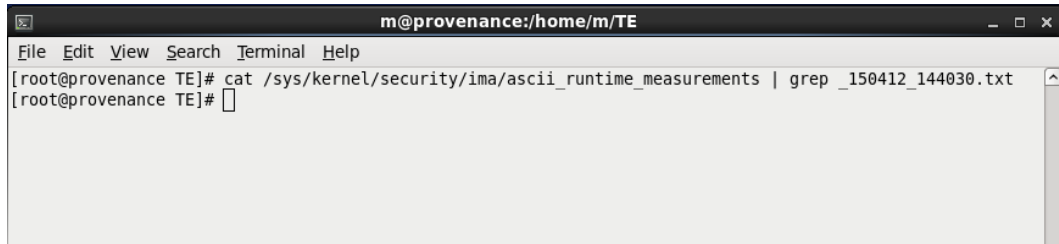
SID,..., etc, as can be seen in the first row in table 5.1. Rows [2-5] in table 5.1, relate to other operations such as close file, read, write and create socket respectively.

Fig.5.2 shows the scenario for collecting provenance logs on the client side. The left side in Fig.5.2 illustrates the components in the client machine which are the provenance generator (in our case, Progger), buffer, and chunk. Progger is a kernel-space logger which potentially empowers all cloud stakeholders to trace their data. The right side of Fig.5.2 is a sample table of Progger's logs. Progger's output contains information (provenance logs) generated by Progger when one of the system calls requests, as shown in the table 5.1. In our framework, Progger writes these provenance logs in the buffer. The logs are maintained by IMA [1](see section 3.3 as will be discussed in section 5.1.2). When the client is operating at runtime, Progger (i.e., the provenance generator) records the provenance logs. Each record in the logs includes an action or event like the records in table 5.1.

Table 5.1: Progger's Log Format

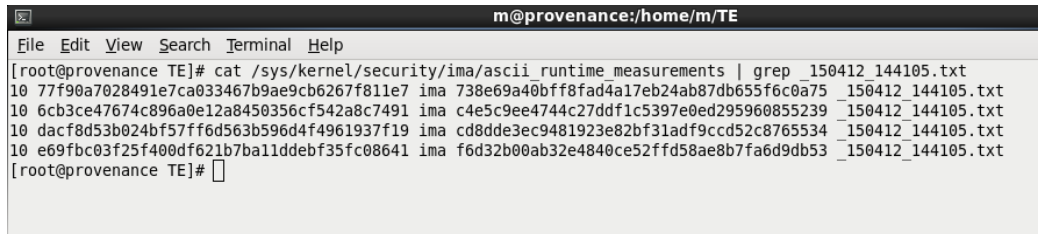
System Call	Progger Log Format
Open	Type,User,PID,PPID,SID,PSID,Program,File,WD,Flags,Mode,FD
Close	Type,User,PID,PPID,SID,PSID,FD
Read	Type,User,PID,PPID,SID,PSID,FD,Offset,HexData
Write	Type,User,PID,PPID,SID,PSID,FD,Offset,HexData
Socket	Type,User,PID,PPID,SID,PSID,Program,SockFD,sType,sProtocol,sFamily

We hash the Progger software code using the extend function to update the old value of the PCR, as illustrated in the equation 3.1, and store the hash value in PCR-23. Hence, any attacks tampering with the Progger code can be detected by checking the hash value of PCR-23. A buffer is prepared for newly generated provenance logs. A time counter is used to evaluate the age of the buffer. Once the buffer is filled with Progger's provenance logs or the time counter exceeds a fixed amount of time (i.e., timeout), the provenance logs inside the buffer are removed to a newly created chunk. After that, the buffer is cleared and the time counter is reset. The idea of using the time counter together with the buffer size is that in some cases the data may not fill the buffer for a long period of time (i.e., no arrival of new provenance logs). By setting a maximum time as well as a maximum fill, the time setting can over-ride the fill requirement and even if the buffer is not full, it will still be extracted to the chunk and the buffer and the timer will reset. Similarly, if the buffer fills before the time limit is reached, the fill setting over-rides the time setting and the content is removed to the chunk even though the time limit has not been reached. The buffer content can also be taken out to a chunk in the event that the client machine is halted at any time (e.g., power disruption). For these reasons, we take the time counter into consideration in addition to the buffer size.



```
m@provenance:/home/m/TE
File Edit View Search Terminal Help
[root@provenance TE]# cat /sys/kernel/security/ima/ascii_runtime_measurements | grep _150412_144030.txt
[root@provenance TE]#
```

Figure 5.3: Measuring the chunk file by IMA



```
m@provenance:/home/m/TE
File Edit View Search Terminal Help
[root@provenance TE]# cat /sys/kernel/security/ima/ascii_runtime_measurements | grep _150412_144105.txt
10 77f90a7028491e7ca033467b9ae9cb6267f811e7 ima 738e69a40bff8fad4a17eb24ab87db655f6c0a75 _150412_144105.txt
10 6cb3ce47674c896a0e12a8450356cf542a8c7491 ima c4e5c9ee4744c27ddf1c5397e0ed295960855239 _150412_144105.txt
10 dacf8d53b024bf57ff6d563b596d4f4961937f19 ima cd8dde3ec9481923e82bf31adf9ccd52c8765534 _150412_144105.txt
10 e69fbc03f25f400df621b7ba11ddeb3f35fc08641 ima f6d32b00ab32e4840ce52ffd58ae8b7fa6d9db53 _150412_144105.txt
[root@provenance TE]#
```

Figure 5.4: Using IMA to detect tampering could happen for the chunk

5.1.2 Create Chunk

Fig.5.2 shows the chunk component. Once the buffer is full or when timeout is reached, a new chunk is created to store the provenance logs retrieved from the buffer. The chunk size is equal to the amount of data retrieved from the buffer. We used the IMA technology to measure the chunk by verifying the PCR-10 (as explained in chapter 3, IMA measures all system-sensitive files - executables, mapped libraries, and files opened for reading by root and stores the hash value of the measurements in PCR-10). IMA can help the administrator to detect operations occurring in a specific sensitive file (i.e., the chunk). For example, in Fig. 5.4 the file 150412-144105.txt is a created chunk file. This chunk file can be detected as a tampered chunk, where in Fig. 5.4 there are four rows after the IMA command; each row means that an operation happened in this file. The number '10' located at the beginning of each row in Fig. 5.4 refers to the PCR-10 that is the default PCR for IMA measurement. We use the command `"cat /sys/kernel/security/ima/ascii-measurements — grep (the file name)"` to check the sensitive file when we want to know if it has been tampered with or not.

The second and fourth columns are hash values. The hash value in the

fourth column is the hash value of the chunk file (i.e., *filedata-hash*). The hash value in the second column is called *template-hash*, and is the result of concatenating the SHA1 hash values of PCR-0 to PCR-7 (called *boot-aggregation*) with the hash value of the chunk file. This template-hash is calculated using the formula

$$Template - Hash = SHA1(filedata-hash || boot-aggregation) \quad (5.1)$$

. This result will be stored in PCR10 with boot-aggregation. The file is named with its creation timestamp and that helps the administrator to find the provenance by specifying the time that the administrator is looking for.

We extended our program (Algorithm 4.1) to utilise an unused/ available PCR, (i.e., PCR-16) to detect any unauthorised changes to this program. Hence, any tampering affecting the program will be detected. The administrator should also frequently remotely check PCR-17 (as explained in 5.4) on client machines to make sure that the provenance was created in secure mode. PCR-17 is related to the run time environment used by Intel-TXT technology, as discussed in section 3.4. When Progger writes provenance logs direct into the buffer, and then transfers these provenance logs to the chunk, this helps the administrator to monitor the chunk file. After the chunk file is created and filled with provenance logs from the buffer, no-one can tamper with or modify this file. If the chunk file in a client machine is tampered with (even by a root user), it will be detected by IMA (e.g., Fig. 5.4, IMA mechanism explained in section 3.3). Transferring data provenance from the buffer to the chunk file allows us to monitor this chunk file. Otherwise, we cannot decide if modification (tampering) happened in Progger (root user) or as a result of a malicious attack. To address that problem we asked Progger to write provenance logs directly into the buffer, and then move these provenance logs into the chunk file.

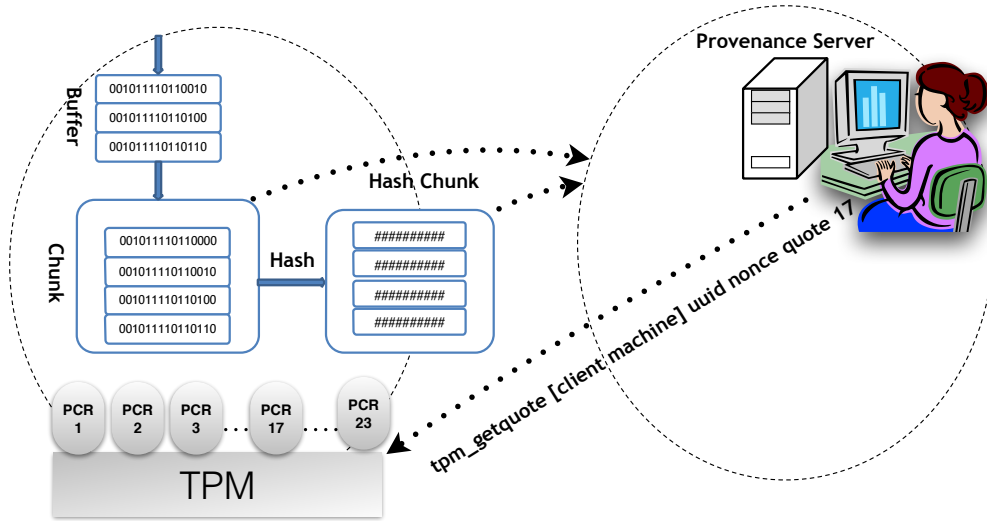


Figure 5.5: The mechanism of hash chunk and how the administrator remotely check the status of client machine

5.1.3 Hash Chunk and Transfer the Chunk to the Provenance Server

Hashing the chunk provides integrity for the provenance, and that helps us to know whether the chunk has been tampered with or not. Fig. 5.5 shows that the chunk file is hashed in the client machine once it is created. Periodically chunks, along with their hash values, are transferred to the provenance server. The administrator, who has access to the provenance server and is already the administrator for the TPM chip can check the PCR-17 (see section 3.4) of client machines remotely using TPM-Quote (explained in section 5.4) or OpenPTS tool to make sure that the chunks are created and transferred in a trusted environment(Secure Mode).

To hash the chunk we use Keyed-Hash Message Authentication Code (HMAC). The key that we use for hashing is shared by the client machines and the provenance server.

Fig. 5.5 shows that the administrator uses the `tpm_quote` command to check the status of the client machine (`tpm_getquote`). This command measures the current value of PCR, and returns this hash value with the nonce

and Universally Unique Identifier (UUID).

5.2 Provenance Server

The aim of this server is to collect provenance from client machines and allow the administrator to examine these provenance logs, then decide if these have been tampered with or not.

Fig.5.6 shows the steps that the administrator carries out to make sure that the provenance was generated in a trusted environment (PCR-17). If the administrator wants to check a specific chunk file, he will hash the appropriate chunk file from the provenance server and compare the hash value with the hash value that is already stored in the provenance server. If the comparison does not match, it provides evidence of tampering. The administrator should check the runtime measurement from the PCR-17 value frequently on the provenance server. This will allow the administrator to detect any attacks on the server (explained in chapter 3.4). Additionally, PCR-23 in the provenance server is allocated for backup software, to make sure that the software is working in a secure environment and that no-one can tamper with the software code. PCR-10 will help the administrator of this server to know what happens to the provenance logs file or hash files using IMA features to measure sensitive files like Fig.5.3 and Fig.5.4.

The hash function (used to hash chunks) applies a keyed-hash message authentication code (HMAC) (e.g., HMAC-SHA1). The secret key used for the HMAC is shared by all the machines with the support of the TPM.

Chunks from the provenance server are then backed up in the backup server. The provenance server receives data provenance logs from different clients. These logs are used to provide tamper-evidence for the client machines to guarantee that these logs were created and transferred in secure environments (explained in section 3.4). All provenance logs in the provenance and backup servers are encrypted using Opal technology [2], which provides confidentiality

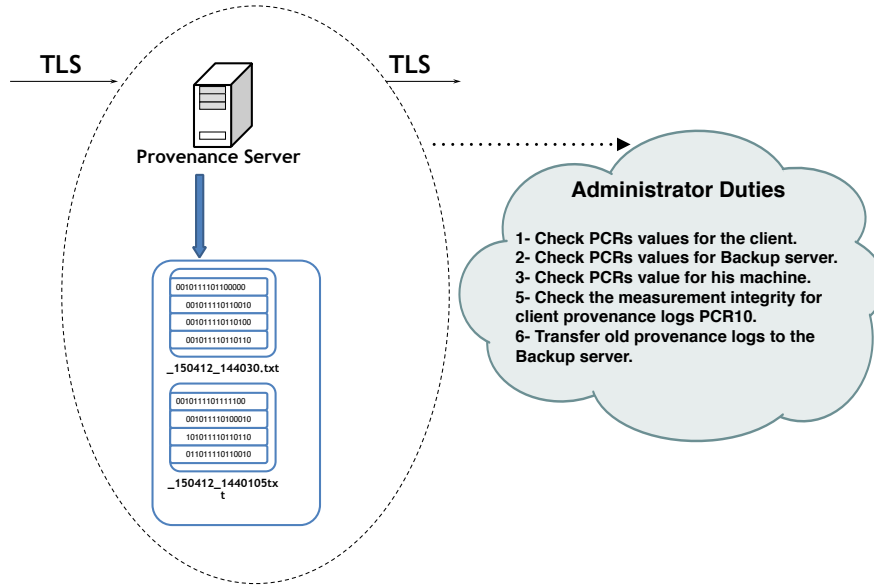


Figure 5.6: The operations on Provenance Server

for the provenance.

When the administrator wants to examine a specific chunk file at a specific time, he can find this file easily, since the file store is based on the time that it was created. Then he hashes this file and compares the result of the hash file with the hash from the original client, to detect whether tampering has occurred in this file.

Tboot (Trust boot) was used in our framework. The LCP checks the appropriate MLE and SINIT module to be loaded. If the MLE policy is not met, an Intel-TXT reset occurs. As a result, a Late Launch will never be executed until the system is restarted. However, the inability to perform a Late Launch does not protect against someone using the system because it does not have the ability to prevent the boot process, and so may allow others to use the system. Though so, it stops users from accessing Locality 2 of the TPM which is responsible to extend PCR 19-22. NVRAM has a limited number of write cycles during the TPM's lifetime, but the use of a symmetric

master key that is only read from NVRAM in the common case can greatly extend its life.

Late Launch does not enforce anything; it simply provides a means by which measurements are taken and stored in the TPM. As a result we enforce different actions. For example, if the goal is to build disk encryption, the OS could be encrypted with a key sealed to specific PCRs (17-22) which match the desired/trusted OS. The important part is if the proper environment has not been loaded (e.g. a malicious user changed the boot loader, and did not trigger a Late Launch) the PCRs values will not match, hence the key to decrypt the OS will be inaccessible (UNSEAL will not work).

The administrator should check the runtime measurement from PCR-17 values frequently on the provenance server. This will allow the administrator to detect any attacks on the server.

PCR-23 in this provenance server we allocated for backup software to ensure that the software is working in a secure environment. PCR-10 will help the administrator of this server to know what happens for data provenance files or hash files using IMA features to measure sensitive files such as chunk or hash chunk files (e.g., Fig. 5.3).

Fig.5.7 shows the mechanism for measuring the machine components and how it is stored in the TPM. In this section we will focus on how Intel-TXT works, what the administrator reads from PCR-17, and how it indicates a secure environment. Briefly, for the SRTM stages, CRTM is executed by the CPU and used to measure the BIOS firmware. Then it will return the result of the measurement (SHA-1 hash) back to the TPM. Then execution passes to the BIOS and so on for all components.

In the DRTM stage, the processor is not measured by the SINIT AC Module. Whenever a secure CPU-contained environment is established, the module signature is validated and communicated to the PCR17 that can measure the Measured Launch Environment (MLE) and transmits this to the PCR 18.

Both provide a means of measuring the running environment, which in-

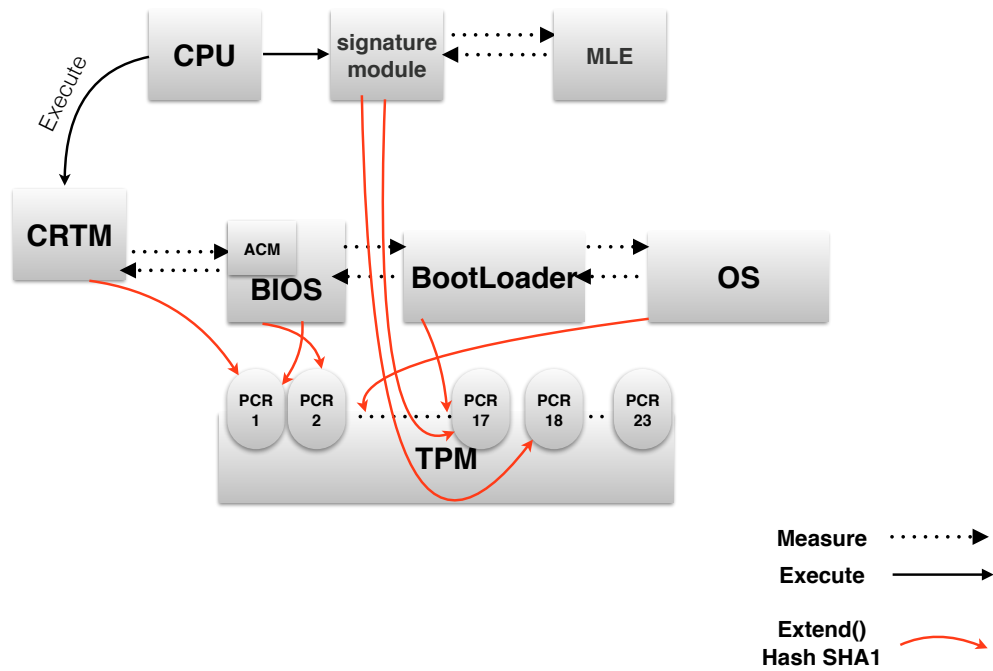


Figure 5.7: Intel-TXT technique and Tboot

volves sending measurements to the TPM PCRs; mainly PCR-0 for SRTM and PCR17 for DRTM.

5.3 Backup Server

Fig.5.8 shows the backup server. The backup server is designed for archiving purposes. Since we cannot keep the logs and provenance logs for any length of time in the administrator server (provenance server), for storage purposes we use a backup server. The backup server has two databases; the first one is used for provenance log files (chunk file) and the second database is used for hash chunk files.

We use the TPM Storage Root Key (SRK) (explained in section 3.2.2) to secure these provenance logs. These keys are stored inside the TPM and never leave it. This sealed (encrypted) data (provenance logs) allows the administrator to protect the data itself by binding the data with PCRs specified (e.g., Fig. 5.8).

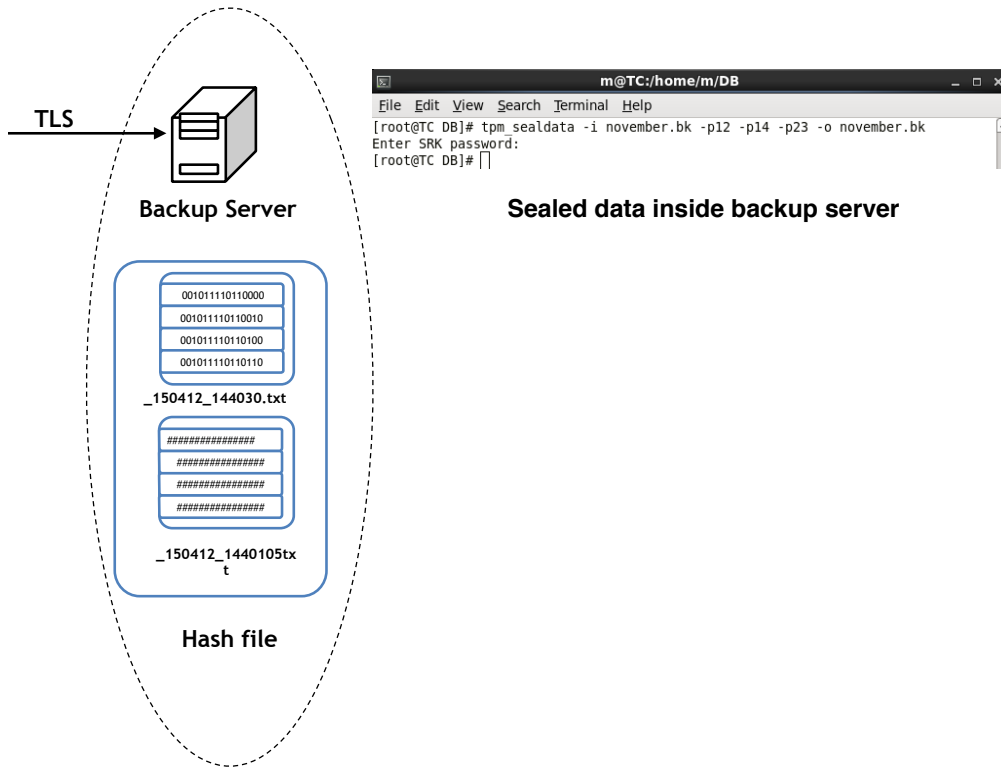


Figure 5.8: Backup Server

TPM Seal outputs a ciphertext, which contains the sealed data and information about the platform configuration required for its release. The TPM includes a random number generator that can be used for key generation. The backup servers have two databases for chunks and their hash values. Hence, when tampering occurs in a chunk or its hash value, the tamper will be detected by comparing the new hash value of the tampered data with the data stored inside the databases.

The idea of using the `TPM_sealdata` command is that TPM has 24 PCRs and at boot time, all PCRs are initialised to known-value (e.g., PCR-0, PCR-1 for BIOS). The only way to change the values of writable PCRs (e.g., PCR-23, PCR-16) is to use the Extend function (invoking TPM operation). When TPM invokes the Extend function, it updates the value of the PCR indicated by index SHA-1 hash of the previous value of PCR concatenated with the data provided (equation 3.1). These PCRs values cannot be changed without invoking TPM operations, since they are shielded inside the TPM

chip. Therefore, the TPM presents a simple interface for binding data to the current platform configuration. The seal command in Fig. 5.8 takes a set of PCRs (we use in Fig.5.8 PCR-[12], PCR-14, PCR-[23]) indices as input, and encrypts the provenance data provided using its Storage Root Key (SRK), a key that never leaves the TPM. The SRK outputs the resulting ciphertext, along with an integrity-protected list of the indices provided and the values of the corresponding PCRs at the time the Seal was invoked. It is also possible to provide the Seal command not only with the PCR indices of interest, but also with the values those PCRs should have before Unseal will decrypt the data. The Unseal command takes in a ciphertext and PCR list created by the Seal command. The TPM verifies the integrity of the list of PCR values, and then compares them against the current values of those PCRs. After this, if they match, the TPM decrypts the file.

Using TPM keys guarantees that the provenance logs which we decrypted using these keys can decrypt if and only if the system is running in a well-known situation (secure mode), which is indicated by the PCRs we chose when we decrypted our data. Because PCR values reflect the running environment, the sealed operation uses those values as keys to encrypt the data.

5.4 TPM-Quote

In this section we will explain the remote attestation technique. Remote attestation checks the status of a remote machine (attester) to determine whether it is running in a secure environment or not. The machine that the administrator uses is called the challenger. The architecture for remote attestation consists of two major components; the integrity measurement architecture (which is the measurement list in the attester and is explained in section 3.3), and the remote attestation protocol. We will use the remote attestation protocol recommended by IBM [45].

We use TPM-Quote [6] for remote attestation. TPM-Quote is a collecting

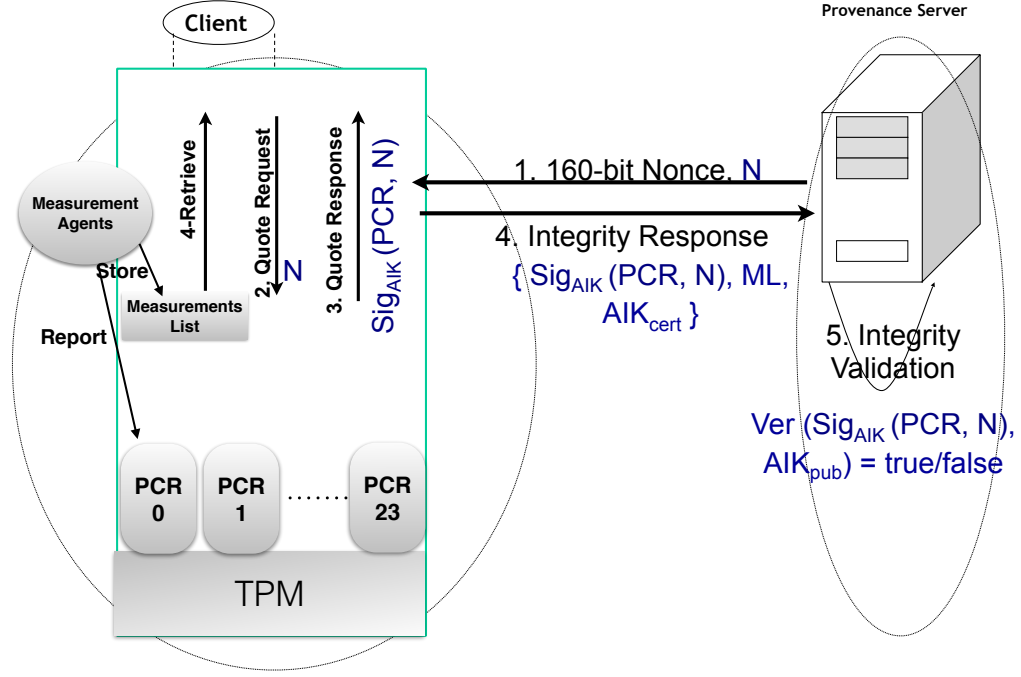


Figure 5.9: Remote Attestation Using TPM-Quote

program that provides support for TPM-based attestation using the TPM quote operation. As we know that the TPM has 24 PCRs, these PCRs contain hash values referring to specific values for specific components in the machine (e.g., PCR-0 for BIOS, PCR-17 for secure environment). Subsequently, any change in this hash value is evidence that tampering has occurred. During provisioning, the hash values of the PCR are compared with the hash value produced by TPM.

In Fig. 5.9 shows the steps of TPM-Quote recommended by IBM [45]. TPM-Quote allows us to use the TPM random number generator to produce a nonce value and an Attestation Identity Key (AIK) including a public and private key pair. The provenance server sends the nonce value and AIK to a remote machine (e.g., the backup server and client machine to provide authenticity between them). Then, the remote machine uses its private key to sign the PCR values which are requested for attestation by the server, and then returns the signed PCR values to the provenance server. The provenance server

verifies the signed PCR values using the public key of the remote machine. This nonce helps protect the signed PCR values against replay attacks. Using this technique, we achieve authentication between two machines and ensure that the remote machine runs in a secure mode.

In fig 5.9 the challenger requests the measurement list from the attester. This request contains a nonce value. (generated by the challenger TPM) as step 1. After the attester receives the request with the nonce value (step 2, 3) it retrieves the measurement list (Fig 5.9, step 4). In step 5, the challenger validates the integrity of the attester machine by comparing the value it already has with the value that it has just received from the attester, as shown in Fig 5.9.

When the administrator takes TPM ownership he is asked for the SRK key password. Once he enters the password, keys must be generated. AIK is one of these keys, and the public part of the key is used for remote attestation.

Chapter 6

Framework Advantages

In this chapter we will explain some of the principal advantages of our implementation. With the support of TPM, our framework can provide *admissible, complete, authentic, reliable and believable* aspects of tamper-evidence for data provenance.

6.1 Guarantee that Data Provenance is created and transmitted in a secure environment

The values of PCRs can provide static and dynamic roots of trust. Provenance logs are created, stored and transferred between machines, and if any changes occur in the provenance logs or to the machine components (e.g., bootloader and OS kernel), the values of the PCRs will change and can be provided as tamper-evidence for the data provenance. Since these PCRs cannot be accessed and tampered with by unauthorised users, the provenance logs, along with the tamper-evidence can be guaranteed to be admissible, complete, authentic, reliable and believable evidence, usable in a court of law. The following subsections explain how we provide tamper-evidence for each framework component:

6.1.1 Tampered Chunk

In this framework, chunks represent collections of provenance logs. Hence, tampered chunks mean tampered provenance logs. The aim of IMA is to detect whether files have been accidentally or maliciously altered, whether in remote or local machines. Where IMA maintains a runtime measurement list and stores hash values inside TPM (PCR-10)s, the measurement list cannot be compromised by any software attack. Tampering in chunk files can be detected by the IMA module through PCR-10.

6.1.2 Tampered Provenance Generator

The program of the provenance generator (in our framework this is Progger) may also be tampered with, but this tampering can be detected. For example, an attacker changes the Progger code and places a modified Progger program in a client machine. However, the hash value of the original Progger is stored in PCR-23. Hence, the hash value of the new Progger program is different from the value inside PCR-23 and the administrator can remotely attest this change.

6.1.3 Trusted BIOS Configuration

An attacker can access the BIOS of a client machine (e.g., by using a compromised BIOS password), and then change the BIOS configuration or update the BIOS firmware. Any changes in the BIOS can be detected by PCR-0 and PCR-1. For example, if the BIOS firmware is updated, this change in the BIOS version can be detected.

6.1.4 Tampered Bootloader

An attacker may launch boot live attacks e.g., an evil maid attack [53]. Any bootloader that is not the same as the legitimate bootloader tracked by PCR-4, PCR-5, and PCR-8 will be detected by the TPM chip.

6.1.5 Change Kernel OS

If the OS kernel is changed by an attacker or updated by a new compromised kernel, any changes incurred to the OS kernel can be detected by PCR-20.

6.2 Data Provenance with Integrity is Guaranteed

In our framework, all chunks of provenance logs are hashed using HMAC on the client machines. Any changes occurring to a chunk during the data at transmission or at rest can be detected by comparing the hash value of the tampered chunk with the original hash value using the HMAC. With this technique, the integrity of the data provenance can be guaranteed.

6.3 Data Provenance with Confidentiality is Guaranteed

The provenance and backup servers apply TCG Opal technology to provide full disk encryption so that confidentiality of the provenance logs can be achieved. For the backup server, the TPM-sealed feature using the TPM SRK key is applied to encrypt the provenance logs. The private parts of the TPM keys never leave the TPM chip; this provides enhanced confidentiality for these keys and as a result, for the provenance logs.

Moreover, the confidentiality of communications between the provenance server, backup server, and client machines is guaranteed by TLS connections. This confidentiality preserves the privacy of the data provenance.

6.4 Data Provenance with Availability is Guaranteed

Originally, the provenance logs are generated and stored in the client machines. These logs may be destroyed; for example, a virtual machine stores provenance logs, and then the virtual machine is terminated so the logs will no longer be available. With the proposed framework, the availability of the provenance logs can be guaranteed by storing the logs in the provenance and backup servers for short-term and long-term usage, respectively.

6.5 Remote Attestation

One of the most important features of our framework is that we use the TPM to provide remote attestation. To the best of our knowledge, there is no other technology that can provide integrity and confidentiality for the evidence and remote attestation at the same time. We use a secure method to prevent Man-In-The-Middle attacks (MITMA), as discussed in section 5.4, and in the event that an attack does occur, to ensure that it will be detected.

Remote attestation allows the administrator to read the PCR values remotely. Each PCR denotes a specific component in the machine (e.g., PCR-0 for BIOS), so any change in the expected value of a specific PCR will be an indicator that tampering has occurred in this component. In this way we can check the system status in any stage, whether *at run-time* or *at boot-time*.

Chapter 7

Evaluation

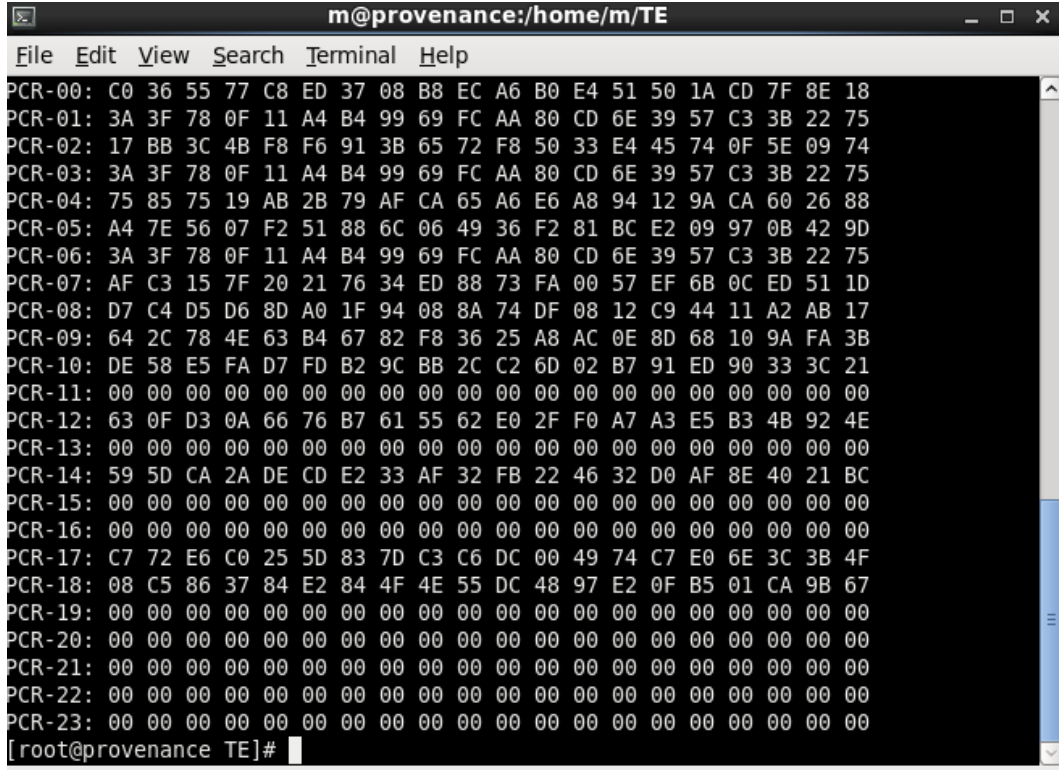
In this chapter we will discuss some experiments that we conducted to detect tampering in the machine at boot time and run time. As discussed earlier, our framework is based on TPM and Intel-TXT technology. We also used IMA to detect tampering in sensitive files, especially those that contain provenance logs. We will present the results mentioned in chapter 6.

Following this, we will evaluate our framework, by comparing it with other solutions that provide tamper-evidence.

7.1 Detecting tampering in machine components

1. Detecting Tampering in BIOS

The hash values that are stored in PCR[0] to PCR[7] by measuring the boot stage (Fig. 1.1) components (SRTM) can tell us if tampering has occurred in these components. These components are what the system should start runs with. Fig. 7.1 shows 24 TPM PCRs. These PCRs are hash values; each one contains 160 bits. The PCRs from [0] to [7] are for the SRTM as was explained in detail in chapter 3. When the hardware components from the CRTM and BIOS are measured and hashed, the values are stored inside these PCRs. Any changes in these



```

m@provenance:/home/m/TE
File Edit View Search Terminal Help
PCR-00: C0 36 55 77 C8 ED 37 08 B8 EC A6 B0 E4 51 50 1A CD 7F 8E 18
PCR-01: 3A 3F 78 0F 11 A4 B4 99 69 FC AA 80 CD 6E 39 57 C3 3B 22 75
PCR-02: 17 BB 3C 4B F8 F6 91 3B 65 72 F8 50 33 E4 45 74 0F 5E 09 74
PCR-03: 3A 3F 78 0F 11 A4 B4 99 69 FC AA 80 CD 6E 39 57 C3 3B 22 75
PCR-04: 75 85 75 19 AB 2B 79 AF CA 65 A6 E6 A8 94 12 9A CA 60 26 88
PCR-05: A4 7E 56 07 F2 51 88 6C 06 49 36 F2 81 BC E2 09 97 0B 42 9D
PCR-06: 3A 3F 78 0F 11 A4 B4 99 69 FC AA 80 CD 6E 39 57 C3 3B 22 75
PCR-07: AF C3 15 7F 20 21 76 34 ED 88 73 FA 00 57 EF 6B 0C ED 51 1D
PCR-08: D7 C4 D5 D6 8D A0 1F 94 08 8A 74 DF 08 12 C9 44 11 A2 AB 17
PCR-09: 64 2C 78 4E 63 B4 67 82 F8 36 25 A8 AC 0E 8D 68 10 9A FA 3B
PCR-10: DE 58 E5 FA D7 FD B2 9C BB 2C C2 6D 02 B7 91 ED 90 33 3C 21
PCR-11: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-12: 63 0F D3 0A 66 76 B7 61 55 62 E0 2F F0 A7 A3 E5 B3 4B 92 4E
PCR-13: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-14: 59 5D CA 2A DE CD E2 33 AF 32 FB 22 46 32 D0 AF 8E 40 21 BC
PCR-15: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-16: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-17: C7 72 E6 C0 25 5D 83 7D C3 C6 DC 00 49 74 C7 E0 6E 3C 3B 4F
PCR-18: 08 C5 86 37 84 E2 84 4F 4E 55 DC 48 97 E2 0F B5 01 CA 9B 67
PCR-19: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-21: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-22: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-23: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[root@provenance TE]#

```

Figure 7.1: PCRs in TPM

components will change the hash values inside the PCRs. This will help the administrator (locally or remotely) to detect any tampering in these components. It is very important for the administrator to make sure that the machine components are well-known (this term is defined in section 1.6).

2. Detecting Tampering in the Bootloader

The bootloader is a program that loads the main operating system or runtime environment for the computer after completion of the self-tests [66]. Thus, the bootloader is one of the most important components in the computer system. The measurement value of the bootloader is hashed and stored securely inside the TPM in PCRs[8-14].

In our implementation we used TrustedGRUB, then PCRs[8-14] were allocated inside the TPM for the TrustedGRUB bootloader. If an attack such as an evil maid [53] or live boot attack tries to change the bootloader to access to the system, the TPM will detect this. When the hash value

```

dracut:/# cat /sys/class/misc/tpm0/device/pcrs
PCR-00: C0 36 55 77 C8 ED 37 08 B8 EC A6 B0 E4 51 50 1A CD 7F 8E 18
PCR-01: 3A 3F 78 0F 11 A4 B4 99 69 FC AA 80 CD 6E 39 57 C3 3B 22 75
PCR-02: 17 BB 3C 4B F8 F6 91 3B 65 72 F8 50 33 E4 45 74 0F 5E 09 74
PCR-03: 3A 3F 78 0F 11 A4 B4 99 69 FC AA 80 CD 6E 39 57 C3 3B 22 75
PCR-04: 7D 80 0C E1 6A B7 18 A2 17 9C 6B B5 06 14 12 DF 5A A1 94 49
PCR-05: BA 48 8F D8 69 8E 47 2F CF C6 72 D9 4D 66 CB F2 49 EE AA 7D
PCR-06: 3A 3F 78 0F 11 A4 B4 99 69 FC AA 80 CD 6E 39 57 C3 3B 22 75
PCR-07: AF C3 15 7F 20 21 76 34 ED 88 73 FA 00 57 EF 6B 0C ED 51 1D
PCR-08: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-09: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-10: 0B 97 4A 3E C7 FA B3 6D 75 1B 1F 88 A0 27 4B 57 55 29 7B 2C
PCR-11: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-12: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-13: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-14: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-15: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-16: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-17: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-18: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-19: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-20: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-21: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-22: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-23: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
dracut:/#
dracut:/#

```

Figure 7.2: Detect BootLoader Tampered

of the bootloader is stored in PCRs[8-14], any change in the bootloader will change the values inside PCRs[8-14], as we can see in Figure 7.2. Fig. 7.2 shows the PCRs inside the TPM; as we know, each PCR contains a 160-bit hash value and the PCRs from [8-14] for the TrustedGRUB bootloader. If an attacker tries to change the bootloader for the machine this will cause the values of the PCRs to change and this will help the administrator to detect the attack.

3. Detecting tampering in the OS kernel or the OS

In the same way that we can detect changes in the BIOS or the bootloader, we can detect changes due to attack in the OS or in the kernel.

7.2 Detecting Tampering at runtime

Fig. 7.3 shows the mechanism for measuring machine components from boot time till run time, In our framework we use the Intel-TXT feature to detect

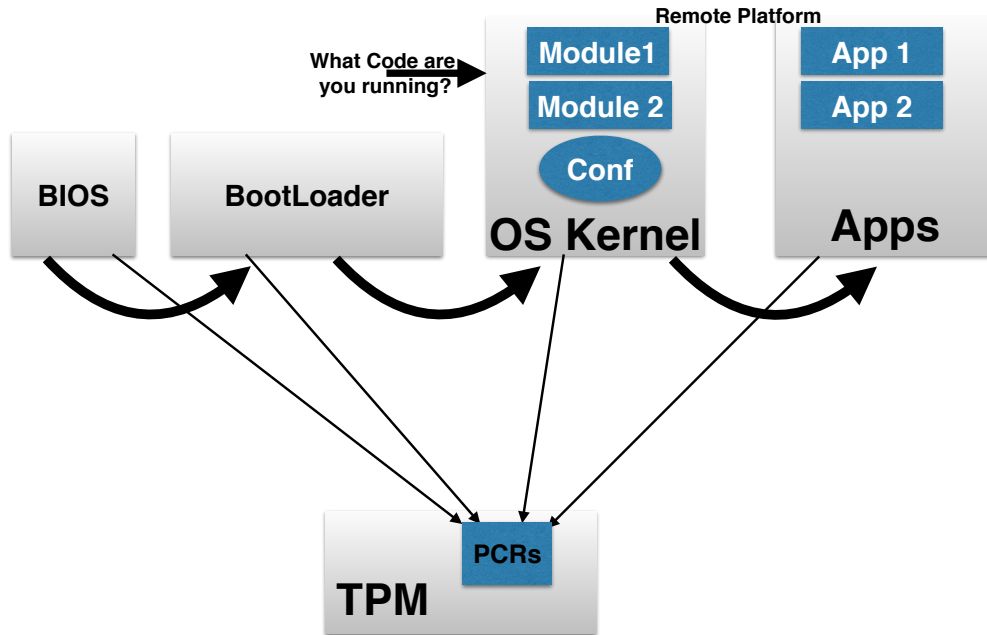


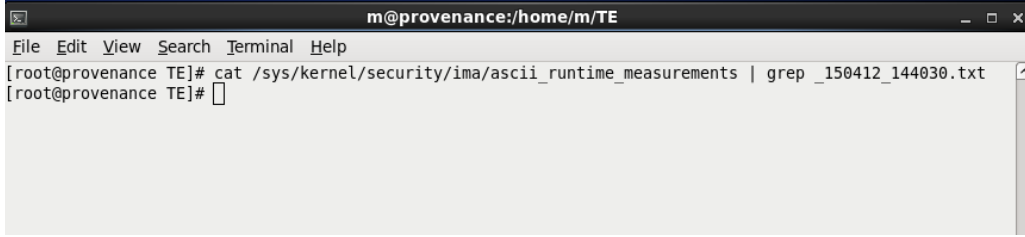
Figure 7.3: TPM keep the values of SRTM and DRTM inside PCRs

whether any attacks occur during runtime.

1. Detecting a Tampered Provenance Generator:

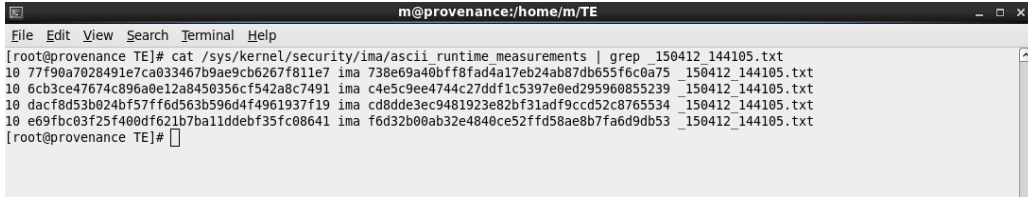
The provenance generator is very important since it is the tool that generates provenance logs. We need to provide integrity for this tool. Therefore, we hash its code and store this hash value inside PCR-23 on the machine this tool works on. In this case, if an attacker tries to change the code of the provenance generator (e.g., Proggen), the administrator can detect this by checking the value of PCR-23 remotely using a remote attestation technique. (See chapter 5).

2. Detecting Tampered Chunk: Proggen's output files are very important when they are the provenance logs. When these logs are to be used as evidence, it is very important to provide confidentiality and integrity for these files (chunk files). Fig. 7.4 shows a file *150412_144030.txt*, which was created on April 12, 2015 at 14:40:30. The administrator can check the events on this file using IMA technology which is used to detect tampering on sensitive files, as explained in section 3.3. The bottom part of Fig. 7.4 shows that file *150412_144105.txt*, was tampered with



```
m@provenance:/home/m/TE
File Edit View Search Terminal Help
[root@provenance TE]# cat /sys/kernel/security/ima/ascii_runtime_measurements | grep _150412_144030.txt
[root@provenance TE]#
```

File Created by Progger Without tampering



```
m@provenance:/home/m/TE
File Edit View Search Terminal Help
[root@provenance TE]# cat /sys/kernel/security/ima/ascii_runtime_measurements | grep _150412_144105.txt
10 77f90a7028491e7ca033467b9ae9cb6267f811e7 ima 738e69a40bff8fad4a17eb24ab87db655f6c0a75 _150412_144105.txt
10 6cb3ce47674c896a0e12a8450356cf542a8c7491 ima c4e5c9ee4744c27ddf1c5397e0ed295960855239 _150412_144105.txt
10 dacf8d53b024bf57ff6d563b596d4f4961937f19 ima cd8dde3ec9481923e82bf31adf9ccd52c8765534 _150412_144105.txt
10 e69fbc03f25f400df621b7ba1ddeb3f35fc08641 ima f6d32b00ab32e4840ce52ffd58ae8b7fa6d9db53 _150412_144105.txt
[root@provenance TE]#
```

After the file Created it Tampered

Figure 7.4: Comparison Between Files Created By Progger

by someone. The hash values in the fourth column are the hash values of the file (*150412_144105.txt*), concatenating with the hash values of PCR[0-7] (*template hash* which are in the second column on the left). The hash concatenations stored in PCR-10 will help to detect tampering, whether locally or remotely, by the administrator using TPM-Quote (see TPM-Quote in section 5.4).

7.3 Results

In this section we will evaluate our framework against other solutions. We will compare the results that are presented by these solutions with our framework.

Table 7.1 shows the comparison between different solutions that provide tamper-evidence or partial tamper-evidence, as defined in section 1.6. We evaluate these solutions against our framework based on very important factors. The solution should be applicable in physical and virtual machines. We also evaluate our solution against others based on availability, and finally the evaluation is based on solutions that provide full tamper-evidence or partial

tamper-evidence.

The first mechanism evaluated with our framework was Progger [40]. Using Progger to provide full tamper-evidence is not adequate. Progger generates provenance logs and provides integrity for these logs. But we need a mechanism that guarantees this provenance *at-creation* and *at-storage* in a trusted environment. In addition, the Progger code must be protected to guarantee that no malicious users can tamper with it. Finally, the mechanism must have the ability to measure the system at boot process and that cannot be done by Progger.

The second mechanism is Forensix [26]. The Forensix tool generates data logs and collects them in database servers. After this, users can submit SQL queries to retrieve the events that occurred in the machine (e.g., PID, start-time, end-time). This mechanism provides availability of provenance logs. However, its authors do not mention any technology which provides secure, integrity-preserving environments for the database server. This potentially makes it easy for malicious users to mask their traces by modifying the logs in the database server or within the server where the provenance logs are generated.

The third mechanism is [15]. This mechanism can trap and log activities from system calls in a log file, and then hide the log from the file system. The administrator can unload the module from the system to use this log for tamper-evidence. This mechanism cannot provide full tamper-evidence since it is only measuring system calls. In addition, this mechanism cannot measure the boot process stages, so we do not know what is happening in the system at boot time.

The fourth mechanism we chose in this evaluation is the Provenance-Aware Storage System (PASS) [47]. PASS is a storage system that automatically collects, stores, manages, and provides searches for provenance. This protocol does not provide any kind of trust for the environment in which the provenance was collected. The capture mechanism consists of a set of Linux kernel mod-

ules that transparently record provenance; it does not require any changes to computational tasks. Users can pose provenance queries using nq (new query), a proprietary tool that supports recursive searches over the provenance graph. PASS's capture mechanism often leads to very large volumes of provenance information; another limitation of this approach is that it is restricted to local file systems [25].

The fifth mechanism provides tamper-evidence using the TPM [58]. The authors focus on using the TPM during the boot process, but do not mention other technologies that are used to provide integrity and confidentiality for provenance logs at the run time stage(i.e., IMA, Intel-TXT). As mentioned in chapter 3, the TPM can provide integrity and confidentiality for provenance logs from boot time until the OS takes over. After that we need technology that is compatible with the TPM to provide integrity and confidentiality for provenance logs at run time. The authors also do not provide availability for provenance logs.

Finally, our framework presents a solution that provides integrity and confidentiality for provenance logs at boot time and run time using a TPM. We also provide availability of the provenance logs that were collected by storing these provenance logs in a backup server for archiving purposes. In addition to this we use Intel-TXT technology to provide secure environments on client machines where the provenance is created, as well as for the provenance and backup servers.

If we compare our framework, which is based on a TPM chip in addition to Intel-TXT and IMA technologies, with solution five in table 7.1, it can be seen that using the TPM chip alone is not enough to provide full tamper-evidence. The TPM can measure the components at boot process stages and keep the hash values in a secure register (PCR). That is a very good solution in terms of knowing exactly what has happened in these stages, but it cannot detect events that may occur at run time. Hence, to provide full tamper-evidence for provenance logs, we should be aware of what has happened in the boot

Table 7.1: Evaluation

	Virtual machine	Physical Machine	Availa- bility	Partial Tamper-evidence	Full Tamper-evidence
Progger [40]	✓	✓	✗	✓	✗
Forensix [26]	✗	✓	✓	✓	✗
vfs logger [15]	✗	✓	✗	✓	✗
PASS [47]	✓	✗	✗	✓	✗
TPM [58]	✓	✓	✗	✓	✗
Our framework	✓	✓	✓	✓	✓

process stages and at run time to guarantee that the provenance logs were created and stored in a secure environment. That can be achieved if and only if we use technology that detects the code that is currently running, such as the Intel-TXT technology (explained in chapter 3 and chapter 5). We also use a special backup server to archive the provenance logs and encrypt these provenance logs using TPM keys (i.e SRK keys).

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis aimed to propose a framework that provides tamper-evidence for logs, especially data provenance logs. We focused on the environments in which the logs are *created* and *stored*. We also focused on the provenance logs status from boot time until run-time in order to be aware of what was going on in the machine, and whether these provenance logs were *created* and *stored* in a secure environment or not. Based on this, we tried to find the best technologies that could achieve our aim of creating a solution that would make provenance logs admissible as evidence in a court of law. We found that we needed to integrate some extra technologies with a TPM, such as Intel-TXT and IMA in order to provide (*Reliability, uthenticity, Admissibility, Completeness, and Believability*) for the provenance logs to make them acceptable in a legal enquiry [48].

In this thesis we have discussed past and current tamper-evidence solutions. As we have shown, there are some solutions based on hardware like TPM and others based on software such as hash functions. We investigated the gaps in these existing solutions and found that most cannot provide full tamper-evidence; most of these solutions collect provenance at the application level without knowing what is going on in the machine at the boot stages. In

addition, we found that some of the current tamper-evidence solutions examine provenance logs after they have been collected at application level. As a result, if these solutions cannot provide confidentiality for these logs we cannot accept them, because we cannot guarantee the provenance logs themselves. In contrast, some of these technologies, such as AEGIS, can provide a secure environment during the boot process only, without focusing on run time stages; again, this not enough to provide full tamper-evidence.

We propose a novel tamper-evidence framework for data provenance, and also propose a remote attestation mechanism based on the TPM. Our framework can be applied to cloud computing environments. We also provide much needed tamper-evidence for data provenance and our framework complies with the five major requirements for legal evidence including *admissibility*, *authenticity*, *completeness*, *reliability*, and *believability*. Our Framework focuses on collecting logs and provenance logs that are collected on the machine at different levels from boot time all the way up until run time, and keeping these provenance logs in secure storage using TPM keys.

Our framework focuses on the integrity and confidentiality of provenance logs, but can also be applied to any sensitive files as we saw in chapter 5. We applied the IMA technology to our framework to enable system administrators to check activities on a specific file and read this activity remotely using a remote attestation mechanism (explained in Fig. 3.5).

This framework also assures the integrity, confidentiality, and availability of provenance logs. Confidentiality and integrity of the logs can be provided by using the features of the TPM, including the use of the TPM keys (e.g., SRK); these keys never leave the TPM chip, which provides more confidentiality for our framework. Availability of the provenance logs can be achieved by storing the provenance logs in provenance and backup servers. In the backup server, where the information is used for archiving purposes, we encrypted the provenance logs and the hash values of these provenance logs using the TPM key. These encryption files cannot be decrypted unless the values of the PCRs

which were used at encryption time are the same at decryption time.

We also evaluated our framework against other solutions based on a variety of factors (i.e., VM, PM, Availability, Partial tamper-evidence, Full tamper-evidence). We found that only our framework can provide availability of provenance logs at physical and virtual machines at the same time. Also, based on this evaluation, only our framework can detect tampering occurring in the machine at all stages from boot time all the way up to run time. This is possible because we used Intel-TXT and IMA with the TPM chip to provide integrity and confidentiality for the provenance logs and to allow the administrator the ability to check the confidentiality and integrity of the provenance logs and the system remotely, using a remote attestation mechanism.

8.2 Future Work

Some research directions can be further addressed in future work, as follows:

- *Virtual TPM* – A TPM for virtualisation technologies is challenging since the TPM chip is originally designed for a physical machine. In the physical machine, the TPM chip can be fully used to measure sensitive components inside the machine. However, virtual machine environments have no actual TPM chips and virtual or software-based TPM chips can be easily tampered with.
- *Trusted Cloud Computing* – The rapid adoption and growth of cloud computing has introduced new challenges. One of these challenges arises when cloud computing is used to store data or applications, without users knowing if their data has been tampered with by another user or even if the cloud provider has tampered with the data. Accordingly, cloud computing users want to know that their data is stored securely, or at least in a fashion that if tampering happens, they can detect that. TPM works well with physical machines, but does not scale well with virtualisation [54]. This is because TPM is limited in resources (e.g., 24

PCRs). Therefore, we cannot use a TPM chip to provide a root of trust for large-scale cloud computing, which could have hundreds of virtual machines. Another problem facing us in the attempt to provide roots of trust in cloud computing is the owner of cloud computing (provider). The owner of the data that is deployed in the cloud needs privilege from the cloud provider to get access to some features in the cloud Application Programming Interface (API).

- *Cloud API* – A cloud Application Programming Interface (API) should be provided to help cloud users access the capabilities of the TPM chip. However, this API needs to ensure that cloud users can access the TPM chip even though they usually do not have root access to their subscribed virtual hosts or underlying physical machines.
- *TPM-enabled Private Cloud* – A private cloud environment can be provided by a trusted cloud computing environment using TPM. Open-source cloud software (e.g., OpenStack) can be integrated fully with TPM to provide a trusted environment.
- *Optimisation for Large Scale Environments* – When we implement trusted computing for virtual machines in data centres, there could be a large number of physical and virtual machines, generating a large volume of provenance logs. This large volume of logs being transferred between client machines and the provenance server may be a major bottleneck in the whole system. We would need an optimised system for this scenario.

Chapter 9

List of Publications

1. Mohammad Bany Taha, Sivadon Chaisiri, and Ryan K L Ko. Trusted Tamper-Evident Data Provenance. In IEEE International Symposium on Recent Advances of Trust, Security and Privacy in Computing and Communications held in conjunction with IEEE TrustCom-15, Helsinki, Finland. IEEE, August 2015.

Chapter 10

References

- [1] Linux integrity subsystem. Online [Accessed 29/03/15][linux-ima.sourceforge.net](http://sourceforge.net/projects/linux-ima/).
- [2] Storage work group storage security subsystem class: Opal summary. http://www.trustedcomputinggroup.org/resources/storage_work_group_storage_security_subsystem_class_opal_summary.
- [3] Trusted Boot. Online [Accessed 29/03/15]sourceforge.net/projects/tboot/.
- [4] Which amd npt family 0fh processor supports secure virtual machine mode. Online [Accessed 29/03/15] <http://support.amd.com/en-us/kb-articles/Pages/emb-309-npt.aspx>.
- [5] TrustedGRUB. [Accessed 14/02/15]<http://sourceforge.net/projects/trustedgrub/>, April 2006.
- [6] TPM Quote Tools. Online [Accessed 17/03/15]www.sf.net/projects/tpmquotetools, July 2011.
- [7] Intel® Trusted eXecution Technology. Online [Accessed 29/03/15]<https://software.intel.com/en-us/articles/intel-trusted-execution-technology/>, January 2014.

- [8] TrouSerS. Online [Accessed 29/03/15]<http://sourceforge.net/projects/trousers/files/trousers/>, March 2015.
- [9] Trusted Platform Module. Online [Accessed 29/03/15]http://www.trustedcomputinggroup.org/resources/trusted_platform_module_tpm_summary, 2015.
- [10] Evolution of integrity checking with intel® Trusted eXecution Technology: an intel it perspective. Online [Accessed 02/04/15]<http://www.intel.com/content/dam/doc/white-paper/intel-it-security-trusted-execution-technology-paper.pdf>, August 2010.
- [11] Robert P Abbott, Janet S Chin, James E Donnelley, William L Konigsford, S Tokubo, and Douglas A Webb. Security analysis and enhancements of computer operating systems. Technical report, DTIC Document, 1976.
- [12] Rafael Accorsi. Log data as digital evidence: What secure logging protocols have to offer? In *Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*, volume 2, pages 398–403. IEEE, 2009.
- [13] Rafael Accorsi. Bbox: A distributed secure log architecture. In *Public Key Infrastructures, Services and Applications*, pages 109–124. Springer, 2011.
- [14] Bright Siaw Afriyie. *Concise Ict Fundamentals Volume One*. Trafford Publishing, September 2012.
- [15] Alam Ansari, Arijit Chattopadhyay, and Suvrojit Das. A kernel level vfs logger for building efficient file system intrusion detection system. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 273–279. IEEE, 2010.

- [16] William A Arbaugh, David J Farber, and Jonathan M Smith. A secure and reliable bootstrap architecture. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 65–71. IEEE, 1997.
- [17] Matt Bishop, Michael Dilger, et al. Checking for race conditions in file accesses. *Computing systems*, 2(2):131–152, 1996.
- [18] Matthew Braid. Collecting electronic evidence after a system compromise. *Australian Computer Emergency Response Team*, 2001.
- [19] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Data provenance: Some basic issues. In *FST TCS 2000: Foundations of software technology and theoretical computer science*, pages 87–93. Springer, 2000.
- [20] Harvey M Deitel. *An introduction to operating systems*, volume 3. Addison-Wesley Reading, Massachusetts, 1984.
- [21] Premkumar Devanbu, Michael Gertz, Charles Martel, and Stuart G Stubblebine. Authentic third-party data publication. In *Data and Application Security*, pages 101–112. Springer, 2001.
- [22] Josep Domingo-Ferrer, David Chan, and Anthony Watson. *Smart Card Research and Advanced Applications: IFIP TC8 / WG8.8 Fourth Working Conference on Smart Card Research and Advanced Applications September 20–22, 2000, Bristol, United Kingdom*. Springer, March 2013.
- [23] David Dorwin. Cryptographic features of the trusted platform module.
- [24] Ross Finlayson and David Cheriton. *Log files: an extended file service exploiting write-once storage*, volume 21. ACM, 1987.
- [25] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T Silva. Provenance for computational tasks: A survey. *Computing in Science & Engineering*, 10(3):11–21, 2008.

- [26] Ashvin Goel, W-C Feng, David Maier, and Jonathan Walpole. Forensix: A robust, high-performance reconstruction system. In *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*, pages 155–162. IEEE, 2005.
- [27] Sarah Gordon and Richard Ford. Real world anti-virus product reviews and evaluations—the current state of affairs. In *Proceeding of the 19th NISTNCSC National Information Systems Security Conference Held 22*, volume 2, pages 526–38, November 1996.
- [28] 1620 Users Group. *Report of Systems Objections and Requirements Committee*, volume Appendix F. June 1964.
- [29] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 175–188. ACM, 2007.
- [30] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.
- [31] Adrian Hannah. One key to rule them all: Grub, usb and a multiboot environment. *Linux Journal*, 2011(211):4, 2011.
- [32] Ragib Hasan, Radu Sion, and Marianne Winslett. The case of the fake picasso: Preventing history forgery with secure provenance. In *FAST*, volume 9, pages 1–14, 2009.
- [33] Patrick Hung, editor. *Web Service Composition and New Frameworks in Designing Semantics: Innovations*. IGI Global, 2012.
- [34] IBM. Trusted computing platform alliance announces v.1.0 specifications for trusted computing. Online [Accessed

29/03/15]https://web.archive.org/web/20030719234815/http://www.trustedcomputing.org/docs/tcpa_final.pdf, may 2015.

- [35] Ryan K L Ko. Cloud computing in plain english. In *ACM Crossroads*, 16(3):5–6, 2010.
- [36] Ryan K L Ko. Data accountability in cloud systems. In *Security, Privacy and Trust in Cloud Systems*, pages 211–238. Springer Berlin Heidelberg, 2014.
- [37] Ryan K L Ko, Peter Jagadpramana, and Bu Sung Lee. Flogger: A file-centric logger for monitoring file access and transfers within cloud computing environments. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 765–771. IEEE, 2011.
- [38] Ryan K L Ko, Peter Jagadpramana, Miranda Mowbray, Siani Pearson, Markus Kirchberg, Qianhui Liang, and Bu Sung Lee. Trustcloud: A framework for accountability and trust in cloud computing. In *Services (SERVICES), 2011 IEEE World Congress on*, pages 584–588. IEEE, 2011.
- [39] Ryan K L Ko, Bu Sung Lee, and Siani Pearson. Towards achieving accountability, auditability and trust in cloud computing. In *Advances in Computing and Communications*, pages 432–444. Springer, 2011.
- [40] Ryan K L Ko and Mark A Will. Progger: An efficient, tamper-evident kernel-space logger for cloud data provenance tracking. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 881–889. IEEE, 2014.
- [41] Donald C Latham. Department of defense trusted computer system evaluation criteria. *Department of Defense*, 1986.

- [42] Tang Ling. The study of computer forensics on linux. *International Conference on Computational and Information Sciences (ICCIS)*, pages 294–297, June 2013.
- [43] Peter Macko, Marc Chiarini, Margo Seltzer, and SEAS Harvard. Collecting provenance via the xen hypervisor. *In In Proceedings of 3rd USENIX Workshop on the Theory and Practice of Provenance, TaPP*, 2011.
- [44] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. *arXiv preprint cs/0202005*, 2002.
- [45] Hiroshi Maruyama, Taiga Nakamura, Seiji Munetoh, Yoshiaki Funaki, and Yuhji Yamashita. Linux with ttpa integrity measurement. *IBM Japan, Ltd.(January 28, 2003)*, 2003.
- [46] Microsoft. Bitlocker drive encryption overview. Online [Accessed 19/06/15]<https://technet.microsoft.com/en-us/library/cc732774.aspx>.
- [47] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track*, pages 43–56. Berkeley, CA, USA USENIX Association, 2006.
- [48] Mary A. Nixon. *On Your Own North Carolina Small Claims Court: A Debt Collection Guide for North Carolina Businesses*. Universal Publishers, May 1998.
- [49] Priscilla Oppenheimer. Computer forensics: Seizing a computer. <http://www.priscilla.com/forensics/computerseizure.html>.
- [50] Wyllys Ingersoll Paul Sangster. Pts protocol:binding to tnc if - m. Online [Accessed 29/03/15]http://www.trustedcomputinggroup.org/files/resource_files/C1A987EA-1A4B-B294-D031133E95B20871/IFM_PTS_v1_0_r25_Public%20Review.pdf.

- [51] Kyriacos E Pavlou and Richard T Snodgrass. Forensic analysis of database tampering. *ACM Transactions on Database Systems (TODS)*, 33(4):30, 2008.
- [52] Bart Preneel. The first 30 years of cryptographic hash functions and the nist sha-3 competition. In *Topics in Cryptology-CT-RSA 2010*, pages 1–14. Springer, 2010.
- [53] Joanna Rutkowska. Evil maid goes after truecrypt! Online [Accessed 17/03/15]<http://theinvisiblethings.blogspot.co.nz/2009/10/evil-maid-goes-after-truecrypt.html>.
- [54] Vincent Scarlata, Carlos Rozas, Monty Wiseman, David Grawrock, and Claire Vishik. Tpm virtualization: Building a general framework. In *Trusted Computing*, pages 43–56. Springer, 2008.
- [55] Bruce Schneier and John Kelsey. Automatic event-stream notarization using digital signatures. In *Security Protocols*, pages 155–169. Springer, 1997.
- [56] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, 1999.
- [57] G Shpantzer. Implementing hardware roots of trust: The trusted platform module comes of age. *SANS Analyst Pro-gram*, 40(6):1–15, 2013.
- [58] Arunesh Sinha, Limin Jia, Paul England, and Jacob R Lorch. Continuous tamper-proof logging using tpm 2.0. In *Trust and Trustworthy Computing*, pages 19–36. Springer, 2014.
- [59] SOURCEFORGE. TrouSerS FAQ. Online [Accessed 19/01/15]<http://trousers.sourceforge.net/faq.html>.
- [60] Chun Hui Suen, Ryan K L Ko, Yu Shyang Tan, Peter Jagadpramana, and Bu Sung Lee. S2logger: End-to-end data tracking mechanism for

- cloud data provenance. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 594–602. IEEE, 2013.
- [61] Yu Shyang Tan, Ryan K L Ko, and Geoff Holmes. Security and data accountability in distributed systems: A provenance survey. In *Proceedings of the 15th IEEE International Conference on High Performance Computing and Communications (IEEE HPCC13)*, Zhang JiaJie, China, 2013. IEEE Computer Society.
- [62] PC TCG. Client specific tpm interface specification (tis), version 1.2. trusted computing group, 2003-2013.
- [63] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Advances in Cryptology–EUROCRYPT 2005*, pages 19–35. Springer, 2005.
- [64] Mark N Wegman and J Lawrence Carter. New hash functions and their use in authentication and set equality. volume 22, pages 265–279. Elsevier, 1981.
- [65] James Greene Wiiliam Futral. *Intel Trusted EXecution Technology for Server Platform*. Paul Manning, 2013.
- [66] Yang Xu, Rong-gang Wang, An-yu Cheng, and Rui Li. Design of online upgrade system for the software of vehicle ecu based on can-bus. *International Journal of Advancements in Computing Technology*, 5(1), 2013.
- [67] Aydan R Yumerefendi and Jeffrey S Chase. Strong accountability for network storage. volume 3, page 11. ACM, 2007.
- [68] Jing Zhang, Adriane Chapman, and Kristen Lefevre. Do you know where your data’s been?—tamper-evident database provenance. In *Secure Data Management*, pages 17–32. Springer, 2009.

- [69] Olive Q. Zhang, Markus Kirchberg, Ryan K. L. Ko, and Bu Sung Lee. How to track your data: The case for cloud computing provenance. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 446–453. IEEE, Nov 2011.
- [70] Wenchao Zhou, Ling Ding, Andreas Haeberlen, Zachary G Ives, and Boon Thau Loo. Tap: Time-aware provenance for distributed systems. In *TaPP*, 2011.
- [71] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 295–310. ACM, 2011.

Appendix A

TPM Locality

Locality is an assertion to the TPM that a command's source is associated with a particular component. Locality can be thought of as a hardware based command authorisation [62]. TPM 1.2 supports six levels of locality, locality None and Locality 0-4. PC Client platform usage of locality levels is defined in the PC Client Implementation Specification. TPM 1.2 must support Locality 0-4.

Each PCR, during manufacturing of the TPM, has the locality level set for two types of operations: reset and extends. Even though the Locality 2 is a higher locality. If the PCR wants to allow for both Locality 1 and Locality 2, both bits must be set in the mask. If a command attempts an operation on a PCR, it must be received from the correct locality.

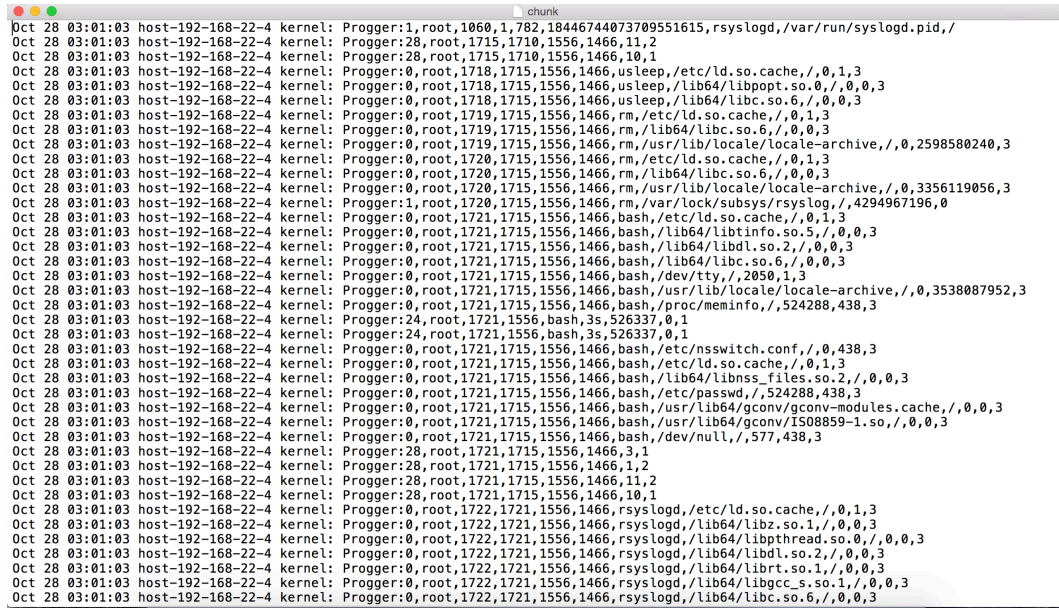
Table A.1: TPM Locality

Locality	Meaning
4	Trusted Hardware/DRTM
3	Software launched by DRTM
2	Controlled by OS/TPM Driver
1	Controlled by OS/TPM Driver
0	SRTM; Default

Table A.2: The Standard Usage of PCRs

PCR	Usage
0	Core BIOS, POST BIOS, Embedded Option ROMs
1	Platform and Motherboard Configuration and Data
2	Option ROM Code
3	Option ROM Configuration and Data
4	IPL Code
5	IPL configuration data
6	State transition (sleep, hibernate, and so on)
7	Reserved for OEM
8	TrustGrub
9	TrustGrub
10	TrustGrub
11	TrustGrub
12	TrustGrub
13	TrustGrub
14	TrustGrub
15	Not Assign
16	Used for debugging
17	Dynamic CRTM
18	Platform defined
19	Used by a trusted operating system
20	Used by a trusted operating system
21	Used by a trusted operating system
22	Used by a trusted operating system
23	Application Support

Appendix B



```
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:1,root,1060,1,782,18446744073709551615,rsyslogd,/var/run/syslogd.pid,/
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:28,root,1715,1710,1556,1466,11,2
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:28,root,1715,1710,1556,1466,10,1
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1718,1715,1556,1466,usleep,/etc/ld.so.cache,,/0,1,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1718,1715,1556,1466,usleep,/lib64/libpopt.so.0,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1718,1715,1556,1466,usleep,/lib64/libc.so.6,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1719,1715,1556,1466,rm,/etc/ld.so.cache,,/0,1,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1719,1715,1556,1466,rm,/lib64/libc.so.6,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1719,1715,1556,1466,rm,/usr/lib/locale/locale-archive,,/0,2598580240,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1720,1715,1556,1466,rm,/etc/ld.so.cache,,/0,1,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1720,1715,1556,1466,rm,/lib64/libc.so.6,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1720,1715,1556,1466,rm,/usr/lib/locale/locale-archive,,/0,3356119056,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:1,root,1720,1715,1556,1466,rm,/var/lock/subsys/rsyslog,,/4294967196,0
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/etc/ld.so.cache,,/0,1,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/lib64/libtinfo.so.5,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/lib64/libdl.so.2,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/lib64/libc.so.6,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/dev/tty,,/2050,1,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/usr/lib/locale/locale-archive,,/0,3538087952,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/proc/meminfo,,/524288,438,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:24,root,1721,1556,bash,3s,526337,0,1
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:24,root,1721,1556,bash,3s,526337,0,1
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/etc/nsswitch.conf,,/0,438,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/etc/ld.so.cache,,/0,1,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/lib64/libnss_files.so.2,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/etc/passwd,,/524288,438,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/usr/lib64/gconv/gconv-modules.cache,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/usr/lib64/gconv/ISO8859-1.so,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1721,1715,1556,1466,bash,/dev/null,,/577,438,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:28,root,1721,1715,1556,1466,3,1
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:28,root,1721,1715,1556,1466,11,2
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:28,root,1721,1715,1556,1466,10,1
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1722,1721,1556,1466,rsyslogd,/etc/ld.so.cache,,/0,1,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1722,1721,1556,1466,rsyslogd,/lib64/libz.so.1,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1722,1721,1556,1466,rsyslogd,/lib64/libpthread.so.0,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1722,1721,1556,1466,rsyslogd,/lib64/libdl.so.2,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1722,1721,1556,1466,rsyslogd,/lib64/librt.so.1,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1722,1721,1556,1466,rsyslogd,/lib64/libgcc_s.so.1,,/0,0,3
Oct 28 03:01:03 host-192-168-22-4 kernel: Progger:0,root,1722,1721,1556,1466,rsyslogd,/lib64/libc.so.6,,/0,0,3
```

Figure B.1: Sample of Chunk File

The measurements taken by Integrity Management Architecture are stored in both binary and ASCII forms `/sys/kernel/security/ima/ascii` runtime measurements and `/sys/kernel/security/ima/binary` runtime measurements respectively

The ASCII version of the measurements can be viewed in plaintext and consists of four columns:

- *PCR number* is the number of the PCR that is extended with the measurement value. In the case of IMA, that it PCR 10.
- *template-hash* the combined hash of the of the contents of the file and the ?filename hint? for the loaded data, expressed as SHA1(**filedata-hash**,

```

m@provenance/home/m
[root@provenance m]# cat /sys/kernel/security/ima/ascii_runtime_measurements | more
10 09c04af7b11f833ce87a826c21db79c3e02df1a8 ima 7ccce77359e77cda71e00b5f66a88fb6236cfc boot aggregate
10 e12d5a82e4a72c4b8435fdeff3e084aa821cbb23 ima 808e809c3bb8b9adbeb9b50041bea096756cd3e9 /init
10 d36c46138fca4dc091e2e0db07dbae2358b3853 ima 3ec3b76addd36e1b88fcd4c6fc3bc695a4d541f /init
10 bedcb1cded5eca88570df398c57c892bb147f04a ima a718a0de8b65ab58acd487f29c1dc94afc2c11da ld-2.12.so
10 33a6567fcd79b26510b574602f13f3e15cb055ad ima db455be4bbe673e9d54a87a3b143628e2ed36c32 ld.so.cache
10 89e016f67ba8eb7102fcd0b9f31bab5fd8e6e0c0b ima e07bba0f5598a41331cca03aee3453ae52c9deb libc-2.12.so
10 8360e31573bad01128e1d8e011ee2f2fc0522045c ima fb1a59cd9ff67fd4c0bb529431ee93d910c4c544 dracut-lib.sh
10 c33392519d281ba922ae7e04f9add63c2f1a8f3a ima b2dd03037bb5922d27ac34180143d365e93788c6 /bin/mknod
10 4ac149698f4f5d643320ecfdba9c1139381d158 ima 384bc1e41edf77dd998062369c7b0cf44088d5ce7 libselinux.so.1
10 51853607df4126f4685b25b420363dc1ed5fe6a3 ima d9a5ca85a6eaf0ee3a32100404165e2a2a4f6461 libdl-2.12.so
10 efcf24a7913551372089d72c29af59bd3f990e49 ima 3360f7ff5fa3dc7a538f30edacf5dbd95d54d8b1 /bin/mount
10 04492c2559983ca72f7eddb76aa86e170538e3c0a ima f985bf43b3f90a1e4965a546e1b34f469359c326 libblkid.so.1.1.0
10 67982689601e37bc22f1b7aaa8b533d8deaf8fb4 ima aafce3cdcfcacbc0796feaffda6c7af2ceda185d3 libuuid.so.1.3.0
10 df63f923dd406367d47fb86aebb897a310f20a3e ima 8efb840c8daa91837496ba14c474967598faf0cf libsepol.so.1
10 6f5c6ef7fd3fa258e303791d0c67e47e799357d1 ima da39a3ee5e6b4b0d3255bfe9f5601890afd80709 mtab
10 150073320cad2d8851e6d31511f3089ce89c2b57 ima caf072e3fd3386cbca7e4c77c94f908dbdf621 mtab
10 1a17c827df174b5b41ebf72db198b85bf973109f ima ee58c560fddab62b0db96c58ec8123ffa3d5e0f1 mtab
10 1f42134c2759b6791c5018d538870b2f9adebeeb ima 9c2922e92a5cfff9f42f8e623dd681c811d668882 /bin/ln
10 7058217dd117fe24f4f92fc47504a50aae1fd365 ima e50c4d3978c70dd80884428e1f1f760f6a842f00 /bin/mkdir
10 6faeddc6587837496fa5adbcfc362c21e9832f4 ima c8ed8d88dd2082fc2c7e611003a460abf217c8b3 mtab
10 51dcb86603efa04b41ee8a76c1d962b15986bfb ima 523007ba232f190b26a9c620a13cda6717a09775 mtab
10 c4d4945f88ea6286d138a857d699acafb9ed17e3 ima 281dd3be929844c38daee3245e3e846135f6425d /sbin/udevadm
10 e9637f39fe4702737d9130bf15ced61579207af ima 798ceeb8d9ef5550400e44727f9dd5ba6ffcc8b5 udev.conf
10 ec6457285bf688a13710e651c0666fa16a9aee5e ima 009aa182ec89dcd39b258be2b48c15ea77dd9c9 01parse-kernel.sh
10 352a1d3036afcd0b7dca9afb9ac06f93e41263faae ima f6dce9dd0e9bfb8e6c2145bf9f6400bf2132ceda 01version.sh
10 7020789348d22dd43ea5a0f148601fc190f89324 ima 07e1509a10b8c89bb3b6391be0923cedbbfbee9 dracut-004-303.el6
10 5bf7eac3f9d0e7ab5c2777aa76aca0465d43ade ima 68d8bb831131c129be0997af3d34af073b1c447 10parse-resume.sh
10 72f17ef4e521fb16867ea44cb0dbdc06790bb18 ima f57439b6da09ab7a26e1ec7a8bc2e843860a724a 10parse-root-opt.sh
10 0bd2bdade60940a2c38675d43024554a37a61e2df ima f07cb161ccf6a86a8b1e040511a3c9984a61a09 20parse-blacklist.sh
10 28e3b0d4551c44cd4bd084ba318e3c050fc4964b ima e5cee805a9e0cb047dcd0d9dc546e629f8ed243a 20parse-i18n.sh
10 9527734312eab5ebbd66e816a3fb9fcbhcc76433 ima 6772b46c817182289178d55bf888ffcd92c77da1 i18n
10 35fac6bd336c36a5f451c8eed91fa21e78a51955 ima 2bb8e905798dcd0e586d50b99e32cc298ac7f905 20parse-insmodpost.sh
10 3205f32f5a0006ecb381c573fbc2667663ebeece8 ima 18453af93844343a61034c58d4e46861dd632ccf 30parse-crypt.sh

```

Figure B.2: Sample list of IMA runtime measurements

filename-hint), where filename-hint is 256-byte long.

- *Filedata-hash* is the hash of the contents of the file containing the loaded or executed data, expressed as **SHA1(filedata-hash)**.
- *Filename-hint* is the filename of the included file, or an identifier for the loaded data (as in the case of ?boot aggregate?, which is the resulting hash from the PCRs 0-7).

Below follows a sample fragment of the IMA runtime measurements obtained from the host where the compute node ran in the implementation setup.

Appendix C

TPM Commands After the administrator takes ownership of TPM, he has the ability to manage and use TPM. These command could be locally or remotely. If the administrator forget the password of the TPM ownership password he then the only way to reset this password is to reset TPM chip through the BIOS. And this add more confidentiality to TPM solution.

Table C.1 shows some of TPM command with the description of each command. These command compatible with TPM 1.2, since we use used this version in our framework. In this table we tried to choose most important command that we frequently use it in our framework.

Table C.1: Some of TPM Commands

TPM Command	Description
TPM_version	Show the version of TPM
TPM_takeownership	Basic command for creating an SRK and an owner for a TPM. Until this command is executed, the TPM cannot do much of anything.
TPM_Extend	Used to update a value in a PCR.
TPMNV_WriteValue	Write to the NVRAM space.
TPMNV_ReadValue	Read from NVRAM space.
TPM_sealdata	seals sensitive input data to the SRK of the system's TPM and optionally a PCR configuration.
TPM_unsealdata	Decrypt the sensitive data based on the values of PCRS which was used on seal time.
TPM_getQuote	TPM signed the PCR values from the chip. The program to obtain a quote, and thus measure the current state of the PCRs.
TPM_loadkey	The program loads the key in BLOB-FILE into persistent storage and registers it using the UUID in UUID-FILE.
TPM_mkuuid	The program generates a TPM UUID and stores it in the file UUID-FILE.
TPM_updatepcrhash	This program updates the PCR composite hash in file OLD-HASH-FILE to produce the file NEW-HASH-FILE.