The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

SHAMON - ESTABLISHING TRUST IN DISTRIBUTED VIRTUALIZED ENVIRONMENTS

A Thesis in

Computer Science and Engineering

by

Luke William St.Clair

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2008

The thesis of Luke William St.Clair was reviewed and approved* by the following:

Patrick McDaniel
Associate Professor of Computer Science and Engineering
Thesis Co-Adviser

Trent Jaeger
Associate Professor of Computer Science and Engineering
Thesis Co-Adviser

Mahmut Kandemir
Associate Professor of Computer Science and Engineering
Chair of Graduate Program of Computer Science and Engineering

*Signatures are on file in the Graduate School.

ABSTRACT

Distributed applications often require complex trust relationships between remote parties in order to enable a wide range of complex functionality. Unfortunately, it is difficult to achieve useful guarantees when interacting with remote systems in the face of an attacker with physical access. Specifically, we need a solution that is both comprehensive and usable for a wide variety of trustworthy distributed applications. This thesis presents such a solution, comprised of two main parts: a secure trusted computing base (sCore) using secure hardware on which applications run in virtual machines and a control infrastructure to manage the virtual machines (TVMI). We use virtualization to separate programs and trusted computing to verify the software stack on which they run. This enables us securely form groups of applications which utilize the separation and trust established by the sCore. In order to do this, we build a secure core of software (sCore), which establish mutual trust with each other to build a distributed reference monitor with strong security guarantees. Using this trustworthy platform, we create an infrastructure for managing coalitions of virtual machines. This enables groups of applications capable of entering well-founded trust relationships with each other. We then use formal verification to prove relevant security properties of the virtual machine control infrastructure which runs on the trusted computing base. We build our sCore and TVMI in order to demonstrate that our solutions are feasible in a realistic setting, deliver acceptable performance, and provide security properties that were not previously available. Additionally, our formal verification demonstrates that the critical properties the TVMI purports to enforce are met by our design. This helps to demonstrate the important problems in distributed, secure virtualization that are not being handled today, and gives a framework for their evaluation and operation.

Table of Contents

List of Figures

Acknowledgments

Thanks to my wife, whose editorial help and constant encouragement were instrumental in helping me to complete this work.

Chapter 1

Introduction

1.1 Introduction

As computer science progresses, we are using computers in an increasingly distributed fashion. Whereas a decade ago, many systems were used in a largely isolated fashion, most useful processes today involve groups of systems collaborating to achieve some goal. However, security for distributed systems has historically involved multi party trust based on weak assumptions, such as strength/secrecy of passwords, social responsibility, or simple key secrecy. Unfortunately, this weak trust model has served to compound problems created by the systemic presence of vulnerable pieces of software. As distributed environments become more widely used for a variety of applications, stronger security guarantees are required.

Consider, for example, an extensive information mining application run by a company wishing to detect consumer trends and preferences. Given the size of many data mining operations, this must take place over many distributed nodes. However, it is important that the data mining results not be leaked, due to privacy concerns and a desire to maintain a competitive advantage. Additionally, the company would also like to ensure that the trends and preferences being returned are correct and not the result of a malicious process inserting invalid data. With current solutions, it is difficult to quantify or prove that these sorts of security goals are achieved.

Additionally, this example introduces the idea of a coalition, which we define as a group of systems with a shared goal. In our example, this is the group of systems collaborating to produce a clearer picture of consumer trends and preferences, while enforcing (or attempting to enforce) a common set of security goals over the set of participating nodes. As shown by this example, it has become increasingly important to be able to establish a coalition secured by something stronger than poorly-based trust. Either blindly trusting or trusting to the secrecy of credentials common to a large number of parties has become less acceptable today, while distributed computing continues to be more widespread. This calls for a new design capable of enabling trustworthy collaboration and communication, in order for distributed systems to achieve their potential in applications where security is a priority.

Clearly, such a solution would have far-reaching impact, given the prevalence of distributed systems in the modern computing environment. Why don't we have a better solution today? In general, the real-world threat model for distributed systems can be quite strong. For instance, many times when a user interfaces with a remote system (for instance, nearly all web browsing), they do not know anything meaningful about the physical or administrative security of that system, since it is not under the user's control. The user usually does not know how many people have root access on that system, what software is installed, or usually even what OS is running. If the remote system is vulnerable to compromise due to an outdated or insecure software stack, the user's interactions with that system may be completely subverted. The common state of distributed computing provides neither the necessary information to discover any of this, nor guidance as to how to interpret or use this information.

This thesis seeks to provide a practical method of building multi-party trust, which can be used to build trustworthy coalitions. In the next few sections, we discuss previous solutions, as well as introduce our approach for multi-party distributed trust. We discuss the guarantees we seek to provide, as well as an understanding of why they were not adequately provided by preceding works. Finally, we discuss how such a system could be used in practice, while still maintaining the security goals provided by the core infrastructure of our solution.

> *Trustworthy distributed coalitions of virtualized applications can be constructed in a such a way as to provide not only strong hardware-based security guarantees of*

integrity for each participant, but also flexible, scalable management for the coalition as a whole.

1.2 Distributed Virtualization

As we attempt to design a solution, we must first examine what tools we have available to us. Given that we want strong guarantees about the state of a remote system's software stack, it is natural to turn to the Trusted Platform Module, or TPM, as a start [39]. The TPM is a tamper-resistant hardware device that comes embedded in a system's motherboard, and supports only a limited set of cryptographic operations. In brief, the TPM can be used to certify the software stack running on a given system, via a process known as attestation [17]. The TPM records the measurements of each piece of software loaded (this will be discussed more in Chapter 3). The TPM does not allow any user of the system to change these measurements, which are cryptographically tied to the TPM itself. Consequently, the TPM can be called upon to attest to the state of the system and present this attestation to a remote party. The remote party can then check the measurement list to see what programs were run, and that the attestation is actually from the TPM. This enables the remote party to make an informed determination as to whether or not to trust the system. This provides a hardware-based root of trust, which gives a remote party a foundation for building trust in the system.

However, this approach has some general weaknesses. Firstly, the attestations received are often quite complex. If a system measures every program, library, and kernel module loaded, it can be difficult to say something meaningful when this attestation exceeds 30,000 lines. For instance, if a remote party sees a hash for a program "/usr/bin/clamasivd", what can he say about this? Is it just a misspelling of the ClamAV virus scanner? Is this simply an innocuous program that is relatively unknown? Is it a rootkit? Proving that some properties hold over such a complex attestation is quite difficult when the list of potential programs that could be in the attestation gets large.

In order to trust a remote machine, however, we must do more than simply trust a set of relevant processes on a foreign system: we must also trust any process that is capable of interfering with the processes we care about. For instance, if we want to trust a remote webserver on a traditional system X, and X is also running a mail server as root, then we must trust both the web server and the mail server. More generally, we must understand and demonstrate that every process and user on the remote system cannot compromise the security goals we want satisfied. A large body of research and practice has shown this is currently an impractical problem to solve completely in commodity operating systems [2, 31]. However, virtualization provides the type of well-defined separation we are looking for in this application. If each application we care about runs in a different virtual machine, and we trust the separation provided by the Virtual Machine Monitor (VMM), then we need only trust the exact set of applications we need. Trusting a VMM's separation is a reasonable choice for two reasons: the amount of code in a VMM is smaller (which assists verification and code auditing), and the goals of the VMM are much simpler than a full-blown operating system. The VMM's separation gives the additional benefit that authorization policies are much simpler to create and understand when virtual machines are the principals involved, as opposed to every possible user and process in a given system. For instance, if two services share the same OS, then to ensure that they cannot interfere with each other, we must analyze the operations available to every user on the system, every process on the system, and every operation all entities can perform. With virtual machines, we must only ensure that sharing between VMs is not enabled (although we can also enable sharing to allow applications to interact in limited ways).

Since the weakness of trusted computing is complexity, while the strength of virtualization is that it provides simple separation, it seems natural to combine the two in pursuit of a workable solution to distributed systems security. In fact, this is precisely what was done in a proceeding work, the Shared Reference Monitor, or Shamon [66]. This work built coalitions of virtual machines, by means of remote attestations passed among virtual machine monitors (VMMs).

The goal is to establish a distributed reference monitor among the VMMs, which provides the guarantees of a traditional reference monitor (tamperproof, verifiable, complete mediation) in a distributed environment. In other words, if these guarantees are met in a given system, and that system can prove those guarantees to a remote party, a trusted coalition can be built among the VMMs where each party trusts the other. Once that coalition is established, virtual machines can run applications to do the important computing work. This leaves the portion of the system that must be verified (the VMM) small, so as to minimize the complexity of the attestation, while leaving the difficult work of separation to the comparatively simple virtualization layer.

1.3 Improving Attestation

However, this solution, as presented in the 2006 ACSAC work on the Shamon system [66] is overly simplistic. This work, as well as nearly every other work in the field [63, 35, 11], says very little about what must actually be in an attestation. In general, only loaded code is required to be included in the attestation, as well as hashes of lower layer software. This is not all that is required to secure a system, however. The data on a system contributes to the security of many processes, in addition to the actual binaries being run. For instance, if the PAM daemon allows passwordless logins for root users, this system no longer conforms to most security specifications, even though the ssh daemon being run is not deliberately malicious. Consequently, we need an approach that verifies everything on a system, not just the loaded code. However, as we have mentioned previously, verifying the code on an arbitrarily complicated system can be very difficult. As described in chapter 3, this requires us to tightly constrain the systems we wish to rigorously verify.

Unfortunately, even in a constrained system, it would be impractical to simply measure every file on a system. Many files must be different in order to properly configure systems in a wide variety of enviornments, and understanding the security implications of each possible difference is a very challenging problem. Instead, we attempt to prove where each piece of code and data comes from. Our realization is that the install media for a distribution of an operating system provides a natural root of trust for data and code, since the original data and code on a system must ultimately come from the installer. Consequently, we utilize this concept to create a Root of Trust Installer (ROTI), which installed a trusted VMM and management infrastructure. The system is designed with the goal that the data and code that are installed can be traced back to the installer. In this way, we can form a real ”whole system” attestation. Once this is done, we can begin to build coalitions based on meaningful trust.

1.4 Trusted Virtualization Infrastructure

The goal of this work is to take the Shamon, which is comprised of VMMs and their associated trusted domains, and form coalitions of virtual machines. Given that each virtual machine is running on the Shamon, the virtual machines themselves would then have certain useful security properties (information flow control, for instance) simply as a result of running on the Shamon. Each user could have many virtual machines, one for each application, that belong to a variety of coalitions which grant different security properties. For instance, a graduate student Alice may have one virtual machine belonging to a coalition in her lab that guarantees that information won't flow out of the lab coalition, and she might have another virtual machine for a massively multiplayer online game that guarantees only that all traffic to the game servers are encrypted. We will discuss how this is achieved in chapter 3.

However, to implement such a flexible model, we must have some trustworthy, usable system to controlling VM coalitions (and the VMs themselves). Previous work in this area has focused primarily on creating a trustworthy environment for program (or virtual machine) execution. However, there are a number of relevant security concerns related to the creation, migration, and execution of virtual machines at the application layer. Consider the case where a user, Alice, wants to start a virtual machine, which she wants it to be a part of a graduate student coalition of virtual machines. We need to create a coalition authority capable of supporting this operation, ensuring Alice's VM conforms to the standards of the graduate student coalition, and watching out for Alice's virtual machine. Alice needs to be able to migrate, start and stop her virtual machine, while continuing to enforce her security goals.

However, we must also consider this situation from the perspective of a University, Grad U. Grad U's sysadmin would like to host a number of research coalitions on department machines. However, he doesn't want any one coalition to achieve an unfair proportion of resource utilization over university machines. Since both Alice and Grad U's sysadmin are integral players in this system, we need to devise a trustworthy system that enables Alice to get virtual machine resources, but only with the permission of Grad U's sysadmin.

Consequently, both the Grad U sysadmin and Alice must be able to trust the virtual machine management infrastructure in different ways. Alice must trust that her security goals will be enforced independant of the location of her VM, while Grad U's sysadmin must be sure that Alice is not consuming an unfair proportion of their computing resources. To this end, we created the trusted virtual machine infracstructure (TVMI), to manage these requirements in a secure, usable fashion. However, in this thesis we do not thoroughly investigate or address resource management issues, as this is a well-developed and complicated field of research, whose results could be applied to this work at a later date.

Figure 1.1 gives an overview of our solution for secure virtual machine management, the TVMI. In this diagram, each machine authority, or MA, is responsible for mediating access to the physical resources. As we will describe in more detail in Chapter 4, the MA simplifies resource management for coalitions, as it acts as the single entry point into an administrative domain of resources. In this way, an administrative domian, or collection of machines under the same scope of administrative control, provides a single interface for requesting resources, as well as a single policy decision point for physical resources. In short, the MA protects physical machines from unauthorized usage, and provides resources to authorized coalitions.

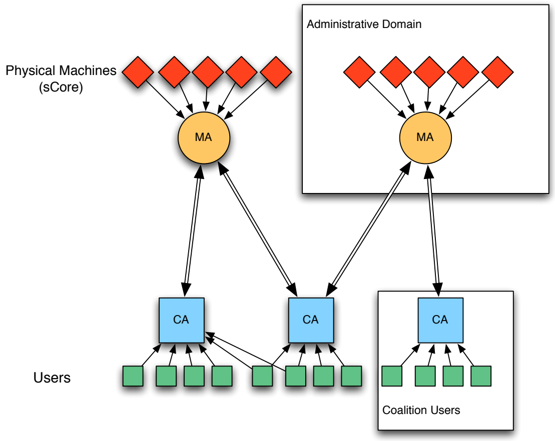

Fig. 1.1. Picture of the full TVMI infrastructure

The coalition authority, or CA, acts as the MA's counterpart as the access point to resources for use by coalition users. Where the MA acts as the resource broker for the physical machines, the CA obtains resources for users and protects their data (in the form of virtual machines). The CA is repsonsible for garnering resources from the MAs and ensuring that user VM data does not leak to an attacker.

As the figure suggests, this design minimizes the trust between foreign parties, and enhances the scalability of the design. Given that our solution is designed to enable internet-scale distributed trust, minimizing the complicated trust relationships between users and remote physical hosts is important.

1.5 Summary

In this paper, we explore previous solutions to these problems in chapter 2. We discuss the history behind these technologies, and explain why they fail to solve the important problems in trusted distributed computing today. In chapter 3, we discuss the ROTI's design and implementation. We discuss why it shores up the holes in current attestation schemes, and how it enables stronger trust relationships between parties. Chapters 4 details the design of the TVMI, which enables secure virtual machine control for users. This enables the formation of useful coalitions and begins to demonstrate how to build a large-scale system based on the Shamon concept. Chapter 5 explores the actual implementation of the TVMI, in addition to formally validating the TVMI's design. Finally, in Chapter 6, we conclude and detail our contributions, as well as provide direction for future work that needs to be done in this area.

Chapter 2

Background

2.1 Introduction

In order to accomplish the goals set forth in the preceding section, we must devise some method of establishing a basis for trust in a remote party. Given that this is such a general problem, it is somewhat unsurprising that much work has previously been done with this goal in mind. The last 10 years, in particular, have generated complex and divergent philosophies with respect to what makes a given remote system secure.

Specifically, there seem to be at least three relevant different approaches to system security that bear mentioning. The first approach, which we will refer to as the reference monitor approach, attempts to define the acceptable operations of a system with some general policy. Even if it does not govern all system operations, it must, by definition, control all operations relevant to a given set of security goals.

Generally, developing a policy for a reference monitor to govern all possible interactions in a system has proven difficult. While there have been some attempts to limit the scope of these policies [32], a different philosophy has become popular in recent years. Generally, the idea is that really specifying the totality of what goes on in an operating system is too difficult. Instead, we should totally isolate services we care about into their own virtual machines, and reason at the much coarser granularity of machines, instead of individual users and processes. This greatly simplifies the view of a system into discrete components.

However, these approaches are not satisfactory for some applications. Some users must have strict assurance that what they think is happening on a remote system, actually is. To do this, current solutions have used a piece of hardware, the Trusted Platform Module (TPM) [46]. The TPM can be used to measure parts of a system. These measurements can be given to a remote party to prove what code has been run. In this way, a remote party can have some assurance that the claimed state of a system actually matches the code that was run.

In this section, we give a general overview of these approaches, and discuss their strengths and limitations. In particular, we show how the strengths and weaknesses of these approaches complement each other, in order to motivate our solution, which we introduce in Chapter 3.

2.2 Reference Monitoring

In 1972, the Anderson report laid out the first criteria for building trusted systems [6]. Most notably, it helped to introduce the idea of a reference monitor into secure systems building. In short, this classical reference monitor approach is used to provide comprehensive enforcement of policies in a given system. In more modern literature, the particular designation of reference monitor is reserved for enforcement architectures which meet certain criteria. The orange book (TCSEC) set forth the most popular definition of a reference monitor today in its description of the reference validation mechanism (RVM). In short, the TCSEC indicates that a reference monitor must provide, with respect to the desired policies, complete mediation, tamperproofness, and verifiability. The core idea is that if these three criteria are met, a reference monitor will be completely capable of enforcing a given policy. Clearly, this model is quite general. Basically, it refers to an architecture capable of really enforcing a security policy, in the presence of a local, malicious adversary. Since this is such a general and useful goal, this idea has become extremely popular in recent years, as numerous OS technologies have progressed to the point of making reference monitors practical and widespread [75, 2].

In recent years, the Linux kernel has begun to include a framework for making a reference monitor known as the Linux Security Modules, or LSM [90]. LSM strives to provide hooks to all security sensitive operations identified so far, which enables complete mediation. In other words, when a security sensitive operation takes place, the appropriate LSM module is called, if it exists and defines a security check for that operation. The idea is that tamperproofness of this system is ensured by its inclusion in kernel space, which should theoretically not change. While this is may not accomplish tamperproofness as stringently as an EAL 5 system may require, kernel space is amongst the most trusted areas in a traditional operating system. As far as verifiability goes, LSM's small amount of code tends to lends credence to the idea that it is verifiable, but the question of whether or not this is verifiable is a little misleading. Since the LSM framework depends on the kernel, it would be difficult to verify both the LSM framework and everything it depends on.

The importance of LSM can be seen in the adoption of practical reference monitoring architectures in recent years. Most notably, the NSA released a reference monitor for the Linux kernel, SELinux, with the goal of creating a usable, mainstream reference monitor suitable for use in production systems [2]. Over time, it has become the de facto reference monitor implementation for the Linux operating system, and is included by default in many modern Linux distributions, as well as TrustedBSD and Trusted Solaris [89, 86]. More specifically, it is a reference monitor designed with the goal of enforcing flexible mandatory access control (MAC) policies.

In general, access control systems govern which operations a given subject can perform on a given object. Most users are familiar with discretionary access control systems, where a given user can change the permissions on an object (say, a PDF report) so that other users can or cannot access the object. MAC systems are an alternative to discretionary access control (DAC) models, in which a central policy administrator sets a policy, and subjects (such as users or processes) are unable to change permissions on objects. MAC, as opposed to DAC, represents a fundamental shift in access control: system specified policy often becomes more important than user specified policy. In other words, a system administrator is able to set and maintain security properties that users cannot break, even if improper sharing is attempted. For instance, in a DAC system, if Alice wants to give everyone the ability to read a top secret file that she has access to, she can. In a MAC system, an administrator can restrict (via a MAC policy) the sharing of top secret data to only subjects that have top-secret clearance.

Additionally, SELinux attempts to enforce a MAC policy which implements the principle of least privilege on a system. If SELinux is running in strict mode, which attempts to regulate every program on a system, a policy must exist for each program (or class of programs) which fully specifies the access level and type the program needs to every object (files, sockets, devices, etc). Clearly, fully specifying least privilege requires a great number of policy statements, since all accesses must be explicitly allowed. In fact, there are more than 50,000 lines of policy in the selinux reference policy [80].

As complex as policy for a single reference monitor can be, there is a great deal of benefit in extending this concept amongst a group of distributed systems. The idea of a distributed reference monitor is to achieve the same reference monitor guarantees found in classical, single host systems, to a distributed environment. In "The Digital Distributed System Security Architecture", Gasser et. al. propose a system by which distributed reference monitors can be designed to satisfy TCSEC requirements up through the B1 designation [36]. However, this work is largely a design infrastructure, which enumerates the properties such a distributed reference monitor would need to have. These include a secured DNS infrastructure (which is quite detailed in the work), secure message channels, authentication, and mandatory access control. While we dispense with the DNS infrastructure in favor of alternate forms of machine identification, the rest of this thesis will detail a specific interpretation of this general design, with important modifications to reflect the modern computing landscape.

2.3 Virtualization

In contrast, virtualization allows system designers to take a different tack than reference monitoring. Instead of designing complex policies to completely mediate access between different objects in a system, virtualization gives security engineers the option to treat entire operating systems (running in virtual machines) as objects. This vastly simplifies the relationships between objects in a system, so that policy is simpler, easier to understand, and easier to verify. IBM first began to use virtualization in the 1970's, for time-sharing on large, expensive mainframe systems [49]. This provided a way to separate users and their jobs simply, in order to provide efficient resource sharing. However, as large mainframe hardware fell out of common use in the 80's and 90's, the case for virtualization weakened, as home computers became more prevalent and the increase in numbers of computers increased to the point where resource sharing became less important. Additionally, PC's began to lack the ability to implement virtualization primitives as hardware changed, so even assuming that the desire for virtualization stayed the same, the technical ability to implement true virtualization solutions fell by the wayside [76].

However, the benefits of virtualization eventually resulted in a resurgence of the technology. Convenience, portability, resource management, and security were only a few of the reasons that virtualization became irresistibly compelling to certain segments of the software industry [22]. However, problems with virtualizing hardware continued to persist, due to fundamental problems with virtualizing the i386 architecture [74].

In response, some open source virtualization applications such as Xen turned to paravirtualization, where some guest instructions are changed to run natively on hardware [10]. This has resulted in significant performance gains over traditional virtualization techniques, at the expense of needing to modify guest OS kernels [92]. Other approaches included creating host-based virtual machine monitors, which provide binary translations of guest instructions [94]. However, the necessary functionality for processor virtualization has been incorporated into many recent processors [4, 27]. As a result, many virtualization solutions have begun to support unmodified guests at greatly increased speed [10, 84, 94].

Therefore, it is natural that virtualization would become popular for providing increased resource usage, portability, and security at little performance cost. From a security standpoint, instead of reference monitor guarantees, virtualization is mostly concerned with providing separation. However, since virtualization has become popular so quickly, little work has been done to strongly analyze the security guarantees that virtualization purports to provide. What little work has been done in this space is not encouraging. As an example, take Tavis Ormundy's work applying fuzzers to VMMs. He found that every VMM was susceptible to some VMM compromise from a VM, and most were susceptible to a cross-vm compromise (meaning that a compromised VM could compromise another VM on the same host by exploiting some flaw in the separation provided between VMs) [73]. While these vulnerabilities only result from implementation errors, it demonstrates that even with a smaller codebase, VMMs still contain security-sensitive errors in practice.

Consequently, the sHype system was developed, which allows administrators who do not trust separation between VMs to specify which VMs can be run on a given machine, either alone or in conjunction with other VMs [78]. Administrators label VMs, and sHype policy specifies which combinations of VM labels can or cannot run on a given physical system. This even allows covert channels to be cut off. For instance, if a VM pegs processor usage to 100%, it may be sending a signal (via resource utilization) to another VM on the same system. In this way, administrators have a defense-in-depth solution to combat implementation issues in virtual machine monitors.

The other potential solution to implementation bugs is simply decreasing the size of the VMM itself. Ideally, as the size of the VMM codebase shrinks, the number of bugs decreases and the amount of carefully checked code increases. VMware has taken this approach with their new ESX 3i product, which contains a VMM and all required drivers in 31MB of code [51]. Only time will tell if this is actually a good defense against implementation errors, but it does demonstrate that as hardware evolves, the management that the VMM is forced to do decreases.

However, virtualization introduces more problems than simply inadequate separation. Virtualization has the capability to radically change the number and flexibility of managed operating systems, which creates new classes of problems. If virtual machines come up and down on a network quickly and frequently, keeping up with an appropriate network configuration is difficult. VMs can be replayed, causing flare-ups of previously contained attacks. VMs create a more diverse environment, which is more difficult to maintain. Data lifetime also changes in a virtual environment. This is just a small, quick sample of some extra problems introduced by virtualization, as outlined in [34, 38].

Additionally, hardware virtualization introduces additional security problems apart from separation and systems management. Because hardware does not currently support virtualization explicitly, designers have had to make a decision: allow guests to access hardware directly, or mediate access to the hardware devices. If access is mediated, the VMM can filter out ”unsafe” instructions for the VMs to execute (commands that access another VM’s memory, for example). However, this creates an additional context switch, and some technologies simply do not operate with reasonable performance under this model (3D video drivers, for example). Unfortunately, if guests are given direct access to devices, functionality such as DMA can be exploited to trivially and completely break virtulization’sseparation [69]. Some fixes for these sorts of issues have been proposed, which range from isolating drivers in their own VMs, to virtualizating all I/O opeations entirely [85, 69].

Additionally, interesting new trends in virtualization have exposed some additional security capabilities of virtualization. Take, for instance, the Overshadow system [23]. Since the VMM on a system controls page table access from the guest OS, the VMM can return encrypted pages to the OS, and cleartext pages to applications. This would allow an application to maintain the integrity of its memory pages in the face of a compromise of the virtualized operating system kernel. In general, the presence of a trusted, protected overseer in an operating environment opens up new avenues of thought relating to how to manage operating systems.

These traditional and non-traditional virtualization approaches trust the VMM and management domain explicitly. If the VMM or management is compromised, then all data to or from the VMM can be modified, thereby circumventing any useful security in the VMM. If we want more comprehensive security, we either need a way of establishing trust in the trusted computing base, or a way to prevent a malicious VMM from compromising the security of a VM. In this work, we attempt the former approach.

2.4 Trusted Computing

In order to find a basis for trusting a VMM, we turn to the concept of trusted computing here. Simply put, trusted computing is largely concerned with proving trust in components of a system, instead of being forced to trust them. In this scenario, we would like to be able to demonstrate trust in the VMM, instead of being forced to trust the VMM because no alternatives are available. Central to this idea is the idea of a trusted reporting mechanism, a way of reporting what the state of a given platform is. In practice, this is most often accomplished with the help of hardware, specifically, a Trusted Platform Module, or TPM. The TPM is a piece of hardware, designed to be tamper-resistant, that presents only a very narrow interface to the user [39]. We will discuss how it can be used as a trusted reporting device, even in the face of a software compromise. If we can utilize this hardware to verify the integrity of a VMM (and associated control plane), we should be able to get stronger guarantees about the separation of applications in VMs in the face of a wider variety of attacks.

The TPM is a low cost, underpowered chip that sits on a slow 10MHz bus on the motherboard. It has a few registers for storage, called platform configuration registers, or PCRs. The only operations that can be performed on these PCRs is to reset them or to extend them. Extending a PCR (with value p) with a value v means setting $p = SHA1(p + v)$. Thus a PCR cannot be set to a particular value, and the value of a PCR cannot be used to reconstruct what values it was extended with. The intuition here is that a user cannot arbitrarily set a PCR to a given value. The only way to do so (assuming that the one-way property and collision resistance of hashes holds) is to extend a PCR with the exact same sequence of measurements that was used to create the PCR state in the first place.

Additionally, the TPM contains persistent storage for a small number of public/private key pairs. In particular, the Storage Root Key (SRK), cannot be moved off the TPM, and since the TPM is tamper-resistant, this private key can only be used by the TPM. Consequently, barring physical attacks, the SRK private key should be impossible for an attacker to recover.

By combining these two functions, the TPM can create a cryptographic attestation of the current state of the TPM. This is called quoting, and simply takes the current PCRs, other state of the TPM, and returns a record of this state, signed with a key that can only be used by the PCR. In this way, a TPM can create a token that testifies to the state of the TPM.

Clearly, this would be much more useful if we could tie the state of the TPM to the state of the system itself. In other words, we would like to be able to say something useful about the state of the system, given the state of the TPM in the TPM's quote. Then, we could use these quotes of TPM state to prove properties about the state of the entire system. In order to accomplish this, an LSM was designed to extend a PCR every time a piece of code is loaded. In this way, if we extend a PCR with the hash of ever piece of code loaded, and we have a list of the code that is loaded, we can verify that the list of conforms to the actual code that was loaded. This LSM became the Integrity Measurement Architecture (IMA) [48].

The attestations generated by IMA can be very complicated, however. If IMA measures every piece of code loaded in a system (including libraries and modules), the total measurement of a system can be very large. More importantly, it can become difficult to reason about this attestation. Consequently, a policy-reduced version of IMA, PRIMA, was created [52]. PRIMA attempts to measure only the pieces of code that effect information flow, pursuant to a given system policy. For instance, if an object O was specified as high integrity by PRIMA policy, any code that interacts with O may be measured. In this way, the set of user level programs that don't effect trusted programs don't need to be measured, which vastly simplifies verification. In this way, PRIMA attempts to make attestation practical, while still preserving the measurements that are important to a system's security goals.

Unfortunately, IMA or PRIMA alone don't capture the full system state necessary. Take, for instance, the rootkit subvirt [58]. Subvirt is actually a rootkit hypervisor, which means it is installed below the operating system layer. Consequently, the operating system can't detect it. Since IMA is an LSM, IMA won't measure subvirt either, since LSMs run at the operating system layer. This demonstrates that we have to measure the whole application stack. The BIOS contains a piece of code (the core root of trust measurement, or CRTM) which extends a PCR with the hash of the BIOS. The BIOS then measures the bootloader, which measures the kernel or hypervisor to be loaded. After the attestation is created, the goal is that the attestation could be presented to a remote party, who would verify the state of the system. Measuring code this way would detect subvirt, since the BIOS would measure the bootloader, which would be modified by subvirt. Then, when a remote party receives the measurement list, they will be able to check that the bootloader was modified, and they can tell that the machine is running unapproved code. See 3.2 for a graphical illustration.

The Bear system from Dartmouth also identifies some of the problems addressed in this thesis [64]. They identify attacks against data, such as the replay of old dynamic data, that can impact the integrity of the system. In particular, it is interesting that the Bear attempts to classify different types of data in a system, and track it differently depending on its type. This design helps to remove the necessity of tracking data for the entire lifetime of the system. However, the only form of attestation is supported only by a remote *Security Admin.* Given that a remote party may or may not trust the *Security Admin*, this is a limiting design decision. Additionally, it negatively impacts the scalability of the solution, since trust between remote administrators has historically not scaled well (see current adoption of cross-domain authentication in Kerberos). Finally, the attestation in the Bear is not comprehensive enough to fully meet all of the requirements of this thesis. More on these attacks will be mentioned in Chapter 3 [42].

Another option for reducing the complexity and size of an attestation is the BIND system [82]. In BIND, the idea is to embed attestation annotations in code, so that a programmer can choose the exact portions of code that should be measured. By measuring these sections of code right before they are executed (as opposed to load time attestation, which measures code before it is loaded), BIND reduces the chance that the code being measured is compromised before it is actually run. Additionally, BIND attempts to tie data to code, so that a remote party can cryptographically verify that a certain portion of code generated a set of data. While this may prove useful for a very narrow set of measured code, it has some problems in the general case. Firstly, it requires programmers to identify and delineate portions of code that are ”critical” to measure. Secondly, this may (and probably should) require multiple measurements for larger pieces of code. Over a large set of programs, this would quickly grow to an unwieldy number of measurements. Additionally, this solution would require a fantastic amount of management, including making sure memory was protected by the TPM, verifying a large number of PCR states, creating tamper-resistant keys, and more. Consequently, the BIND solution solves a large number of problems for a very specific, tightly controlled environment, but is unlikely to scale to a more general problem. We investigate reducing much of this complexity in the next chapter.

It is worth nothing that the TPM is not tamper proof, only tamper resistant. There are facilities built into newer chips that set a tamper bit if tampering is detected, but these are not infallible. In the presence of a sophisticated adversary with detailed knowledge of a TPM, the SRK may not be safe if the attacker can extract the key bits from the chip physically. While no attack on the TPM has yet been reported, a history of devices indicates that non-tamperproof devices are likely to be compromised eventually [8].

As an alternative, secure co-processors, with a much higher tamper-proofness, can be used [87, 30]. The guarantees and speed provided by these co-processors are much better than the TPM provides, but at at least 4 orders of magnitude cost increase. Additionally, many of these are currently on a very restricted distribution schedule.

The other alternative is to do away with a hardware root of trust altogether. The Pioneer system, in contrast, relies on timing instruction execution to ensure that programs are behaving as expected [81]. In general, a piece of code known as the dispatcher establishes a dynamic root of trust. The root of trust then executes code, and by doing a timing analysis, verifies that it runs untampered. This requires a trustworthy channel to set up a root of trust and communicate results. It also requires that the exact state of the hardware be known. Additionally, the approach cannot always, deterministically, detect compromises or tampering. In short, Pioneer provides probabilistic security when environments are very well known, but we wish to solve a more general problem, and it seems that these restrictions are fundamental to the Pioneer approach and not simply implementation shortcomings.

2.5 Trusted Systems

Ideally, we would like to use trusted computing to prove that a reference monitor exists on a remote system and is enforcing the appropriate policies. However, we would like to use virtualization to separate applications from each other, in order to have a greater degree of trust in both the policies governing inter-application communication and the separation of applications. In short, we would ideally be able to use virtualization to support applications, and trusted computing to support the security enforcement for the applications.

This idea lends itself most naturally to the distributed grid computing model. In general, this is a much simpler model than arbitrary distributed services, as instead of support multi-user systems with multiple general applications, distributed grid computing often deals with a set of instances of applications which must communicate with each other. Trusted virtualization clearly facilitates this sort of computational model, as it isolates these applications from other applications/users while providing a trusted computing base upon which to perform distributed computations.

As can be expected, a number of projects have dealt with this topic; perhaps the most comprehensive of these is Lohr et.al [62, 93]. Lohr attempts to provide a generic framework for trusted grid computing, independent of implementation-specific details. In this framework, an attestation service, storage service, and grid service form the TCB which provide a platform for grid computing.

The framework described in Lohr is one of the first works in trusted virtualization that specifically attempts to address the asymetry in trust requirements between the users and providers of resources in trusted, virtualized systems. The framework attempts to provide grounds for satisfying this asymetry, but does not explore how this would be done in practice. Consequently, this work provides an alternate framework and implementation in chapter 4.

Perhaps one of the most interesting decisions in Lohr, et. al., is to use offline attestations instead of traditional attestations. Normally, an entity requesting an attestation provides a nonce, which the attesting entity includes in an attestation. This ensures freshness, without which, an attestation is largely worthless. This work generates attestations and stores them offline, certifying that the system is in state S. Later, when a user wants to submit a job, he uses a key sealed to S to encrypt all sensitive information. Thus if the offline attestation was valid, the job will decrypt successfully. Otherwise, it will not, but no secrets will be improperly revealed. This approach drastically speeds up the time taken to submit trusted requests, but has a major shortcoming: the hosting platform cannot execute any unmeasured code between the submission of the offline attestation and the acceptance of jobs. This restricts the usage of this model to hosting platforms that are mostly static. While this may be a good assumption in the trusted virtualized grid computing space, it does not translate immediately well into arbitrary trusted distributed applications.

Trusted virtualized grid computing also adds management complexity where there may have been less before. The most popular grid computing software currently, BOINC, allows all distributed jobs access to the network in order to submit and retrieve jobs [5]. However, in the general case, jobs may require much more communication. Additionally, it may be highly desirable to separate the networks each set of applications uses to communicate. In a virtualized environment, this challenge is even more critical. In provisioning 100 machines, the cost and speed of the provisioning operation often implies greater care given to network design than does the 10 minutes required to spawn 100 virtual machines clones. Consequently, some flexible way of creating, maintaining, and designing virtual networks is required. SODA (service-on-demand architecture), attempts to solve this problem by approaching this issue as a provisioning problem: by applying provisioning solutions to the issue, they hope to make large scale hosted applications more flexible and manageable. Unfortunately, to tackle such a large problem, they require buy-in from many levels of the software stack: OS, middleware, network. In practice, it may be difficult to just drop existing solutions/OSes into this system, and still achieve the original goals.

Even with more flexible virtual network provisioning, the tools to enforce rich security policies between applications in a trusted virtualization environment were lacking. For instance, consider the case where Goldman-Sachs is running a virtual machine to perform proprietary analysis of stock market trends. Goldman-Sachs must ensure that data from this virtual machine does not leak, even if the VM becomes compromised. But if another investing company, BFP, has a virtual machine running on the same server, and compromises the Goldman-Sachs VM, then they can transmit information via covert channel from VM to VM [60] (for instance, processor utilization, network utilization, etc.). sHype was introduced to the Xen VMM by IBM and helps to solve this problem, among others [78]. sHype allows an admin to apply MAC policies to inter-VM communication on the same platform by labeling VMs and writing policy to govern their interaction. Additionally, sHype allows administrators to specify which VMs are allowed to access the same physical resources simultaneously. As a result, sHype helps to forward trusted virtualization by providing enforcement of policies that do not rely on strict separation between VMs on the same host.

Along these lines, platforms for more general secure distributed services was created. In particular, the Terra system kicked off a series of research in the trusted distributed application space [33]. In Terra, the goal was to produce a system in which applications can be verified and/or isolated, in order to run trustworthy applications. Specifically, the unchanging portion of the application's disk space is measured, and used to ensure integrity. However, this model applies poorly in the case where an application's disk space may change arbitrarily, or where the security-sensitive information changes. Additionally, little consideration was given to verifying the host platform, which ultimately can effect the security of the VMs. Terra is particularly important, however, as one of the first works to seriously consider how to attest to VMs running applications in a very general setting.

The natural extension to Terra is the move from singular, verifiable applications to cooperative, trusted groups of applications. Where Terra was mostly concerned with verification and protection of a single virtual machine, the work in this area began to move towards building trust between trustworthy components in order to form cooperative groups (coalitions). In particular, a series of papers from IBM attempted to group machines (and later virtual machines) into coalitions in order to enable trustworthy cooperative computing.

This series of work relies on a trustworthy, verifiable base on which to run applications. In the Trusted Platform on Demand (TPoD) work, this verifiable base consists of the ”standard” attestation chain through the Linux kernel, after which applications are run and measured [65]. When a remote party wishes to verify the code on a TPoD, an identity CA and validation entity are used to ultimately form the basis of trust.

The Shamon, NetTop, Trusted Virtual Domains, and Trusted Virtual Data Center (TVDC) works expanded on this idea by separating applications into virtual machines [66, 57, 11, 70]. In TPoD, significant complexity is introduced when a verifying entity receives the policy in use on the TPoD. Accurate analysis of the SELinux policy that dictates interaction between programs is difficult. While some works have shown that it is possible to ask questions of an SELinux policy in a reasonable about of time [45], it is difficult to know that all relevant questions have been asked. However, when applications are separated in virtual machines, the channels available for sharing are much more limited, and their interactions are much easier to understand.

Additionally, the Shamon and TVDC papers attempt to extend MAC policies to a distributed setting. In other words, they attempt to ensure that MAC policies (such as SELinux), are enforced not only on the systems on which they are installed, but also between select remote systems. For instance, a MAC policy in a distributed setting may ensure not only that program A cannot access data generated by program B, but also that program B cannot output data on a network that is consumed by program A on a remote host. This idea of trustworthy, distributed coalitions is referred to as a distributed reference monitor, since it attempts to provide reference monitor guarantees in a distributed setting. Chapter 3 will discuss these systems in greater detail, as they form a direct basis for the work in this thesis.

A new direction is emerging in trusted computing in response to the complexity introduced by these solutions. Trying to attest to and measure all code on a system is a very complex task, but a seemingly necessary one. If a program is not measured, what is to say that it cannot tamper with the measured and trusted ones? Policy analysis has historically been quite complex, so may not be a good solution in an arbitrary case. Fortunately, new processor technology has allowed for the execution of small, isolated environments which are protected in memory from even the OS/Ring 0 [3]. The Flicker work is a natural and direct implementation of this processor feature [68]. Flicker simply takes the skinit instruction and executes an application in the resulting measured, isolated execution environment. In this way, trusted applications can execute in a measured environment independent of the rest of the system state. The downside is that applications (or their delivery mechanisms) have to be modified for Flicker's execution environment, and the isolated execution environment makes sharing between protected programs difficult. However, the required modifications to Flicker programs are reasonably simple, and provide a type of "drop in and attest" application delivery that has the potential to make trusted computing a great deal simpler than the current complex solutions.

2.5.1 Shortcomings

Unfortunately, these works suffer from serious shortcomings that motivate the beginning of this thesis. Perhaps most importantly, the trust established between systems in all of these previous works has not been enough to stop an attacker with physical (but not circuit-level) access to the participating machines. For instance, attackers can reboot machines into a malicious operating system and steal data in many solutions. Since an attacker with physical access is our threat model, this is not sufficient to tackle the problems laid out in chapter 1. For instance, if applications or virtual machines are not static, an administrator is trusted (without basis) to make necessary changes. If an attacker either reinstalls the system so that he is the administrator, or changes the administrators credentials by rebooting the machine and modifying the authentication configuration, he can make arbitrary changes in these systems that will compromise the information flow guarantees that we wish to establish.

Additionally, the these systems do not monitor the state of files in the hosting OS (or VMM). Given that important security properties of systems are determined by the contents of files in every major OS, this is a major problem. Even if these files are encrypted, new work has shown that it may be possible to recover these encryption keys from RAM, and use them to modify files on the hard disk [21, 42]. This thesis will explain how many of these problems can be fixed, as well as other problems (such as VM management) which will be introduced in chapter 3 and 4.

Chapter 3

Shamon

3.1 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) [40] 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 [71, 82, 79, 52, 50, 64]. 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 [12], 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 [83]. 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 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 [24, 21, 42], 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 *Shamon* project [66] seeks to build integrity-ensured distributed systems based on the idea of a system-wide reference monitor. A *Shamon* system is built upon of a collection of virtual machine monitors (VMMs) that launch virtual machines on behalf of authorized users. In these virtual machines, any application can run, and the *Shamon* enforces the application's security

---

[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) [40]. Therefore, subsequent *unsealing* is only possible by that same TPM, hardware, and software.

requirements. The hosts mutually vet the integrity of each others' *Shamon* trusted computing bases to ensure that each satisfies the requirements for a true reference monitor [7]: complete mediation of security-sensitive operations; tamper-protection of itself; and verifiability that it enforces the intended security goals. For the *Shamon* system, we call the trusted computing base on a given host the *Shamon Core*, or *sCore* for short, and it consists of the Xen hypervisor and its privileged VM, domain 0. Because access to the *sCore* is tightly controlled and integrity guarantees are sustained over time, users and applications can use *Shamon* hosts to run virtual machines with confidence that their security goals are enforced. Note that this says nothing about the integrity of the user's VMs themselves, but guarantees the system upon which that VM is running has not been compromised. This Chapter deals solely with creating trustworthy sCore: secure management of the virtual machines that will be run on the sCore is discussed in Chapter 4.

Consider a motivating use of a *Shamon* system at a university. Alice the graduate student wakes up at noon and goes to class. Alice joins a live collaboration of class participants by starting a VM in her classroom. At lunch, she surfs the Internet and exchanges personal communication within a VM (perhaps different) at the local student union. After lunch, she heads to the laboratory, opens a work VM containing specialized laboratory software, and performs research while sharing data with the other graduate students. At the end of the day, she meets with her advisor, shares summary data, and exchanges results. She heads home and plays a massively multiplayer game with thousands of other gamers until dawn over the Internet. The ”roles” of Alice's computing environment and the environments in which she interacts evolve constantly; from class participant, personal communication, researcher, advisee, and gamer. What is different in the *Shamon* is that her relationship with the hardware on which she runs her potentially many VMs; to Alice, the hardware is simply a physical resource like a table or chair.

The challenges of this vision go far beyond the previously studied hardware-assisted, secure boot procedures in integrity measurement [71, 82, 79]. *Shamon* must: (1) establish the integrity of the installed *sCore* ; (2) ensure that the integrity of the on-disk *sCore* image is maintained; (3) ensure that the integrity of the running *sCore* is maintained; and (4) ensure that VM's depending on the *Shamon* only boot on integrity-guaranteed *sCore*s (addressed in chapter 4). A failure to completely address any one of these aspects will leave the *Shamon*, and ultimately Alice's VMs running on them, vulnerable.

We address the above challenges by using an authoritative *root of trust installation* (ROTI, pronounced "rŏtē") to drive integrity measurement. To simplify, statements of software integrity must originate from a well-known ROTI image, such as installation media. The system boot mechanism must verify that dynamic data (e.g., system-specific configuration data) originated from a ROTI. Because all data access and executing code are traceable to a ROTI image, the integrity of the booted system can be assured. Further, we provide mechanisms complementary to the ROTI approach to protect system secrets across reboots yet still traces their integrity back to a ROTI. *This represents a departure from traditional integrity measurement, as we seek to measure that the system configuration and software emanates from a trusted source (the installer) rather than extracting measurements of fixed, known states.* This simplifies the process of integrity measurement and allows a diversity of system configurations without undue administrative overhead.

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 chapter, we consider how a *root of trust install* can be used to establish and sustain integrity for a distributed security architecture, called a Shared 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 3.2 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 3.3 a broad design philosophy and outline our working implementation of the *sCore* ROTI in Section 3.4. In Section 3.5 we present an assessment of the costs associated with ROTI installation and subsequent integrity measurement, and conclude in Section 3.6.

3.2 Background

The *Shamon* project [54] 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 [7]. Previous integrity measurement approaches [71, 82, 79] 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.

3.2.1 *Shamon* System

We begin by defining a *Shamon* system, shown in Figure 3.1. 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 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.

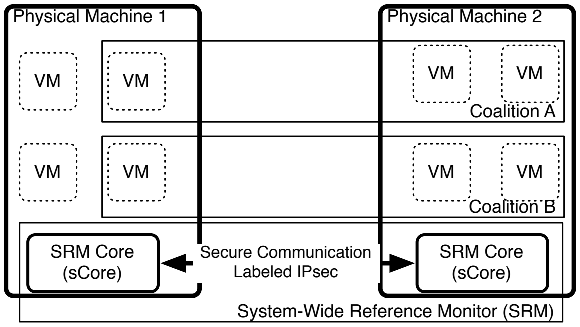

Fig. 3.1. A System-Wide Reference Monitor (*Shamon*) system: Coalitions of virtual machines (VMs) may run across physical machines, but the *sCore* components collaborate to form a *Shamon* that enforces the VM communication requirements within a coalition.

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.*

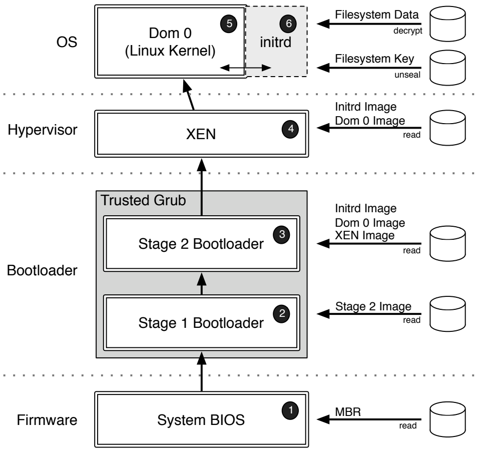

Fig. 3.2. The integrity measurement in the *sCore* VM's boot sequence.

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* [16, 15, 55, 61] 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*.

3.2.2 *Shamon* Integrity Measurement

We now examine the application of traditional integrity measurement to an individual *sCore*. While a variety of approaches have been proposed [82, 71], we apply the approach that appears most appropriate for our system and software architecture, Trusted Platform on Demand [65] (TPoD) for integrity measurement of the *sCore*'s privileged virtual machine (Dom 0, in Figure 3.2) and the Linux Integrity Measurement Architecture [79] (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 [40] 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 [9] 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 3.2, integrity measurement of the *sCore* boot process (up to the Dom 0 Linux kernel) consists of a deterministic sequence of well-defined load operations outlined 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 privileged 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 [48, 79] 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.

3.2.3 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 [44].

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 [24]. 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.

3.3 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 3.3.1), outline the key design tasks for constructing high integrity *sCore* (Section 3.3.2), and show how this design will justify *sCore* integrity (Section 3.3.3).

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 3.5, 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.

3.3.2 *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 3.4.

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 traceable to the ROTI installation.

Booting: The booting system uses statements generated at installation by the ROTI to generate integrity measurements that bind the specific 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

---

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

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.

3.3.3 *sCore* Integrity

We show how ROTI-based integrity measurement approximates classical integrity, in this case Biba integrity [12]. 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 3.4.2), 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 [83].

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

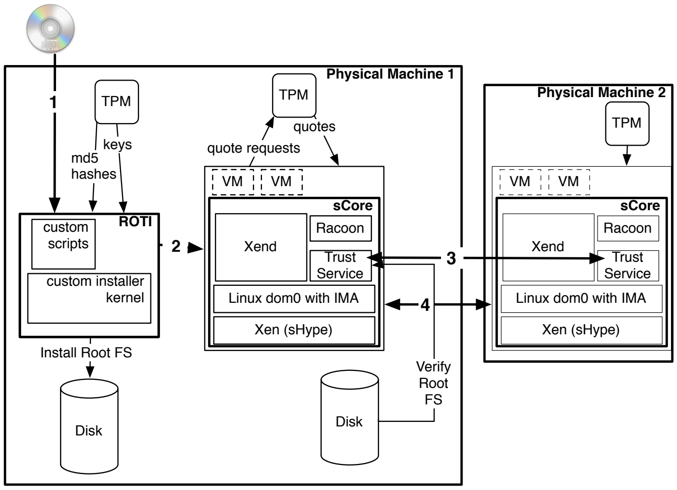

Fig. 3.3. *sCore*'s lifecycle: (1) a ROTI installs the *sCore* by generating a verifiable root filesystem and creating TPM keys; (2) the *sCore* boots verifying its own root filesystem's integrity; (3) two *sCore* attempt to form a *Shamon* , each trust service makes a quote request to generate an attestation; and (4) a successful *Shamon* join results in an IPsec tunnel between the two systems.

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 3.4). 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.

3.4 Implementation

Our prototype *sCore* is shown in Figure 3.3. 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 [78] and SELinux Labeled IPsec [53] to authorize

---

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

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

Fig. 3.4. **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).

inter-VM communications for enforcing *Shamon* MAC policies [66]. We have extended this Linux kernel with the Integrity Measurement Architecture (IMA) patch [48, 79]. 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 3.4. 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 3.4 also shows which software was modified for the *sCore*. The 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 [46].

3.4.1 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.

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

---

[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.

to vulnerabilities. However, due to the large number of complications with respect to the plethora of patching stratagems 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`).

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*.

3.4.2 Booting the *sCore*

As the *sCore* is booted, the individual stages (see Figure 3.2) 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 [48, 79]. 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 3.2.3). 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.

3.4.3 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) [71]. 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 3.4 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 our *trust service* which performs mutual attestations with other *sCore* to build *Shamon*. The other programs initialize the system

---

[5] However, methods to authenticate DHCP have been proposed [29].

(`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 [79]), 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 [66]: (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 bootcycle, 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.

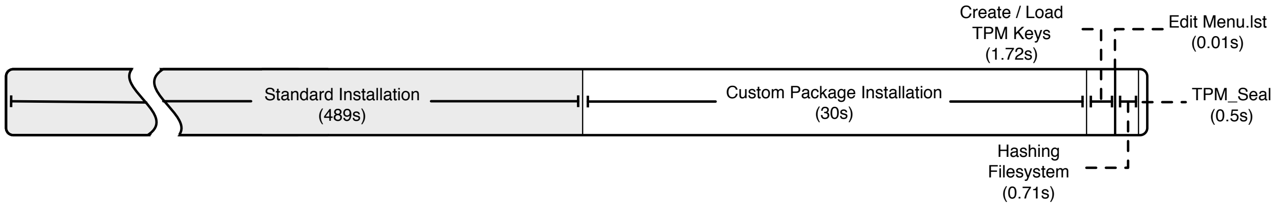

Fig. 3.5. A performance breakdown of the major tasks in the install phase. The majority of the time is consumed by the standard install.

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*.

3.5 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 3.4.1. 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 3.5, 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 [79], 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 hashes 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 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.

3.6 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.

Chapter 4

Trusted Virtual Machine Infrastructure - TVMI

4.1 Introduction

In Chapter 3 we introduced a scheme to create remotely verifiable VMMs and control infrastructure capable of enforcing complex MAC policies to form a reference monitor. However, little was said about how the system would actually be used. For instance, in order to protect the sanctity of data on these Shamon systems, all user accounts needed to be disabled to prevent any tampering of the system by remote administrators or users. Consequently, these systems have no way to start virtual machines since there is no daemon capable of servicing remote virtualization requests by default. Since there are no users in the system, no such service can be manually started or installed. Moreover, whatever virtual machine controls we include must be quite flexible to fully take advantage of the trust relationships that the Shamon architecture supports, while **not compromising the security guarantees that the sCore provide**.

Given that we have created a platform in Chapter 3 truly capable of providing stringent security guarantees, the natural next step is to create a management infrastructure capable of interacting with the sCore which comprise the TCB of each physical machine. Since the goal of the Shamon is an Internet-scale infrastructure for trusted virtualization, we must look outside the obvious solutions in order to find something scalable which works in the Internet model. Having disparate authentication, policy, and mechanisms for each machine capable of hosting virtual machines will not work at this scale. Instead, some method of authentication and authorization that is significantly more efficient than pairwise negotiation is required.

However, as we have discussed in chapter 2, enabling trustworthy virtual machine controls is more complex than simple access control. We need to enable different usage models than traditional systems, under tighter constraints. Take, for instance, a typical computer science lab setting. Often, there are a number of systems available for personal use, as well as additional desktop or server machines scattered around for research. As research projects change and data is collected, these machines are often wiped clean, followed by a re-installation of the system's OS and configuration for a particular research purpose. The purpose of machines changes so frequently, and the experimental nature of many research projects makes security (especially isolation) so important that it represents a good example of a system where a usable shared reference monitor that provides VMM services is required. If the users in this system can easily create, start, stop, and migrate virtual machines around platforms that they can verify, they can gain a great deal in security, flexibility, and utilization. More specifically, users know that when they run virtual machines, certain security goals will be enforced on those VMs.

Additionally, part of the strength of the general shared reference monitor approach is that it is capable of supporting coalitions of virtual machines. These coalitions use guarantees provided by the sCore to establish mutual trust in each other, which enables secure operations that were not previously possible. Take for example, trusted distributed computation. With a trusted coalition, distributed computation jobs don't have to be repeated, they can simply be handed off to a trusted member of a coalition. Since the remote node is part of the coalition and thus has been shown to be enforcing the coalition's policy, it can be trusted to perform the distributed computation in many instances. It is important for our virtual machine control infrastructure to support meaningful coalitions as a central motivating example of the Shamon work as a whole.

This work is also concerned with identifying the major players involved with trusted, distributed virtualization. Each of the entities in the Shamon system has different security goals and concerns. Moreover, some of these goals are in opposition to each other. For instance, members of a coalition may want all the resources possible, whereas owners of physical machines may not want to give out all of these resources. In general, we wish to say something about the motivations of the parties involved, and devise a system whereby each party is capable of achieving the guarantees it requires. This is particularly important in a wide area network setting, such as the Internet. Administrators of machines may not (and probably will not) know each entity that wishes to use their physical resources for running virtual machines. We show that a trust model which authenticates users as belonging to a larger group makes more sense in this setting. We also discuss the necessary infrastructure required to support this model, and show its feasibility.

To summarize, we need to support the following items in order to realize the vision of a Internet-scale trusted, distributed, reference monitor comprised of trustworthy sCore:

- Standard virtual machine operations such as creating, deleting, starting, stopping, and migrating VMs
- Coalition operations such as creating, joining, and leaving a coalition
- Scalable authentication and authorization for owners and users of physical resources

In order to accommodate these usage models, we create a Trusted Virtual Machine Infrastructure (TVMI), by which users can control their virtual machines. By enabling the creating and starting of virtual machines on different secure, userless VMM platforms (the sCore), we show that the shared reference monitor paradigm can both be made usable and general enough to be used in many applications. In order to enable a wider variety of usage patterns, the TVMI must enable useful virtualization control operations while preserving the important security guarantees of the Shamon system. Most importantly, our design ensures that every party involved is free to pursue their own self interest, without compromising another party's security requirements.

Additionally, we show that it is possible to prove certain security guarantees about the TVMI in the face of a network attacker capable of interleaving multiple protocol instances, sniffing all network traffic, and interjecting arbitrary data onto the network. This is done with formal protocol verification tools, in order to ensure that every party in the system achieves the guarantees it needs. In doing so, we create a system that provably provides both security and provable guarantees to the parties involved.

The TVMI forms an important part of the Shamon design as a whole: the TVMI is the portion which will actually provide the services which users care about. The sCore should remain in the background, providing services silently, while the TVMI does the difficult work of accommodating complex requests while maintaining the security requirements of the users in the system.

In this chapter, we attempt to address the concerns discussed above in the construction of the TVMI. We give a design and description of the TVMI in this chapter, while implementation and validation details are reserved for chapter 5. Specifically, we give an overview of the TVMI design in section 4.2. We then detail how this design was achieved by exploring the design requirements and goals of the TVMI in section 4.3. We then describe, in detail, the design of a protocol to accomplish these goals in section 4.4.

4.2 Overview

Figure 4.1 gives an overview of our solution for secure virtual machine management, the TVMI. Section 4.3 will give a more detailed overview regarding how this design was achieved, and section 4.4 will detail the contents of messages that are passed, but this section aims to give a description of the system as a whole.

The fundamental idea behind the TVMI is that there are two management entities in the system. The MA, or machine authority, is responsible for mediating access to the computing resources provided by the physical machines. The CA, or coalition authority, is responsible for protecting the users data (in the form of VMs). Between each other, the CA and the MA

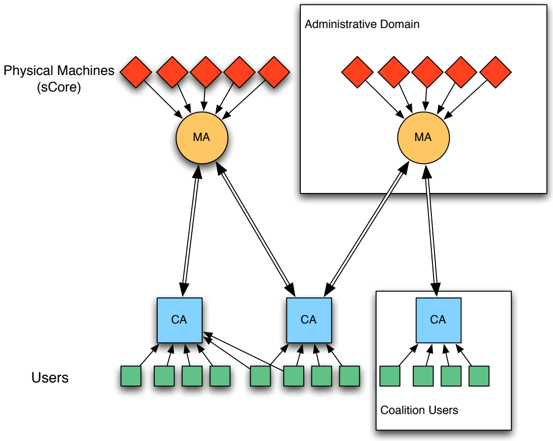

Fig. 4.1. Picture of the full TVMI infrastructure

negotiate usage of the resources the MA has (physical machines) and the resources that the CA wants (machines to run VMs).

The CA's role in the TVMI is to manage the coalition and act on the user's behalf in controlling the execution of virtual machines on physical hosts. When the user wants to run a virtual machine (or perform some other action on a coalition VM), the CA acts as his agent to make sure the request is carried out efficiently and securely. Therefore, the CA must manage authentication and authorization for users to access the virtual machines in the coalition. For instance, the CA must maintain a record of which users are allowed to perform which operations on which virtual machines. Once a user submits an authorized request to perform some action (starting a VM, for instance), the CA negotiates with an MA for use of a physical machine (an sCore, as described in chapter 3). However, the CA does not wish to run a user's VM on just any machine. What if the physical machine ran a rootkit that copies any VM it receives to an attacker's machine? Doing so would violate the CA's goal of protecting a user's data. Consequently, the CA must also ensure that coalition security goals are achieved on the physical machines it uses to fulfill user requests. In order to do so, the CA first requests a physical resource

S from the MA. The CA then asks for an attestation from S, and checks to see if the state of software on S is compatible with the coalition's security goals. By performing the attestation check itself, the CA ensures that the final executor of the user's request conforms to the user's security needs. This design enables the CA to act as an agent for the user, since the CA takes user requests and fulfills them, according to a pre-defined set of security requirements.

The MA, on the other hand, is unconcerned with protecting user data. Instead, the MA acts as an agent for the owner of the physical machines, ensuring that physical resources are only given out to coalitions according to a policy that the machine owner dictates. Practically, this is reasonably simple. The sCore register with the MA when they boot up and establish a shared key with the MA. The MA then receives requests for resources from the CA, and determines which sCore should be used. The MA then returns a public key to the CA, which allows it to verify the sCore's attestation (as described above), so that the CA can effectively enforce the coalition's security goals. The MA also includes an authorization token in its reply to the CA, which the CA will present to the sCore to prove that it is allowed to submit the approved request. In short, the MA provides a central access point for resource management, public key infrastructure for attestation keys, and mediation of physical resources.

4.3 Requirements

Before designing the TVMI, we must understand what a trusted virtual machine manager must be able to accomplish. In general, the TVMI must be able to fulfill security and usability goals for multiple participants in the system and must be able to do so when faced with an attacker who has physical access to all physical machines, including the ability to reboot, toggle power, insert media and foreign devices, etc. We do not consider an attacker capable of circumventing the physical security of the TPM, however, by means of circuit-level tampering.

The first participant is the end user who wishes to perform some action on a virtual machine. Clearly, the user must have some mechanism for performing actions on his virtual machines which are being run by the sCore in the Shamon. Just as with a VMM that is not part of a distributed reference monitor, the user must be able to start, stop, suspend, and migrate a VM. If a user can perform these actions, a user does not give up any flexibility in the Shamon system as opposed to basic remote virtual machine management, as these are the primitive operations performed on virtual machines today (other operations, such as load balancing, are simple compositions of these primitives). Ultimately, the user wishes to run his virtual machines on some type of secure VMM or trusted computing base (TCB).

However, to completely realize the potential of the design in Chapter 3, a user must be able to participate in a coalition of virtual machines. Without this, a user can trust a given sCore, but is unable to build meaningful trust between virtual machines in a collaborative group (coalition). This functionality is important to our design: if applications run in virtual machines, and applications need to cooperate to provide needed functionality (as recent trends in distributed/Web 2.0 applications have demonstrated), virtual machines in the TVMI need to be able to form trusted, cooperative groups, which we call coalitions. The coalitions will then be used to form distributed, transitive, trust relationships.

In general, the idea is that only VMs that conform to specific coalition policies will be allowed to join the coalition. For instance, if the coalition policy for VMs is that each VM must be running an SELinux strict policy, two VMs in the coalition would be able to assume the other was running the SELinux strict policy. This precludes the need to perform expensive pairwise verifications with each and every VM they interact with for each interaction that requires this property.

Consequently, we wish to support a user joining a VM to a coalition, listing all possible coalitions, and confirming another VM's membership in a coalition. By doing so, we support the basic coalition primitives, and allow useful trust to be constructed between virtualized applications. This allows a member in a coalition to see which VMs are also members and thus interact with them different than if they were untrusted.

But in order to fully understand the security requirements that these operations require, we must first consider the goals of each entity that participates in the TVMI. First, we consider the end user, who wishes to control virtual machines. A user has somewhat different security requirements in a trusted virtualized environment than a traditional virtualized environment. In particular, the user must be able to verify a number of things about the sCore on which his VMs run. Firstly, he must ensure that it is not running any malicious code when his VM starts. The user must also be able to verify that the sCore contains a policy for network communications compatible with the user's goals. Additionally, the sCore must make sure all of these goals continue to satisfied if the VM migrates. In other words, the VM must not "leak" to an unacceptable environment or run without being directed by the user.

This is a user's most fundamental security requirement: VM data must not leak to a party except that which the user approves. This means that the VM itself must not run or be transferred to an unapproved party, nor may the VM itself leak data as it runs.

The other fundamental entity in the system is the owner of the physical machine(s). Here, the security requirement is simple: the owner of the sCore wants to ensure that the only jobs that run on the sCore are jobs that have been "approved" by him, in some way. In other words, every job that runs on the sCore must be subject to some policy check, as designated by the machine owner. This is critical in an Internet-scale open system, as the owner of a set of physical machines wants to prevent abuse of the physical resources being provided.

4.3.1 TVMI Requirements

The Shamon architecture attempts to provide much more functionality than a single user's management of their own virtual machine, however. Users should be able to have a large number of virtual machines, for one. If users want to migrate these virtual machines with reasonable performance, this may imply some centralized, managed storage. Migration between sCore can occur without networked storage, but this means the entire VM image will have to be migrated from system to system, which can be prohibitively time consuming between remote networks.

Additionally, users would like to join coalitions of virtual machines, where all member of the coalition conform to some set of security goals. In some way, users will have to determine what goals which sets of virtual machines will enforce and which coalitions they will belong to. In some way, the coalition as a group will have to decide on these policies and group membership.

Given that much of the benefit from the Shamon architecture comes from location-independent security, the TVMI should be able to support the preservation of security goals on whichever sCore a user's VM executes. For instance, if a user moves his VM from sCore A to sCore B, whatever management architecture the user used to move the VM should provide the necessary tools to ensure that sCore B is still enforcing a super-set of the security goals the user desires.

Additionally, users will need to be able to manage their virtual machines in the Shamon. If a user starts a web server VM and then forgets about it, he will need to be able to locate it later. Since it may migrate without him knowing, he will need some way of finding it that is updated as migration events occur. Additionally, if a user has a large number of VMs, it may be difficult to track the state of all VMs in the Shamon when the user may or may not be constantly monitoring the Shamon (a user on a laptop, for example, is likely to not always be able to receive real time updates).

While these problems may be solvable in a distributed manner, introducing a centralized Coalition Authority (or CA) provides a great number of features with vastly reduced complexity as opposed to a fully distributed design. For instance, if each user must be aware of each sCore, and each VM must be able to make an authentication and authorization decision about each user in the universe, complexity must increase exponentially. Including a centralized authority for a coalition is a design decision we make in order to enable the management of virtual machines in a coalition in a relatively simple way. Additionally, introducing the CA allows a coalition to be managed by one entity, even when many users may have VMs in a coalition. Instead of each user managing their VMs, the CA manages the VMs in a coalition that many users belong to (and users may belong to many coalitions).

The CA is responsible for administration and protection of virtual machines in a coalition on behalf of their users. It must provide authorization and authentication appropriately, so that a user may not access another user's virtual machine if it is not permitted. In short, the CA must enforce the user's security requirements on the user's behalf. A user's VM must not leak (either the disk image of the VM or data from the VM when it is executing) except in accordance with security goals associated with the VM. Additionally, a user's credentials must be not be leaked by the CA, as this could lead to unauthorized access by another party.

Many of these requirements exist to enable multi-party trusted communication as a part of a coalition. The Shamon enables coalitions capable of establishing multi-party, distributed trust. Consequently, a user must be able to include his virtual machines in a coalition, and ensure that coalition goals are enforced over all members. For instance, if a goal of the coalition is that no top-secret data be allowed outside the coalition, then the user must have some way of knowing that all members of the coalition are actually subject to this requirement. Otherwise, the value of a coalition is severely limited, to the point of obsolescence.

The CA represents a general design decision for the TVMI - we choose to delegate a user's requirements to a centralized entity in order to obtain easier management, easier coalition formation, and the ability to continually ensure that security goals are being met. In the same vein, we introduce a centralized point of delegation for the owners of physical systems: the Machine Authority, or MA. The idea behind the MA is that the owners of machines in a single administrative domain would delegate resource allocation decisions to the MA. Without the MA, coalitions or users would have to know about the constantly changing landscape of physical machines in order to allocate resources for running VMs. Moreover, authorization and authentication for every coalition would have to be managed at every physical machine in some fashion. This situation is undesirable, so the MA is introduced as a proxy to an administrative domain of resources. In this way, each administrative domain has a single policy decision point for

The MA provides a central point of contact for CA's wishing to use pools of resources without needing to be able to locate each and every sCore. Therefore, the MA handles authorization and authentication of coalitions, for the purpose of managing resources for coalitions. Additionally, the MA can optionally operate as a trusted keyserver for each sCore's public endorsement key (EK). This allows an MA to be trusted to describe and proscribe resources for the CA, while allowing the CA to verify that the physical machine meets its requirement. This will be discussed in greater detail in section 4.4.

The inclusion of the MA is particularly important, as the goals of users may conflict directly with the goals of sCore administrators. Users generally wish to use all available resources, whereas people who own machines may wish to reserve portions of computing power for personal use. Since users "own" data and computing jobs, whereas MA's "own" machines, by introducing these entities into the system, we capture the sometimes fundamental conflict between resource providers and resource consumers. Our design must accommodate these competing goals.

Additionally, the concept of an MA allows much better scalability for our system. We envision that owners of domains of machines will rely on a small set of MAs to manage these resources in the Shamon architecture. The CAs can then manage a much smaller number of resource pools (MAs), instead of attempting to maintain a directory of all possible sCore, their current utilization, and their public keys.

To summarize, we briefly summarize the requirements of each entity in the full TVMI, which were derived from the basic requirements discussed at the beginning of this section:

- User: must have username/password remain secret (except to CA), user's VM must not leak except according to user-defined policy

- CA: must not leak user VM except according to user-defined policy to attested sCore, must keep the credentials it uses for authorization to the MA secret
- MA: resources on sCore are only given with the MA's approval
- sCore: must be able to determine if a job is approved by the MA

4.3.2 Completeness

It may not be immediately clear why these are the full and complete set of requirements for the entities in the TVMI. Given that we will demonstrate that these requirements are met by the TVMI, it is important that the properties that we are proving are sufficient to ensure the correct operation of our protocols. Fortunately, each entity has a narrowly defined set of interests. By demonstrating that these interests line up with the requirements above, we informally suggest that this list of requirements is sufficient.

- The user, ultimately, cares about his data. Since his data resides solely in his VMs in this system, we must simply ensure that his data cannot leak except with his permission. We deal with run-time leakage of data in Chapter 3, so the TVMI must ensure that VMs cannot run on systems that do not meet user defined policy. Additionally, the user wants to ensure that only he can make requests, so his password must stay secret.
- The CA simply acts as an agent for the user, so his requirements are similar. The CA must make sure that VMs do not leak (except as requested). Although the CAs are managing multiple users, the goals for each individual VM is the same - do not leak to an improper platform (sCore).
- The MA only wants to ensure that all requests that are sent to the sCore are first considered and approved by the MA. If this happens, the MA is controlling which jobs actually run on the sCore, and is thus controlling the usage of his physical machines. Much as the user

Variable	Description
U	The end user of the TVMI
CA	The coalition authority
MA	The machine authority
S	The sCore which will ultimately fulfill the resource request from the user
E_{AIK^+}	The public Attestation Integrity Key
R_{att}	Request for attestation
N_x	Nonce x
$A(N_x)$	An attestation (including the TPM quote) based on nonce N_x
U_x	Username of entity x
P_x	Password of entity x
R	Resource request
$counter$	monotonically increasing counter

Fig. 4.2. Description of variables used (in order of appearance) in the design of the TVMI protocol

only has VMs, the MA only has machines to control, so if the MA is controlling the only resource it has, it is clear that this is the only requirement the MA needs.

- The sCore is in an interesting position, because it is a resource itself. However, to ensure correct operation of the TVMI, the sCore should obey the commands of the MA (and only the MA). This means the same requirement is in effect as for the MA - the sCore should only serve jobs that the MA has approved.

4.4 Design

In this section, we describe the design of the TVMI, which fulfills the requirements laid out in section 4.3 and includes the entities which are introduced and discussed in the preceding section (user, MA, CA, sCore). As an example of the TVMI's operation, we consider the case where a user U wants to start a virtual machine VM_{user} on a remote sCore S, designated in figure 4.3 as "sCore." In the following illustrations, we use the naming conventions established in figure 4.2

Before the protocol begins, each sCore must establish a shared symmetric key with its MA. This key will be used to later prove that a given request came from the MA. In order to do this, the sCore and the MA participate in a standard authenticated Tiffie-Hellman exchange, using MA_{AIK^+} and S_{AIK^+} as public keys. After doing so, the symmetric key $K_{sCore,MA}$ is agreed upon.

Additionally, we introduce the operation of requesting an attestation, $REQUEST(A, B)$. This designates the process of A requesting and verifying an attestation from B. This operation occurs a few times in the description of the protocol, so it is abbreviated here for simplicity. Additionally, an attestation of B's system state may be cached on A, in which case $REQUEST(A, B)$ could return immediately, if A is willing to accept a slightly out-of-date attestation in its cache. The following is a description of what happens in the $REQUEST(A, B)$ exchange, exactly.

1) $A \rightarrow B : R_{att}, N_1$
2) $B \rightarrow A : A(N_1)_{B_{AIK^+}}$

Now, we present the details of the TVMI protocol in terms of $REQUEST(A, B)$, and assuming the authenticated Diffie-Hellman exchange to create $K_{S,MA}$ has already taken place.

1) $REQUEST(U, CA)$
2) $U \rightarrow CA : R, \{U_{user}, P_{user}\}_{CA^+_{cert}}$
3) $REQUEST(CA, MA)$
4) $CA \rightarrow MA : \{U, P, R\}_{MA_{AIK}}$
5) $MA \rightarrow CA : \{HMAC(K_{sCore,MA}, R, counter), R, counter, (sCore_{AIK})\}_{MA_{AIK}}$

- $REQUEST(CA, S)$

6) $CA \rightarrow S : HMAC(K_{sCore,MA}, R, counter), R, counter$

The user U first sends a request for attestation to the CA. The CA then forms an attestation of itself and its sCore, and returns the result to U. U then passes a username and password over SSL to the CA, along with the request to start VM VM_{user} in the **send_request** message (see figure 4.3 for message names). This implies that a PKI is in place which allows U to know the CA's public SSL key.

After this exchange, U has attested to the software stack on the CA (and its underlying sCore, to ensure that no "bare-metal" tampering is occurring). As a consequence, U has verified that the CA will not release VM_{user} to an sCore that will not enforce the CA's coalition policies. This follows from the fact that U can check the enforcement infrastructure available in the CA, and can see what types of policy decisions it will make by inspecting its software to ensure that the CA is running the trustworthy versions of the relevant programs. The user also knows that the CA is not running software that will leak his username and password, so it is safe for U to send this information. In short, U now trusts the CA to perform actions on U's behalf, since U has verified that the CA's software is capable of enforcing its security goals.

The CA is presented with a user's credentials, and by the assumptions we make in this system, treats the resource request as coming from user U. At this point, the CA must begin to negotiate the usage of sCore resources on U's behalf via an MA. The CA issues a request for attestation to the MA, and the MA replies with an attestation of both itself and its sCore (REQUEST(CA,MA)). The CA then sends credentials to the MA in the **send CA request**(we use a coalition name and password, but nearly any authentication scheme could be used). The MA then replies with a token in the **send token** message, which certifies that the MA has approved the resource request. Additionally, the MA includes the public portion of S's Attestation Integrity Key, or AIK.

The attestation of the MA lets the CA believe that it can release its credentials to the MA safely, since the MA's software has been verified protect this information. The attestation exchange also lets the CA believe that the AIK it receives is, in actuality, the public key of S's

AIK. Without this assurance, an malicious MA colluding with a malicious sCore could return an invalid AIK. When the CA goes to verify the sCore's attestation, anything could be returned, signed under the invalid AIK, and it would appear valid. Additionally, the CA is also able to believe that the MA will not poorly or unfairly schedule its virtual machine, since the MA's scheduling code has been verified by the CA.

Finally, as discussed below, the token returned to the CA in the **send token** message will allow the CA to prove to S that it has received authorization from the MA for the operation it wishes to perform (starting VM_{user}).

The token returned to the CA must provide a number of guarantees, both to the CA and S. S must be able to ensure that this is not a replay or duplicate, and is intended for S, not some other sCore S'. S must also be able to prove that the token actually came from the MA. Finally, the token must only authorize the operation requested by U (starting VM_{user}), so as not to allow the CA carte-blanche access to S. In order to do this, when an sCore first starts, it contacts its MA, and constructs a shared key S_k via an authenticated Diffie-Hellman exchange. This shared key is used to HMAC the token, which looks as follows: $HMAC(S_k, \{request\}_{P_{AIK}}, seed)$, where AIK is S's AIK. In this way, when the token is presented later, S knows exactly which request was authorized by the MA. S also knows that this is not a replay, since the seed value cannot be changed, except by the MA (and S keeps track of which seed values it has seen). Finally, S knows the token was intended for him, since the request is encrypted with his public key.

All that remains is for the CA to verify the integrity of S. This job falls to the CA and not the MA because it is the CA's duty to protect U's data. The MA is only concerned with restricting access to physical resources (sCore). Therefore, before sending the token in the **send CA request to sCore** message, the CA performs a standard attestation of the sCore. As discussed above, S knows that the request was approved by the MA, since the token is checked. The CA checks the attestation it receives from S to ensure that S enforces the desired coalition policies. The request then is transmitted to S, which always serves the request if the token can be validated.

Note that this design assumes a limited public key infrastructure. If a user wants to use the TVMI, he must trust the CA's public SSL key, in order to secure his login credentials. The rest of the public key infrastructure is tied into distribution of public AIK's for sCore. In some way, the MA must obtain the public keys for sCore it manages, and the CA must obtain public AIKs for the MAs it wishes to use. We will discuss current and potential implementations of these schemes in section 5.4.

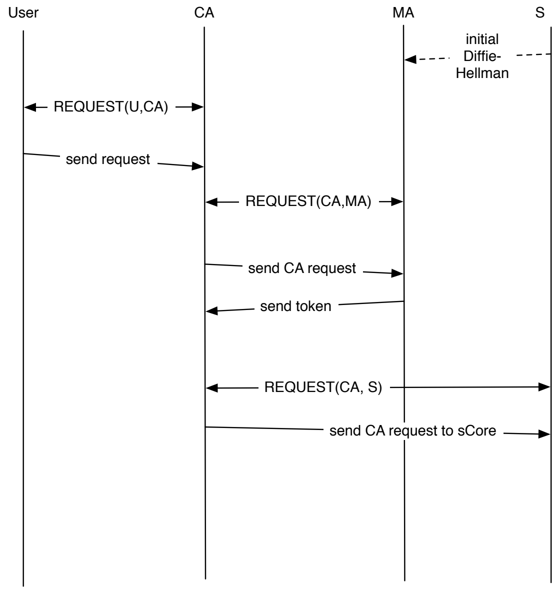

Fig. 4.3. Protocol to communicate a User request to a fulfilling sCore

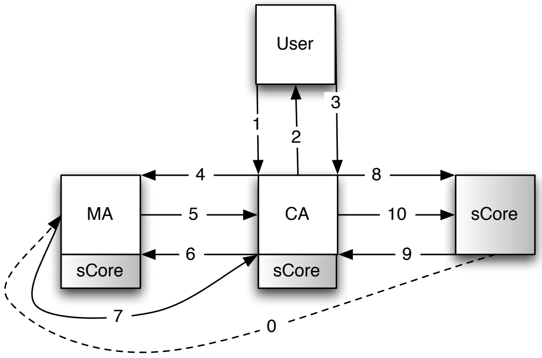

Fig. 4.4. Illustration of the protocol to communicate a User request to a fulfilling sCore (view 2)

Chapter 5

Validation of the TVMI

5.1 Introduction

In Chapter 4, we introduce the TVMI in order to provide secure virtual machine management for users and owners of physical machines. However, little to no discussion was devoted to whether or not we had met these goals. Given that the TVMI is so complex (more than 11 messages required for some requests), some discussion of whether or not this protocol satisfies the requirements laid out in section 4.3 is required.

In section 5.3.2, we use formal protocol analysis tools to evaluate the security claims we have made for the TVMI in chapter 4. Given that few solutions exist for trusted virtual machine management, it is important to formally validate our solution. Otherwise, it would be difficult to understand the security provides since it is not built on or derived from an existing validated work.

Additionally, we wish to provide a brief description of our TVMI implementation. The actual implementation of the TVMI required a number of design decisions, and we include a brief discussion of some alternative designs, including their strengths and weaknesses, in section 5.4. Finally, we perform some common operations within the TVMI in order to evaluate its performance and operation in section 5.5.

5.2 Trust in the TVMI

In in interest of clarity, we briefly summarize the trust model and key distribution problems implicit in the protocol description in section 4.4.

- User: For U to interact with the CA, U must have a public SSL key for the CA, signed by some Certificate Authority that U trusts. Although this requirement is slightly burdensome (a PKI adds complexity to the implementation), this is not altogether unusual for similar services. For instance, VMware's VirtualCenter product uses the same scheme, as does Citrix [51]. In reality, this is one of the simplest and best-understood ways for untrusted clients to interact with a trusted server when keys are not known ahead of time.

- CA: After the user submits a request to the CA, the CA must know the public portion of each MA's AIK. Otherwise, the CA is unable to verify that the MA is behaving appropriately. If the MA is misbehaving, the MA can DoS the CA by providing sCore information that is invalid, and will result in a failed job submission each time. More importantly, the MA could knowingly return an invalid AIK^+ for the sCore which will fulfill the final request. This would allow an attacker to impersonate an sCore by returning an valid sCore's attestation (verified in Proverif). Consequently, the CA must be able to attest to the MA.

  Note that we could do away with the CA's attestation of the MA if we were willing to accept a few weaknesses. First, the public AIKs would have to be signed by a trusted third party. The current state of TPM attestation does not point to this being a reasonable assumption, but it is conceivable that sufficient organization could emerge that makes this practical. We would also have to accept that a malicious MA could schedule VM_{user} poorly, but this would not compromise security, as we will see later. It would, however, mean that a VM could be scheduled on a busy sCore and starved.

- MA: The MA only trusts that it has the public AIKs for each sCore that it manages. In general, the distribution of AIKs between sCore and their managing MA should occur infrequently enough in most instances (only when a new machine is added to the trusted set) that this may occur out of band, which is what we assume in our implementation. In general, however, a TPM PKI such as the one found in Trousers [1] is a better, more general solution. The downside (and why is was not used here) is that it is not actually used in practice. Without a number of people using this solution, it loses much of its value.

- sCore: The sCore needs the public AIK for the MA it is registered with. As described earlier, a custom install image, or an automatic download of AIKs from a set location or external media also can solve this problem, at the cost of some risk.

Clearly, little trust is assumed in this system, but key distribution is something that must be considered carefully. If standard PKI practices are used for the CA, the main burden of key distribution will be adding new keys to the MA when sCore enter its domain of control. Since few MAs will become/cease to become available to a given CA, the distribution of AIKs in this scenario is minimal.

5.3 Validation

5.3.1 Informal demonstration of correctness

In this section, we wish to informally establish that two of our security goals, confidentiality of user credentials and CA credentials, are accomplished. This discussion is intended to give an intuition as to **why** the TVMI protocol is secure. A formal discussion of the TVMI in section 5.3.2 will demonstrate **that** it is secure.

The user's and CA's credentials are only sent over the network encrypted with the CA public certificate and the MA's AIK, respectively. These private keys are only owned by the CA and MA, respectively, so they are protected from eavesdroppers. Past this, the user can use the attestation it receives to verify that the CA's code will not leak these credentials. The CA can use its attestation of the MA to similarly verify that the MA will not inappropriately leak these credentials unless it is compromised.

We now investigate the requirement that the CA not leak a user's VM except by user-defined policy. Since the CA attests to the sCore, the CA can verify that the sCore conforms to the user defined policy. However, in our implementation, the CA trusts that attesting to the MA ensures that the sCore's AIK (used by the CA to verify the sCore's attestation) corresponds to the sCore. Finally, if a VM is actually to be sent across the network, the CA and sCore use their public/private keypairs to establish a private key to be used in encrypting the VM to be sent. Consequently, if the CA is able to verify the environment in which the VM will run (attestation from the sCore), and the VM is encrypted as it travels across the network, then the VM data does not leak except to environments by user-defined policy.

Consequently, the user's goal of only leaking VM data via a user-defined policy is accomplished. Since the user verifies the CA's code, it knows that the CA will behave as outlined above. Since we've shown that the CA, in its proper behavior, will not leak VM data inappropriately, and we've shown that the CA is behaving properly, the user knows that his VM will not be leaked inappropriately.

Finally, we must show that the MA provides complete mediation for the execution of resource requests. This involves showing that an sCore will only execute an incoming resource request if it has been approved by the MA, and only once (no playbacks of requests should be allowed). This is equivalent to showing that the HMACed token given to the sCore by the CA in step 10 cannot have been forged, is not a playback, and is destined for the appropriate sCore. If the token has not been forged, then it is from the MA. Assuming this, since the token contains the request to be executed, and it is from the MA, then the MA has approved the request in the token (somewhat obviously, an uncompromised MA is defined not to create tokens with resource requests that are NOT approved). Thus we must only show that the token cannot be created by anyone other than the MA, it cannot be replayed, and it is meant for the sCore that receives it.

The only time the key shared between the sCore and MA is sent over the network, it is encrypted with the MA's AIK, which we assume only the MA has access to. By this assumption, only the sCore and the MA have $K_{sCore,MA}$. Then only the MA or the sCore could have created the HMACed token, since only the MA and sCore have this key. The sCore would not create this message if it were not compromised (as it falls outside the protocol specification). Compromised sCore fall outside the goals we define in section 4.3. Therefore, the token was generated by the MA. Since the sCore name (we use IP addresses in our implementation) is signed by the MA's AIK, we know that the token is destined for a specific sCore. Finally, since the MA increments the counter included in the message each time a request is sent to an sCore, we know that this token is not a replay. This requires the counter not to "loop back" on itself.

The sCore is just a physical resource of the MA, so it is somewhat disingenuous to say that the sCore trusts the MA (it is akin to saying that a computer terminal trusts its user), but in this scenario, the sCore does whatever the MA tells it to do. In other words, the MA has complete control over what VMs are executed on a given sCore. We will discuss other alternative trust models in section 5.4.

5.3.2 Formal demonstration of correctness

Following the description of the TVMI in section 4.4, we discuss the correctness of this approach. Clearly, other protocol designs are possible, and some of these alternate design decisions will be discussed in section 5.4. We wish to show that the design of the TVMI does not include superfluous messages while accomplishing all of the goals laid out in section 4.3. In other words, we must show that the TVMI protocol design is both entirely necessary and sufficient to accomplish our goals.

In order to do this, we use a formal verification tool, Proverif [13], to help determine the correctness of our protocol. By doing so, we demonstrate in this section that the TVMI protocol fulfills the security goals we need. In section 5.5, we demonstrate that the TVMI actually performs the required actions. These two proofs together demonstrate the sufficiency of our approach. We will briefly discuss necessity in section 4.4, but no formal proof is offered.

5.3.2.1 Introduction to Proverif

Our approach utilizes a tool named Proverif, which attempts to determine satisfiability of certain properties in security protocols. In general, we generate a model of the TVMI in Proverif, which we then query for security properties that we specify (as enumerated in section 4.3). However, the exact operation of Proverif bears discussion, as it effects the results of our validation.

In general, Proverif is a tool which turns a model (whose format is discussed shortly) into a series of Horn clauses (a disjunction of literals with at most one positive literal). Then, Proverif applies the rules defined in the model in an attempt to derive as many facts as possible. This derivation process is not discussed in detail here; the basics can be found in the original paper on Maude, a loosely related system [28]. Proverif adds some important functionality to the approach taken in Maude. Firstly, Proverif understands the idea of a nonce in a much more useful way. If two pairs of parties in a Proverif model send each other nonces, Proverif understands the idea of a nonce and will use two different values for the nonces that are sent. This becomes particularly important when interleaving protocols, as using the same nonce for both pairs may result in some attacks being found that are not vulnerabilities in practice.

Additionally, Proverif is capable of finding more attacks than previous attempts at automatic protocol verification. Perhaps most importantly, Proverif attempts to execute an arbitrary number of each process (loosely coupled with entities in protocols) defined in the system. For instance, if a protocol involves Alice and Bob, Proverif does not simply interpose Eve between Alice and Bob, but rather may have 127 instances of the Alice process, 46 instances of the Bob process, and Eve may intercept and use data from all of them. A corollary to this is that each step in each process may be executed more than once, instead of a single time. For instance, if a message is encrypted by a process, Proverif may attempt to encrypt the message multiple times to achieve some desired derivation. These are very important points in practice, as Proverif is able to detect much larger classes of attacks than previous tools.

In practice, Proverif accepts models that use the process calculus [59]. While a lengthy discussion of this calculus model is outside the scope of this thesis, the general idea is to model a system in terms of simultaneously executing entities called processes. These processes employ message passing over predefined channels in order to communicate. In practice, this message passing ends up being the focus of automatic protocol verification, as the channel abstraction provides a convenient way for the attacker to interact with the model. If the attacker can place or receive arbitrary messages on a defined channel, he has something approaching the high end of attack capabilities in most remote attack scenarios.

The process calculus helps to define the constraints of the model, the expected and allowed operations/reductions, and the channels which pass messages. However, Proverif also requires a language to specify the processes themselves. Since Proverif is, at its heart, a Horn clause solver (tied closely to the same functionality in Prolog [26]), it is only natural that writing processes in terms of Horn clauses would be an option. However, Proverif also includes a newer, more flexible language, called the Pi calculus. The Pi calculus, which we will describe in more detail in the next section, allows us to specify how a process operates. In particular, it allows a more straightforward specification (more closely tied to the protocol description itself) than does the Horn clause representation. This means that Proverif must translate the Pi calculus to Horn clauses [14]. In this way, the Pi calculus acts as a simpler ”frontend” to the Horn clauses that are fed into prolog to do the final reachability queries.

Proverif queries the derivations that are performed on these Horn clauses to determine if a set of goals can be met. The ”goals” in Proverif are most often related to security properties. For instance, a goal may be that an attacker can obtain a secret s. In proverif, the line ”query attacker:s” would designate that as the derivations are being performed, if there is a way an attacker could obtain s, print out the sequence of derivations that leads to this knowledge. In this way, Proverif not only determines if a given property can be satisfied, but it also details how it is satisfied. More details will be given in section 5.3.2.2.

In order to do this, Proverif attempts to apply all the reduction rules available to it to the messages an attacker can receive. For instance, the following code defines the encryption and decryption functions for shared key cryptography.

```
(* Shared key cryptography *)
fun enc/2.
reduc dec(enc(x,y),y) = x.
```

Thus when an attacker sees a token, Proverif attempts to apply the decryption rule, which will succeed if the key, y, is known along with the encrypted token. Proverif attempts to apply as many reduction rules as possible in order to achieve the maximal possible information an attacker can know. More details on this specific operation are given in the original Proverif work [13], but the reader should have an intuitive understanding that Proverif applies reductions in order to determine how much information an attacker can have.

Proverif is not without its limitations, however. It will never return a false positive (if a positive is that a query cannot be satisfied), but it can return either false negatives, that an attack is possible but cannot be enumerated, or nothing (non-termination). For instance, in the example above, if Proverif says that ”query attacker:s” cannot be satisfied, then we know that it cannot. However, the program may not terminate, or it may return that an attack exists but cannot be detailed. In this case, the attack may or may not exist, but Proverif cannot determine this. This is actually a very different problem than the classical Halting Problem, however - it comes more from how the satisfiability algorithm runs, as at its heart, this is a problem with satisfiability with reasonable performance, not halting.

5.3.2.2 Verification Results

During the verification process, we discovered some unresolved issues in the Proverif implementation of the protocol. For instance, the user's request was incorrectly transmitted in the clear from the CA to the MA in the proverif implementation. As a result, an attacker was able to substitute a malicious request for the request from the user that was verified by the CA. In this way, an attacker was able to submit any job request he wanted (including adding, deleting, or modifying coalitions), after the policy decision point for these operations in the CA.

Proverif found this (and many more attacks) during the creation of the Pi-calculus Shamon model while the model and the types of authentication and secrecy checks to be performed were being debugged. As an additional example, consider the case where the initial exchange between the sCore and the MA is just a standard public key encryption, instead of a Diffie-Hellman exchange. In this case, an attack can replay the registration message to the MA, and change the shared key, which results in a DoS against the sCore (it can no longer decrypt messages form the MA describing user requests). In fact, Proverif makes a distinction between these types of attacks, by designating an "active" and "passive" attacker model. In our situation, we modeled the TVMI under an active attacker, as our final goal for the system is to provide trusted virtual machine management even in a situation where the attacker has physical (but not circuit-level) access to any and all machines (excepting DoS attacks, since an attacker can always unplug machines).

The exact Proverif traces are too long to reproduce here in full, but we reproduce relevant sections here to demonstrate what Proverif proves, exactly.

```
...
Rule 52: attacker:nonce_from_ca_242 & attacker:sign(g(y_243),skma[]) &
 attacker:pk(skma[]) -> attacker:att(sign(nonce_from_ca_242,skscore[]))
Rule 53: attacker:(enc(x_252,f(y_253,g(n0[pk(skma[]),sid_254]))),
 penc(x_255,pk(skscore[]))) & attacker:nonce_from_ca_256 & attacker: sign
 (g(y_253),skma[]) & attacker:pk(skma[]) -> end:received_vm(pk(skscore[]))
Completing...
nounif attacker:sign(m_335,skma[])/-5000
nounif attacker:sign(m_339,k_340)/-5000
Starting query ev:received_vm(pkstub_83) ==>
         ev:attestedByCA(pkstub_83)
```

The beginning of this trace is an example of some of the derivations Proverif performs using the reductions specified in the protocol description. This is not particularly noteworthy, other than to demonstrate how much information an attacker can put together in a protocol this complicated. Proverif then queries the attacker's information for a specific security property. In this case, we check to see if the following property holds: if an entity receives a VM, then it has been attested by the CA. Proverif has been able to determine that this property holds, because an event (received_vm) is raised when a VM is received in the protocol, and another event (attestedByCA) is raised when a CA attests to an sCore. In this snippet, Proverif has determined that if the received_vm event is raised, the attestedByCA event must have also been raised.

```
...
Rule 53: attacker:(enc(x_836,f(y_837,g(n0[pk(skma[]),sid_838]))),
         scorevmenc_839) & attacker:nonce_from_ca_840 &
         attacker:sign(g(y_837), skma[]) & attacker:pk(skma[]) ->
         end:endRequestReceivedByMA(x_836)
Completing...
nounif attacker:sign(m_912,skma[])/-5000
nounif attacker:sign(m_916,k_917)/-5000
Starting query ev:endRequestReceivedByMA(rstubma_669) ==>
         ev:beginRequestReceivedByMA(rstubma_669)
RESULT ev:endRequestReceivedByMA(rstubma_669) ==>
          ev:beginRequestReceivedByMA(rstubma_669) is true.
```

In this example, Proverif has determined that if a request is fulfilled, the request was received by the MA. An important property of Proverif to note here is that events take arguments. In the snippet above, Proverif is checking not only that if if the "beginRequestReceivedByMA" was raised, the "endRequestReceivedByMA" was raised, but also that the actual request raised is the same in both instances. This prevents an attacker from modifying a request that was legitimately approved by the MA.

```
Rule 52: attacker:nonce_from_ca_1385 & attacker:sign(g(y_1386),skma[])
        & attacker:pk(skma[]) -> attacker:att(sign(nonce_from_ca_1385,
         skscore[]))
Rule 53: attacker:(enc(x_1393,f(y_1394,g(n0[pk(skma[]),sid_1395]))),
         scorevmenc_1396) & attacker:nonce_from_ca_1397 & attacker:
         sign(g(y_1394),skma[]) & attacker:pk(skma[]) ->
         end:endRequest(x_1393)
Completing...
nounif attacker:sign(m_1469,skma[])/-5000
nounif attacker:sign(m_1473,k_1474)/-5000
Starting query ev:endRequest(rstub_1226) ==> ev:beginRequest(rstub_1226)
RESULT ev:endRequest(rstub_1226) ==>
          ev:beginRequest(rstub_1226) is true.
```

These events correspond to the user submitting a request and an sCore serving the request. The **endRequest** event is raised in the Proverif Shamon model whenever a request is served by an sCore. The **beginRequest** event is served whenever a user makes a request. Together, this derivation shows that if an sCore serves a request, it can only be in response to a user making **the same request**. As above, this shows us both than an attacker cannot modify a request, and also that an attacker cannot submit an improper request on a user's behalf.

Finally, we wish to show that certain secrecy requirements hold:

```
Starting query not attacker:username[]
RESULT not attacker:username[] is true.
Starting query not attacker:password[]
RESULT not attacker:password[] is true.
Starting query not attacker:vm[]
RESULT not attacker:vm[] is true.
```

After performing as much derivation as possible, Proverif does a straightforward Prolog query of the information that has been derived for the attacker. In the case of our secrecy requirements, this returns a straight yes/no answer. In the above snippet, we see that an attacker is unable to determine the username or password of the user, as well as the contents of the VM.

In order to demonstrate that Proverif finds attacks, we comment out the portion of the model in which the sCore checks that the public key used in the original Diffie-Hellman exchange belongs to the MA. This was an actual attack we encountered in the creation of the model.

```
goal reachable: attacker:rstub_2601 -> end:endRequest(rstub_2601)
rule 50 end:endRequest(rstub_2625)
  2-tuple attacker:(enc(rstub_2625,f(n0[pk(k_2620),sid_2621],g(x_2613))),
scorevmenc_2623)
    rule 14 attacker:enc(rstub_2625,f(n0[pk(k_2620),sid_2621],g(x_2613)))
      hypothesis attacker:rstub_2625
      rule 3 attacker:f(n0[pk(k_2620),sid_2621],g(x_2613))
        any attacker:x_2613
        rule 1 attacker:g(n0[pk(k_2620),sid_2621])
          0-th attacker:sign(g(n0[pk(k_2620),sid_2621]),skscore[])
            rule 47 attacker:(sign(g(n0[pk(k_2620),sid_2621]),skscore[]),
pk(skscore[]))
```

```
              duplicate attacker:pk(k_2620)
    any attacker:scorevmenc_2623
  any attacker:nonce_from_ca_2624
  rule 7 attacker:sign(g(x_2613),k_2620)
    rule 4 attacker:g(x_2613)
      any attacker:x_2613
    any attacker:k_2620
  rule 11 attacker:pk(k_2620)
    any attacker:k_2620

Goal of the attack :
end:endRequest(a_21[])

out(ca_to_user, pk(skscore_28))
out(ca_to_user, pk(skma_29))
out(ca_to_user, pk(skca_30))
out(ca_to_user, host(pk(skscore_28)))
out(ca_to_user, host(pk(skma_29)))
out(ca_to_user, host(pk(skca_30)))
in(ma_to_score, pk(a_22))
out(ma_to_score, (sign(g(n0_27),skscore_28),pk(skscore_28)))
out(ma_to_score, pk(a_22))
in(ca_to_score, a_26)
out(ca_to_score, att(sign(a_26,skscore_28)))
in(ca_to_score, (enc(a_21,f(n0_27,g(a_24))),a_25))

event(endRequest(a_21))

An attack has been found.
RESULT ev:endRequest(rstub_1989) ==> ev:beginRequest(rstub_1989)
is false.
```

Here, Proverif gives a listing of the rules used to derive the information an attacker needs. It also gives the sequence of messages an attacker sees and sends (the lines beginning with in/out), as well as a description of what security property was violated. In this case, an attacker was able to trigger the **endRequest** event without the **beginRequest** event being generated as well. In the Shamon system, this corresponds to an attacker being able to submit a request on the user's behalf, and having it executed by an sCore.

The examples presented here conform closely to the requirements in 4.3.1. The examples above demonstrate that the user's username/password remain secret (except to the CA), the user's VM does not leak except by the user's request. As for the CA, we showed that a request and VM were only given to an attested sCore, and the CA's credentials (for use on the MA) were kept secret. Finally, we showed that only requests seen first by the MA were run on sCore. This encompasses the requirements presented in section 4.3.1, so we have proven the properties we are concerned with.

5.3.3 Summary

Our Proverif model proved (by queries of maximal derivable information) that the following properties hold:

- The user's password is never leaked
- The request fulfilled by the sCore is the request that the user made
- Each request that an sCore fulfills is verified by the MA
- No VM is leaked to an attacker
- Each sCore that runs a VM is attested by the CA

In this way, we demonstrate that information flow properties are preserved, all requests are verified in order to give control to the owners of the physical machines, and basic authentication properties are preserved. Currently, there is no established way to verify that these are the "correct" set of properties for us to care about. However, these properties broadly cover the only resources of interest in the TVMI (VMs and physical machines), which strongly suggests that this set of requirements is a superset of the necessary requirements.

5.4 Implementation

Our implementation of the TVMI helps demonstrate the practicality of this approach and brings to light issues with standard practices in trusted computing today. We demonstrate that the protocol specification outlined in section 4.4, which was necessitated by the goals lined out in section 4.3, is both practical and possible. Additionally, we discuss many of the design decisions we encountered in the implementation of the TVMI, in order to give a better understanding of the tradeoffs made in the current description of this system.

Our first major design decision is in whether or not the TVMI should be a closed system. In other words, what types of users will be allowed to make requests to the TVMI through the CA? We could have required U to be running on or in an sCore, and thus performed mutual attestation when submitting a request to the CA. This would have ensured that a compromised system could not make malicious requests with cached or leaked user credentials. We felt that this was too restrictive, however. We felt that it was more important that users be able to interact with the TVMI using existing hardware and standard operating systems. The current scheme allows this, at the price of trusting that user credentials have not fallen to the wrong hands. This also closely mimics how most grid systems work today, which allows us to apply much of the existing research in securing untrusted job submission agents to this step of the TVMI [18].

The agent participating in the protocol on each physical sCore is the trust daemon, as described in Chapter 3. In Chapter 3 this daemon was responsible only for marshalling and verifying attestations. With the addition of this work, this trust daemon also becomes responsible for participating in this application level protocol. Since virtualization controls currently occur at the application level instead of in the kernel (or inside the network protocol stack for things like migration), running this protocol at the application level inside the trust daemon maintains consistency with current approaches. For instance, the xend daemon, which processes remote requests for virtualization-related operations, operates at the application level for commands like suspend, resume, and migrate. Since we are performing similar operations, it makes sense to leave this protocol implementation at the application layer in order to keep up with the rapidly changing [91] virtualization management space. In the future, however, much of this could be moved inside the kernel [52], especially as it relates to an authorization and authentication protocol like the one proposed in section 4.4.

This trust daemon is firstly responsible for managing attestations. This includes keeping track of which systems are attested, verifying attestations, expiring attestations, and marshalling attestations. Currently, the trust daemon interfaces with the Integrity Measurement Architecture (IMA) via sysfs, but as these services move into the kernel, this becomes unnecessary.

More specifically, the trust daemon creates an XML document as an attestation in response to an attestation request, as suggested by IBM for Direct Anonymous Attestation [47]. This allows the attestation to be flexibly extended as more information needs to be included. It also establishes a strict, human-readable form that can be easily parsed for verification. An example attestation is shown here:

```
<ATTESTATION>
<RESULT>OK</RESULT>
<NONCE>123456</NONCE>
<MEASUREMENTLIST>
   <MEASUREMENT>
      <NUM>1</NUM>
      <SHA1>9797edf8d0eed36b1cf92547816051c8af4e45ee</SHA1>
      <STATUS>clean</STATUS>
      <NAME>boot_aggregate</NAME>
   </MEASUREMENT>
   <MEASUREMENT>
      <NUM>2</NUM>
      <SHA1>adf9d684f2e94acdfc3870195fac85686e37d64b</SHA1>
      <STATUS>clean</STATUS>
      <NAME>/bin/sh</NAME>
   </MEASUREMENT>
   <MEASUREMENT>
      ...
</MEASUREMENTLIST>
<QUOTE><
   <NONCE>40E2010000000000000000000000000000000000</NONCE>
   <COMPOSITE>00020004000000144B1F253495010968C1FE15439EAFBD02C021CE9F<COMPOSITE>
   <SIGNATURE>7556B8BBD3A366F2998B3C583730DCDAB26939
      D2175C38D814A04659E306962446022573DC2094C4C
      ...
      29A85ACC17D2A8D7B96C04C
   </SIGNATURE>
   <VERSION>01010000</VERSION>
   <FIXED>51554f54</FIXED>
</QUOTE>
</ATTESTATION>
```

The first portions of the attestation are simple to understand. MEASUREMENTLIST denotes an ordered list of which programs were run on the system returning the attestation. This includes the order they were run (NUM), the name of the program (NAME), and the SHA1 hash of the binary. This information is all obtained via sysfs from IMA. The QUOTE portion comes directly from the TPM itself. This includes the nonce (provided by the requesting party, to ensure freshness of the attestation), as well as TPM version and other bookkeeping. The important portion is the state of all platform configuration registers (PCRs), which encompass the state of the TPM itself. The verifying party can then attempt to reconstruct the hashes in the PCRs from the hashes provided in the measurement list. This works because when the system reboots, the TPM is zeroed. Then, every time a program is loaded into memory, IMA extends a PCR with the hash of the binary that is loaded. Extending works as follows:

```
PCR := SHA1(PCR+MEASUREMENT)
```

Therefore, a verifying party with knowledge of each hash that a PCR was extended with (in our case, this is the MEASUREMENTLIST), can reconstruct the expected state of the PCR. By comparing with the actual state reported in the QUOTE, we can determine if the state reported by the TPM corresponds to the list of programs that was purportedly executed. If they do not match, the verifying party knows that the reporting is faulty, and rejects the attestation.

There are a few assumptions built into this model. First, a given PCR cannot be arbitrarily changed. Otherwise, a malicious party could simply set a PCR to correspond to a well-known sequence of acceptable execution, report that sequence of execution to a remote party, and thus represent a system state that does not correspond to reality. Fortunately, the TPM prohibits changing a PCR to a given value. If an attacker could extend a PCR to get a certain value, then he could manipulate the TPM PCRs into a useful state. However, since SHA1 is not currently though to be reversible, an attacker cannot determine what values were used to extend the TPM given just the SHA1 value. Finally, as described in 3, IMA not be subverted by a lower layer of software. To avert this, we measure all ”lower” layers in the software stack, to ensure that they are behaving properly.

One of the biggest difficulties in the implementation of the TVMI is ensuring that the CA and MA can generate these attestations for themselves (since they run in virtual machines). Traditionally, virtual machines do not have access to a trusted computing device such as a TPM [39] to generate these attestations. Without a device like this, an IMA-like approach cannot be applied to virtual machine attestation. Fortunately, a virtual TPM, or vTPM is available as part of the Xen codebase [10]. This can be accessed just like a TPM, from the point of view of the virtual machine.

However, our implementation requires the CA and MA to attest both to themselves and their underlying platform (the sCore). Somehow, these need to be linked. The vTPM implementation from IBM does just that - the vTPM is linked to the underlying TPM, so that attestations produced by the vTPM are still tied to the physical platform [67]. Unfortunately, the IBM vTPM implementation is not publicly available. However, the vTPM included in the xen-tools package (developed by Intel) also does not contain this functionality, and also includes an additional caveat: the physical TPM and the vTPM cannot both be used concurrently, or even sequentially. Until this is fixed, the IBM vTPM is released, or the TPM supports multiple simultaneous accesses (not likely to happen, as this would compromise security properties), the TVMI cannot operate exactly as described. Instead, in our implementation, the CA and the MA run on virtual machines on different physical machines than their sCore. The rest of the protocol can then remain unchanged.

The other approach to tying a VMs attestation to its sCore is by a direct hypercall by the VM. If an additional hypercall is added to the VMM, the VM has a way of directly asking the VMM for an attestation of the underlying platform, in addition to the VM attestation. This is different than our implementation, in which the CA/MA send a message to their sCore, asking for an sCore attestation. The hypercall approach has the advantage of being simplest, and fastest. However, adding a hypercall adds extra interfaces by which the VM may compromise the VMM. Additionally, it does not immediately tie the VM attestation to the sCore attestation. Some cryptographic link, such as the one found in the vTPM would still be required.

Unfortunately, no matter how these attestations are generated, they are still load-time attestations which attest to the state of the system when software is loaded. However, after the software is loaded, compromise of loaded programs can occur, undetected. Since programs are measured at load-time, if they are compromised later (at runtime), the measurement list will not reflect this compromise (unless other malicious programs are loaded as a result of the compromise). An emerging field of runtime attestation solutions [72] may solve this problem, but these solutions are insufficiently advanced for current implementations. Additionally, many of these solutions (for instance, those requiring page-level memory attestations) require much more complexity than can be comfortably included in an out-of-the-box setup that enables meaningful attestation [33].

An additional problem is the DoS threat exposed by allowing attestations to be requested by unauthenticated parties. Since it takes a significant amount of time to generate an attestation, allowing anonymous parties to request an attestation leads to a potential DoS attack. As a mitigation for this, when a CA or MA begins to experience a high amount of load, they may optionally request the requesting party to verify a client puzzle before the attestation would be generated. This would provide no performance degradation under low loads, since the client puzzle would not be presented. Under a DoS attack, this would help to slow down the rate of requests considerably [41, 88, 43]. However, under a widely distributed DDoS attack, the attestation service is still likely to become overloaded.

After the attestations have been processed and the trust daemon approves a command, the trust daemon itself executes the command on the sCore, via the Request base class, which takes a request specification and translates it into a direct command to the xen management (xm) daemon, xend. As such, we are able to eliminate xend as a network facing daemon (if we do not require migration capabilities), since it is no longer required to execute commands relating to the management of virtual machines. The trust service, which participates in the TVMI negotiations, takes care of this instead. Using the trust daemon to accept incoming virtualization requests helps to reduce the attack surface available to a remote attacker.

The rest of the sCore implementation follows closely from the description in chapter 3, with the exception of an addition to the installation process. In general, an sCore must know the address of its MA, in order to exchange a secret key that the sCore later uses to verify the MA's authorization of a resource request. In our implementation, the MA's address is given to the sCore trust daemon explicitly. A more robust approach would be to allow the user to specify the address of the MA during installation, or preseed it so as to retain a fully automatic install. However, this introduces the possibility that a fully verifiable sCore could associate with the ”wrong” MA. This would not lead to a compromise of security goals, but this does mean that an attacker with physical access could put an sCore under the purview of a different MA. This would allow an attacker to represent resources as coming from a different MA, perhaps for billing purposes.

An alternative implementation would get the addresses of the appropriate MA(s) from DNS records. A custom field can be added for a domain, which returns the MA for a given address. This requires the DNS infrastructure to be in place for the domain in question, which is why it was not used here (to maintain simplicity). However, this allows systems to flexibly determine their MA in the presence of a physical attacker as long as the network and DNS infrastructure is not tampered with. Using these three approaches, static, specified at install time, and DNS, administrators have a wide range of options for associating an sCore with an MA.

5.4.1 Storage

Although the implementation of the TVMI works as expected, it is missing an important piece of the puzzle of virtualization infrastructure: secure storage. When VMs migrate in a performance sensitive way (live migration [25], for example) they must share storage to obviate the need to transmit a VM's disk over a network. This implementation forgoes this feature for complexity's sake, but a production grade system must find a way to take this into account. Some solutions that may be applicable exist [56, 20, 37], but they are not discussed here.

This is a particularly important question to resolve, as it will govern how the coalitions in the TVMI will operate. For instance, if there is no shared storage, a VM must be copied securely from one sCore to another, which prevents most useful performance-sensitive migration because disk migration takes too long to maintain a semblance of availability. Additionally, the VM has now left the control of the CA. In the TVMI, we have assumed that VMs are stored in some secure, shared storage facility. If they are copied from sCore to sCore, some mechanism will have to be implemented to prevent information leaking to a physical attack who can reboot the machine and inspect the hard drive. This seems trivially achievable with full disk encryption whose keys are sealed to a trusted TPM state, but this begins to get complicated when memory attacks are considered [21, 42]. Furthermore, this sealing/encryption policy must be communicated in an attestation to enable one sCore to trust another to maintain the secrecy of encrypted VMs. It is difficult to underestimate the importance of secured storage, as it underpins both the security and performance models of this work. However, this is a broad area of research, with many different and divergent solutions, so it is not discussed in great detail here.

5.5 Experiment

Our implementation of the TVMI is built on the same open source platform as the sCore in chapter 3, as described in section 5.4. To demonstrate the usability and effectiveness of this protocol, we run an experiment as outlined in figure 5.1.

As described in section 5.4, the current vTPM implementations restrict our ability to use both the vTPM and TPM simultaneously. Consequently, we run the CA and MA on different physical systems than their sCore. In order to demonstrate that this works reasonably, we run an experiment for a user to login to the CA, and run the command to start a virtual machine on a remote sCore that the MA chooses.

Each physical machine is a Dell Optiplex 745, with dual core 1.86 GHz processors. All systems had 1G of memory. These machines are all running on the same 10Mbps network. The sCore were created according to the specifications found in 3. Most importantly, this means they are running a version of Ubuntu Linux with a modified 2.6.24 kernel.

In order to demonstrate the effectiveness and usability of this system, we run an experiment with the goal of having a user start a VM. The user used a client program to select commands to run for a menu, while supplying details such as coalition name, VM name, etc. For instance, the user might pick "Start VM" from a menu, and supply VM and coalition name, and send the request out. In this particular instance, all appropriate exchanges and verifications took place (as described in section 4.4), and a user's VM was started on a remote sCore.

Even over 6 physical machines (the design only specifies 4), this protocol took a total of 4.8 seconds to execute (2.8 seconds when MA and CA were re-used). Table 5.2 gives a listing of how long messages take to process, both for the first execution of a command and the second execution (when the CA and MA attestations are cached).

Table 5.2 demonstrates that the bulk of the processing time (approximately 3 seconds) is tied up in TPM quoting operations, which take .987 seconds apiece when quoting all PCRs (around .90 seconds if only 1 PCR is quoted). Additionally, this demonstrates one of the strengths of the TVMI architecture: it is much faster when the same CA and MA are reused by the user for future operations and attestations are cached. Very little processing time is required for the other messages, which reflects the relatively small size of the code involved in addition to the basic trust daemon in Chapter 3 (between 1000 and 2000 lines).

Clearly, the time delay for this protocol can be somewhat problematic for extremely performance sensitive applications. However, there are a few things that mitigate this. Firstly, attestations can be valid for a certain period of time. As shown above, caching attestations and re-using MAs and CAs speeds message processing up significantly. In many situations, it makes sense for the CA to use the same MA for subsequent requests. For instance, a coalition may logically "belong" on machines in a single administrative domain, controlled by the MA. In an educational setting, for example, a single MA would likely be responsible for a large number of machines, and a coalition based in that educational domain could decide to try to use those machines first. In this case, once a user submits his first request, subsequent attestations between CA and MA, as well as attestations between the user and the CA, will be cached.

In order to maximize the scalability of this protocol, this sort of caching will have to be emphasized wherever possible. In the future, resource allocation algorithms should focus heavily on re-using previously used resources so that attestations will be cached, wherever possible. If the final sCore has enough resources available, even the attestation between sCore and CA can be cached. This brings the total protocol time well under 2 seconds, a very low latency compared to the latency of the VM operations that are being requested. In short, a most-recently-used allocation strategy is likely to yield good performance speedups for virtual machine operations.

The other potential source of mitigation for this slowdown is from the TPM itself. We tested some TPMs (Atmel, version 1.2) in which the quoting function took more than 8 seconds in some instances (if not enough data was available in the entropy pool of the TPM). Generally, there is a good deal of variability over TPM models. Future generation of TPMs are likely to be a great deal faster, which will greatly decrease the running time of this protocol [77].

This experiment does expose one of the current weaknesses of trusted computing: the speed of the TPM is a detriment to overly complex attestation schemes. Consequently, decisions had to be made in this protocol to avoid pairwise attestation of all parties involved. Very little has been said on the topic of TPM performance, and we hope that this work will help to raise awareness of this as a performance-limiting issue. Server TPMs, however, have the potential to largely mitigate this problem [19]. Little is known about the speeds of these devices, however, so they are not included here.

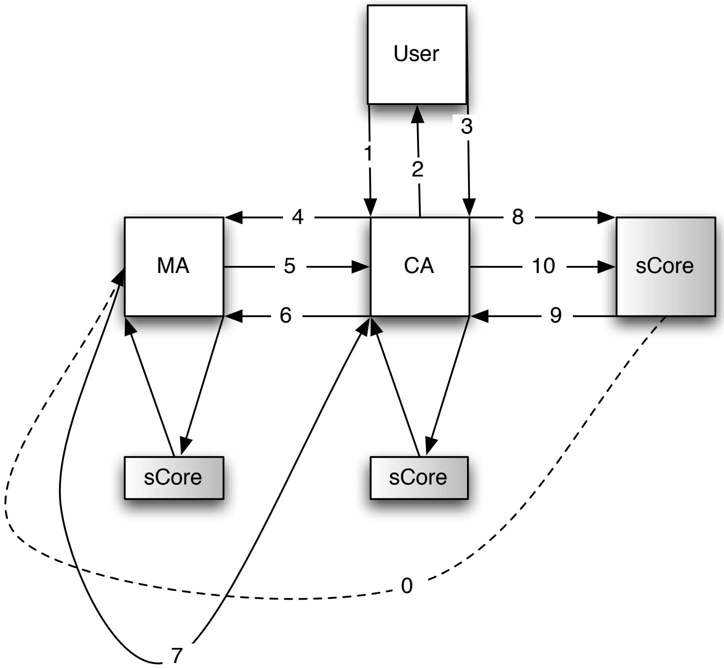

Fig. 5.1. Illustration of the protocol to communicate a User request to a fulfilling sCore

Message Type	Time w/ Attestation (in ms)	Cached Attestation
sCore attestation request	967.50	1.98
request from user (on sCore)	001.57	1.57
request VM attestation	001.35	1.35
CA process user request	3.95	3.95
Attestation response	16.77	16.77
MA response to CA	1.06	1.06
MA process CA request	1.01	1.01
User request to CA	6.39	6.38

Fig. 5.2. Description of times to process each type of message

Chapter 6

Conclusion

6.1 Future Work

By creating a ROTI, and by implication, a small TCB capable of being fully verified, we begin to enable distributed virtualization to operate with more stringent security goals. However, future work needs to be done in order to fix problems in the sCore and enable a richer set of applications.

One of the first problems that must be fixed is how to measure programs that run other programs. For instance, take the python interpreter. When a python program runs, the python interpreter runs first, and loads the python program. Unfortunately, the only thing measured is the python interpreter itself. So running "python /rootkit.py" and "python good_program.py" both show up in the IMA measurement list as "/usr/bin/python". In general, the model of measuring loaded code by not be sufficient to fully measure a system's state. A new approach may be needed, or some rationale provided that modifying scripting interpreters is the only action we need to take.

This additionally relates to the problem of runtime attestation mentioned earlier. If a piece of code is compromised, and a rootkit is injected, IMA will not measure the rootkit, because it is not new code being loaded into memory. In short, when code runs other code, IMA fails to detect it. Runtime attestation may allow us to say that a system is in a "good" state now, as opposed to current attestation, which can only certify the software running on a system at time t, where it may be time $t + \epsilon$ now.

Additionally, load-time attestations are still quite complex. The vastly reduced TCB (the sCore) introduced here is manageable, but if significantly more software is introduced, a method of reducing the attestation size will have to be devised. As discussed in chapter 2, PRIMA does this by only attesting high-integrity programs and their inputs. If determining this set of high-integrity inputs is difficult or impossible, it will be necessary to find a way to represent full attestations in a less complex way. Work towards this goal is currently ongoing.

Related to this, it is becoming critical to figure out a way to attest to virtual machines, as well as their hypervisors and control VMs (dom0, for instance). As seen with the problems relating to the vTPM in section 4, this is not currently a well-developed solution. In the Terra paper, for instance, VMs were simply attested as a giant blob of their filesystem [33]. While this is a great, simple solution for VMs whose entire disks are well known and do not change, for most VMs, this would quickly become unverifiable. Consequently, a new solution must be devised, if the virtual machines themselves are to be trusted, and not simply their host.

Even if the virtual machines are attested, there is still no adequate storage system for securing VMs in the trust model we assume (attacker with full physical and administrative access to the system). We still must find a way to store VMs, while maintaining the confidentiality and integrity of their data. This could be done in an sCore that seals data to a current system state, but we are interested in pursuing additional options that do not increase the size of the sCore itself.

Once this is done, more robust coalition operations can be performed. In its current incarnation, coalition management is a proof-of-concept. Some operations, like coalition joining, make little sense in the current environment because of the lack of current VM attestation mechanisms. If there were a mechanism to attest to VMs, a coalition join operation could make a much more informed decision as to whether or not to allow VMs to join or not. VM migration has already been discussed, and is another example of an operation which could be supported with more investigation into the trusted infrastructure of the TVMI.

Additionally, coalition management is more complex than the proof-of-concept implementation in this work would lead one to believe. Coalitions should support (and often require) complicated policy. While this work designs a framework to enforce a wide variety of arbitrary policy, more work is necessary to determine how the TVMI will be used in practical situations. Moreover, more work needs to be done in order to fully investigate the precise sequence of operations which should occur for each coalition request. This work, conversely, is primarily focused on achieving the most simple, secure version of these operations possible.

6.2 Conclusion

This work demonstrates that meaningful trust can be built between distributed systems using a root of trust in hardware. Using a root of trust, we can certify that code and data remains unchanged from a trusted source. This differs from previous approaches, most of which do not take the source of data and code into account at all. We allow the system to change and evolve in an organic way, while at the same time protecting it from arbitrary changes from users. This work helps to demonstrate that meaningful trust requires attestation of a system in its entirety, but it also requires allowing changes, as even minimal systems are not entirely static.

Additionally, we explore the security goals likely to be present in a distributed, trusted virtualization model. Users of VMs have a set of security and resource goals that they must fulfill, while machine owners may wish to enforce resource controls based on coalition identity. We provide an example of how to negotiate both sets of constraints simultaneously, while ensuring that the security goals of the user are enforced, regardless of the VM's location.

Thus, this work serves to both enhance the usability and security of distributed, trusted virtualization. Although more work is necessary to fully realize the potential of this idea, it is clear that this approach enables distributed security with guarantees that were not previously possible. In the future, we hope these principles will be applied to a wide variety of disciplines, including hosting, distributed computation, internet save/restore, and commodity computing.

Clearly, secure, distributed computing holds much promise, and given the current direction of computing technology, it will soon become a requirement to enable the full suite of applications desired by users.

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] Secure virtual machine technology. http://www.amd.com/.

[4] AMD. *Pacifica Technology: Secure Virtual Machine Architecture Reference Manual.* AMD, 2005.

[5] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[6] 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.

[7] 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.

[8] Ross J. Anderson and Markus G. Kuhn. Tamper resistance – a cautionary note. In *USENIX Workshop on Electronic Commerce Proceedings*, pages 1–11, Oakland, CA, November 1996.

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

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

[11] Stefan Berger, Ramón Cáceres, Dimitrios Pendarakis, Reiner Sailer, Enriquillo Valdez, Ronald Perez, Wayne Schildhauer, and Deepa Srinivasan. Tvdc: managing security in the trusted virtual datacenter. *SIGOPS Oper. Syst. Rev.*, 42(1):40–47, 2008.

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

[13] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW '01: Proceedings of the 14th IEEE workshop on Computer Security Foundations*, page 82, Washington, DC, USA, 2001. IEEE Computer Society.

[14] Bruno Blanchet. From secrecy to authenticity in security protocols. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 342–359, London, UK, 2002. Springer-Verlag.

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

[16] Matt Blaze, Joan Feigenbaum, and Jack 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.

[17] Andrea Bottoni, Gianluca Dini, and Evangelos Kranakis. Credentials and beliefs in remote trusted platforms attestation. In *WOWMOM '06: Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, pages 662–667, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Ali Raza Butt, Sumalatha Adabala, Nirav H. Kapadia, Renato J. Figueiredo, and José A. B. Fortes. Grid-computing portals and security issues. *J. Parallel Distrib. Comput.*, 63(10):1006–1014, 2003.

[19] David Carroll Challenger. Managing private keys in a free seating environment. `http://www.freepatentsonline.com/7095859.html`, august 2006.

[20] Avik Chaudhuri and Martín Abadi. Formal security analysis of basic network-attached storage. In *FMSE '05: Proceedings of the 2005 ACM workshop on Formal methods in security engineering*, pages 43–52, New York, NY, USA, 2005. ACM.

[21] P. Chen, C. Aycock, W. Ng, G. Rajamani, and R. Sivaramakrishnan. Rio: Storing files reliably in memory, 1995.

[22] Peter M. Chen, Peter M. Chen, and Brian D. Noble. When virtual is better than real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[23] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, Seattle, WA, USA, March 2008.

[24] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. 14th USENIX Security Symposium*, August 2005.

[25] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[26] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[27] Intel Corp. Intel trusted execution technology. `http://www.intel.com/technology/security/index.htm`.

[28] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In *In Proc. of Workshop on Formal Methods and Security Protocols*, 1998.

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

[30] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. Building the ibm 4758 secure coprocessor. *Computer*, 34(10):57–66, 2001.

[31] William Enck, Sandra Rueda, Joshua Schiffman, Yogesh Sreenivasan, Luke St. Clair, Trent Jaeger, and Patrick McDaniel. Protecting Users From "Themselves". Technical Report NAS-TR-0073-2007, Network and Security Research Center, Department of Computer Science and Engineering, Pennslyvania State University, University Park, PA, USA, June 2007.

[32] Timothy Fraser. Lomac: Low water-mark integrity protection for cots environments. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 230, Washington, DC, USA, 2000. IEEE Computer Society.

[33] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan 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, October 2003.

[34] Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: security challenges in virtual machine based computing environments. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 20–20, Berkeley, CA, USA, 2005. USENIX Association.

[35] Scott Garriss, Ramón Cáceres, Stefan Berger, Reiner Sailer, Leendert van Doorn, and Xiaolan Zhang. Towards trustworthy kiosk computing. In *HOTMOBILE '07: Proceedings of the Eighth IEEE Workshop on Mobile Computing Systems and Applications*, pages 41–45, Washington, DC, USA, 2007. IEEE Computer Society.

[36] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proc. 12th NIST-NCSC National Computer Security Conference*, pages 305–319, 1989.

[37] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–284, New York, NY, USA, 1997. ACM.

[38] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, JUN 1974.

[39] Trusted Computing Group. Tcg tpm specification. `https://www.trustedcomputinggroup.org/home`.

[40] Trusted Computing Group. `http://www.trustedcomputinggroup.org/`, March 2005.

[41] Saikat Guha and Paul Francis. An end-middle-end approach to connection establishment. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 193–204, New York, NY, USA, 2007. ACM.

[42] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proc. 17th USENIX Security Symposium*, July 2008.

[43] J. Alex Halderman and Brent Waters. Harvesting verifiable challenges from oblivious online sources. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 330–341, New York, NY, USA, 2007. ACM.

[44] James Hendricks and Leendert 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.

[45] Boniface Hicks, Luke St. Clair, Sandra Rodrigues, Trent Jaeger, and Patrick McDaniel. A Logical Specification and Analysis for SELinux MLS. In *12th ACM Symposium on Access Control Models and Technologies (SACMAT)*. ACM, June 2007. Sophia Antipolis, France.

[46] IBM. Ibm research - gsal - trusted computing. `http://domino.research.ibm.com/comm/research\_projects.nsf/pages/gsal.TCG.html/`.

[47] IBM. Integrity measurement architecture. http://domino.research.ibm.com/comm/research_projects.nsf/pages/

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

[49] IBM Corporation. *IBM System/360 Principles of Operation.* pub-IBM, eighth edition, 1968.

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

[51] VMware Inc. Vmware esx 3i. `http://www.vmware.com/products/vi/esx/esx3i.html`.

[52] 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.

[53] Trent Jaeger, Serge Hallyn, and Joy Latten. Leveraging IPsec for mandatory access control of Linux network communications. Technical Report RC23642 (W0506-109), IBM, June 2005.

[54] Trent Jaeger, Patrick McDaniel, Luke St. Clair, Ramon Caceres, and Reiner 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.

[55] 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.

[56] Li Junrang, Wu Zhaohui, Yang Jianhua, and Xia Mingwang. A secure model for network-attached storage on the grid. In *SCC '04: Proceedings of the 2004 IEEE International Conference on Services Computing*, pages 604–608, Washington, DC, USA, 2004. IEEE Computer Society.

[57] Yasuharu Katsuno, Yuji Watanabe, Sachiko Yoshihama, Takuya Mishina, and Michiharu Kudoh. Layering negotiations for flexible attestation. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 17–20, New York, NY, USA, 2006. ACM.

[58] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.

[59] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998.

[60] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[61] 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, February 2003.

[62] Hans Lhr, HariGovind V. Ramasamy, Ahmad-Reza Sadeghi, Stefan Schulz, Matthias Schunter, and Christian Stble. Enhancing grid security using trusted virtualization. In Bin Xiao, Laurence Tianruo Yang, Jianhua Ma, Christian Mller-Schloer, and Yu Hua, editors, *ATC*, volume 4610 of *Lecture Notes in Computer Science*, pages 372–384. Springer, 2007.

[63] J. Marchesini, S.W. 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.

[64] J. Marchesini, S.W. 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.

[65] 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.

[66] Jonathan M. McCune, Trent Jaeger, Stefan Berger, Ramon Caceres, and Reiner Sailer. Shamon: A system for distributed mandatory access control. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 23–32, Washington, DC, USA, 2006. IEEE Computer Society.

[67] Jonathan M. McCune, Trent Jaeger, Stefan Berger, Ramon Caceres, and Reiner Sailer. Shamon: A system for distributed mandatory access control. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 23–32, Washington, DC, USA, 2006. IEEE Computer Society.

[68] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, 2008.

[69] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing Network Virtualization in Xen. In *USENIX Annual Technical Conference*, 2006. Best paper award.

[70] Robert Meushaw and Donald Simard. NetTop: Commercial technology in high assurance applications. *Tech Trend Notes*, 9(4):1–8, 2000.

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

[72] Jr. Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association.

[73] Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. In *VirtSec*, 2007.

[74] J. Robin and C. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor, 2000.

[75] Terry Rooker. The reference monitor: an idea whose time has come. In *NSPW '92-93: Proceedings on the 1992-1993 workshop on New security paradigms*, pages 192–197, New York, NY, USA, 1993. ACM.

[76] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. May 2005.

[77] Ahmad-Reza Sadeghi, Marcel Selhorst, Christian Stüble, Christian Wachsmann, and Marcel Winandy. Tcg inside?: a note on tpm specification compliance. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 47–56, New York, NY, USA, 2006. ACM.

[78] 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, December 2005.

[79] 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, August 2004.

[80] Beata Sarna-Starosta and Scott D. Stoller. Policy analysis for security-enhanced linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, April 2004. Available at http://www.cs.sunysb.edu/˜stoller/WITS2004.html.

[81] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. *SIGOPS Oper. Syst. Rev.*, 39(5):1–16, 2005.

[82] Elaine Shi, Adrian Perrig, and Leendert 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.

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

[84] Parallels Virtualization Software. Parallels - virtualization for os x. `http://parallels.com`.

[85] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor.

[86] Sun Microsystems. Trusted Solaris 8 Operating System. `http://www.sun.com/software/solaris/trustedsolaris/`, February 2006.

[87] J. D. Tygar and Bennet Yee. Dyad: A system for using physically secure coprocessors. In *IP Workshop Proceedings*, 1994.

[88] Brent Waters, Ari Juels, J. Alex Halderman, and Edward W. Felten. New client puzzle outsourcing techniques for dos resistance. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 246–256, New York, NY, USA, 2004. ACM.

[89] Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[90] C. Wright, C. Cowan, J. Morris, S. Smalley, G. KroahHartman, s modules, and G. support. the linux kernel. in linux security modules: General security support for the linux kernel, 2002.

[91] Xensource. Xen-changelog info page. `http://lists.xensource.com/mailman/listinfo/xen-changelog`.

[92] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Evaluating the performance impact of xen on mpi and process execution for hpc systems. In *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 1, Washington, DC, USA, 2006. IEEE Computer Society.

[93] Xin Zhao, Kevin Borders, and Atul Prakash. Svgrid: a secure virtual environment for untrusted grid applications. In *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing*, pages 1–6, New York, NY, USA, 2005. ACM.

[94] Dennis Zimmer. *VMware Server and VMware Player. The way forward for Virtualization.* BoD, 2006.