Justifying Integrity Using a Virtual Machine Verifier

Abstract

Emerging distributing computing architectures, such as grid and cloud computing, depend on the high integrity execution of each system in the computation. While integrity measurement enables systems to generate proofs of their integrity to remote parties, we find that current integrity measurement approaches are insufficient to prove runtime integrity for systems in these architectures. Integrity measurement approaches that are flexible enough have an incomplete view of runtime integrity, possibly leading to false integrity claims, and approaches that provide comprehensive integrity do so only for computing environments that are too restrictive. In this paper, we propose an architecture for building comprehensive runtime integrity proofs for general purpose systems in distributed computing architectures. In this architecture, we strive for classical integrity, using an approximation of the Clark-Wilson integrity model as our target. Key to building such integrity proofs is a carefully crafted host system whose long-term integrity can be justified easily using current techniques and a new component, called a *VM verifier*, that can enforce our integrity target on VMs comprehensively. We have built a prototype based on the Xen virtual machine system for SELinux VMs, and find distributed compilation can be implemented, providing accurate proofs of our integrity target with less than 4% overhead.

1 Introduction

With the emergence of a variety of distributed computing architectures, such as grid computing [3], cloud computing [1], and stream processing [13], many applications are moving from centralized to distributed architectures, even architectures involving multiple administrative interests. Cloud computing, in particular, has gained significant mindshare in the past year. Cloud architectures provide a simple interface to configure and execute complex, large-scale applications for a modest fee without committing capital to resources or administration.

In such distributed architectures, it is difficult for either the computing architecture (e.g., cloud) provider or the customer to determine whether the entire distributed computation was executed in a high integrity manner on all systems. Concerns may arise because one of the systems may use outdated, vulnerable code, may have misconfigured policies or services, or may have been previously attacked with a hidden rootkit, triggered by malicious code or data [6]. As both the base system and guest VMs use conventional systems support, such as Xen and Linux, these integrity problems may occur in either the computing architecture or the customer's use of that architecture. While architecture providers assert their diligence in providing a protected environment, such as military grade physical protection of the EC2 cloud data centers and efforts to improve Xen security [4], we aim for a principled approach for proving high integrity distributed computations.

While our ultimate goal is a single approach for generating proofs of integrity for a complete distributed computation, we find that current approaches for proving integrity for a single system are insufficient for these architectures. Integrity measurement mechanisms build proofs of system integrity, but to date such proofs either enable verification of partial integrity for general purpose systems [29, 22, 26, 15, 32, 31] or enable verification of comprehensive integrity for limited (e.g., single application) systems [33, 24, 25]. Integrity measurement approaches that only justify incomplete integrity can give a false sense of integrity when used on general purpose systems, unless they used very carefully, so we aim for comprehensive integrity. Previous approaches to prove VM system integrity [23] used incomplete integrity [29], so the verifier had to make assumptions about the integrity of dynamic data on the base system and guest VM data (e.g., system-specific configurations and application data) that may lead to vulnerabilities. Integrity measurement approaches that require custom systems are not practical for these emerging, distributed computing architectures, as their limitations (e.g., a single application and complete system restarts) are not compatible with these architectures.

In this paper, we design and implement a comprehensive integrity measurement approach for general purpose systems. The design of our approach is based on three insights. First, we leverage emerging work on practical integrity models [30, 18, 35] that define comprehensive and practical integrity model for general purpose systems. Our approach generates proofs that a remote party can use to determine whether a virtual machine's execution adheres to an approximation of Clark-Wilson integrity [30]. Second, comprehensive integrity measurement [24, 33] requires secure initialization (i.e., secure boot [5]), as well as runtime integrity. We leverage a secure initialization approach that binds boot-time integrity to the installation of the system [34]. Third, we find that measuring comprehensive integrity for general purpose systems results in a large number of integrity-relevant events that must be captured and justified to the remote party. Instead of recording all events and requiring the remote party to deduce their impact, we leverage enforcement of comprehensive integrity to remove the need to convey these events. By enforcing an integrity property, we only need to prove to the remote party that the VM was initialized securely and that this integrity property is enforced comprehensively at runtime.

In this work, we make the following contributions:

- We define an integrity measurement approach that starts from well-defined origins (e.g., installation) and enforces a comprehensive view of system integrity to generate practical integrity proofs for VM-based distributed compute nodes.
- We demonstrate a two-level integrity measurement approach whereby a fixed base system, whose integrity can be easily verified, enforces a well-defined integrity property on application VMs, removing the need to verify the application VM details in favor of verifying the base system's ability to enforce a particular integrity property.
- We deploy our integrity measurement approach on application VM systems that perform distributed compilations (`distcc`) and web serving (Apache) to evaluate our approach. For Apache, we show how to deploy an application VM such that CW-Lite integrity [30] can be comprehensively enforced. For `distcc`, we show that the overall performance of distributed compilation using our integrity measurement architecture is increased by less than 4%.

In Section 2, we describe an example scenario and review our goals for its integrity. In Section 3, we examine current integrity measurement techniques and identify the need for comprehensive statements of system initialization and enforcement to justify its integrity. In Section 4, we describe the architecture of our integrity measurement approach. In Section 5, we detail its implementation on Xen systems running Linux application VMs. We perform an evaluation of the security and performance of the implementation on `distcc` and Apache application VMs in Section 6. In Section 7, we conclude.

2 Problem Definition

2.1 Scenario

To motivate this problem, consider the scenario in Figure 1. Canonical's Personal Package Archives (PPA) service compiles developer-supplied source code into Debian packages for all Ubuntu-supported platforms. This enables developers to concentrate on the development of source code while PPA does the work of building and hosting binary packages for the Ubuntu user community.

The specific architecture of the PPA build system is not public, but we know that they use a Xen-based virtual machine systems to host compiler nodes. Using this, we envision the PPA system architecture as shown in Figure 1. Here, the PPA Launchpad frontend receives source packages from developers. As PPA may receive many packages from many developers and needs to support several platforms for such compilation, we envision a set of distributed compilations implemented by one or more `distcc` distributed compilers. The PPA frontend initiates a `distcc` compilation through a `distcc` controller, which retrieves the package to be compiled from a package archive where

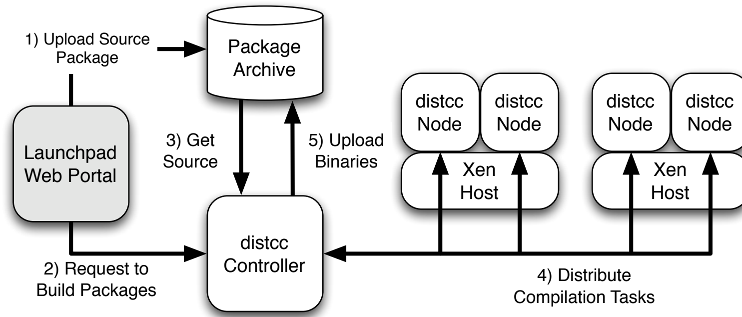


Figure 1: Canonical's Personal Package Archives (PPA) provides compilation services for developers to build binary packages for the variety of Ubuntu platforms. We envision such a service as a distributed compilation, whereby a web portal uploads packages and processes compilation requests using a distributed compilation system, `distcc`.

it was uploaded previously. The controller distributes the job to multiple nodes that perform the actual compilation. Note that `distcc` can also be configured to enable nodes to offload work to subnodes. The `distcc` controller will ultimately return the compiled code back to the package archive, where PPA can then post the binary packages for users to download.

Clearly, Canonical, as well as the developers and Ubuntu user community depend on the integrity of the build process, but this build process presents several challenges. First, the portal and the archive are permitted to interact with untrusted parties (users and developers), but the build process depends on them preserving their integrity. Second, compiler nodes may delegate their work to other systems without the knowledge of the portal. We need to ensure that all systems that are depended upon for high integrity operations are verified as high integrity. Third, we do not know whether the components are configured correctly. Such configuration goes beyond code verification to access control policies, firewalls, access to archive storage, and configurations.

A prior proposal for the Shamon system [23] enabled the construction of integrity-verifiable distributed computations. However, the Shamon architecture imposes two restrictions that prevent the challenges above from being addressed. First, a centralized node verifies the integrity of all compute nodes. In our architecture, no single node may see all the systems. Second, the VMs in a Shamon system are isolated from all untrusted systems. Clearly, that is not practical in an environment where developers provide the input (source packages), and users download the output (binary packages). being from high integrity sources. Third, the initial integrity of a VM is determined by the centralized component, so it cannot support VMs whose integrity is defined elsewhere. In general, the main flaw in the Shamon approach is that it does not consider the possibility that the application VMs may need to be trusted to make integrity decisions. Instead of providing a high integrity platform for running VMs whose runtime integrity can be evaluated, Shamon dictates tightly defined VM integrity and isolates those VMs from untrusted components. This is too restrictive for an open system, such as PPA, so we propose a more general approach that assesses VM integrity in an open environment.

2.2 Integrity

We claim that to demonstrate that a distributed computation is performed correctly, each system (VM and base) that participates in the computation must satisfy the runtime integrity requirements for that computation. Historically, runtime integrity has been assessed using integrity models. The first comprehensive integrity models were proposed by Biba [8]. In Biba integrity, subjects and objects are assigned integrity labels based on their initial integrity state, and these labels are arranged in an integrity lattice where information can only flow from higher integrity entities to lower integrity entities. For example, a subject can only read objects higher (or equal) in the integrity lattice and write objects lower (or equal) in the integrity lattice. The initial integrity state of the system must be ensured, which the Clark-Wilson integrity model [10] makes explicit by defining an *integrity verification procedure* (IVP), a process

to verify the integrity of the system at initialization time to guarantee a high integrity starting point.

Runtime integrity is necessary to ensure that all the inputs that the compilation nodes depend upon, such as the code `distcc` runs, its configurations, the source code it compiles, and compilation results from others, are sufficiently protected from untrusted entities. From a `distcc` node's perspective, it is important that the host system and `distcc` application must be derived from high integrity origins. In addition, the inputs to the node, source code and compilation results from others must also be from high integrity nodes. Finally, any `distcc` node must be able to protect itself from low integrity inputs that it may receive. Because these nodes are connected to the network, there is no guarantee that all network messages will be from other `distcc` nodes. If each `distcc` node protects its runtime integrity effectively, including verification that it only obtains inputs from other nodes that protect their runtime integrity, then it may be possible to build a proof for the PPA server that such compilations are done correctly. In this paper, we will examine what each node needs to do to prove its own runtime integrity for commercial applications, such as `distcc`.

A major challenge in proving runtime integrity is that high integrity systems may receive untrusted inputs (e.g., from the network). The Clark-Wilson integrity model [10] previously identified this problem, and stated that high integrity programs (called *transformation procedures* in the Clark-Wilson model) must be able to immediately discard or update untrusted inputs. However, formal assurance of the correctness of these programs is necessary to justify such behavior in the Clark-Wilson model. While formal assurance for programs appeared to be emerging as a viable technology in 1987 (i.e., when the Clark-Wilson model was proposed), it is now widely believed that formal assurance of complete applications is impractical, so new interpretations of integrity have recently been proposed [30, 18, 35]. These interpretations require that high integrity programs are designed to only accept untrusted inputs at interfaces where integrity decisions can be enforced. This enables designers to focus on just the code where integrity-relevant events occur, so full formal assurance is not necessary. We will leverage one such model, called CW-Lite [30], in providing practical runtime integrity, where CW-Lite only permits untrusted inputs to high integrity programs through *filtering interfaces* identified by the program.

3 Related Work in Integrity Measurement

We claim that current integrity measurement approaches are not sufficient to build accurate proofs of runtime integrity for `distcc` nodes and other commercial applications. Our goal is to use our assessment of integrity measurement below to identify a set of requirements for a runtime integrity measurement architecture.

We identify three key tasks in building a proof of runtime integrity: (1) initialization; (2) enforcement; and (3) measurement. While current integrity measurement approaches support these tasks in some form, they are either incomplete (i.e., do not satisfy runtime integrity) or demand limitations on the system deployment (i.e., are not practical for commercial systems).

We start at the end with the *measurement* task. Several integrity measurement approaches [26, 29, 22] only perform measurement of a system's operations that impact integrity, called *authenticated boot*. The idea is to measure every event that impacts the integrity of the system, which we call *integrity-relevant events*, and send a proof of these events to a remote party who determines whether these events result in a high integrity system or not. These measurement-only approaches are insufficient because they do not measure all integrity-relevant events. They measure the loading of executable code, but none of these systems place requirements on data or external sources. As a result, a `distcc` compiler code may be correct, but it may use incorrect configurations or libraries or depend on untrusted nodes for code or inputs. Despite not measuring every integrity-relevant event, such approaches generate a large number of low-level measurements (e.g., code hashes) that a remote party must interpret to determine the integrity of the system. If a `distcc` node may be used for many platforms, it may be difficult for a remote party to determine whether these measurement really imply a high integrity system.

Next, some integrity measurement approaches perform a degree of integrity *enforcement* in addition to measurement [15, 32, 31]. For example, the Satem [32] system only allows the system to execute approved code objects. BIND [31] only uses a high integrity input if it was generated from operations verified to be high integrity. There

are two advantages to having enforcement in addition to measurement. First, it reduces the ad hoc proofs generated by measurements alone, so the remote party has a only needs to verify that the system follows some integrity guidelines. In Satem proofs, a remote party only needs to verify that the Satem system enforces code execution completely and that it uses input from an authority that the party trusts. Second, enforcement can reduce the number of integrity-relevant events that need to be included in a proof. PRIMA [15] enforces a mandatory access control (MAC) policy that aims to protect processes with high integrity labels. As a result, we don't need to measure code whose integrity is not guaranteed, we just have to show that such code cannot impact the high integrity processes.

Unfortunately, these enforcement approaches are limited in two ways: (1) enforcement is incomplete and (2) enforcement comes too late. First, each of these systems, Satem, BIND, and PRIMA, demonstrate a useful integrity enforcement, but we need all three of these enforcement mechanisms and more if we are to cover all integrity-relevant events. The remote party will need to verify that the code executed is acceptable (Satem), that the resulting process is protected from low integrity processes (PRIMA), that high integrity inputs are truly from high integrity sources (BIND), and that low integrity inputs are really handled according to Clark-Wilson requirements (none). Second, all of these approaches depend on a high integrity initialization (*secure boot*) of the system, which provides the function of a Clark-Wilson IVP. As a system could be modified in prior boot cycles, ensuring that the initialization of a system satisfies integrity requirements is challenging.

A few integrity measurement approaches build a proof of integrity from a secure boot. Outbound Authentication checks each layer of the system before loading the subsequent layer [33]. Flicker uses specialized hardware support to load a program from a secure state [24]. Both Flicker and Outbound Authentication build proofs using integrity measurement hardware justifying the integrity of the secure boot process. However, both run only one application in a tightly prescribed environment. For example, a Flicker computation takes a formatted executable, input parameters, and predefined stack region, and runs that executable on those inputs without support from any OS or services outside the Flicker environment. In both cases, the number of integrity-relevant events that can occur while a computation is running are minimized. No integrity-relevant events are permitted in a Flicker computation, and outbound authentication supports software updates only. While both approaches enable the execution of individual applications [12, 25], neither is intended to support integrity measurement of a `distcc` node, which runs several processes, uses conventional OS services, and depends on the integrity of remote nodes. The Bear approach aims for secure booting of general purpose systems [21], although the basis for initial integrity is ad hoc (e.g., as in measurement-only systems) and enforcement of all integrity-relevant events is incomplete.

3.1 Our Goal

Our goal is to define an integrity measurement approach that provides: (1) effective initialization to define a high integrity system foundation; (2) comprehensive enforcement to make integrity guarantees predictable to remote parties and reduce the number of integrity-relevant events that must measured; and (3) measurement of integrity-relevant events and enforcement properties necessary to justify the integrity of distributed computing, such as the `distcc` system. We are motivated to tackle this challenge using prior work on developing an initialization mechanism, called a *root-of-trust-installation* [34] (ROTI). The ROTI binds the integrity of a privileged VM to its installer. The ROTI enables a secure boot of the privileged VM to yield a proof that a particular installer generated this system, and that the system protects its integrity at runtime. If a remote party trusts the installer, then it can trust the execution of the privileged VM. We use the ROTI approach to justify the integrity of the base system, but we cannot use it for application (`distcc`) VMs, as it requires that the system's persistent state is largely static. We aim for a way to take advantage of the a ROTI-justified base system to enforce integrity on application VMs, enabling a proof of a comprehensive integrity property, in this case CW-Lite.

4 Architecture

In this section, we present our architecture for enforcing CW-Lite integrity on Application VMs and reporting their integrity to remote verifiers. Figure 2 shows an overview of our architecture in which a *host system*, a host system

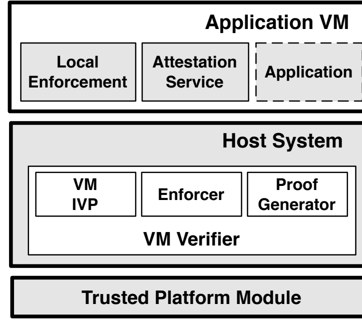


Figure 2: The overview of the VMV architecture. The host system's integrity is proven using an IVP satisfying installation. The VM Verifier contains services to establish, enforce, and report the integrity of an application VM. The local enforcement mediates integrity-relevant events and the attestation service handles VM attestation requests.

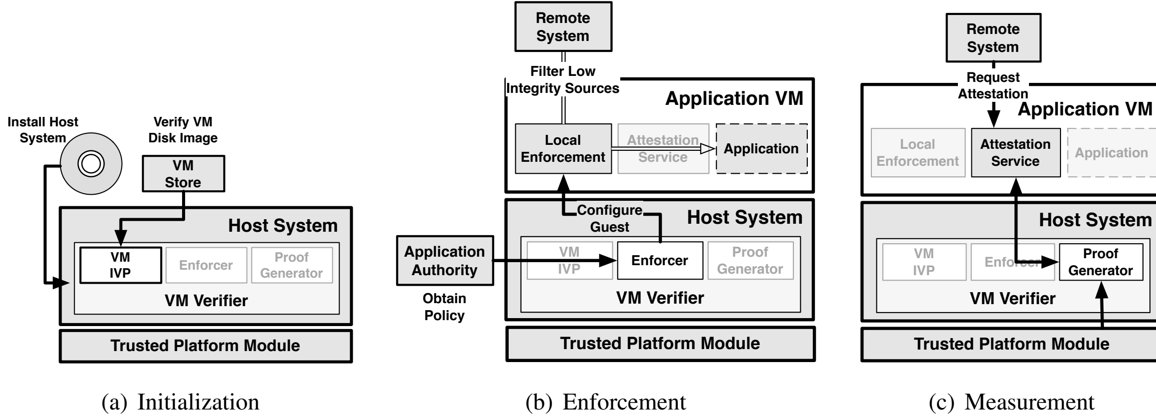
whose integrity can be verified using an extension of the ROTI approach [34], enforces an integrity property over the execution of *application VMs*. Such enforcement is implemented and measured by a new component called the *VM Verifier*. The VM Verifier consists of three parts: (1) a *VM IVP*, which performs VM initialization by verifying that VM's integrity prior to loading; (2) an *enforcer*, which provides integrity enforcement by ensuring that VMs run under the control of an integrity property; and (3) a *proof generator*, which uses measurements to produce attestations of the VM's integrity for remote verifiers. In this architecture, an attestation consists of a measurement of the host system, using traditional integrity measurement approaches, measurement of the integrity property, and any additional measurements necessary for the enforcer to show the mechanisms by which it enforces that property.

Application VMs are extended by two services in our architecture: (1) *local enforcement* may optionally be uploaded into the VM by the enforcer to supplement the enforcement of integrity policies and (2) an *attestation service* that assists applications in obtaining proofs from the proof generator. We note that local enforcement may be trusted by the VMV enforcer to achieve the integrity property. In this case, the enforcer must justify the integrity of local enforcement and its ability to protect the integrity of the Application VM. The attestation service need not be trusted, as all integrity proof statements are signed by the proof generator and TPM.

4.1 Initializing the Host System and Application VMs

Initialization defines mechanisms for securely booting the Host System and Application VM necessary to ensure the initial integrity of trusted components. Figure 3(a) demonstrates the initialization process of the Host System, which is derived from the root-of-trust-installation (ROTI) approach [34]. First, a trusted installer (e.g., on a CD-ROM) installs and configures a high integrity host system, including our *VM Verifier*. The resultant system is bound to the installer via a ROTI Proof. This proof contains the hash of the installed system and the identity of the installer, binding the system origin to the installer. The proof is cryptographically bound to the machine's TPM using a key certified by the TPM's private endorsement key. We state the IVP proof as $\mathcal{I}_{AIK^-}$, where AIK^- is the private Attestation Identity Key produced by the TPM. When the host system is booted, a host IVP process measures the installed system into the TPM, so it can be verified by a remote party in the same manner as a code load.

Once the host has been initialized, it must ensure that the VM is also initialized in a similar high integrity manner. The process of starting a VM occurs when a signed request by an application authority is received by the VMV. In Figure 3(a), the VMV downloads a VM's disk image from a VM store server to the VM IVP. The VM IVP using the verification requirements in the signed request to validate the integrity of the VM downloaded. Such requirements could include attestations of the last run of this VM. This is analogous to the BIND approach [31] where each use of an input requires an integrity verification, but the VMV approach is stronger, as no VM is used without validation of requirements from a trusted authority.



(a) Initialization (b) Enforcement (c) Measurement

Figure 3: (a) demonstrates the installation of the host system verification of VMs before execution. (b) The application policy is obtained and used by the enforcer to configure the VM. (c) Attestation requests are forwarded by the attestation service to the proof generator, which builds an attestation and returns it to the requester.

4.2 Enforcing Integrity over Application VMs

During the lifetime of the VM, all integrity-relevant operations must be mediated by the VM reference monitor. This requires mediation of both local and remote events. In our architecture, we assume each application VM has an accompanying application policy $\mathcal{P}_{App}$, which defines the integrity policy that is to be enforced on the VM. In the case of CW-Lite integrity, an application policy specifies the trusted subject labels that define the application VM's trusted computing base, the code that can run in those subject labels, including the code with filtering interfaces for low integrity inputs.

After a VM's integrity has been initialized as in Section 4.1, the enforcer obtains the VM's policy from its application authority. Figure 3(b) shows this process. Once downloaded, the enforcer installs the attestation service and local enforcement into the VM and configures them according to the policy. In our implementation (Section 5), we employ an integrity enforcing kernel that mediates code loading and an integrity policy in the VM. Certain operations are difficult to enforce externally to the VM, such as the enforcing access control, so we allow the VMV enforcer to configure such local enforcement in the VM. Of course, the integrity of such enforcement must be carefully configured and justified to remote parties. Our architecture also supports the use of integrity monitoring approaches external to the VM. This enables techniques such as VM introspection [19] from the host or kernel integrity monitors [20] to be additionally used to ensure CW-Lite integrity (or future integrity models).

During the execution of the application VM, all network interfaces are mediated so that integrity-relevant services are required to establish IPsec tunnels with high integrity sources. This ensures only verified sources can affect these services. Figure 3(b) illustrates how a connection is setup. Before a connection is established, local enforcement mediates the request to ensure that all high integrity network communications are justified by an attestation of the remote node. If the remote node is also concerned with the local system's integrity, it may request a mutual attestation of the VM.

4.3 Measuring Integrity of Application Execution

We now turn to how the Application VM proves its integrity to a remote verifier. To do this, the VM must produce a proof that demonstrates (a) the host system running the VM is able to enforce an application policy on the VM and (b) such a VM is actually running on that host system. To prove (a), the host system must be provably booted into a known good state, which protects itself from degrading in integrity. Additionally, the host must demonstrate how it enforces the integrity property on the VM. Proving (b) requires associating the running VM with the attesting host system's physical identity (e.g. the TPM's signing key) via an identity key assigned to the VM.

4.3.1 Generating the Attestation

Figure 3(c) shows the steps in generating the VM's integrity proof. A remote system first initiates the attestation process by sending a nonce N to the Application VM's attestation service. This nonce is then forwarded to the host system's VMV. Next, the proof generator builds an integrity proof for the host system, $\Phi_{Host}(N)$. We represent the proof as:

$$\Phi_{Host}(N) = \text{QUOTE}(P, N)_{AIK_\alpha^-} + \mathcal{M} + \mathcal{I}_{AIK^-} \quad (1)$$

Here, $\text{QUOTE}(P, N)_{AIK_\alpha^-}$ is the result of the TPM's quote operation, which provides a signature of the measured state of the host system using the private private key of a TPM AIK_α^-. The measurement list $\mathcal{M}$ contains a list of the measurements of the integrity-relevant operations in the host, including all code loaded on the host, used to compute the hash chain value P. Also included is the result of the host system's IVP proof $\mathcal{I}_{AIK_\beta^-}$, which compared to the file system hash recorded in the TPM by the IVP process. Next, the generator creates the proof for the Application VM, $\Phi_{VM}(N)$ as follows:

$$\Phi_{VM}(N) = K_{VM}^+ + \mathcal{P}_{App} + \Phi_{Host} + \text{SIG}(K_{VM}^+ || \mathcal{P}_{App} || \Phi_{Host}(N))_{AIK^-} \quad (2)$$

When the VM is first configured by the enforcer, a fresh IPsec key pair K_{VM} is generated for the VM, which is used to uniquely identify the VM's identity (i.e., the VM identity key). This key pair is placed in the VM's local key store. This identity is tied to the VM proof by a signature of $\Phi_{Host}(N)$ concatenated with the VM's public key and an identifier for the policy $\mathcal{P}_{A}pp$ that the VMV is enforcing on the VM. This proof, $\Phi_{VM}(N)$ ties the integrity of the host system and the enforced integrity policy of the VM to the VM's public key using the host system's authoritative private key. Next, $\Phi_{VM}(N)$ is passed back to the VM's attestation service, which sends $\Phi_{VM}(N) + K_{VM}^+$ to the remote verifier.

4.3.2 Verifying the VM's Integrity

When the remote verifier receives the VM's attestation, it first validates the signatures in the proofs for correctness. It must then determine the host system's integrity. To do this, it inspects $\Phi_{Host}(N)$ by first checking that the quote QUOTE is correct by reconstructing the PCRs P with $\mathcal{M}$ and verifying both the signature and N are correct. This check ensures the reported measurement list came from the machine associated with the AIK. The verifier then assesses $\mathcal{I}_{AIK^-}$ to confirm that the host system's IVP process measured the same installed system as in the IVP proof and then compares the entries in $\mathcal{M}$ against a set of known-trusted measurements. We envision that an authority over the VM will provide the information necessary for obtaining these measurements. This check confirms (a) the host booted into a trusted state and that (b) only code from that installation was loaded. Thus, a remote verifier is able to ascertain the integrity of the host system using traditional integrity measurement techniques.

The verifier then compares $\mathcal{P}_{App}$ against the policy it expects the Application VM to use. Given such a policy, the host configures the VM to support a policy $\mathcal{P}_{VM}$. This resultant policy represents the combined changes the enforcer makes to the VM to support $\mathcal{P}_{App}$. Since the installer is trusted to generate a host that is able to enforce an application policy on the VM, a successful validation of $\Phi_{Host}(N)$ implies $\mathcal{P}_{VM}$ satisfies $\mathcal{P}_{App}$. Moreover, the host system has claimed key K_{VM}^+ speaks for a VM with the integrity specified by $\Phi_{VM}(N)$. Using this key, the verifier can establish an IPsec tunnel with the application VM that cryptographically ties both the VM's identity and integrity to the tunnel.

5 Implementation

We now describe our implementation of our architecture. We first detail the host system and its VMV components to enforce the application specific integrity policy on VMs. Next we explain how the VM is modified by the VMV to mediate both CW-Lite integrity locally in the VM and inputs from remote systems. Finally we show how a remote verifier obtains and validates the integrity of the reporting system.

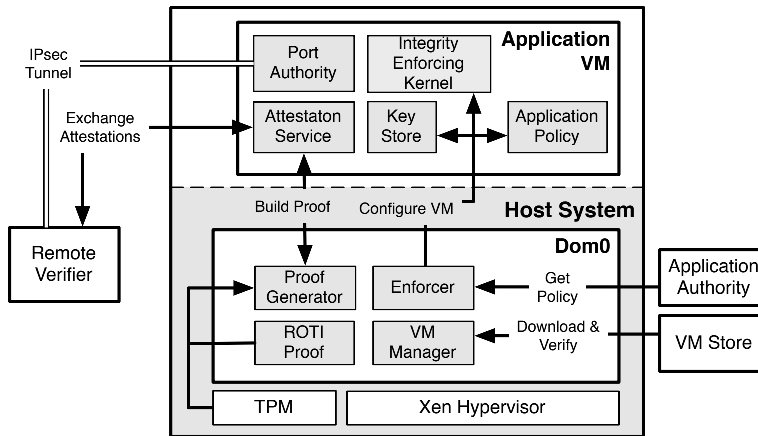


Figure 4: Application VMs run on a Xen host system. The *VM manager* obtains and validates VM disk images. The enforcer manages the VM's runtime integrity according to an application policy. This is done within the VM by mediating integrity relevant events locally with an integrity enforcing kernel and remotely from network inputs using the *port authority*. Finally, the attestation service works with the proof generator to build an attestation of the host system and vouch for the VM's integrity as well as its ownership of an IPsec key stored in the VM's key store.

5.1 Overview

Figure 4 shows the implementation of our prototype design. The host system is implemented as a Xen VM system with the host VM domain (Dom0) containing the VMV. The VMV is comprised of several services including the *VM manager*, enforcer, and attestation service. The VM Manager handles application VM disk images and verifying their integrity. The Enforcer installs the programs and policy files in the VM disk image to enforce CW-Lite integrity. It also supports the mediating VM integrity from outside the VM using integrity monitoring techniques [20, 19]. Finally, the VMV's proof generator produces attestation of the host system using the TPM's quote and signing operations and vouches for the integrity of the application VMs running on the host. Within the application VM, the enforcer installs the *port authority*, which verifies the integrity of remote systems before establishing IPsec connections. It also sets up the VM's attestation service to forward requests from remote parties to the host system's attestation service. We now discuss how these components are implemented and how they meet the goals of our architecture.

5.2 Initializing the Host System and Application VMs

In this section, we describe how the host system is installed and the VM's integrity is initially verified using the VM manager.

Installing the Host System One of the goals when building the host system was simple integrity verification and the ability to justify the integrity of its VMs. To that end, we designed an installer based on the ROTI approach [34]. After the early boot phase (CRTM, Firmware, Bootloader) is measured and extended to the TPM's PCRs, the bootloader measures the installer. The installer then proceeds to configure the system and measures the resulting filesystem. The TPM then performs a quote operation using a hash of the current time and a signing key bound to the TPM. This resulting quote contains the measurements of the system's state at the time of the installation and a measurement of the installed filesystem. When the system is booted, a script in the initial ram disk hashes the filesystem and stores this measurement in the TPM. This measurement can be compared by a remote verifier against the ROTI proof to determine if the host system was booted into a state produced by a trusted installer.

Host System Our host system consists of a Xen Hypervisor 3.2 with the Flask / Xen Security Module (XSM) [11] built-in. XSM is used for VM mandatory access control and restricts which devices and VMs are accessible by each domain. The host VM (Dom0) contains a stripped down Linux distribution using the 2.6.24 Linux kernel. Since the LSM framework does not support stacking of multiple security modules, we modified SELinux to perform the same functions as IMA [29]. However, current work to provide a dedicated framework for integrity measurement modules is in development, which would obviate the need for this modification [2]. In order to protect Dom0 from modification, the installer removes all interactive user accounts and configures the filesystem to be mounted read-only. This required moving normally mutable files such as `/etc/mtab` to an in memory temporary file system. Further details on which files needed to be moved can be found in the ROTI paper. Additionally, a strict SELinux reference policy is used to limit access to kernel interfaces like `/dev` and `/proc` and administrative programs.

Managing Application VMs The VMV consists of the VM manager, enforcer, and proof generator services. The VM manager receives requests from an application authority to start and stop VMs and validates the request using certificates obtained at install time. Once a request is received, the VM image is downloaded from its VM store, which we implement as a simple web service. The VM manager then checks the VM disk integrity. Included with the image is an integrity proof generated by the previous host system to run the VM. The VMV checks this proof using the verification technique for remote attestations described later in Section 5.4. By ensuring the previous host was able to enforce the application policy's CW-Lite properties, the host is able to build *transitive trust* of all previous runs of the VM. This concept is similar to BIND [31], but we additionally verify that the host system was initialized to a high integrity state before each run.

5.3 Enforcing Integrity over Application VMs

Our implementation enforces a CW-Lite runtime property on VMs as specified by an application policy. This policy defines a TCB using SELinux subject types and a set of trusted code. We now look at how the VMV's enforcer is implemented to use this policy to achieve CW-Lite integrity in the VM.

The VMV's enforcer is invoked by the VM Manager to configure the VM and create its domain for execution. First, the application policy, a signed Debian package, is retrieved from the application authority web service. The VM image is mounted locally and the package installs the policy files, attestation service, and port authority components. We describe the contents of the policy package further in Section 6.1. Next, the enforcer configures the VM domain's RAM, vCPU, networking configurations, and XSM security label. Since VMs other than Dom0 do not have access to physical platforms, it is critical to link VM's identity to a physical platform like the EK stored in the TPM [7]. To solve this issue, the enforcer generates an IPsec key pair and places a copy of it in the VM and VMV's key store. The VMV vouches for this key by generating a fresh certificate for each attestation of the VM. Finally, the hypervisor is invoked to create and start the VM domain.

The enforcer runs the VM with a Linux kernel using a modified SELinux LSM that performs a validation check on all code loaded into trusted process. This allows the enforcement of an application policy that specifies both the permissible code and process that must be protected. This is similar to PRIMA [15], but additionally enforces high integrity code loading instead of just measuring it. At boot time, the enforcer supplies an initrd that loads two policy files from the policy package via the sysfs files `\selinux\ts_load` and `\sys\kernel\security\enforce`. The first policy file defines the SELinux subject types that represent the TCB. The second is a list of acceptable code hashes. These hashes are defined by the distribution and vendors that application VM's uses. Whenever a kernel module is loaded, a process memory maps a file (such as a library) as executable, or an `execve` call triggers the `bprm_set_security` LSM hook[1], the validation procedure is called.

Figure 5 shows a flowchart of our validation procedure. The kernel first examines whether the current process' label is one of the policy specified trusted subjects. If it is not, no further examination is performed since the process is not part of the TCB. Otherwise, the measurement list is checked to see if the file had been previously measured. If

[1] This is the first point during an `execve` when a process begins to transition to a new subject type

it was not, the code is hashed and compared to the policy database of trusted code hashes. A successful match will cause the hash of the file to be placed in the measurement list. Otherwise, the operation that triggered the validation check is denied. If the file had been previously measured, it must have been in the acceptable hash list. Hence, the triggering operation is permitted to continue as normal.

Applications frequently depend on inputs from remote sources. In order to protect those applications and other trusted network facing services from low integrity sources, our architecture defines that an Input Verifier mediate all network communication. We implement this verifier as the port authority, a service the VMV installs in every VM. This service configures the VM's `iptables` to default deny all non-IPsec traffic except for sources defined by the application policy. When a connection to or from a remote party is attempted, the port authority requests an attestation of the remote party. This attestation is verified using the application policy's verification criteria (see Section 5.4). If the remote party is deemed high integrity, the other party may optionally request a mutual attestation. Along with the attestation is a certificate of the attesting system's IPsec key. Once both sides have agreed to permit the connection, an IPsec security association is created and an IPsec tunnel is established using this key.

The IPsec tunnel permits the connection to pass the `iptables` firewall and identifies the inputs as coming from a high integrity source. Additionally, the Port Authority periodically requests fresh attestations from the remote party. If the integrity proof fails to pass the verification criteria, the IPsec tunnel is torn down, and the security association is removed. Additionally, aggressive dead peer detection is used to discover if a peer has lost connection. The connection is also torn down when this occurs to prevent the possibility of the remote party rebooting into a low integrity system and using the negotiated IPsec session keys.

5.4 Measuring Integrity of Application Execution

Figure 4 illustrates the process of a remote party requesting an attestation from the application VM. First a nonce is sent to the application VM, initiating the procedure. When the VM's attestation service receives the request, the nonce is forwarded over an internal network socket to the VMV's proof generator, which uses it to obtain a TPM quote. The proof generator then creates a fresh certificate for the VM's public IPsec key using a hash of the key, the nonce, and the application policy package. This certificate is signed using the private portion of the TPM's AIK. Finally, TPM quote, the certificate, and the ROTI proof are forwarded to the remote party via the VM's attestation service. Alternatively, the VM manager may also request an attestation of the VM when it is shutdown. In this scenario, the manager uses a hash of the VM disk image as a nonce to the proof generator. The resulting attestation binds the current state of the VM (after shutdown) to the host's integrity.

To validate an attestation, the requester first obtains the public AIK of the host's TPM. There are various methods for implementing this including the use of a trusted third party or direct anonymous attestation [9], but it is outside the scope of this design. Next, the attestation is checked for syntactical correctness including signatures (quote, certificate, ROTI proof), the value of the nonce, and that the PCRs match the measurement list. Next, the ROTI proof's installer hash is check against the set of known good installers. If this passes, the file system hash the quote must match the one in the ROTI proof. Success means the host booted into a trusted state. Next, the measurement list is checked for illegal measurements. This ensures only files from the installer's distribution were loaded in the host. This good measurement set is defined by a verification criteria obtained by the verifier . Further discussion of how this is defined is in Section 6.1. A successful validation assures the verifier that the host system is enforcing an acceptable integrity policy on the VM associated with the attested IPsec key. This process also allows the host to revoke a VM's key by no longer providing a valid certificate when requested.

6 Evaluation

6.1 Security Evaluation

We now evaluate how our design achieves the two goals of enforcing runtime integrity in a VM and the correct reporting of that enforcement. We first examine the security of our implementation host system in resisting com-

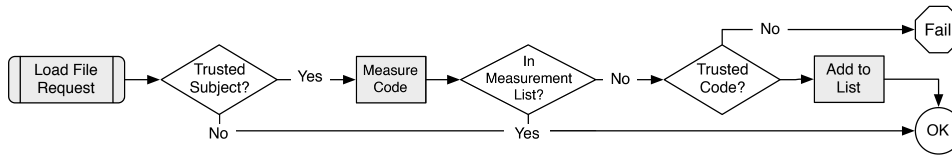


Figure 5: Process flow of the PRIMA enforcing kernel's code enforcement procedure.

promise. We then use an Apache webserver application VM to study how the VMV enforces a proof of concept integrity policy on the VM. We then analyze the approach's ability to report the VM's integrity to remote parties.

6.1.1 Evaluation of Enforcement

Proper enforcement of VM runtime integrity policy requires the following: (1) a high integrity installation of the host system; (2) a the host system and its enforcing components to function with high integrity and to be protected from compromise.

Installation To configure the host system, we first create an installer that is trusted to configure a high integrity host system. In our implementation, we built an installer using a modified debian-installer with an ISOLINUX bootloader. This image is burned to a CD and run at boot time. One threat to the installation is the presence of malicious firmware that could interpose itself during the configuration. One possible solution is to modify the ISOLINUX bootloader to use the special late launch CPU function. Both AMD's `skinit` and Intel's `senter` implementation securely clears the processor pipeline and cache enabling the installer to be first measured and executed free from the effects of previously executed code. While we have not implemented this, approaches like OSLO [17] have done this with BIOS bootloaders that work with ISOLINUX.

host System To protect the Dom0's integrity during runtime, we must protect the installed files from modification and ensure that all runtime state depends only on installed code and data. To protect the installation, the installed filesystem is mounted read-only with all mutable files stored in a ram-based filesystem. This prevents runtime modification and creation of files from persisting across reboots. We also removed all interactive user accounts and login abilities to prevent user administration that may result in modification of the installed system. To protect the runtime state, we use a strict SELinux MAC policy to limit access to kernel and device interfaces and files (e.g. `/sys` or `/dev`). Network access to Dom0 is limited to a few services, including the VM IVP and the proof generator, for which we provide filtering interfaces. While Xen has been shown to be relatively secure compared to other hypervisors [28], a multitude of vulnerabilities have been found [36]. Approaches such as privilege separating Xen domains [27] and the use of SMM based integrity scanners [20] offer ways of mitigating these threats.

Constructing an Application Policy Package Protection of Application VM integrity depends on the design of the application policy. In our system, we aim for CW-Lite integrity [30], so an application policy defines the application-specific inputs necessary to justify CW-Lite integrity: (1) the set of high integrity code for the Application VM; (2) the set of trusted subject labels in the VM's mandatory access control (MAC) policy; and (3) the criteria to assess remote systems. At present, `distcc` does not have an SELinux policy module necessary for building a complete application policy package. So, to evaluate runtime integrity enforcement on the VM, we created a proof of concept application policy package for an Apache webserver VM. Apache is a significantly more complex system than `distcc`, so we expect that the same approach will apply once an SELinux policy module is available.

First, the policy file containing a list of acceptable code hashes for the Application VM, was obtained by parsing all packages in the Ubuntu 8.10 repository for executable files, libraries and kernel modules. The resulting set contained 34,000 hashes resulting in about 648KB. The policy package also includes the strict SELinux policy and Apache policy module.

Operation	Computational Time (seconds)
Attestation Exchange	1.087 ± 0.010
Generate IPsec Configuration File	0.00027 ± 0.000022
Generate IPsec Connection Specification	0.417 ± 0.020
IPsec SA Setup	0.588 ± 0.026

Table 1: Port authority micro-benchmarks showing time required for building an integrity verified IPsec tunnel.

Second, the trusted subjects file identifies the MAC policy subject labels that must load only trusted code. SELinux uses type enforcement to label process so we identified a set of subject types that represent the target system TCB. We used PALMS [14] to find this set. PALMS is a tool written in XSB Prolog [37] that verifies the integrity of a TCB by querying an SELinux policy with an initial TCB set. The queries search for the types that create flows into the TCB types. Once identified, the policy author decides whether to add those types to the TCB or trust the receiving TCB type must filter the low integrity input. For our analysis, we used the strict SELinux reference policy and the Apache policy module. We then seeded the PALMS tool with a set of proposed TCB types [16] and repeatedly queried the tool until no new types were reported by the previous query. However, there are some subjects, such as `initrc_t`, that require input from subjects that are not in the TCB. In those cases, we assume that such subjects will have filtering interfaces to sanitize potentially low integrity input [30].

Third, policy is necessary define the criteria used to verify the integrity of remote systems that provide input. Our criteria contains the hashes of the trusted installer image and application policy packages. In addition, the acceptable code and data measurements that a trusted host system would have are included. This means all of the load time code and policy measurements that would appear in Dom0. Since a trusted installer places the same files in every system it installs, these measurements would be included with the installer image.

VM Integrity Within the application VM, both local and remote dependencies must be mediated to assure high integrity computation. In our Apache application VM for example, Apache depends on not only the host system, but the guest OS, supporting programs, and any external data such as a database with web content. In our design, the host installs a reference monitor in the VM comprised of the integrity enforcing kernel and the Port Authority. The kernel uses SELinux to enforce the application policy installed by the VMV's enforcer. The trusted processes that form the TCB are all mediated so that only trusted code is loaded and that low integrity flows only enter these processes through types that the policy assumes have filtering interfaces. Remote dependencies, like the database, are verified by the port authority. As a result, only verified network sources can connect to trusted services via a combination of `iptables` and IPsec rules specified by the application policy's verification criteria.

6.2 Performance Evaluation

We evaluated the overhead introduced by our implementation on the operation of a normal application VM system. Specifically, we focused on overhead from initializing the host, starting a VM, local enforcement, and finally from mediating network connections. We setup a testbed of 3 Dell PowerEdge M605 blades with 8-core 2.3GHz Dual Quad-core AMD Opteron processors, 16.0GB RAM, and 2x73GB SAS Drives (RAID1). These systems were connected over a private Gigabit Ethernet switch on a quiescent network using IPsec in ESP mode. Each of these blades formed a host system using a Xen 3.2 hypervisor and Ubuntu 8.10 as the Dom0 host. To test the performance of application VMs on our system, we constructed a `distcc` VM image designed to be deployed on our cluster with minimal configuration. Each VM consisted of a 362MB Ubuntu 8.10 installation, 1GB of RAM, a single vCPU and a Xen 2.6.24 Linux kernel modified with our PRIMA enforcing LSM module.

Initialization At boot-time, the IVP script in the initrd hashes the filesystem before root rotates to the primary partition. This procedure uses SHA1 over 630MB installed filesystem followed by an extend operation to store the measurement in PCR 13 of the TPM. The speed of SHA1 in userspace (tested with `openssl`) is normally 270MB/s

Task	Time (mins.)		% diff.	Lines of Code
	Unattested	Attested		
Linux Kernel	7:01.84	7:17.24	3.65%	6,450,761
OpenSSH	0:22.67	0:23.74	3.98%	68,626

Table 2: 6 node `distcc` cluster compilation time with and without reattestations every 30 seconds.

on our machines, but the total IVP time takes approximately one minute. We found the added delay was due to the kernel's empty page cache at boot slowed the hashing process. One possible improvement would be to incrementally measure the filesystem as the files are requested instead of measuring everything at once. This approach would require a more time consuming verification since the aggregate filesystem hash might be nondeterministic and thus requiring inspection of a full measurement list of files.

Starting a VM involves verifying the disk image, the policy package, configuring the VM and finally creating the Xen domain. For our VMV verification policy, verifying VM and policy package integrity requires checking for a valid signature on both. The combined time to verify our compressed 137MB `distcc` VM image and 1.5MB policy package took about 1 second to perform. A more sophisticated verification such as validating the VM integrity proof will increase the time accordingly. The average time to mount the VM disk, install programs and configuring the enforcement policies added approximately 1.35 seconds to the Xen domain creation procedure.

Enforcement The VM's PRIMA enforcing kernel introduces an initial delay to the program execution due to hashing of the code and looking up the result in the policy hash table. Since the average size of an executable was 39KB (with the largest being 6MB) and in-kernel SHA1 is 135MB/s on our setup, code hashing added a negligible time. The PRIMA module adds on average 12.85 ms with negligible variance to the execution time after the program is hashed. This means the initial delay to hash a program is relatively short.

Measurement Before a connection is established with the VM, the port authority must create an IPsec tunnel. Table 1 shows the individual steps in creating that connection. Unsurprisingly, the majority of the 1.53 seconds required to setup the IPsec connection come from the attestation, which is due to the slow speed of the TPM.

Application Performance We used our setup to compile two different code-bases: the Linux 2.6.28 kernel and `openssh`. We gathered macro benchmarks of the compilation time with and without the port authority to determine the effect of verifying remote host integrity, setting up IPsec connections, and polling remote hosts on the compilation time. During the run with port authority enabled, all connections required mutual attestation, and a repeated attestation every thirty seconds. As shown in Table 2, the overhead introduced by our system was minimal with less than 4% increase in compilation time. After profiling the build process we found the majority of the overhead was due to IPsec on network throughput. We further experimented with different encryption suites and null encryption, but found similar times. These results indicate that our system has no significant impact on application performance beyond that introduced by IPsec.

7 Conclusion

In this paper, we proposed an architecture for proving comprehensive runtime integrity for application VMs. The architecture was based on a carefully crafted host system whose long-term integrity can be justified easily using current techniques and a new component, called a *VM verifier*, that can enforce our integrity target on VMs comprehensively. We built a prototype VM Verifier for SELinux VMs, and find that integrity-verified distributed compilation can be implemented with less than 4% overhead.

References

[1] Amazon Elastic Compute Cloud (EC2). Available at `http://aws.amazon.com/ec2`.

[2] integrity: Linux Integrity Module(LIM). Available at `http://lwn.net/Articles/287790/`.

[3] G. Alliance. The Globus Toolkit. `http://www.globus.org/toolkit/`.

[4] Amazon. Amazon Web Services Security. White paper. `http://s3.amazonaws.com/aws_blog/AWS_Security_Whitepaper_2008_09.pdf`, September 2008.

[5] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. In *In Proceedings 1997 IEEE Symposium on Security and Privacy*, pages 65–71. IEEE Computer Society, 1997.

[6] A. Baliga, P. Kamat, and L. Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data (Short Paper). In *2007 IEEE Symposium on Security and Privacy*, May 2007.

[7] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2006. USENIX Association.

[8] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, MITRE, April 1977.

[9] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, New York, NY, USA, 2004. ACM.

[10] D. D. Clark and D. Wilson. A Comparison of Military and Commercial Security Policies. In *1987 IEEE Symposium on Security and Privacy*, May 1987.

[11] G. Coker. Xen Security Modules (XSM). `http://www.xen.org/files/xensummit_4/xsm-summit-041707_Coker.pdf`.

[12] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *Computer*, 34(10):57–66, 2001.

[13] B. Gedik, H. Andrade, K.-L. Wu, P. Yu, and M. Doo. SPADE: The System S Declarative Stream Processing Engine. In *Proceedings of the 2008 ACM SIGMOD Conference*, June 2008.

[14] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A Logical Specification and Analysis for SELinux MLS Policy. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 91–100, New York, NY, USA, 2007. ACM.

[15] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: Policy-Reduced Integrity Measurement Architecture. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2006.

[16] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

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

[18] N. Li, Z. Mao, and H. Chen. Usable Mandatory Integrity Protection For Operating Systems. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, May 2007.

[19] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proc. 17th Usenix Security Symposium*, San Jose, CA, July 2008.

[20] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux Kernel Integrity Measurement Using Contextual Inspection. In *STC '07: Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, pages 21–29, New York, NY, USA, 2007. ACM.

[21] J. Marchesini, S. Smith, O. Wild, and R. MacDonald. Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Technical Report Computer Science Technical Report TR2003-476, Dartmouth College, 2003.

[22] H. Maruyama, F. Seliger, N. Nagaratnam, T. Ebringer, S. Munetoh, S. Yoshihama, and T. Nakamura. Trusted platform on demand. In *IBM Technical Report RT0564*, 2004.

[23] J. M. McCune, T. Jaeger, S. Berger, R. Caceres, and R. Sailer. Shamon: A System for Distributed Mandatory Access Control. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 23–32, Washington, DC, USA, 2006. IEEE Computer Society.

[24] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, New York, NY, USA, 2008. ACM.

[25] J. M. McCune, A. Perrig, and M. K. Reiter. Safe Passage for Passwords and Other Sensitive Data. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, Feb. 2009.

[26] Microsoft Corporation. Next generation secure computing base. http://www.microsoft.com/resources/ngscb/, May 2005.

[27] D. G. Murray, G. Milos, and S. Hand. Improving xen security through disaggregation. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 151–160, New York, NY, USA, 2008. ACM.

[28] T. Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. http://taviso.decsystem.org/virtsec.pdf.

[29] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, USA, Aug. 2004.

[30] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium*, February 2006.

[31] E. Shi, A. Perrig, and L. van Doorn. BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 154–168, Washington, DC, USA, 2005. IEEE Computer Society.

[32] S. Shrivastava. Satem: Trusted Service Code Execution across Transactions. In *SRDS '06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 337–338, Washington, DC, USA, 2006. IEEE Computer Society.

[33] S. W. Smith. Outbound Authentication for Programmable Secure Coprocessors. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Oct. 2002.

[34] L. St. Clair, J. Schiffman, T. Jaeger, and P. McDaniel. Establishing and Sustaining System Integrity via Root of Trust Installation. In *Proceedings of the 2007 Annual Computer Security Applications Conference*, Dec. 2007.

[35] W. Sun, R. Sekar, G. Poothia, and T. Karandikar. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 248–262, Washington, DC, USA, 2008. IEEE Computer Society.

[36] R. Wojtczuk. Subverting the Xen hypervisor. www.blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf.

[37] XSB. XSB - Logic Programming and Deductive Database System for Unix and Windows. http://xsb.sourceforge.net/.