TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Datenverarbeitung

Runtime integrity framework based on trusted computing

Chun Hui Suen

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technische Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr. -Ing.)

genehmigten Dissertation.

Vorsitzender:	Univ. -Prof. Dr. -Ing. U. Schlichtmann
Prüfer der Dissertation:	
	1. Univ. -Prof. Dr. -Ing. K. Diepold
	2. Univ. -Prof. Dr. -Ing. G. Sigl

Die Dissertation wurde am 02.02.2012 bei der Technische Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 26.06.2012 angenommen.

Runtime integrity framework based on trusted computing

Chun Hui Suen

Runtime integrity framework based on trusted computing

Chun Hui Suen

July 2, 2012

Lehrstuhl für Datenverarbeitung
Technische Universität München

Abstract

I present in this dissertation, a technique to measure the integrity of an operating system, so that the user can verify that all critical software components, including the operating system kernel, is running in a known valid state. The technique solves a key problem of providing continuous runtime verification of kernel memory-space. The measurement is integrated with a trustworthy verification chain from the firmware, host machine, hypervisor, guest machine to applications, based on existing techniques from Trust Computing and guest security mechanisms.

This is accomplished by checking the guest kernel against a known reference, to provide instant feedback on changes in its integrity. A Trusted Platform Module (TPM) is used to provide a complete integrity measurement chain from the hardware to the host and guest system. A working implementation of the entire framework has been achieved for a 64-bit Linux host and guest system, using QEMU and KVM as two different virtualization techniques. The implementation has been verified to correctly detect integrity changes in the guest, while maintaining a minimal performance overhead.

The technique is generally portable to other operating systems. It is implemented as an integrity measurement framework for the Linux kernel, which can be extended to utilize additional measurement capabilities of the guest operating system, forming a more in-depth measurement. Prototypes for such extensions are implemented using two existing Linux security modules. An example of trusted authentication and host-based intrusion detection has been used as proof-of-concept application scenarios for the integrity measurement framework. Benchmarking on the system shows that the integrity measurement has minimal impact on the guest machine performance, with only slight overhead during the guest machine boot time. Correctness and security strength of the framework were verified using functional and penetration testing.

Contents

1 Introduction

1.1 Motivation

In a time of increasing security concerns, the focus of this dissertation is to propose and implement a framework to enhance software integrity. As is good computer security practice, security needs to be handled in depth, meaning that a variety of security mechanism targeting different depths of IT infrastructure. This depth can be classified broadly into network-based and host-based techniques. Further dividing this, the network security can be handled at the edge, known as perimeter defense, and also on the internal network. Similarly, security can be applied at different depths of the software stack on the host, namely application, kernel-space, hypervisor (if virtualization is used) and on the firmware. Table 1.1 shows a high-level classification of commonly used security techniques at different depths. While there is a mixture of perimeter and in-depth defense on the network side, most security solutions on the host are on the application level.

Current security mechanisms protecting the kernel space are limited, because a compromised kernel can completely hide signs of infection by directly modifying the kernel's file and process listing function. This puts limitations on anti-rootkit protection as it is non-trivial to detect rootkits based on rootkit signatures. Driver signature verification is a means of protecting the kernel by checking the signature of an executable or driver before loading it into the kernel. However, this can often be overridden by the user to support unsigned drivers.

Furthermore, secure booting is needed to provide verification of the firmware, hypervisor and kernel during the boot process. However, current secure boot only verifies each loaded component once during boot time, providing no continuous monitoring, especially of the kernel. Unlike network devices which operate independently on the network, the software stack on the host is inherently dependent on underlying components. Rather than independent security techniques that are applied at each level, it is necessary to integrate these protection schemes to maintain the overall effectiveness of the mechanism.

Since rootkits work at the kernel memory space, they are able to intercept core system functions to hide their presence and directly access the file and network. Potential dangers are the loss of high-value data such as confidential files or credentials by means of keylogger or file access. The use of encrypted outbound connection initiated by a rootkit can evade both host (because of rootkit's stealth properties) and network level (because of encryption) firewall. Security against rootkit is necessary in high security scenario where the platform is used to store or access high value data, such as financial transaction, health care data, privacy information, cryptographic keys, confidential material, and so on. Cur-

Network-based		Host-based			
Perimeter	Internal network	application	kernel-space	hypervisor	firmware
VPN		Anti-virus			
Firewall		Malware detection			
Intrusion detection		Personal firewall			
	VLAN	Software white-list			
	Deep packet inspection		Anti-rootkit protection		
			Driver signature check		
			Trusted booting		

Table 1.1: Classification of security techniques

rent security mechanisms simply do not provide sufficient protection for unknown rootkits, where signature-based detection fails at the kernel level.

Thus, the motivation of this dissertation would be to develop an extensible integrity framework, which forms a continuous trust chain from the firmware to applications, while providing continuous runtime monitoring of all active components within the software stack, including the kernel. Such a solution would be necessary to provide effective non-signature based detection of rootkits, including zero-day attack.

1.2 State-of-the-art

On the cutting-edge of improving operating system kernel security, two general approaches are taken. One track works on proposing completely new OS architectures, which provide new security properties, while another uses new techniques to improve security in traditional monolithic kernel designs. New OS kernel architecture such as the use of micro-kernel design [15, 22, 25] defines clearly separated components and OS functions, improves the OS security and reliability as proposed in [63]. Both the MINIX3 OS proposed in [63] and the L4 kernel from [25] has a similar structure where the core kernel handles memory management, scheduling and inter-process communication, while the basic OS functions are implemented as additional layers. Hardware drivers handles the IO communication to a specific hardware device, while servers depend on the drivers to provide abstract basic OS services such as file system, network communication, security policies and so on. Finally, system and user applications are executed as processes which depend on the respective servers. A fundamental difference to monolithic kernel, is that none of the drivers, servers and processes runs in the kernel memory space, allowing the kernel to enforce protection policies against all the components.

By dividing the kernel into modules at different layers, the Trusted Computing Base

(TCB), which is the set of software required to implement the necessary security policies, is reduced from the entire monolithic kernel to the micro-kernel and security related modules. As suggested by [50], this greatly reduces the security perimeter of the TCB, and makes the TCB easier to verify.

In terms of finding actual vulnerabilities in the kernel, a fundamental security risk of using procedural programming language, such as C, is its flexible use of pointers and unsafe types. The lack of type and memory boundary checking opens the possibility to a large class of vulnerabilities. Prototype kernels using newer programming language paradigms such as Microsoft's Common Language Runtime (CLR) or Haskell, have been demonstrated in [36, 66], respectively. Singularity [36] is a kernel implemented entirely in CLR. It uses the implicit type checking in the CLR environment to prevent buffer overflow attacks. The HouseOS [66] goes a step further to provide formal verification [42]. It implements the kernel in Haskell with a defined hardware model for physical memory, virtual memory, IO ports and interrupts; such that it can be formally proven that there is no memory corruption. These techniques protect the kernel from entire classes of vulnerabilities which plagued code developed in machine-bound languages such as C and assembly languages.

However, both approaches of changing the kernel design or the programming model cannot be applied to existing operating systems. The industry has only shown limited adoption of micro-kernel design in specialized fields, due to problems of performance and code migration. Also, the new programming models have exceptions to their formal proofs, due to direct memory access from the hardware.

Techniques that are applicable to existing operating system are based on memory monitoring techniques being applied to the kernel-space. These techniques utilize virtualization to check for vulnerable conditions in the kernel. Various work in [46, 21, 54] attempt to detect data and code corruption. However, one fundamental challenge in verifying runtime memory is the dynamic nature of loading executable code. On the x86 instruction set, relative jump commands are limited to a single byte jump addresses. To support code segments that are large than single byte address space, most binaries cannot be compiled in a location independent manner. Location independence refer to the property that code can be loaded and executed anywhere in memory regardless of the base address of the code segment. Location independence is thus achieved through a process known as dynamic relocation, which must take place before the code is executed. Due to relocation, the code is modified every time it is loaded, which poses a problem for verification in the memory space. Furthermore, dynamic linking imports functions from external libraries such that library functions can be shared between multiple applications to save memory. This also brings runtime variation to the executable memory. The challenge can be observed in [54], where the code corruption detection ignores jump addresses in the code memory due to the dynamic relocation and linking. Further work was done in Patagonix [44], which verifies jump addresses within the same module. The technique proposed in this dissertation serves to extend the solution by verifying external import addresses as well.

In terms of cross layer security integration, Trusted Computing (TC)[9] provides a means of linking trust measurements of various software components starting from the firmware.

However, existing trusted boot process, such as the Linux integrity measurement architecture [56, 23] and Microsoft bitlocker [48], only perform a one-time verification of each of the loaded software component during boot-time. This dissertation integrates kernel space monitoring techniques with Trusted Computing to provide trust measurements which are 'live' (runtime verified).

1.3 Problem description

In order to achieve the goal of building a comprehensive integrity framework as motivated in 1.1, the key challenges that have to be solved are as follows:

1. Overcome the problem of code modification due to dynamic relocation and linking
2. Develop an efficient technique to monitor kernel memory space continuously in runtime
3. Provide a continuous trust chain linking the verification of firmware, hypervisor, kernel and user-space applications, such that the entire OS is eventually verifiable

In order to achieve reliable runtime verification, a systematic approach to handle code relocation and linking is needed, such that the 'live' kernel executing in the kernel memory space can be efficiently verified against a known version of the kernel. The memory monitoring technique also needs to be efficient, such that the end solution is practical for normal usage. Based on this, a complete trust chain needs to be established from the firmware, hypervisor, kernel to the user-space application, such that the security mechanism at the application level can produce a trustworthy report of the entire system to the end-user.

1.4 Overview and main contributions

In order to overcome the problem of code modification due to relocation and linking, this dissertation provides a background on the 2 most common binary executable file formats in section 3.4.1, and describes how this is related to problem of dynamic relocation and linking in section 3.4.2. As the relocation process is specific to each executable binary, a pre-processing stage is necessary to extract information from a known Linux kernel such that it can be verified efficiently in runtime. The design of this pre-processing stage is described in section 3.4. Using this database, an efficient verification process is proposed in section 3.6. Implementation details are covered in section 4.2.

On the foundation of existing techniques in the space of kernel-space monitoring [46, 21, 54], the verification scheme is combined with existing software and novel hardware-based virtualization technique to achieve near-native performance, while maintaining continuous kernel memory monitoring in runtime. The background on virtualization is covered in sections 2.3 and 2.4, while sections 3.5 and 4.3 cover the design and implementation of the runtime monitoring scheme.

This continuous verification scheme for the kernel is integrated with upstream measurement of a trusted boot process using techniques developed in Trusted Computing. Section 2.2 provides background information to Trusted Computing and section 3.6.4 describes how the security state of the kernel is monitored in runtime. The combined integrity information is also reported up the software stack into the guest kernel and application layers by means of a virtual Trust Platform Module (vTPM). Various design options of such a vTPM is discussed in sections 4.5 and 4.6. The verification measurement is then integrated with existing application level security mechanism, so as to give a continuous chain of trust in the entire operating system. Integration of two existing application level security mechanism in Linux is described in section 5.1, while sections 5.2 and 5.3 shows how a complete trusted file encryption and a host intrusion detection system can be built on this trusted framework, respectively.

Section 6.1 verify the correctness of the framework based on functional and penetration testing methods, while section 6.2 evaluate the security of the system based on the existing uses cases. Performance of the framework is discussed in section 6.3 and is shown to achieve near native performance.

2 Background on Trusted computing and virtualization

This chapter will briefly describe background information about the supporting components around the main topic of this dissertation. The topics discussed serve to provide a foundation to aid in the understanding of the proposed core ideas. Section 2.1 on trust and integrity sets the basic definition of integrity, which is main focus of this dissertation. After that, a brief introduction on the main concepts and the components of Trusted Computing and the Trusted Platform Module (TPM) is covered in section 2.2. The TPM is an important component that integrates with the integrity measurement framework. Basic TPM operations relevant to the dissertation as well as the larger framework of the TCG specifications relevant to this work will be explained. Then, section 2.3 will describe various forms of virtualization to give a general introduction to the field of virtualization, with emphasis on software and hardware-based virtualization that is used in this dissertation. Finally, an introduction to memory management on the PC(x86 architecture) platform is given in section 2.4, as a necessary prerequisite to understand virtualized page management.

2.1 Trust and integrity

In this context, trust is defined as the confidence that an entity or system operates as predicted. In an ideal case, this would mean that the behavior of a system matches it's given specifications exactly. However, the verification of the binary code against its specifications is a non-trivial task, involving static and possibly dynamic software analysis. Thus, in this dissertation, the definition of trust measurement will be restricted to that of the integrity or immutability of the binary code of software, such that the behavior of the software is consistent with any previous verbatim instances of itself.

An integrity measurement is a measure of some parameters of a component or system, in order to characterize its behavior. Anti-virus software, for example, protects the integrity of a system by detecting any malicious change to files, executables, system registry and components, from within the OS. This approach tends to fail if a rootkit manages to install itself into the kernel of the OS through a kernel vulnerability and hides its presence from the anti-virus software. Other similar security "hardening" approaches such as *SELinux* [64] or *AppArmor* [10], control the interaction between users, processes and critical system objects (resources) rather than detecting particular code signature of malicious software. However, they also suffer from the same weakness if the kernel itself is compromised, due to the lack of runtime monitoring or a continuous chain (or tree) of trust starting from a secure root of trust. A chain (or tree) of trust, is an integrity measurement method, where each component measures the integrity of its dependent components, such that any change along this measurement chain is detectable.

The work in trusted computing as described in section 2.2, solves the problem of chain of trust, but fails to address the runtime nature of integrity measurement. Drawing from the trusted computing concept proposed by the Trusted Computing Group (TCG) [9], and previous work in property-based attestation [30], kernel-space monitoring [46, 21, 54, 52], an extension of existing kernel-space monitoring techniques with binding to the Trusted Platform Module (TPM), is discussed here. This dissertation proposes the design of an integrity measurement framework. The framework measures the runtime integrity of the kernel of a guest OS, using kernel-space monitoring techniques. Many of the basic concepts proposed in this dissertation are based on previous works, and described in more detail in sections 2.2 and 3.2.

2.2 Trusted computing (TC)

Trusted Computing (TC) is a general term used, in this context, to describe the collection of technology developed and specified by the Trusted Computing Group (TCG)[9]. The initial focus of trusted computing was on the development of a hardware root of trust for a software stack, known as the Trusted Platform Module (TPM) [8], and its associated software (such as the Trusted Software Stack(TSS)). The TCG has since expanded to include specifications related to architectural considerations of Trusted OS, trusted networking, usage of TPM in mobile or embedded environment and so on.

Trusted computing enables a trusted measurement, enforced through cryptography, of a software stack with a root-of-trust embedded in the hardware. A software stack refers to the entire set (or stack) of software necessary to boot, and also, those which makes up the entire operating system, starting from the BIOS ROM to boot-loader, operating system loader, operating system kernel, drivers, system software and user applications. The specifications of the TPM and TSS itself do not explicitly state how the entire software can or should be measured. They only specify the means to make trustworthy reporting of measurements, storage of the measurements on the TPM and the usage of the stored measurements. Policy enforcement based on the TPM and its trusted software stack is achieved by binding the sensitive operation to cryptographically-protected keys, which are, in-turn, bound to a particular set of measurements.

2.2.1 Trusted Platform Module (TPM)

The Trusted Platform Module (TPM) is a chip, usually physically bound to the mainboard of a computer, which performs the necessary core cryptographic functions and holds the

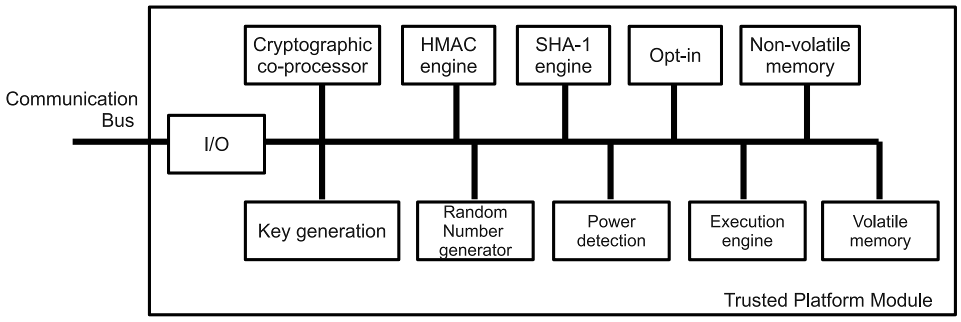


Figure 2.1: Components of a TPM

core root-of-trust for reporting and storage, to enable the various functions as specified by TCG. Figure 2.1 shows the basic building blocks of a TPM.

The cryptographic co-processor, HMAC engine, SHA-1 engine, key generation and random number generator are the key cryptographic components of the TPM. The opt-in component allows the TPM to be enabled or disabled completely, so that the owner of the machine always has the choice to disable TPM usage all together. Non-volatile storage is used to store internal keys such as the Storage Root Key, Endorsement Key (see section 2.2.3) and additional data in the non-volatile storage area. The rest of the component is used for the normal execution logic of the TPM.

Locality

With the TPM 1.2 specifications, the concept of locality is introduced. Locality is an attribute associated with TPM commands received by the TPM. It is not determined by the TPM itself, but is signaled to the TPM during the instruction cycle. On the PC platform, the TPM locality is determined via the Memory-Mapped Input Output (MMIO) memory location used to access the TPM. The MMIOs are page-aligned, and each page is used for one locality. The use of page-aligned MMIOs enables a trusted OS kernel to allow or to block the usage of a specific locality using memory page permissions.

Each locality has a loosely defined role as described in table 2.1, as extracted from section 2.1 of [6]. Each locality has pre-defined restrictions on some Platform Configuration Register (PCR) access as described in table 2.2. The TPM can only be active in a specific locality at any one time. Thus, in the situation of simultaneous access from multiple localities, a higher locality always takes precedence over lower localities. The use of locality allows the TPM to be controlled by different roles, and to perform normal or privileged operations.

Locality	Description
4	Trusted Hardware. This is the Dynamic RTM
3	Auxiliary components. (optional)
2	“runtime” environment for the Trusted Operating System
1	Environment for use by the Trusted Operating System (T/OS)
0	Legacy environment for the Static RTM and its chain of trust
legacy	Locality 0 using TPM 1.1 type I/O port access

Table 2.1: TPM Localities

Platform Configuration Registers (PCR)

The trusted measurements of a TPM are stored in a set of registers, known as the Platform Configuration Registers (PCR). From the concept of transitive chain-of-trust, the PCR values form a representation of the current state of the system since boot-up. Keys used by the TPM can be bound to a chosen set of PCR values, to ensuring that a key can be used only in a per-determined system state corresponding to the set of PCR values.

The basic operation on the PCR is the extend operation, described by:

$$\begin{aligned} PCR_{n,i} &= PCR\_Extend_i(ExtendValue_n) \\ &= hash(ExtendValue_n \,|\, PCR_{n-1,i}) \end{aligned}$$

Thus, the PCR hash of register i after its n^{th} extend operation, $PCR_{n,i}$, is always dependent on the current extend value, $ExtendValue_{n,i}$, and the set of all previously extended values, $\{PCR_{a,i} | a \subset 0 \ldots n-1\}$. In the TPM 1.2 specification, SHA-1 is the defined hash operation and both the PCR hash and the extend value are 20-byte strings, corresponding to the size of an SHA-1 hash. A reset operation on the PCR resets the PCR hash to either all zeros or all one bits. The ability to perform the extend or reset operation on the PCR is determined by the current locality of the TPM, as described in table 2.2 (compiled from sections 7.2 and 7.3 of [6]). PCRs 0 to 15 cannot be reset, but can be extended in all localities. PCRs above 16 can be reset, depending on the current locality of the TPM, but have restrictions on the extend operation.

Transitive chain-of-trust

The transitive chain-of-trust defines a concept of measuring and passing control to the next module as illustrated in figure 2.2. The measurement process begins from the Core Root-of-Trust for Measurement (CRTM), which is the BIOS on the PC platform. The CRTM measures the next component to be loaded, which is the boot sector (assuming a boot from hard disk). The measurement is stored into the Core Root-of-Trust for Reporting (CRTR), which is the TPM, via the TPM_EXTEND operation of the TPM. A cryptographic

PCR Index	PCR Usage	Reset locality	Extend locality
0	CRTM, BIOS and Host platform extensions	n.a.	0-4
1	Host Platform configuration	n.a.	0-4
2	Option ROM code	n.a.	0-4
3	Option ROM configuration and data	n.a.	0-4
4	IPL code (usually the MBR)	n.a.	0-4
5	IPL Code configuration and data	n.a.	0-4
6	State transition and wake events	n.a.	0-4
7	Host platform manufacturer control	n.a.	0-4
8-15	Static Operating System	n.a.	0-4
16	Debugging	0-4	0-4
17*	Associated with the DRTM	4	2-4 †
18*	Host platform defined	4	2-4
19*	Trusted Operating System	4	2,3
20*	Used by Trusted Operating System	2,4	1-3
21*	Used by Trusted Operating System	2	2
22*	Used by Trusted Operating System	2	2
23	Application	0-4	0-4

* Reset to -1 upon platform reset (all one bits). Reset to 0 upon DRTM entry.
† First extend operation after reset only in locality 4.

Table 2.2: TPM 1.2 PCR usage

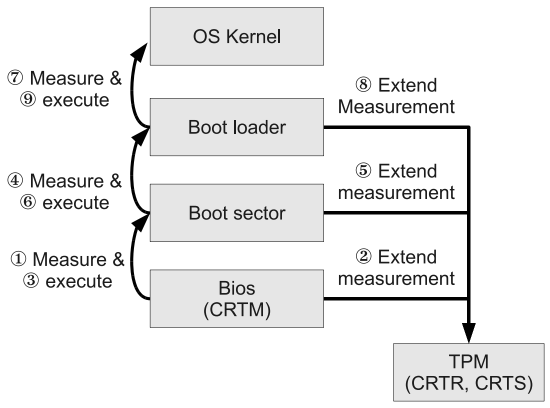


Figure 2.2: Measure and extend

hash (SHA-1 in the case of TPM specifications 1.1 and 1.2) calculated over the code to be executed is used as the measurement value. After the measurement is reported, control is passed onto the next module, boot sector, and the same cycle is repeated until the operating system is loaded. It is thus a requirement that every loaded component is TPM-aware (including the BIOS, boot sector and boot loader), and has to perform the 'measure and extend before execution' procedure.

In this manner, a transitive chain-of-trust is formed, in which the trustworthiness of each loaded component is inherently dependent on each previous component, except the CRTM which has to be implicitly trusted. Thus, in order for the chain-of-trust to be trustworthy, two conditions must be satisfied:

1. The CRTM must reset the PCRs to known entries upon system reset, be trustworthy and cannot be modified.
2. Every software component must be measured and extended into the TPM, before execution.

Based on these two conditions, the system's state can be fully represented through the PCRs. This also means that if the chain of measurements starting from the CRTM to the current component is integral, it implies that the components loaded are also integral.

2.2.2 Trusted Software Stack (TSS)

The Trusted Software Stack (TSS) is the software stack within an operating system used to provide a software interface to the TPM, much like how a network stack provides an usable software interface to a network device. As defined in section 1.2 and 1.3 of [18], the TSS consists of 3 layers (as shown in figure 2.3):

- Trusted Device Driver Library (TDDL)

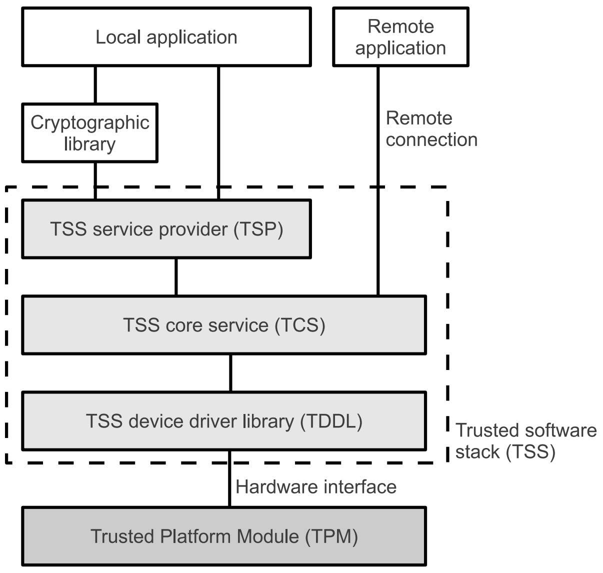

Local application
Remote application
Cryptographic library
Remote connection
TSS service provider (TSP)
TSS core service (TCS)
TSS device driver library (TDDL)
Trusted software stack (TSS)
Hardware interface
Trusted Platform Module (TPM)

Figure 2.3: Trusted Software Stack

- Trusted Core Services (TCS)
- Trusted Service Provider (TSP)

The TDDL provides the low level TPM vendor-specific interface to the TPM hardware. This allows the TCS and TSP to have a common interface to different TDDLs when accessing TPM from different vendors and on different operating systems. On Microsoft Windows Vista and above, the TDDL has been replaced by the Trusted Base Service (TBS) as a solution to multiplex TPM access between the TSS and the operating system itself. Furthermore, it also allows for security policies to be defined for specific TPM commands.

The TCS provides a common set of core services in the TSS, and manages multiplexed access of the TPM from multiple service providers, either local or remote. This allows the TPM to be used simultaneously by multiple local and remote applications.

Finally, the TSP and a supporting cryptographic library, provide the top-level service interface used by a local TC application. Remote applications can connect, through the TSP on the remote-end, to a TPM via a Simple Object Access Protocol (SOAP) connection to a local TCS. The TSP interface (TSPI) is the main interface used by applications to perform all the necessary functions to create and manipulate keys, perform sealing and unsealing, attestation, extending the PCR and other operations.

2.2.3 TPM Keys

In addition to the PCRs, the TPM also manages an entire hierarchy of keys. Only two keys are actually stored within the TPM, the Endorsement Key (EK) and the Storage Root Key

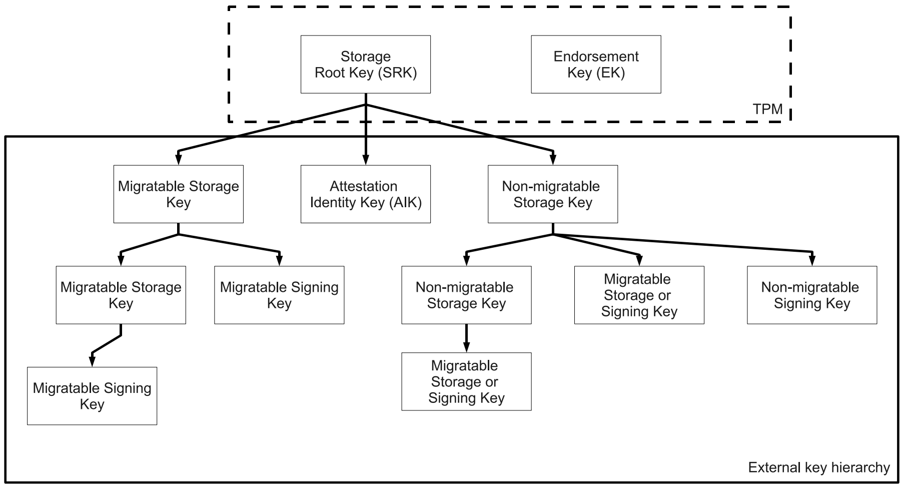

Storage Root Key (SRK)
Endorsement Key (EK)
TPM
Migratable Storage Key
Attestation Identity Key (AIK)
Non-migratable Storage Key
Migratable Storage Key
Migratable Signing Key
Non-migratable Storage Key
Migratable Storage or Signing Key
Non-migratable Signing Key
Migratable Signing Key
Migratable Storage or Signing Key
External key hierarchy

Figure 2.4: TPM key hierarchy

(SRK). The EK is created during manufacturing time and is unique to each TPM. The SRK is created by the TPM internally when ownership of the TPM is taken. Both EK and SRK are asymmetric keys and their private part never leaves the TPM. The SRK is used as the root key of the key hierarchy as a Key Encryption Key (KEK). Besides the EK and SRK, all other keys are stored externally to the TPM as an encrypted key blob, and are managed by the TSS (See Figure 2.4). Child keys are encrypted with its parent key with the following wrap operation:

$Encrypted Child Key Blob = RSA ENCRYPT_{Parent Key}(Child Key)$

The top level key blobs are encrypted by the SRK, while subsequent keys are encrypted by their respective parent keys. To retrieve a particular key, the respective top level storage key blob is loaded into the TPM, and the TPM unwraps the key blob using the SRK and retrieves the public and private part. With this key residing in the TPM memory, the respective child key blob is loaded and unwrapped. This process is repeated until the required key is retrieved.

There are 4 types of keys supported by the TPM: Storage key, Signing key, Attestation Identity Key (AIK) and legacy keys. Storage and signing keys can be set as migratable or non-migratable, while AIKs are always non-migratable. Storage keys can be used to encrypt or decrypt arbitrary data of restricted length (known as the SEAL operation) and can be used to wrap child keys. However, non-migratable keys can only be wrapped by another non-migratable key. Signing keys are used for signing operations only and cannot be used for encryption of data or keys. AIKs can be used for signing a non-migratable key or in a QUOTE operation (see remote attestation). Legacy keys from the older TPM 1.1 specification can, however, perform both encryption and signing. In addition, each key can optionally be bound to a password and a set of PCR values before the key can be used.

2.2.4 Remote attestation

Remote attestation is the process of attesting one or more properties of a target machine (attester) to a verifier over the network. In the context of Trusted Computing, the properties to be proven are a chosen set of PCRs. The AIK is used for remote attestation by creating digital signatures using a QUOTE or CERTIFY_KEY operation. The QUOTE operation generates a signature of the current PCR measurements, over a pre-defined structure TCPA_QUOTE_INFO [2]. This structure contains a version number, a hash of the chosen set of PCRs and an arbitrary 20-byte string which is usually used as a nonce for the verifier.

Alternatively, the AIK can be used for the CERTIFY_KEY operation which generates a signature of a non-migratable TPM key, over the TCPA_CERTIFY_INFO [2] structure. This structure contains a version number, the properties of the key, hash of the key's public part and nonce. The key properties include, among other things, its type and the PCRs that the key is bound to if any.

In order to prove that the AIK used comes from a genuine TPM, an AIK credential has to be obtained from a Privacy CA (PCA). Figure 2.5 shows the process of obtaining an AIK credential, usually in the form of an X.509 certificate, from a Privacy CA. An alternative method known as Direct Anonymous Attestation (DAA) is also available starting from TPM 1.2 specifications, but it will not be discussed in this dissertation.

The AIK proof contains the TPM's endorsement certificate (which includes the TPM's public part of the EK), the public part of the AIK generated by the TPM and additional information related to the TPM. This is sent securely to the PCA by encrypting it with the PCA's public key. The PCA has the role of verifying the TPM's endorsement certificate to ensure that the TPM is genuine, before issuing an AIK certificate signed by the PCA to certify that the generated AIK is genuine. Since the EK is always linked to a unique platform, the PCA protects the privacy of users by disassociating the AIK from the EK of the attester.

2.2.5 Trusted Infrastructure

The TPM and TSS specifications from TCG define the hardware and software interfaces, necessary for accessing and using the TPM on a local or remote platform. However, they do not define or imply the usage of the TPM or its trusted services in the actual context of protecting information and identity within the operating system and a network of systems.

Thus, the Infrastructure Working Group (IWG) within TCG defines a broad set of specifications, covering interoperability and connectivity (via Trusted Network Connect) between systems using a TPM or other TCG technologies. The IWG covers topics related to certificate management, migration, attestation, practical use cases and the management of integrity.

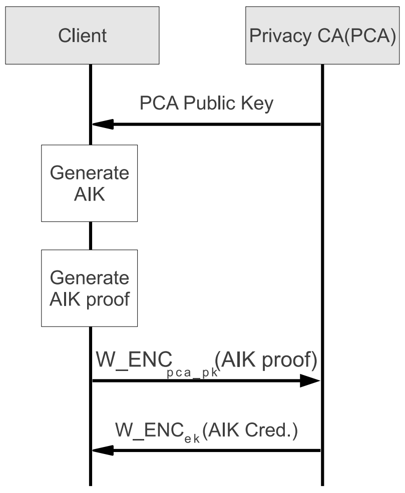


Figure 2.5: AIK credential generation via a Privacy CA

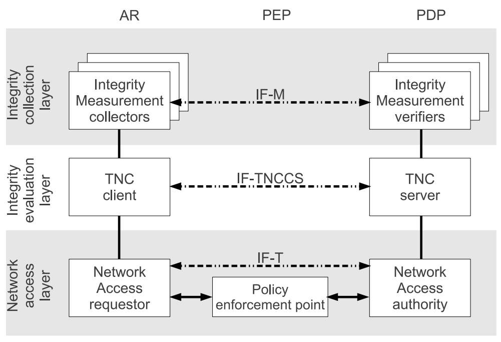


Figure 2.6: TNC protocol stack

2.2.5.1 Trusted Network Connect

Trusted Network Connect (TNC) is a derivative from the IWG, describing a multi-layered network protocol designed to perform remote attestation between two machines. In addition to the most fundamental QUOTE and CERTIFY_KEY operations provided by the TPM, the TNC specifies the necessary framework to perform remote attestation within an organization. It is designed to integrate into the existing network access protocol such as 802.1x and Transport Layer Security (TLS), to perform authentication and encryption. Figure 2.6 (source: [13]) shows the TNC protocol stack, defining 3 communication layers from the Access Requester (AR) to the Policy Enforcement Point (PEP) and Policy Decision Point (PDP).

AR refers to the client computer which is requesting to connect to the restricted network, through the network edge at PEP, which is the main enforcement point. Usually, the PEP can be a switch, firewall or VPN gateway, depending on the location of the AR with respect to the restricted network. The PDP is the entity which decides, based on defined policies, if the client is allowed to join the network. IF-T provides a secure transport session between the AR and the PEP, with defined bindings to 802.1X and TLS. IF-TNCCS and IF-M are high level protocols layered over IF-T, to transmit control messages and integrity measurements, respectively. The use of integrity measurement such as the PCRs from a TPM augments the information available about the AR to the PDP, which can then be used in the policy decision on whether to accept the AR into the network. Due to the complexity of the proposed implementation, the applications of integrity measurement presented in chapter 5, will draw ideas but not utilize the specifications from IWG and TNC directly.

2.2.6 Previous work on property attestation

An early implementation of transitive chain-of-trust achieved by directly measuring loaded binary code was demonstrated by an implementation from IBM called Linux Integrity Measurement Architecture(IMA) [56], and also in Windows Vista's and Windows 7's bitlocker [48]. This was thus termed "binary attestation" from previous literature [30]. IMA has been integrated into the mainstream Linux kernel since version 2.6.30, and it involves a Linux system which measures and logs every loaded component from boot time, including the kernel, kernel modules, initial ramdisk and all subsequent binaries. Binary attestation is a measure-before-load approach. This means, any component that is needed by the system (such as binary code, system libraries, files, etc.) is first measured before being loaded. The composition of all measurements will attest to the integrity of the chain of components that make up the running system. This approach, however, suffers from extensibility (ability to add or update system components) and privacy problems (knowledge of installed hardware and software) as highlighted in [30], leading to the development of property-based attestation [30], and how it can be used to resolve extensibility issues and protect the privacy of the attested platform from the party requesting attestation.

Property-based attestation, however, still has shortcomings in that it cannot verify the runtime behavior of a component or system, as they are only measured once during load time. To overcome this problem, ideas from runtime kernel-space monitoring techniques were considered: such as Linux Kernel Integrity Measurement (LKIM) in [46] and nickle [54]. These works directly monitor the memory of a guest virtual machine, in order to determine its integrity state, and enable the integrity of the guest machine to be continuously monitored, throughout its entire uptime.

2.2.7 Dynamic Root-of-Trust

Dynamic Root-of-Trust for Measurement (DRTM), also known as late launch, is a feature which allows a separate root-of-trust for measurement to be re-established after boot time. The advantage of this approach is that the BIOS (other than setting up the hardware properly) and other low level boot process do not have to be trusted, and the CRTM is started after the Trusted Computing Base (TCB) is loaded into memory. This allows for more flexibility in terms of the boot process.

DRTM is initiated via a special privileged instruction that resets the processor state and executes the given secure loader from memory. OSLO [40] is one such specialized loader which was written to support the DRTM feature on AMD processors. OSLO is initially loaded into memory via a conventional boot loader such as GRUB[34] or SYSLINUX[35]. It then reloads itself using the *SENTER* (AMD-specific) instruction, which performs the following steps:

- stopping multi-processor mode if active,
- resetting the processor to a defined state,
- resetting the TPM PCRs 17 to 22 to all zeros,
- making a hash of the defined secure loader (OSLO) and extending this hash into PCR 17 (as defined in table 2.2),
- setting the TPM locality to 4,
- jumping to the entry point of the secure loader (OSLO).

After the OSLO stub regains control after the *SENTER* instruction, it hashes and extends the following modules in the boot chain into PCR 19, before handing over control to the next module. Two additional modules are loaded to configure the necessary environment before the Linux kernel and ramdisk are started. Using this approach, the dynamic root-of-trust begins from the SENTER instruction and all executable modules that follow are measured in a chain-of-trust based on this new root late in the boot process, thus being named 'late launch'. On the PC platform, unfortunately, the kernel is not completely independent of BIOS, as the proper initialization of hardware in the Linux kernel would still depend on the Advanced Configuration and Power Interface (ACPI) table loaded by BIOS for proper Plug-and-Play (PNP) hardware detection. The advantages and problems of DRTM are also discussed in [40].

2.3 Understanding virtualization

Virtualization technology has become an increasingly popular trend, starting from server consolidation to cloud computing and software-as-a-service (SaaS). The following sections aim to give a general overview of the general classes of virtualization technology in use.

Virtualization of software can be generally abstracted into 2 kinds: virtualization of a complete system and virtualization of the software environment around a single or group of processes. The former, system virtualization, deals with emulating a complete system including the processor, memory and hardware peripherals, such that an operating system can run on top of the virtual hardware.

The latter, application virtualization, deals with virtualization of a software environment that emulates a thin layer between either a single process or a group of processes and the underlying OS. This is used in application virtualization such as ThinApp [70], where the standard OS calls made by an application are redirected to a 'sandbox' environment, thus providing isolation between the application and other components and resources on the system. In the same way, the WINE runtime environment [32] and ming libraries virtualize the system API of Windows for a Linux process, and Linux API for a Windows process, respectively. A similar concept is also used in container-based virtualization [69], where independent trees of process and resources are isolated from each other, and form simultaneous OS instances which are independent, albeit a minimal common kernel and kernel services. These approaches provide isolation and security, while keeping a minimal performance and resource overhead. The different kinds of virtualization techniques, target various application scenarios, and have different levels of isolation and performance.

The rest of this dissertation will, however, focus only on system virtualization. With system virtualization, the host operating system is the one which is running the virtualization mechanism, either as a user process or kernel component. One exception to this definition applies to type 1 hypervisors (see section 2.3.5) where the hypervisor is independent of any OS. The guest operating system, consisting of the guest kernel and applications, is running on top of a virtualized processor, memory and hardware. The following sections describe the different techniques used in system virtualization.

2.3.1 Binary translation

In a general case, the virtualized guest has a different instruction set as the host. Thus, the most straight-forward approach is to interpret each instruction of the virtualized guest cycle-by-cycle, and modify the state of the virtual Central Processing Unit (CPU), respectively, just like the microcode in a CPU core. For the special case where the guest and host have the same instruction set, an optimized approach can be done by running parts of the guest code directly on the host processor, and intelligently interpreting or modifying parts of the guest code to prevent the guest system from exiting its isolated environment. This is demonstrated in the VirtualBox virtualization software [31].

2.3.2 Dynamic code translation

A more optimized approach to straight-forward binary translation, known as dynamic code translation, is conceptually similar to a Just-In-Time (JIT) compiler. In dynamic code translation, a block of code in the guest memory is interpreted into host instructions (possibly in parallel with code execution), and the translated block is cached to speed up interpretation during repetition loops. Like in optimized JIT compilers, these translated blocks can be parameterized with known runtime constants, such that they are more optimized than a generic translation. The Tiny Code Generator used in QEMU [17] is one such example.

2.3.3 Full virtualization

A distinction between the various virtualization techniques is the difference between full virtualization and para-virtualization. Full virtualization simply means that the entire virtual hardware is emulated, including the processor, its memory management functions and also all hardware devices. This usually means that any OS which runs on an x86 instruction set can run in the virtualized environment under the same set of virtualized hardware without modification. Thus, the greatest advantage of full virtualization is its ability to run unmodified OS. Before the introduction of hardware-supported virtualization for x86, this isn't possible without performing binary translation on privileged x86 instructions. This is mainly due to the fact that the x86 instruction set was not designed initially to be virtualized. Thus, the practical use of full virtualization on x86 platforms remains limited due to its high CPU overhead. However, with the recent hardware (processor) support for virtualization, the performance overhead for full virtualization has been greatly reduced.

2.3.4 Hardware-supported virtualization

Hardware-supported virtualization pushes the performance of virtualization even further, by providing direct support for virtualization on the processor. The guest system runs natively on the host processor, and control is returned to the host system only to handle special events such as interrupts, IO access, memory management instructions and other privileged instructions. This increases the performance of virtualization and also reduces the complexity of the virtualization mechanism.

Hardware-supported virtualization on the PC(x86)

On the PC architecture, hardware-supported virtualization was introduced by both AMD and Intel, albeit with incompatible differences in their specifications. The core idea revolves around defining a memory structure, known as the Virtual Machine Control Block (VMCB), which stores the hardware registers, internal states and also defines the exit condition to return control to the hypervisor. Figure 2.7 shows the 3-step cycle of hardware virtualization. Before virtualization mode is entered, the VMCB memory structure is configured with the necessary state of the guest machine and defined exit conditions. Then, a privileged instruction (VMRUN for AMD or VMLAUNCH for Intel) is called to enter virtualization mode. During this transition, the content of the VMCB are swapped with the host's (hypervisor) CPU registers and states. The guest virtual machine runs natively on the processor with the loaded state, in this virtualized mode. Virtualization exits either by calling the VMEXIT explicitly, or when an exit condition previously defined in the VMCB is met. In this case, the processor registers and state is swapped back into the VMCB, and control is returned to the host (hypervisor). Upon exit, the hypervisor usually performs management operations on memory or emulates virtual hardware, before the cycle is re-entered via step 1. Special

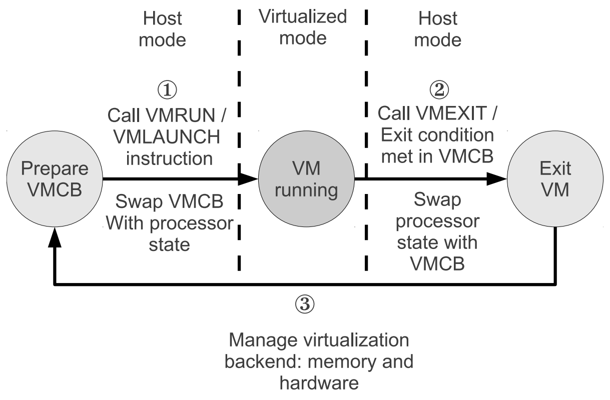

Host mode
Virtualized mode
Host mode
①
Call VMRUN / VMLAUNCH instruction
Swap VMCB With processor state
②
Call VMEXIT / Exit condition met in VMCB
Swap processor state with VMCB
Prepare VMCB
VM running
Exit VM
③
Manage virtualization backend: memory and hardware

Figure 2.7: Hardware virtualization cycle

care is taken to manage the caching of the translation look aside buffer (TLB), by assigning unique identities for each guest virtual machine on the TLB cache lines.

However, the frequent updates of the paging table in the guest, which have to be handled each time by the hypervisor, impose high overhead. In the second generation virtualization, the concept of nested page table (also known as extended page table) is introduced to resolve this problem. A nested page table defines a map from the guest physical memory address to the real physical memory address, thus reducing the need to switch between the guest and host to update the page table. Memory isolation of the hypervisor is also improved via the specification of an Input-Output Memory Management Unit (IOMMU), which explicitly controls the memory access from hardware peripherals to the main system memory.

2.3.5 Hypervisor architecture

In addition to the different techniques of performing the actual virtualization, virtualization systems also differ in their architecture, especially with respect to the presence of a host operating system. Figure 2.8 shows the 3 main virtualization classes, each of which is described briefly.

Type 1 hypervisor (bare-metal)

Type 1 hypervisors are virtualization kernels which run directly on the host hardware, and are also known as bare-metal hypervisors. Since there are no additional layer between the hypervisor and the hardware, type 1 hypervisors have direct hardware access and memory management functions (something which is usually done by an OS kernel), in order to work properly. To make the hypervisor small and robust, actual hardware management is usually carried out on behalf of the hypervisor, by one or more privileged guest OS.

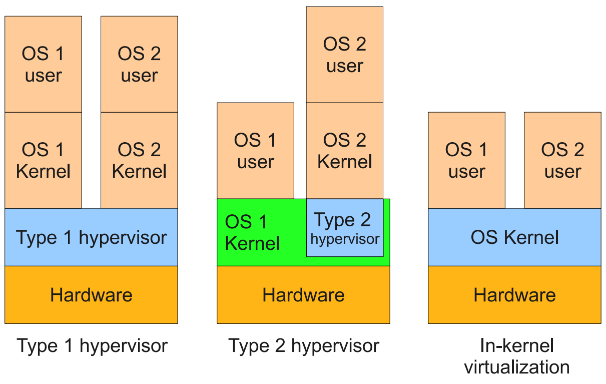


Figure 2.8: 3 main classes of virtualization

Type 2 hypervisor

Type 2 hypervisors are hosted or embedded into an existing OS kernel, so as to leverage the existing hardware and memory management functions within the host kernel. This reduces the complexity of the hypervisor, by re-using existing kernel functions. Having a host OS also allow tools to be easily deployed on the host OS to call management functions on the hypervisor.

In-kernel virtualization

In-kernel virtualization is a form of virtualization where the key kernel functions are logically separated into independent partitions, such that one such partition appears as an independent OS. Compared to the 2 previous schemes, this is not strictly speaking true virtualization as there is no underlying components which are virtualized. However, due to its increasing popularity, in-kernel virtualization has been seen as an alternative lightweight virtualization technique. Under this scheme, one or more partitions remain as a privileged partition in order to create, manage and destroy other partitions. In-kernel virtualization usually have much lower overhead as the bulk of the kernel remains unchanged, except that process trees and other related data structures are maintained independently for each partition.

Para-virtualization

Para-virtualization was first used in the XEN hypervisor, at a time where hardware supported virtualization was not available. It uses a modified Linux guest kernel, which is aware of the hypervisor API, to get around the limitations of slow virtualization on the x86 processor. This allows guest virtual machines to run natively (without translation) on the processor, while actively passing control over to the XEN hypervisor to manage memory on its behalf. Thus, the XEN hypervisor is able to control memory allocation of all its guest virtual machines, and apply the necessary isolation policies. Para-virtualization remains to date, the only method to run code natively on the processor (without any translation or emulation) on the x86 instruction set without using the hardware virtualization support. However, the disadvantage of this approach is that the guest kernel needs to be modified for the hypervisor.

A hybrid approach between full virtualization and para-virtualization is the use of para-virtualized drivers in the guest virtual machines. These drivers are specifically designed for virtualized guests and use a hypervisor interface which is usually more efficient and less complex than its real hardware counterpart. In general, they are able to achieve better performance and are thus used for high-throughput hardware devices such as storage, network and display. Virtio drivers [55] are a set of standardized virtual guest drivers and interfaces which can be used on multiple hypervisors and guest OSs. Para-virtualized drivers, however, do not imply that the guest virtual machine is running under para-virtualization. (For example, KVM uses full-virtualization while supporting para-virtualized Virtio drivers.)

2.3.6 QEMU and KVM virtual machine

As mentioned in 2.3.2, the QEMU emulator [17] uses dynamic code translation to achieve reasonable virtualization performance. The QEMU code has a framework that targets multiple guest instruction sets, including x86, PowerPC, ARM and other processors. It also has support for a variety of virtualized hardware configurations. In addition to full-virtualization, QEMU can also be used to perform process-only virtualization. The QEMU code is portable and can be executed on a variety of host platforms, since the guest instructions are translated into compiled host instructions. Thus, the QEMU is a flexible open source emulator, supporting a variety of guest and host targets. This dissertation focuses only on full-virtualization of 64-bit x86 guests on a x86 host.

The Kernel-based Virtual Machine (KVM) [16], started as a kernel module for enabling hardware-supported virtualization. This is necessary, due to the privileged instructions needed to perform context switches between host and virtualized modes, which can only be performed from within the kernel. The userland side of the KVM code has been integrated into QEMU, so as to perform hardware emulation only. The kernel module of KVM performs memory management and virtualization using the processor, while the userland component handles hardware emulation using the same QEMU base. This dissertation uses KVM as an example of hardware-supported virtualization of a 64-bit x86 guest.

2.4 Memory management on the PC

Memory management on the PC platform (32 and 64-bit) is based on the concept of paging, which is the mapping of units of continuous memory, known as a page, from a virtual address space to the physical address space. The physical address space is the actual physical address of the memory as seen by the external (front-side bus) interface of the processor. The virtual address space is the memory address as seen from the code execution perspective from within the processor. This mapping is handled in hardware by the processor, and is defined by the segment registers and the page table. The purpose of paging is to allow blocks of physical address space to be allocated in a non-continuous block, while being continuous in the virtual space.

Due to backward compatibility with the 8088 and 80286 processor, paged memory mode is only available after the processor enters the protected mode (32-bit). In addition to the page table, the segment descriptors perform an additional offset between the virtual and physical address space. Segmented memory is also available only under protected mode. However, in most modern OS, only the page table is used for memory mapping, but only a very straight-forward memory segmentation scheme is used, with one segment for kernel and another for user-space. The following sections will describe the 32-bit and 64-bit page table in greater detail.

2.4.1 Control registers

As in most other instruction sets, the x86 has general purpose registers, memory registers, status flag registers and so on. However, a special group of registers, known as the control registers, determine the specific mode of the processor. Due to backward compatibility reasons, the x86 processor has a variety of memory modes such as the 16-bit real mode (compatible with 80286), 32-bit protected mode and also the 64-bit long mode. The control registers control the transitions between these modes, enabling memory paging and other privileged operations. The x86 architecture has the control registers CR0, CR2, CR3, CR4, and additionally, the Extended Feature Enable Register (EFER), of which CR2 is used for storing page fault address and CR3, for storing the base physical address of the page table. The other registers are used for controlling processor mode and options. For compatibility, both 32-bit and 64-bit capable x86 processors always start in the 16-bit real mode and a series of transitions must be undertaken by the boot loader and kernel to reach the required memory mode.

2.4.2 32-bit paging table

When the processor enters the 32-bit mode, memory access can be either linear or paged. Linear memory access allows direct access to a maximum of 4GB of physical memory addresses. However, the paging mode is used more often, to provide a mapping between linear memory addresses (as seen by the processor) and physical addresses. This mapping is controlled by the OS kernel so as to map fragmented physical memory pages into continuous linear memory blocks, and to enforce memory protection.

The paging mechanism is organized into 3 levels, known as the Page Directory (PD), Page Table (PT) and a last 12-bit page offset, as shown in figure 2.9a (source [39]). The CPU Control Register 3 (CR3), stores the physical address of the base of the page directory. To translate a linear address into a physical address, the page table is traversed. The most significant 10 bits of the linear address are used as the index into the page directory, which has exactly 2^{10} Page Directory Entries (PDE). Each PDE is a 32-bit value containing usage flags and the physical address to the base of the associated Page Table (PT). The next 10 bits in the linear address are used as an index into this page table, which also has exactly 2^{10} Page Table Entries (PTE). The PTE is similar to the PDE and is a 32-bit value containing usage flags and the physical address to the base of a 4 kbyte memory page. The least significant 12 bits are used as the offset into this 4 kbyte physical page. The linear address is mapped to this physical address. Furthermore, read and write access to this page is also controlled by the page attributes within the PDE and PTE linked to this page. There are variations to this scheme with page size of 2 Mbytes and 4Mbytes, which are not shown in figure 2.9a. (see [39])

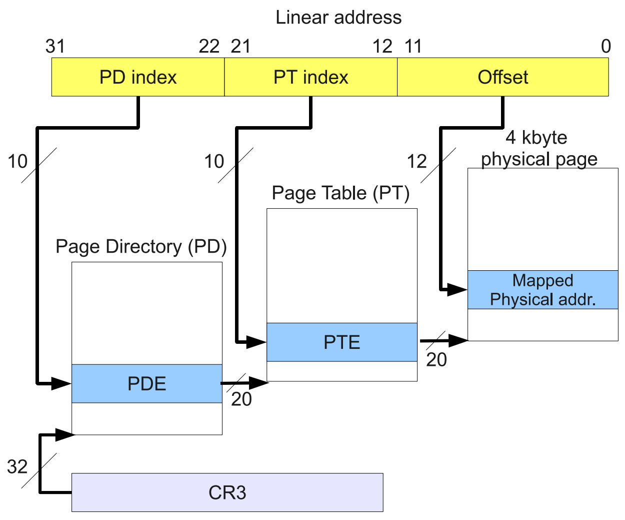
2.4.3 64-bit paging modifications

The 32-bit page table has a maximum addressable physical memory space of 4 Gigabyte (corresponding to 2^32), which is much higher than the usual amount of actual memory available at the time the i386 was introduced to the personal computer. However, with improved memory technology and decreasing cost of production, memory size was increasing and the 4 Gigabyte limit was starting to become a limiting factor to the amount of usable memory. In terms of security, the original 32-bit paging scheme allows pages to be marked read-only or read-write, but does not differentiate between code and data, meaning that there is no executable or non-executable page differentiation.

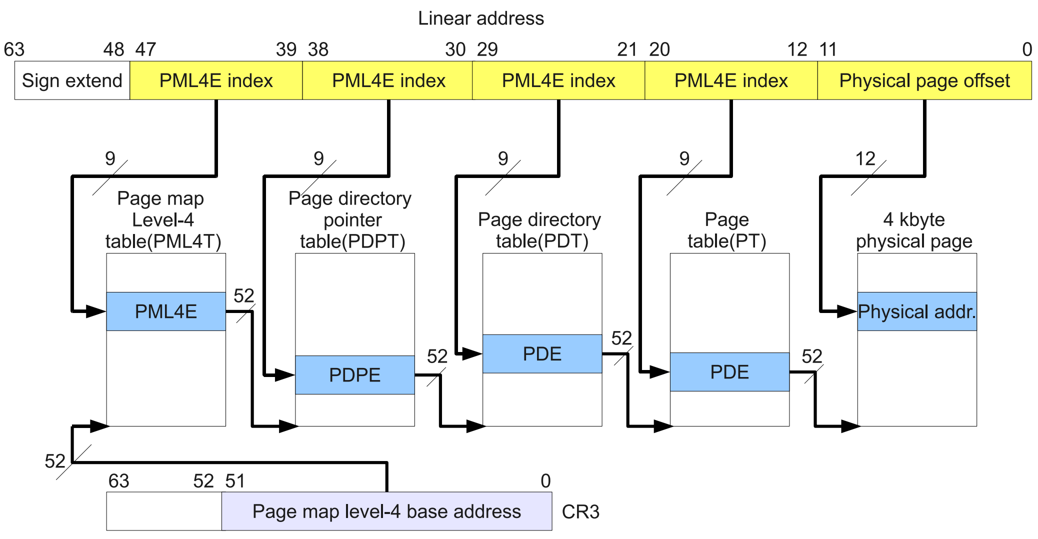
The first incremental modification to address the two problems is the Physical Address Extension (PAE) mode, which allows 32-bit linear memory to be mapped into the 52-bit physical memory. The PAE achieves this by introducing a fourth level table, the Page Directory Pointer Table (PDPT). This reduces the PD and PT addressing bits from 10 to 9 bits, while allowing each PD and PT entry to expand from the normal 32-bit into 64-bit entries. Thus, the full addressing space of the expanded PD and PT entries can address up to 52 bits of physical memory locations. Furthermore, an additional bit is added to the page attribute, which marks a page as non-executable, known as the No-execute (NX) bit. The PAE mode affects only the format of the page table, allowing existing 32-bit applications to work without modification. Thus, only changes to the memory management unit of the kernel need to be modified to support PAE.

Full 64-bit support was introduced in the ia64 and AMD64 instruction sets. The ia64 instruction set used by the Intel Itanium processor, is not backwards-compatible with the i386 instruction set. It removes all legacy functions included in the 80286 and i386 processors. The AMD64 instruction set, however, is a continuation of the i386, and introduces an additional 64-bit 'long' paging mode.

Figure 2.9b (source [39]) shows the 5-level paging used in the AMD64 64-bit paging mode. This paging mode has a 48-bit virtual address space, with a 52-bit architectural limit for physical memory addressing. The page table is traversed in a similar manner as before, albeit with more levels. The 64-bit CR3 now points to the Page-map level-4 Table



(a) 32-bit page table with 4k physical pages



(b) 64-bit page table with 4k physical pages

Figure 2.9: 32 and 64-bit page table formats

(PML4T), which provides the base address for the PDPT, which, in turn, points to the PDT and so on. The No-execute (NX) bit is also supported in the 64-bit paging mode.

The following chapters in this dissertation will focus on the host and guest operating systems using either the PAE or 64-bit paging mode. This is due to the support of the NX bit, which is crucial for the implementation of memory protection and detection of code execution in the guest operating system.

3 Integrity measurement

This chapter describes the proposed technique for integrity measurement as a core component of the integrity measurement framework. Integrity measurement will be used in this context to refer to the measurement and evaluation of the integrity of runtime code, against a given static reference binary. Since the motivation for this dissertation is to target existing operating systems, it is designed to be passive, requiring no modification to the operating system to trigger or support the measurement. Virtualization is used as a means to directly inspect the memory of the target operating system, in order to measure its integrity. This technique is generally applicable to different operating systems.

Section 3.1 discusses the threats which integrity measurement is meant to address, providing the basic assumptions and motivation of the work. In section 3.2, previous works in the field of integrity measurement and ideas that lead to the proposed concept are discussed. The sections that follow will describe the steps for integrity measurement in detail.

The first step in integrity measurement begins with building a pattern database (see 3.4) from the given reference binary of a target kernel. This is an offline process, which parses the compiled guest kernel and drivers, and prepares a compact database suitable for runtime verification. The second step involves detecting execution of new code (see 3.5) within the guest OS, by hooking appropriately into the virtualization layer. This detection triggers the third step of verifying the runtime code (see 3.6) against the reference database. Based on the verification, the integrity state of the guest OS is determined and reported via a virtual TPM.

3.1 Assumptions and Threat model

In order to achieve the goal of making trustworthy integrity measurements of the entire software stack (OS and applications), it would require a chain-of-trust from the hardware (as provided by the core roots-of-trust of the TPM) to the user application. Thus, possible threats would be attacks which can compromise any of the components along this chain. Since many applications already exist to protect (with the help of the kernel) user applications, integrity measurement in this dissertation will focus mainly on the threat of:

T1: malicious code change in the guest kernel

T2: malicious code change in the hypervisor or virtualization layer

Preventing or detecting threat T1 is the main goal of the integrity measurement framework, while T2 relates to the security strength of the virtualization layer itself. In this situation, four possible attack vectors are considered:

V1: exploiting the kernel system call or writing into the guest kernel memory

V2: exploiting hypercalls or interfaces to the hypervisor

V3: memory write into the hypervisor code or data

V4: pre-boot change to kernel or hypervisor (permanent change on hard disk)

Attack vector V1 corresponds directly to threat T1 and should be detectable by the framework. This is achieved by actively monitoring code pages in the guest kernel. Thus, any new page marked as an executable page, is being examined and reported to the integrity measurement module. Since the enforcement of page attributes is performed by the processor itself, the basic assumption here is that the CPU works as specified.

Attack vectors V2, V3 and V4 are three possible vectors of threat T2. Exploitation of the hypercall (V2) in virtual machine hypervisor can be reduced to a minimal by exposing as few interfaces as possible, to reduce the attack surface. As proposed in this dissertation, the only point of interaction between the guest kernel and the integrity measurement framework is through the emulated devices (virtual hardware and TPM).

V3 relates to the general problem of direct memory write into the hypervisor memory space either from software (processor controlled) or through hardware via the Direct Memory Access (DMA) channel, Peripheral Component Interconnect (PCI) bus or Bios functions. A software-based attack is prevented via paging control of the guest kernel, while hardware-based attack can only be prevented with an IO Memory Management Unit (IOMMU). In the absence of an IOMMU, this can only be prevented by virtualizing all busmastering hardware (e.g. DMA controller or PCI devices) in the system.

Attack vector V4 can be detected by having a continuous chain of hash measurement of each loaded component during the boot process, as described in detail in section 4.7.1. Thus, any modification to the host and its data will be detected as it results in a difference in hash measurement during the trusted boot process. An evaluation of various kinds of attacks is discussed in chapter 6.

3.2 Previous work in integrity measurement

3.2.1 Literature study

Early work on integrity measurement was developed along with the ideas of Host-based Intrusion Detection System (HIDS), which detects malicious software activity from within the host. This is in contrast to Network-based Intrusion Detection System (NIDS), where detection of an attack is carried out on the network level via the network infrastructure or a dedicated device on the network monitoring network traffic and patterns. Tripwire [43] is a well-known example of an HIDS, where files on a host are regularly checked against a known configuration of all critical files. Deviations from this configuration will invoke an alert to a centralized server. This allows network administrators to configure a system securely, and ensure that the configuration is not altered by malicious software or users. The advantage of HIDS over NIDS is its increased visibility into the individual processes and resources on the host system, identifying problems at its source.

HIDS itself is, however, more vulnerable to attacks from malicious software or users. This has led to the use of virtualization as an isolation mechanism between the HIDS and the target system which runs as a guest operating system [38]. Similar ideas were applied to high-interactivity honeypots [57] and high security systems [47], where virtualization is used to isolate the susceptible honeypot or a vulnerable user environment, from the monitoring system which resides in a secure virtualization host system.

Even though virtualization is hardly a new concept, being used on mainframe processors from the 70s to support legacy software and multiple OS instances, its usage on non-mainframe systems was limited, due to the high performance overhead of virtualization. But with increasing processor speed and eventually, hardware-supported virtualization, more virtualization systems and hypervisors (see section 2.8) were developed for different processors. Following this trend, integrity and security-focused virtual machines and hypervisors were also developed.

One example is the nickle [54] virtual machine, based on QEMU virtual machine [17], which attaches itself to the dynamic code translation mechanism of QEMU to check for code changes in kernel memory space. This approach is very flexible and has been shown to apply well to different virtualization tools as well. However, the implementation is limited to software-only solution, and does not utilize hardware-supported virtualization which has become widespread in the past years. Patagonix [44] is a XEN [19] based hypervisor using hardware virtualization. It performs integrity measurement of kernel and application space binary code. Argos [53] is another modification to QEMU, targeted at honeypot implementation, to correlate buffer-overflows to their respective incoming attack vectors.

Bitvisor [62] and Secvisor [21] are two hypervisors, with a small code footprint, focused on security applications. Both are using hardware-based virtualization provided by Intel and AMD, thus making the hypervisor much simpler than software or para-virtualization solutions. Bitvisor implements mandatory encryption by virtualizing specific hardware drivers to provide transparent disk and network encryption. While Secvisor is focused on protection of kernel space memory by enforcing mandatory W$\otimes$X mode and kernel code measurement. W$\otimes$X is a security feature first implemented in the OpenBSD[4] operating system. It is a memory management policy that enforces a restriction that memory pages are either executable or writable, but not both. This prevents attacks based on buffer and heap overflow, and is easily implemented on processors supporting non-execution page properties.

In addition, Linux Kernel Integrity Measurement (LKIM) in [46] is proposed to directly monitor the memory of a guest virtual machine, in order to determine its integrity state.

This allows for the integrity of the guest machine to be continuously monitored, throughout its entire uptime.

3.2.2 Debug stub-based integrity measurement prototype

An initial prototype for integrity measurement was developed along with similar concepts proposed in [46, 38]. The prototype uses the remote Gnu Debugger (GDB) interface provided by QEMU, to insert breakpoints into the guest virtual machine. The breakpoint locations are chosen using the symbol table generated during the compilation of a Linux kernel. Thus, by placing breakpoints at the kernel system call address, which is the main entry point from user to kernel space, all interactions between user and kernel, can be inspected. This allows for an in-depth analysis of the calling process, function arguments and respective kernel data structures.

A disadvantage of this approach is that it is inherently slow, due to the insertion of breakpoints which have to be checked at every translation cycle. Furthermore, the computation of breakpoint addresses is derived from the debug symbol table of the Linux kernel. The debug symbol table is a compiler generated table specifying address locations of global variables and functions. Thus, the use of the debug symbol table makes the approach version specific. Another major drawback is that this method only considers specific entry points into the kernel. This does not cover the entire attack surface of the kernel, and furthermore, kernel entry points may change with each kernel interface update.

3.2.3 Nickle as a base concept

Due to these inherent drawbacks, a revised integrity measurement concept was based on nickle [54], which also uses the QEMU virtual machine (version 0.9.0). Nickle inserts an additional hook into QEMU's instruction translation cycle to perform comparisons between the code byte which is currently being executed, and a verified code store. The verified code store mirrors the guest virtual machine's physical memory, with the restriction that write access to the verified code store has to be verified. The verified code store starts out completely empty, and each discrepancy between the code store and the physical memory at the currently translated instruction, implies that newly loaded code was executed. Only execution at the ring 0 privilege is considered. In the case of the Linux OS, this usually means that a new kernel module was loaded into memory. The specific kernel module that is loaded is identified and verified. If the verification is successful, the newly loaded code is copied into the verified code store.

3.3 Outline of integrity measurement

Based on previous work, the overall goal of integrity measurement is to verify if all code execution within a guest virtual machine is confined to a set of defined binary executables.

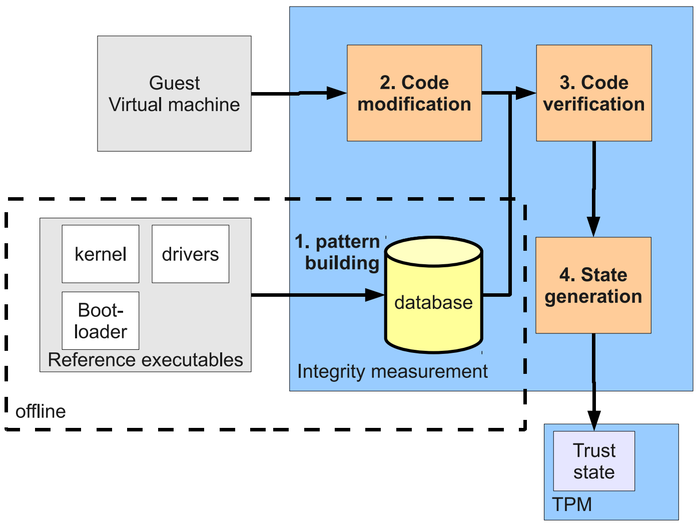


Figure 3.1: Overview of integrity measurement

The output of the framework is an integrity state of the guest OS, indicating if altered kernel components have been loaded.. Figure 3.1 shows a conceptual flow of the integrity measurement process.

The entire process has an offline and online step. The first step, pattern building (see section 3.4), is an offline process which constructs a reference database from a given set of binary executables. The database is used later for an efficient verification process. At the second step, the target guest virtual machine is monitored (see section 3.5), to detect new binary code being executed in the guest OS. For each block of code that is loaded and awaiting execution, the code is verified using the database (see section 3.6) in step 3. Finally, based on the verification, the current integrity state is decided and reported (see section 4.6) to the TPM. The initial concept of this mechanism was published in [59].

3.4 Pattern building

The proposed integrity measurement differs from typical code signing verification methods [37], in that detection is not performed on a file level, but rather, in memory. This provides the advantage of being independent of the security mechanism in the guest kernel, and is applicable even while the kernel itself is being started. In-memory verification is, however, more complicated as the loaded image of a binary object differs from its file representation.

Thus, the pattern building step is an offline method that reconstructs the in-memory image of the binary, using the same algorithm used by the kernel. The process is OS specific, but is simplified by the fact that the way binary object files are loaded into memory is usually clearly specified (based on the executable binary format) with minor exceptions, depending on the actual operating system. A key problem in constructing an offline reference is the dynamic loading of the image at an allocated base address. Except for the trivial case of the very first loaded binary, the base address where every other module is loaded cannot usually be predicted, as it would depend on the runtime memory allocation. Thus, the offline pattern has to be flexible enough to adapt to a changing base address.

On the PC platform (i386 and amd64), operating systems use the Memory Management Unit (MMU) of the processor to map virtual into physical memory pages. The PC platform has a minimum page size of 4k bytes, and this is the smallest allocatable page (See section 2.4). Thus, most kernel binaries are designed to be loaded on a page-aligned base address, with the sections being continuous in virtual memory. This has been verified to be the case for Linux, freeBSD and 64-bit windows kernels. This assumption simplifies the image construction process and allows for some optimization in the verification process.

Thus, the invariant and variant parts of the image to be verified are handled separately. The invariant part remains unchanged, while the variant part is modified by the OS during loading, depending on the base address and external memory references. The following section 3.4.1 will explain the two common executable binary formats, so as to gain an understanding of binary relocation, import symbols and how the variant pattern is built.

3.4.1 Executable binary formats

Executable and Linkable Format (ELF) and Portable Executable (PE) are 2 commonly used binary code container file formats. ELF is widely used on modern UNIX-like systems including Linux and BSD, and is designed to be completely position-independent (binary code can be loaded easily at any base memory location). The need for additional information to support position independence arises from the fact that absolute jumps to 32 or 64-bit locations are necessary in compiled code, as a relative (to the current instruction pointer) jump is limited to single byte offsets on the x86 architecture. ELF handles all position-dependent jumps by marking the positions of such jumps within the compiled code in a relocation table.

In addition to relocation of the jump location within the same binary code, both formats support modification of the appropriate memory address of function entry points, external to the binary code, such as external libraries. This is defined in the export table of the file, which indicates the required external functions and the locations to be modified within the binary code.

3.4.1.1 ELF Format

The ELF format is the file format used for all binary code files on Linux and BSD-based systems, such as executables, kernel modules (.ko), shared libraries (.so) and linkable object files (.o). ELF format itself is cross-platform, and can contain code for different instruction sets and OS as indicated in the ELF header. The ELF format can be used both as an object file in the linking process or as an executable and shared object file format. Therefore, it supports both a linking and an execution view.

Section type	Section name in file	Typical section attribute	Purpose
Code	.text	A, X	Stores compiled code
Read-only data	.rodata	A	Stores read-only data. This is usually a storage place for constants defined in a program
Read-write data	.data	W, A	Stores read-write data, with initialized values. This is usually used for global variables with pre-defined initial values.
Empty data	.bss	W, A	An uninitialized memory space is defined. This is usually used for uninitialized variables.
Relocation data	rel.<section name>	A	This is a table defining the relocation information to another section

A = Allocate memory for this section
X = Executable data
W = Writable data

Table 3.1: Common sections in an ELF file

Figure 3.2a shows a typical ELF format. It always starts with an ELF header, followed by a program header (required for execution view). Together, they provide general information on the file, platform and pointers to the important sections in the file. Commonly defined sections are those used for storing code, read-only data, read-write data and empty data allocation block. Table 3.1 shows a few commonly found sections. In addition, special sections are used to store section names, symbol names, relocation addresses, debug information and so on. Each section has defined attributes to show how the section should be handled in memory.

3.4.1.2 PE Format

The PE format has a similar structure, but is used mainly on Microsoft Windows, Microsoft Windows Mobile and Microsoft's Common Language Runtime (CLR) based systems. One major difference to the ELF format is that the PE format assumes a fixed based position for each piece of binary code. Relocating a PE binary is achieved by a 'rebasing' process, where the code is parsed and absolute jumps are modified based on the difference between the desired base location and the original base location.

3.4.2 Code loading in Linux

Figure 3.2a illustrates how an ELF binary is loaded into memory. When the binary code is loaded in a system (either executable or shared library), the binary code required for binary execution is copied into memory, discarding unnecessary information (such as headers, tables, embedded debug information, meta-data, etc.). After that, the jump addresses are fixed for the relocated memory location and also, to externally linked libraries. The binary loader, usually in the OS kernel, reads the ELF file and its respective headers, and extracts the position of each execution-related section. Each section is tagged with flags, but the main section types are executable (read-only), read-only data, read-write data and uninitialized data. The binary loader places the sections in a newly allocated memory block, and performs relocation and linking of libraries.

A specific feature of Linux kernel modules is the differentiation between an initialization and a main section. The former allows initialization functions which are run only once during module start-up to be declared in a separate code section. To save memory, this section can then be unloaded after the module has started. Thus, the Linux kernel makes two separate memory allocations for initialization and main sections.

Within each allocated block, sections such as those defined in 3.1, are packed sequentially in the following order:

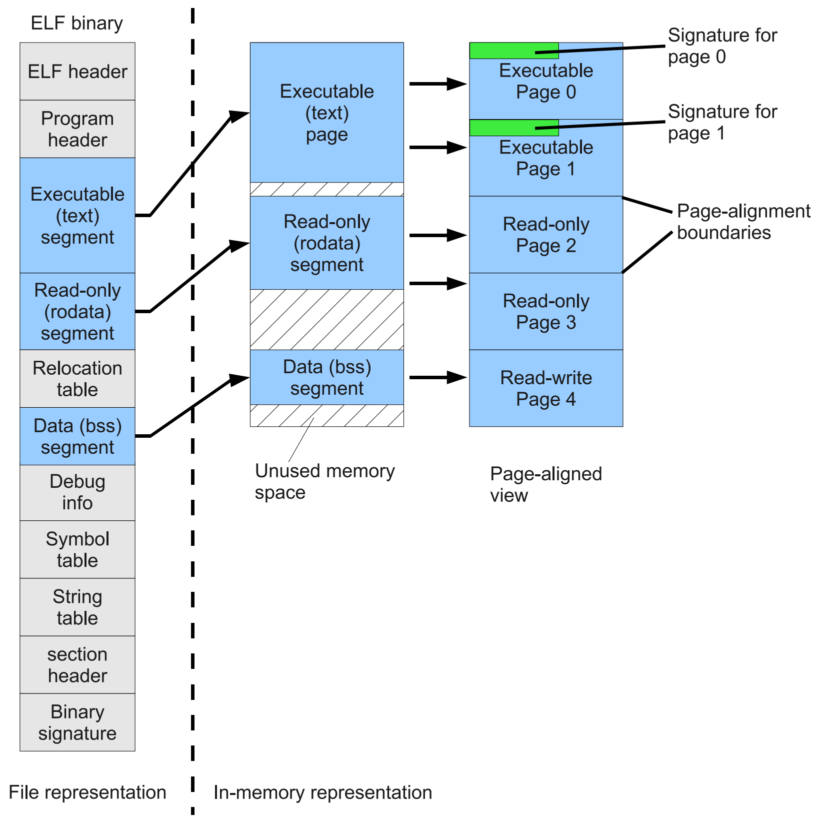
1. Read-only and executable (such as .text)
2. Read-only data (such as .rodata)
3. Read-write data (such as .data and .bss)

When multiple sections with the same attributes exist, they are ordered based on their order of appearance in the original binary file.

3.4.3 Signature database

The relocation and linking process of an ELF-formatted binary leads to differences between its binary content on file, and its corresponding content after being loaded completely in memory. This is illustrated by the rightmost representation of figure 3.2a. The 4 main differences are:

1. only execution-related sections that are eventually loaded into the in-memory binary image (file and section headers, tables, debug information, meta-data and the like, are discarded)
2. memory locations within the binary that are corrected based on relocation (relocation)
3. memory addresses of external library functions that are written into the image (linking)



(a) File to in-memory of ELF binary

Page signature	Module name	Module page index
<Sig 1>	driver1	0
<Sig 2>	driver2	0
<Sig 3>	driver2	1

(b) Signature database

Figure 3.2: Binary measurement

4. each individual section in the ELF that is loaded, starting on a new page (alignment)

Based on these observations, an efficient method to detect the loaded module is developed, without further introspection into the kernel. This method utilizes a simple page signature for each loaded page. Since the sections are loaded on page boundaries, this page signature would be invariant to the dynamic base address where the module is loaded. For simplicity, a short fixed length signature from the start of each page is used as the page signature. This is shown in figure 3.2a, where the in-memory block is divided into its respective pages.

In order to compute these signatures, the given ELF binary is extracted in memory in the same order performed by the kernel. Then, signatures of code pages associated with code sections are collected into a signature database. For each database entry, a reference to the original ELF binary and the index of the page are also recorded.

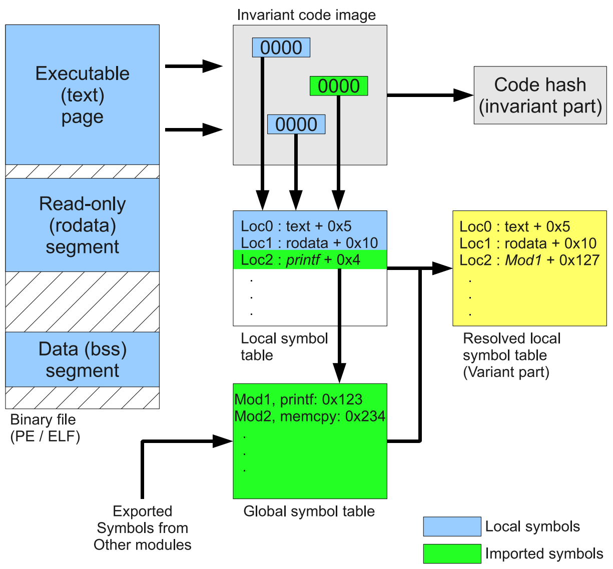
Table 3.2b shows a simplified view of the information saved in a signature database. For each module, multiple page signatures are extracted for each page that code sections of the module span. By using the signature database, it is now possible to efficiently find the closest match of a page from some newly loaded kernel module, to the most likely entries in the database.

3.4.4 Module database

The page signature is only an optimized approach to finding a list of possible matching modules, without verifying the integrity of the entire module. Therefore, in order to verify its integrity, the rest of the module content must also be checked. This is achieved by constructing a projected in-memory image of the loaded binary from the file representation, as shown in figure 3.3a. Since the relocation address and external library addresses cannot be known beforehand, a simple solution adopted from [54], is to ignore those bytes by inserting a string of zeros at each of these memory locations. By ignoring all dynamic parts, both the constructed and live in-memory image should be identical. Thus, a cryptographic hash can be used to efficiently represent the invariant portion of the image. This code hash can be used to verify the integrity of the live in-memory image. Table 3.3b shows a simplified view of the information collected in the module database, which consists of the module name, size, relocation information extracted from the binary and the code hash. By considering the symbol table of all modules, all relocation addresses can be resolved to either a local offset, or an offset to another module. This is covered in detail in address resolution in section 4.2.1. The module and page database together form the pattern database, generated as a result of the offline step of integrity measurement.

3.5 Detection of code modification

Detection of changes to code pages is a key step in protecting the kernel level code. This step detects the new code executed in the guest kernel, or detects changes to previously



(a) Pattern decomposition

Module name	Module size	Relocation positions and meta-data	Hash
driver1	0x1700	0x10: driver1 + 0x0005 0x20: driver1 + 0x1010 0x30: driver2 + 0x0127 ...	<hash 1>
driver2	0x3000	...	<hash 2>

(b) Module database

Figure 3.3: Pattern building process

verified code, forcing it to be re-verified. Thus, this detection is a key step in detecting malicious changes to the guest kernel.

An important point to note is that, integrity of data (not code) is not checked. This is due to the transient nature of data, and the complexity involved in understanding the intrinsic structure of data; it is deemed to be outside the scope of this dissertation and deserves to be handled as a topic in itself. Unlike data, code is almost always unchanged once it has been loaded into memory, except for the case of self-modifying code or code patching. Code patching refers to modifying parts of the existing loaded code to modify its behavior. It is a technique used to insert additional features to the system, or by malicious code to attack a running system. Code patching is done by some anti-virus software on the windows kernel to add special functions necessary for the anti-virus software to work. The Linux kernel also uses code patching to add the optimized assembly code into the kernel based on the detected CPU model. (See section 4.2.1 regarding code patching)

The most common approach to verifying code is to verify its integrity at the point of loading [37]. This is usually performed by the kernel, when loading a binary code into kernel or userspace memory. However, the verification is usually done at the file level, meaning the binary file is signed and its signature is verified (such as using public key encryption based certificates), before loading the binary.

The file-based code verification feature is already available on many existing operating systems, but the main difference to the proposed detection method is that code memory detection is done completely independent of guest kernel. This detection scheme is performed by the host introspecting into the virtualized guest memory area. This provides the additional assurance on the strength of the measurement, since effectiveness of the detection is independent of the integrity state of the guest OS and its security mechanism.

Furthermore, file-based verification is performed only at the time when the binary is loaded, while memory detection is an on-going process which detects code injection into memory over the entire uptime of the OS. Memory detection will identify code modification or patching which can occur due to malicious code trying to modify the kernel behavior. This can occur when flaws in the kernel or any of its loaded drivers contain vulnerabilities which can be exploited to bypass the code signature check altogether.

The actual technique used for detection of memory modification is closely tied to the virtual machine's implementation. Two methods are discussed in this dissertation.

3.5.1 Detection during code translation

The first approach is integrated directly on the translation cycle of the QEMU virtual machine. Figure 3.4(a) illustrates the steps which QEMU takes to translate and execute a set of instructions in the guest virtual machine. QEMU translates each instruction from a block of code in the guest memory (usually terminated by a jump instruction), into a block of code executables in the host environment, known as a translation block (TB). To avoid re-translating frequently used blocks, this translation block is stored into QEMU's translation cache. Within the translation cache, the end of each translation block is linked to the start of other translation blocks (depending on the result of the block), so that re-translation is minimized. Infrequently used TBs are removed from the cache when it gets filled up, so that frequently called code remains in the cache. Changes to the source block in the guest memory also cause the corresponding TB to be flushed from the cache, such as in the case of self-modifying code. The QEMU execution environment on the host also contains a processor state, which is a representation of registers and state of the emulator processor. Each execution cycle of a TB uses and modifies this emulator state. Figure 3.4(b) shows the translation process of QEMU in a flow chart form.

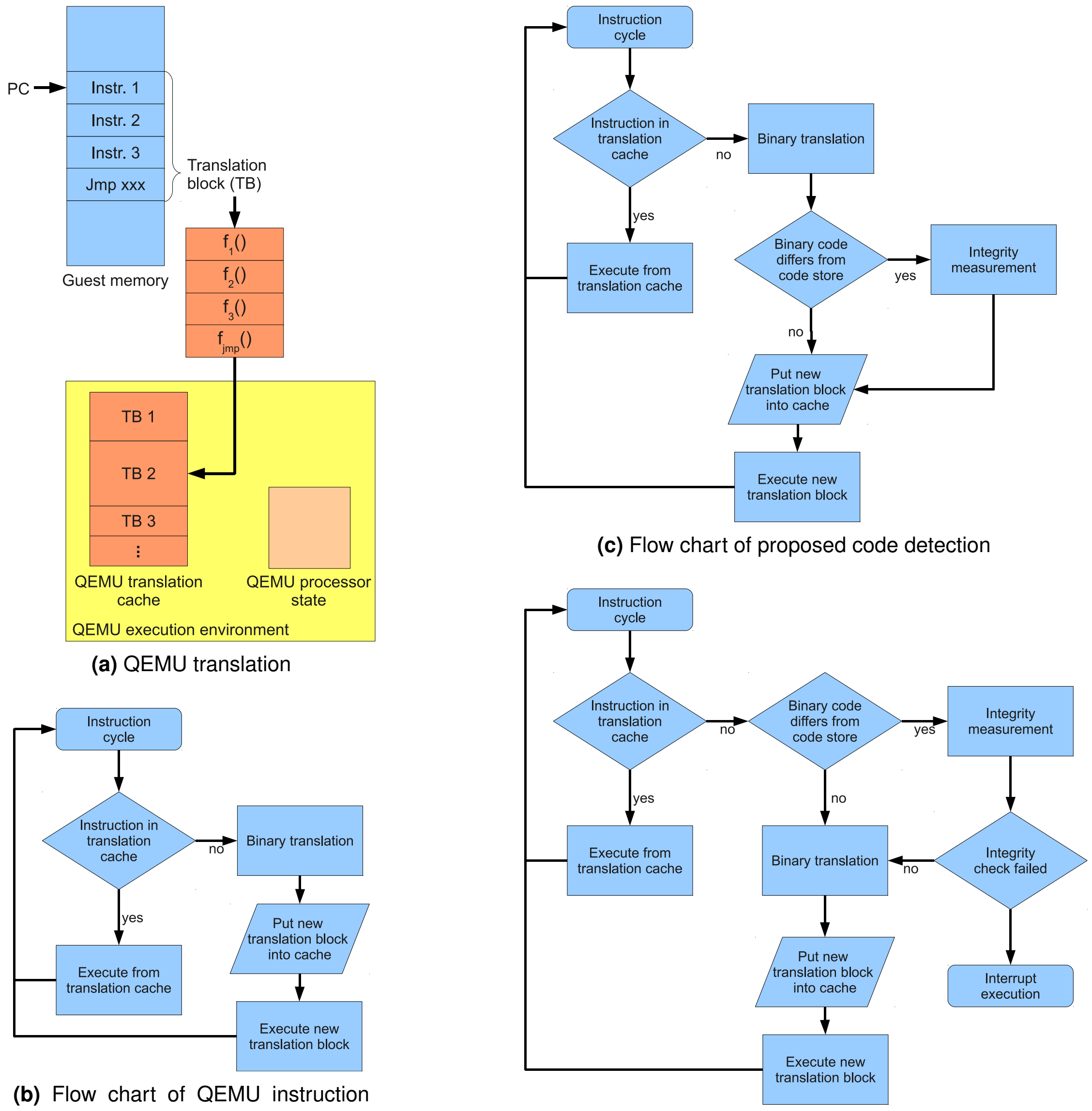
Code change detection is achieved by checking if a TB is part of a previously verified code module. In order to avoid performing verification for every new TB, a verified code store is used to cache code that has already been verified. This code store acts as a mirror of the guest's physical memory, but contains only verified code. Whenever a new block of code is verified, it is copied into the code store. The code store is then used as a reference to check if subsequent TBs are verified.

Figure 3.4(c) illustrates the flowchart of the proposed code detection scheme. After a new block of code is translated, the source block is compared with the code store to check if it has been previously verified. If they are identical, the TB is accepted as verified code and execution proceeds. However, if they differ, or if no previous verification was performed at that memory location, the verification process is invoked. Upon positive verification, the block is copied into the code store before proceeding with execution. A failed verification would mark the guest's integrity state as unverified, before proceeding with execution. Translation blocks executed directly from the translation cache do not require re-verification, as they cannot be modified by the guest.

Comparing the integrity measurement framework to Nickle

As mentioned in 3.2.3, the proposed code detection scheme was developed, starting from the concept proposed in nickle [54]. Improvements were developed to address the shortcomings of the nickle implementation. Figure 3.4(d) shows the flowchart of the nickle code detection scheme, which differs in its operation flow, compared to the proposed scheme in 3.4(c).

The new scheme was proposed to address some of the problems identified in nickle's code detection. Firstly, nickle uses values stored in the registers (which are usually associated with function parameters) as a means to identify the code which is currently being loaded. This approach works because under Linux, new kernel modules are loaded by the module loading function which also calls the module initialization code. By means of reverse engineering, the memory location of the structure containing the module name can be found from one of call arguments (usually stored in a register). However, this approach is highly dependent on the operating system's code, including its specific version and the compiler used (the actual choice of register for function argument is determined by the compiler). Thus, this identification method is not easily applicable to other operating



(a) QEMU translation

(b) Flow chart of QEMU instruction translation

(c) Flow chart of proposed code detection

(d) Flow chart of nickle code detection

Figure 3.4: Comparison of code detection process

systems, and generally impossible for closed source operating system such as Windows, without knowledge of its internal memory structures.

The proposed integrity measurement scheme identifies new code not by specific registers or memory location, but rather, the newly loaded code block itself is identified via page-aligned signatures. This approach is thus independent of the specific kernel module loading scheme. It allows the detection scheme to be applied to different operating system easily.

Furthermore, nickle allocates a verified code store which is as large as the guest's physical memory. This code store is inefficient, as most of the allocated memory is mostly redundant, as only a mirror of the kernel code memory is required. In a modern system, the kernel code memory usage is extremely small (order of 10MB for Linux), compared to cache, data and user code memory (of the order of gigabytes). This problem is resolved in the new scheme via a memory data structure which only stores pages which have verified code. Each page is tagged with a physical address, so that it can represent a sparsely populated physical memory space.

Conceptually, the proposed scheme aims at providing a trusted integrity measurement without any form of intervention from the guest system. Thus, a combination of the hardware TPM and a virtual TPM provides a complete measurement chain of the system hardware, the host system, and also the guest kernel and applications. The passive behavior (no intervention) of integrity measurement allows for maximum flexibility of software execution inside the guest environment, while trusted applications (on a clean system boot) can co-exist with secrets being bound to the trusted integrity measurement.

3.5.2 Detection via memory management unit

As introduced in 2.3.4, hardware-supported virtualization gives a much better virtualization performance compared to binary translation. Thus, an alternative code modification detection scheme is developed for the case of hardware-supported virtualization. This implementation is based on Kernel Virtual Machine (KVM), which shares some common code with QEMU for hardware emulation. Both implementations share the same userland code base.

No binary translation is done for KVM, but rather, the guest code is executed natively on the processor. As described in section 2.3.4, the hardware virtualization cycle involves the management of memory, hardware virtualization and a VMCB state for each virtual machine. The main task of the kernel component of KVM is to perform memory re-mapping between the guest memory address space and the actual physical address space. Memory re-mapping is necessary to confine the guest's memory access to its allocated block, so as to cleanly isolate the guest memory from the host and other guest machines.

3.5.2.1 Shadow paging

One method of performing memory re-mapping is to use the well-established shadow paging technique. Shadow paging maintains a host-controlled page table, which maps guest linear address into machine physical address. This page is synchronized with the guest's page table, albeit with an offset into the memory allocated for the guest virtual machine.

The software memory management unit of KVM handles the synchronization of the guest memory allocation and the real memory allocation specified in the shadow page table. Figure 3.5 shows the general idea of a shadow page table. The guest operating system controls the guest page table, which is used to map the guest linear memory space into the guest physical memory space. The guest physical memory space does not correspond directly to physical memory of the host, but rather, to the block of memory allocated to the guest by the host. Thus, the guest physical memory has to be mapped onto real (host) physical memory space through memory management functions. Figure 3.5 depicts a continuous block of memory allocated to the guest, but this is a simplification and is not necessarily true in general, since defragmentation of the allocation occurs after virtual machines are started and stopped.

Standard shadow paging

As seen in figure 3.5, the shadow page table actually by-passes the guest page table completely, and maps the guest linear memory into real physical memory space. The guest page table is, however, still important as it indicates the 'intention' of the guest operating system. Thus, the basic mechanism of shadow paging works by capturing memory allocation events as they occur in the guest virtual machine, and updating the shadow page table, respectively.

The shadow page table replaces the guest page table by controlling the guest's read and write access to the CR3 system register, which stores the base address of the page table. The guest virtual machine is prevented from updating the CR3 with its own page table, but rather, the shadow page table is loaded instead. Thus, when a new CR3 is loaded by the guest operating system, the guest page table is parsed completely, and the shadow page table is populated with the same structure, except that the physical address of each entry is resolved to a location on the real physical address allocated to the guest.

When a page fault occurs in the guest, control is returned to the host and the source of the page fault is determined. Four possible situations may arise with page faults in the guest:

1. If the faulting address is not present, writable (on write attempt) or executable (on execution attempt) in the guest page table, this fault is forwarded back to the guest operating system by re-injecting it into the guest machine.

2. If the faulting address is present in the guest page table but not in the shadow page

table, it implies that the guest has allocated a new page table, and a corresponding entry is inserted into the shadow page table.

3. In order to update the 'dirty' bit information on the guest page table and internal KVM 'dirty' bitmap, such pages are marked initially as read-only, so as to capture attempts to write to the memory address in the shadow page table.

4. The faulting address is a Memory-Mapped Input-Output (MMIO) location, and thus, the read or write access has to be passed to the hardware emulation layer.

In addition, shadow paging also has to handle the 'invalidate page' (INVLPG) instruction from the guest. This would normally invalidate an entry on the cached page table, which would cause the corresponding entry in the shadow page table to be deleted. The above-mentioned algorithm is a general overview of the shadow paging mechanism implemented in KVM. There are slight variations in shadow paging implementations in other hypervisors such as XEN or Virtualbox.

Modified shadow paging for code detection

Based on the standard shadow paging algorithm, the page synchronization can be modified to perform code detection using the following rules. Memory page with the executable attribute will be associated with the following shadow pages:

- Executable page: Read-only and Executable
- Writable page: Read-write and non-executable

Whenever a transition to an executable shadow page is made, the code verification process is automatically invoked. This method protects verified code from being modified. Performing a write to such code pages would invoke a re-verification, and next time, code would be executed from that page. Using this simple concept, the shadow paging algorithm can be modified to detect new code pages in the guest virtual machine (VM). Details of the implementation are discussed in section 4.3.

3.5.2.2 Nested paging

Nested paging (also known as extended page table from Intel) is the hardware supported memory management supporting virtualization directly. With a processor supporting nested paging, an additional nested page table defined a memory map between the guest physical memory and the real physical memory. This eliminates the need for a shadow page table. Nested paging brings about significant performance improvement, since the handling shadow page table updates, being one of the major overhead, are now performed in hardware.

However, since the entire guest physical to real physical memory translation process is now handled by the processor, there is no easy hooking point to detect new code pages

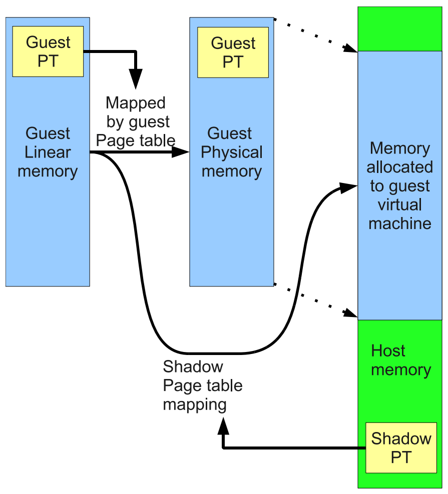

Guest PT
Guest Linear memory
Mapped by guest Page table
Guest PT
Guest Physical memory
Memory allocated to guest virtual machine
Host memory
Shadow Page table mapping
Shadow PT

Figure 3.5: KVM Shadow paging

used by the guest machine. Furthermore, incompatible differences between the Nested Page Table (NPT) from AMD, and Extended Page Table (EPT) from Intel, force any possible solution to be processor specific. Thus, there is, currently, no proposed solution for code change detection for the case of nested paging.

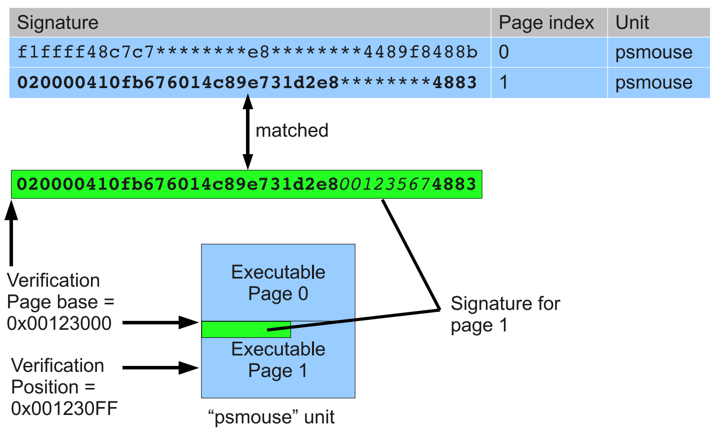
3.6 Code verification

This section describes the code verification process which can be depicted as a 3-step procedure. The first step is to identify a list of possible modules that can match the target memory area. For each of the possible matches, the memory block is verified against its invariant portion and also the variant portion (such as dynamic memory addresses and code patches). The 'search and match' processes are based on the database derived in section 3.4.4. Finally, the resulting verification state is determined by the matching pattern. A special unverified state is entered if the code cannot be matched to any known patterns. Whenever a state transition occurs, the new state is reported to the TPM via trusted reporting described in section 4.6.

3.6.1 Pattern matching

In section 3.4.4, the pattern database consists of a signature database and a module database. Each signature entry contains a fixed length pattern, which is the page signature at the beginning of each page boundary of the module's image. Specific portions of the pattern are also masked out based on the variant portions of the module.

Thus, whenever a new code block (via translation in 3.5.1) or new code page (from



Signature	Page index	Unit
f1ffff48c7c7********e8********4489f8488b	0	psmouse
020000410fb676014c89e731d2e8******4883**	1	psmouse

Figure 3.6: Matching memory to signature database

memory management in 3.5.2) is detected, the fixed length signature at the page boundary of the new code page is extracted and matched against the signature database.

Figure 3.6 illustrates the matching process to find a possible module. Generally, the verification position can occur anywhere in memory. The page boundary corresponding to the verification position is known as the verification page base, which is computed by ignoring the lower 12 bits of the address. A fixed length of 20 bytes is extracted from this position, and checked against the pattern database. Asterisk (*) values in the pattern database are used as placeholders for ignored byte positions in the pattern. These correspond to dynamic data within the signature. The matching process returns a list of matching modules and their respective page offset from the module's starting address. For each potential match, they are further verified by checking its invariant and dynamic content as described in sections 3.6.2 and 3.6.3.

3.6.2 Verification of invariant memory area

The binary code, when loaded into memory, consists of pockets of dynamic areas due to relocation and linking of modules. However, the bulk of the binary code is static and can be easily verified in a compact form using a cryptographic hash. Thus, to first ignore the dynamic portions, a copy of the entire module is made in memory, and each dynamic byte location is overwritten with a zero value. A hash is then calculated over the entire image. This makes the hash independent of dynamic variations in the image. The computed hash is then compared with a pre-computed hash (calculated in the same way) that is stored in the unit database. An identical hash would mean that every invariant byte in the image is identical to the off-line reference used to build the pattern database.

Algorithm 3.1 Recursive verification

```
function VerifyModule( module )
    1. For each memory location l in module
        a) verify location l
        b) Return fail if verification fails
        c) if l depends on external module B
            i. call VerifyModule( B )
```

3.6.3 Verification of dynamic memory area

The next step in verification involves checking the dynamic portions of the target binary code. Each entry in the relocation meta-data of the unit database contains a relative relocation address, relocation type and associated parameters. Relocation and linking entries are generally classified into local (within the same unit) and foreign entries (referenced to another unit). Local relocations are verified against the base address of the unit while foreign entries are verified against the base address of another unit. If the dependent unit is not verified yet, it is recursively checked.

In addition to relocation addresses, kernel patching done in the Linux kernel is also considered. Since Linux kernel patching is done in place, there is no change in the code length before and after patching. Thus, both the original and patched version of the code is stored as 2 possible alternatives to a particular code string at a defined location. Details of the verification process are described in section 4.2.1.

Recursive verification problem

When performing the dynamic verification on units with dependencies, recursive verification is necessary when the verification order is reversed, compared to the dependency order. During loading of Linux kernel modules, the modules are often divided into an initialization and a main section. So as to save memory, this allows initialization functions which are only run once during module initialization, to be unloaded after the module has started. Thus, two separate memory allocation are performed for the initialization and main sections. The verification has to be done separately for both sections because when initialization section gets unloaded and overwritten with other data in time, only that section should be de-authorized.

Thus, the initialization section of a module may be dependent on the main section of the module. This can be detected if that section imports functions from the main section. In this case, the verification process has to recursively verify the main module and any other dependent modules, as shown in algorithm 3.1.

3.6.4 Integrity state

Welter proposed in [72], the concept of an integrity tag which indicates the current integrity state independent of the actual hash result of binaries. Appropriate certificates are used to describe the relation between the binaries and its associated integrity state. The advantage of such a tag is the separation between the integrity state and the actual binary hash of the measured component. This allows the binary component to be updated to newer versions, while maintaining a consistent PCR value set (consisting of only known tag values) to applications using the PCR and remote attestation. Such an approach greatly simplifies the verification process of PCR values at the remote end.

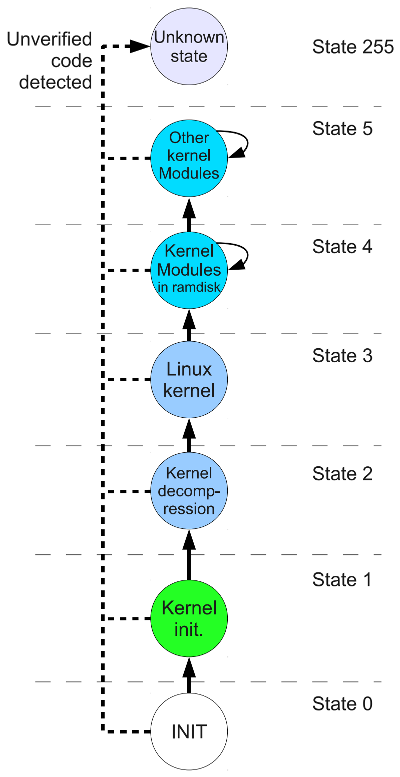
A similar concept is applied to the resulting state of the verification process. The integrity state of the guest kernel is represented by an integer state index, which corresponds to a general loading state of a guest OS. Two predefined states are the starting state (state 0) and the undefined state (state 255). The starting state is the state where the guest VM is first started, while the undefined state is a state which is entered when no matching reference binary can be found for the detected code. States can be used to separate different phases of the boot process. This enforces a particular boot sequence, and also allows the remote verifier to check that the guest OS is in a particular state during attestation.

Figure 3.7a shows the state transition used for a guest Linux used in this dissertation. States 1 and 2 represent special initialization sections of the Linux kernel which is used to set up the correct processor environment and load the full kernel. By state 3, the full monolithic kernel is loaded. After all initialization is completed, the kernel unpacks and executes the ramdisk, which loads additional kernel modules. These modules are grouped into state 4. Finally, all other modules are grouped into state 5, which also represents the normal operation of the guest OS. At any state, if unidentified code is executed, the system will enter state 255, which is the unknown state. Once the integrity state is set to the unknown state, it will always remain in this state until the guest OS is terminated or rebooted. Every state change is then reported to the TPM as described in section 4.6.

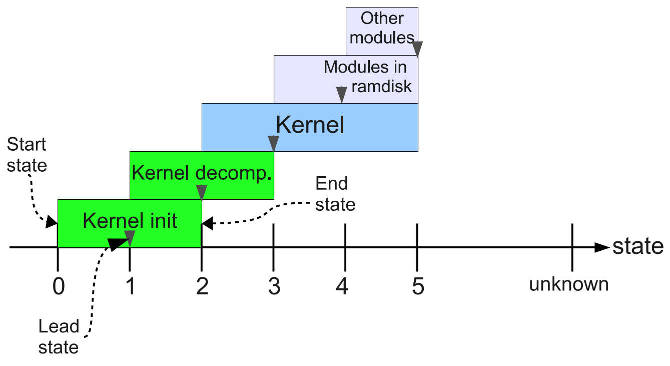
3.6.5 Optimized states

The integrity state transition is controlled by 3 parameters, which are assigned to each module: start state, lead state and end state. The 'start state' and 'end state' parameters define the range of states in which a module is allowed. A module that is executed before the start state, or after the end state, would also be considered to be unverified code, and cause a jump to the unknown state. When a particular module is verified, it will cause the current state index to increment to a minimum value defined in the 'lead state'.

Using these simple rules, the state transition is incremented from 0 to 5 in a normal OS booting process. In addition to the benefits of having a clear state separation, the grouping of modules also brings some performance benefits while performing pattern matching. This is because the grouping of modules reduces the number of valid candidates at a



(a) State diagram



(b) State transition example

Figure 3.7: Integrity state transition

particular state, thus reducing the number of comparisons needed during the matching process.

4 Host-side implementation

Chapter 3 covered the basic concept of integrity measurement, while this chapter will discuss the in-depth implementation details of integrity measurement and reporting on the host machine. It describes how a complete integrity measurement chain is built, starting from the BIOS (which is the root of measurement), and extending into the host virtual machine and finally into the guest virtual machine.

The chapter starts with a brief explanation of the build system and open source dependencies in section 4.1, followed by an explanation of the implementation details of integrity measurement pre-processing in section 4.2. This includes the types of relocation representation used, so as to fit different operating systems' operation. The actual implementation of the runtime monitoring technique, described in the chapter 3, is discussed in section 4.3. Finally, the implementation of the virtual TPM is explained in sections 4.5 and 4.6, with references to existing work done in the field of TPM virtualization.

4.1 Code base and build system

All implementations used and tested in this dissertation are developed with a common makefile, to automatically download all necessary components, apply patches and compile the necessary components. As the main contribution of this work lies in the proposed verification technique, the code is designed as a common library, known as *libvimm*. The two important functions of *libvimm* are to parse and load the pattern database, and to perform verification on the guest memory when invoked. Abstract functions for memory and TPM access serve to keep the *libvimm* code free of virtual machine specific implementations. Thus, this library can be easily integrated into QEMU and KVM virtual machine implementations, and possibly other virtual machines as well. Version 11.1 of the QEMU code is used as the base version of the development process of both prototypes.

4.1.1 Building blocks

During the course of development, the following open source code and tools were incorporated into the development system:

Linux kernel [1] - The Linux kernel, with the included KVM module, is used as the host and guest operating system, but was separately compiled. Both kernels were pinned to the Ubuntu-modified 2.6.31 version of the Linux kernel. The Ubuntu modified kernel was chosen as it was pre-packaged with the *AppArmor* module, which is used in integrity measurement within the guest OS.

qemu/kvm emulator [17] - for binary translation and hardware-supported virtualization using the official version 11.1
tpmd - tpm emulator version 0.6.1 [12]
libelf - C library for parsing ELF file [67]
pefile - a python PE file parsing library for analyzing windows PE files, version 1.2.10-63 [28]
Apparmor - a Linux security module (packaged in the Ubuntu Linux kernel) for security policy [10]
digsig - a Linux security module performing code signature verification (patch onto Linux kernel) [20]
bochs emulator bios - for developing the modified bios, version 2.3.7 [11]
TPM 1.1 virtual hardware - virtualization from the VMKNOPPIX project [14]
TPM 1.2 virtual hardware - Code from XEN hypervisor for TPM 1.2 virtualization [19]
TrustedGRUB - modified GRUB boot loader for trusted boot, version 1.1.4 [23]
OSLO - Open secure loader for performing Dynamic CTM for AMD-V processor, version 0.4.5 [40]
nickle - Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing [54]
Based on these libraries, the following components were developed:

- **libvimm** - a C library integrated into QEMU/KVM, for performing runtime verification (as described in section 3.6)
- modifications to QEMU and KVM for performing code modification detection (section 3.5)
- Offline processing tools based on pefile and libelf to generate the pattern database (section 3.3)
- Modifications to *AppArmor* and digsig to integrate the guest virtual machine measurements
- Additional modifications and integration code to make other components work together

Finally, the system was tested for both a Linux guest (2.6.31) and 64-bit windows vista SP2.

4.1.2 Common virtual hardware

Since both binary-translation and hardware virtualization prototypes use the same QEMU code base, they share the same hardware emulation layer. The following additions were made to the qemu hardware emulation:

- Bios modification - ACPI table to indicate presence of a TPM

- Virtual TPM hardware interface based on the TPM-TIS [5] version 1.1(Atmel TPM) and 1.2 (infineon TPM)

4.2 Offline parsing tools

This section describes the implementation details of generating the pattern database, as discussed previously in sections 3.2b and 3.4.4. The offline tools created for parsing and generating the pattern database take place in 3 stages. Since the verification process assumes that binary code is loaded on page boundaries, the lower page boundary of the first section is used as the reference base address. The first stage in generating the pattern database is to parse the ELF or PE binary file (with the help of libelf and pefile libraries, respectively). For each section within the file, the following information is calculated or extracted:

- base offset of section, relative to the reference base address
- section length
- offset of symbols in a section, relative to the reference base address

Additionally, for every executable section, the following information is extracted:

- raw binary code
- position and information on relocation (including imported symbols) addresses, relative to reference base address
- Imported symbols used in the section

Using the base offset of each section, the position of all local relocations is re-calculated relative to reference base address. In order to verify foreign symbols of a module efficiently, a global symbol table is generated from all export symbols in the kernel and its drivers. When the relocation entry points to a foreign symbol, the corresponding relative offset of the symbol is calculated with respect to its base address. Using this offset, it is then possible to re-calculate the actual symbol address by adding the offset to the current base address of the external module. This information is then stored in the relocation entry of the module as a relocation target.

A hash is also necessary for the invariant part of the binary code. Thus, every relocation address is masked out using zero byte values in the section data. Finally, the combined text sections are concatenated together, and a hash over all text sections is computed. Thus, after the first parsing stage, the signature database is updated with signatures to each leading 20 bytes at the start of every page boundary of the concatenated text section. SHA-1 is the cryptographic hash algorithm used for the code hash calculation. In order to generate the module database, the following information is saved as a text file for every module:

- type of binary file (32 or 64 bit)
- unit length (text sections only)
- hash of concatenated text sections (relocation positions masked out with zeros)
- relocation offsets relative to base address
- parameter of relocation offsets (see section 4.2.1)

Database packing

In order to facilitate parsing and hashing of the database, the final stage combines both the unit and module database into a compact binary form. This binary form uses a combination of fixed and variable size structures to store information on each pattern and module. The final result is a single binary file containing the entire pattern database. This binary file is hashed and extended into PCR 20 (see section 4.7.2) as an indicator of the database used for verification.

4.2.1 Deterministic verification of relocation address

Another significant improvement to the nickle scheme is the handling of relocation addresses during the verification process. Table 4.1 shows all the relocation types supported in the proposed scheme. The first 3 types support masking out either a 4-byte, 8-byte or variable length block within the code memory. The original approach in nickle uses this approach to simply ignore relocation memory locations in the module. The 4 and 8 bytes correspond exactly to a 32 or 64-bit relocation memory address. By simply ignoring these locations in the code verification process, the invariant part of the code can be easily verified.

Alt-instructions and para-instructions in Linux kernel

Alt-instructions and para-instructions are short pieces of code which is patched over the Linux kernel in runtime. Alt-instructions are selectively activated to replace generic code with its processor-optimized version, based on the detected processor model. Para-instructions are used to replace normal kernel code with a version suited as a virtualized guest kernel, so that the same compiled Linux kernel can be used for host or guest without recompilation. Collectively, these techniques are known as runtime code patching. The use of variable length masking supports ignoring such blocks of patched code.

Improved verification

However, simply ignoring relocation memory addresses opens a security vulnerability. The memory address of a function can be redirected to a malicious function, without being

Type	Parameters	Description
Ignore 32	Relocation address	ignore a 4-byte address
Ignore 64	Relocation address	ignore a 8-byte address
Ignore Block	Relocation address, block length	ignore a variable length block
Reloc 32S	Relocation address, base module, module offset	Relocation address with signed 32-bit offset from a module
Reloc 32_PC	Relocation address, base module, module offset	Relocation address with signed 32-bit offset from a module (written relative to program counter)
Reloc 64S	Relocation address, base module, module offset	Relocation address with signed 64-bit offset from a module
Reloc 64_PC	Relocation address, base module, module offset	Relocation address with signed 64-bit offset from a module (written relative to program counter)

Table 4.1: Relocation types

detected. Thus, an improved scheme is proposed to calculate the expected memory address and compares the expected value with the value in memory, during the verification process. In order to achieve this, information from the relocation table is necessary. It is stored using the 4 additional relocation types defined in table 4.1. The relocation types are differentiated into 32-bit and 64-bit memory locations, whether the relocation is an absolute or relative jump. These records store the offset of the relocation address from the base of the module, and the target of the relocation. The target can point to either another relative address within the same module or an external symbol. External symbols are parameterized with a base module and offset tuple, calculated from the global symbol table. Compared with the approach of ignoring relocation addresses, checking the validity of each address makes the integrity verification process more robust since every single code byte is now verified.

4.3 Hardware-based virtualization monitor

As introduced in section 3.5.2, an alternative detection scheme using hardware-based virtualization is used as an efficient alternative to binary translation. The hardware virtualization monitor is based upon the existing virtualization mechanism in the Kernel Virtual Machine (KVM). KVM uses the same userland and hardware emulation layer as QEMU, but relies on a KVM kernel module to perform hardware-based virtualization. This is necessary to utilize the privileged virtualization instructions from within the kernel. The kernel

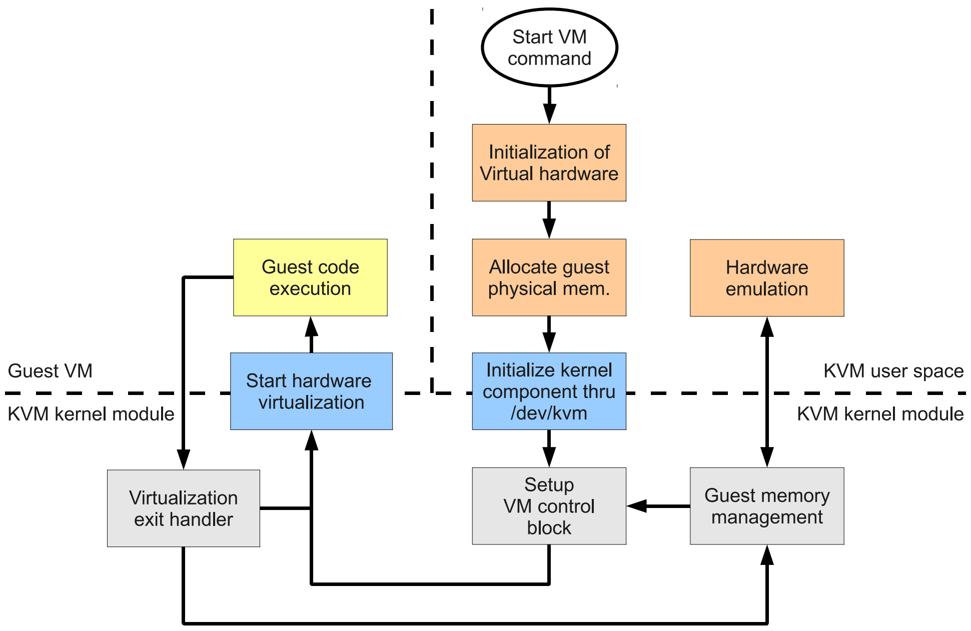

Start VM command
Initialization of Virtual hardware
Allocate guest physical mem.
Initialize kernel component thru /dev/kvm
Setup VM control block
Guest code execution
Start hardware virtualization
Hardware emulation
Guest memory management
Virtualization exit handler
Guest VM
KVM kernel module
KVM user space
KVM kernel module

Figure 4.1: Flowchart of KVM

module has a base component which provides common KVM functions, kernel interface, and a processor specific component to implement the virtualization.

Figure 4.1 shows the execution sequence of a KVM virtual machine. It starts from the QEMU-based userland component performing an initialization of the virtual machine. The virtual hardware is initialized and physical memory for the guest VM is allocated. After this, control is transferred to the KVM kernel component via a KVM device driver (*/dev/kvm*). The kernel component starts the virtual machine using the hardware-supported virtualization. Subsequently, KVM handles exits from the guest OS, performing time-slicing and memory management. Upon request of a controlled memory location or IO request, control is returned to the userland component to emulate the hardware. After completing the memory or IO request, control is returned to the kernel component to resume VM execution.

Page permissions

KVM manages the memory of the guest VM using a page table. Each page table entry defines the base address of the associated page, and also the page permissions and properties. Page permissions relevant to integrity management used in this dissertation are:

- RW (Read/Write) bit: When the RW bit is cleared, write access to the page is denied and raises a page fault. This behavior is enforced in supervisor mode only when the WP (Write Protect) bit in the CR0 register is also set.

- NX (No Execute) bit: When the NX bit is set, execution of code within the page is denied and raises a page fault. In the Intel documentation, this is known as the eXecute Disable (XD) bit.
- US (User/Supervisor) bit: When the US bit is cleared, the page is only accessible (read, write or execute) when the processor is in supervisor mode. Accessing the page while in the wrong mode causes a page fault.

Interrupts, faults and exceptions are events triggered by external hardware, the processor or software. The operating system installs handlers to handle such events when they are raised. KVM also manages the guest memory by capturing faults and interrupts in the guest OS. A page fault is raised when the processor is unable to resolve a virtual memory address to the physical address through the page table, or when the required operation (read, write or execute) violates the defined permission for that page.

The privilege level of a processor defines a general execution mode and is used for protection of resources and privileged instructions. On the i386 and x64 architecture, the privilege level is defined from 0 to 3. On most common operating system, the kernel of the OS is operating at privilege level 0, while user code is running at privilege level 3. Privilege levels 0 to 2 are defined as supervisor mode while level 3 is defined as user mode.

Shadow page table

In the absence of nested paging, KVM uses shadow paging to manage guest pages in the guest virtual machine. This dissertation focuses only on the shadow paging approach. Shadow paging is used to map the guest's memory usage into a restricted region defined by the host. Thus, the physical addresses as 'seen' by the guest virtual machine are mapped to a memory region defined by the host, as shown previously in figure 3.5. Thus, shadow paging achieves the goal of remapping the guest physical address to a host-defined region, by installing a host-controlled shadow page table. When the guest virtual machine tries to install a new page table by setting the CR3 register, this raises a virtualization exit event. KVM handles the event and replaces the guest page table with the shadow page table, which is controlled by KVM.

Handling of the shadow page table is managed by the kernel component of KVM. To perform integrity measurement, the kernel component is modified to capture all code execution at supervisor privilege levels by enforcing a W$\otimes$X policy (first introduced in [4]). This means that the writable and executable property of a code page must be mutually exclusive (see the next section 4.3.1). Thus, the first execution of a previously unverified page will cause an exit from the virtualization loop in the kernel component and return control to the user-space component. The user space component will then verify the code using *libvimm*. A list of physical page addresses of verified code is shared between the kernel and user-space. After the verification by *libvimm*, this list is updated on the kernel side, before control is returned to the kernel component to continue virtualization.

4.3.1 Page management in a hardware virtualization monitor

The basic underlying concept of the W⊗X policy is to force the page attributes into one of two modes. The first mode is read-only and executable, and the second mode is writable but non-executable. Read-only access is enforced using the read/write (RW) bit in the page table entries, while the ability to execute is controlled using the No Execute (NX) bit used in 64-bit memory handling. Even if a page is defined as writable and executable by the guest operating system, the integrity measurement framework detects the operation requested (read, write or execute) and chooses the appropriate mode. This mode enforcement allows a switch in operation between write access and execution to be detected via a page fault. Taking the Linux kernel as an example, such switching is not uncommon as memory pages for kernel modules are allocated as writable and executable in the unmodified Linux kernel. A kernel patch [45] exists to switch the page permission of the code pages to non-executable, but this is not integrated into the mainline Linux kernel. The pages are first filled with the associated code and data, before they are executed. Thus, mode enforcement appropriately switches the page permissions between writable and executable modes.

In addition, multiple pages can point to the same physical page, but with different access permissions. The *peek()* function in the Linux kernel creates such a situation where a generic page window is created into any physical memory location. Thus, concurrent write and execute access to the same physical page is prevented by maintaining a list of physical pages which are already tagged as executable. Multiple links to such pages will force all links to be one of the enforced modes.

Figure 4.2 shows the modified page fault handler of the KVM's kernel component. The Available-to-software (AVL) bits of the shadow page table are used to mark pages which are managed by the modified page fault handler. The AVL is a 3-bit field which can be freely used by software (ignored by the processor for page mapping). Since these bits are not used by the original KVM module, they are used to mark the assigned mode of the shadow page. Bit 0 of the AVL indicates that the page is monitored by the integrity management, and bit 1 indicates that the page has been marked as executable. This marking is necessary to handle memory pages which have been marked writable and executable by the guest kernel. Such pages need to switch between enforced writable and executable modes, while re-invoking integrity measurement before being made executable.

The lower section of figure 4.2 shows the additional page handling mechanism. Based on the processor state and page attributes, the shadow page is either read-only and executable, or writable and non-executable. This fulfills the W⊗X condition. Non-executable guest pages are not considered, since they are inconsequential to integrity measurement. For the special case of guest pages which were set as read-write and executable, these pages require switching between writable and executable modes. Thus, a by-pass is inserted before the KVM's page handler, to perform this mode switching without affecting the normal page handling. Physical pages which have been verified are added to a verified list of pages, while any write access will remove pages from this list. Thus, the memory pages are protected even in the case of concurrent access from multiple guest pages. Just like the QEMU case, the user-space component of this modified KVM uses *libvimm* to verify the integrity of new code pages, and perform subsequent reporting of the integrity state.

4.4 Effectiveness on Windows kernel

Implementing integrity measurement for the Windows kernel involves modifying the offline parsing tool to support the PE file format (section 3.4.1.2). The modified parsing tool generates a binary pattern database in the same format which can be loaded by the integrity measurement framework. No changes were necessary in the runtime code detection or code verification. A major difference between the Linux and Windows kernel while performing verification, lies in their respective paging policies.

Memory paging on Windows uses a technique known as demand paging, which saves memory by loading pages from the executable file as needed. Thus, the virtual memory range allocated to the given binary starts out being unlinked to any physical page. As each page gets accessed, it raises a page fault and the kernel allocates the necessary physical page and loads the respective page from file into memory. This technique saves memory as unused portions of a binary never get loaded and do not use up physical memory. Furthermore, when there is insufficient physical memory, infrequently used pages can be de-allocated to provide memory for other binaries. A few exceptions are the base kernel file (ntoskrnl.exe) and its statically linked files, which are loaded into memory by the windows boot loader (NTLDR) rather than the kernel itself. These files are loaded wholly into memory before the kernel is started without any demand paging mechanism.

Demand paging has serious implications for integrity measurement, since only loaded pages are available in memory. Thus, the entire binary module cannot be verified but rather, only loaded pages. To address this problem, binaries that use demand paging are separated into its smallest loadable units, which are individual pages. Thus, when such a binary is loaded and executed, each page is verified independently, and relocation references that cross the page boundary are changed into foreign symbols.

In addition to the issue of demand paging, the Windows kernel also generates dynamic code during runtime. Such code is generated during the execution of the windows kernel, and thus, is not represented in the respective binary file. One such example of dynamic code is trampoline tables. A trampoline table is a memory array used by the hardware abstraction layer to store jump instructions. Performing such jumps is sometimes necessary to change processor states. Under Windows, the locations of these trampoline tables are dynamic, but can be derived relative to other loaded modules. Since there is currently no clear method to verify the content of the trampoline tables, they are simply ignored in the integrity measurement process. Due to other dynamic changes to the binary code which

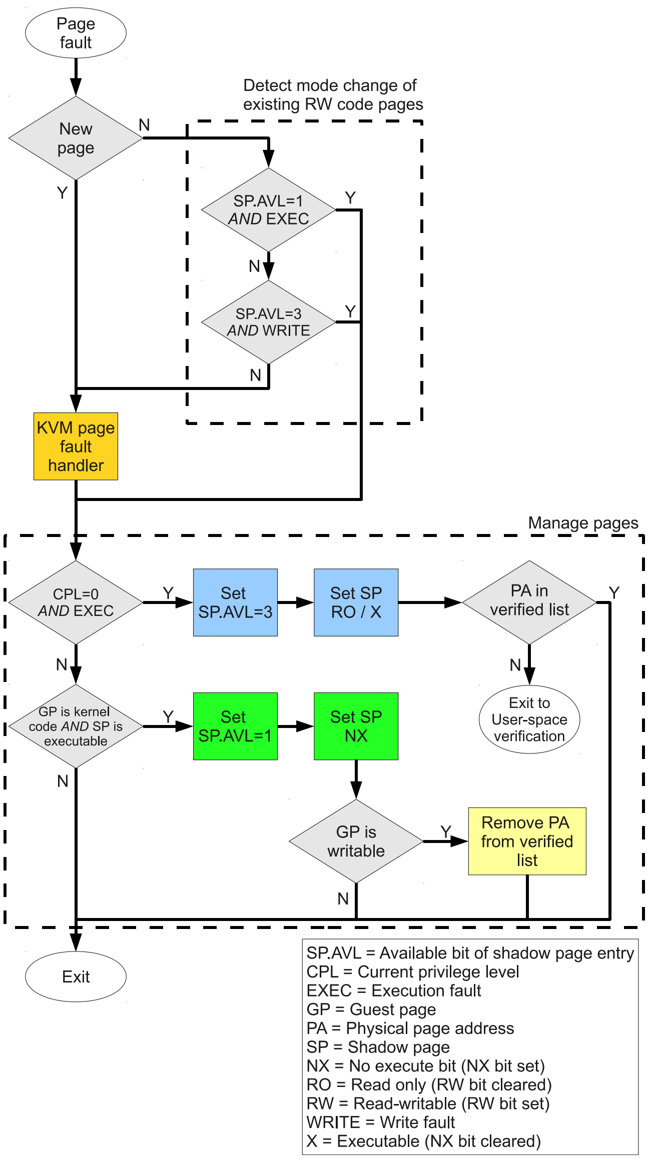

Page fault
New page
N
Y
Detect mode change of existing RW code pages
SP.AVL=1 *AND* EXEC
Y
N
SP.AVL=3 *AND* WRITE
Y
N
KVM page fault handler
Manage pages
CPL=0 *AND* EXEC
Y
Set SP.AVL=3
Set SP RO / X
PA in verified list
Y
N
Exit to User-space verification
N
GP is kernel code *AND* SP is executable
Y
Set SP.AVL=1
Set SP NX
N
GP is writable
Y
Remove PA from verified list
N
Exit
SP.AVL = Available bit of shadow page entry
CPL = Current privilege level
EXEC = Execution fault
GP = Guest page
PA = Physical page address
SP = Shadow page
NX = No execute bit (NX bit set)
RO = Read only (RW bit cleared)
RW = Read-writable (RW bit set)
WRITE = Write fault
X = Executable (NX bit cleared)

Figure 4.2: Modified page fault handler

cannot be pre-determined, the integrity measurement algorithm is only partially successful in verifying the Windows kernel, and is applicable up to early stages of the Windows boot process.

4.5 Virtual TPM

4.5.1 Previous work in virtual TPM

The application of a virtual TPM (vTPM) starts from the need to multiplex the TPM access between multiple parties, such as different guest virtual machines. Even though the TSS stack performs the function of managing TPM-related resources and also performs proper multiplexing of the TPM access for concurrent applications, the concurrent use of TPM for multiple virtual machines poses new challenges. The chain-of-trust as represented by the set of PCR registers, while applicable for a single OS cannot effectively represent the individual chains associated with independent virtual machines. Furthermore, guest virtual machines can be restarted but most of the PCR values on the hardware TPM cannot be re-initialized. Thus, the general solution to this problem is to virtualize the TPM.

A TPM emulator implementation [68] exists for the Linux platform, which emulates the TPM as a serial character device, responding to byte-level instructions from the TSS. The TPM emulator runs as a background service, which creates a UNIX named pipe to accept connections. To support legacy software which expects a TPM device driver, a simple tunnel is created via a kernel module to forward all data stream between a virtual TPM device driver stub and the named pipe.

A virtual TPM manager was written for the XEN hypervisor, which can manage simultaneous instances of the TPM emulator, and connect each virtual machine instance to its dedicated TPM emulator instance. This allows each virtual machine instance to effectively take control of its own unique TPM instance. No such manager implementation exists for other hypervisors or virtualization platforms. An open issue with virtual TPM instances is the establishment of trust between the hardware and virtual TPM, and the treatment of the endorsement key in a virtual TPM.

4.5.2 Endorsement key problem in Virtual TPM

The virtual TPM implementation in [68] shows how multiple virtual machines can each be supported by its respective virtual TPM instance. However, a critical issue lies in the trust linkage between the hardware TPM and the virtual TPM. A hardware TPM can be trusted based upon the assumption that it is physically mounted onto the system board and cannot be easily replaced or tempered. An Endorsement Key (EK) with its corresponding endorsement certificate is generated and saved into the hardware TPM during the manufacturing process. Since the TPM is the root-of-trust, it is implicitly trusted. The hardware TPM can be trusted based on its endorsement certificate. The endorsement certificate associated with the EK is normally signed by the TPM manufacturer and inserted securely during manufacture-time. This forms the trust basis in which a Privacy Certification Authority can trust that the system contains a valid and genuine hardware TPM based on a known TPM manufacturer's certificate.

This endorsement certificate is, however, not transferable to the virtual TPM since the endorsement key cannot be exported outside the hardware TPM. Since no authority exists to endorse virtual endorsement keys on the virtual TPM, the trust chain linking the hardware TPM to the virtual TPM gets broken. A few suggestions have been proposed in [68] and [58] to resolve this problem. No new solution will be presented as this issue is outside the scope of this dissertation. In this dissertation, it will be assumed that a corporate body will sign virtual endorsement keys generated by the virtual TPM. The virtual TPM itself can be protected by sealing its internal state to the physical TPM.

4.6 Trusted reporting

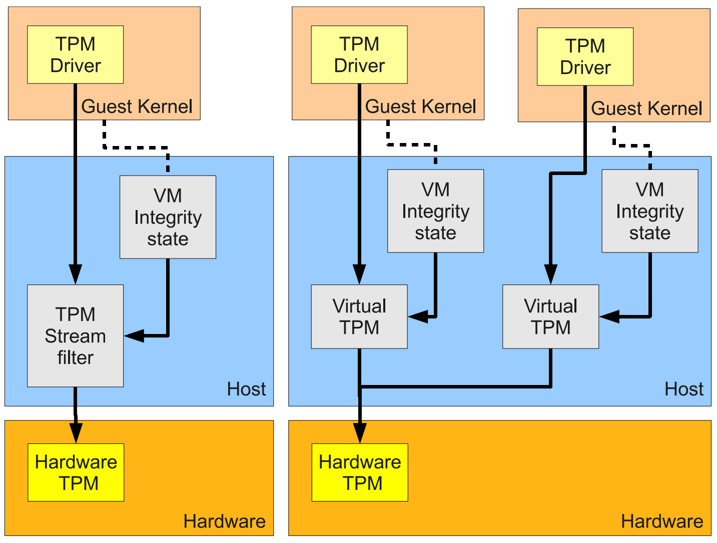
The final stage in integrity measurement is to report the current state in a trustworthy manner. The TPM is used for this purpose as it has been designed for trusted reporting. In contrast to reporting hashed measurement of the binary code, the reporting of an integrity state is proposed, as discussed in section 3.6.4. In the context of property-based attestation [30], the integrity state can be seen as a property. The advantages of property-based attestation have been discussed in [30], and this dissertation serves to demonstrate a practical realization of an actual property in runtime.

4.6.1 TPM sharing models

Measurement reporting is done via the "extend" operation of the TPM (see section 2.2.1). Unlike para-virtualization, the integrity measurement framework emulates a TPM hardware interface, with no additional virtual machine specific interface (hypercall) for the guest kernel. Thus, the only way to transmit information is via the virtualized IO interface. To provide compatibility for the existing software infrastructure, the low level TPM interface based on the TPM Interface Specification (TIS) [5] was chosen as a suitable hardware interface for the purpose of measurement reporting into the guest OS. This interface allows the guest OS to make integrity reports to the underlying integrity measurement framework. Two models of multiplexing TPM between the host and the guest OS are proposed in this dissertation.

Single client model

The first model is targeted at a client system scenario where users generally run only a single VM. In this case, only a single instance of the VM exists, and can be associated directly with the hardware TPM through the host as shown in figure 4.3(a). In this approach, a simple filter module checks the low-level TPM command byte stream (TIS) sent



(a) Single client model **(b)** Server model

Figure 4.3: TPM measurement reporting

by the guest OS, and forwards allowed commands directly to the hardware TPM. The filter module allows specific commands to be blocked by the host, and also enables the host to concurrently perform stateless TPM operations.

Since no virtual TPM is involved in this model, there is no trust chain problem as described in section 4.5.2. Furthermore, all secrets sealed by the client platform are secured by the hardware TPM and is not vulnerable to a compromised host.

Server model

However, the client sharing model is not suitable for a server or data center use case, where typically multiple virtual machines are running simultaneously. Each VM may start, stop or restart independently, and require its own independent TPM state. This can be achieved through a virtual TPM as discussed in section 4.5. Figure 4.3(b) shows a virtual TPM-based sharing model, where each virtual machine is assigned its own virtual TPM instance. The low-level TPM command byte stream (TIS) is sent through a filter module and allowed commands are forwarded to the virtual TPM. The PCR values associated with the measurement of the hardware, hardware boot loader and host integrity are extended into each new virtual TPM instance. In this way, the PCR measurement starts with a common state based on the hardware and host, but branches off independently for each VM instance. The integrity measurement process which is tied to each virtual machine is also allowed to perform stateless TPM operations on the virtual TPM.

Advantages of the server model include a faster TPM access, as the software-based

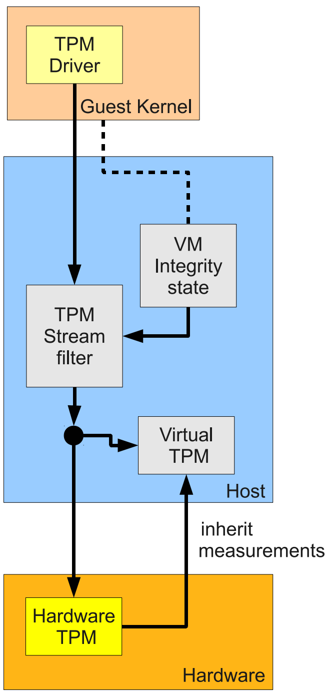


Figure 4.4: TPM filtering implementation

virtual TPM is generally faster than the hardware TPM. Also, VMs can be restarted independently, since new virtual TPM instances can be restarted for each VM.

4.6.2 TPM filter implementation

In order to test and implement both TPM sharing models in 4.6.1, the TPM interface in the virtualization is implemented as a flexible filter as shown in figure 4.4. This filtering implementation consists of 2 stages. The first stage is a filtering logic which receives the complete TPM byte stream from the virtual machine. The filtering logic is able to reject blocked TPM commands, or forward the TPM commands. The second stage is a variable output stage which can send TPM commands to either a virtual or hardware TPM. In the client sharing model, the output stage connects to the hardware TPM directly. For the server model, the output stage connects to the virtual TPM associated with the respective guest VM.

In both sharing models, the filter logic checks for the TPM extend command, and blocks any extend operation on PCR lower than 22. The filter protects the lower PCRs which are used by the host and integrity measurement, while allowing the guest VM access to the guest controlled PCRs. This is further discussed in section 4.7.2, where a differentiation between host and guest controlled PCRs is described.

4.6.3 Dynamic Root-of-trust measurement

As explained in section 2.2.7, Dynamic Root-of-Trust Measurement (DRTM) allows the processor to start a separate root-of-trust for measurement via a trusted loader. The DRTM mode of operation can be used by inserting the OSLO loader [40] into the chain of modules to be loaded by the boot loader. The OSLO loader is divided into separate modules, each performing separate functions. Every piece of binary is compliant to the GNU multiboot specification [27], and can be executed as a chain using a multiboot capable bootloader such as GRUB or SYSLINUX. The OSLO loader uses 4 separate multiboot compatible binaries, in the provided order, to perform the entire DRTM chain:

- *oslo* - performs the SKINIT operation [24] to put the TPM into locality 4 while resetting the PCRs 17-22. This operation also automatically hashes and extends the oslo binary into PCR 17.
- *beirut* - performs hashing of command line arguments of the remaining multiboot modules into PCR 19
- *pamplona* - reverts the Device Exclusion Vector (DEV) protection and clears the global interrupt flag, so that the Linux kernel can be loaded without modifications
- *munich* - loads and executes the Linux kernel and ramdisk since the Linux kernel (bzImage format) is not multiboot compatible.

The entire boot sequence is shown in table 4.2.

4.7 System integration

In order to build a complete working system of the entire integrity measurement framework, the various components described so far have to be integrated. The framework, including code detection and verification, is integrated as part of the virtualization process. The pattern building is done as a separate offline process of building the database, and thus, does not require integration into the measurement framework. Other external components that need to work together with the framework are the TPM emulator, host OS infrastructure (such as the boot loader), host kernel and host file system.

A session controller spawns both the virtual TPM emulator and the VM. The controller connects the VM to the virtual TPM emulator via uniquely named pipes, and also, sets the appropriate execution parameters. Furthermore, the persistent state of the TPM emulator has to be managed via a file-based storage. This is achieved by a simple configuration file for each virtual TPM and VM pair.

As for the rest of the host OS, it is created as a minimal ramdisk file system, so that its binary measurement of the host, which is performed once at boot-time, is always consistent. Section 4.7.1 describes the boot sequence of the integrated measurement framework.

Stage	Binary name	Function
0	BIOS	Hardware setup, loads boot sector
1	Boot sector	Loads boot loader
1	Boot loader (TrustedGRUB)	Loads OS
2	oslo *	Perform SKINIT operation, measures and extends next module
2	beirut *	Perform additional hashing of command line of modules
2	pamplona *	Setup environment compatible for Linux kernel loading
2	munich *	Loads and executes Linux kernel bzImage and ramdisk
2	Linux kernel bzImage	Linux kernel
2	Linux ramdisk	Linux ramdisk
3	Session manager	Start TPM emulator and virtual machine
4	Guest BIOS	Hardware setup, loads boot sector (QEMU based)
5	Guest Boot sector	Loads boot loader
5	Guest Boot loader (GRUB)	Loads OS
6	Guest kernel and modules	Guest kernel
6	Guest applications	Guest applications

* present only if DRTM is used

Table 4.2: Boot sequence

4.7.1 Boot sequence

The boot sequence of the framework varies slightly, depending on whether if DRTM is used. Table 4.2 summarizes the important steps in the boot sequence of the integrated system in order of execution. After the session manager is started, the execution sequence branches off, depending if the single client or the server model is applied. In the case of the single client model, there is only a single chain of execution, and all TPM operations are sent to the hardware TPM. In the server model, stages 4 to 6 are carried out independently for each guest VM, and the TPM operation of each VM is sent to its virtual TPM instead.

4.7.2 Completing the trust chain

In order to complete the trust chain from the boot-up to the guest OS, the trust chain and the associated measurements need to be collectively stored in the TPM. Table 4.5 summarizes the PCR configuration on the hardware TPM. The table is differentiated into the case where Dynamic Root-of-Trust (DRTM) is applied or not.

PCRs 0-4 are used by BIOS to perform self-measurement and also the measuring of the Initial Program Loader (IPL) found within the Master Boot Record (MBR) of the boot device. The IPL measures and executes stage 1 of the GRUB boot loader, which, in turn, measures and starts stage 2 of GRUB. Both measurements are extended into PCR 8 and 9, respectively. GRUB is fully loaded after stage 2 is started, and it, in turn, starts the chain of binaries to load the host OS. In the case without DRTM, it loads the Linux kernel, ramdisk directly, extending hash measurements into PCRs 12-14. In the case of DRTM, it loads the OSLO loader, extending hash measurements into PCR 12 and 14. The OSLO loader initiates the DRTM sequence before loading the host Linux kernel and ramdisk. Table 4.5 shows the different PCRs used in each case, as the DRTM process resets PCR 17-22.

PCRs 20-23 are used for guest VM specific purposes. PCR 20 contains a hash of the binary database file used for integrity measurement, while PCR 21 contains the actual runtime integrity state from the measurement process. PCRs 22 and 23 are used by the guest OS itself. In the single client model, this group of PCRs are extended directly into the hardware TPM. In the server model, the PCR values on the hardware TPM are hashed together and written to PCR 0 of the virtual TPM when the VM is started. Subsequent guest-related measurements are extended into the virtual TPM only.

PCR Index	Controlled by	Data measured
0-3	BIOS	BIOS specific
4	BIOS	measurement of IPL
8-9	IPL	Measurement of TrustedGRUB stage 1 and 2
12	TrustedGRUB	GRUB command line
13	TrustedGRUB	Host Linux ramdisk
14	TrustedGRUB	Host Linux kernel
15	Host system (ramdisk)	Host file system
20	QEMU / KVM	pattern DB signature
21	QEMU / KVM	Integrity state tag
22	Guest kernel LSM	*AppArmor* policies / digsig public key / state
23	Guest apps	Guest application data

(a) PCR configuration for non-DRTM prototype

PCR Index	Controlled by	Data measured
0-3	BIOS	BIOS specific
4	BIOS	measurement of IPL
8-9	IPL	Measurement of TrustedGRUB stage 1 and 2
12	TrustedGRUB	GRUB command line
14	TrustedGRUB	OSLO Stub
17 *	CPU (skinit)	OSLO loader
19 *	OSLO	OSLO modules, host Linux kernel and ramdisk
20 *	QEMU / KVM	pattern DB signature
21 *	QEMU / KVM	Integrity state tag
22 *	Guest kernel LSM	*AppArmor* policies / digsig public key / state
23	Guest apps	Guest application data

* DRTM protected PCR values

(b) PCR configuration for DRTM based prototype

Figure 4.5: PCR configuration using only the hardware TPM

5 Integrity measurement applications

While the previous chapters explained the concept and implementation of the integrity measurement framework, this chapter discusses how integrity measurement can be extended and integrated with additional tools within the guest OS and remote hosts.

The first section deals with modifying existing Linux security mechanisms on the guest, so as to extend the integrity measurement to form a more complete measurement chain covering application-level security. The first scheme is a simple code-signature verification enforcement, which will be augmented with trusted measurement to form a complete trust chain. The second is *AppArmor*, a process-level resource policy enforcement scheme, which will also be augmented with trusted measurements to form a complete trust chain. Both schemes are implemented in the guest kernel and extend a hash into a PCR register based on the loaded enforcement policy.

To demonstrate the application of the integrity framework, sections 5.2 and 5.3 discuss two real world scenarios of applying integrity measurement to personal and corporate platforms. The former uses integrity measurement for trusted authentication and protection of personal data. This security system uses the TPM to verify the entire software stack on the user's behalf. The user's data can only be decrypted only under the condition that the user's credential is valid and the system is integral. The latter use case applies integrity measurement to a trusted intrusion detection system (IDS). This allows a remote server to maintain a VPN connection which is bound to the runtime integrity of the client machine. The connection is automatically broken whenever the client's integrity state is compromised at any level of the software stack. This allows a corporate network to protect itself against compromised hosts connected to the network.

5.1 Userland protection

This section describes the extension of the runtime memory measurement of the guest kernel space, covered in chapters 3 and 4, to the user space. Rather than applying the same code verification technique, which might incur high overhead, existing security mechanisms in the guest kernel are modified to perform measurement of user space programs. Security mechanisms in the kernel are able to take advantage of the process information available to the kernel for perform measurement. This also allows the security mechanism to be more fine-grained by using the additional information.

Two measurement approaches based on Linux Security Modules are discussed here. The effectiveness of userland protection is a research area in itself and will not be discussed in this dissertation. Rather, the description will be on how these techniques can be integrated into the integrity framework using Trusted Computing.

5.1.1 Linux Security Module (LSM)

The Linux Security Module (LSM) API enables security modules implemented in the Linux kernel to insert hooks into core kernel functions, to control the access of resources in the kernel. The hook functions allow specific operations in the kernel to be explicitly intercepted by the security module. This makes it possible to implement well-defined security models, such as mandatory access control (MAC). The SELinux [64] LSM is one example of MAC for the Linux kernel. The scope of LSM covers a wide variety of kernel functions including program execution, I/O operations, socket and pipe operations, resource labeling and audit operations.

Currently, existing LSM modules are mostly designed for security enforcement rather than integrity measurement. Thus, integrity measurement can be achieved by reporting security breaches through the TPM interface. Since the kernel and the LSM module are already measured by the underlying integrity framework, the measurements provided by the LSM module extends the trust chain into the domain of application integrity.

The LSM uses a guest-controlled PCR (see section 4.7.2), which is different from the set of PCRs used for measurement of the guest kernel by the host. In this way, the management of the LSM-based measurement is independent of the integrity measurement framework. The validity of the measurement is ensured based on the fact that the guest kernel itself is integral. Two LSM-based modifications are presented in the following sections. They perform code-signature verification (software white-listing) and process monitoring, respectively.

5.1.2 Digsig security module

Digsig [20] is an LSM module which reads and verifies a digital signature embedded into ELF binaries, so as to verify the integrity of the binary before execution. It requires the support of two userland applications. The bsign [3] application is used to sign the ELF binaries and embed the code signature back into the binary. Another application inserts the corresponding public key into the digsig module via a *sysfs* file. This public key is used to verify the code signatures. No verification is performed before the public key is inserted. After the key is loaded, the digsig module verifies the signature of every ELF binary before they are executed. To support the integrity measurement framework, the digsig module was modified in the following manner:

1. Digsig extends a hash of the signature public key sent from the userland application, into the LSM-controlled PCR.

2. For simplicity, digsig does not allow the inserted public key to change once it has been inserted. (write-once until reboot)

3. After the public key is inserted, digsig extends a known value into the LSM-controlled PCR if any loaded binary fails the signature verification.

As binaries are not verified before the public key is inserted, the insertion process should be done as early as possible in the boot-up process. Inserting the key during the ramdisk stage of Linux booting can maintain the integrity of the boot chain, as the ramdisk is already measured by either the framework or the trusted bootloader (see section 4.7.1).

Since the public key is used in verifying binary signatures, this key should not change frequently. A simple solution is to enforce a write-once policy, where the key can only be inserted once with no further change possible. This would be sufficient for a use case where binaries are signed with a common root key. An alternative approach is to allow multiple public keys. This is similar to the case of having multiple root certificates within a trusted root store. Execution of binaries with signatures that matches any of the keys would be accepted.

Finally, the module has to report cases of failed signature verification by extending a known value into the PCR reserved for LSM usage. This completes the trust chain by reporting the execution of any unsigned binaries. With this approach, the integrity framework can be extended to monitor the integrity of all loaded binaries within the guest OS. Therefore, an attestation of the PCRs would prove that a known key has been used to verify all binaries in the guest and that no invalid binary has been executed at the time of attestation.

5.1.3 AppArmor security module

The decision not to apply the same kernel memory detection technique to measure userland applications was consciously taken. It was done to reduce the complexity of the signature database. It also allows greater security granularity within the LSM modules, which can make use of additional process information available in the guest kernel for integrity verification. This is demonstrated, to a greater extent, in the use of *AppArmor* [10] as an integrity checking extension to the integrity framework.

AppArmor is another LSM module which enforces Mandatory Access Control (MAC). Like other MAC systems, *AppArmor* enforces explicit access of processes to file, network and other resources on the system. The file access control is based on file paths rather than file system based labels (such as in SELinux), so that no modification to the file system is necessary. *AppArmor* enforcement targets individual processes rather than users. All rules are written in a plain text format, defining access rights of a particular process to file or network resources. This is known as a profile, and is inserted into the kernel via a special file in the *AppArmor*'s security file system (securityfs). Profile insertion is usually carried out early in the system boot-up process, so as to have *AppArmor* enforcement in place before starting other processes.

AppArmor has two modes of operation: complain and enforcement modes. Complain mode only logs all access violation against the given profile. This mode is often used in the creation of an initial profile for a process. Enforcement mode enforces the profile and restricts access to defined resources. Like in digsig, *AppArmor* requires a set of userland applications to perform the loading, unloading and modification of policies in *AppArmor*.

Figure 5.1 shows a simple flowchart of how *AppArmor* works. *AppArmor* profiles are inserted into the *AppArmor* LSM module via a device file mounted in the security file system (securityfs) used to control kernel security configuration. These profiles are stored in a profile database. To ensure maximum effectiveness over all processes, they have to be inserted early in the boot process.

When a kernel system call is made by a userland application, hooks within the kernel API handler are used by the installed LSM to check if the call has to be audited. If so, control is passed first to *AppArmor* to process the API call. Within *AppArmor*, each of these system calls is handled by checking if the profile associated with the process is allowed to perform the requested operation. In enforcement mode, disallowed operations are rejected, while in complain mode, a log entry is created for each disallowed operation.

To extend integrity measurement from the host into guest-level measurements, the *AppArmor* module was modified to hash the profile and extend this hash into the LSM-controlled PCR (see section 4.7), every time a profile is loaded, unloaded or modified. As in the case of digsig, the *AppArmor* policies should be loaded in the initial ramdisk phase to maintain the chain-of-trust, and policies should be in enforcement mode. As a result, the final value in the PCR can be computed and should remain unchanged throughout the lifetime of the system. An attestation to that known value in the PCR would signify that the correct set of policies has been loaded and not been tempered at the time of attestation. This implies that the correct profiles are being enforced. Due to the nature of *AppArmor*, only policies which are in enforcement mode are actually consequent to the integrity of the overall system.

5.2 Trusted authentication and disk encryption

In order to showcase the application of integrity measurement, the following sections will discuss how the integrity management framework can be integrated with the existing security infrastructure to solve real-world problems. Two use cases will be discussed, covering both usage scenarios for personal and corporate applications.

The first scenario aims to demonstrate the usage of integrity measurement and trusted computing for trusted authentication and storage. This scenario is targeted at protecting systems for personal use. Trusted authentication gives the user confidence that the authentication process inherently verifies the system integrity. Subsequently, trusted authentication invokes the decryption of the user's home directory only if the system is integral. This two-step approach ensures that the user's operating environment and data are integral upon a successful trusted login. Trusted authentication and decryption is achieved by coupling the authentication process to a TPM key tied to a particular integrity state. The

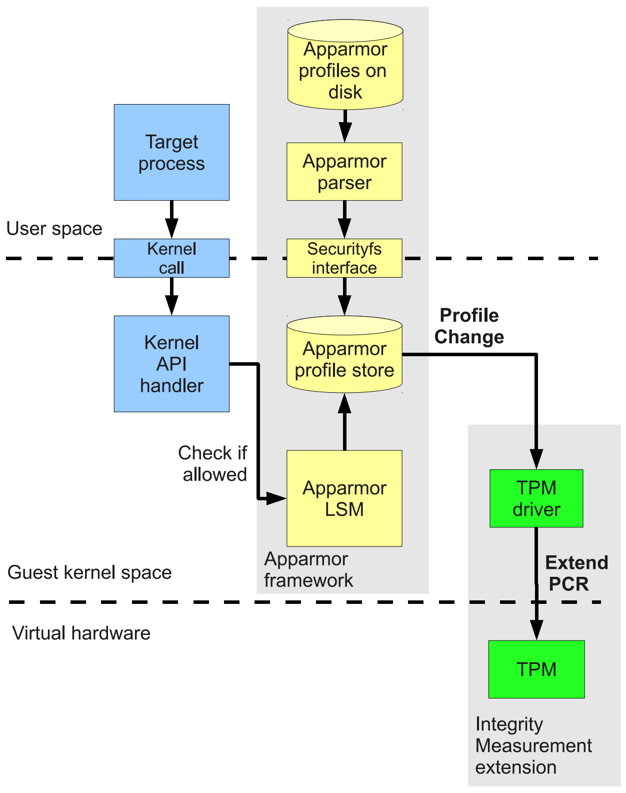


Figure 5.1: *AppArmor* framework with integrity measurement extension

next section discusses weaknesses of existing authentication schemes and the need to have a trusted authentication process.

5.2.1 Weaknesses of authentication schemes

Typical authentication schemes are based on either a secret known to the user (such as a password), or a token that cannot be duplicated (such as a smart card or, to some extent, biometric identity). The strength of the former method lies in its difficulty in guessing the secret, while the strength of the latter method lies in the difficulty of replicating the token or identity.

Considering the secret-based method of using an authentication password, a salted password hash is usually used to protect the security of the user password. A salted password entry consisting of the salt and password hash of user i is defined by:

$salted\_password\_entry_i = (salt_i, hash(salt_i \,|\, password\_string_i))$

The salt is a randomly generated number which makes the hash different even if the password is the same, so as to prevent pre-calculated password hash attacks [51]. The stored password entry consists of a tuple of the random salt and the hash. The password hash is recalculated from the password given by the user during authentication. The user is authenticated by verifying the fact that the recalculated hash matches the stored hash. In this way, even the system administrator has no knowledge of the password used by the user.

Even under the assumption of a strong cryptographic hash, such methods are naturally vulnerable to guessing attacks such as a brute-force attack (guessing all possible password combinations systematically) or dictionary attack (using combination of frequently used words to efficiently guess the password). Thus, for passwords which are relatively short, if the hashed password can be extracted from the system, it can be cracked by using a powerful system to guess the password in reasonable time. This is known as an offline attack on the password.

In the case of a personal system, authentication is usually performed locally and not handled by a secure authentication server. In this case, all unencrypted data can be read or even tempered by a malicious user who has physical access to the system, making an offline attack viable.

5.2.2 Sealing and unsealing operations

In order to prevent an offline attack, the TPM can be used to nullify the effect of extracting the hashed password. The TPM provides two main functions for dealing with data encryption, namely binding and sealing. Data stored in an encrypted form external to the TPM, is bound to the TPM when a TPM storage key is used to encrypt the data using asymmetric encryption. This piece of data can only be decrypted by the same TPM when the encrypted data and the wrapping key are presented again to the TPM through the TSS. In addition, the parent key or chain of parent keys used to wrap the storage key, needs to be unwrapped in sequence, before the bound data can be processed by the TPM. Sealing is a similar operation, with the usage of the storage key (see section 2.4) and the additional condition that selected PCRs in the TPM must be in a specific state. In order to encrypt a large piece of data, a symmetric encryption key is used for encryption. The symmetric key used for decryption can be bound or sealed using a TPM storage key instead of the data itself.

While using the sealing operation, the TPM can be used to restrict access to a piece of secret based on a set of PCR values that represent a known state. The guarantee provided by sealing, so that the integrity measure at the point of unsealing is identical to the requested integrity measure during sealing. As a result, the secret is accessible only when the system is in an accepted integrity state.

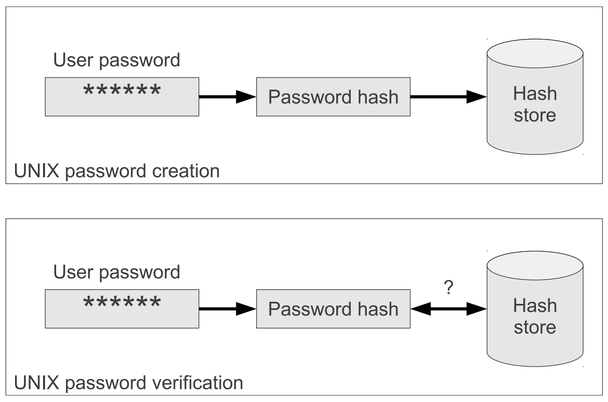
5.2.3 Trusted authentication

With the trust properties of the TPM seal operation, it can be used to build a trusted authentication scheme that verifies the system integrity during login, and to also overcome problems of guessing attacks, as discussed in [29].

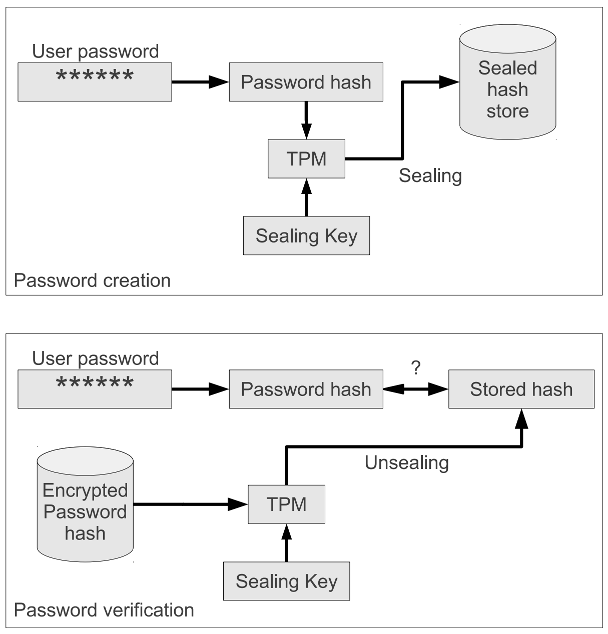
The proposed trusted authentication uses the seal operation of the TPM, to seal the password hash used in the UNIX login mechanism. This provides a variety of advantages. Firstly, the seal operation enforces a verification of the system state as represented by the TPM. When used together with integrity measurement, the integrity of the host hardware, host virtualization, guest kernel and guest applications are fully represented as properties in the PCRs (as shown in figure 4.7.2). In addition, the use of the TPM itself prevents offline attack, since it can only be carried out on the same platform, and only when the system is in the state defined by the respective PCRs. Thus, an attacker is unable to carry out brute force or dictionary attacks by copying the password hashes and carrying out the attacks on a more powerful remote machine. Also, attacks on the same machine booted up into a different state will also fail, since the unsealing operation will not be successful. The use of hardware TPM also restricts the repetition speed in an online (on the same platform) brute force attack since all TPM operations are limited by the speed of the TPM chip. The TPM uses a relatively slow LPC bus so that, while the delay is minimal for single logins involving only a few TPM operations, it becomes significant when used repeatedly in a brute force attack.

5.2.4 Implementation of trusted authentication and storage

The trusted authentication scheme was implemented using the Pluggable Authentication Module (PAM) framework, which is widely used in many UNIX and Linux systems. The authentication scheme was implemented as a PAM module modified from the UNIX authentication module. Figure 5.2 shows the difference between the UNIX and the trusted authentication scheme.



(a) UNIX authentication scheme



(b) Trusted authentication scheme

Figure 5.2: Normal and trusted authentication scheme

After generating a password hash, the module uses the TSS to encrypt the hash using a TPM key. The TSS returns an encrypted blob, which is stored directly in the password file as a base64 encoded string. This avoids the need for a separate database or file. The string is prefixed with a special token to indicate that it is a TPM-encrypted string, so that the password file can still support normal UNIX passwords concurrently. The trusted authentication process is performed by extracting the encrypted data blob from the password database, and decrypting it using a TPM key through the TSS. Simultaneously, the authentication password is hashed and compared with the decrypted data blob. Authentication is successful if both hashes are identical.

The trusted authentication implementation supports two ways of using the TPM key. The first method generates and uses a separate TPM key for each user. This method has the advantage of being able to associate different users to different system integrity measurements, for different levels of trustworthiness. The second method uses a common TPM key for every user, and it is associated with a common integrity measurement for all users. This method has lower management overhead, since only 1 key has to be stored and used for all users.

5.2.5 Drawbacks of the system

As with other trusted computing based systems, there are certain drawbacks to the system. Building a robust backup solution for a trusted computing based system is non-trivial. This trusted authentication suffers from the same problem in that, if a non-migratable key is used for the TPM sealing operation, the key is only usable when the specific TPM in that specified system state. In the case of a hardware failure of the TPM, there is no mechanism to restore the key. If a migratable key is used, a TPM key migration can be performed between the original and target TPMs, so as to keep a safe backup of the TPM sealing key on another system. However, both schemes will fail if the TPM on the original system fails before either the data or key is backed up. A more robust backup solution can be built around the system by making secure copies of the actual authentication hash and symmetric encryption keys used in the trusted authentication and encryption.

Another disadvantage of the system is the inability of the user to verify the integrity of a local physical platform before the user enters his authentication information. This can occur if the original trusted authentication process has been tempered or by-passed completely. The user is generally not able to differentiate between the case of a trusted and non-trusted authentication window, when presented on a screen or user-interface device. One such example would be a malicious authentication screen which is identical to the original screen, but serves to perform no real authentication while capturing the entered authentication password. In this case, the encrypted data is still secure as changing the authentication screen would give rise to a modified integrity measurement. However, the password is already compromised in the process.

This problem is not only specific to trusted authentication but also in almost all authentication mechanisms, such as verifying the integrity of a credit-card machine. One possible solution is to perform a challenge step to the system before user authentication. This could involve using a secure token to verify the system integrity in a manner similar to remote attestation. Alternatively, the system may decrypt and display a simple piece of user-specific data sealed using a TPM key, to prove its integral state.

5.3 Trusted intrusion detection system

This second use case demonstrates how integrity measurement can be part of an Intrusion Detection System (IDS) within a corporate network or data center. The application of trusted computing and the integrity measurement framework enhances the detection capabilities of existing IDS, and the use of virtualization is well suited in a data center scenario.

Intruder Detection System (IDS)

An intruder detection system (IDS) is an entity or system that monitors the network for malicious activities or policy violation. IDS can be mainly classified into Network-based IDS (NIDS) or Host-based IDS (HIDS). The former monitors the network traffic and traffic pattern at the network-level, while the latter checks a host against a given policy, usually with the help of a local agent on the host. Policy violations are usually appropriately logged and reported to the network administrator. The remote attestation operation of the TPM can be used as a form of HIDS, if the PCR values in the TPM are representative of the policy-critical aspects of the host's integrity. Thus, this use case will demonstrate the use of integrity measurement to build a TPM-based HIDS.

Existing Trusted Computing Network Access Control

The Trusted Network Connect (TNC) group and Infrastructure Working Group (IWG) of TCG have already proposed necessary specifications for the trusted network access and infrastructure necessary for a fully trusted network access. A brief introduction was given in section 2.2.5.1. The integrity measurement framework serves to augment this scheme by providing measurement of a virtualized guest kernel, and runtime freshness of the measurements.

5.3.1 Implementation of the Trusted Intrusion Detection System

The HIDS to be demonstrated in this use case will utilize a network access control mechanism to restrict the network access of each host, and simultaneously keep a continuous check on the host integrity. Existing mechanisms for network access control include the widely used IEEE 802.1x standard, and also Virtual Private Network (VPN) for remote access to the network. As a proof-of-concept prototype, the implementation discussed in this section will focus only on the low level trusted channel, which is equivalent to the IF-T channel within the TNC protocol framework. OpenVPN [65], which uses Transport Layer Security (TLS) protocol, will be used as the basis of the trusted channel for ease of integration.

5.3.2 TC-aware Virtual Private Network

OpenVPN [65] creates a VPN channel between a VPN client and a VPN server (also known as a VPN concentrator) using TLS encryption [61]. In contrast to other VPN solutions which use IPSec, OpenVPN uses TLS which is a TCP layer encryption. This simplifies the routing of a TLS connection as it uses standard TCP packets. TLS encryption relies on a key agreement algorithm (such as Diffie–Hellman key exchange) for secure handshake and session key generation. After that, a symmetric cipher is used to encrypt the channel using the session key.

In this implementation, both the VPN client and server are modified to be TPM aware, using the TPM extension to the OpenSSL library. Under this setup, a TPM signing key is used as the client identity key in the secure key exchange between the client and server. In order to verify the integrity of the system, the TPM signing key is bound to a known good state of the PCRs. The validity of this signing key has to be communicated to the VPN server in a separate step beforehand, using an AIK-generated signature.

Figure 5.3 shows the TLS handshake and communication between the VPN client and server. The CERTIFY operation is used to sign the TPM key using a TPM AIK key. For simplicity reasons, the verification of the AIK is performed out-of-band of the TLS session (using known methods such as a privacy CA). Alternatively, we can use a client certificate with a TCG-defined Subject Key Attestation Evidence (SKAE) X.509 extension [7]. This is a more formal framework for verifying TPM-based signatures. The result of the CERTIFY operation provides a proof to the server that the user key is a TPM key which can only be used when the TPM PCR matches a defined set of values. This set of pre-defined values would match an integral client state.

During the actual TLS handshake, after the initial exchange of nonce values, the OpenVPN client uses a TPM plugin that calls the TPM to perform the signing operation on the server nonce. This signing operation verifies that the client possesses the user key, and that implicitly implies that the client TPM PCR matches the defined values. Thus, when used together with the integrity framework, the TLS handshake can be used to imply that the client platform is in a valid state.

Refresh period

A refresh period is also defined in the VPN server. After each refresh period, the VPN server requests a re-negotiation of the TLS handshake between the client and server. This re-negotiation generates a new session key, and forces the TPM key on the client end to be used again, indirectly invoking a check on the PCR as well. This means that

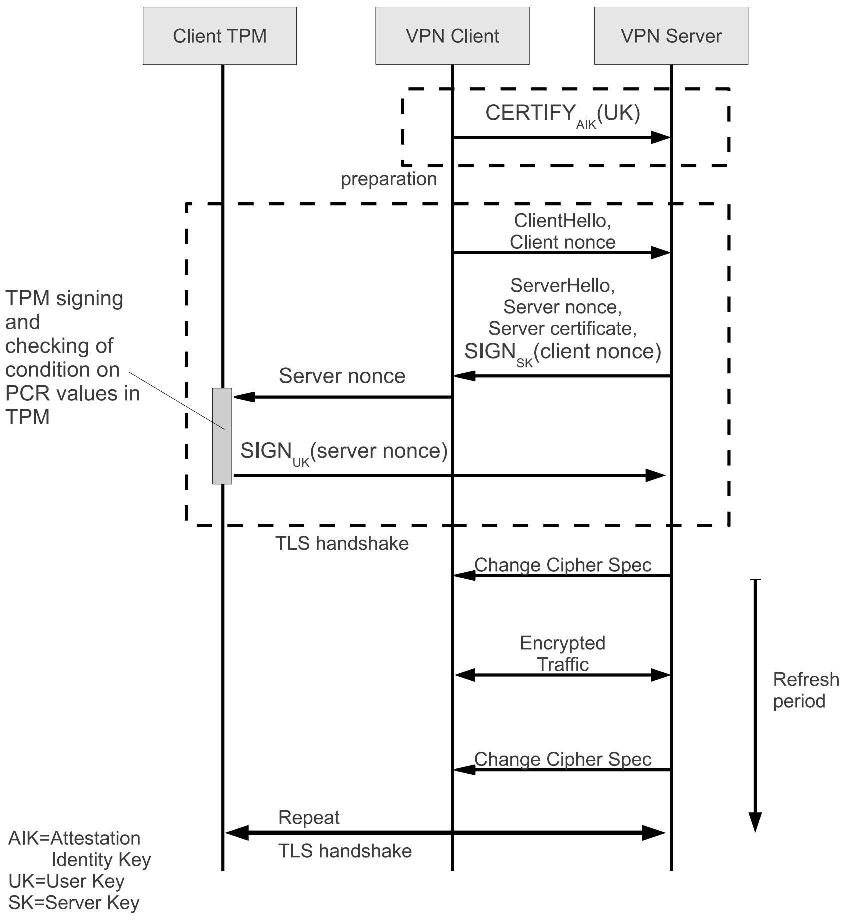


Figure 5.3: Trusted VPN

the integrity on the client end can be re-verified with every TLS re-negotiation cycle. Thus, having a frequent refresh period maintains the freshness of the integrity measurement, as with the disadvantage of higher computational and transmission overhead.

Based on the assumptions that the PCRs are a runtime representation of the integrity of the entire platform, using both the integrity framework and security modules in the guest (as discussed in section 5.1), the VPN server is able to constantly monitor the integrity of all connected clients. In this way, the VPN server can be used as an enforcement point of network-wide platform integrity, automatically controlling clients' access to the network, and disconnecting clients which are compromised with respect to its PCRs measurement. Such setup works effectively like an HIDS, and is able to prevent the proliferation of malicious software spreading throughout a corporate network, by isolating infected hosts from the network. Infected hosts may be placed into a demilitarized zone (DMZ) of the network, where the host can be rectified before re-connecting them back to the trusted network. A similar scheme can be applied to LAN-based operations by replacing a VPN server with a network access control authenticator, such as those based on 802.1X.

This simple setup demonstrates how integrity measurement and trusted computing can be extended to a network-wide scenario to ensure platform integrity. This can be combined with measurement techniques for *Apparmor* profiles as described in section 5.1.3 to form a network-wide MAC policy enforcement.

Areas which are not covered in detail in this implementation are the management of the necessary certificates to prove the correctness of the client's TPM public key, as covered in TCG's specification on SKAE. The high-level policy management in a full TNC deployment is omitted for simplicity. The difference from the existing TNC-like remote attestation scheme is the definition of a refresh period, which determines the freshness of the integrity measurement. This is unique to the integrity measurement framework due to the runtime nature of the measurement, which can be used to produce the most current integrity measurement.

6 Evaluation

This chapter evaluates of the effectiveness of the integrity measurement framework and additional components in capturing the integrity measurement of the entire platform. The two modified Linux Security Modules (LSM) and the two use cases described in the previous chapter are evaluated.

Section 6.1 describes the verification test performed on the system. Functional tests that verify if modification to any of the critical software component would produce an expected change to the PCR values are successfully conducted. A variety of penetration attacks are conducted from within the guest operating system. The results conclude that the system is not compromised under those attack vectors.

Section 6.3 evaluates the performance of the integrity measurement framework, testing both the QEMU and KVM virtualization implementations, together with optimization for state groups. The performance results show that the runtime integrity verification has minimal performance overhead under both situations. During normal OS operation, the guest OS runs at near native performance.

Finally, section 6.4 discusses known problems which are generally applicable to trusted computing based systems, and specific limitations of the integrity measurement. Understanding the problems of the current system establishes the basis for proposing future work on the integrity measurement framework in section 7.1.

6.1 Verification tests

This section summarizes the results of a series of tests performed on the integrity measurement framework. The tests are used to verify that every component in the trust chain is properly measured. This involves 3 verification tests covering static verification of the measurement chain, dynamic attack of the guest kernel and verification of the LSM modules in the guest kernel. These tests serve to check that the integrity measurement framework is functioning as expected, and is able to detect a range of static and dynamic attacks on the system. While formal verifications give a stronger proof, they are generally impractically hard to apply to a large heterogeneous system consisting of hardware, hypervisor, kernel and supporting software. Thus, the testing strategy uses a more 'black-box' approach. Revisiting the threat model discussed in section 3.1, the following verification tests which aim to examine some of the identified attack vectors of the system are discussed.

6.1.1 Test environment

The integrity measurement framework is evaluated on a Linux host running 64-bit Ubuntu 10.04, on a test machine with the following specifications:

- HP Compaq 6535b laptop
- AMD Turion(tm)X2 Dual Core Mobile RM-74 processor
- 4GB of system RAM
- Infineon TPM 1.2

Both the host and guest kernels were modified versions of the Ubuntu Linux kernel version 2.6.31.4. The QEMU and KVM virtual machine uses version 0.11.1 of the QEMU code base, and version 0.6.1 of the TPM emulator.

6.1.2 Simple modification test

The first test verifies the relationship between each component along the trust chain, and the integrity measurement captured via PCR values. For each test case, a single byte change was introduced into each component in a way that does not disrupt its functional behavior. The condition of not modifying the functional behavior was introduced so that the chain can complete every test case normally, without any changes to the operation of the software. The final PCR values were compared with a reference set of PCR values based on unmodified components. This test verifies the effectiveness of the integrity framework against attack vector V4, where permanent changes were made to the boot-up components of the host and guest systems.

Table 6.1 lists the PCRs which were changed when a component was modified. Four separate test chains were performed, with two sets each performed on a system with and without DRTM. For each of these sets, both the single client model (using hardware TPM) and server model (using virtual TPM) were tested. For the client model columns, only one set of values representing the changed PCRs is shown because the client model only uses the hardware TPM (hwTPM). On the other hand, there are two sets of values, separated by a slash, representing changed PCRs on the hardware TPM (hwTPM) and the virtual TPM (vTPM), respectively.

By comparing the results in table 6.1, with the definition of PCR usage in table 2.2, it is clear that the BIOS image influences only PCR 0. PCRs 1,2,3,6 and 7 are extended by the BIOS, but their significance cannot be determined. Changing the Master Boot Record (MBR) has correctly changed PCR 4, which is reserved for the Initial Program Loader (IPL).

Comparing with the PCR usage allocation given in table 4.5, it is clear that Trusted GRUB is appropriately measured in PCR 8 and 9 in two stages. The modules loaded by Trusted GRUB are represented by PCR 14, while PCR 12 is used for the command line instructions used in loading each case. OSLO and its associated helper programs affect PCRs 17 and 19 appropriately and also affect PCR 14 since OSLO itself is loaded by Trusted GRUB. The remaining components are started by the host platform. For the server model, only the virtual TPM is affected since the session manager extends these components into individual virtual TPMs. PCR 0 of the virtual TPM is extended by the session manager, using a combination of PCRs 0-19 on the host. A hash of the pattern database is extended into PCR 20. PCR 21, which is the main PCR used by the integrity measurement framework itself to report measurement state, shows the integrity of the loaded guest kernel and kernel modules. PCR 22 is used by the LSM in the guest kernel to report measured configuration. The guest ramdisk is not explicitly measured, as the integrity of the guest OS is dependent on the kernel integrity and the use of LSM, and not on the integrity of the ramdisk itself.

The conclusion of this modification test suggests that the integrity measurement framework is able to detect static changes in the entire trust chain from hardware to guest VM.

6.1.3 Kernel memory attack

Exploiting the kernel loading interface

Runtime tests were also carried out on the integrity framework to determine the system's ability to detect runtime changes in the guest VM. The first test simulates a rootkit attack from a super-user, or a malicious user who is able to elevate his privilege level to that of a super-user. A single byte change was applied to an existing kernel module and loaded into the kernel using the standard *modprobe* utility. This test is a simulation of the attack vector V1, which uses a guest kernel system call to make changes to the guest memory area.

The test result shows that, when a modified kernel module is loaded, the integrity framework detects the altered module. This changes the integrity state into the unknown state, which is being extended into PCR 21. The integrity framework successfully detects the simulated rouge module.

Exploiting the /dev/mem interface

A second test simulates an attack on an existing vulnerability in the kernel. The */dev/mem* block device was used to make direct read and write access to a part of the kernel code memory. The */dev/mem* block device is a special device file which maps all read and write access to the entire physical memory of the host. Thus, it is a very generic and privileged memory operation that can normally be performed only by the kernel. This attack is a simulation of a variant of attack vector V1, where the guest memory is altered directly.

The use of */dev/mem* is thus security-critical and is often exploited by rootkits and other malicious software to make modifications to the kernel. The only legitimate use of */dev/mem* is to access BIOS and ACPI information necessary for calling video interrupts,

Tested component	PCRs changed			
	Non-DRTM		DRTM	
	client model hwTPM	server model hwTPM / vTPM	client model hwTPM	server model hwTPM / vTPM
BIOS image	0,5	0,5 / 0	0,5	0,5 / 0
MBR (GRUB stage 1)	4	4 / 0	4	4 / 0
GRUB stage 2 (first sector)	8	8 / 0	8	8 / 0
GRUB stage 2 (remaining sectors)	9	9 / 0	9	9 / 0
GRUB command line	12	12 / 0	12	12 / 0
Oslo loader	n.a.	n.a.	14,17	14,17 / 0
Pamplona, beirut, munich loaders	n.a.	n.a.	14,19	14,19 / 0
Host kernel	14	14 / 0	14,19	14,19 / 0
Host ramdisk	14	14 / 0	14,19	14,19 / 0
Database	20	- / 20	20	- / 20
Guest kernel	21	- / 21	21	- / 21
Guest ramdisk	-	- / -	-	- / -
Guest kernel module	21	- / 21	21	- / 21
LSM configuration	22	- / 22	22	- / 22

Table 6.1: Modification test results

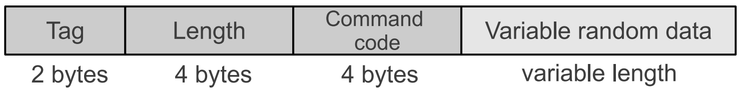


Figure 6.1: Generic TPM command

DOS emulation and so on. Since such information is located below the 1MB memory region, the STRICT_DEVMEM kernel option is used to restrict */dev/mem* access to this region. However, for the purpose of this test, the STRICT_DEVMEM option is disabled to intentionally attack the Linux kernel.

This attack also raised an expected response in the integrity framework, changing the integrity state and modifying PCR 21. This test demonstrates the ability of the integrity framework to detect general forms of modification to code pages in the guest VM, in ways that can be expected from a rootkit.

6.1.4 TPM device attack

Since hardware devices are the only interfaces between the guest VM and the hypervisor, such interfaces can be exploited as in the case of attack vector V2. The TPM interface will be evaluated here, as it is a new interface added to the set of emulated devices in QEMU. A vulnerability in the interface can lead to a potential breach from the guest to the host system. Fuzzing [49] is performed on the TPM interface to check for vulnerabilities. Fuzzing is a penetration testing method, with which random data or random mutations of data are used to discover software vulnerabilities. This can be applied to both communication protocols or file formats. Clear signs of vulnerability in the software can include memory faults and program crashes. A first generic fuzzing attack sends a long random byte stream to the TPM device from the guest. No peculiar behavior was discovered, as most of the messages were rejected by the TPM emulator on the host, based on the invalid header format.

A second attempt uses a more structured approach. By using the common format of all TPM request commands, as shown in figure 6.1, a simple java-based fuzzer was implemented. The fuzzer generates random commands using known tags and command codes, while producing random data in the variable portion. A large stream of such random commands was sent through the TPM stream, without signs of software vulnerability in the virtual machine or TPM emulator. This shows that the parsing of TPM commands is sufficiently robust, with respect to the simple fuzzing attack, and no vulnerability was detected.

6.1.5 Evaluation of integrity measurement using Linux Security Modules

In addition to the runtime integrity framework, the Linux Security Modules (LSM) modified to work with trusted computing was also tested. The *digsig* and *AppArmor* LSMs described in section 5.1 were verified. Both LSMs correctly extend to PCR 21 (see table 4.7.2) as part of the trust chain. In the case of *digsig*, a hash of the signature verification public key is extended into PCR 21. After that, execution of binaries which do not have an embedded signature will fail. There are, however, exceptions such as scripts and bytecode files, which cannot be verified, as they are not in the ELF format. Since *digsig* is restricted to verification against a single key at a time, this restricts the signing authority to be a single source. This places a restriction on large-scale deployment, since all binaries must be signed by the same authority.

For the case of *AppArmor*, the hash of all loaded and unloaded profiles are extended into PCR 21. They are extended in the order in which the profiles are loaded or unloaded. The loading operation is differentiated from the unloading operation via an additional extend operation with a known value. This has been verified in the working implementation. After all the *AppArmor* profiles are loaded, processes matching profiles in enforcement mode, are restricted only to resources defined in the profile. Since the *AppArmor* log is not extended into the PCR, the PCR 21 value only effectively assures the enforcement of a correct set of profiles.

The implemented measurement mechanism in both LSMs, only allows the configuration of each of the LSM to be measured. The enforcement of the respective restrictions still depends entirely on the correct operation of each LSM. This is especially true since logging information is not extended into the PCR.

6.1.6 Integrity of PCR extensions

The trustworthiness of the entire integrity framework is only as good as the integrity of the PCR measurement results. Thus, it is critical that the PCR measurement itself cannot be covertly manipulated. The PCR measurements are affected by the extend and reset TPM operations, which are controlled by the TPM filter discussed in section 4.6.2. The TPM filter blocks all extend and reset operations to PCRs below 22 from the guest. With a small kernel modification, PCR 22 is blocked from the user space, and is verified that only PCR 23 can be manipulated from within the guest OS userland. PCR 22 is accessible only from within the guest kernel and is used by the kernel to report configuration of the LSM.

This restriction clearly defines the role of guest applications and the guest kernel to use PCR 23 and 22, respectively, while all other PCRs are managed exclusively by the host. Within the host, in addition to the boot time measurements, extending to the PCR is controlled by the session manager, which isolates integrity measurements between concurrent VMs.

6.2 Evaluation of integrated use cases

The sections 6.2.1 and 6.2.2 discuss the evaluation of the use cases presented earlier in sections 5.2 and 5.3. These use cases showcase the overall end-to-end usefulness of the integrity measurement framework. The system is checked if it operates as expected, and problems of the system are discussed.

6.2.1 Evaluation of trusted authentication and storage

The trusted authentication and storage use case demonstrate the use of integrity measurement applied to an offline platform. By using the PAM framework, the authentication mechanism and local storage encryption is protected via integrity measurement. As covered in section 5.2, the trusted authentication has two main modes of operation. The first mode uses individual storage keys for each user to encrypt authentication hash. The second mode uses a global storage key for encryption of all authentication hashes. In addition, an option is provided to use binding or sealing keys for both modes. All four mode combinations were verified to work as expected.

The mode of using individual storage keys has the advantage of having no single key that can become the single breaching point into the authentication scheme. However, under the assumption that TPM keys are secure, neither mode has any cryptographic advantage over the other. The use of binding keys instead of sealing keys ignores the PCR values when using the key, thus does not making use of the integrity measurement framework at all. The use of binding keys only hinders password attack on the authentication scheme without taking integrity into consideration. The sealing keys are bound to the PCRs 0-22, and thus depend on the integrity of the entire platform. Changes in the PCR values will cause authentication to fail. In order to build a robust system, a fail-safe scheme of reverting to the standard UNIX login is included to allow system administrators to repair the system in case of an integrity failure. This can be achieved by creating a specially privileged account based on UNIX login with a sufficiently strong password.

The storage encryption uses *encfs* which uses a user password to decrypt the user's encryption key. This is modified to combine the encrypted authentication hash with the user password to decrypt the storage key. This modified scheme is thus dependent on both a successful authentication and an integral platform. A changed integrity measurement would cause the decryption of the authentication hash to fail, and consequently, the user's storage key cannot be extracted.

6.2.2 Evaluation of Intrusion Detection System

The Intrusion Detection System (IDS) discussed in section 5.3 is a host-based IDS built on OpenVPN. OpenVPN uses TLS for building its encrypted channels, which use a standard TCP data stream. This has the advantage of being easier to route but also has the inherent disadvantage of not hiding session information in the TCP layer, such as port and sequence numbers. One disadvantage is that it allows traffic patterns of connected clients to be easily observed by an eavesdropper.

The IDS implementation uses a modified OpenVPN which utilizes a TPM key for the TLS handshake. The key used is a TPM signing key, which has a usage condition bound to PCRs 0-22. It has been verified that any change in the underlying integrity measurement would prevent the key from being used. Under the server model, a virtual TPM is used to enforce the key usage condition. The verification process has uncovered a bug in the TPM emulator which has been fixed thereafter.

In order to maintain high freshness of integrity measurement, the server is configured with a low re-negotiation timeout (defined using the *reneg-sec* parameter) and a low transition window (defined using the *tran-window* parameter). This would force the client to perform a TLS re-negotiation at every *reneg-sec* second, where the re-negotiation must complete within *tran-window* seconds. This has been verified to successfully re-check the integrity measurement of the client at every *reneg-sec* second. A re-negotiation incurs additional 23 kbytes of transmission overhead per re-negotiation. The system has fulfilled its purpose of being able to constantly maintain the platform integrity of its connected clients, effectively working as a network enforcement point of a host-based IDS.

6.3 Performance

6.3.1 Performance overhead

In addition to the measurement strength of the integrity measurement framework, its performance is also evaluated. Both the binary translation and hardware virtualization implementation were tested. Table 6.3 summarizes the performance of both implementations, as overhead over the same virtual machine without integrity measurement. Since most of the verification is performed only during VM start-up, the boot-up time was measured and additional CPU and memory benchmarking is taken during normal operation of the guest VM.

The boot-up performance is calculated based on the average boot-up time from the start of the VM to the first login prompt. CPU and disk operations within the VM are measured using the *sysbench* [33] benchmarking software being executed within the VM. The table shows a comparison of KVM and QEMU with and without integrity measurement.

Due to the difference in virtualization techniques, QEMU is generally slower than the KVM implementation, except for memory transfer which has more initial overhead in KVM due to the additional page faults when the memory is just initialized. The integrity measurement of the guest kernel itself incurs very minimal overhead in both implementations.

Figure 6.3 compares the boot time for the KVM and QEMU VMs, when connected to a virtual TPM (vTPM) and a hardware TPM (hwTPM). The results show a very slight delay when communicating with a hardware TPM. This confirms the fact that the hardware TPM, being connected on the slow LPC bus, is slower than an emulated TPM running on the CPU. 88 TPM operations were carried out during the entire boot process, which gives an average of 3ms difference between a single TPM operation in hardware and virtual TPM.

	Native(host)	KVM(no IM)	KVM (IM)	QEMU(no IM)	QEMU(IM)
Boot time	-	8.80s	9.10s	59.22s	60.05s
CPU-intensive benchmark	38.12s	38.51s	40.53s	56.72s	55.72s
Memory-intensive benchmark	1.30s	8.79s	24.90s	3.03s	3.06s

IM = Integrity measurement

Table 6.2: Comparing performance overhead of virtualization and integrity measurement

	KVM+vTPM	KVM+hwTPM	QEMU+vTPM	QEMU+hwTPM
Boot time	8.28s	8.63s	59.21s	59.42s

Table 6.3: Comparing performance between virtual TPM and hardware TPM model

6.3.2 State optimization

As discussed in section 3.6.5, a set of modules is grouped into each integrity state and this provides some performance benefits. Table 6.4 shows the performance of the integrity measurement during the verification of modules in the guest OS, with and without integrity state grouping. The first measured parameter is the number of comparisons performed against the pattern database during the boot-up sequence, and it shows a slight reduction with state grouping. The second parameter measures the loading time required for the kernel and all kernel modules, resulting in a slight reduction in the module loading time as well. Due to the nature of the binary-translation type virtualization (QEMU), more comparisons against the pattern database are needed.

	Number of comparisons		module loading time	
	QEMU	KVM	QEMU	KVM
Without grouping	51061	30522	63.91s	10.46s
With state grouping	47683 (6.6%)	26351 (13.7%)	63.21s (1.10%)	9.69s (7.36%)

% reduction of comparison in brackets

Table 6.4: State grouping optimization results

6.4 Known problems

6.4.1 Existing problems of Trusted Computing

This section discusses some known problems of trusted computing, which have been investigated to various degrees. Version 1.2 of the TPM and TSS specifications uses SHA-1 as the only hashing algorithm defined in the standard. This is seen as a restriction due to the SHA-1 algorithm being deemed insecure in the long-term by some cryptographic experts [41, 71].

In terms of remote attestation, there is a lack of a well implemented public infrastructure to verify endorsement keys of a TPM, which is necessary to protect the privacy of users. To date, no genuine platform certificates are issued by any platform manufacturers, and only one TPM manufacturer actually creates signed endorsement certificates. To the best of the author's knowledge, only one experimental public privacy certification authority exists to support a purely anonymous creation of AIK in the public domain. This greatly restricts the practical application of attestation in the public domain. However, remote attestation can still be used in the context within an organization, by generating AIK signed by the organization. This loses the privacy benefit as the identity of the attester is usually known to the organization during the signing process.

Implementation of the Trusted Software Stack (TSS) also faces a number of security risks. The Linux TSS service uses standard TCP sockets to listen for incoming connections on the local loopback interface. Thus, restricting access to the TSS service is possible only by crafting well-defined firewall rules to prevent unauthorized access to the TSS service. This is important to prevent unauthorized changes to the PCR values through the TSS service, or unauthorized access to TPM keys by malicious software.

As mentioned in section 4.5.2, a trusted link between the hardware and virtual TPM remains an open question in terms of using TPM in a virtualized environment. For the single virtual machine case, this can be trivially resolved by using the actual hardware TPM for the virtual machine, as demonstrated in the "client" TPM model (see section 4.6.1) used in this dissertation.

6.4.2 Current limitations of integrity measurement

One major limitation of the integrity measurement framework is that only executable code is measured and used as a basis for integrity. Interpreted code, such as bash scripts, java and python programs, is not considered in the measurement process. Furthermore, dynamic data in memory used in the kernel or user applications are also not measured. All of these pose a significant threat, as the data or critical configuration information can greatly influence the output of executables. Using a MAC policy enforcement provides a solution to some of these potential problems.

7 Conclusion

As discussed in chapter 1, the goal of this dissertation is to find a solution to efficiently and effectively measure a monolithic kernel to generate a live integrity measurement that detects changes in runtime. This integrity measurement has been integrated with Trusted Computing techniques to generate a continuous chain-of-trust from the firmware to the hypervisor and host, guest kernel and applications, such that the entire operating system is verifiable in runtime. This goal has been met through the integrity framework which is proposed, described and evaluated in the previous chapters.

The major novel contribution is a technique to measure and monitor runtime changes of a Linux operating system kernel and determine if all executable code within the kernel is consistent with the compiled binary reference of the same kernel. This technique is integrated with the Trusted Platform Module (TPM), to form a complete verification chain from firmware to applications. With this 'live' integrity measurement that was not available before, security sensitive applications are able to use this integrity measurement to protect critical resources or attest itself to a 3rd party. The practical application of the measurement framework is demonstrated through integration with existing security mechanisms in Linux, and applied to two uses cases: trusted authentication and encryption, and trusted intruder detection system.

The effectiveness of the measurement framework has been tested to detect both static and dynamic modifications to the system. The performance overhead of applying integrity measurement is low for both proposed virtualization techniques. Based on the evaluation of the integrity measurement framework and the supporting mechanisms and tools, the entire trust chain has met the initial goal of a complete solution to effectively measure a complex monolithic operating system, using a modern Linux kernel as an example. The low performance overhead shows that this can be carried efficiently using existing virtualization technology for practical use.

7.1 Future work

The work pioneered in this dissertation is, however, far from completion. There is much potential to extend the work in various areas. One major area which is not handled in this dissertation is the management infrastructure that is necessary for the verification of measurement values such as the Platform Configuration Registers (PCR) between remote parties during an attestation. This problem is complicated by the fact that an organization normally has a heterogeneous collection of systems, with different hardware, operating systems, applications and a variety of versions for each of these components. This would imply that a proper versioning mechanism is necessary for the pattern database discussed in this dissertation. Furthermore, cryptographic signing of the database and an appropriate Public Key Infrastructure (PKI) may be needed to verify the integrity of the database versions. In addition, the use of TPM counters may be needed to implement database revocation and prevent roll-back attacks. Future work in this area will complement the work in this dissertation, which focuses mainly on the core technique for integrity measurement of the kernel.

In terms of extension to the integrity measurement framework, possible future work involves applying the same technique to other operating systems such as Windows, BSD and Mac OS X. The initial success has been achieved in the case of Windows, although more in-depth work is necessary to adapt the technique to the special behavior of the Windows kernel. The various flavors of the BSD operating system stand to benefit from the integrity measurement framework, as they share the same ELF binary format used by Linux.

The application of integrity measurement using newer processors which support nested page tables can also be investigated. This new feature allows for even faster virtualization, by performing virtualized memory management using hardware. Adapting the memory management technique proposed in this dissertation to nested page tables will allow for near native virtualization speed.

In order to produce more fine-grained measurement of an operating system, application-specific modifications can be made to security sensitive software, so that they can make use of the integrity measurement natively. This has advantages over generic approaches like using Linux Security Modules (LSM), because the application can perform verification of application-specific data. Such applications can also make use of the integrity measurement directly for security-sensitive operations.

On the system level, the use of integrity measurement in forward-looking scenarios, such as building a trusted data center, can be investigated. One proposal of a trusted data center based on Trusted Virtual Domains (TVD) has been proposed in [26]. Such a concept can be extended with runtime integrity measurement, by revising the framework to maintain runtime integrity of the TVD as a whole.

7.1.1 Open source

In order to assist further development of the work discussed in this dissertation, the implementation of the integrity measurement framework and related tools for building the pattern database is released on a public repository [60] under an open source license. This is done in order to encourage future development, as well as adaptation of the framework to other virtualization techniques and operating systems.

List of Figures

List of Tables

Nomenclature

ACPI	Advanced Configuration and Power Interface
AIK	Attestation Identity Key
AMD64	64-bit processor from AMD
AR	Access Requestor
AVL	Available-to-software bits
BSD	Berkeley Software Distribution
CLR	Common Language Runtime
CRTM	Core Root-of-Trust for Measurement
CRTR	Core Root-of-Trust for Reporting
DAA	Direct Anonymous Attestation
DEV	Device Exclusion Vector
DMA	Direct Memory Access
DMZ	Demilitarized zone
DRTM	Dynamic Root-of-Trust for Measurement
EK	Endorsement Key
ELF	Executable and Linkable Format
EPT	Extended Page Table
GDB	Gnu Debugger
HIDS	Host-based Intrusion Detection System
hwTPM	Hardware Trusted Platform Module
i386	80386 processor from Intel

ia64	64-bit processor from Intel, incompatible with AMD64
IDS	Intrusion Detection System
IMA	Integrity Measurement Architecture
INVLPG	Invalid page fault
IO	Input output
IOMMU	Input-Output Memory Management Unit
IPL	Initial Program Loader
IWG	Infrastructure Working Group
JIT	Just-In-Time
KEK	Key Encryption Key
KVM	Kernel-based Virtual Machine
LKIM	Linux Kernel Integrity Measurement
LSM	Linux Security Module
MAC	Mandatory access control
MBR	Master Boot Record
MMIO	Memory-mapped Input Output
MMU	Memory Management Unit
NIDS	Network-based Intrusion Detection System
NPT	Nested Page Table
NTLDR	Windows NT loader
NX	No-execute bit
PAE	Physical Address Extension
PCA	Privacy Certification Authority
PCI	Peripheral Component Interconnect
PCR	Platform Configuration Register
PDE	Page Directory Entry

PDP	Policy Decision Point
PDPT	Page Directory Pointer Table
PE	Portable Executable format
PEP	Policy Enforcement Point
PKI	Public Key Infrastructure
PML4T	Page-map level-4 Table
PNP	Plug-and-Play
PTE	Page Table Entry
RW	Read/Write bit
SKAE	Subject Key Attestation Evidence
SOAP	Simple Object Access Protocol
SRK	Storage Root Key
T/OS	Trusted Operating System
TB	Translation block
TBS	Trusted Base Service
TC	Trusted Computing
TCB	Trusted Computing Base
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCS	Trusted Core Services
TDDL	Trusted Device Driver Library
TLB	Translation lookaside buffer
TLS	Transport Layer Security
TNC	Trusted Network Connect
TPM	Trusted Platform Module
TSP	Trusted Service Provider

TSPI	Trusted Service Provider Interface
TSS	Trusted Software Stack
TVD	Trusted Virtual Domain
VM	Virtual machine
VMCB	Virtual Machine Control Block
VPN	Virtual Private Network
vTPM	Virtrual Trusted Platform Module
vTPM	Virtrual Trusted Platform Module
x86	family of 32-bit processors for the PC, starting from the 80386. Also loosely refer to the 64-bit instruction set processor from AMD and Intel.

Bibliography

[1] Ubuntu kernel team resources, Sep 2000. http://kernel.ubuntu.com/.

[2] Tcpa main specification version 1.1b. Technical report, Trusted Computing Group, Feb 2002. http://www.trustedcomputinggroup.org/resources/tcpa_main_specification_version_11b.

[3] Bsign: Corruption & intrusion detection using embedded hashes, Aug 2003. http://packages.ubuntu.com/karmic/bsign.

[4] Openbsd unix operating system version 3.3, May 2003. http://www.openbsd.org/33.html.

[5] Pc client work group pc client specific tpm interface specification (tis) version 1.2. Technical report, Trusted Computing Group, July 2005. http://www.trustedcomputinggroup.org/developers/pc_client.

[6] Pc client work group specific implementation specification for conventional bios specification. Technical Report Version 1.2, Trusted Computing Group, July 2005.

[7] Subject key attestation evidence(skae) extension. Technical report, TCG Infrastructure Workgroup, June 2005. http://www.trustedcomputinggroup.org/developers/infrastructure.

[8] Tpm main specification. Technical report, Trusted Computing Group, July 2007. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[9] Trusted computing group specification architecture overview version 1.4. Technical report, Trusted Computing Group, Aug 2007. http://www.trustedcomputinggroup.org/resources/tcg_architecture_overview_version_14.

[10] Apparmor, Nov 2008. http://www.novell.com/linux/security/apparmor/.

[11] Bochs ia-32 emulator project, Jun 2008. http://bochs.sourceforge.net.

[12] Tpm emulator, Nov 2008. http://tpm-emulator.berlios.de/.

[13] Trusted network connect (tnc). Technical report, Trusted Computing Group, Feb 2008. http://www.trustedcomputinggroup.org/developers/trusted_network_connect.

[14] Vmknoppix, Aug 2008. http://www.rcis.aist.go.jp/project/knoppix/vmknoppix/index-en.html.

[15] Gnu hurd, Oct 2009. http://www.gnu.org/software/hurd/hurd.html.

[16] Kernel based virtual machine version 0.11.1, Dec 2009. http://www.linux-kvm.org/.

[17] Qemu open source processor emulator, July 2009. http://www.qemu.org/.

[18] Tcg software stack (tss) specification version 1.2. Technical report, Trusted Computing Group, March 2009. http://www.trustedcomputinggroup.org/resources/tcg_software_stack_tss_specification.

[19] Xen hypervisor, Nov 2009. http://www.xen.org/.

[20] D. Gordon S. Hallyn Makan Pourzandi V. Roy A. Apvrille. Digsig: Run-time authentication of binaries at kernel level. In *Proceedings of LISA 2004: Eighteenth Systems Administration Conference*, pages 59–66, Nov 2004.

[21] M Luk N Qu A Perrig A Seshadri. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *21st ACM Symposium on Operating Systems Principles*, October 2007.

[22] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[23] Sirrix AG. Trustedgrub, Nov 2009. http://sourceforge.net/projects/trustedgrub/.

[24] AMD. *AMD64 Architecture Programmer Manual*, vol. 2 system programming edition, Sept 2007.

[25] Robert Baumgartl, Martin Borriss, Hermann Härtig, Claude-Joachim Hamann, H. Hartig, Cl. j. Hamann, Michael Hohmuth, Jean Wolter, Lars Reuther, and Sebastian Schönberg. Dresden realtime operating system. In *in Proceedings of the First Workshop on System Design Automation (SDA'98*, pages 205–212, 1998.

[26] Stefan Berger, Ramón Cáceres, Dimitrios Pendarakis, Reiner Sailer, Enriquillo Valdez, Ronald Perez, Wayne Schildhauer, and Deepa Srinivasan. Tvdc: managing security in the trusted virtual datacenter. *SIGOPS Oper. Syst. Rev.*, 42:40–47, January 2008.

[27] Erich Stefan Boleyn Bryan Ford. Multiboot specification version 0.6.96, Jan 2010. http://www.gnu.org/software/grub/manual/multiboot/multiboot.html.

[28] Ero Carrera. pefile - a multi-platform python module to read and work with portable executable files, Mar 2009. http://code.google.com/p/pefile/.

[29] T. Knothe C.H. Suen, M. Loy. Portable trusted file encryption service. In *Retrust 2008 Conference*, 2008.

[30] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. A protocol for property-based attestation. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 7–16, New York, NY, USA, 2006. ACM.

[31] Oracle Corporation. Virtualbox version 3.2.0, May 2010. http://www.virtualbox.org/.

[32] Alexandre Julliard et al. Wine version 1.2.1, Oct 2010. http://www.winehq.org.

[33] Alexey Kopytov et al. Sysbench: a system performance benchmark version 0.4.10, Dec 2008. http://sysbench.sourceforge.net/.

[34] Erich Boleyn et al. Grand unified bootloader version 1.97, October 2009. http://www.gnu.org/software/grub/.

[35] H. Peter Anvin et al. The syslinux project version 3.86, April 2010. http://syslinux.zytor.com/wiki/index.php.

[36] Martin Abadi Mark Aiken Paul Barham Manuel Fahndrich Chris Hawblitzel Orion Hodson Steven Levi Nick Murphy Bjarne Steensgaard David Tarditi Ted Wobber Brian D. Zill Galen Hunt, James R. Larus. An overview of the singularity project. Technical report, Microsoft Research, Oct 2005. MSR-TR-2005-135.

[37] Simson Garfinkel and Gene Spafford. *Web Security and Commerce*. O'Reilly and Associates, 1997.

[38] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[39] Intel. *Intel 64 and IA-32 Architectures Software Developerś Manual*, volume 3a: system programming guide, part 1 edition, June 2009.

[40] Bernhard Kauer. Oslo: improving the security of trusted computing. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–9, Berkeley, CA, USA, 2007. USENIX Association.

[41] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2n̂ work. Cryptology ePrint Archive, Report 2004/304, 2004. "http://eprint.iacr.org".

[42] Richard B. Kieburtz. P-logic: Property verification for haskell programs, 2002. the programatica project, http://www.cse.ogi.edu/pacsoft/projects/programatica, 2002.

[43] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[44] H. A. Lagar-Cavilla L. Litty and D. Lie. Hypervisor support for identifying covertly executing binaries. In *17th USENIX Security Symposium*, pages 243–258, 2008.

[45] Siarhei Liakh. Ro/nx protection for loadable kernel modules, Oct 2010. https://patchwork.kernel.org/patch/90047/.

[46] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. Linux kernel integrity measurement using contextual inspection. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*, pages 21–29, New York, NY, USA, 2007. ACM.

[47] R. Meushaw and D. Simard. Nettop: Commercial technology in high assurance applications. 2000. http://www.vmware.com/pdf/TechTrendNotes.pdf, 2000.

[48] Microsoft Corporation. Secure startup-full volume encryption: Technical overview. Technical report, Microsoft Corporation, April 2005.

[49] B. P. Miller, L. Fredrikson, and B. So. An empirical study of the reliability of unix utilities. *Comm. of the ACM*, 33(12):32, December 1990.

[50] Dod trusted computer system evaluation criteria, December 1985. DoD 5200.28-STD.

[51] Philippe Oechslin. Making a faster cryptanalytical time memory trade-off. In *CRYPTO 2003, 23rd Annual International Cryptology Conference*, Aug 2003.

[52] B.D. Carbone M. Sharif M. Wenke Lee Payne. Lares: An architecture for secure active monitoring using virtualization. *IEEE Symposium on Security and Privacy*, pages 233–247, May 2008.

[53] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. European Conference on Computer Systems:15–27, 2006.

[54] X Jiang D Xu R Riley. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection*, pages 1–20. Springer, Sept 2008.

[55] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *Operating Systems Review*, 42(5):95–103, 2008.

[56] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a tcg-based integrity measurement architecture. *13th USENIX Security Symposium*, pages 223–238, 2004.

[57] K. Seifried. Honeypotting with vmware: basics. Jan 2002. http://www.seifried.org/security/ids/20020107-honeypot-vmware-basics.html.

[58] Frederic Stumpf, Andreas Fuchs, Stefan Katzenbeisser, and Claudia Eckert. Improving the scalability of platform attestation. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 1–10, New York, NY, USA, 2008. ACM.

[59] C.H. Suen. Vimm: Runtime integrity measurement of a virtualized operating system. *Journal of Universal Computer Science*, 16(4):554–576, feb 2010.

[60] C.H. Suen. Open source secure qemu and kvm virtual machine for research, Dec 2011. http://sourceforge.net/projects/qemu-secure/.

[61] E. Rescorla T. Dierks. The transport layer security (tls) protocol version 1.1. Technical report, IETF Network Working Group, Apr 2006. http://www.ietf.org/rfc/rfc4346.txt.

[62] H Eiraku K Tanimoto K Omote T Shinagawa. Bitvisor: a thin hypervisor for enforcing i/o device security. volume ACM/Usenix International Conference On Virtual Execution Environments of *Visors*. ACM, 2009.

[63] Tanenbaum, Herder, and Bos. Can we make operating systems reliable and secure? *COMPUTER: IEEE Computer*, 39, 2006.

[64] Fedora Project Documentation Team. *Fedora 13: Security-Enhanced Linux User Guide*. Fultus Corporation, 2010.

[65] OpenVPN Technologies. Open vpn version 2.1 rc 19, July 2009. http://www.openvpn.net.

[66] Rebekah Leslie Andrew Tolmach Thomas Hallgren, Mark P. Jones. A principled approach to operating system construction in haskell. volume 10th ACM SIGPLAN International Conference on Functional Programming, 2005.

[67] Petr Machata Ulrich Drepper, Roland McGrath. elfutils - a shared library which allows reading and writing of elf files on a high level, Apr 2009. https://fedorahosted.org/elfutils/.

[68] R Sailer L van Doorn R Perez. vtpm: Virtualizing the trusted platform module. In *15th USENIX security Symposium*, July 2006.

[69] Steven J. Vaughan-Nichols. New approach to virtualization is a lightweight. *IEEE Computer*, 39(11):12–14, 2006.

[70] VMware. Vmware thinapp version 4.6, August 2010. www.vmware.com/products/thinapp/.

[71] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full sha-1. In *In Proceedings of Crypto*, pages 17–36. Springer, 2005.

[72] Oliver Welter. Data protection and risk management on personal computer systems using the trusted platform module. *University library of the Munich University of Technology*, 2008. http://mediatum2.ub.tum.de/node?id=646387.