Journal of Universal Computer Science, vol. 16, no. 4 (2010), 554-576
submitted: 10/10/09, accepted: 21/12/09, appeared: 28/2/10 © *J.UCS*

VIMM: Runtime Integrity Measurement of a Virtualized Operating System

Chun Hui Suen
(TU Muenchen, Germany
chunhui.suen@mytum.de)

Abstract: This paper discusses the design of the Virtualization Integrity Measurement Monitor (VIMM) framework, which aims to provide runtime integrity measurement of a virtualized guest OS. Kernel memory and additional hardware state changes are constantly monitored and aggregated into a combined guest OS state, which is reported to a Trusted Platform Module (TPM), thus providing a trusted integrity measurement in runtime. This measurement can then be used for data protection (sealing of secret keys) and remote attestation based on the runtime integrity of the guest OS.
Key Words: Security and Protection, Management
Category: D.4.6, K.6.5, D.2.9

1 Introduction

Managing trust in an operating system (OS) is an important security concern, and an increasingly difficult task due to the complexity of modern OS. In this context, I define trust as the confidence that an entity or system operates as predicted. In an ideal case, this would mean that the behaviour of a system matches its given specifications exactly. However, the verification of the binary code against its specifications, is a non-trivial task, involving static and possibly dynamic software analyses. Thus, I will restrict the definition of trust measurement in this paper, to that of the integrity of its binary code of the software, such that the behaviour of the software is consistent with any previous instances of itself.

An integrity measurement is a measure of some parameters of a component or system, in order to characterise its behaviour. 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 fails 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 [sel08] or AppArmor [app08], controls the interaction between processes and critical system objects (resources) rather than detecting particular code signature of malicious software. They also suffer from the same weakness if the kernel itself is compromised, due to the lack of runtime monitoring and a contineous 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 dependant components, such that any change along this measurement chain is detectable.

The work in trusted computing as described in section 2.1, solves the problem of chain of trust, but failed to address the runtime nature of integrity measurement. Drawing from the trusted computing concept proposed by the Trusted Computing Group (TCG) [TCG07b], and previous work in property-based attestation [CLL+06], kernel-space monitoring [LWPM07, LQPS07, JXR08, CSP08], an extension of existing kernel-space monitoring techniques with binding to the Trusted Platform Module (TPM), is discussed here. This paper proposes the design of the Virtualization Intergrity Measurement Monitor (VIMM) framework. The VIMM 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 paper are based on previous works, and described in more detail in sections 2.1 and 2.2.

2 Related work

2.1 Trusted computing and property-based attestation

The Trusted Computing Group (TCG) produced the TCG specifications for a chain-of-trust based on a hardware root of trust [TCG07b], known as the Trusted Platform Module (TPM) [TCG07a]. The original integrity measure as proposed by the TCG, termed "binary attestation" from previous literature [CLL+06], was demonstrated by an implementation from IBM called Linux Integrity Measurement Architecture(IMA) [SZJv04], which has been integrated into the mainstream Linux kernel since version 2.6.30. IMA 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 [CLL+06], leading to the development of property-based attestation [CLL+06], and how it 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 the shortcoming in that it cannot verify the runtime behaviour of a component or system, as they are only measured once during load time. To overcome this problem, I took into account ideas from runtime kernel-space monitoring techniques: such as Linux Kernel

Integrity Measurement (LKIM) in [LWPM07] and nickle [JXR08], to directly monitor the memory of a guest virtual machine, in order to determine its integrity state, and also other proposals described in the next section. This allows for the integrity of the guest machine to be continuously monitored, throughout its entire uptime.

2.2 Security VM and hypervisors

A number of security focused virtual machines and hypervisors were proposed and already exist. The nickle [JXR08] virtual machine, based on QEMU virtual machine [qem09], 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 on different virtualization tools as well. However, the implementation is limited to software-only solution, and does not utilize hardware virtualization which has become widespread in the past years. Argos [arg06] is another modification to QEMU, targeted at honeypot implementation, to correlate buffer-overflows to their respective incoming attack vectors.

Bitvisor [bit09] and Secvisor [LQPS07] are two hypervisors, with a small footprint, focused on security applications. Both are using hardware-based virtualization provided by Intel and AMD, thus making the hypervisor much simpler compared to software or para-virtualization solutions. Bitvisor implements mandatory encryption by virtualizing specific hardware drivers to provide transparent disk and network encryption. Secvisor is focused on protection of kernel space memory by enforcing mandatory w$\otimes$x mode and kernel code measurement. w$\otimes$x operation means that code pages are either executable or writable but not both, thus preventing attacks, like buffer overflow in the kernel.

2 prototypes are presented in this paper in sections 8.1 and 8.2. The former is based on QEMU [qem09] which uses binary translation while the latter, based on hardware virtualization support. Both will be discussed in detail in their respective sections.

3 Assumptions and Threat model

3.1 Threats

The goal of achieving a trustworthy integrity measurement of the entire software stack (OS and applications), would require a chain-of-trust from the hardware (as provided by the core root-of-trust of the TPM) to the user application. Thus, possible threats would be any attack which can compromise any of the component along this chain. Since many applications already exist to protect (with the help of the kernel) user applications, this paper will focus mainly on the threat of:

T1: malicious code change in the guest kernel

T2: malicious code change in the hypervisor/virtualization layer

Preventing or detecting threat T1 is the main goal of the VIMM framework, while T2 relates to the security strength of the VIMM hypervisor itself. 4 possible attack vectors are considered below:

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

V2: exploiting hypercall to hypervisor

V3: memory write into the hypervisor code or data

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

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

Exploitation of the hypercall (V2) in VIMM can be reduced to a minimal by exposing as little interfaces as possible, to reduce the attack surface. As proposed in this paper, the only points of interaction between the guest kernel and the VIMM framework are through the emulated devices (TPM and Bios functions).

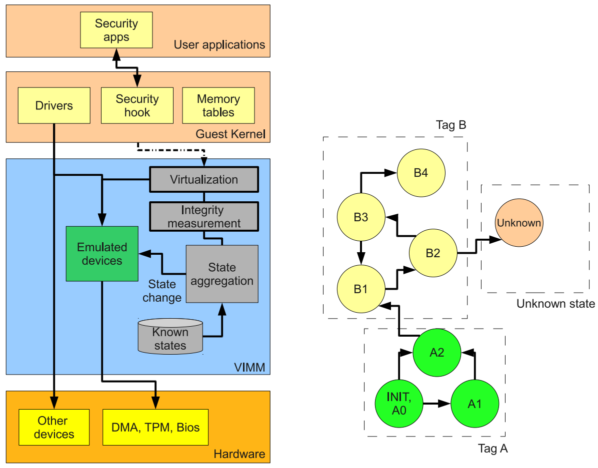
V3 relates to the general problem of direct memory write into the hypervisor memory space either from software (processor controlled) or through hardware via the DMA channel, PCI bus or Bios function. Software-based attack is prevented via paging control of the guest kernel, while hardware-based attack 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 bus-mastering 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 8.8. Thus, any modification to the hypervisor, its data and the guest kernel will be detected as it results in a difference in hash measurement during the trusted boot process.

4 Integrity Model

4.1 Overview

Figure 1a shows an overview of the VIMM framework. The functional components of the hypervisor are: virtualization, integrity measurement, state aggregation and device emulation. The virtualization mechanism handles mainly



(a) VIMM System overview (b) System state graph

Figure 1: VIMM Hypervisor

memory paging in the guest, to restrict the accessible pages from the guest kernel, and detect new code pages loaded in the kernel. The new code pages are measured and a new state is computed. The implementation goals of the VIMM framework are:

1. to have a small footprint (code size and memory usage)
2. to achieve runtime integrity measurement of a guest OS
3. to support easy update of measurement reference

To achieve the first goal of a small footprint, while still being efficient, VIMM is designed to sit as a thin layer between the guest OS and underlying hardware. In order to perform runtime integrity measurement, control is passed to the VIMM framework on intercepted instructions, interrupts, IO port or memory page access, which allow VIMM to re-compute and update the integrity measurement if necessary. The third goal is achieved by having a pattern database (see section 5.2.1) loaded as separate modules and is discussed in detail in section 8.8.

4.2 System states

Measurement of kernel memory space can be generalized into a system state, consisting of measurable parameters. A generalized system state can consist of the guest kernel memory pages, CPU states (registers, virtualization states) and also hardware states. To reduce the memory overhead in the hypervisor, a hash of the measured properties is used to uniquely identify each state.

4.3 Aggregated state or "Tag"

Based on the concept of property-based attestation, an aggregated state, or a "tag", representing a level of trustworthiness of the system, is a more useful result compared to the raw system state or system state hash. This is achieved by comparing the system state or state transition against a given state model. Figure 1b shows such a state model. Each circle is a unique system state or state hash, and a group of states is classified under a single "tag". Eventually, a unique identifier for the tag is extended into the TPM upon a tag transition. If the state leaves the graph completely, then the system goes into an undefined (untrusted) state, represented by a special "unknown" tag.

In the VIMM framework, the state model is defined by a pattern database (see section 5.2.1) of verified states and their respective tags; each verified state represents a unit of executable code running in the guest OS (such as kernel, kernel drivers and so on). Additional conditions on disallowed hardware states can also be placed on each tag. Section 8.8 provides a more detailed discussion on the complete state graph used in the VIMM framework.

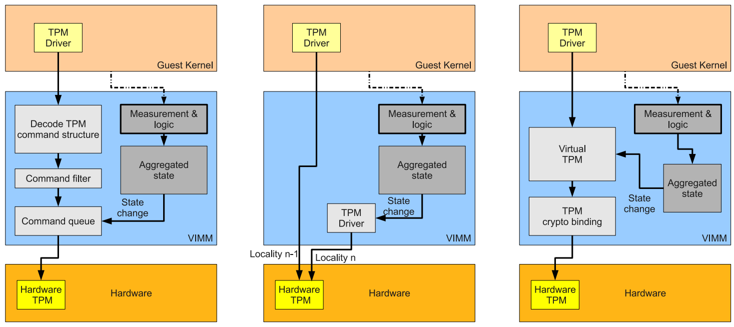
The use of a pattern database together with tags is integral to the VIMM framework, since runtime integrity measurement is more complicated than verifying static data. Furthermore, the use of tags add a level of indirection, which provides additional flexibility. The use of tags would allow the author or distributor of the code to generate and sign a pattern database, from which the tags are derived. This also protects the privacy of the attestant by not revealing the actual running state of the system, but rather only attesting that the system is in a verified state. This use of a pattern database thus brings additional benefits for managing software updates.

4.4 Measurement reporting

Measurement reporting is done via "extend" operation into the TPM. Unlike para-virtualization, the VIMM framework works as a thin layer between the guest OS and hardware with no direct interface (hypercall) for the guest kernel. Thus, the only way to transmit information is via hardware interfaces. To provide compatibility for existing software infrastructure, TPM is a suitable hardware interface for the purpose of measurement reporting. Three models of multiplexing TPM between VIMM and the guest OS are proposed here.

TPM filtering model

The first model (shown in Figure 2a) is to filter access to a real TPM, by filtering and blocking specific commands related to the measurement. The goal of filtering, is to allow the VIMM hypervisor to have exclusive control of a set of



(a) TPM stream filtering (b) TPM sharing (c) Virtual TPM

Figure 2: TPM measurement reporting

PCR registers. This filtering mechanism is achieved by intercepting the Memory Mapped IO (MMIO) pages of the TPM device and parsing the TPM command stream. The "extend" operation for a specified set of PCR registers is blocked. Furthermore, VIMM would call the "extend" command of TPM, to report a change in the aggregated state of the system. This is possible due to the stateless nature of the "extend" operation (unlike other TPM operations). Since the TPM command stream has a simple header format, most commands can be buffered without parsing the specific content of the command. Commands from the guest are first buffered before being sent to the real TPM, so as to prevent denial-of-service via an incomplete command.

TPM sharing model

TPM sharing model (shown in Figure 2b) uses the locality feature defined in version 1.2 of the TPM Interface Specification (TIS) [TCG05a], to separate the locality of the guest from the locality of the VIMM hypervisor. Since each locality is defined in a separate memory page location, a higher locality of TPM can be used by VIMM, and the guest kernel is restricted to a lower one. The TIS specification also provides the means for a higher locality to abort lower locality access and seize immediate control of TPM. This can be used by VIMM to immediately report integrity measurement changes. This approach greatly simplifies the sharing of TPM between VIMM and the guest kernel. The disadvantage is that it is not completely transparent to the guest TPM driver. The guest TPM driver has to be configured to use a lower locality and be able to detect and resume normal operation after VIMM temporarily seizes control of TPM.

In the VIMM configuration as shown in section 8.8, locality 2 is chosen for VIMM while locality 1 is allocated to the guest OS, according to the recommendations from [TCG05b]. Under this configuration, the guest OS is restricted to extending into PCR 20 while VIMM gains exclusive use of PCR 19.

Virtual TPM model

A third approach (shown in Figure 2c) would be to completely virtualize TPM by having a built in virtual TPM. In this case, the hardware and virtual TPMs are completely decoupled. Cryptographic binding between the two can be achieved in a variety of methods, as mentioned in [SvDP06]. This approach has higher complexity compared to the previous approaches, as the entire TPM must be emulated. However, it has the advantage of having a faster TPM, as the cryptographic operations are performed on the CPU rather than on the slower hardware TPM chip. This is particularly useful in server and high-security scenarios where the security application is closely bound to TPM, such that unsealing operations is called for every critical operation performed. Such applications also take full advantage of the runtime nature of the integrity measurement. This is in contrast to applications which "cache" unsealed secrets, which no longer provide any trustworthiness guarantee once the secret has been unsealed.

5 Binary code measurement

In order to measure binary code in a generic way (works with different operating system), binary code is measured upon its first execution after being loaded into memory. Thus, the VIMM framework needs to verify the in-memory image of the binary code which can differ from the binary file which is used to load the image. This can be understood by looking into the executable binary file formats in section 5.1.

5.1 Executable binary formats

Executable and Linkable Format (ELF) and Portable Executable (PE) are 2 commonly used binary file formats. ELF is widely used on modern unix-like systems including Linux and bsd, and is designed to be completely position-independant (binary code can be loaded easily at any base memory location). The need for special handling to allow position independance, 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-dependant jumps by marking the positions of such jumps within the compiled code in a relocation table.

PE is the default binary format used on every modern version of microsoft windows, and is functionally similar to ELF. One major difference is the PE format assumes a fixed based position for each piece of binary code. There is no relocation table on PE format; thus, 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.

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.

Thus, when the binary code is loaded in a system (either executable or shared library), the binary code necessary for binary execution is loaded into memory, discarding unnecessary information (such as headers, tables, embedded debug information, metadata, etc). After that, the jump addresses are fixed up for the reallocated memory location and also to externally linked libraries.

Figure 3a illustrates how an ELF binary is loaded into memory. The binary loader, usually in the OS kernel, reads the ELF file and its respective headers, and extracts the position of each execution-related segment. Each segment is tagged with flags, but the main segment types are executable (read-only), read-only data, read-write data and un-initialized data. The binary loader places the segments in a newly allocated memory block, and performs relocation and linking of libraries.

5.2 In-memory code measurement

After understanding the binary format and its loading process from section 5.1, it can be seen that there are 3 main differences between the binary image on file and its corresponding image in memory:

1. only execution-related sections are eventually loaded into the in-memory binary image (file and section headers, tables, debug information, metadata and the like are discarded)

2. memory locations within the binary are corrected based on relocation

3. memory addresses of external library functions are written into the image (linking)

Therefore, the process of measuring in-memory code involves constructing the in-memory image from the binary file format, and then comparing the constructed image with the in-memory image. Since the relocation address and external library addresses cannot be known beforehand, a simple method, adopted from [JXR08], is to ignore those bytes by inserting a string of zeros at each of these memory locations. A hash of the constructed image, with jump locations zeroed, is computed and used in the verification process.

5.2.1 Pattern database

Since VIMM does not have prior information on where the binary code is loaded in linear memory space, finding the exact match is a time-consuming process. As a consequence, a few steps were taken to optimize the process. Firstly, a few observations on binary code loading on most modern OS can be made:

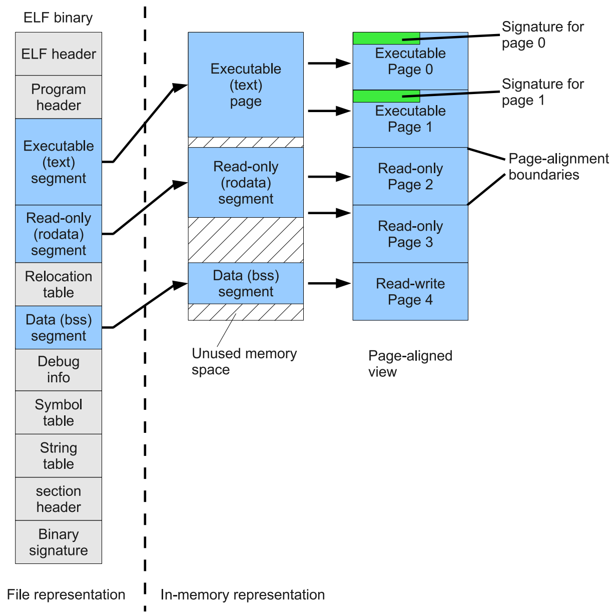
- binary code is loaded page-aligned
- a unit of binary code is continuous in linear memory space

Based on these 2 assumptions, the constructed image can be divided into page-aligned blocks, and a signature for each block is generated by extracting a fixed length of leading bytes of each block (see figure 3a). Each independantly verifiable block is termed as a 'unit'. In the case of the Linux OS, kernel modules are divided into an initialization part and normal (known as 'core') part. The initialization part is only executed once during the kernel module loading time and is subsequently unloaded to save memory. The initialization and core parts of the kernel driver are, thus, independantly handled as 2 separate units in the pattern database.

These signatures are compiled into a database which allows VIMM to efficiently identify the binary code which is loaded and its memory range linear memory. Figure 3b shows an example of the entries for finding the corresponding unit based on the page signature, while figure 3c shows the entries in the database used for verifying the entire unit.

When the guest OS jumps into some unverified code, the leading bytes of the page of that memory location is used to efficiently find the corresponding unit and page index. Based on the page index and corresponding unit size, the start and end of the unit can be computed. A copy of the in-memory image is made and the jump locations are zeroed out. After that, a hash of this image is made and compared with the pre-computed unit hash.

Some practical problems of this measuring method arises from execution-time code patching done in Linux kernel and kernel modules to replace generic code with alternative code blocks. This is done to patch in optimized code for a particular processor type, instructions for secondary processors in a symmetric multiple processing (SMP) system, as well as alternative code for para-virtualized operations. The current implementation requires a simple patch to Linux to disable alternative code patching, which does not change the functionality of the kernel in non-SMP system, albeit losing some performance.



(a) File to in-memory of ELF binary

Page signature	Unit	Unit page index
<Sig 1>	usb_init	1
<Sig 2>	usb_core	1
<Sig 3>	usb_core	2

(b) Signature database

Unit	Unit size	Ignore locations	Hash
usb_init	900	10,25, ...	<hash 1>
usb_core	6000	4,42,56, ...	<hash 2>

(c) Unit database

Figure 3: Binary measurement

6 Guest OS measurement

Being able to ensure kernel level (ring 0) code integrity is important for a secure system but not sufficient. Likewise, the measurements from the VIMM framework can be extended to include fine-grained measurements, that can only be performed from within the guest kernel. Two possible extensions are to perform code signing and verification during load time, as well as behaviour verification of running processes. The Linux Security Module (LSM) framework enables additional security modules in the Linux kernel to insert hooks into the kernel to perform additional fine-grained monitoring of resources and processes. As mentioned earlier, SELinux and AppArmor are 2 such commonly used security modules employing the LSM framework.

However, the existing LSM modules were not designed with integrity measurement in mind. As a result, simple modifications can be made to combine the measurements provided by VIMM and additional measurements from the LSM modules, by extending each of these measurement values into two different PCRs on TPM. This separation can be made based on the TPM separation methods discussed in section 4.4. In this way, the management of the LSM-based measurement is independant of VIMM, but the validity of the measurement is ensured based on the fact that the guest kernel is integral.

In this paper, two LSM based modifications are presented to perform code signing and verification, and process monitoring are presented in sections 8.6 and 8.7, respectively.

7 Verification mechanism and applications

The VIMM framework is designed to provide integrity measurement in a non-mandatory manner. This means that it will not prevent malicious code from executing, but they should be detectable through inspection of the PCR values. Confirmation of the system (guest kernel and processes) in an integral state can be easily proven by verifying the PCR values of TPM. Two main mechanisms used by the TPM to support verification of the PCR are the "seal" and remote attestation operation.

Local data sealing

To verify the PCRs locally, the TPM "seal" operation can be used to bind a piece of secret to a set of PCR values that represent a known state. The guarantee provided by sealing is that the integrity measure at the point of sealing is identical to the time of unsealing. As a result, the secret is accessible only when the system is in a known state.

Remote attestation

In order to proof the PCR values to a remote party, an identity key has to be generated by TPM and signed by a trusted privacy CA. In the course of remote attestation, the "quote" operation on a TPM generates a signature of a selected set of the current PCR values (together with a nonce and other information) using the identity key. This signature is a cryptographic proof of the PCR values (which implies the integrity of the system) to the remote party.

7.1 Application scenarios

Due to the fact that presenting a detailed application is not within the scope of this paper, some possible scenarios are discussed here, instead, to illustrate the useful applications of the VIMM framework and integrity measurement.

7.1.1 Personal data protection

Encryption can be employed to protect personal data, with the encryption key sealed to the PCRs of TPM. This ensures that the integrity state at the time of unsealing is identical to that at the time of sealing. To this end, if the encryption key was sealed when the runtime integrity measurement was in a particular tag, unsealing of the key at a later time is only possible if the system is in the same integrity tag, thus ensuring that the data is only accessible when the system is integral.

7.1.2 Integrity verified connection to critical resources

In the scenario of a client computer connecting to a remote host, remote attestation can provide a means to verify the PCR values, and hence integrity, of one or both parties. Integrity measurement may find useful applications in:

1. Online banking, where integrity measurement ensures that the client machine is integral when connecting to the bank website

2. Virtual Private Network(VPN) server of a company, where integrity measurement is used to ensure that the the client machines used by employees connecting to the corporate network from an external location is in a integral state before gaining access to the corporate network

3. Network authenticator of an internal network, where integrity measurement is used to monitor the runtime state of its clients, as a means for a network-wide Intrusion Detection System (IDS)

In order to integrate remote attestation into existing applications, modifications to existing network protocols have to be made. The Trusted Network Connect (TNC) specification [TCG08] from TCG defines an architecture for implementing the necessary components, and the associated modification to the 802.1X protocol [iee04] or the Transport Layer Security (TLS) [RD06], so as to carry information related to remote attestation. Alternatively, simpler extensions to the TLS [LUNB06] also exist as proposed from the IETF working group. The TLS-based extensions would be useful for cases 1 and 2 while the 802.1X extension, useful for cases 2 and 3.

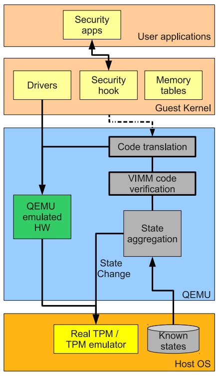


Figure 4: VIMM on QEMU virtual machine

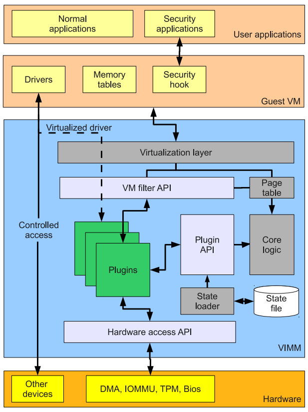
8 Implementation

The VIMM framework is designed to perform runtime measurement of the guest OS. The key resources which can be measured from outside the guest OS without explicit OS-specific code, are memory, hardware states and events (such as IO access, interrupts and traps). In ensuring integrity, the measurement of code in memory is achieved through a separation between verified and non-verified code. Two prototypes were developed using the VIMM concept; the first based on QEMU [qem09] which is a binary-translation type virtual machine running on a host OS, and the second prototype is based on hardware supported virtualization (known as HVM) and runs directly on the processor.

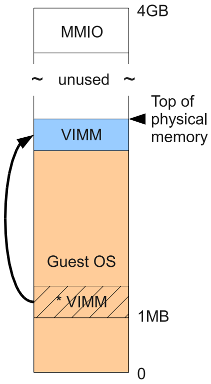
8.1 QEMU-based virtualization

QEMU is a binary-translation virtual machine, running on a host OS. The QEMU-based VIMM prototype is representative of other binary-translation type virtual machines such as virtual box, VMware player and workstation, and so on. It is developed based on code from nickle [JXR08], but modified to measure and report code execution via TPM, rather than preventing unverified code execution. Even though QEMU is not a bare-metal hypervisor (running as the lowest level software), it is functionally similar to the HVM-based hypervisor in section 8.2.

In developing this implementation, the QEMU emulator has been patched to emulate a TPM device which can connect to either a TPM emulator on the host using a unix pipe or a real TPM. Like in nickle, a separate mirrored memory area is used to store only verified code. A function is inserted into the binary translation cycle of QEMU, to check new code by comparing the current memory area and a mirrored memory area containing verified code. Mirroring is



(a) Detailed structure of VIMM using HVM



(b) Memory allocation of VIMM using HVM

Figure 5: Implementation of "self-shadow" mapping in HVM based VIMM

performed in the physical memory space, thus avoiding problems of changes in the page table mapping.

The verification mechanism is implemented as described in section 5.2. Upon successful verification, the entire module is copied into the verified mirrored memory area, while a failed verification will result in a state change, eventually extending an unknown state tag into the TPM PCR.

8.2 HVM-based hypervisor

Hardware Virtual Machine (HVM) was introduced by AMD and Intel in recent processors, as an additional instruction set to directly support virtualization on the processor. This allows a hypervisor to intercept critical operations in the guest OS, and virtualize the guest by controlling the guest page table among other operations. This second prototype uses the HVM support from AMD to implement the VIMM framework as a bare-metal hypervisor.

Figure 5a shows the detailed structure of the prototype HVM-based VIMM hypervisor. The design of the HVM-based VIMM is based on the concept proposed in Secvisor [LQPS07], but is extended to consider the kernel memory together with hardware states as part of a generalized state. VIMM also focuses on integrity measurement and reporting rather than prevention of attacks, with specific application for trusted computing and property-based attestation.

Using the Secure Virtual Machine (SVM) instructions defined by AMD, VIMM sets itself as the virtualization host before starting the guest kernel OS using the VMENTER instruction. After control is being passed to the guest OS, it is automatically returned to VIMM upon pre-defined conditions, such as changes to CR0,3,4, page fault and so on. In this way, VIMM has control over the important hardware state of the guest OS and also any changes to the guest page table. Traditionally, shadow paging is used for the purpose of controlling a guest page table, such as those found in the xen-hypervisor [xen09]. However, a novel approach, known as self-shadowing, is presented in section 8.4, as a special case optimization when only a single guest OS is virtualized.

Identification of verified or non-verified code is handled by using the No Execute (NX) bit (see AMD specifications in [AMD07]) of the page table attribute in 64-bit mode. The NX bit allows a page to be marked as not executable, thus causing any instruction execution within such a page to raise a page fault exception. Since VIMM can directly control the page table of the guest, it marks all new pages with NX. Once a page fault occurs because the guest tries to execute an instruction in an NX flagged page, the verification process is performed as described in section 5.2. Upon successful verification, the NX flag is removed from the associated pages, while a failed verification will cause a state change, eventually extending an unknown state tag into the TPM PCR.

Due to the complexity of the SVM instruction and paging complexities, this prototype is currently only fully working for a simplified guest kernel and not a fully fledged Linux kernel.

8.3 Memory paging on HVM hypervisor

In the first generation HVM processors from AMD and Intel, there was no support for Nested Page Table (NPT) [AMD07] or Extended Page Table (EPT) [Int08]. NPT or EPT enables the hypervisor to have a separate guest-physical to system-physical memory paging. In the absence of NPT or EPT, a software solution using the HVM is still possible, by intercepting all paging related instructions in the guest OS, to maintain a Shadow Page Table (SPT). The SPT

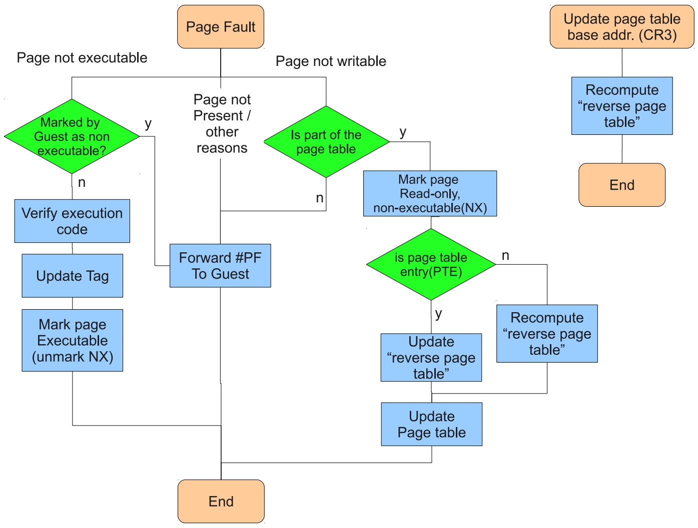


Figure 6: Self-shadow flowchart

is the actual page table provided to the processor, but is synchronized by the hypervisor to changes made by the guest kernel's local copy. In this way, the hypervisor can have complete control of the guest page table by controlling the synchronization between the two tables, adding any necessary differences.

Using NPT, EPT or SPT alone is not sufficient as it only handles memory access performed by the CPU, but does not handle memory access from hardware devices. Therefore, when using a Direct Memory Access (DMA) controller on older ISA bus, or bus mastering on PCI bus, the memory access would point to the wrong location. In order to solve this problem, an IOMMU is necessary to translate memory access and provide appropriate isolation to protected memory pages. The disadvantage of relying on the IOMMU is that it is not available on all processors supporting HVM.

An alternative to using an IOMMU is to have no offset between the guest physical address and the system physical address. This direct mapping would allow DMA and bus mastering to work without intervention from VIMM. On the AMD processor, Device Exclusion Vector (DEV) can be used to provide isolation of the hypervisor and other protected memory pages. DEV defines a range of memory which is not accessible from the hardware. In this way, by placing the hypervisor in the DEV range, it is protected from hardware-based attacks.

8.4 "Self-shadow" paging

The "Self-shadow" paging mechanism makes clever use of the direct mapping of the guest OS, to further optimize performance and memory usage. Figure 5b shows how the VIMM hypervisor is loaded initially from the 1MB location by the boot-loader, but is relocated to the top of the physical memory area during runtime, before loading the guest OS into its final position. Self-shadow paging uses the guest page table as the actual page table, while controlling write access from the guest to the page table. The algorithm requires a reverse page table that maps physical memory addresses to its corresponding page table entries. This table enables VIMM to determine if a memory write operation is actually updating a part of the page table.

The flowchart for the algorithm is illustrated in figure 6. When the page table is initialized or updated by the guest through changing the CR3 register, the reverse page table is computed by traversing through the page table to find the reverse mapping. When a page fault occurs in the guest, due to a memory write to a non-writable page, the reverse page table is checked to determine if the write location is part of the page table and corresponding level within the page table. If the location is not part of the page table, the fault is forwarded to the guest kernel for handling. Finally, depending on the position of the updated page table entry, the reverse page table and the guest page table are updated, and marked read-only and non-executable (NX flag).

When a page fault occurs due to execution on a non-executable page, if this was also marked as non-executable by the guest kernel, the fault is forwarded to the guest kernel. Otherwise, the code verification process described in section 5.2 is performed and the security tag updated if necessary. The page is marked as executable again and the control is returned to the guest. All other cases of page faults are directly forwarded to the guest kernel.

The self-shadow technique improves performance, by allowing the processor to update flags (dirty bit, accessed bit) without exiting into the hypervisor, and also eliminates the need to maintain and synchronize a shadow page table.

8.5 Device virtualization

The TPM sharing model is successfully tested in the prototype implementation of VIMM, using the "seize" function of the TIS specification to temporarily take control of the TPM from the guest. Since the prototype implementation of VIMM uses a direct mapping memory model for the guest OS, any memory area not blocked by VIMM via page table or DEV can be directly accessed by the guest OS.

8.6 Patched digsig

Digsig is an LSM module which reads and verifies a digital signature embedded into ELF binaries, so as to verify the binary before loading or executing it. It requires the support of two userland applications. One is used to insert the signing public key into the module via a sysfs file, which is used to verify the signatures. No verification is performed before the public key is inserted. The other is the bsign [Sin03] application which is used to sign the ELF binaries and embed the signature. The digsig module was modified to integrate with the VIMM framework, by making the following changes:

1. Digsig extends a hash of the signing 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.

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 must be done during the initial ramdisk phase of the boot-up process, so as to maintain the chain-of-trust. This is because the initial ramdisk is already measured by either VIMM or the trusted bootloader (see section 8.8).

With this approach, the VIMM framework can be extended to monitor the runtime integrity of the guest kernel, and also the integrity of all loaded binaries within the guest OS. Therefore an attestation of the PCR with the correct public key hash would prove that a known key is used to verify the binaries and no invalid binary has been loaded.

The decision of not using the VIMM framework to measure userland applications was consciously taken, in order to reduce the complexity of the signature database which must be loaded into VIMM, and to allow LSM modules to make use of additional process information during verification, which is not available to VIMM.

8.7 Patched AppArmor

AppArmor is another LSM module which enforces process behavior based on a policy language. Similar to digsig, AppArmor requires a set of userland applications to perform the loading, unloading and modification of policies in AppArmor. The policy defines resources (such as files or network connections) which the process is allowed to use or access. Each policy can be either in enforcement mode, where disallowed resources are denied, or in logging mode, where access to disallowed resources is logged.

The AppArmor module was modified to integrate with the VIMM framework, by hashing the policy file and extending the hash into the LSM-controlled PCR, every time a policy file 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 the 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 modified until the time of attestation, thus implying that the correct policies are being enforced.

8.8 Boot sequence and chain-of-trust

The boot sequence in VIMM varies slightly, in terms of the QEMU and HVM based prototypes. Figures 7a and 7b show the PCR usage of each prototype, respectively. PCRs 0-4,8 and 9 are measured by BIOS and the trusted bootloader (TrustedGRUB [SS09]) as part of the boot process as suggested in [TCG05b]. In the first case, since VIMM is a part of the emulator, it cannot be measured directly but rather, is implicitly trusted. A signature of the pattern database is extended into the VIMM controlled PCR (PCR 19) before runtime tags are extended. In the latter case, the VIMM hypervisor, pattern database, guest kernel and guest initial ramdisk are loaded by the TrustedGRUB, and are thus measured via TrustedGRUB. PCRs 19 and 20 are used by VIMM and the LSM module, respectively, to extend the runtime tag and the output of LSM.

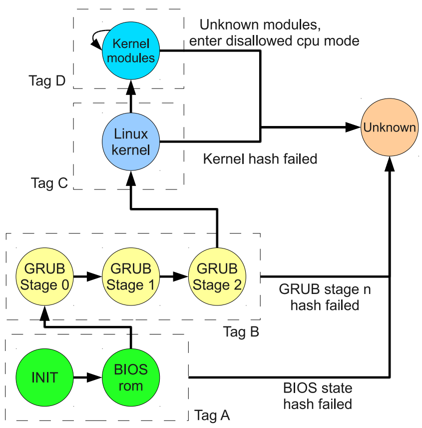
Figure 7c shows the state transition for VIMM on the QEMU based prototype, on a Linux kernel. Tags A to D show the normal transition from boot-up to the running state of the Linux kernel. Likewise, if the PCR is extended in the order of A to D without any transition to the unknown state, it implies that the guest OS is in a verified state.

PCR Index	Controlled by	Data measured
0-3	BIOS	BIOS specific
4,8,9	BIOS,TrustedGRUB	measurement of MBR,TrustedGRUB
12	TrustedGRUB	GRUB commandline
14	TrustedGRUB	Linux kernel, initial ramdisk
19	VIMM	pattern DB signature, VIMM runtime state tag
20	LSM	AppArmor policies / digsig public key

(a) PCR configuration for QEMU-based prototype

PCR Index	Controlled by	Data measured
0-3	BIOS	BIOS specific
4,8,9	BIOS,TrustedGRUB	measurement of MBR,TrustedGRUB
12	TrustedGRUB	GRUB commandline
14	TrustedGRUB	VIMM hypervisor, pattern DB signature, guest kernel
19	VIMM	VIMM runtime state tag
20	LSM	AppArmor policies / digsig public key

(b) PCR configuration for HVM-based prototype



(c) VIMM runtime state on a Linux kernel (extended into PCR 19)

Figure 7: Full configuration

8.9 Performance

A test of the performance of the QEMU-based prototype shows a minimal overhead of 3-4% incurred by VIMM, in CPU and file-access and module-loading benchmarks. The performance of module-loading has been improved over nickle, due to the caching and optimizations on the pattern database.

9 Conclusion

The proposed design of the VIMM framework aims to achieve runtime integrity measurement of a guest OS, while maintaining a small code and memory footprint. An aggregated state is computed from a variety of runtime measurement parameters, including the current memory pages of the guest kernel. This is reported securely to the guest kernel via the TPM by extending a known tag value into a VIMM controlled PCR. As a result, VIMM achieves a form of property-based attestation, based on the runtime integrity of the guest kernel, which allows applications to bind secrets to TPM, and perform remote attestation bound to the system's runtime integrity. The integrity measurement is further extended into the guest OS, by employing LSM modules to maintain integrity within the OS using fine-grained measurements only possible from within the OS.

Two implementations of the VIMM framework have been discussed. The first implementation is based on the QEMU emulator and performs code verification within the binary translation cycle of the emulator. The second implementation uses the SVM instructions from the AMD processors to run directly without a host OS. It uses a self-shadow paging mechanism to optimize performance for the special case of a single virtual machine. Based on tests on the QEMU prototype, the performance overhead is minimal at 3-4%.

Possible future work would be to investigate attack vectors specifically for the VIMM framework, and to test the performance of the HVM-based prototype on the same Linux kernel. An integrated LSM module combining AppArmor with digsig, would make an ideal LSM for a complete integrity measure.

References

[AMD07] AMD: *AMD64 Architecture Programmer Manual*, vol. 2 system programming edition, Sept 2007.

[app08] Apparmor;, Nov 2008 http://www.novell.com/linux/security/apparmor/.

[arg06] Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation; European Conference on Computer Systems:15–27, 2006.

[bit09] *BitVisor: A thin hypervisor for enforcing I/O device security*, volume ACM/Usenix International Conference On Virtual Execution Environments of *Visors*. ACM, 2009.

[CLL+06] Chen, L., Landfermann, R., Löhr, H., Rohe, M., Sadeghi, A.-R., and Stüble, C.: 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.

[CSP08] Carbone, B., Sharif, M., and Payne, M. W. L.: Lares: An architecture for secure active monitoring using virtualization; *IEEE Symposium on Security and Privacy*, pages 233–247, May 2008.

[iee04] 802.1x port-based network access control; Technical report, IEEE Computer Society, Dec 2004 http://standards.ieee.org/getieee802/download/802.1X-2004.pdf.

[Int08] Intel: *Intel Virtualization Technology for Directed I/O*, Sept 2008.

[JXR08] Jiang, X., Xu, D., and Riley, R.: Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing; In *Recent Advances in Intrusion Detection*, pages 1–20. Springer, Sept 2008.

[LQPS07] Luk, M., Qu, N., Perrig, A., and Seshadri, A.: Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses; In *21st ACM Symposium on Operating Systems Principles*, October 2007.

[LUNB06] Latze, C., Ultes-Nitsche, U., and Baumgartner, F.: Transport layer security (tls) extensions for the trusted platform module (tpm); Technical report, IETF Network Working Group, Apr 2006 http://tools.ietf.org/html/draft-latze-tls-tpm-extns-01.

[LWPM07] Loscocco, P. A., Wilson, P. W., Pendergrass, J. A., and McDonell, C. D.: 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.

[qem09] Qemu open source processor emulator;, July 2009 http://www.qemu.org/.

[RD06] Rescorla, E. and Dierks, T.: The transport layer security (tls) protocol version 1.1; Technical report, IETF Network Working Group, Apr 2006 http://www.ietf.org/rfc/rfc4346.txt.

[sel08] Security-enhanced linux;, Nov 2008 http://www.nsa.gov/selinux.

[Sin03] Singer, M.: Bsign: Corruption & intrusion detection using embedded hashes;, Aug 2003 http://packages.ubuntu.com/karmic/bsign.

[SS09] Selhorst, M. and Stueble, C.: Trustedgrub;, Nov 2009 http://sourceforge.net/projects/trustedgrub/.

[SvDP06] Sailer, R., van Doorn, L., and Perez, R.: vtpm: Virtualizing the trusted platform module; In *15th USENIX security Symposium*, July 2006.

[SZJv04] Sailer, R., Zhang, X., Jaeger, T., and van Doorn, L.: Design and implementation of a tcg-based integrity measurement architecture; *13th USENIX Security Symposium*, pages 223–238, 2004.

[TCG05a] 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.

[TCG05b] Pc client work group specific implementation specification for conventional bios specification; Technical Report Version 1.2, Trusted Computing Group, July 2005 http://www.trustedcomputinggroup.org/developers/pc_client.

[TCG07a] Tpm main specification; Technical report, Trusted Computing Group, July 2007 http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[TCG07b] 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.

[TCG08] Trusted network connect (tnc); Technical report, Trusted Computing Group, Feb 2008 http://www.trustedcomputinggroup.org/developers/trusted_network_connect.

[xen09] Xen hypervisor;, Nov 2009 http://www.xen.org/.