Establishing and Sustaining System Integrity via Root of Trust Installation

Luke St.Clair*, Joshua Schiffman, Trent Jaeger, Patrick McDaniel
Systems and Internet Infrastructure Security Laboratory
The Pennsylvania State University, University Park PA 16802

Abstract

Integrity measurements provide a means by which distributed systems can assess the trustability of potentially compromised remote hosts. However, current measurement techniques simply assert the identity of software, but provide no indication of the ongoing status of the system or its data. As a result, a number of significant vulnerabilities can result if the system is not configured and managed carefully. To improve the management of a system's integrity, we propose a Root of Trust Installation *(ROTI) as a foundation for high integrity systems. A ROTI is a trusted system installer that also asserts the integrity of the trusted computing base software and data that it installs to enable straightforward, comprehensive integrity verification for a system. The ROTI addresses a historically limiting problem in integrity measurement: determining what constitutes a trusted system state in a heterogeneous, evolving environment. Using the ROTI, a high integrity system state is defined by its installer, thus enabling a remote party to verify integrity guarantees that approximate classical integrity models (e.g., Biba). In this paper, we examine what is necessary to prove the integrity of the trusted computing base (sCore) of a distributed security architecture, called the Shamon. We describe the design and implementation of our custom ROTI sCore installer and study the costs and effectiveness of binding system integrity to installation in the distributed Shamon. This demonstration shows that strong integrity guarantees can be efficiently achieved in large, diverse environments with limited administrative overhead.*

1 Introduction

Traditional distributed systems are built upon the assumption that the systems have *integrity* (i.e., they have not been compromised). As evidenced by the many serious vulnerabilities exploited in the wild, it is difficult to believe that such an assumption is reasonable. Integrity measurement hardware, such as the Trusted Computing Group's (TCG's) Trusted Platform Module (TPM) [12] provides a mechanism that may be used to generate integrity statements for individual machines. A variety of approaches that leverage the TPM to provide integrity measurement guarantees have been proposed [26, 31, 30, 19, 16, 23]. However, none of these approaches have been accepted as a basis for guaranteeing the integrity of distributed systems in practice.

We argue that integrity measurement is not being accepted in practice because current approaches do not satisfy classical integrity guarantees. In classical integrity models, such as Biba [5], the integrity of a system depends on integrity of all the files it reads and executes. If a process executes a low integrity program or reads data that has been modified by a low integrity subject, then the process must also be low integrity. The TPM-based integrity measurement approaches are effective for measuring well-known, static files, such as program code, but are not effective at measuring system-specific files (e.g., configurations) or dynamic files because the remote party cannot be expected to know the current value of these files. Further, Smith identifies that a high integrity system must also protect its secrets (e.g., private keys) to prevent attackers from masquerading as the system [32]. In TPM-based measurement, sealing[1] is used to ensure that data, including secrets, is only released to approved software configurations. However, once the data is unsealed it may later be accessible to unauthorized subjects through compromised software, misconfigured systems, and uncleared memory.

We identify three types of problems in using prior, TPM-based integrity measurement approaches: (1) untracked modification of system-specific and dynamic data; (2) loss

*This material is based upon work supported by the National Science Foundation under Grant No. CNS-0627551, "CT-IS: Shamon: Systems Approaches for Constructing Distributed Trust". Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[1] To simplify, the TPM *seals* secrets in a storage device by encrypting the source data using a key derived from the TPM-internal secrets and unique fingerprints of the hardware and running software (via PCRs) [12]. Therefore, subsequent *unsealing* is only possible by that same TPM, hardware, and software.

of system secrets to low integrity code; and (3) the lack of control of software loads after verification. First, since system-specific and dynamic data cannot be verified using TPM measurements, incorrect administration and/or compromised programs may modify this data, and integrity measurement will be ignorant of such vulnerabilities. Second, system secrets may be compromised due to the same vulnerabilities as for dynamic data, but they may also be leaked across bootcycles. It is well-known that many system BIOS's do not clear physical memory [8], so a high integrity system may be rebooted into a malconfigured system that retrieves the unsealed secrets from memory. Third, while the TPM enables the verification of the software executed by a system, it does not prevent the execution of low integrity software in the future. Further, the wider the variety of software executed on a system, the more likely a compromise of that system. While we do not want to "lock down" systems to a single, immutable configuration, we would like an approach that enables systems to be installed where the integrity of data as well as software may be tracked until the next installation.

The remainder of this paper focuses on *install*, *boot*, and *runtime* activities flowing from the above requirements. We consider initially how a system can be installed such that it can subsequently present evidence that all relevant software and configuration originated from a known and trusted ROTI (install). Second, we consider how the system is booted such that it guarantees that the system state is driven only by those ROTI-installed system components (boot). Finally, we develop techniques to produce evidence that the system continues to execute within that stable, known state (runtime), e.g., the system runs only the installed software and that the software and secrets are not compromised.

While conceptually simple, it turns out that The ROTI approach requires several other complimentary design and implementation decisions to achieve a high integrity system. For example, once installed from a ROTI, the *sCore* must be protected to prevent later compromise or misconfiguration. Further, it is not obvious that the ROTI approach is flexible enough or performs well enough to support practical systems. Might it be too expensive to trace data integrity to its installation? Might the system require changes that cannot be traced to the ROTI? In this paper, we define a generic set of integrity requirements and demonstrate that the *sCore* software stack be used effectively and integrity-verified based on these requirements. We do *not* claim that the ROTI approach applies to general systems, but where comprehensive integrity guarantees are required, the ROTI approach shows how to specialize systems to enable such guarantees to be achieved and proven to remote parties.

In this paper, we consider how a *root of trust install* can be used to establish and sustain integrity for a distributed security architecture, called a Shard Reference Monitor or *Shamon* system. A *Shamon* consists of a set of trusted computing bases, called *Shamon Core* or *sCore*, one for each physical platform, that jointly enforce a single, mandatory access control policy. In order to build a *Shamon*, it is imperative that each individual *sCore* be high integrity. We explore this objective herein by detailing the requirements of a *Shamon*, including their measurement and construction. We show that our enhanced *sCore* ROTI installation (which should seldom be needed) can be completed in less than 10 minutes on a commodity desktop, of which less than 10% is related to the ROTI-specific functions. Further, we show that the boot time overhead associated with the ROTI integrity verification is nominal ($< 5\%$). As supported by these experiments, our claim is that systems that require strong integrity guarantees, such as the *sCore*, can be practically installed and run based on the ROTI principle.

The rest of the paper is structured as follows. We begin in Section 2 by describing the *Shamon* architecture and the traditional integrity measurement approaches and the security challenges of extending them to a high integrity system indefinitely. This includes a detailed discussion of the attack vectors an adversary may use to exploit a *Shamon* system. From this analysis, we develop in Section 3 a broad design philosophy and outline our working implementation of the *sCore* ROTI in Section 4. In Section 5 we present an assessment of the costs associated with ROTI installation and subsequent integrity measurement. We discuss related work in more detail in Section 6 and conclude in Section 7.

2 Background

The *Shamon* project [18] leverages integrity measurement techniques to enable the combination of high integrity reference monitors on multiple physical machines into a single unit that still satisfies the reference monitor requirements [3]. Previous integrity measurement approaches [26, 31, 30] leave several decisions unspecified, such that it is possible to build a *Shamon* system that can be verified as high integrity when it is in fact under the control of an attacker. Our goal is to develop an approach that ensures that when integrity measurement claims a component is high integrity, it is high integrity relative to the *Shamon* approach. In a *Shamon* system, this means building a high-integrity, verifiable base upon which VMs can run. While the policies governing the creation, execution, and verification of VMs themselves is critical to the security of this system, we defer the discussion of these issues to a future work.

2.1 *Shamon* System

We begin by defining a *Shamon* system, shown in Figure 1. A *Shamon* (i.e., system-wide reference monitor) is a reference monitoring service for distributed applications. As shown, distributed applications consist of sets of virtual

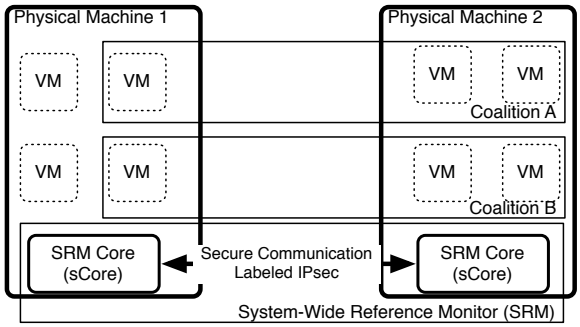

Figure 1. A System-Wide Reference Monitor (*Shamon*) system: Coalitions of virtual machines (VMs) may run across physical machines, but the *sCore* components collaborate to form a *Shamon* that enforces the VM communication requirements within a coalition.

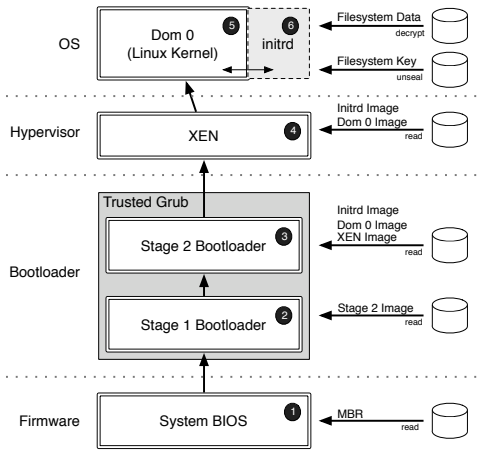

Figure 2. The integrity measurement in the *sCore* VM's boot sequence.

machines (VMs), called *coalitions*, that execute on one or more physical platforms. For example, Alice's work VMs may comprise Coalition A and her gaming VMs may comprise Coalition B. The mapping of coalitions to physical platforms is many-to-many: many coalitions may run on a single platform and, as stated above, a coalition may span multiple physical platforms.

In order for a physical platform to become a member of a *Shamon* system, it must run a high integrity software base, called the *sCore*. The *sCore* provides VM communication primitives for distributed applications and access control over those communications. That is, each *sCore* contains a reference monitor that is capable of enforcing a mandatory access control (MAC) policy over VM communications. A *Shamon* is constructed from multiple *sCore*, so a single, comprehensive MAC policy can be enforced over a set of VMs (i.e., coalition) that comprises a distributed application. For example, different VMs in Alice's coalition may have different permissions. In a gaming coalition, the VM permissions may be determined by user identity and/or their roles in the game. In a work coalition, Alice's different laboratory applications may have different permissions. Each *sCore* justifies its compliance to a common (*Shamon*) MAC policy, and ensures its tamper-resistance using virtual machine isolation and secure, tamper-detectable communication channels (e.g., IPsec). The end result is that *the combination of integrity-verified sCore in a Shamon provides the same function as a single reference monitor, but the Shamon spans multiple physical platforms.*

The main breakthrough that enables the implementation of a *Shamon* system is hardware-supported, integrity measurement. Traditionally, access control is enforced on individual machines with little or no guarantee that other machines are enforcing a compatible policy. If the machines are in the same administrative domain, we may provide a common policy to each, but there is still no guarantee that the machines are really enforcing that policy (i.e., they may be erroneously or maliciously misconfigured). Distributed access control using *trust management* [7, 6, 20, 21] requires that each system develop its own representation of its access control policy, making coherent enforcement impractical. Hardware-supported integrity measurement enables systems to verify the function of others, thus enabling two machines to verify their reference monitor guarantees and join forces to compose a system-wide reference monitor, our *Shamon*.

2.2 *Shamon* Integrity Measurement

We now examine the application of traditional integrity measurement to an individual *sCore*. While a variety of approaches have been proposed [31, 26], we apply the approach that appears most appropriate for our system and software architecture, Trusted Platform on Demand [24] (TPoD) for integrity measurement of the *sCore*'s privileged virtual machine (Dom 0, in Figure 2) and the Linux Integrity Measurement Architecture [30] (IMA) for *sCore* VM's services. Here, we detail integrity measurements, how they are constructed, and the semantics of their integrity guarantees. In the next section, we describe integrity vulnerabilities relative to the *sCore* approach, motivating the need for an approach that provides a more precise justification for integrity.

The TPM [12] is a device that provides a limited computing platform and a small amount of storage that is protected from the host machine. The TPM's limited computing platform supports operations for extending a hash chain (*extend*), signing hash chain values for remote parties to verify (*quote*), encrypting data (*seal*), and decrypting by particular system configurations (determined by the current hash chain values, *unseal*). A TPM hash chain represents

a sequence of files loaded into the system. Some files may be executables and some may be data files (e.g., configuration). The idea is that each software component measures (i.e., performs a TPM *extend*) any software or key file before it loads it. Note that the *authenticated boot* semantics of the TPM [4] means that it only extends measurements, but does not enforce integrity itself. If this resultant sequence includes only high integrity files loaded in an acceptable order, then a remote party can verify the system as high integrity, using the signed hash chain value generated via *quote*. Note that verification depends on the remote party being able to determine the high integrity hash value for each file measured, so current TPM approaches are only used to measure executables and static data files.

As shown in Figure 2, integrity measurement of the *sCore* boot process (up to the Dom 0 Linux kernel) consists of a deterministic sequence of well-defined load operations oulined by TPoD. A caveat to the previous description is that a *core root of trust measurement* (CRTM) is necessary to bootstrap the measurement process by measuring itself and the rest of the BIOS prior to loading the next layer of software, the stage 1 bootloader in the master boot record (MBR) off the primary boot drive (as configured in BIOS settings). This CRTM is stored in a ROM section of the BIOS, and thus is at least partially resistant to tampering, provided the BIOS is implemented correctly. Stage 1 then measures stage 2 prior to loading it, and stage 2 measures the Xen hypervisor and privledged operating system kernel (the domain 0 kernel). Since the Xen hypervisor does not include integrity measurement software, the bootloader (i.e., stage 2) measures the domain 0 kernel and its initrd image, even though the Xen hypervisor loads this kernel. As long as we trust the measured Xen hypervisor to load the kernel specified by the bootloader, this is acceptable.

The *sCore* also includes services running in user-level on the Dom 0 kernel (i.e., in the privileged, Dom 0 Xen VM). To ensure *sCore* integrity, all of these user-level services must be measured, and we use the Linux IMA [15, 30] to do so. IMA enables automatic measurement of all software (e.g., executables, libraries, and kernel modules) and can be used to measure static data files when specified by the software.

For high integrity data whose values may be system-specific or change over time, the remote parties cannot predict the data's values, so their integrity cannot be verified by measurement alone. Integrity measurement approaches use the TPM to encrypt this data (i.e., using TPM *seal*) and decrypt it only when a certain software configuration has been loaded, using TPM *unseal*. For the *sCore*, a *sCore* key would be sealed that would enable decrypting of the *sCore* data. The *sCore* should verify the integrity of the system that sealed the key, but this choice of sealing system is a *sCore* design choice.

2.3 Potential *sCore* Vulnerabilities

In identifying potential vulnerabilities of the above integrity measurement approach for the *sCore*, we first define a threat model. We consider both remote attackers and a limited local attacker. Remote attackers may provide malicious input to the *sCore* to try to inject code or modify dynamic data. Integrity measurement should enable justification that our *sCore* can protect itself from such threats.

We also consider the threat of a local attacker who can control the configuration of the *sCore*, but does not attack the TPM itself. Such a local attacker may be a significant threat because installing software or rebooting an *sCore* is much easier and less conspicuous than a hardware attack on a TPM. Also, we do not address local attacks on the firmware of devices other than the host computer. Others have proposals to address this problem [13].

Using the integrity measurement approach above, an *sCore* may be vulnerable to the following types of threats:

- **Untracked Modification of Data**: Malformed inputs from remote users and misconfigured system due to local users may result in the malicious modification of dynamic data (e.g., system configuration files). For example, `/etc/resolv.conf` contains a list of system-specific DNS servers, so if an attacker could replace these with a list of malicious servers they would compromise system integrity. The value of a DNS server list may not be meaningful to a remote party, so the only viable solution is to seal the data to protect its integrity. Sealing is vulnerable to misconfiguration or malconfiguration by local attackers, high integrity programs compromised by remote attackers, and even low integrity software run by either. In the last case, even when sealing records the low integrity system state, this evidence would be erased by a subsequent sealing using a trusted system. An *sCore* must be able to justify the integrity of its installed data, even if those data's values cannot be predicted in advance.
- **Loss of System Secrets**: Unsealed secrets may be lost by compromised high integrity software, the execution of untrusted software in subsequent bootcycles and various hardware leaks. Even if we control all paths that a remote attacker may use to compromise our software, integrity measurement does not prevent low integrity software from being run that may simply leak the secrets. Further, rebooting the *sCore* presents some problems because the contents of memory may persist across a reboot [8]. For example, not all Intel BIOS's clear memory on reboot, so a local attacker may be able to reboot into a non-*sCore* system that is able to retrieve *sCore* secrets, such as IPsec private keys, from memory. The *sCore* design must ensure that secrets cannot be used by attackers should they be leaked.

- **Integrity after Verification**: After a remote party verifies the integrity of a system, integrity measurement does not guarantee that the integrity is maintained into the future. We identify three potential problems: (1) authenticated boot does not prevent the loading of low integrity software, whereas the *sCore* requires that no low integrity software be loaded after attestation; (2) an insufficiently-managed *sCore* may contain software that does not adequately protect the system from malicious inputs; and (3) a local attacker may be able to reboot into a non-*sCore* system, while maintaining the *Shamon*'s communication channel. First, authenticated boot does not prohibit the execution of low integrity software after verification, so a remote party cannot be sure that a system remains high integrity. Second, if the *sCore* services receive input from any potentially malicious sources, the integrity of the system may be compromised after verification. Finally, if a local attacker can reboot a system fast enough and locate the current IPsec session state, a non-*sCore* system may be able to use an *sCore* communication channel. The *sCore* design must use integrity measurement in a manner that maintains integrity after verification for a running *sCore*, and retracts *sCore* connections when the *sCore* is terminated.

3 Solution Approach

The goal of our solution is to ensure that *sCore* integrity can be traced back to acceptable roots of trust. For integrity measurement, the roots of trust are the TPM itself, whose processing is protected from the host, and the BIOS's *core root of trust measurement* (CTRM), which bootstraps the integrity measurement process. To design a high integrity system, we claim that *all facets of the system must be linked to a root of trust in integrity.* The lack of this facility in current integrity measurement leads to the vulnerabilities detailed above. We claim that one additional root of trust is necessary (Section 3.1), outline the key design tasks for constructing high integrity *sCore* (Section 3.2), and show how this design will justify *sCore* integrity (Section 3.3).

3.1 Core Root of Trust Installation

We claim that it is important to leverage the installation process itself in establishing and maintaining system integrity. We define trust in the installation as a *root of trust installation* (ROTI). A ROTI is an installer system provided by a system distributor. When a system is installed, the ROTI loads all software and configures all system-specific data. All software, system-specific data, and secrets can be traced to the ROTI. Further, the set of *sCore* software is limited to restrict the amount of dynamic data and the ways that it can be modified. The result is that ROTI-based, integrity measurement can prove that the *sCore* software and data originate from the ROTI, such that a remote party can verify the integrity of a system only having to trust the CRTM, TPM, and ROTI. Integrity measurement reverts to proving association of *sCore* software and data with these entities.

In this section, we show how integrity measurement is justified by this design. However, the main challenge in this paper is to show that the ROTI is a practical way to justify integrity. First, system distributors already provide system installations as a unit, even with signed files, so the practical foundation of verifiable installations is present. Second, the ROTI is a well-defined installer system provided with such installations, so the remote party can verify system-specific data are provided by a particular ROTI via sealing by that ROTI. Third, as detailed in Section 5, the cost of installation and verification based on the ROTI are modest. Since the *sCore* is designed to be a reliable trusted computing base, it should not be modified frequently, it should not require arbitrary system administration and system changes after installation that could introduce uncertainty into its integrity. Our claim is that the ROTI limits the flexibility of system configuration in ways that are reasonable for trusted software and fundamental to achieving system integrity.

3.2 *sCore* Design

The *sCore* design includes three key tasks to enable integrity measurement to link the system to roots of trust. The specific function and implementation of these tasks are described in Section 4.

Installation: A ROTI installs a *sCore* system. The TPM builds statements that a remote party uses to verify that all software and system-specific configuration data is tracable to the ROTI installation.

Booting: The booting system uses statements generated at installation by the ROTI to generate integrity measurements that bind the speific boot of the *sCore* to the ROTI and TPM. The *sCore* uses the TPM to generate system secrets (e.g., IKE private keys) on each bootcycle (and erase them on shutdown), linking them to the TPM. Only the TPM stores secrets that span multiple bootcycles.

Runtime *sCore* : The *sCore*'s user-level software is limited to a near-minimal number of services necessary to bootstrap user VMs[2] and monitor their communications, as necessary for the *Shamon*. Fixing the *sCore* software packages enables the remote party to predict the expected *sCore* software, which makes verification more predictable. It also limits the number of open network ports in the *sCore*, thus simplifying the task of showing that high integrity services protect themselves from malice.

[2]The code loaded into user VMs is not limited by this approach, although the *Shamon* policy may restrict it.

3.3 *sCore* Integrity

We show how ROTI-based integrity measurement approximates classical integrity, in this case Biba integrity [5]. The *sCore* is a two-level system, where each *sCore* is high integrity and inputs from any other subjects are low integrity.

Requirement 1: High integrity *sCore* installation: *A remote party will accept an sCore as* high integrity *if it can prove: (a) all executing sCore software originates from an acceptable ROTI and (b) all sCore data originates from an acceptable ROTI.*

Since the ROTI installer is trusted and all software and system-specific data can be verified as originating from the ROTI, then the *sCore* installation is high integrity. As the ROTI records the set of hashes for all software and installed files on the root filesystem, a remote party can verify that the files have the expected hashes. For system-specific files, their hashes can be verified based on those generated at install time. We find that a few files in the root filesystem may be modified at runtime (see Section 4.2), but they can be handled as exceptions.

Informally, the goal of the ROTI installer is to establish an acceptable system state, as defined by the goals of the system, with respect to both data and code. While a remote party may use measurement lists to verify running code, data correctness is much more difficult to verify, as potential values for data may vary widely from system to system. The ROTI acts as a trusted party to establish acceptable operational system data that varies from system to system. Consequently, this means that if the ROTI installer turns out to be untrustworthy, the code can still be verified by a remote party, but the data on the machines (config files, for example) may be malicious.

Requirement 2: High integrity *sCore* across bootcycles: *When an sCore system is booted, we have two requirements: (a) a sCore must verify the integrity of its system-specific data in a manner that can itself be verified by a remote party and (b) a sCore must limit any secrets to a single bootcycle if they may appear in the its memory in cleartext.*

First, a remote party depends on the *sCore* to demonstrate that it successfully validated its system-specific data in order to justify requirement 1b on a boot. This measurement must bind the ROTI to a value representative of this data. Our approach uses a process that does not depend on such data to compare expected and actual values of such data. Second, the *sCore* has a small number of keys that its uses (i.e., appears in cleartext in *sCore* memory), such as its IKE private key. Our *sCore* design generates such keys on each bootcycle to prevent their theft and use in an untrusted system. Using integrity measurement, we associate the keys with the bootcycle by using integrity measurement to record the new certificate when it is generated. The *sCore* design takes steps to prevent the use of such secrets after boot as well. This requirement goes beyond the traditional Biba requirements to prevent masquerading as required by Smith [32].

Requirement 3: High integrity *sCore* at runtime: *After verifying a high integrity sCore according to Requirement 1, a remote party will continue to accept an sCore as* high integrity *if it can additionally prove: (a) that it has checked the integrity of all the software will be loaded by the sCore and (b) all sCore software protect themselves from malicious input (e.g., code injection).*

First, the *sCore* restricts the software that can be loaded (i.e., into the domain 0 VM) to a prescribed set, so the remote party can tell that: (1) all the *sCore* software is measured and (2) no other software will be loaded. Also, the *sCore* does not allow users to login to the system (i.e., there are no such programs at the *sCore* level and no user identities), so user modification of the *sCore* at runtime is not possible [3]. Since all *sCore* processes are identified at verification time, the system will retain its Biba integrity throughout its run. Second, Biba requires that processes accept no low integrity inputs. However, the *sCore* has four software components that must have network interfaces (see Section 4). Each supports only a small number of legal commands, so a detailed evaluation of the correctness of input filtering is possible. We do not perform such filtering at present, but the system design makes such filtering practical.

4 Implementation

Our prototype *sCore* is shown in Figure 3. The ROTI is an Ubuntu Linux installer kernel version 2.6.20 that we modified to load our near-minimal *sCore*. The *sCore* consists of a Xen hypervisor version 3.0-unstable running a paravirtualized Linux domain 0 kernel version 2.6.18 with SELinux [2]. We use the Xen sHype [28] and SELinux Labeled IPsec [17] to authorize inter-VM communications for enforcing *Shamon* MAC policies [25]. We have extended this Linux kernel with the Integrity Measurement Architecture (IMA) patch [15, 30]. The key *sCore* services (i.e., the ones that implement *Shamon* operations) are the: (1) `trustd` (i.e., the *trust service*) that implements *Shamon* operations; (2) the IKE daemon `racoon` that creates secure (IPsec) communication channels to connect *sCore* into *Shamon*; and (3) `xend` that bootstraps user VMs,

The *sCore* user-level software is shown in Figure 4. The software is collected into groups depending upon whether it is only run at initialization, is only accessible to local processes, or is a network-facing daemon (by *type*). Figure 4 also shows which software was modified for the *sCore*. The

[3] Note that an user modification of the root filesystem would be detected at boot-time.

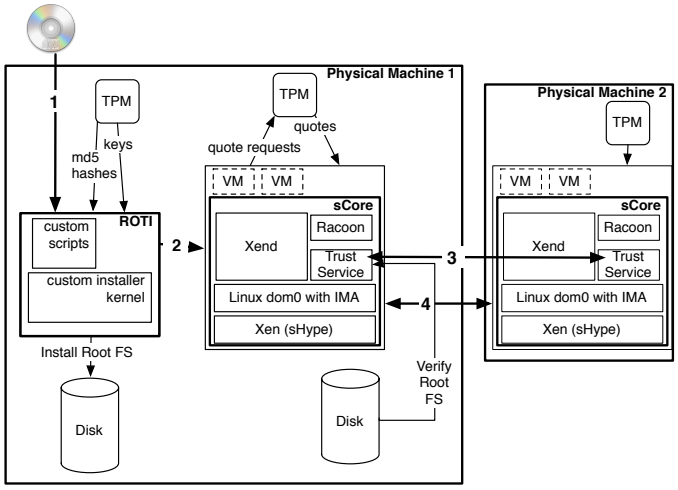

Figure 3. *sCore*'s lifecycle: (1) a ROTI installs the *sCore* by generating a verifiable root filesystem and creating TPM keys; (2) the *sCore* boots verifying its own root filesystem's integrity; (3) two *sCore* attempt to form a *Shamon* , each trust service makes a quote request to generate an attestation; and (4) a successful *Shamon* join results in an IPsec tunnel between the two systems.

trust service is a new component specifically for the *sCore*, and `racoon` and TPM utilities have been modified to work with the trust service. The TPM utilities software is derived from IBM Research's TPM software [14].

4.1 Installing the *sCore*

A typical installation requires the user to answer 10-20 questions regarding the configuration of their system. These questions cover a variety of topics, such as system preferences (e.g., language selection, keyboard, etc.), disk partitioning, network setup, user names and passwords, and any additional packages that the user may want to install. We pre-seed the `debian-installer` with a file called `srm.seed` that provides answers to these questions. Supplying the pre-seeded answers to most of the question is straightforward (e.g., we use the DHCP client to obtain the network configuration, and no user accounts are created), but for disk partitioning, there are several legitimate answers. In our implementation, we choose separate boot and root partitions to provide the option of an encrypted root file system. However, other "safe" choices are possible. Ultimately, we intend to define an interface for the user to choose among these "safe" options for partitioning.

The *sCore* ROTI consists of a custom installer kernel (e.g., including TPM libraries), `initrd`, the `debian-installer`, and the set of packages that may be loaded. The `debian-installer` installs the `ubuntu-standard` virtual package, which in turn, installs its dependencies. This part of the install includes base libraries, such as `libc`, and a minimal software install (from the Ubuntu folks perspective). Some of these packages are not used in the *sCore*, so they may be removed. The standard install is followed by an installation of custom packages that includes: (1) customized TPM software utilities (for generating measurements, attestations, and unsealing); (2) a customized kernel package containing the Xen hypervisor, the paravirtualized Linux kernel (customized to include Linux IMA), and supporting configurations and scripts; (3) `ipsec-tools` packages; (4) our *sCore trust service*; and (5) any additional packages required to fulfill dependencies. These packages are md5-hashed, and their values are then signed with our ROTI's GPG key[4] and included in a file `Release.gpg`. The ROTI validates each software package against its hash prior to installation.

Type	Programs	Source	Purpose
System Initialization	Initialization	Std	User-space initialization
	`openssl`	Std	Generate IPsec key pair
	TPM utilities	Mod	TPM ops
Local Daemons	`udevd`	Std	Used by Xen
	`logd`	Std	Logging daemon
	`getty`	Std	Terminal support
Network Daemons	`dhclient`	Std	DHCP client
	`racoon`	Mod	IKE daemon
	`xend`	Std	Load User VMs
	`trustd`	New	*Shamon* trust service

Figure 4. User-level software in the *sCore*: (1) System initialization software is run at startup only; (2) Local daemons are not network accessible; and (3) Network daemons have at least one network interface. Source indicates whether the *sCore* version is unmodified (Std), modified (Mod), or new for the *sCore* (New).

Although we do not implement it, this is the stage of the installation where signed patches could be automatically downloaded from the same remote package source used in previous stages of the installation. This would give the ROTI installer the ability to stay up-to-date with respect to vulnerabilities. However, due to the large number of complications with respect to the plethora of patching strategems employed today, we choose to discuss this in future work.

Once all the packages are installed, post-installation scripts complete the configuration. First, this script evicts the old TPM state and creates new TPM keys for signing (i.e., quoting) attestations. Second, it generates an entry in the Grub bootloader's configuration file (i.e., `menu.lst`).

[4]Since we modified some of the packages to be installed, we had to generate our own GPG key for the ROTI. The intention is for the distributor to sign their version of the *sCore* installation.

The custom installer also links the installed root filesystem to the ROTI. The ROTI computes a hash for each file in the root filesystem and collects these hashes into a single file called `md5sums.txt`. Since the root filesystem is of moderate size, this operation is practical (about one second). The ROTI then uses the TPM to *seal* the file `install_md5sums.txt` to the current PCRs for the running installer, so that it can be opened (i.e., unsealed) only by the trusted *sCore* when it is booted. When the file is unsealed, the *sCore* measures the sealing PCRs (i.e., of the installer) to link the file to the ROTI. Note that this file need not be secret to the *sCore*.

4.2 Booting the *sCore*

As the *sCore* is booted, the individual stages (see Figure 2) collect integrity measurements to justify that the *sCore*'s integrity can be linked to its ROTI installation. Booting the *sCore* involves booting the Xen hypervisor and Linux domain 0 kernel, verifying the integrity of the root filesystem, initializing the system, and starting the *sCore* services. Each step is accompanied by integrity measurement tasks.

First, the Trusted Grub bootloader [1] boots the Xen hypervisor. Prior to booting, Trusted Grub measures the Xen hypervisor, domain 0 Linux kernel, the stage 2 bootloader, the `initrd`, and the command line boot parameters. The installed Grub configuration file `menu.lst` specifies the necessary measurements, and Trusted Grub's current functionality supports such measurements.

Next, the Xen hypervisor loads the Linux domain 0 kernel. Our domain 0 kernel is a Linux 2.6.18 kernel modified to run as a Xen virtual machine (i.e., paravirtualized) and extended to perform integrity measurement using the Linux Integrity Measurement Architecture (IMA) patch [15, 30]. The bootloader measures the kernel, so we depend on the integrity of the Xen hypervisor to ensure that the correct domain 0 kernel is loaded. Using Linux IMA, each user-level executable, libraries, and kernel modules are automatically measured.

Initialization of the *sCore* user-level services starts by verifying the integrity of the root filesystem using the *install rootfs quote* from the installation. A script in the `initrd` checks the hashes of each file in the root filesystem with the hashes in `md5sums.txt`. The integrity of `md5sums.txt` is verified by ensuring that PCRs of the sealing system correspond to a legitimate ROTI. To enable remote verification, an IMA measurement entry containing the sealing PCRs (i.e., the ROTI PCRs) and file name is recorded. The file contents are system specific, so they need not be provided in the measurement.

We detected that a small number of files (three) in the root filesystem are modified in the course of a *sCore* initialization. These files include `mtab`, `blkid.tab`, and `blkid.tab.old`. For example, `mtab` maintains a list of currently mounted filesystems, so it is written on each initialization. There are a number of options for handling these exceptional cases: (1) verify these files locally using trusted program; (2) move the files out of the root filesystem (e.g., link to a file in `/var`); or (3) submit the modified versions to the remote party for verification (since the number is small). As some files may be security-sensitive, such as `mtab`, that a mechanism to validate some exceptions will be necessary.

Next, the *sCore* generates the IPsec keypair and IPsec certificates for the bootcycle. Recall that we generate a fresh keypair on each boot to prevent the theft of such secrets from memory (see Section 2.3). The IPsec keypair are generated using `openssl`. We bind the new key pair's certificate to the TPM by generating an IMA measurement of the certificate. This binds the keypair to the bootcycle and TPM. In future work, we will then simply modify ipsec-tools to check that the attestations being exchanged during racoon's negotiation include the `openssl` certificate used to secure the connection.

Finally, the *sCore* must bootstrap itself as a networked device capable of participating in a *Shamon* . We use DHCP to obtain an IP address for the *sCore* . Thus, the *sCore* includes a DHCP client in its software stack. The DHCP client is the only service that accepts unauthenticated input currently [5]. Next, the *sCore* must be able to locate the authorities for joining *Shamon* . This includes one or more *Privacy CAs* and one or more *Shamon Authorities*. The former enable the *sCore* to securely obtain the public keys of other *sCore* systems. The latter enables the *sCore* to identify other *sCore* and their mapping to distributed applications. These identities are provided by the installer.

4.3 Running the *sCore*

Once the *sCore* is initialized completely, it can participate in one or more *Shamon*. In order to join a *Shamon*, the *sCore* must convince a remote party that it is a legitimate, high integrity *sCore* via an attestation (i.e., a freshly signed integrity measurement) [26]. After a successful attestation, the *sCore* officially joins the associated *Shamon*, downloads the *Shamon* mandatory access control (MAC) policy, and runs and migrates virtual machines (VMs) for the *Shamon* distributed applications. Over its lifetime, the *sCore* must protect itself from malicious modification and the loss of communication secrets (i.e., IPsec keys).

First, Figure 4 lists the software that is running in our *sCore* prototype. All software is loaded at initialization time, and no further software is executed by the *sCore*. The fundamental *sCore* services are `xend` which launches and migrates virtual machines, `racoon` which is an IKE daemon (specifically, for the `ipsec-tools` IPsec suite), and

[5] However, methods to authenticate DHCP have been proposed [10].

our *trust service* which performs mutual attestations with other *sCore* to build *Shamon*. The other programs initialize the system (`init` and `getty`), support service functions (such as logging in `logd`), and obtain an IP address (`dhcpc`, as described above).

A *Shamon* join invokes the *trust service* to perform a mutual attestation with a remote *sCore* . The *trust service* receives a nonce from the remote *sCore*(i.e., a challenge), and generates an attestation (e.g., response) using its TPM (e.g., see the Linux IMA attestation protocol [30]), plus the trust service generates the challenge for the other *sCore* to do its attestation. The *attestation quote* is sent with the IMA measurement list to the remote *sCore*. A successful attestation requires that: (1) the IMA measurement list of hashes (i.e., software loads and verification of the root filesystem) correspond to the hash aggregate signed in the attestation and (2) that the remote party accepts the ROTI that generated the root filesystem (whose integrity check is in the IMA measurement list).

Once a *Shamon* join is complete, the *trust service* updates the MAC policy for the *sCore*. The MAC policy is the current policy being enforced on VM communications. It consists of three components [25]: (1) the Xen sHype policy that governs local VM communication; (2) the SELinux policy for Labeled IPsec that governs remote VM communications; and (3) the IPsec policy that links cryptographic provisioning with MAC of communication. These policies are initialized based on input from the *Shamon* authorities, but each *Shamon* defines its own MAC policy components. We note that these policies persist only within one boot-cycle, so there is no impact on the root filesystem or the integrity of the *sCore* itself for future boots.

The near-minimal *sCore* must protect itself from malicious input. Only `dhcpc`, `racoon`, `xend`, and our *trust service* may accept messages from remote parties. Further, all `xend` messages must originate from Labeled IPsec tunnels. While we do not provide a formal proof of secure input handling, verification is practical given the small number of programs. The *trust service* messages are limited to *sCore* initialization/updates, attestation requests/responses, and MAC policy updates. Furthermore, `xend` is written in Python and has a carefully-designed module to filter input. Evaluating input filtering of these services is future work.

The remaining challenge is to prevent an IPsec session from being hijacked by the reboot of an untrusted system that can read memory from the previous boot. This is only a problem when a machine is rebooted with the power on. On a normal shutdown, a script `/etc/init.d/stop` is invoked to clear the IPsec state from the kernel. Some systems crash and reboot automatically without a shutdown, so the *sCore* implementation must account for this as well. A *sCore* crash should be infrequent and should not automatically reboot the system, but addressing this specifically is future work. From the remote party's perspective, it must detect a broken *sCore* connection. To do this, we use the IPsec dead peer detection messages set at 10 second intervals (i.e., longer than the currently practical reboot time) to detect whether a peer is not longer an active *sCore* . After this time, a mutual attestation is required to reconnect the *sCore*.

5 Evaluation

We evaluate the ROTI-based *sCore* by measuring its installation, boot, and runtime overheads. All of the following experiments were run on Dell Precision 380 machines with 2.8 GHz Pentium D processors, 1G of memory, and 120G PATA disks. The installer is based on Ubuntu Edgy (6.10), and installs the March 2nd Xen-Unstable build of Xen, which uses a patched 2.6.18 Linux kernel.

During installation, the ROTI performs several tasks beyond what is included in a normal installation, detailed in Section 4.1. We measure the performance of each of the following discrete tasks: (1) install *sCore* -specific software packages; (2) create TPM signing keys; (3) update the bootloader configuration in `menu.lst` to boot the custom kernel; and (4) build the root filesystem integrity file `md5sums.txt` and TPM-Seal the file (binding it to the ROTI). Fortunately, these operations have a minimal impact on performance. As can be seen in Figure 5, the normal operations involved in installing an operating system dominate the total time to install *sCore*, as the operations we add only comprise 8.4% of the total installation time.

We also examine the overhead of the resultant *sCore* boot compared to a Xen system boot. The only additional operations the *sCore* requires at boot-time are: (1) the integrity measurements of Linux IMA; (2) the IPsec key pair generation using `openssl`; and (3) the root filesystem integrity validation. The IMA integrity measurements cost on the order of milliseconds for the small amount of measurements made [30], and the IPsec key generation is also fast at 0.62 seconds.

The root filesystem validation requires hashing the entire root filesystem and comparing to the expected hashes in `md5sums.txt`. The hash computation consumes an average of 1.36 seconds. As we timed the boot sequence, the total boot process took an average of 69 seconds, making the overhead added by key generation and filesystem validation quite small (less than 3%).

After boot, most of the overhead in the *sCore* drops out. IMA has already hashed the programs that are loaded and extended the appropriate PCRs, the root filesystem was hashed once and does not need to be hashed again, and the list of md5 hashses has already been unsealed. At this point, the only notable performance impact on *sCore* is the exchange of attestations that takes place before encryption communication between the *sCore* is established. This adds

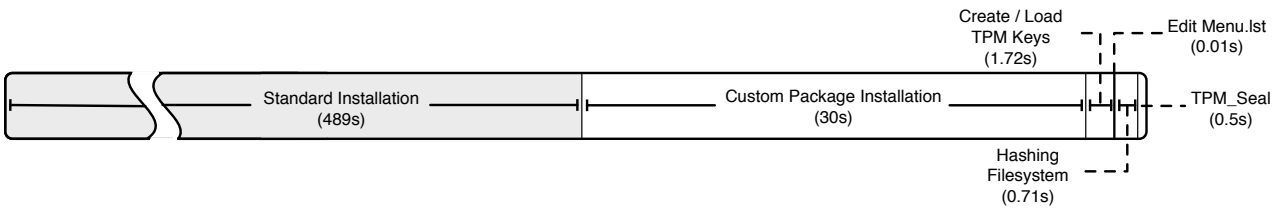

Figure 5. A performance breakdown of the major tasks in the install phase. The majority of the time is consumed by the standard install.

2.31 seconds to the average IPsec negotiation, and needs only be performed at every phase 1 security association time out. Of this, 0.93 seconds is used to make the attestation to be given to the remote party, and 0.07 seconds is used to verify the remote party's attestation. The remainder of the time is devoted to network communications and the actual transmission of nonces and attestations.

6 Related Work

Most integrity measurement approaches involve measuring the software of a system and/or its static files (or memory) [24, 26, 30, 29, 11]. We have shown that these approaches are insufficient, but there are other integrity measurement approaches. An alternative is to verify the inputs to high integrity operations and measure the code and outputs of those operations, such as is done in the BIND system [31]. However, this implies that only some operations in high integrity software are really integrity-critical, but these are difficult to identify and separate. All operations of all software in the *sCore* appear to be integrity-critical. Another alternative is the PRIMA integrity measurement approach that ensures that high integrity data is only modified by high integrity processes [19]. However, PRIMA does not ensure that the data was installed in a high integrity fashion and does not guarantee integrity after the attestation.

The Bear system from Dartmouth [23] identifies some of the challenges of addressed in this work. They identify attacks against data, such as the replay of old dynamic data, that can impact the integrity of the system. The Bear divides the system into long-lived core, medium-lived software, and short-lived data. An *enforcer* is a long-lived component that verifies the integrity of medium-lived software and tracks the values of short-lived data. Verification is based on information from a remote *Security Admin*. The Bear provides some useful general ideas, but does not ensure that the Security Admin can be trusted by remote parties, does not address long-term secrets, and does not envision managed system configurations, such as the *sCore*.

We are aware of emerging research that uses hardware features to guarantee that only authorized code is ever executed by a system [22]. Such work uses the execute protections of the `x86` hardware to prevent a page from being executed until it is authorized. If we know all the software that can be executed on a system, such as the *sCore*, then we can use such techniques to limit execution to just that software. We see these types of techniques complimentary to the *sCore*. We would use such a function, but we still need the ROTI to just a high integrity installation, filters to protect against malicious inputs for data attacks, and the protection of system secrets.

The *Shamon* approach also leverages virtual machine (VM) technology to enforce mandatory access control (MAC) across a distributed system. Virtual machines provide a layer of isolation and VM communication is coarser-grained than OS system calls. Thus, work is underway to add MAC to VM systems (e.g., sHype and Xen Security Modules [28, 9] for Xen). Further, MAC in a distributed system depends on a secure communication mechanism that can convey security labels. We use SELinux's Labeled IPsec [17], but other alternatives exist [27]. It is unclear which approach may prove most effective in the future.

7 Conclusions

In this paper, we developed an approach to building and verifying high integrity systems based on a *root of trust installation* (ROTI). The ROTI links both software and system-specific configuration files back to the trusted installer that generated them. While the ROTI idea is straightforward, a number of challenging design decisions must be made to implement it correctly using the TPM hardware. Developed from a clearly defined set of requirements that must be met to build a high-integrity system, we have explored the systemic requirements of *installing*, *booting*, and *measuring the runtime integrity* of the ROTI-installed *sCore*s. The implementation and experiments demonstrate that we can build a practical large-scale integrity-measured distributed *Shamon* system. In the future, we will explore further use of the *Shamon* for constructing large distributed application environments, leveraging the homogeneity of the integrity-assured components (*sCore*) to enable distributed trust.

References

[1] GRUB TCG Patch to support Trusted Boot. `http://trousers.sourceforge.net/grub.html`.

[2] Security-enhanced Linux. http://www.nsa.gov/selinux. http://www.nsa.gov/selinux.

[3] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, The Mitre Corporation, Air Force Electronic Systems Division, Hanscom AFB, Badford, MA, 1972.

[4] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 1997.

[5] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE, April 1977.

[6] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust Management System, version 2. IETF RFC 2704, Sept. 1999.

[7] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 164, Washington, DC, USA, 1996. IEEE Computer Society.

[8] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. 14th USENIX Security Symposium*, August 2005.

[9] G. Coker. Xen security modules (xsm). In *Xen Summit*, September 2006.

[10] R. Droms and W. Arbaugh. Authentication for DHCP Messages. RFC 3118, IETF, June 2001.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating System Principles(SOSP 2003)*, Bolton Landing, NY, USA, Oct. 2003.

[12] T. C. Group. http://www.trustedcomputinggroup.org/, March 2005.

[13] J. Hendricks and L. van Doorn. Secure bootstrap is not enough: Shoring up the trusted computing base. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[14] IBM. Ibm research - gsal - trusted computing. http://domino.research.ibm.com/comm/research_projects.nsf/pages/gsal.TCG.html/.

[15] IBM. Integrity measurement architecture for linux. http://www.sourceforge.net/projects/linux-ima.

[16] A. Iliev and S. W. Smith. Protecting user privacy via trusted computing at the server. *IEEE Security and Privacy*, 3(2):20–28, 2005.

[17] T. Jaeger, S. Hallyn, and J. Latten. Leveraging IPsec for mandatory access control of Linux network communications. Technical Report RC23642 (W0506-109), IBM, June 2005.

[18] T. Jaeger, P. McDaniel, L. S. Clair, R. Caceres, and R. Sailer. Shame on Trust in Distributed Systems. In *Proceedings of the First Workshop on Hot Topics in Security (HotSec '06)*, Vancouver, B.C., Canada, July 2006.

[19] 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.

[20] T. Jim. SD3: A Trust Management System with Certified Evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, Oakland, CA, USA, 2001. IEEE Computer Society.

[21] N. Li, B. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):128–171, Feb. 2003.

[22] P. Loscocco, 2007. NSA. Personal Communication.

[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. Technical Report Computer Science Technical Report TR2003-476, Dartmouth College, 2003.

[24] 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.

[25] J. McCune, T. Jaeger, S. Berger, R. Cáceres, and R. Sailer. Shamon: A system for distributed mandatory access control. In *Annual Computer Security Applications Conference*, 2006. To Appear.

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

[27] Netlabel - explicit labeled networking for linux. http://netlabel.sourceforge.net/.

[28] R. Sailer and *et al*. Building a MAC-based security architecture for the Xen opensource hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*, Miami, FL, USA, Dec. 2005.

[29] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 308–317, 2004.

[30] 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.

[31] 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*, Oakland, CA, USA, May 2005.

[32] S. W. Smith. Outbound authentication for programmable secure coprocessors. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Oct. 2002.