Trusted Integrity Measurement and Reporting for Virtualized Platforms

(Work-in-Progress)

Serdar Cabuk[1], Liqun Chen[2], David Plaquin[2] and Mark Ryan[3]

[1] serdar.cabuk@gmail.com
[2] Hewlett-Packard Laboratories
{liqun.chen, david.plaquin}@hp.com
[3] University of Birmingham
m.d.ryan@cs.bham.ac.uk

Abstract. Verifiable trust is a desirable property for computing platforms. Current trusted computing systems developed by Trusted Computing Group (TCG) provide verifiable trust by taking immutable snapshots of the whole set of platform components. It is, however, difficult to use this technology directly in virtualized platforms because of complexity and dynamic changes of platform components. In this paper, we introduce a novel integrity management solution based on a small Software-based Root of Trust for Measurement (SRTM) that provides a trusted link to the integrity measurement chain in the TCG technology. Our solution makes two principal contributions: The first is a key management method, by which a verifier can be convinced that the SRTM is a trusted delegatee of a Trusted Platform Module (TPM). The second is two integrity management services, which provides a novel dependency relation between platform components and enables reversible changes to measured components. This extended abstract of the paper focuses on the key management method and shows the high level idea of these two services. Details of the dependency relation, the reversible changes, and the Xen implementation may be found in the full version of the paper.
Keywords. Integrity measurement and reporting, platform virtualization, software-based root of trust for measurement

1 Introduction

Trusted Computing has been proposed as a means of providing verifiable trust in a computing platform. The basic idea of TCG integrity measurement and reporting solution is an integrity measurement chain refered to as the chain of trust. The root of the chain is a *Trusted Platform Module (TPM)*, which maintains a number of *Platform Configuration Registers (PCRs)* that are the cryptographic hash values of every component code in the chain, and then reports the PCR values to a local or remote user (also called verifier) by using digital signatures. By verifying the signatures the verifier obtains a trusted report of the platform configuration.

In the past ten years virtualization has gradually become a popular technology to achieve security and performance requirements on computing platforms. In essence, virtualization enables simple consolidation and isolation of multiple virtual machines (VMs) on the same platform. Virtualization has also introduced new challenges to platform measurement and reporting systems. In particular, such a system now needs to retain more information about the state of the platform and keep track of complex trust dependencies between platform components. Consider a web appliance that comprises a web server and a database server that run on different compartments on the same physical host. The correct (i.e., expected) operation of this appliance depends on the correct operation of each component that runs in its own isolated execution environment. To verify the expected behavior, thus, requires the integrity service to keep track of the integrity measurements of each component including the virtualization layer and also understand the logical trust dependency between the web server and its database counterpart. In this paper, we refer to this logical dependency between platform components as a *hierarchical dependency*.

Existing TCG solutions use the TPM as the sole repository for integrity measurements and reporting of single systems.They take immutable snapshots of a whole platform, which are then used as proof of trustworthiness [6, 7, 11, 21]. They do not, however, provide granular verifications of platform components such as individual VMs and applications. Further, current solutions do not support authorized changes to be made to measured components and deem all such changes to be malicious [24]. This is impractical for modern computing environments, which undergo a constant bombardment of security patches and policy changes. As a result, a new challenge is to use existing TCG integrity measurement and reporting solutions in virtualized platforms. An integrity measurement and reporting service now needs to retain more information about the state of the platform and keep track of complex trust dependencies between platform components.

Our work was intended to build a new solution extensible to the present TCG result. In this paper, we introduce a novel integrity management solution based on a small *Software-based Root of Trust for Measurement (SRTM)*[1] that provides a trusted link to the TPM-based integrity measurement chain. In the chain, the TPM records the integrity of a small number of components only, and SRTM does the most workload in the measurement and reporting. The SRTM is part of the platform Trusted Computing Base (TCB), and is isolated from other components in the virtualization manner.

We summarize our contributions as follows:

1. Our solution extends the single level of the TPM measurement functionality to the two levels, i.e., it enables the verifier to become convinced of an integrity report provided by two integrity measurement components, TPM and SRTM, under the condition that the verifier has an authentic copy of the public part of TPM's Attestation Identity Key (AIK). This is achieved by a novel key management method.

[1] Please do not confuse this with Static Root of Trust for Measurement.

2. We propose a novel integrity management service to improve the existing integrity management solutions. This service explicitly represents integrity dependencies between platform components using a dependency graph and introduces a new distinction between reversible and irreversible changes to measured components.

The remainder of this paper is organized as follows. We provide background on trusted computing and virtualization in the next section. Section 3 outlines the motivation of this work and Section 4 gives an overview of our solution. Section 5 presents a key management method. Section 6 presents two integrity management services. In Section 7 we discuss related work, and finally in Section 8 we draw conclusions.

2 Background

2.1 Trusted computing

Trusted Computing technology enables third parties to remotely attest and verify the configuration of a computing platform in a secure manner. The TCG threat model addresses unauthorized subversion of platform components (software, logs, etc.) that can potentially result in violation of system policy. Existing *trusted platforms* typically contain a component that is at least logically protected from subversion (i.e. resilient to software attacks). 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).

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.

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 6.1. A secure bootloader, such as OSLO [13], 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*." [10] 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 [18, 23]. In particular, we consider the Xen VMM [5].

VMMs are increasingly used in the development of secure computing systems [3, 22, 4]. 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 [22]). 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 [7], or to provide the illusion of a physical TPM to applications running within a VM [1].

3 Motivation of this work

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 *core root of trust for measurement (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 dynamically created components and dependencies between these components. This model also does not consider dynamic changes to platform configurations that may be reversible under certain conditions; all such changes are deemed malicious regardless and the particular component is untrusted until it is restarted.

A traditional integrity management system in the TCG technology employs the TPM as the sole repository for integrity measurement and reporting (see Section 7). Such schemes are fundamentally limited by the hardware capabilities of a TPM and the aggregate nature of the `extend` function:

1. A TPM contains a small, limited amount of memory (PCRs). The TCG specification recommends that a TPM has at least 16 PCRs [24]. 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.
3. The `extend` function of the TPM introduces an artificial dependency on the order in which the measurements are aggregated. As a result, n platform components will yield $n!$ possible integrity measurements depending on the order they are loaded.
4. It is not possible to reverse the inclusion of a measurement in a TPM register. It is therefore, impossible for a platform component to report a non-malicious change to its integrity and revert back to a trusted state without restarting.

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 applies to 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 [1]. However, by allocating independent virtual PCRs to each component, it is no longer possible to represent real dependencies between components[2]. Further, because virtual TPMs emulate the behavior of a hardware TPM, it remains impossible to revert changes. Software measurement support is required to address the limitations of the above hardware capabilities.

To illustrate a reversible change, consider a Virtual Private Network (VPN) client on a VM that refuses to connect to the Corporate VPN when it detects another active network connection, i.e., to prevent any bridging possibility between the two. This state can be captured with a dynamic configuration file that is measured and reported. When all other connections are terminated, the VPN client allows the connection, thus reverting back to the 'trusted' state. This would be impossible in the original TCG model without restarting the VM.

4 Overview of our solution

In this paper, we propose a novel integrity management solution based on a small Software-based Root of Trust for Measurement (SRTM). This special VM component is part of the platform TCB, and is isolated from other software components. Other software VM 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 measurement chain.

In the chain, the TPM measures and records the integrity of a small number of components only, e.g. the CRTM, BIOS, Boot Loader, Virtual Machine Monitor (VMM) and the SRTM, and their configuration into its PCRs. The SRTM measures and records integrity of the remaining VM components in the platform, each of which might have a virtual TPM (vTPM), and their configuration into its *Component Configuration Register* (CCR). Since the SRTM is a piece of software, there is no limitation to the number and construction of CCRs.

Regarding to how the VM components are measured and reported, our solution includes two integrity management services, one providing static measurements and a flat trust dependency relation between the components and one providing dynamic measurements and a hierarchical trust dependency relation between them. In a flat hierarchy, the integrity of a VM depends on the underlying TCB only, i.e., logical relations between the VMs cannot be represented using this model. The hierarchical model, however, can represent logical dependencies between the VMs and application that live on these VMs. As an example, the integrity of a Java application can be represented as an aggregation of the

[2] 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.

integrity of the application, the Java Virtual Machine (JVM), the VM kernel and the TCB.

Our solution has the following design objectives:

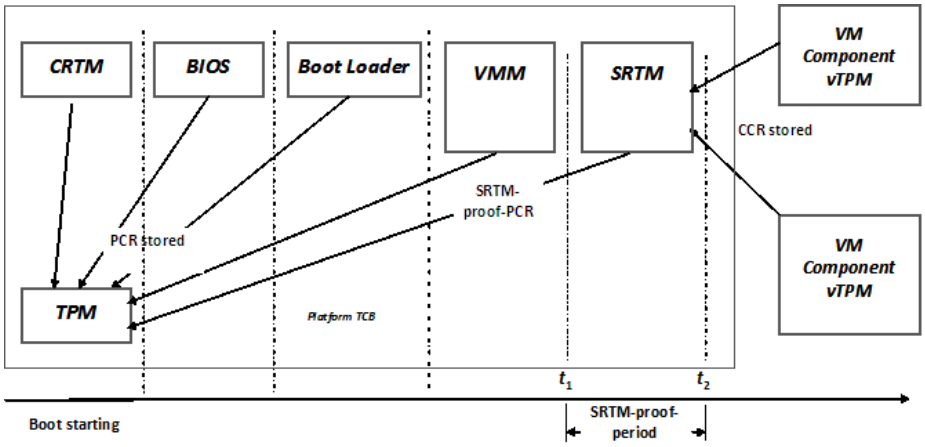


Fig. 1. The position of the SRTM within the integrity measurement chain.

Unlimited measurement storage It allows the storage of individual integrity measurements for an arbitrary number of components.

Explicit dependency representation It has the explicit and unambiguous representation of an arbitrary number of dependencies between platform components. Therefore, there is no false or artificial dependencies introduced by aggregation.

Static integrity management It provides a superset of the functionality of a traditional TPM, with respect to static integrity.

Dynamic integrity management It enables the integrity state of a platform component to revert to a previous state in a controlled and verifiable manner.

Limited workload of TPM It requires very few TPM resources for measuring and reporting a complicated platform.

Minimal TCB In order to guarantee the trustworthiness of the solution, 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 [17] or TCB reduction techniques as described in [10].

Platform independence The solution is not limited to a single hypervisor technology. The implementation was carried out on both Xen and an L4 microkernel [14, 15]. The Xen version is described in the full paper. It should also be possible to use other technologies such as VMware [23].

5 A key management method

As mentioned before, our solution extends the single level of the TPM measurement functionality to two levels, i.e., it enables a verifier to become convinced of an integrity report provided by two integrity measurement components, one by the TPM and one by the SRTM, under the condition that the verifier believes the TPM is trusted, and the verifier has an authentic copy of the public part of the TPM's *Attestation Identity Key (AIK)*. In order to build a strong cryptographic link between the TPM and the SRTM, we create an *Integrity Report Key (IRK)* for the SRTM, which is bound to the AIK. In this section, we introduce a number of ways by which to create, store and certify the SRTM IRK.

The integrity report of an arbitrary component measured and recorded by the SRTM is a joint report by the TPM and SRTM, e.g. a signature under the AIK and IRK. More specifically, the TPM records and reports the PCR corresponding to the SRTM; and the SRTM records and reports the CCR corresponding to the individual virtualised component. There are many possible ways to create such a joint report. We recommend an easy method, which does not require any change to the existing TPM commands. We furnish the SRTM with abilities in relation to the CCR similar to those that the TPM has in relation to PCRs.

The SRTM IRK is an asymmetric key pair consisting of the public IRK and private IRK. The verifier needs to be given an authentic copy of the public IRK. Then, the core technique of this solution is how to create, certify and store such an IRK. The difficulty is that the SRTM is a piece of software and has no natural capability to store any key or password secretly. To achieve this, one needs to decide:

- Which component shall create the key. There are three possible answers: a trusted external key management entity ("M"); the TPM ("T"); or the SRTM itself, at runtime ("S").
- Which component shall create the certificate for the key. Here, there are two possibilities: M and T.
- How the key will be stored. We identify five solutions, numbered 1-5 below.

We consider all the possible cases, where each case is denoted by a three character string letter-number-letter. For example, M1T means the IRK is created by the key management entity (M), the private key of the IRK is held using the storage scheme "1" below, and the key is certified by the TPM (T).

Each of our solutions works by assuming a short time period from t_1 to t_2 (shown in Figure 1). In this period, it may be assumed that all requests to the TPM come from the SRTM, since the only other active software component is the trusted VMM. The period is achieved as follows.

During the boot process, a specific set of PCRs is extended until the time t_1 when the SRTM is about to be loaded. Then the SRTM is loaded and at some time t_2 during its execution, it further extends a PCR. We call the period t_1 to t_2 the *SRTM-proof-period*. The value of the PCRs during the time period SRTM-proof-period is called *SRTM-proof-PCR*.

The IRK can be created, retrieved or certified only during this period. The verifier can assume that during this period, there is no mistrusted entity that can obtain the IRK. This idea is similar to the volume encryption key retrieval mechanism employed by Microsoft BitLocker [16]. There are five storage schemes for holding the IRK as follows:

1. S holds the private IRK. When S shuts down, it first stores the private IRK in T. When S restarts, it retrieves the key from T during SRTM-proof-period.
2. Same as 1, but it stores the private IRK in M. When S restarts, it retrieves the key from M during SRTM-proof-period.
3. The IRK is a TPM key on T. The key usage password (called authdata in TCG terminology) for IRK is stored on the TPM and is retrieved by S during SRTM-proof-period.
4. As Scheme 3, but the authdata for the IRK is instead stored on M, and retrieved during SRTM-proof-period.
5. The private IRK is stored by M. The authdata for the IRK is stored by T, and retrieved during SRTM-proof-period.

Note that in order to release the private IRK (as Schemes 2 and 5) or the authdata (as Scheme 4) in the SRTM-proof-period, we require M to be available in this period. One possible way to achieve this is that M is represented by a trusted device, such as a smart card or a secure USB memory drive. This device can communicate with the TPM, request the current PCR values and verify whether it is now in the SRTM-proof-period.

This yields 30 cases, since the first letter can be M, T, S; the second number can be 1, 2, 3, 4, 5; the final letter can be M, T. Due to the page limited, we explain one case only and leave the other cases to the full paper.

Case M1T. M creates the IRK, and securely transmits it to S during the SRTM-proof-period. In order to guarantee the transmission happens in the SRTM-proof-period, M asks T to create an encryption and decryption key pair locked to the SRTM-proof-PCR. M then encrypts the private IRK with the TPM public encryption key and makes it available for S. S asks T to decrypt the private IRK, which can only be done in the SRTM-proof-period, since the decryption key is locked to the SRTM-proof-PCR. After it obtains the private IRK, S extends the PCR to stop anyone else to get the key. Note that after the platform is switched off, S no longer holds the private IRK. So the key must be decrypted during every boot process. The IRK is certified under the TPM AIK during the SRTM-proof-period; to achieve this, S asks T to make a quote with the public part of the IRK as an external input. The result of the quote function is a signature under the AIK of the SRTM-proof-PCR and the public IRK.

Please note that to create and certify the IRK, the trusted external key management entity M does not have to be available on-line in the SRTM-proof-period. The TPM public encryption key can be required in advance. Then, the IRK can be created, certified and encrypted off-line.

6 Two integrity management services

In this section, we introduce two integrity management services, namely *Basic Integrity Management (BIM)* and *Hierarchical Integrity Management (HIM)*. For reasons of limited space, we only give a high level explanation for each service and leave details of their architecture, implementation and application examples in the full version of this paper.

6.1 Basic integrity management

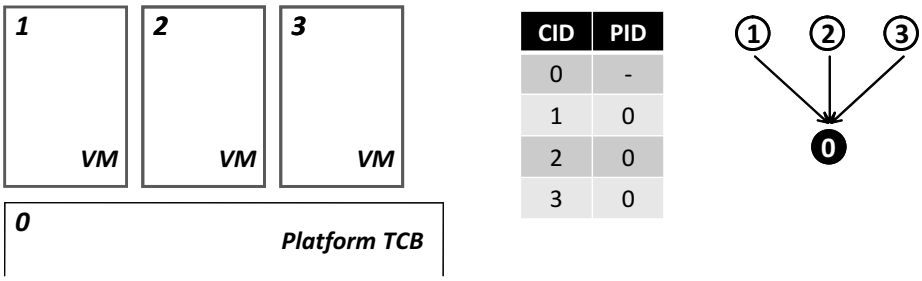


Fig. 2. Simple integrity use case – a flat hierarchy.

The basic integrity service stores static integrity measurements of VM components that are arranged in a flat hierarchy, such as the one shown in Figure 2. 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 main difference is that while PCRs are shared among all the components of a platform, each component has its own CCR. The number of CCRs is unbounded. The measurements are held together in a global CCR table. The BIM service has the following two features: static measurements and simple trust dependency.

Static measurements. The BIM service mimics TPM measurement capabilities but stores integrity measurements in software rather than hardware. Each registered VM 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. VM components use this operation to report ongoing measurements when their contents change. While components are free to use their CCR, they ideally should have a policy that guarantee the integrity of the chain of trust. In particular, as in the TCG chain of trust, the component should extend its CCR before performing an operation that could have an impact on its future behavior. This usualy translates into reporting any piece of software that will be executed by

the component in its own execution environment priori to executing it. Beyond this immediate requierement, a component is free to use its CCR to report other type of information that it would consider sensitive. For example, a kernel would report the loading of device driver and configuration of critical access control lists affecting the kernel's security. The specifics of when or 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 component that does the initial measurement. In the BIM service, this can only be the SRTM in a static (platform TCB) component.

Simple trust dependency. The BIM service implements a flat hierarchy to capture the integrity dependencies between platform components. In this model, the integrity of VM components solely depends on the integrity of the underlying platform TCB. We show an example flat hierarchy in Figure 2. The components labeled *one*, *two*, and *three* are VMs running directly on the trusted platform. Component *zero* is the platform TCB that includes the SRTM. 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. However, the VMs are independent of one another and therefore do not have a trust dependency.

In what follows, we depict the integrity relationships between components using a dependency graph, and represent it using a dependency table. Figure 2 shows a simple graph and its dependency table equivalent. 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 as well. However, the reverse is not true. For example, the integrity of the child component *one* (VM_1) depends on the integrity of the parent component *zero* (TCB) but not VM_2 or VM_3.

The flat hierarchy arises, because a VM component can only be started by a trusted component. Since the TCB is static and platform-wide, it is not possible for a VM component to start – and hence become a parent of – another VM component. Therefore the BIM cannot manage, for example, the integrity of an application started within a VM. The BIM serves as a basis to build the hierarchical model which addresses this limitation.

6.2 Hierarchical integrity management

To overcome the shortcomings of the BIM service, the HIM service has the following two new features: dynamic measurements and hierarchical trust dependency.

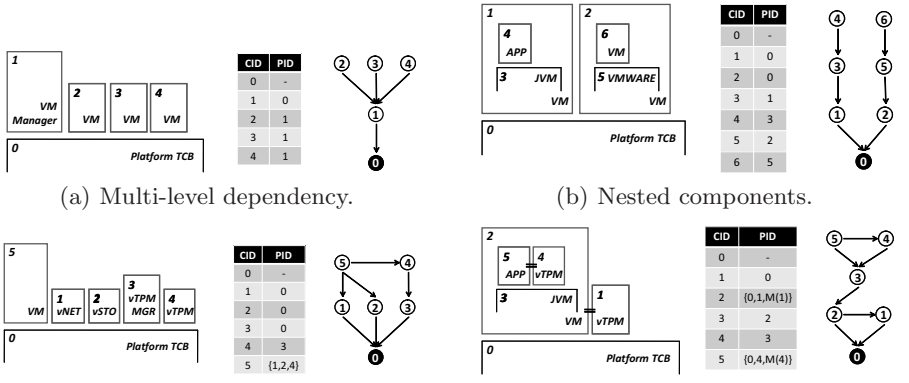
Dynamic measurements. HIM allows multiple registers and resettable registers to be assigned to a single component. Such a component is referred to as a *dynamic component* ("dynamic" because its integrity state may change multiple

times and return to a previous state, i.e one CCR might revert to its previous value). This increases flexibility and allows a component to revert back to a trustworthy configuration if permitted by its change policy. We propose two types of component changes, namely *irreversible changes* and *reversible changes*.

An irreversible change is stored, computed and used the same way as for the BIM . Such a change is suitable for 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 a device driver as the driver may make a change to kernel memory that will persist even after it is unloaded.

A reversible change allows a component to report a previous integrity without having to be reinitialized. Such a change is suitable for 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. As an example, consider a use case with VM_A, VM_B and a Firewall VM (VM_{FW}), where VM_{FW} filters traffic for VM_B. Under the unexpected rule set R_α for VM_{FW}, VM_A does not trust VM_B regardless of its state. Under the expected rule set R_β, however, VM_A trusts VM_B only if its current measured state is as expected. Note that VM_{FW} can alternate between rules R_α and R_β without requiring restart.

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) Multi-level dependency.

(b) Nested components.

(c) Disaggregated services.

(d) Virtual TPM binding.

Fig. 3. Hierarchical integrity use cases.

Hierarchical trust dependency. The hierarchical model can capture any integrity dependency between platform components using a dependency graph. We represent this relation between components as a directed acyclic graph, as shown in Figure 3. To illustrate these more complex trust relationships, consider the following use cases.

Figure 3(a) shows a multi-level dependency. Component *one* is a service that manages the life-cycle of components *two*, *three*, and *four*. All components are VMs. 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 3(b), we see a nested dependency relationship. Components *one* and *two* are VMs, which themselves contain further VMs: component *three*, which is a Java VM, and component *five*, which is a VMware hypervisor. These nested VMs 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 VM 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.

Figures 3(c) and 3(d) illustrate more complicated use cases. In Figure 3(c), we see a multiple dependency relationship. Component *five* is a VM that uses services from components *one*, *two*, and *four*. These components are small VMs 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*).

Figure 3(d) 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.

When a new component is loaded, it is the responsibility of its parent component to set up its dependencies correctly before allowing the new component to execute. Failure to do so is incorrect behavior on the part of the parent. This is analogous to the standard TCG case in which a parent is required to measure a new component before transferring control to it.

At a later time, additional dependencies may also be added which were not known to the parent at load time. Care must be taken with this functionality as it provides a potential avenue for a denial-of-service attack. If a malicious party can add a spurious dependency on a component that he controls, then by altering that component he can stop the dependee from passing an integrity check. One solution is to only allow a component and the components that it already depends upon to modify its dependencies.

7 Related work

Berger *et al.* [1] 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 their solution, 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 dynamic components, and can enhance virtual TPMs by providing a binding between them and the platform TCB through the use of the SRTM.

The basic approach of extending the chain of trust rooted in the TPM using a software-based measurement component like the SRTM has also been described in [12] and is the origin of the L4 implementation of our framework. We extended the original idea with a novel key management method and support for graph-based dependencies among measured components that can also be dynamic. In addition to [1, 12], Gasmi et al. [8] also describes a two-level attestation scheme that distinguishes between static and dynamic configurations.

Several systems have been previously described that use virtual machine monitors to isolate trusted and untrusted components. Terra [7] 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 VM 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. Another similar system is the Microsoft's proprietary proposed Next-Generation Secure Computing Base (NGSCB [6]), partitioning a platform into two parts running over a VMM: an untrusted, unmodified legacy operating system, and a trusted, high-assurance kernel called a *nexus*. Our work, in comparison to both the above models, introduces two practical concepts in preserving integrity: dynamic measurement of each component and maintaining a graph-based hierarchical trust dependency between them. Our solution performs fine-grained policy-based integrity checks on components with less overhead, rather than an integrity check on the entire software stack, which is bundled with its own operating system, and requiring frequent third-party attestation.

Recent Intel and AMD processors support *dynamic root of trust for measurement*, which allows a root of trust for measurement of code to be established after an insecure boot. To launch such code, software in CPU protection ring 0 (e.g., kernel-level code) invokes the SENTER instruction on Intel (or SKINIT instruction on AMD). As part of the SENTER/SKINIT instruction, the processor first causes the TPM to reset the values of the dynamic PCRs 17-23 to zero, and then transmits the secure code to the TPM so that it can be measured (hashed) and extended into PCR 17. This architecture can be used for late launch of a security kernel or virtual machine monitor, and in particular, the SRTM. The SRTM and its SRTM-proof-period is still required for the secure retrieval of the SRTM key. We also mention that the reversible CCRs we consider are more flexible than dynamic PCRs, since the latter can be arbitrarily reset, while the former allow finer-grained reversal policies.

Sailer *et al.*'s implementation of a TCG-based integrity measurement architecture [21] was one of the earliest works to demonstrate the use of a TPM to verify the integrity of a system software stack. In [11], 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.* [19] 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 solution 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 [9], 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) [2], which shares hardware resources among virtual workloads while providing isolation with a mandatory access control policy enforced by the sHype security architecture [20].

8 Conclusions

In this paper, we have introduced a novel integrity management solution that improves on the integrity measurement and reporting capabilities of present

Trusted Computing solutions. In essence, our solution implements a special VM component SRTM that provides a secure link to the CRTM. Our main contributions are a key management method, by which a verifier can have confidence in the key used by the SRTM, and two integrity management services, which are able to cope with proliferation of measured components and dependencies between them as well as dynamic changes to platform components. Details of the service architectures, their Xen implementation and application examples will be given in the full version of this paper.

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. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. P. England, B. Lampson, J. Manferdelli, and B. Willman. A trusted open platform. *Computer*, 36(7):55–62, 2003.
7. 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.
8. Y. Gasmi, A. Sadeghi, P. Stewin, M. Unger, and N. Asokan. Beyond secure channels. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing (SAC'07)*, pages 30–40. ACM, 2007.
9. 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.

10. 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.
11. B. Jansen, H. V. Ramasamy, and M. Schunter. Policy enforcement and compliance proofs for Xen virtual machines.
12. B. Kauer. Authenticated Booting on L4. `http://os.inf.tu-dresden.de/papers_ps/kauer-beleg.pdf`, 2004.
13. B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of the 16th USENIX Security Symposium*. USENIX Association, 2007.
14. The Fiasco micro-kernel, 2004. Available from `http://os.inf.tu-dresden.de/fiasco/`.
15. 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.
16. Microsoft. Bitlocker drive encryption. `http://www.microsoft.com/windows/windows-vista/features/bitlocker.aspx`.
17. 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.
18. Qumranet. KVM: Kernel-based virtualization driver. `http://kvm.qumranet.com`, 2006.
19. 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.
20. 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.
21. 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.
22. 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.
23. 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.
24. 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.