Improving Policy Verification Capabilities of Trusted Platforms

Serdar Cabuk, David Plaquin, Theodore Hong, Derek Murray
HP Laboratories
HPL-2008-71

Keyword(s):
Trust computing, virtualization, policy verification, integrity management

Abstract:
Verifiable trust is a desirable property for computing platforms – a user has a fundamental interest in knowing whether a computing platform about to be used behaves as expected. Current systems provide verifiable trust by taking immutable snapshots of a platform which digest the complex set of platform components and dependencies into relatively few measurements. Moreover, any change to the platform is deemed malicious, and it is only possible to revert to the previous state by restarting the computer. In this paper, we introduce a novel extensible integrity management framework that addresses these problems. Our framework makes two key contributions: To improve integrity management, we explicitly represent the dependency relation between platform components, which provides the user with more information about the state of the platform. To enable change management, we distinguish reversible changes to measured components from the established irreversible changes. We have implemented a prototype of this framework, based on the Xen virtual machine monitor. In addition, we demonstrate the use of our framework for policy enforcement by implementing a credential management service on top of it.

External Posting Date: June 21, 2008 [Fulltext] Approved for External Publication

Internal Posting Date: June 21, 2008 [Fulltext]

© Copyright 2008 Hewlett-Packard Development Company, L.P.

Improving Policy Verification Capabilities of Trusted Platforms

Serdar Cabuk, David Plaquin
Hewlett-Packard Laboratories
Bristol, United Kingdom
{Firstname.Lastname}@hp.com

Theodore Hong, Derek Murray
University of Cambridge Computer Laboratory
Cambridge, United Kingdom
{Firstname.Lastname}@cl.cam.ac.uk

June 11, 2008

Abstract

Verifiable trust is a desirable property for computing platforms – a user has a fundamental interest in knowing whether a computing platform about to be used behaves as expected. Current systems provide verifiable trust by taking immutable snapshots of a platform which digest the complex set of platform components and dependencies into relatively few measurements. Moreover, any change to the platform is deemed malicious, and it is only possible to revert to the previous state by restarting the computer. In this paper, we introduce a novel extensible integrity management framework that addresses these problems. Our framework makes two key contributions: To improve integrity management, we explicitly represent the dependency relation between platform components, which provides the user with more information about the state of the platform. To enable change management, we distinguish reversible changes to measured components from the established irreversible changes. We have implemented a prototype of this framework, based on the Xen virtual machine monitor. In addition, we demonstrate the use of our framework for policy enforcement by implementing a credential management service on top of it.

1 Introduction

Trusted Computing has been proposed as a means of providing verifiable trust in a computing platform. However, as virtualization becomes more popular and platform changes (such as security patches) occur more frequently, the established model for Trusted Computing is insufficient to cope in real-world scenarios. We therefore introduce an extensible integrity management framework that is better suited to deal with complicated trust dependencies and change management.

The goal of Trusted Computing is to enable third parties to remotely attest and verify the configuration of a computing platform in a secure manner. Existing *trusted platforms* typically contain a component that is at least logically protected from subversion. The implicitly trusted components of a trusted platform – in particular, the hardware Trusted Platform Module (TPM) – can be used to store integrity measurements, and subsequently report these to users (or remote entities) with a cryptographic guarantee of their veracity. Users can then compare the reported measurements with known or expected values, and thereby infer whether the platform is operating as expected (e.g., it is running the expected software with the expected configuration while enforcing the expected policies).

Present implementations of Trusted Computing technology can take immutable snapshots of a whole platform, which can then be used as proof of trustworthiness [23, 11, 14, 10]. They do not, however, provide more granular verifications of platform components such as individual virtual machines (VMs) and applications. The platform is treated as a whole, and while it is possible to store integrity measurements of VMs and applications, the limited amount of storage in a TPM means that it is not possible to represent individual components and the dependencies between them. Furthermore, it is not possible to manage

changes to measured components. The current scheme advocated by the Trusted Computing Group (TCG) deems all such changes to be malicious [26]. This is certainly impractical for modern server environments, which undergo a constant bombardment of security patches and policy changes. In 2007 alone, Microsoft released 11 security related patches for the Windows operating system [1], while a typical enterprise anti-virus application will undergo two to five updates in an average week [18].

In this paper, we introduce an extensible integrity management framework that addresses these two shortcomings. To improve integrity management, we explicitly represent integrity dependencies between platform components by giving individual registers to each component to store their integrity measurements, and chaining these components together in a dependency graph. To improve change management, we introduce a new distinction between reversible and irreversible changes to measured components. A reversible change is one that can be undone and is guaranteed not to have any permanent effects. The introduction of reversible changes allows the platform integrity to be modified temporarily, for example when a device is hot-plugged and then removed. Although the platform may no longer be considered trustworthy during the time that the change holds, its integrity can be safely restored after the change is undone.

Our resulting framework gives a better understanding of a platform's security properties, which can be used in policy verification. Like existing Trusted Computing implementations, our services can be used to grant access to protected resources (such as encrypted storage) only when the policy is satisfied; however, unlike existing implementations, these policies can be more fine-grained, dynamic, and flexible. Our prototype implementation, built on the Xen virtual machine monitor [9], includes the integrity management framework and a credential manager service, which demonstrates the use of enhanced policy checks to control access to security credentials.

The remainder of this paper is organized as follows. Section 2 provides background on Trusted Computing and virtualization. Section 3 outlines the motivation and high-level design for our integrity management framework. Section 4 presents the basic framework, which provides integrity services to individual components; Section 5 extends this into reversible integrity changes and an explicit dependency graph, and provides use cases for this model. Section 6 presents some examples of security services that could make use of our framework. Section 7 describes our prototype implementation of the framework and the credential management service on Xen. Finally, in Section 8 we discuss related work, and in Section 9 we draw conclusions.

2 Background

Virtualization and Trusted Computing have gained prominence in the past ten years as commercial interests have led to consolidating multiple virtual machines on a single physical host. Virtualization enables simple consolidation and isolation while Trusted Computing promises increased security guarantees. In this section, we introduce Trusted Computing technology in Section 2.1 and virtualization in Section 2.2.

2.1 Trusted Computing

Trusted Computing technology aims to provide a cryptographic guarantee of the integrity of a computing platform. Arbaugh *et al.* developed AEGIS [3], the architecture on which most subsequent Trusted Computing systems are based. AEGIS is responsible for introducing two fundamental concepts: the use of cryptographic hashes (integrity measurements) of platform code to demonstrate integrity, and the *chain of trust*.

A piece of code has integrity if it has not been changed in an unauthorized manner during a defined period of time. Any change, however small, to the code would result in a complete change in the hash value: the hash is therefore a concise means of representing the code. The integrity of an entire platform can be

captured by starting the boot process with a *core root of trust for measurement (CRTM)*, which might be a BIOS boot block, for example. The CRTM loads the next component in the boot process, measures (hashes) it, and stores that measurement in a secure location. That component then carries out whatever processing is necessary before loading and measuring the next component, and chaining the measurement to the secure log. This process repeats until all trusted components are loaded. The integrity of the whole platform can then be proved by induction over the log of integrity measurements.

AEGIS inspired the most common Trusted Computing architecture, which is defined by the Trusted Computing Group [26]. In this architecture, every computer contains a secure co-processor, known as a Trusted Platform Module (TPM), which enables the enforcement of security policies by controlling access to cryptographic material and primitives. It also provides secure storage in the form of Platform Configuration Registers (PCRs), which may only be reset or *extended*. Extension is used to represent an entire chain of trust in a single register, and we discuss this further in Section 4. A secure bootloader, such as OSLO [15], is required to ensure that the initial state of the TPM reflects the first component that is loaded. Thereafter, all subsequent platform components, including the operating system kernel and device drivers, can be securely loaded by the preceding component.

A further consideration is the Trusted Computing Base (TCB). This term is used inconsistently in the literature, and we prefer the definition from Hohmuth *et al*, who refer to "the set of components on which a subsystem S depends as the *TCB of S*." [13] Therefore a single platform could contain multiple TCBs, depending on the set of applications that runs on it. In this work, we refer to the *platform TCB* as the set of components on which all other platform components depend, and the *application TCB* as the set of components on which a particular application depends. This distinction can be illustrated by considering the following scenario. A web browser depends on an HTML renderer for correct execution: therefore the renderer is in the application TCB of the browser. However (assuming a sensible implementation), the renderer could not compromise the entire platform: therefore it is not in the platform TCB.

2.2 Machine Virtualization

Virtualization makes it possible to partition the resources of a computer platform – such as memory, CPU, storage, and network connections – among several *virtual machines (VMs)*, which provide an interface that resembles physical hardware. A *virtual machine monitor (VMM)* runs beneath the VMs and is responsible for securely (and fairly) multiplexing access to the physical resources. In addition, to preserve isolation between the VMs, the VMM executes privileged instructions on behalf of the guest VMs. In our work, we consider an architecture whereby the VMM is the only code that runs at the highest privilege level; alternative approaches place the VMM inside a host operating system kernel [20, 25]. In particular, we consider the Xen VMM [9].

VMMs are increasingly used in the development of secure computing systems [7, 24, 8]. The typical argument for using a VMM is that the amount of code is relatively small by comparison to a full operating system: the Xen VMM comprises approximately $100,000$ lines of code, while a recent version of the Linux kernel comprises approximately over 6 million lines of code. The compactness of a VMM therefore makes it more trustworthy than a monolithic kernel. It can therefore be argued that it is feasible to include a VMM inside a minimal TCB. Note that security flaws *within* a VM are not solved by a standard VMM (although specialized VMMs, such as SecVisor, do address this problem [24]). However, the isolation properties of a VMM ensure that the compromise of one VM cannot affect another VM. Therefore, virtualization can be used to host applications from mutually distrusting organizations on the same physical machine, or to provide a sand-box for executing untrusted code.

Trusted virtualization extends the concepts from Trusted Computing, such as chains of trust, into virtual machines. These can be used to attest the state of a VM to a third party [11], or to provide the illusion of a physical TPM to applications running within a VM [4].

3 Design Overview

The typical design for a trusted platform comprises a hardware TPM and software integrity management services. These services measure platform components, store integrity measurements as immutable logs and attest these measurements to third parties. The services use the TPM to provide a link with the CRTM. In a non-virtualized platform, with relatively few components to be measured, this model is sufficient. However, it does not scale to complex virtualized platforms that have a plethora of components and dependencies between these components. In this section, we first discuss the limitations of the existing model. We then present the high-level design goals that motivate our integrity management framework.

3.1 Hardware Limitations

Current integrity management systems typically employ the TPM as the sole repository for integrity measurements (see Section 8). Unfortunately, such schemes are fundamentally limited by the hardware capabilities of a TPM:

1. A TPM contains a small, limited amount of memory (PCRs). The TCG specification recommends that a TPM has at least 16 PCRs [26]. Therefore, for portability, we cannot assume that a TPM will have any more than 16 PCRs. Hence, it is not feasible to store individual measurements for a large number of virtualized platform components.

2. The limited number of PCRs is typically addressed by aggregating measurements in the same register. Where two components are independent, this introduces a false dependency between them. Furthermore, the definition of the `extend` function introduces an artificial dependency on the order in which they are aggregated.

3. It is not possible to reverse the inclusion of a measurement in a TPM register. Therefore, it is impossible for a platform component to report a change to its integrity (e.g. by the dynamic loading of some code, or the connection of a new device) and revert back (after unloading/disconnection).

To illustrate these limitations, consider the following example. A server platform hosts tens of small VMs, each of which runs a particular service. To keep track of the platform integrity on a traditional TPM-based system, the measurements must be aggregated, because there are more VMs than PCRs. For example, it might be necessary to store measurements for a virtual network switch and a virtual storage manager in the same PCR, which creates a false integrity dependency between these two VMs. If a malicious change is made to the virtual network switch, and this change is reported to the appropriate PCR, the integrity of the storage manager also appears to be compromised. The same is true for all other VMs whose measurements are aggregated in that PCR.

It would be possible to extend the set of PCRs by giving a virtual TPM to each platform component [4]. However, by allocating independent virtual PCRs to each component, it is no longer possible to represent real dependencies between components[1]. Furthermore, since the virtual TPMs emulate the behavior of a hardware TPM, it remains impossible to revert changes.

3.2 High-level Design

It is clear that software measurement support is required to address the limitations of hardware capabilities. We refer to the set of software components that comprise the integrity framework as the *software root of*

[1]Some virtual TPM designs share a fixed number of PCRs between all virtual TPMs and the hardware TPM, and these could be used to express dependencies. However, the reliance on the hardware TPM leads to the same limitations as a single-TPM scheme.

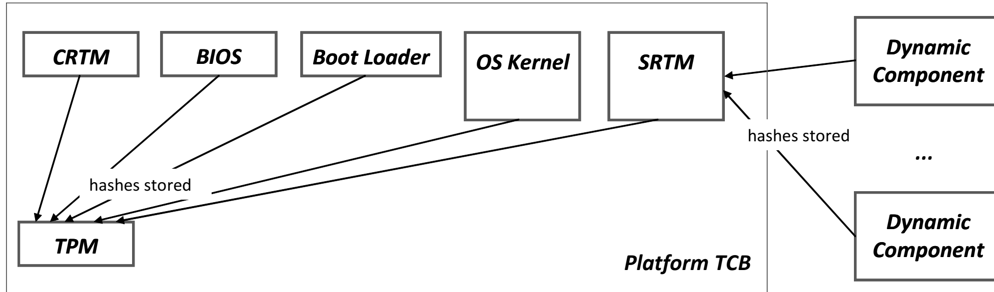


Figure 1: The position of the SRTM within the overall integrity management framework.

trust for measurement (SRTM). These components are part of the platform TCB, and should be isolated from other components; for example, by virtualization. Dynamic components outside the platform TCB rely on the SRTM to store measurements on their behalf, rather than the underlying TPM. Figure 1 illustrates the position of the SRTM within the overall integrity management framework.

Our framework has the following design objectives:

Unlimited measurement storage The framework should allow the storage of individual integrity measurements for an arbitrary number of components.

Explicit dependency representation The framework should allow the explicit and unambiguous representation of an arbitrary number of dependencies between platform components. There should be no false or artificial dependencies introduced by aggregation.

Static integrity management The framework should provide a superset of the functionality of a traditional TPM, with respect to static integrity.

Dynamic integrity management The framework should enable the integrity state of a platform component to revert to a previous trusted state in a controlled and verifiable manner.

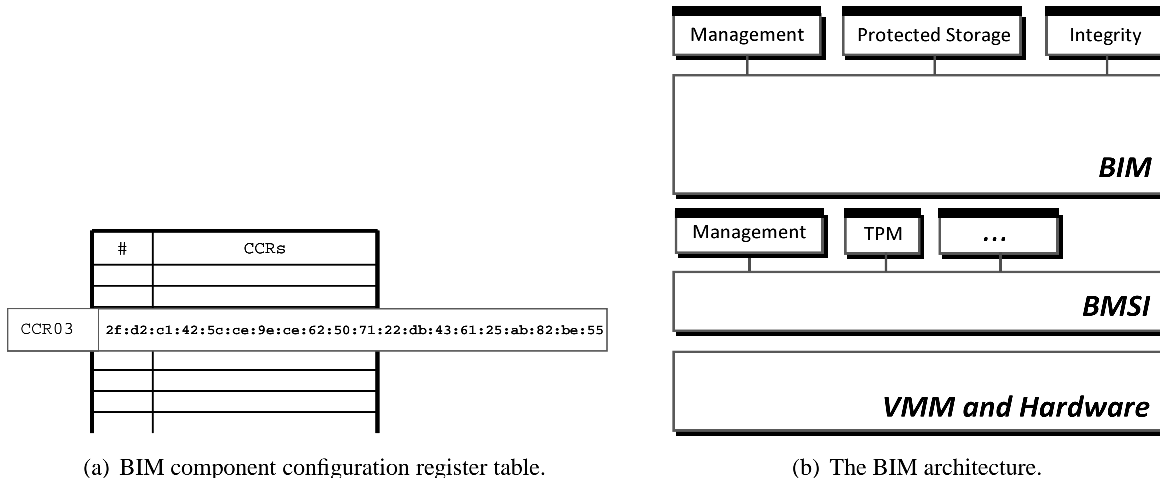
Link to hardware TPM The software framework should be linked in a chain of trust to the hardware TPM. This can be achieved by storing the measurements for the SRTM and other static components in the platform TCB (such as the hypervisor and any physical device drivers) in the TPM. As this set of components is small and non-changing, the limitations of a hardware TPM do not come into effect.

Minimal TCB In order to improve the trustworthiness of the framework, the SRTM and other components in the TCB should have a minimal amount of code and size of interface. This paper does not focus on minimizing the TCB, but a possible approach would involve using disaggregation [19].

Platform independence The framework should not be limited to a single hypervisor technology. Although the implementation (see Section 7) was carried out using Xen, it should be possible to use alternative technologies, such as VMware [25] or an L4 microkernel [17].

4 Basic Integrity Management

In this section, we present a basic design for the SRTM service that we introduced earlier. This platform-independent service provides the minimal functionality needed to manage the integrity of dynamic (non-TCB) platform components, which will be extended further in Section 5. Section 4.1 sets out the basic measurement model, while Section 4.2 describes the corresponding service architecture and interfaces.



(a) BIM component configuration register table.

(b) The BIM architecture.

Figure 2: Basic integrity management components.

4.1 Measurement Model

The Basic Integrity Management (BIM) service stores static integrity measurements of dynamic components that are arranged in a flat hierarchy, such as the one shown in Figure 3. Each component has a single Component Configuration Register (CCR) associated with it. A CCR is analogous to a PCR and holds integrity measurements for that component. The measurements are held together in a global CCR table similar to the one depicted in Figure 2(a).

4.1.1 Static Measurements

The BIM measurement model mimics TPM measurement capabilities but stores integrity measurements in software rather than hardware. Each registered dynamic component is assigned a BIM CCR to which its measurements are reported. This is achieved by an `extend` operation, which stores a new measurement in a CCR by hashing it together with the current value of the CCR. Dynamic components use this operation to report ongoing measurements when their contents change. For example, a firewall service would extend its CCR if its rule-set was about to be changed. The specifics of when/how measurements are taken is component-dependent, but the logic that performs this activity must be trusted to report changes faithfully. This behavior is assured by the initial measurement of the component by the component that starts it. In the BIM model, this can only be a static (platform TCB) component.

This measurement model provides better scalability than models that use the TPM as the sole repository for measurements. By using software registers, the BIM can store a virtually unlimited number of individual measurements. Hence, no aggregation is needed. However, the measurements are still accumulated and the CCRs are irreversible. That is, recording a measurement M_1, followed by a changed measurement M_2, followed by M_1 again, results in a different value than the original recording of M_1 alone. Hence, components are not allowed to change in any way without permanent loss of integrity. Even if a change is later undone, the component cannot return to its previous trust state. In Section 5, we will address this problem by employing dynamic registers for reversible measurements.

4.1.2 Simple Trust Dependency

The BIM service implements a flat hierarchy to capture the integrity dependencies between platform components. In this model, the integrity of dynamic components solely depends on the integrity of the underlying

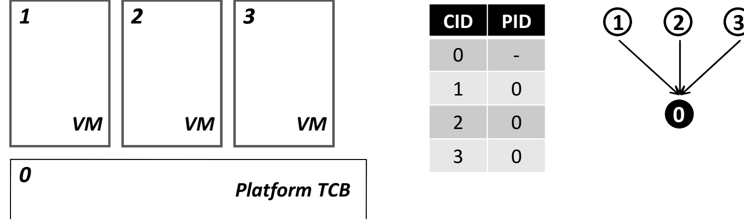


Figure 3: Simple integrity use case – a flat hierarchy.

platform TCB. We show an example flat hierarchy in Figure 3. The components labeled *one*, *two*, and *three* are virtual machines running directly on the trusted platform. Component *zero* is the platform TCB that includes the SRTM (in this case, the BIM service). Each VM depends only on the platform TCB underneath. If the integrity of the TCB (component *zero*) is compromised, then the integrity of all of the VMs is compromised as well. However, the VMs are independent of one another and therefore do not have a trust dependency. As an example, if the integrity of VM_1 is compromised, the integrity of VM_2 and VM_3 remains intact.

In what follows, we depict the integrity relationships between components using a dependency graph, and represent it using a dependency table. Figure 3 shows a simple graph and its dependency table equivalent. For example, the second row in the dependency table states that the integrity of the child component *one* (VM_1) depends on the integrity of the parent component *zero* (TCB).

In the simple BIM model, there is always a single trusted component (the platform TCB) on which all other components depend. This yields the "flat hierarchy" dependency graph and table in Figure 3. The flat hierarchy arises, because a dynamic component (such as a VM) can only be started by a trusted component. Since the TCB is static and platform-wide, it is not possible for a dynamic component to start – and hence become a parent of – another dynamic component. Therefore the BIM cannot manage, for example, the integrity of an application started within a VM. However, the BIM serves as a basis to build the hierarchical model which addresses this limitation, which is introduced in Section 5.

4.2 The BIM Architecture

As shown in Figure 2(b), BIM services are grouped under three interfaces:

Integrity interface This interface provides functions to report and quote integrity measurements of dynamic (i.e., non-TCB) components. Components use this interface to extend their register values when they detect significant changes to their measured content. A component is only allowed to alter its own register, while an integrity quote can be requested by any entity. Using the underlying TPM interface, the latter operation returns a signed integrity digest that contains the measurements of the dynamic component and the platform TCB. Using this digest, a third party can verify the complete integrity chain.

Protected storage interface This interface provides functions to store and reveal secrets on behalf of dynamic components. These secrets are bound to the integrity of the TCB and the owner component, i.e., they are revealed if and only if the integrity of the component and its ancestors (in the BIM case, the platform TCB) is intact. The BIM uses the underlying TPM interface for sealing and unsealing data to and from the TPM, which automatically implies a verification check on the TCB. Verification of the component's integrity can be done either by the BIM or delegated to a third-party verifier. Our prototype implements the former case in which the BIM needs to store the expected measurements for comparison. We use the TPM sealing operation itself to do so and use the CCR values at the time

Integrity	Description
`extend`	Takes a hash value as an argument and irreversibly extends the component CCR with that hash.
`quote`	Takes arbitrary external data (i.e., nonce) and returns a quotation of the current TCB measurements, the nonce, and the component CCR value signed by a TPM attestation identity key (AIK).
Protected Storage	**Description**
`seal`	Takes data to be protected, seals it to the TPM binding it to current TCB and CCR measurements, and returns the sealed (encrypted) blob.
`unseal`	Takes the sealed blob and unseals and returns the data iff the integrity of the TCB and the component are verified as intact.
Management	**Description**
`register`	Takes the initial measurement, adds the component to the dependency table, and fills the CCR with the initial measurement.
`delete`	Deletes the component and all its sealed data.

Table 1: BIM integrity, protected storage, and management interfaces.

of sealing as the future expected values. We concatenate these values to the secret and seal the whole blob. The unsealing operation at a later time returns not only the sealed secret but also the expected set of measurements that we compare to the CCR values at that time.

Management interface This interface provides functions to register dynamic components to the framework so that their integrity can be tracked by the BIM. The BIM is a passive service, and so only registered components are tracked. As discussed in the previous sections, the initial measurement of the component is provided from outside by a trusted component that measures and initiates the component. The interface also allows the deletion of components and their sealed data.

Table 1 details the individual functions provided by each interface. As shown in Figure 2(b), the BIM, in turn, makes use of the Basic Management and Security Interface (BMSI), which provides a platform-agnostic interface to the underlying hypervisor and hardware TPM. In particular, the BMSI provides functions that enable the BIM to access the TPM and establish a link to the hardware root of trust. The implementation of the BMSI is discussed in more detail in Section 7.

5 Hierarchical Integrity Management

In this section, we present an enhanced design for the SRTM service that we introduced in Section 3. This platform-independent service features dynamic measurements and a component hierarchy that we use to manage the integrity of dynamic (non-TCB) platform components more effectively. We describe the security model for measurements in Section 5.1. We describe the service architecture and interfaces in Section 5.2.

5.1 Measurement Model

The Hierarchical Integrity Management (HIM) service stores integrity measurements in a CCR table as illustrated in Figure 2(a). To overcome the shortcomings of the BIM model (e.g., irreversible measurements), we have extended it by introducing two new concepts: dynamic measurements and hierarchical trust.

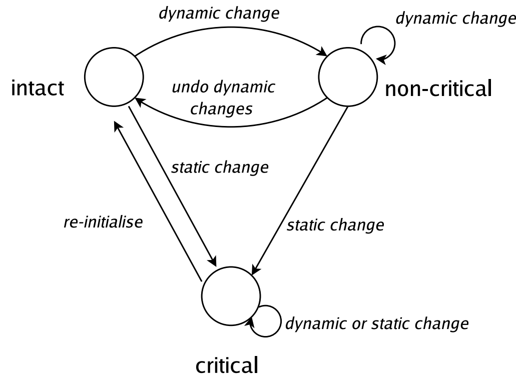


Figure 4: Transition diagram for component integrity states. A component in the non-critical state can be made intact by undoing dynamic changes, but the critical state can only return to the intact state by re-initialization.

5.1.1 Dynamic Measurements

The HIM measurement model enhances the BIM model in two ways. First, HIM allows multiple registers to be assigned to a single dynamic component. This way, component measurements can be tracked with better granularity. Second, HIM supports dynamic measurements that can be reported to a resettable register. This increases flexibility and allows a component to revert back to a trustworthy configuration if permitted by its change policy.

Change types. We distinguish two types of component changes. More specifically: An *irreversible change* is one that requires the component to be restarted before its integrity can be re-established. Such a change is one made to the integrity-critical part of the component; that is, to the code or other data of the component that has a potential impact on the future ability of the component to implement its intended functionality correctly. An example of an irreversible change is a kernel loading an untrusted device driver as the driver may make a change to kernel memory that will persist even after it is unloaded.

A *reversible change* is one in which the component is permitted to re-establish integrity without being completely reinitialized. Such a change is one made to a non-critical part of the component; that is, to code or other data of the component that has no direct or potential impact on the component's future security. A component still loses its integrity if a change is made to it. However, depending on the exact nature of the change, we may permit the component to regain integrity (and therefore trust) by undoing the change and returning to its previous state. For example, changes to configuration parameters are often reversible – e.g. changing the identity certificate that a component uses. The integrity management system will need to note such a change in order to fully report the state of the platform, but the certificate may be safely changed back without causing security implications. Another example might be loading a trusted kernel module that is known not to leave any side effects after being unloaded.

The categorization of a change as reversible or irreversible is component-dependent and will be set by each component's own change-type policy. For example, a policy stating that all changes are irreversible reduces to the static measurement model. A component that permits reversible changes is referred to as a *dynamic component* ("dynamic" because its integrity state may change multiple times).

Measurement reporting. Recording dynamic measurements requires two measurement registers, a *static register* and a *dynamic register*, rather than the single register used in the static measurement model. Irreversible changes are reported to the static register in the same way as in the static measurement model; that is, the `extend` operation is used to combine the new measurement with the existing register value to obtain the new register value.

$$extend(R, M) = hash(R||M)$$

where R is the value of the register and M is the measurement.

By contrast, reversible changes are reported to the dynamic register by *replacing* the previous value held in that register, using the `reset` operation.

$$reset(R, M) = M$$

We can see that attempting to reverse an irreversible change does not return the static register to its initial state:

$$R_{final} = extend(extend(R_{initial}, M_2), M_1) = hash(hash(R_{initial}||M_2)||M_1) \neq R_{initial}$$

However, reversing a reversible change *does* return the dynamic register to its initial state:

$$R_{final} = reset(reset(R_{initial}, M_2), M_1) = reset(M_2, M_1) = M_1 = R_{initial}$$

The exact nature of the reporting activity and the corresponding change-type policy is component-dependent. However, the logic that performs this activity must be a part of the initial measurements so that we can trust the component to report the changes to the correct register.

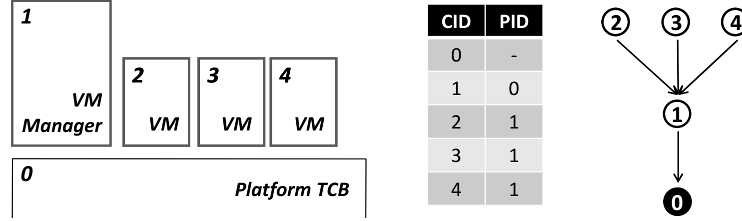
Integrity states. Depending on the measurement values stored in its static and dynamic registers, a dynamic component can be in one of three local integrity states: intact, non-critical, and critical. The component is in the *intact state* if and only if the values in the static and dynamic registers are consistent with the expected measurement values. The component is in the *non-critical state* if and only if the value in the static register is consistent with the expected measurement value but the value in the dynamic register is not. In all other cases, the component is in the *critical state*. As shown in Figure 4, the foregoing arrangement enables a dynamic component that has only been subject to non-critical changes to be restored to the intact state. A component that is in the critical state cannot be restored to any other state unless re-initiated with an expected configuration (during which both registers are reset).

Security states. Depending on the integrity state, a component can be in three security states: trustworthy, secure, and insecure. A component is *trustworthy* if and only if it is in intact state. A component is *secure* if and only if it is in intact or non-critical states. In all other cases, the component is deemed *insecure*.

Example use case for dynamic registers. Digital Rights Management (DRM) services control the distribution of media content onto computing platforms. It is possible that a DRM service will not push video content to a computing accessory if, for example, an external recording device is plugged to it. In this case, software that detects and installs the plug-and-play drivers for the recording device must be part of the static measurements. However, the state in which a recording device is detected in the system can be reported dynamically. In fact, this can be reflected in the dynamic register for a secure DRM player application. As long as the recording device is connected, no content is downloaded. Once the user unplugs the device, the dynamic register is reset and content can be pushed to the player without requiring the application to be restarted.

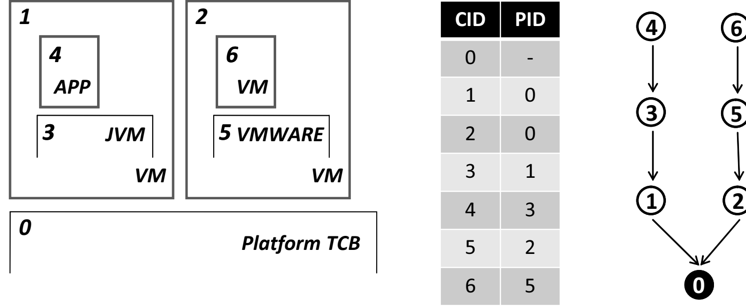
5.1.2 Hierarchical Trust Dependency

We enhance the BIM dependency model by introducing a hierarchy of trust dependencies that we represent as a directed acyclic graph. In such a graph, the edges indicate trust dependencies where the integrity of the component at the origin depends on the integrity of the component at the destination. If the integrity of the destination component is compromised, then the integrity of the origin component is always compromised



CID	PID
0	-
1	0
2	1
3	1
4	1

(a) Multi-level dependency.



CID	PID
0	-
1	0
2	0
3	1
4	3
5	2
6	5

(b) Nested components.

Figure 5: Hierarchical integrity use cases.

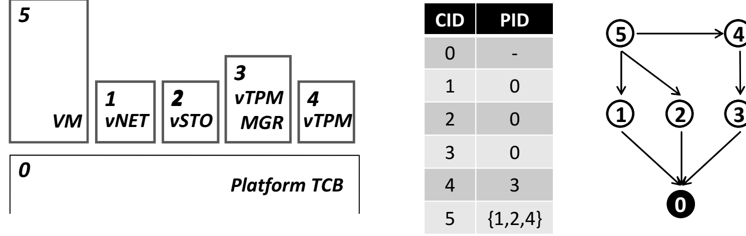
as well. However, the reverse is not true. To illustrate these more complex trust relationships, consider the following use cases.

In Figure 3, we see the simple flat hierarchy as previously described in Section 4. The components labeled *one*, *two*, and *three* are virtual machines running directly on the trusted platform. Component *zero* is the platform TCB that includes the SRTM (in this case, the HIM service). Each VM depends only on the platform TCB underneath. If the integrity of the TCB (component *zero*) is compromised, then the integrity of all of the VMs is compromised as well. However, the VMs are independent of one another and therefore do not have a trust dependency. As an example, if the integrity of $VM_1$is compromised, the integrity of VM_2 and VM_3 remains intact.

Figure 5(a) shows a more complex multi-level dependency. Component *one* is a service that manages the life-cycle of components *two*, *three*, and *four*. All components are virtual machines. The latter VMs are independent of one another, as before, but their integrity depends on that of the domain manager, whose integrity in turn depends on the TCB.

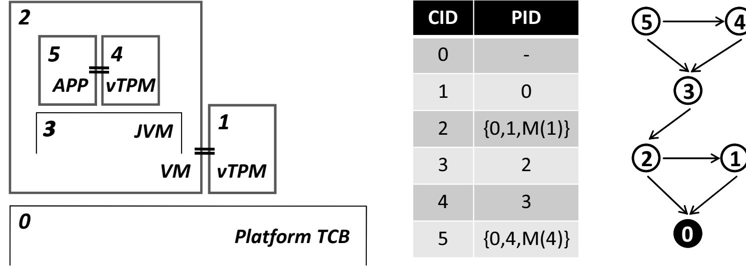
In Figure 5(b), we see a nested dependency relationship. Components *one* and *two* are virtual machines, which themselves contain further virtual machines: component *three*, which is a Java virtual machine, and component *five*, which is a VMware hypervisor. These nested virtual machines support guest components: component *four*, a Java application, and component *six*, a VMware guest. Within component *one*, a traditional linear chain-of-trust applies: Java application depends on Java virtual machine depends on operating system. A similar chain can be found within the VMware component. However, these two chains of trust are independent of one another, and both depend ultimately on the underlying platform TCB.

Figure 6 illustrates more complicated use cases. In Figure 6(a), we see a multiple dependency relationship. Component *five* is a virtual machine that uses services from components *one*, *two*, and *four*. These components are small virtual machines that provide virtual networking, virtual storage, and virtual TPM services, respectively. Further, the integrity of the virtual TPM depends on the integrity of the virtual TPM manager domain (component *three*).



CID	PID
0	-
1	0
2	0
3	0
4	3
5	{1,2,4}

(a) Disaggregated services.



CID	PID
0	-
1	0
2	{0,1,M(1)}
3	2
4	3
5	{0,4,M(4)}

(b) Virtual TPM binding.

Figure 6: More complicated use cases. Dashed lines denote implicit dependency.

Figure 6(b) shows a similar VM grouping example which we intend to explore further in future work. In this example, we use miniature virtual TPM services to assist and enhance the integrity measurement capabilities of the framework. In this design we bind a single virtual TPM to a component (application or VM) and delegate component measurements to this virtual TPM. The virtual TPM then replaces the component CCRs to provide more granular run-time measurements for the component it is attached to. The measurements for the virtual TPM service itself is still held by its own CCRs. As an example, the integrity of component *two* now depends on the integrity of component *one* (its attached virtual TPM) and the run-time measurements taken by this virtual TPM (e.g., during authenticated VM_2 bootstrap). We refer to this measurement set as M(*one*). The same holds for the application component *five* and its attached virtual TPM service component *four*. The present HIM implementation does not yet support virtual TPM attachment.

5.2 The HIM Architecture

The HIM service implements the same integrity, protected storage, and management interfaces as the BIM service as presented in Section 4, but with the following enhancements.

The HIM integrity interface provides an `extend` function that alters the value of the static CCR in the same way as the BIM equivalent. To support dynamic measurements, the interface also provides a `reset` function that is used to report to the dynamic register and overwrite its value. In addition, to support hierarchical integrity dependency, the `quote` function is modified. This function now returns the aggregated integrity measurements of the component in question. Specifically, the signed quote now contains the TCB integrity measurements plus the measurements of the component and all its ancestors hashed in a single value.

In the HIM protected storage interface, the `seal` and `unseal` functions are enhanced to support component dependency and dynamic measurements. The `seal` function now binds the stored secret to the integrity of all the trust chains that reach the component in question from the TCB; that is, the subgraph of all paths from that component to the root TCB. Hence the integrity state of components not on a path

between that component and the TCB is ignored. For example, in the nested use case in Figure 5(b), an integrity compromise in the VMware compartment will not affect the ability of the Java application to unseal previously sealed information, as long as the Java compartment remains intact.

Lastly, the HIM management interface provides `register` and `delete` functions. The `delete` function is the same as in the BIM. However, the `register` function now takes a dependency list as a parameter that specifies additional ancestor components the component depends on besides the one that registers the component.

6 Policy Verification for Security Services

In this section, we introduce example security services that leverage the HIM framework for policy verification and access control. Our examples include a credential management service (Section 6.1), a virtual TPM service (Section 6.2), and a virtual network service (Section 6.3).

6.1 Credential Management Service

Protected storage services provide secure access to secrets that are sealed to the underlying TPM on behalf of their owners. It is expected that these services retain control over these secrets and enforce the associated access control policies at all times. By contrast, most storage services such as [23] and the HIM provide one-time verification, and are therefore susceptible to a time-of-check to time-of-use vulnerability. This occurs because these services release the stored secret to the requesting component once they verify the necessary policies (e.g., HIM unseal successfully verifies the aggregate integrity). Once the secret is revealed, these services can no longer restrict access to it if the component undergoes a malicious change.

To enable ongoing policy verification and enforcement, we designed and implemented a credential management service (CMS) that uses the integrity management framework to provide secure access to secrets while maintaining control at all times. Unlike the HIM unseal operation, CMS credentials are never revealed to requesting services directly but are always held securely by the CMS. In essence, the CMS is a reference monitor that mediates and provides access to secured data through a well-defined interface.

The CMS interface is comprised of management and service interfaces. Components use the management interface to register component credentials with the CMS. To do so, the `register` function takes the credential as input and seals it to the underlying TPM. The interface also provides a `discard` function which deletes the stored credential. The service interface provides access to the credentials through a generic `access` function. We have designed this interface as an extensible plug-in interface; that is, the exact nature of the interface depends on the nature of the stored credential and the type of functionality needed. For example, if the stored credentials are cryptographic keys, we offer a plug-in service that provides encryption/decryption capabilities so that components can use the interface to encrypt/decrypt data without seeing the actual key. Regardless of the functionality provided, the CMS uses the HIM to verify the aggregate integrity prior to each access to the secret.

6.2 Virtual TPM Service

A natural extension to the CMS functionality would be to provide a miniature TPM interface to the various platform components, as illustrated in Figure 6(b). This enables these components to have a standardized interface as in [26] to prove their integrity and provides a strong identity for each component. Such an approach has already been taken through TPM virtualization [4] which gives each VM a TPM interface implemented by a virtual TPM service. However, it is not yet clear what the best mechanism is for establishing a secure binding between a virtual TPM and its platform TCB.

Our framework could be used to bridge the gap between virtual TPM services and the platform TCB. For example, a central trusted CMS service could be used as the single secure repository for virtual TPM keys. Access to these keys would require verification of the complete HIM integrity chain, including verification of the platform TCB. For example, to sign a quote request, a virtual TPM would use the CMS interface to gain access to its signing key.

6.3 Virtual Network Service

Virtualization provides direct isolation of computing resources such as memory and CPU between guest operating systems on a physical platform. However, the network remains a shared resource as all traffic from guests will eventually end up on the same physical medium. Various mechanisms can be used to provide network isolation between network domains, as described in [6]. In general, encryption must be used for isolation when network traffic is delivered over an untrusted shared physical medium.

Using our framework in combination with the CMS, one could design a virtual network (vNET) service which provides isolation through an encryption layer such as IPSEC. In this setting, the vNET service would store its credentials (e.g., network encryption key) in the CMS, in combination with the expected CCR values of the service and any ancestor service it depends on (including any potential network configuration information). Because the key is held by the CMS and not revealed to the vNET service, any change in the integrity of the service or its ancestor components would result in the network link becoming unavailable for the VM connected to this specific vNET. As a result, the capability of a VM to communicate with its peer within a considered domain would implicitly prove its trustworthiness, which would provide continuous authentication as opposed to relying only on an initial handshake as most network authentication mechanisms do.

7 Implementation in Xen

In this section, we describe a prototype implementation of the integrity management framework and the credential management service on the Xen virtual machine monitor [9]. The implementation features the management and service interfaces of both. Note that although we present our implementation with Xen, the framework could equally be implemented on an alternative virtualized or microkernel-based platform (e.g., the L4/Fiasco [16] microkernel).

7.1 Infrastructure Overview

The various components of the integrity management framework are provided by one or more virtual machines, running on top of the Xen virtual machine monitor. The use of virtualization isolates the trusted platform from a misbehaving guest operating system, and all communication with the trusted platform passes through well-defined interfaces. Our implementation is based on Xen version 3.0.4, a VMM for the IA32 platform, with the VMs running a paravirtualized version of Linux 2.6.18. For inter-domain communication, we employ the light-weight communication library introduced in [2].

Figure 7 illustrates our implementation on Xen. In the present prototype, all framework components and the CMS are implemented as libraries and services running in the Xen privileged management domain Dom0. However, as we have defined interfaces between each of the components, it should be straightforward to move towards a disaggregated approach as described in [19]. The framework components are arranged in a layered stack. At the lowest layer is the basic management and security interface (BMSI) that provides libraries for domain life-cycle management (libM), basic TPM access (libT), and integrity management (libI). At the core services layer are the integrity manager services BIM and HIM that provide basic and hierarchical integrity management, respectively. Also in this layer are the CMS and the domain management

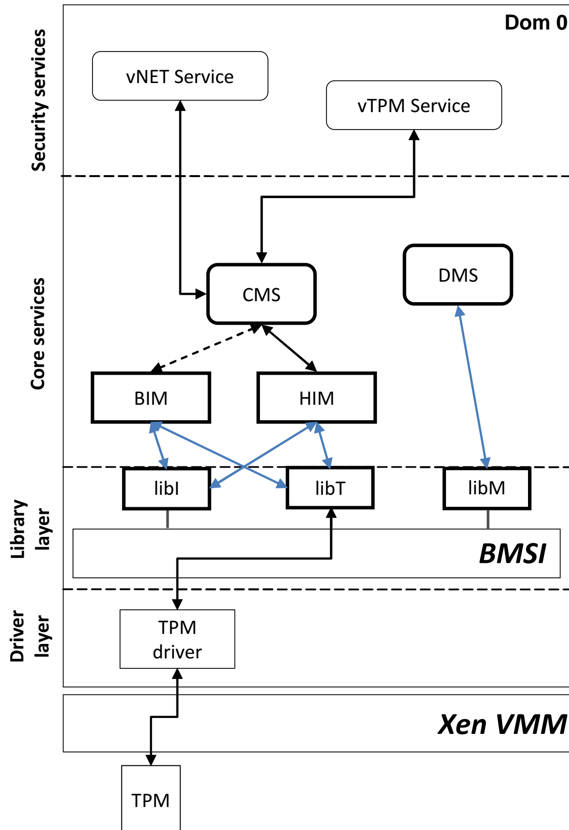


Figure 7: Illustration of the prototype in a layered stack.

service (DMS). At the highest layer are the security services that use the framework for various purposes. The platform TCB consists of the static components up to and including the SRTM (the BMSI libraries and the integrity managers). However, for simplicity, we also include the CMS in the platform TCB. The measurements of these components are reported to the underlying TPM. The application TCB consists of the platform TCB plus the security services that run on top of it. The measurements of the latter are reported to the SRTM.

7.2 Component Design

In the present prototype, we have implemented the highlighted components depicted in Figure 7, namely the BMSI libraries, BIM and HIM services, CMS, and DMS. In this section, we present the details of these components including both the BIM and HIM; however, due to space constraints, we present an example use case that uses only the HIM.

7.2.1 BMSI Libraries

The Basic Management and Security Interface (BMSI) provides a common and extensible interface to the underlying hypervisor (i.e., Xen) and the TPM. The BMSI provides libraries for domain life-cycle management (libM), basic TPM access (libT), and integrity management (libI).

libM This library provides hypervisor-agnostic management functions to upper layers. At its lowest level, the library manages allocatable resources called *Protection Domains (PDs)*. A PD is an executable

component that receives an allocation of memory and CPU cycles, and is scheduled by the hypervisor. On Xen platforms, a PD is equivalent to a Xen domain (virtual machine). In this prototype, we use libM to implement the Domain Management Service (DMS). This service manages the life-cycle of PDs and uses the integrity managers to keep track of PD integrity. We refer the reader to [19] for further details on the libM and DMS implementation.

libT This library provides the minimal functionality to access the integrity and protected storage interfaces of the TPM. Security services (e.g., BIM and HIM) use this library to obtain a signed quotation of the TCB measurements and to seal/unseal data to/from the TPM. To do so, libT uses the TPM functions `TPM_Quote()`, `TPM_Seal()`, and `TPM_Unseal()` as described by the TPM specification [26].

libI This library stores and provides access to the integrity measurement and dependency tables. The `getMeasurement()` function returns a measurement list that includes the integrity measurements of the component and its ancestors. In the BIM case, a single value is returned. The `setMeasurement()` function extends the value of the component register. The `resetMeasurement()` function overwrites the value of the dynamic register. The `addComponent()` function adds an entry to the dependency table and sets its dependencies as specified. It also adds an entry to the measurements table and records the initial measurements. The `deleteComponent()` function checks that the specified component has no successors and removes it from the table.

7.2.2 Component Interactions

The BIM and HIM services implement the interfaces presented in Sections 4 and 5, respectively. Similarly, the CMS service implements the interfaces presented in Section 6.1 and uses a cryptographic service as a plug-in for block encryption and decryption. On a Xen platform, we use these services to manage the integrity of VMs and applications running on these VMs.

VM integrity management is incorporated into VM life-cycle management. To assist both, the DMS uses the BMSI library libM and the HIM service. The VM start-up phase in Figure 8 depicts the interaction among these components. During this phase, the DMS invokes libM, which prepares resources for the VM, measures the VM image (comprising the kernel, an optional initial ramdisk and command-line parameters), and stores the measurement in the CCR for that VM. This performs a function similar to a secure bootloader, and it is the responsibility of the kernel to measure any components which it subsequently loads. The DMS also registers the new VM with the HIM service, and configures any dependencies between the new VM and existing VMs. The HIM uses libI to store this information in the measurement and dependency tables. Following the successful completion of the above steps, the DMS starts the VM.

The HIM service additionally allows applications running in VMs to be registered with the framework. The application start-up phase in Figure 8 depicts the case in which the VM that was started in the previous phase loads and registers a DRM service with the HIM. In this case, the VM becomes an ancestor of the service and provides its initial measurements. As a result, the cumulative integrity of the service now includes the VM's measurements as well as the platform TCB measurements.

The last phase in Figure 8 depicts a use case in which the DRM service that was started in the previous phase attempts to decrypt encrypted media content using a key that is stored on the TPM on behalf of this service. The DRM service invokes the CMS service interface to request access to this key. The CMS then invokes HIM unseal to retrieve the key from the TPM. HIM unseals the key if and only if the underlying policies regarding the key's release are satisfied. In this case, the key is unsealed from the TPM and returned to the CMS if the integrity of the platform TCB is intact. On receiving the key from the HIM, the CMS performs further verification. It compares the expected CCR values of the DRM service and its ancestor VM (unsealed along with the key) to the current CCR values. If the measurements match, the CMS uses its cryptographic service to decrypt the block, which is then returned to the DRM service. Note that any

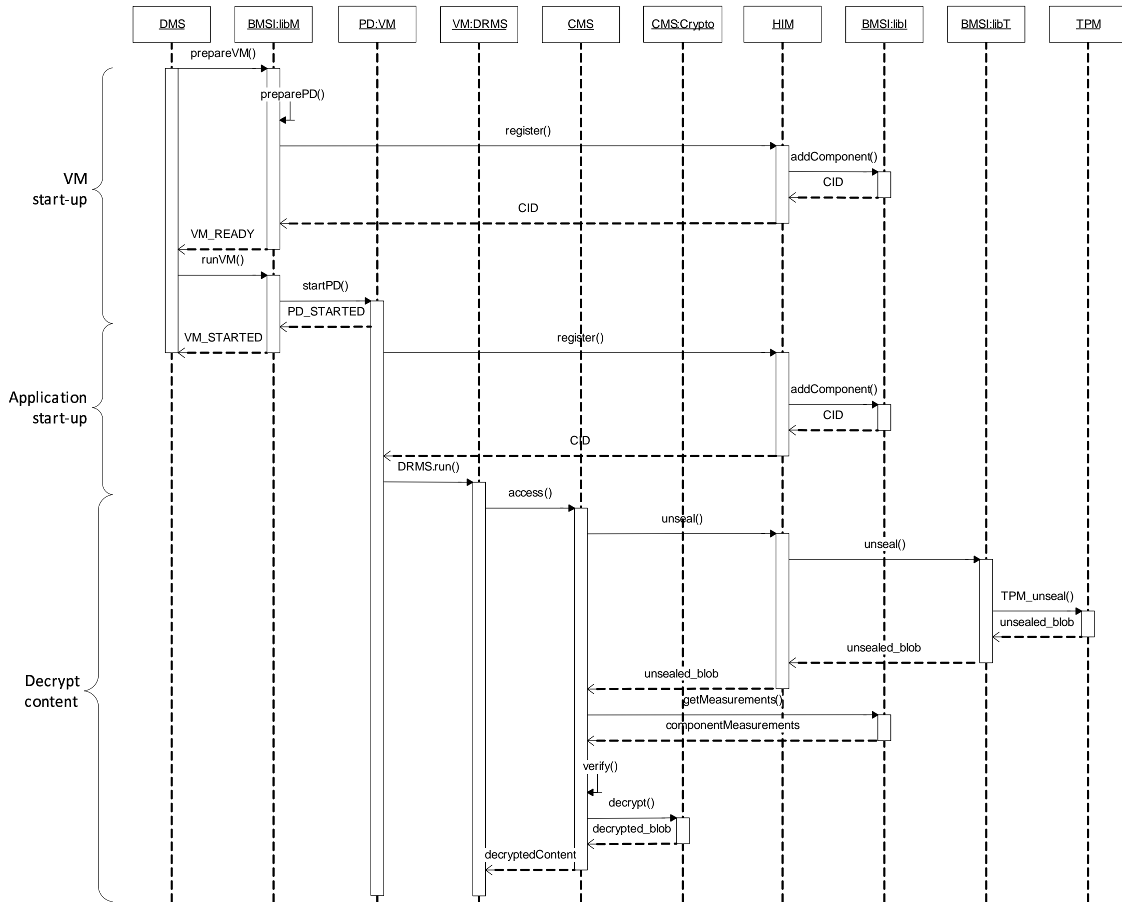


Figure 8: Sequence diagram of interactions between the framework services for a DRM application use case.

subsequent access requests to the key will also follow a similar verification cycle, with the exception that HIM (hence TPM) seal is omitted because the CMS caches the key internally.

8 Related Work

Berger *et al.* [4] implemented a virtual TPM infrastructure in which each virtual machine is assigned its own virtual TPM that provides multiplexed access to the underlying hardware TPM. In comparison to virtual TPMs, our work uses a single integrity management framework that encompasses all components in order to explicitly represent trust dependencies between them. Our framework is complementary to virtual TPMs in that we can use virtual TPMs to gather more granular run-time measurements for our components, and can enhance virtual TPMs by providing a binding between them and the platform TCB through the use of CMS.

Several systems have been previously described that use virtual machine monitors to isolate trusted and untrusted components. Terra [11] is an architecture that uses a trusted virtual machine monitor (TVMM) to bring the security advantages of "closed box" special-purpose platforms to general-purpose computing hardware. The TVMM ensures security at the virtual machine level, isolating VMs from one another, providing hardware memory protection, and providing cryptographic mechanisms for VMs to attest their integrity to remote parties, even providing protection from tampering by the platform owner. Microsoft's

proposed Next-Generation Secure Computing Base (NGSCB [10]) operates in a similar manner, partitioning a platform into two parts running over a virtual machine monitor: an untrusted, unmodified legacy operating system, and a trusted, high-assurance kernel called a *nexus*. Our work builds on both to examine how integrity measurements can be stored and maintained.

Sailer *et al.*'s implementation of a TCG-based integrity measurement architecture [23] was one of the earliest works to demonstrate the use of a TPM to verify the integrity of a system software stack. In [14], Jansen *et al.* propose an architecture for protection, enforcement, and verification (PEV) of security policies based on a tree structure containing integrity log data, where each node contains the data for one component and its children contain the data for its subcomponents. PEV approaches the problem of trust flexibility and extensibility by defining a generalized attestation protocol. A verifier sends an attestation request containing an XML descriptor that defines a projection function returning the subset of the integrity log of interest to the verifier. Sadeghi *et al.* [21] extend the TCG notion of trust in a different direction by proposing attestation that is not based directly on hardware/software hashes but on abstract platform properties. Rather than checking a large list of permitted platform configurations, their system checks whether or not a given platform possesses valid certificates attesting to the desired properties. Such property certificates are issued by a trusted third party that associates concrete configurations with the properties they provide. Our system differs from these in providing a more granular verification of components such as individual virtual machines and applications within a platform, representing dependencies among them, and managing changes to measured components.

Other orthogonal previous work has explored distributed trust and mandatory access control. Griffin *et al.* investigated secure distributed services with Trusted Virtual Domains [12], which are intended to offload security analysis and enforcement onto a distributed infrastructure. Berger *et al.* use this abstraction in the Trusted Virtual Datacenter (TVDc) [5], which shares hardware resources among virtual workloads while providing isolation with a mandatory access control policy enforced by the sHype security architecture [22].

9 Conclusions

In this paper, we have introduced a novel integrity management framework that improves on the integrity measurement and policy verification capabilities of present Trusted Computing solutions. In particular, our framework is able to cope with the proliferation of measured components and dependencies between them as well as dynamic changes to platform components. In essence, the framework implements a small, software-based root of trust for measurement (SRTM) that provides a secure link to the core root of trust for measurement (CRTM). We have implemented our framework on the Xen virtual machine monitor and proposed several ways in which security services could take advantage of this architecture for policy verification and access control.

We anticipate integrity and trust management to become especially useful for application and service level components. We will therefore continue to investigate further potential uses for our framework by user level applications. In the short term, we plan to implement CMS-aware services such as a virtual network service based on [6] that uses the CMS to store encryption keys. The virtual TPM service is also particularly interesting. In the long term, we plan to investigate various ways of exploiting our framework to help enhance the security properties of virtual TPM services (e.g., their binding to the platform TCB). Conversely, we plan to use virtual TPM services to help enhance the measurement capabilities of the HIM framework and provide more granular run-time measurements compared to a single CCR.

Acknowledgments

This work has been partially funded by the European Commission as part of the OpenTC project (www.opentc.net). We thank Chris I. Dalton, Robert F. Squibbs (HP Labs), Carsten Weinhold (TU Dresden), and our partners in OpenTC for valuable discussions and inputs.

References

[1] Microsoft security advisories archive. http://www.microsoft.com/technet/security/advisory/archive.mspx.

[2] M. J. Anderson, M. Moffie, and C. I. Dalton. Towards trustworthy virtualisation environments: Xen library OS security service infrastructure. Technical Report HPL-2007-69, Hewlett-Packard Development Company, L.P., April 2007.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[4] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2006. USENIX Association.

[5] S. Berger, R. Cáceres, D. Pendarakis, R. Sailer, E. Valdez, R. Perez, W. Schildhauer, and D. Srinivasan. TVDc: Managing security in the Trusted Virtual Datacenter. *ACM SIGOPS Operating Systems Review*, 2008.

[6] S. Cabuk, C. I. Dalton, H.V. Ramasamy, and M. Schunter. Towards automated provisioning of secure virtualized networks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, 2007.

[7] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[8] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 351–366, New York, NY, USA, 2007. ACM.

[9] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[10] P. England, B. Lampson, J. Manferdelli, and B. Willman. A trusted open platform. *Computer*, 36(7):55–62, 2003.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 193–206. ACM Press New York, NY, USA, 2003.

[12] J. L. Griffin, T. Jaeger, R. Perez, R. Sailer, L. van Doorn, and R. Caceres. Trusted Virtual Domains: Toward secure distributed services. *Proc. of 1st IEEE Workshop on Hot Topics in System Dependability (HotDep)*, 2005.

[13] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB size by using untrusted components: Small kernels versus virtual-machine monitors. In *Proceedings of the 11th ACM SIGOPS European workshop: beyond the PC*. ACM Press New York, NY, USA, 2004.

[14] B. Jansen, H. V. Ramasamy, and M. Schunter. Policy enforcement and compliance proofs for Xen virtual machines.

[15] B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of the 16th USENIX Security Symposium*. USENIX Association, 2007.

[16] The Fiasco micro-kernel, 2004. Available from `http://os.inf.tu-dresden.de/fiasco/`.

[17] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[18] A. Marx. Outbreak response times: Putting AV to the test. `http://www.avtest.org`, 2004.

[19] D. G. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *Proceedings of the ACM conference on Virtual Execution Environments*, March 2008.

[20] Qumranet. KVM: Kernel-based virtualization driver. `http://kvm.qumranet.com`, 2006.

[21] A. R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. *Proceedings of the 2004 workshop on New security paradigms*, pages 67–77, 2004.

[22] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype: Secure hypervisor approach to trusted virtualized systems. *IBM Research Report*, 2005.

[23] R. Sailer, X. Zhang, and T. Jaeger. Design and implementation of a TCG-based integrity measurement architecture. *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13 table of contents*, pages 16–16, 2004.

[24] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350, New York, NY, USA, 2007. ACM.

[25] J. Sugerman, G. Venkitachalam, and B-H Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[26] Trusted Computing Group. TCG Specification Architecture Overview. Trusted Computing Group: `https://www.trustedcomputinggroup.org/groups/TCG_1_3_Architecture_Overview.pdf`, March 2003. Specification Revision 1.3 28th March 2007.