Justifying Integrity Using a Virtual Machine Verifier

Joshua Schiffman, Thomas Moyer, Christopher Shal, Trent Jaeger, and Patrick McDaniel
Systems and Internet Infrastructure Security Laboratory
Computer Science and Engineering Department, Pennsylvania State University
University Park, PA 16802
{*jschiffm,tmmoyer,cshal,tjaeger,mcdaniel*}*@cse.psu.edu*

***Abstract*—Emerging distributed 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*, which comprehensively enforces our integrity target on VMs. We have built a prototype based on the Xen virtual machine system for SELinux VMs, and find that distributed compilation can be implemented, providing accurate proofs of our integrity target with less than 4% overhead.**

***Keywords*-cloud computing, integrity measurement, virtual machines**

I. Introduction

With the emergence of a variety of distributed computing architectures, such as cloud computing [1] and stream processing [2], 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, have misconfigured policies or services, or may have been previously attacked with a rootkit, triggered by malicious code or data [3]. 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 [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 [5]–[10] or enable verification of comprehensive integrity for limited (e.g., single application) systems [11]–[13]. Integrity measurement approaches that only justify incomplete integrity can give a false sense of integrity when used on general purpose systems, unless they are used very carefully, so we aim for comprehensive integrity. Previous approaches to prove VM system integrity [14] used incomplete integrity [5], 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 [15]–[17] that define 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 [15]. Second, comprehensive integrity measurement [11], [12] requires secure initialization (i.e., secure boot [18]), as well as runtime integrity. We leverage a secure initialization approach that binds boot-time integrity to the installation of the system [19]. Third, we find that measuring integrity of general purpose systems

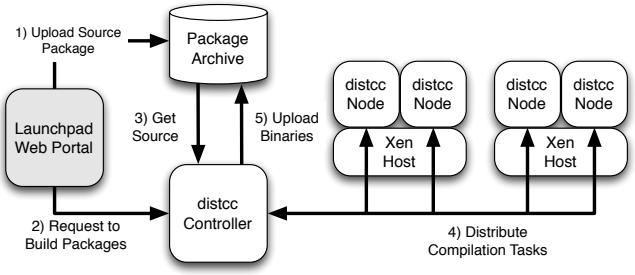


Figure 1. A Personal Package Archives provides compilation services for developers to build binary packages for a variety of platforms. We envision such a service as a distributed compilation cluster (e.g. `distcc`, whereby a web portal uploads packages and processes compilation requests.

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 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 thoroughly 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 computing nodes.
- We demonstrate a two-level integrity measurement approach whereby an easily verified fixed base system 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 comprehensively enforce CW-Lite integrity [15] on an application VM, and demonstrate that only a 4% increase in compilation time results from using our architecutre in a `distcc` cluster.

In Section II, we describe an example scenario and review our goals for its integrity. In Section III, we examine current integrity measurement techniques and identify the need for comprehensive statements of system initialization and enforcement to justify its integrity. In Section IV, we describe the architecture of our integrity measurement approach. In Section V, 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 VI. In Section VII, we conclude.

II. Problem Definition

A. Scenario

As an example of a distributed service, consider Canonical's Personal Package Archives (PPA), which compiles developer-supplied source code into packages for all supported platforms . This enables developers to concentrate on the development of source code while the 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 Xen-based virtual machine systems to host compiler nodes. Thus, 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 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, and configurations.

A prior proposal for the Shamon system [14] enabled the construction of integrity-verifiable distributed computations. However, Shamon 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 Shamon are isolated from all untrusted systems. That is not practical in an environment where developers provide the input (source packages), and users download the output (binary packages). Third, the initial integrity of a VM is determined by a 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 trust application VMs 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.

B. Integrity

We claim demonstration of a distributed computation's correctness requires each participating system (VM and base) to 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 [20]. 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 [21] 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 all inputs 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 be derived from high integrity origins. In addition, the inputs to the node must also be from high integrity nodes. Finally, nodes must be able to protect itself from low integrity inputs. 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 is required of each node to prove runtime integrity for a commercial application like `distcc`.

A major challenge in proving runtime integrity is handling untrusted inputs. The Clark-Wilson integrity model [21] previously identified this problem, and stated that high integrity programs (called *transformation procedures*) must be able to immediately discard or update untrusted inputs. Additionally, formal assurance of the correctness of these programs is necessary to justify such behavior. However, it is now widely believed that formal assurance of applications is impractical. Thus, new interpretations of integrity have been proposed [15]–[17]. These require high integrity programs to only accept untrusted inputs at interfaces where integrity decisions can be enforced thus making full formal assurance unnecessary. We will leverage one such model, called CW-Lite [15], in providing practical runtime integrity, where CW-Lite only permits untrusted inputs to high integrity programs through *filtering interfaces* identified by the program.

III. Related Work in Integrity Measurement

We identify three key tasks in building a runtime integrity proof: (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.

We start with the *measurement* task. Several integrity measurement approaches [5]–[7] only measure system's operations that impact integrity. The idea is to measure every event that impacts the integrity of the system (i.e. *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. These measurement-only approaches are insufficient because they do not place requirements on data or external sources. As a result, a `distcc` compiler code may be correct, but it may use incorrect configurations or depend on untrusted nodes for inputs. Despite not measuring every integrity-relevant event, such approaches generate a large number of low-level measurements that must be parsed for verification. Even worse is the case verifiying heterogeneously constructed `distcc` nodes whose measurements may be different but valid.

Next, some integrity measurement approaches perform a degree of integrity *enforcement* in addition to measurement [8]–[10]. For example, the Satem [9] system only allows the system to execute approved code objects. BIND [10] 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 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 [8] 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 using secure boot. Outbound Authentication checks each layer of the system before loading the subsequent layer [11]. Flicker uses specialized hardware support to load a program from a secure state [12]. Both techniques 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. While both approaches allow execution of individual applications [13], [22], 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 [23], although the basis for initial integrity is ad hoc and enforcement is incomplete.

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* [19] (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.

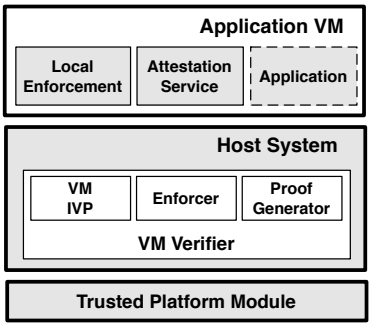


Figure 2. The overview of the VMV architecture. The host system's integrity is proven using an IVP satisfying installation. The VMV contains services to establish, enforce, and report the integrity of an application VM. The local enforcement mediates integrity-relevant events and the attestation service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

IV. 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 an integrity verifiable *host system* (e.g. using a ROTI approach [19]) enforces an integrity property over the execution of *application VMs*. Such enforcement is implemented and measured by a new component called the *VM Verifier* (VMV). The VMV 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 traditional integrity measurement of the host system, measurement of the enforced integrity property, and any additional measurements necessary for the enforcer to show the mechanisms by which it enforces that property.

We extend application VMs with two services: (1) *local enforcement* that 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.

A. Initializing the Host System and Application VMs

Initialization defines mechanisms for securely booting the host and VM necessary to ensure the initial integrity of

trusted components. Figure 3(a) demonstrates the initialization process of the host, which is derived from the root-of-trust-installation (ROTI) approach [19]. First, a trusted installer (e.g., on a CD-ROM) installs and configures a high integrity host system, including our VMV. 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 verify the VM is also initialized in a similar high integrity manner. The process of starting a VM occurs when an application authority sends a signed request to 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 VM's last run. This is analogous BIND [10] 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.

B. 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. We assume each application VM has an accompanying application policy $\mathcal{P}_{App}$, defining the integrity policy to be enforced on the VM. Our CW-Lite integrity policy specifies the trusted subject labels that define the application VM's TCB and the code that can run in those subject labels.

After a VM's integrity has been initialized (Section IV-A), the enforcer obtains the VM's policy from its application authority (Figure 3(b)), installs the attestation service and local enforcement into the VM, and configures them according to the policy. In our implementation (Section V), we employ an integrity enforcing kernel that mediates code loading using an integrity policy in the VM. Certain operations are difficult to enforce externally to the VM, such as 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 externel integrity monitoring approaches such as VM introspection [24] from the host and kernel integrity monitors [25].

While the application VM is running, all network interfaces are mediated high integrity services only recieve input from high integrity sources over IPsec. Before a connection to a remote system is established, local enforcement traps the request and verifies the remote system's integrity is justified by an attestation (Figure 3(b)). Optionally, a mutual attestation of the local VM may be requested by the remote node. Finally, the IPsec connection is established allowing inputs to the system.

C. Measuring Integrity of Application Execution

In order to prove an Application VM's integrity to a remote verifier, 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. (a) requires the host system must be verifiably booted into a known good state that protects its integrity. Additionally, the host must demonstrate 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.

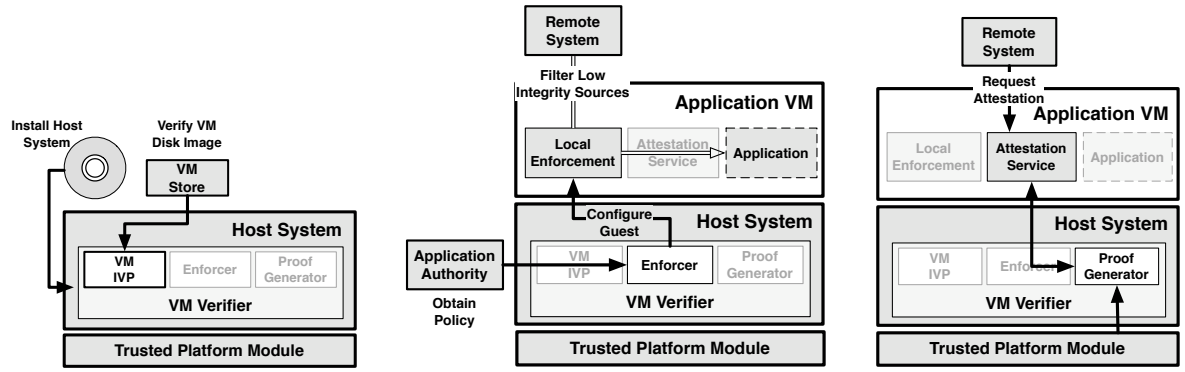
1) Generating the Attestation: Figure 3(c) shows the steps in generating the VM's integrity proof. A remote system first initiates the attestation process by sending a nonce N to the Application VM's attestation service. This nonce is then forwarded to the host system's VMV. Next, the proof generator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

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

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

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

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. 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 $\mathcal{P}_{App}$ 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.



(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.

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 $\mathcal{P}_{VM}$. 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 enforces 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.

V. 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 VMV modifies the VM to mediate both CW-Lite integrity events in the VM and inputs from remote systems. Finally we show how a remote verifier obtains and validates the integrity of the reporting system.

A. 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 [24], [25]. 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.

B. Initializing the Host System and Application VMs

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 used an install based on the ROTI approach [19], which configures the host filesystem and genrates a proof of the system's state and filesystem at the time of the installation. A remote verifier can compare this proof to the runtime measurements of the host to detect any deviations between boots. The resulting host system consists of a Xen Hypervisor 3.2 with MAC enforcement of VMs and resources by the built-in Flask / Xen Security Module (XSM) [26] built-in. Dom0 runs a

stripped down Ubuntu distribution and a 2.6.24 Linux kernel modifed to enable SELinux and IMA to run concurrently. A dedicated integrity measurement framework is currently being added to Linux, which will obviate the need for this modification [27]. Further details on the host's configuration is describe in the ROTI work.

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 V-D. 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 [10], but we additionally verify that the host system was initialized to a high integrity state before each run.

C. 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 uses 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 VI-A. 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 [28]. 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 [8], 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.

Our validation procedure proceeds as follows. 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 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 to high integrity services except for those that can filter low integrity inputs (via 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 V-D). 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 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

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

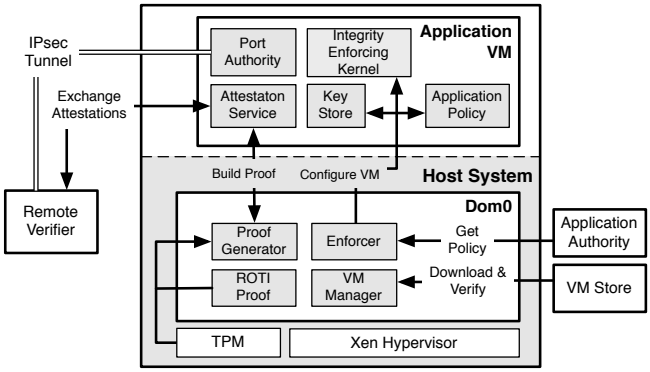


Figure 4. The *VM manager* obtains and validates VM disk images. The enforcer manages the VM's runtime integrity according to an application policy by mediating integrity relevant events in the VM with an integrity enforcing kernel and remotely from network inputs using the *port authority*. The attestation 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.

system and using the negotiated IPsec session keys.

D. 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 [29], 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 VI-A. 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.

VI. Evaluation

A. Security Evaluation

We now evaluate how our design achieves the goals of enforcing and reporting VM runtime integrity. First we examine our proof of concept system's ability to resist compromise. We then study how the VMV enforces an integrity policy on an Apache webserver VM and reports this to remote parties. Proper enforcement of VM runtime integrity requires (1) a high integrity host system and (2) the host system's enforcing components to be protected from compromise. The security of the ROTI configured Xen-based host has been studied in previous work [19], but it is important to note that a multitude of vulnerabilities have been found recently [30]. Approaches such as privilege separating Xen domains [31] and the use of SMM based integrity scanners [25] offer ways of mitigating these threats.

Constructing an Application Policy Package: How the system protects the application VM's integrity is defined by its application policy. In our system, we aim for CW-Lite integrity [15], 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 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, we obtained the list of acceptable code hashes for the Application VM 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. Second, we obtain the trusted subjects that 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 [32] to find this set. Further details of this process can be found in our previous work [33]. Third, our verification criteria for remote systems 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

Operation	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 I
PORT AUTHORITY MICRO-BENCHMARKS SHOWING TIME REQUIRED FOR BUILDING AN INTEGRITY VERIFIED IPSEC TUNNEL.

comprises all load time code and policy measurements that should 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 desgin, 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 into types consdidered high integrity. High integrity types may only receive flows from high integrity sources or from low integrity flows via a filtering interfaces, which immediately discards or upgrades inputs to high integrity. This is similar to DLM's notion of endorsers [34]. Remote inputs, are similarly mediated via the port authority. As a result, only verifiably high integrity network sources can connect to trusted services unless the service exposes a filtering interface that can handle unverified sources.

B. Performance Evaluation

To evaluate the practicality of deploying a VMV system, we constructed a `distcc` cluster. We developed a ROTI installer that configures each host system, a VMV service, and a distcc VM image with an application policy. We then evaluated the overhead introduced by our implementation on compilation time. Specifically, we focused on overhead from initializing the host, starting a VM, local enforcement, and finally from mediating network connections. We use 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 in RAID1. These systems were connected over an isolated Gigabit Ethernet switch on a quiescent network. Each blades formed a host system using Xen 3.2 and Ubuntu 8.10 as Dom0. Each `distcc` node VM consisted of a 362MB Ubuntu 8.10 installation, 1GB RAM, one vCPU and a Xen 2.6.24 Linux kernel modified with our PRIMA enforcing LSM module.

Initialization: At boot-time, the IVP script runs SHA1 on the 630 MB filesystem and logs it in PCR 13 in the TPM before root is rotated to the primary partition. SHA1 in userspace (tested with `openssl`) is normally 270MB/s on our machines, but the total IVP time takes approximately one minute. 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 complicate the verification as the measurements would be added nondeterministicly thus requiring inspection of a full measurement list of files instead of a single aggreagate.

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 I 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 II, 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. Our results indicated the largest performance impact came from processing IPsec traffic. When we experimented with running miltiple compliation tasks concurrently, we found TPM response time increased only when additional hosts were connected to the profiled system.

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 II
6 NODE DISTCC CLUSTER COMPILATION TIME WITH AND WITHOUT REATTESTATIONS EVERY 30 SECONDS.

This is due to the slow clock speed of he TPM, causing a significant queuing effect. However, this has no direct affect on the performacne of the compilation itself, except if the attesation delay becomes large enough to invalidate the link. On possible method of addressing this issue is to use aysnchronous attestation techniques [35].

VII. 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," http://aws.amazon.com.

[2] 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*.

[3] 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.

[4] Amazon, "Amazon Web Services Security," White paper. http://s3.amazonaws.com.

[5] 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.

[6] 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.

[7] Microsoft Corporation, "Next generation secure computing base," http://www.microsoft.com/resources/ngscb/.

[8] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: Policy-Reduced Integrity Measurement Architecture," in *Proceedings of the 11th ACM SACMAT*, 2006.

[9] S. Shrivastava, "Satem: Trusted Service Code Execution across Transactions," in *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, 2006.

[10] E. Shi, A. Perrig, and L. van Doorn, "BIND: A Fine-Grained Attestation Service for Secure Distributed Systems," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, 2005.

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

[12] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: an execution infrastructure for tcb minimization," in *Proceedings of the 3rd ACM SIGOPS/EuroSys 2008*.

[13] J. M. McCune, A. Perrig, and M. K. Reiter, "Safe Passage for Passwords and Other Sensitive Data," in *Proceedings of NDSS*, 2009.

[14] J. M. McCune, T. Jaeger, S. Berger, R. Caceres, and R. Sailer, "Shamon: A System for Distributed Mandatory Access Control," in *Proceedings of the IEEE 22nd ACSAC*, Washington, DC, USA, 2006.

[15] 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.

[16] 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.

[17] W. Sun, R. Sekar, G. Poothia, and T. Karandikar, "Practical Proactive Integrity Preservation: A Basis for Malware Defense," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*.

[18] 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*.

[19] L. St. Clair, J. Schiffman, T. Jaeger, and P. McDaniel, "Establishing and Sustaining System Integrity via Root of Trust Installation," in *Proceedings of the 23rd ACSAC*, 2007.

[20] K. J. Biba, "Integrity Considerations for Secure Computer Systems," MITRE, Tech. Rep. MTR-3153, April 1977.

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

[22] 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*, vol. 34, no. 10, pp. 57–66, 2001.

[23] 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," Dartmouth College, Tech. Rep. TR2003-476, 2003.

[24] 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, Jul. 2008.

[25] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell, "Linux Kernel Integrity Measurement Using Contextual Inspection," in *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, New York, NY, USA.

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

[27] "integrity: Linux Integrity Module." [Online]. Available: http://lwn.net/Articles/287790/

[28] 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*, Berkeley, CA, USA, 2006.

[29] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *Proceedings of the 11th ACM conference on Computer and communications security*, 2004.

[30] R. Wojtczuk, "Subverting the Xen hypervisor." www.blackhat.com/presentations/bh-usa-08/Wojtczuk.

[31] D. G. Murray, G. Milos, and S. Hand, "Improving xen security through disaggregation," in *Proceedings of the 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2008, pp. 151–160.

[32] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel, "A Logical Specification and Analysis for SELinux MLS Policy," in *Proceedings of the 12th ACM SACMAT*, 2007.

[33] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger, "Measuring integrity on mobile phone systems," in *Proceedings of the 13th ACM SACMAT*, 2008.

[34] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, pp. 410–442, 2000.

[35] T. Moyer, K. Butler, J. Schiffman, P. McDaniel, and T. Jaeger, "Scalable Web Content Attestation," in *To appear in Proceedings of the IEEE 25nd ACSAC*, 2009.