The Pennsylvania State University
The Graduate School

PRACTICAL SYSTEM INTEGRITY VERIFICATION IN CLOUD COMPUTING ENVIRONMENTS

A Dissertation in
Computer Science and Engineering
by
Joshua Serratelli Schiffman

© 2012 Joshua Serratelli Schiffman

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2012

The dissertation of Joshua Serratelli Schiffman was reviewed and approved* by the following:

Trent R. Jaeger
Associate Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Patrick D. McDaniel
Professor of Computer Science and Engineering

Bhuvan Urgaonkar
Associate Professor of Computer Science and Engineering

Constantino Lagoa
Professor of Electrical Engineering

Lee Coraor
Associate Professor of Computer Science and Engineering
Graduate Office

*Signatures are on file in the Graduate School.

Abstract

Online applications have become the *de facto* medium through which modern computing services are offered. This model not only reduces administrative costs, but enables companies to shift their physical infrastructure to virtualized environments like cloud computing platforms. However, with this move to remotely administered services come serious risks. Since users no longer control the systems they rely upon, they must *assume* they were correctly configured to protect their sensitive data. As history has demonstrated, even the most well funded companies are prone to compromises, which may lead to the loss of countless confidential customer records. If the world is to continue adopting this computing model, then a greater emphasis must be placed on building verifiable systems that customers can inspect.

In this dissertation, we explore the design challenges in building verification frameworks that overcome the limitations of current verification techniques for detecting unsafe and compromised systems. Existing approaches leverage trusted computing hardware like the Trusted Platform Module (TPM) to securely record and attest to integrity-relevant events occurring on the proving system. However, these approaches are insufficient for verifying today's high performance and highly connected environments. First, we developed the Root of Trust for Installation, a method for bootstrapping trust in virtual machine (VM) hosts that form the basis of many cloud offerings. Second, we designed a remote integrity verifier to address many of the difficulties that attestation-only verification causes. Using this Integrity Verification Proxy, we are able to verify heterogeneous integrity requirements at the proving system without the delay and complexity of traditional integrity measurement. Finally, we incorporated our research into the Cloud Verifier, a framework for verifying the integrity of instances hosted on clouds. This permits cloud administrators, customers, and external clients to verify integrity criteria without having to directly inspect the configuration of the entire platform. Our proof-of-concept implementation and evaluation demonstrates the feasibility of building a verifiable, yet functional cloud platform. While this work represents only a starting point, we believe it will lead to a greater understand of how today's online services can be designed in a more transparent way.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This dissertation is the culmination of my work over the last six years, but to me, it has come to represent perhaps one of the most significant milestones in my life. Throughout my time in graduate school, I have learn lessons academic, social, economic, and deeply personal that have shaped me in ways that only the crucible of this crazy research driven life can accomplish. It cannot be overstated that, if it were not for the support, guidance, and inspiration I received from my friends, family, and colleagues during that time, I would never have made it this far.

First, I would like to thank my advisor, Trent Jaeger, for his guidance throughout these long years. It was his strong interest in my abilities that brought me to the graduate program at Penn State. He has been a role model for me in my academic career and has instilled in me the importance of doing foundational research with a profound impact on science as a whole. He has been a source of certainty even during the darkest days when I was questioning the value of my work and believed in me when I did not. I would be remiss if I forgot to mention his wife Dana, who would invite our lab to their house for the holidays or just to watch a football game. She has always been a kind and thoughtful person and it has been a pleasure to think of both of them as my friends.

I would also like to thank the other members of my dissertation committee. Patrick McDaniel has been one of the most influential researchers in my academic life. His intense energy, strong principles of rigorous evaluation, and uncanny ability to cut through the technical muck and reveal the underlying narrative, has guided me through my development as a researcher. He has also been a dear friend and sympathetic to the troubles I have faced. Even in the most trying times, when the impossible deadlines loomed before me, he recognized the importance of getting away to relax, refresh, and to grab a cup of coffee. I would also like to thank his wife, Megan, for hosting countless events and bringing food to our starving lab. Bhuvan Urgaonkar has been helpful in directing my research in the area of distributed systems and has always sparked interesting conversations. Finally, I would like to thank Constantino Lagoa for agreeing to be my outside member, even with his busy schedule. I feel his unbiased comments have greatly improved the quality of this dissertation.

Of course, faculty alone do not ensure successful completion of a Ph.D. dissertation. I have my strong family of friends in the Systems and Internet Infrastructure Security

(SIIS) lab to thank for that. From my first day, Patrick Traynor, Kevin Butler, and Will Enck became ever watchful mentors, who continue to provide me with invaluable support and direction that I will never be able to repay. Patrick taught me much of leadership and developing a research plan. Kevin demonstrated great compassion and was always there to listen to my concerns. Will amazed me with his technical skills and deep understanding of the Linux kernel. Tom Moyer was a constant friend to me throughout the program. He shared with me the burdens of trusted computing, the frustrations of finicky hardware, and the intoxicating effects of too much caffeine and terrible fast food. Never will I forget a HotFriday. Luke St. Clair was my first collaborator and he taught me much of the trials and tribulations of graduate school. Machigar Ongtang always made me laugh and made life in the lab a little bit happier. Steve McLaughlin has been an amazing student, researcher, and friend. Our conversations could wander for hours on topics far and wide. We should all hope to hold our work to his level of precision and scrutiny. Divya Muthukumaran joined the Ph.D. program with me and experienced the same hills and valleys as I did. She has a creative mind, is wildly talented, and I am glad to have worked on several papers with her. She is certainly bound for great success.

I want to give special thanks to three students. Mike Lin and I have had some crazy adventures across the globe. I'll never forget his Python-fu and, what I can only assume is a fashion-forward dress style. Dave King, is an amazing friend. I met Dave in undergrad and was surprised not only to see him for his last years as a Ph.D. student in graduate school, but as a student under the same advisor as me. I've learned much about coping, sports, coding, and politics from him and avidly follow his blog (when he updates). Last, but not least, I want to thank Hayawardh Vijayakumar. While I have worked with many students to complete some of my more insanely complicated systems, Hayawardh has been the most impressive of the bunch. His ability to absorb simply staggering amounts of technical detail is a sight to behold. No other student could bring himself up to speed to work in my niche area as he has, and make significant contributions to the project on a regular basis. Even though he complains his mind is going and says he cannot code to save his life, his body of work says otherwise.

I have also been blessed with many opportunities outside of Penn State, which have balanced my academic knowledge with industry experience. Reiner Sailer at IBM has been extremely helpful in my early years when I was struggling with the TPM and learning how to hack the kernel. His jovial demeanor and honest opinions have made my time at IBM and conferences enjoyable. Xinwen Zhang has become one of my closest collaborators outside of Penn State. At Samsung, we worked on numerous projects that resulted in several papers and even my first patent application. I always make it a point to meet up with Xinwen at conferences to grab lunch and talk about future work. My final internship at Microsoft Research introduced me to Jay Lorch and James Mickens. They, along with Bryan Parno, who I knew through my TPM work, were the best managers and friends I have ever had at an internship. I learned a great deal from how we brainstormed ideas and had a blast solving puzzles and playing Dominion at lunch. I will always think fondly of those times.

On my journey to graduate school, I met many dear friends who have helped prepare me for graduate school and teach me how to enjoy life. In undergrad, I met Chris "The Hammer" Cavage, who has the most amazing musical and coding talent I have ever known. Chris introduced me to a circle of friends, which we lovingly call RISE. Justin Dunmyre has been my mentor as a computer scientist and a researcher. His ability to literally and figuratively juggle an inordinate number of objects and responsibilities as a student, husband, and teacher has encouraged me to push on when I was overburdened. Our shared passion for games, mathematics, coding, and Japanese made us fast friends, but his patience and wisdom is what I cherish the most. His wife Elizabeth has also been a wonderful friend and provided desperately needed helped during my rushed wedding. Bruce Shearer was my roommate during my senior year. I saved him from a horrid housing situation and he gave me perhaps one of the closest friendships I have ever had. He has been with me through the most bizarre adventures from Boston at PAX to the Village Traders & Refreshment Outpost in Orlando. There is virtually nothing I wouldn't do for Bruc(e), except maybe getting him that sandwich. Devin Ronge has been another friend I could not have lived without. He was there for me through my internships, graduate school, and all the times in between. His coding prowess knows no bounds and is always available at the drop of a hat to help. I would also like to thank Erik Jensen, Mark "VOR" Noecker, and Celina Pettis for being great friends and putting up with me for all of these years.

Before undergrad, I also had an amazing collection of friends who defined my early life. I want to thank John Haste, my best friend, for being there for me since we were two years old. Erik Steigler introduced me to the concept of a LAN party and administered the servers that would lead me to my Wife. I want to thank Dave Rudy for his insatiable lust for virtual resource acquisition, Jason Dietrick for his great taste in music and phenomenal artistic skills, and Matt Martz, who showed me the way of rock climbing, arcade cabinets, and Buddha.

I would like to give my deepest gratitude to my family. To my mother Lori Serratelli, you taught me the importance of humility, compassion, and to listen when others simply talk. Your courageous fight through cancer, heart surgery, and two political campaigns, while still raising a family and running a law firm, is the stuff of legends. I would be glad to achieve just a fraction of what you have accomplished. My father, Steven Schiffman, introduced me to computers and taught me to enjoy life. His unwavering integrity has given me a strong respect for both justice and leniency. He is my role model as a human being and I will always strive to be like him. My brother, Andrew Schiffman, has been with me for nearly my whole life. While our teenage years may have been rough, he have always shared many of the same passions that I have. He has matured into an amazing person and I hope to always be your friend.

I would like to conclude with my most sincere appreciation to my wife, Mira Hidajat. When I met Mira, little did I know that we would become such close friends. Watching her go through her dissertation writing and defense greatly prepared me for what was to come. It is her support and love that carried me over the finish line and I am excited

to live the rest of our lives together.

Dedication

To Mom and Dad, for making this possible, to Andy, for being my lifelong friend, and to Mira, cinta kamu sayangku.

Chapter 1

Introduction

Computers have revolutionized nearly every aspect of modern life. Connected over networks small and large, they enable the transfer of massive amounts of data and perform countless calculations across vast arrays of machines. This has rendered previously insurmountable tasks trivial. The next computing revolution has manifested in the plethora of online services that have appeared, which range from simple web applications to sophisticated cloud computing platforms. The rise of these convenient and inexpensive services has shifted traditionally locally administered installations to remote server farms and data centers.

However, this outsourcing has led to an insidious yet understated problem. We have come to rely upon remote systems outside of our control without ever checking how they were configured. Without confirmation they are behaving as expected and without compromise, our trust in these networked services is often unwarranted. Yet, users frequently rely on them for storing, processing, and redistributing sensitive data. Moreover, a myriad of vulnerabilities exist that have and will continue to be exploited by malicious attackers at any time. Recent reports of data theft at data centers [1] and widespread botnet infection rates [2] should increase our distrust in these networked systems.

Much effort in both the research and commercial communities has been dedicated to hardening systems from would be attackers. Work on strong mandatory access control (MAC) systems [3, 4, 5], security kernels [6], security enhanced languages [7], and code signing [8] have significantly raised the bar for attackers and reduced the attack surfaces they can exploit. Similarly, privilege separation [9], virtualization [10], and sandboxing [11] techniques have further mitigated the damage that untrusted or compromised

entities can wreck. Unfortunately, administrative error and unforeseen vulnerabilities still permit dedicated attackers to circumvent and disable these protections without detection. To address this, hardware-assisted *secure boot* approaches [12] and formal assurance [6, 13] have been employed to ensure systems execute only trustworthy and provably secure code as dictated by administrators. However, these techniques confer no evidence of these guarantees to remote parties, which further motivates need for verifiable systems.

1.1 Why Trust is Misplaced

To understand why users are willing to accept data from potentially malicious parties, we must examine how trust decisions are made. In the absence of concrete information, people often fall back to more traditional forms of establishing trust. Name recognition and reputation are primary examples that people use to assert trust in remote systems. On the Internet, DNS names give an identity to IP address that can be associated with a real world entity. Trust in that entity's reputation to protect their systems from harm and to uphold service level agreements (SLAs) with the client usually placate most clients' fears or at least offer possible recourse when a breach of contract occurs.

However, attacks on DNS [14, 15], whereby an attacker's IP address is substituted for the legitimate one, have rendered DNS name associations invalid. The rise of Public Key Infrastructures (PKIs) [16], created through certificate authorities (CAs) like Thawte, Verisign, and Comodo, have enabled clients to transparently authenticate systems and establish secure communication channels [17, 18] despite hostile networks. Thus, cryptography and trust in third party authorities have only strengthened the trust asserted by name alone. Moreover, advances in cryptographic hardware [19, 20] further minimize the risk of stolen private keys that stand to further undermine the guarantees of PKIs.

One drawback of trust by reputation alone is that it says nothing about the integrity of the underlying systems for which that identity speaks. Despite this, network communication and the services they facilitate continue to thrive unverified for two main reasons. First, users have largely ignored the threat of relying on unverified and potentially malicious remote systems. This is evidenced by the countless worms, viruses, trojans, and botnets that plague the Internet, but elicit little demand for integrity verification. The second and more interesting reason is the false sense of security that secure communication provides. Strong cryptographic protocols like SSL [17] and IPsec [18] enable clients to establish communication channels to servers with guarantees of confi-

dentiality, integrity, authenticity, and non-repudiation. However, these protections only extend to data in-flight. A compromise at the host or theft of the public key used to authenticate the connection can still subvert the integrity of data before it ever reaches the channel. However, for the average user, a valid SSL connection implies, incorrectly, the correctness of all data over that connection.

Trust in a system's integrity implies a belief that the system is functioning in a manner expected by the relying party. Assessing integrity of remote systems is challenging because of the dynamic nature of computers and the difficulty of inspecting their runtime state. Moreover, the definition of integrity may vary from client to client. Such definitions may also be flawed if a client is willing to trust vulnerable or even malicious software to produce mission critical inputs. For example, a client that trusts an old version of an operating system (OS) with known vulnerabilities is thus willing to accept data from it even though its weaknesses could be exploited at anytime.

Another difficulty is collecting information about a system's current *configuration*. To accurately assess system integrity, a verifier requires knowledge of all code and data involved in producing inputs to the relying system. Reliably obtaining such evidence from a system that may already be compromised is nearly impossible without a tamper-proof reporting mechanism. Furthermore, today's servers are highly interconnected and consist of numerous applications that receive data from multiple external sources. As a result, a user implicitly relies on the integrity of numerous and potentially unknown systems when interacting with networked services, but verifying those systems is difficult in practice.

1.2 Integrity Measurement

To address these challenges, researchers have explored techniques for gathering and reporting information about a system's current configuration. This information gives remote verifiers evidence necessary to trust these remote systems [21]. Integrity Measurement (IM) is the name given to the broad class of techniques for recording critical system events that affect a system's integrity. IM was born out of work that attempted to convince remote clients of the integrity of secure cryptographic coprocessors like the IBM 4758, called *Outbound Authentication (OA)* [22]. Through carefully designed hardware protections and controlled software deployment, these single-purpose devices could provably demonstrate to clients that a particular set of signed code had loaded. This certificate chain formed an *attestation*, by the co-processor, of the history of events that led

to its current configuration. Using this, clients could obtain a proof that a 4758-enabled webserver was using trustworthy code before releasing sensitive data to it. What the OA model demonstrated was that a system could be trusted, not because it was guaranteed to load trustworthy configurations (i.e. secure boot), but because it could prove it arrived at a trusted configuration. This notion that a system is trustworthy after proving it booted into a trustworthy configuration is referred to as *authenticated boot*.

The OA model also revealed that reliable IM techniques require the ability to gather measurements independent of the underlying system's integrity. Later, specialized hardware approaches like Copilot [23] used system bus monitoring to record events. Others used a strictly software-based approach that detected tampering by timing code execution [24, 25]. However, the utility of these IM solutions was too limited because of their reliance on either costly specialized hardware that most consumers would not purchase or highly specific and constrained system deployments.

Ultimately, any effective and flexible approach for attesting to system integrity in commodity systems would require some hardware-based root of trust to bootstrap the trust establishment processes. The Trusted Computing Group (TCG), successor to the Trusted Computing Platform Alliance (TCPA), filled this requirement with the production of the inexpensive and widely deployed Trusted Platform Module (TPM) [20]. This passive, low-power cryptographic coprocessor facilitated many trusted computing operations. In particular, the TPM provides tamper-evident measurement logs to support reliable attestations of system integrity and public key generation and management to authenticate the platform. In addition, IM support has been incorporated into much of today's software stack. Starting from the CPU [26, 27], firmware, and bootloader [28, 29], IM extensions extend up into the kernel [30] and application layer [31]. Even support for virtual machine monitors (VMMs) to bootstrap trust in virtual machines (VMs) that lack a physical platform [26, 27, 32] has been added.

1.3 Verification Challenges

Even with the groundwork laid, current IM schemes are insufficient for proving general integrity requirements in a flexible and practical manner. Here we enumerate these challenges and how they impede integrity verification.

- **Current TPM-based IM approaches do not convey semantically meaningful evidence that satisfies general integrity requirements**. Verifiers are free to define arbitrary integrity targets with heterogeneous requirements. Thus,

IM frameworks must be designed to collect sufficient integrity-relevant information to satisfy these possible targets. However, most IM implementations simply gather hashes of files deemed important by a particular administrator. This leaves the burden of interpreting a system's integrity from insufficient data. This is often insufficient to satisfy rigorous integrity targets based on classic and practical integrity models [33, 34, 35, 36] that require more than lists of "good" file hashes to assess system integrity. This includes verifying arbitrary files, system updates, runtime attacks on code integrity, and external data, which may lead to further integrity violations.

- **Attestations do not guarantee ongoing integrity.** Since attestations only represent a snapshot of the system's configuration, future events that degrade its integrity will be missed by the verifier. As a result, many protocols that incorporate IM require either periodic re-attestation or implicit trust the system's integrity will never change. The former leads to more complicated and costly communication, while the latter may miss compromises after initial verification altogether. What verifiers need is a means providing long-term integrity guarantees without the need for repeated verification.
- **It is impractical to record and report all integrity-relevant operations.** Generating frequent attestations is not just a burden for the verifier, but also for the attesting system. Since the TPM is a slow, commodity coprocessor, it cannot physically handle generating more than one attestation per second in most implementations [37]. This is a non-starter for most production servers that handle thousands of clients a second. The issue stems from the fact that the TPM was designed to facilitate an authenticated boot model, whereby remote verifiers obtain an attestation proving the system booted into a trustworthy distribution. However, as we stated above, general purpose platform configurations do change at runtime and thus additional measurements must be recorded. Because of the TPM's poor performance, it is impractical to send all events to a remote verifier to constantly monitor the proving system's integrity. As a result, most IM frameworks focus on less frequent events that are simple to verify. This again reduces the effectiveness of current IM techniques by excluding potentially pertinent information.
- **Systems today rely on a multitude of interrelated virtual and physical machines.** Even with a reliable method of verifying a single system, remote services are facilitated by numerous systems. Moreover, they are often virtualized platforms that appear and disappear to meet demand. Common computing platforms like cloud make it even more difficult to assess remote system integrity because they are opaque and hide the underlying hosts that spawn and move these

virtual machine instances. They are also hesitant to reveal the exact code and data that go into configuring their systems and typically deny direct access to the machines. If we are to verify remote service integrity, we must account for both the application and the entire platform that contributes to it.

In addition to these challenges, other issues can undermine the effectiveness of integrity verification. Since integrity is based on the subjective trust assumptions each verifier uses, we do not attempt to guarantee that all integrity targets can be verified or even met. Moreover, we do not address attacks on hardware or trust assumptions made by the verifier that are incorrect. Given these limitations, we consider proving the security of general purpose systems is out scope for our work.

1.4 Thesis Statement

Establishing trust in remote systems will become more important as a greater number of services and sensitive computation is outsourced to external administrative domains. While secure communication and current IM approaches have begun to explore this space with limitations, our insights suggest it is possible to achieve more useful integrity guarantees. This leads to the thesis of this work:

It is possible to efficiently verify distributed system integrity for stakeholders with heterogeneous integrity requirements.

By stakeholders, we mean the relying parties that depend upon the integrity of the system they wish to verify. This document answers the following three research challenges that lead to a solution that demonstrates the veracity of this claim.

- *How can we establish trust in the initial integrity of general purpose systems that depend on large trusted computing bases?* Key to verifying system integrity is establishing the system's initial integrity at boot. However, traditional IM approaches that rely on known good configuration values are insufficient for most servers that contain multiple applications, users, and sources of data that are system specific and evolve over time. Through our work on designing verifiable roots of trust, we have found it is possible to provably bind the integrity of an entire filesystem to a trusted installer. In Chapter 3, we demonstrate how these techniques can be applied through physical presence and over an untrusted network to produce verifiable system installations.

- *How can we leverage enforcement and comprehensive integrity measurement to reduce verification effort for relying parties?* Our work on the IVP in Chapter 4 enables the creation of secure communication channels that ensure the channel's endpoint meets a remote verifier's integrity criteria for the duration of the connection. By having the IVP check a system's integrity on behalf of the relying party, compromised systems can be cut off before they can transmit low integrity data. By combining various IM techniques coupled with enforcement at the channel's endpoint, we show it is possible to support a diverse set of integrity criteria without forcing the relying party to repeatedly request and comprehend complex attestations.
- *How can we extend integrity verification efficiently into distributed systems?* Today's services are supported by large networks of cooperating servers and cloud computing platforms. It is difficult to verify the integrity of all systems involved in a computation because a verifier may not be able to directly access those systems. We demonstrate a multilayered approach for verifying cloud computing infrastructures in Chapter 5, which exemplify today's distributed computing platforms. We leverage our practical integrity criteria to establish transitive trust among the cloud's components and show how it can support a large number of clients despite the prohibitive overhead of the TPM.

1.5 Opportunities for Improving Integrity Verification

Current IM approaches are insufficient for constructing verification framework capable of verifying heterogeneous requirements in large distributed platforms like clouds. The research presented in this thesis goes beyond the ideas of current techniques by leveraging the several key insights. Throughout the remaining chapters, these insights will guide our design decisions as we build the IM frameworks necessary to prove our thesis.

- **Comprehensive integrity monitoring supports more targeted and expressive integrity requirements.** A verifier's integrity target defines the set of requirements a proving system must satisfy to be considered trustworthy. However, current IM frameworks are too ad hoc to support the many of these requirements like runtime integrity measurements. Instead, IM frameworks should incorporate more fine-grain integrity monitoring techniques like runtime memory introspection to broaden the integrity-relevant events that can be observed. While not all integrity targets will require such fine-grained monitoring, an effective approach should be flexible enough to satisfy many integrity requirements.

- **Enforcement reduces a system's measurement surface.** While processes can perform a variety of operations that impact integrity, not all of them will lead to an integrity violation. Unfortunately, a verifier cannot be sure they were benign without inspection or some guarantee they will not cause a violation. By mediating integrity-relevant operations to deny unsafe actions, the verifier only needs to inspect the integrity of the reference monitor that implements this mediation and its policy. This shrinks the system's "measurement surface" thereby reducing the number of operations that must be monitored by the IM framework and assessed by the verifier.
- **Basing integrity requirements on practical integrity models enables the design of comparable integrity criteria.** The integrity targets that can be verified are limited by what can be observed through a system's IM framework. Defining a integrity requirements based on an ad hoc set of measurements may not yield meaningful integrity properties and makes it difficult to compare the integrity of systems with heterogeneous configurations. Thus, using an integrity target that is both meaningful and comparable is crucial to verifying remote systems. Integrity models provide strong integrity guarantees on which to base a verifier's *integrity criteria*, but classic models like Biba strict [33] and Clark-Wilson [34] are too difficult for general-purpose systems to satisfy. Instead, we find practical integrity models [35, 38, 36] allow for more flexible integrity enforcement. Basing criteria on such models enables verifiers to compare criteria and determine transitive trust relationships among verified systems.
- **Attestation is just a starting point.** Attestation is frequently used as the sole mechanism for verifying remote systems, which leads to various design issues that are difficult to overcome. First, attestation was designed to report a snapshot of the proving system's configuration (e.g., at boot time) and thus is ill suited for maintaining a consistent view of a proving system's configuration. Second, it is difficult to reason about the semantics of measurements taken within a complex system and sent via attestation because the verifier may not have the necessary context to do so. Finally, as we have mentioned before, current trusted hardware is not performant enough to handle attestation as a high frequency operation. Instead, we find that attestation should be limited to just verifying a small, protected, and *static* component within the proving system. This component can then act as a trust anchor for the remote verifier on top of which more sophisticated verification can be performed.

Chapter 2

Trusted Computing Background

In this chapter, we present the core concepts of integrity verification. Throughout this dissertation, we will refer to *integrity criteria* as the specification that a verifier defines for assessing the trustworthiness of remote systems. Integrity measurement focuses on collecting the integrity-relevant information on a proving system needed to verify that the criteria is satisfied. Finally, *integrity reporting* deals with transmitting that information to remote verifiers securely. After discussing criteria definitions, we detail early IM techniques like secure boot systems and specialized hardware devices. We then introduce the TPM and explain how it has taken a central role in IM research and provide an overview of current techniques that employ it. This will motivate our current work in the next chapters.

2.1 Defining Integrity

A system's configuration is defined by the combination of executing code and data that controls its behavior. Malicious programs, malformed inputs, and sinister users can all subvert the system's intended behavior and thus its integrity. A system's perceived integrity expresses a relying party's trust in that system to provide correct inputs and function as expected. Accepting inputs from untrusted systems can lead to further integrity degradation if those inputs are able to further compromise the relying systems. Integrity verification aims to provide clients with a means of assessing remote system integrity before interacting with such systems.

2.1.1 Integrity Targets

In order to assess system integrity, a verifier defines an *integrity target* that the proving system (we will call the prover) must meet. A common integrity target used by the TCG [39], an industry group focused on developing trusted infrastructure standards, is a system where all executed code starting from boot-time comes from a trusted distribution. While this target is highly subjective, it is commonly used by TCG-like IM protocols [30, 40, 41, 42, 43, 31].

Targets that are more principled have been proposed [41, 44] that aim to meet computable integrity models [33, 34, 36, 35, 38]. These models specify information flow requirements among system processes and object to protect system integrity at runtime. Classic integrity models like Biba strict [33] required all trusted processes to execute only formally assured [13] code and only receive inputs from sources of an equal integrity class or higher. Clark-Wilson [34] permitted these processes to handle lower integrity data from untrusted sources through formally assured guard processes that would discard or *endorse* the data's integrity by upgrading its integrity class.

While these early models offered a strict definition of integrity, they are often difficult if not impossible to satisfy in practice. One reason is that trusted processes must often receive lower integrity data. Consider a trusted webserver process running Apache that services requests over a network. In general, network borne data is untrustworthy because the integrity of its origin is hard to ascertain. As a result, Apache must accept untrusted inputs and thus violate the integrity model. Even if the developers had hardened the program from malicious network data, the requirement for formal assurance is often too prohibitively expensive and time consuming to be practical.

Recent IM approaches have looked to practical integrity models like Usable Mandatory Integrity Protection (UMIP) [38], Practical Proactive Integrity Protection [35], and Clark-Wilson Lite (CW-Lite) [36] as a more feasible way of defining integrity targets. Practical integrity models aim to maintain system integrity similar to classic models, but provide solutions to tricky edge conditions where untrusted data must be accepted into the system. They do this by leveraging user and application developer knowledge to act as authorities for determining which processes and interfaces are trusted to handle low integrity inputs. For example, the UMIP model uses discretionary access control (DAC) permissions on binaries to label the integrity of processes in which they execute, while the CW-Lite model lets programs indicate through a system call when they are using filtering interfaces trusted to handle low integrity data.

2.1.2 Integrity Criteria

Once an integrity target has been chosen, a verifier must decide how to prove a remote system satisfies it. *Integrity criteria* specifies the set of *requirements* and *assumptions* the verifier uses make this decision. As discussed above, requirements are informed by the verifier's integrity target, but may not be straightforward to test given the information provided by the prover. Consider the Biba integrity model [33], which requires all trusted processes execute only formally assured code as one requirement. The verifier could decide to require the prover to present evidence that only trusted code binaries are present on its system at boot-time. However, this might be overly restrictive for a system with strong access control to confine untrusted code. Instead, she could request a log of all executed binaries, but this would not account for code loaded later. In practice, researchers have typically designed their requirements around what can be collected by the available IM approach in place on the prover. We will discuss this more in the following section and the effect it has on integrity verification.

Another challenge to designing requirements is their complexity. Some properties like the trustworthiness of code are not simple to assess. To address this, verifiers use trust assumptions to ease verification and represent a subjective belief in the correctness of code, data, or even identities. Integrity criteria often rely on authorities to speak for integrity instead of enumerating every assumption. OS distributors, application developers, and corporate administrators are examples of such authorities. One side-effect of the subjective nature of trust assumptions is that a prover meeting one verifier's criteria for an integrity target may not satisfy another verifier's criteria for the same target. Moreover, some criteria may be overly generous and deem even the malicious system as trustworthy.

2.2 Trusted Platform Module

The introduction of the TPM [20] provided a widely deployed commodity platform for building trust in remote systems and facilitating integrity verification. It provides the advantages of a hardware-based root of trust for collecting integrity measurements without requiring administrators to purchase expensive or custom devices. The TPM is part of a larger TCG framework for designing trusted systems, but we focus on the TPM in this thesis as it is the most relevant, flexible, and prevalent component for IM. We now examine the TPM's several key features that administrators and verifiers can leverage for IM protocols.

2.2.1 Cryptographic Operations

Firstly, the TPM is a cryptographic co-processor with RSA, AES, SHA-1 hash [45] functionality. It also contains a set of monotonically increasing counters that increment either manually or on system reboot. The TPM uses a hardware random number generator (RNG) to produce nonces and keying material. A portion of general purpose nonvolatile RAM (NVRAM) is also available for storing persistent secrets or reference data. A special key stored in the TPM, called the Storage Root Key (SRK), is used to encrypt various secrets like RSA key pairs before they are transmitted. This ensures these objects never appear in plaintext outside of the TPM.

2.2.2 Platform Configuration Registers

The TPM's main feature is its bank of *Platform Configuration Registers (PCRs)* for storing integrity measurements. Each 160 bit PCR maintains a SHA-1 hash-chain of integrity measurements[1]. PCRs are reset to an initial value (usually 0) at power on, and are *extended* with measurements. The TPM's extend function updates a specific PCR with the hash of the measurement concatenated with the PCR's current contents. In other words, extend(x, m) takes a PCR number x and measurement m, and updates PCR x:

$$PCR[x] = \text{SHA-1}(m \mid\mid PCR[x])$$

where $||$ is the concatenate operator. Given the list of measurements that went into the PCR, a verifier can reconstruct the aggregate to check the accuracy of the list. Since the PCR cannot be reset except by rebooting the machine, malicious code is prevented from replacing previous measurements with false ones. This is critical for producing a chain of trust where a program measures *before* executing code or receiving inputs.

There are either 16 or 24 PCRs depending on the TPM's specification version. Each register is dedicated to specific types of measurement. PCRs 0-7 are the *static root of trust for measurement (SRTM)* registers. These record the boot-process starting with from motherboard's firmware up to the bootloader. It is called static because it follows the expected, invariant boot sequence. PCRs 8-16 are for general purpose measurements produced during runtime. Finally, 17-22 are special registers called the *dynamic root of trust for measurement (DRTM)* registers. In recent CPUs, special *late launch* instructions [27, 26] have been introduced to essentially reboot the system into a small binary

[1]While the TPM accepts measurements of arbitrary amounts of data, the measurements are always hashed through a SHA-1 function. Thus, we will use the term measurement and hash interchangeably throughout this thesis.

like a bootloader. The CPU's late launch instruction 1) initializes the DRTM registers to a special starting sequence only it can set, 2) measures the binary extends it into a DRTM registers, and 3) passes execution to the code. This enables systems to securely boot into a verified launch environment even if the static boot process is untrusted due to malicious firmware, option ROMs, or virtual machine monitor rootkit [46].

2.2.3 Attestation

The TPM is part of a TCG defined attestation protocol for producing a signed statement called a *quote* of the TPM's PCRs that is bound to the identity of the physical platform. The TPM signs a quote using a short-lived RSA key pair generated by the TPM, called an Attestation Identity Key (AIK). To link this AIK to the physical chip, each TPM has unique Endorsement Key (EK) pair burned into its chip by either the manufacturer or administrator. When a verifier wishes to authenticate an AIK's identity, a trusted third party called a Privacy Certificate Authority (PrivacyCA) vouches for its authenticity by verifying the EK's credentials associated with the AIK. This lets verifiers know which platform produced the AIK without directly linking the AIK to the physical chip. Alternatively, a zero-knowledge proof protocol called Direct Anonymous Attestation [47], can be used to avoid the PrivacyCA altogether.

Once an AIK has been negotiated with the verifier, the attestation protocol proceeds as follows. First, the verifier requests provides a nonce N for freshness and a list of PCRs $\hat{X}$ to report. The TPM generates its quote as $(PCR[\hat{X}], N)_{AIK^-}$, the list of requested PCRs and the nonce signed by the private portion of the AIK in the TPM. The proving system then sends the quote and the list of measurements ML recorded in those PCRs to the verifier. An attestation is then:

$$\text{Attest}(\hat{X}, N) = (PCR[\hat{X}], N)_{AIK^-} + ML$$

The measurement list lets the verifier inspect the recorded events and the PCRs enable the verifier the check integrity of the list. Finally, the signature binds the quote to that specific TPM and nonce, and provides integrity protection.

2.3 Integrity Measurement and Reporting

Defining integrity criteria is only the first part of integrity verification. In this section, we detail the relevant IM techniques and the protocols they used for reporting *attestations*

of system integrity. Integrity measurement approaches aim to gather integrity-relevant information necessary for assessing a verifier's integrity criteria. An IM framework in place on the proving system must do this in a reliable and complete manner. A failure to detect criteria-violating events may lead the verifier to incorrectly believe in the prover's integrity. Such failures may stem from IM frameworks that may be circumvented or are incomplete. Moreover, the information presented by the prover must be semantically meaningful. If the system configuration conveyed by the prover is too difficult to reason about, the verifier might not be able to determine if her requirements are satisfied. Integrity reporting is also important part of verification. The proving party must be able to generate timely, secure, and authentic *attestation* of system integrity. Since system integrity is constantly in flux, it is important to design a reporting mechanism that is timely and efficient. If the verifier cannot detect changes as they occur, then future integrity violations may be missed.

2.3.1 Early IM Approaches

Integrity measurement has its roots in research that explored assuring system integrity independent of verification at runtime. In particular, *secure boot* mechanisms focused on controlling the boot process. The Aegis system [12] used a special on-board guard coprocessor to monitor each binary loaded into memory and compare it to a set of expected binaries. If a mismatch occurred, then the system would switch to a special recovery mode that let the administrator recover the system. This is similar to recovery techniques used in current Google platforms like Android devices and Chrome OS laptops [48].

The first attempt at integrity verification began with the introduction of the IBM 4758 secure coprocessor line. These small form factor devices let administrators run security-sensitive code and manage cryptographic secrets in a hardware-protected location. Since the 4758 would be trusted to perform sensitive operations like negotiating SSL connections on behalf of remote clients, the designers wanted a method of proving the device identity and the integrity of code deployed on it. This was realized through OA [22] that used certificate chains to identify the installed code and data resident on the system. Clients could obtain these certificates from the device and inspect the authorities that signed for the integrity of the installed code. However, this provides only *load-time* verification and lacked an ability to verify the ongoing integrity of long-lived entities susceptible to corruption. Moreover, the OA approach cannot apply to architectures with less hardware protection between code layers and with multiple concurrent

applications.

Other early approaches to integrity verification also used specialized hardware to gain access to integrity-relevant information without the risk of compromise by the system it monitored. Co-pilot [23] used a device that interposed on the PCI bus to provide a locally connected administrator with hashes of kernel memory when unexpected values were detected. Several projects [49, 50] used a similar device to introspect on kernel memory for anomaly and intrusion detection. While these approaches are capable of detecting a wide range of events, they require verifiers with specific domain knowledge of the proving system to make sense of the memory hashes.

While specialized hardware provides the measurement framework protection for the system they are monitoring, it is often impractical to deploy such devices broadly and cheaply. Software-only techniques [24, 25] have also been explored to obviate the need for specialized hardware. These approaches use timing to detect unusually long execution times that would indicate code modification. This is similar to watchdog timers on embedded devices that detect corrupted firmware [51]. However, timing lacks applicability to more hardware with more complicated instruction sets and multicore processors where such measurements are difficult to predict.

2.3.2 Current IM Approaches

The TPM is only a passive device with basic hardware support for collecting early boot measurements. TPM equipped motherboards have a *core root of trust for measurement (CRTM)* that measures the BIOS, option ROMs and other parameters. Beyond that, an IM framework must support the collection of additional measurements. At the bootloader, the Trusted Grub [52] bootloader records all parameters, binaries, and additional files before loading them. The OSLO was the first bootloader to also support the late launch feature in AMD CPUs [28]. tboot [29] offered similar support for Intel processors' Trusted eXecution Technology (TXT) [26]. In addition, tboot uses the TPM's NVRAM to implement a limited secure boot feature that halts the system when invalid SRTM and DRTM measurements are detected.

Extending the IM framework into the Linux kernel was first introduced in IBM's Integrity Measurement Architecture (IMA) [30]. IMA is implemented as a Linux Security Module (LSM) that measures all code, kernel modules, and specified data files loaded on a system. IMA was later added to the mainstream kernel as a Linux Integrity Module (LIM) [53]. Additional LIMs can be created to collect arbitrary data based on a policy that specifies which combination of filesystem, DAC, and LSM metadata indicate an

integrity-sensitive file operation.

2.3.2.1 Application Support

At the application level, the TCG's Trusted Software Stack (TSS) [31] provides a standard for userspace programs to use the TPM to report program-specific integrity measurements. It also exposes other TPM cryptographic features like key management, encryption, and attestation. The open source daemon Trousers [31] is one example of a TSS implementation. Other applications have explored integrating IM features into webservers [54], remote code execution [55], and authentication servers [56]. Haldar et al., presented a modified Java Virtual Machine that could monitor the runtime integrity of Java programs through stack and heap introspection.

Another popular application of the TPM has been whole disk encryption. Services like Symantec's WDE [57] use the TPM as a cryptographic key store and multi-factor authenticator for accessing encrypted disks and files. Microsoft's Bit Locker Drive Encryption [58] encrypts the system's main disk partition and only releases the key if the SRTM measurements match the correct Windows boot binaries. In effect, these encryption services function like secure boot controls where the system cannot boot unless a predefined set of measurements are recorded at boot. However, runtime compromise of the system after decrypting the disk can still lead to leaked data.

2.3.2.2 Secure Communication

The TPM's physically protected identity secrets lend itself naturally to authentication. Combined with integrity verification, the TPM can facilitate integrity verified network communication. The TCG's Trusted Network Connect [42] specification describes how to integrate integrity verification into enterprise networks. The protocol uses the TPM's AIK and an enterprise PrivacyCA to identify the system. In addition, clients must provide valid attestations before joining and then to periodically re-attest to their integrity while on the network. Several researchers have also proposed [59] to integrate integrity verification into TLS protocol. Other work [60, 61, 32] has incorporated TPM attestations into public key certificates to bind integrity states to platform identities. Prototype implementations of RADIUS and EAP-TLS authentication services for networks have also been demonstrated [62]. Finally, the OpenTC PET [43] architecture uses a service resident on a virtual machine host to generate attestation on behalf of a VM and mediates a TLS connection to that VM if the client accepts the attestation.

A major limitation of these approaches is that they under specify what information

should be measured. In particular, they recommend that a verifier simply inspect the SRTM measurements for a correct boot process. Moreover, the reported information of these approaches is only valid as long as the attested system's configuration does not change. This requires the relying client to continually request new attestations at regular intervals to detect if the malicious changes have occurred. However, simply repeating attestations is impractical as the TPM is a resource limited device. Typical attestation times take over one second [37], which is further exacerbated by requests from multiple systems.

2.3.2.3 Minimizing the TCB

Not all processes and data on a system are integrity-relevant. Untrusted user processes can be confined from trusted ones using a strong MAC policy and not all data affects the integrity of code. For frameworks like IMA that capture a broad class of events indiscriminately, the attestations it produces may contain superfluous measurements that the verifier must parse. Even if every measurement is important, large trusted computing bases (TCBs) are more difficult to trust. This is due to the amount of code that must be verified.

Recent work has explored reducing the TCB that must be measured. PRIMA [41] leveraged the system's MAC policy to obtain the system's information flow graph to identify the trusted processes and interfaces that received low-integrity inputs. BIND [40] ran code in an isolated environment to reduce attestation size. In addition, it tied the results of a computation with the OS and memory protected execution environment to enable relying parties to verify an input's source before accepting it. Flicker [63] took this approach further by leveraging late launch, but suffered from performance overhead due to the TPM's speed. TrustVisor [64] reduced the overhead of Flicker by reimplementing parts of TPM in software and managed program execution in a VMM. Our previous work on designing verifiable VM host installer [65] generates a system with a static filesystem that simplifies verification by eliminating accumulation of TCB data.

2.3.2.4 Virtualization

One challenge of extending integrity verification to virtualized systems is the lack of a physical platform with which to bind attestations. The first application of IM to VM systems was the Terra [10] VMM, which recorded measurements of a VM's disk image and memory pages, but the granularity of measurement requires a semantic reconstruction of the program space to verify. The virtual TPM (vTPM) [32] by contrast provided

a virtualized TPM to VMs so they could report their own integrity information. This required careful management of virtual EKs and AIKs so that a verifiable chain to the VMM's TPM was always present.

Other projects have focused on monitoring guest VM integrity to detect specific integrity violations. Patagonix [66] and SIM [67] monitored changes to code pages against known good code. Virtual machine introspection (VMI) techniques like Livewire [68], xenaccess [69], and Vici [70] expose general purpose memory introspection interfaces to read VM memory. While VMI has been leveraged to build intrusion detection system (IDS) and IM systems, they did not specify a general attestation protocol for remote verifiers to use.

SecVisor [71] and HookSafe [72] provide integrity protection of sensitive kernel data structures. Our Virtual Machine Verifier [44] monitored VM integrity through a proxy in the VMM. Clients could verify the VMM and request a policy the VMM used to determine VM integrity. The VMV also offered integrity protection by modifying the VMs to use a secure execution kernel that denies execution of untrusted code and a network manager that only allows trusted network-facing daemons to connect to integrity verified clients.

2.3.2.5 Distributed Systems

In distributed systems, auditing has been used to detect software faults and provide accountability. Instead of examining the configuration of participating systems, approaches like PeerReview [73] collect logs of signed messages sent and received by participating nodes to detect application specific protocol violations. Accountable Virtual Machines [74] used VM replay approaches presented earlier in ReVirt [75] to verify the log by re-executing the inputs in a known good VM and comparing the outputs to those in the log. Trinc [76] used an additional secure hardware component to provide tamper-evidence to message logs. These approaches are limited by the scope of their monitoring, which is often application specific or captures only a certain class of inputs.

DStar [77] used Decentralized Information Flow Control [78] to ensure programs in a distributed system only received inputs from integrity labels they specify. However, these systems do not verify the integrity of the OS, which must protect and manage these data flows. The Trusted Platform on Demand [79] architecture provided basic integrity verification for provisioned virtual machines. The Shamon [80] shared reference monitor defined an approach for joining cooperating VMMs that share identical enforcement policies. However, the Shamon did not specify how to manage VMMs with heterogeneous

policies or monitor runtime integrity of these systems and their VMs.

Chapter 3

Network-based Root of Trust for Installation

System installation is a particularly important step because it defines the system's starting state. Administrators spend a great deal of time configuring systems to prevent illicit modification and ensure this process was done correctly is critical. In large data centers, installation is often performed over through network installation mechanisms, such as disk cloning. However, network-based installation is vulnerable to a variety of attacks, including compromised machines responding to installation requests with malware. In this chapter, we explore the design challenges of building an installation verification mechanism. We then propose the network-based Root of Trust for Installation, an installer that binds the state of a system to its installer and disk image.

3.1 Introduction

Data centers and large enterprises often use network installation and monitoring to manage the integrity of their deployed systems. This enables administrators to focus on hardening fewer systems that are cloned over the network to a multitude of machines. Once installed, remote monitoring tools and services can be used to automate the process of detecting anomalies in system behavior, intrusions, and illicit modifications to the filesystem. These mechanisms aim to provide *trusted distribution* where by an administrator can verify a system was installed as intended and that nothing has secretly modified the system since installation [81]. Without trusted distribution, it is difficult to ensure the ongoing correctness of a system at runtime.

Unfortunately, network-based installation introduces new challenges to the already difficult process of verifying system installation. The most common method of initializing a network installation is to load a bootstrap program over the network, thus eliminating the need for installation media like optical disks. However, the addition of network access opens the possibility for malicious parties to compromise installer code or corrupt the disk image in flight. Even after installation, malicious modifications can compromise security-critical files, which may be difficult to detect in specialized files like configuration scripts that lack a well known correct state. In the presence of such subversive code and hard to verify data, monitoring tools may be tricked into reporting that nothing wrong has happened. Ultimately, a method is need for proving to an administrator that a system has been securely installed and not been modified since that installation.

To achieve this goal, we propose the network-based Root of Trust for Installation (netROTI), an installation method that links a filesystem to its installer and the disk image used prior to configuration. If an administrator trusts their installer and disk image, then they can trust systems booted from filesystems derived from such an installation. Using the netROTI approach, administrators can configure their hardened images, and run the netROTI to install all of their machines automatically. They can then receive a proof from each machine, showing if it was booted from a compromised filesystem. An implementation of the netROTI for a Eucalyptus [82] cloud environment adds only an 8 second fixed overhead plus 3% of image download time to the network installation process, and verification can be automated for the administrator. The result is that secure network installation, even over an untrusted network, can be automated.

3.2 Network Boot Installation

In this section, we first describe the process of network installation. Next, we discuss the possible attacks on this procedure and security guarantees required for a trusted installation.

3.2.1 Current Network Installation

Companies and universities with large system deployments install and maintain their systems differently from the typical desktop user. While individual systems are typically installed using optical disk or a USB drive, the long installation process, need for physical media, and specific customizations for that environment make it impractical to use the same approach for hundreds of machines. Instead, network-based installation techniques

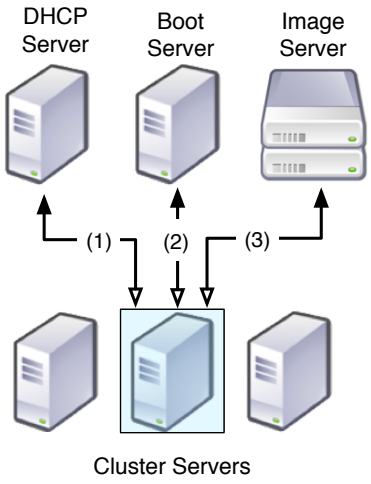

Figure 3.1. Network installation bootstrapped via PXE Boot. The client system in blue loads the PXE Boot firmware, which (1) initiates a DHCP request on the local subnet to setup basic networking and locate a Boot Server. After obtaining this address, (2) the client requests a NBP from the Boot Server and executes it. The bootloader may request additional files from the Boot Server such as modules and an initial ramdisk to setup the installer client environment. Finally, (3) the installer connects to the Image Server and begins transferring the disk image to the target system's hard disk. After configuring the image, the system reboots.

using customized automated installer images or disk cloning are used to rapidly upgrade out-of- date systems or restore compromised servers to their proper state.

From speaking with administrators in several large companies and our own university, which supplies computing resources for over 40,000 students, we found the most common disk cloning tools to be Symantec's Norton Ghost [83], Acronis True Image [84], or custom designed tools that use a variety of free and open source utilities. Other services automate installation tasks like Microsoft's Windows Deployment Services [85] and Rocks [86]. These tools function by loading a client at boot time over the network that connect to a management server download files like a pre-configured disk image or installer programs. This reduces deployment time and allows administrators to harden a single installation and replicate it among systems that perform similar tasks like VMMs in a cloud or employee workstations.

While there are several methods of bootstrapping an installer client, one of the most common methods is the Preboot Execution Environment (PXE) [87]. Figure 3.1 briefly illustrates a network install using the PXE protocol. First, the system to be imaged (we call the client) boots into the PXE firmware (usually loaded by the BIOS from the NIC's firmware). Next, the client starts the protocol by (1) broadcasting a DHCPDISCOVER request on port 67 with additional PXEClient extension tag. A DHCP or ProxyDHCP

server responds with a DHCPOFFER on port 68 providing an IP address and a list of Boot Servers. The client then (2) sends a DHCPREQUEST to a Boot Server and gets a DHCPACK message with the file name of a Network Boot Program (NBP) it retrieves from the Boot Server via TFTP. The client then executes the NBP, which may request additional files such as a kernel or modules required for the client's hardware.

The installer client is setup by the NBP either using files download from the Boot Server or by retrieving them using protocols like NFS or HTTP. Finally, the client (3) contacts the Image Server and requests a disk image. After the image has been written to the hard drive, the client performs additional configuration stpdf like setting the hostname and networking. Finally, the machine reboots into the newly imaged OS.

3.2.2 Attacks on Network Installation

Ensuring the correct operation of systems within large installations like a data center, requires the administrators to be able to prove their systems have been installed and configured with high integrity code and data. While the techniques mentioned above automate installation, they do not enable administrators to verify whether the system has booted from a properly installed filesystem. Potential attacks on the installation procedure or modification of systems later could corrupt a server and lead to a host of attacks from within the data center. We now consider some of these attacks and then discuss the guarantees that must be satisfied to ensure the a server has booted from a high integrity installation.

The first place a server can be corrupted is during installation. In the process described in Figure 3.1, the client system could potentially load a malicious PXE firmware from the NIC installed during a previously compromised state. Other attacks have been demonstrated [88] that allow remote attackers to compromise NIC firmware over the network. In either case, such firmware could lead to direct attacks on the system's memory. Another vector for attack exists when the PXE client searches for the Boot Server. Since the PXE client relies on information from local DHCP or Proxy DHCP servers, a compromised server acting as a rogue DHCP server on the local subnet could trick the client into downloading a malicious NBP and install a rootkit. At the network level, an attacker could modify data sent on the wire to the client if unencrypted or perform a man-in-the-middle attack to tamper with the installation. Even after installation, a system may be vulnerable. Numerous attacks exist that place rootkits or make malicious changes to the filesystem that persist even after a system reboots.

3.2.3 Securing Network Boot Installation

To secure network installation, it is necessary to show the installed system is derived from the expected origins, installer, and disk image. While not everyone may trust the installer or disk image, those that do would be willing to work with such a system if it could be verified. In this case, we envision that large data center administrators would be able to leverage such trust because they specify the installer and disk images that can be loaded.

Verifying installation is not very useful by itself, however, as the machine will be immediately rebooted after installation and may be rebooted multiple times before any subsequent re-installation. Thus, any network installation must enable verification that a system was booted from an expected installation. Thus, our method enables an administrator to verify that the filesystem at boot time is linked to the installation origins, the installer and disk image. We note that this does not prevent the system from coming under runtime attacks, such as buffer overflows, but these attacks will be detected if they modify the filesystem on the next reboot.

Any secure network installation must be practical. A key question is whether the installed filesystem is sufficiently stable to enable such a verification. In an initial experiment [65], we found that only three files of privileged VM system configuration were modified dynamically during its execution. Also, manual updates to systems are prohibited in our approach, as they are ad hoc. For administrators, a clean, automated install is preferable to manual modifications anyway. An administrator may push specific updates to all their systems automatically, but these cannot be linked to the installer. We envision that software on the installed system can extend the install-time proofs, if authorized.

3.3 The netROTI Method

We now introduce netROTI, a network-based installation method that links the resulting system verifiably to a particular source. We first define our trust and threat models to establish the scope of our solution and then detail how the netROTI augments the installation process. Finally, we describe the protocol used to verify the filesystem's origin.

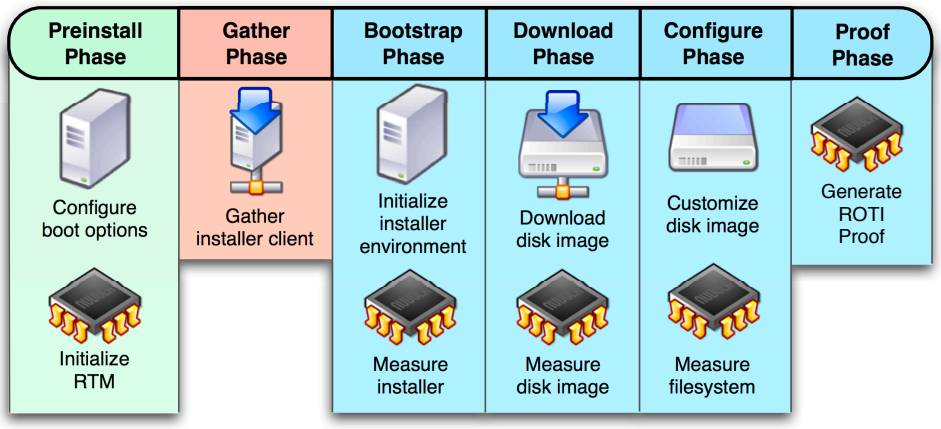

Figure 3.2. Timeline of the installation process. The administrator configures the client in the preinstall phase. The client then downloads the installer and bootstraps a secure environment, which measures the installer. Next, the client downloads, measures, and configures a disk image to place on the local disk. Finally, the resulting filesystem is measured and a proof of the system's Root of Trust for Installation (ROTI proof) is generated.

3.3.1 Trust and Threats for Designing a netROTI

For our design, we assume a trust model where the physical hardware is safe from attack and is implemented correctly. We also trust there exists an administrator or software provider with the authority to deem code and data (e.g., the installer and disk image) as trustworthy. While we make no assumptions that such trust is placed correctly, our goal is to prove that a system is linked to a particular origin certified by one or more authorities. Thus, a verifier can determine their trust in a system based on its trust in the ability of authorities to certify their systems. We also trust the data center administrators and hence do not address insider attacks. Finally, we do not consider attacks on the cryptographic algorithms used nor attacks on the PKI or authentication procedures like direct anonymous attestation [47] to establish identities.

For our threat model, we consider an attacker that can modify or inject data on the network, is able to impersonate various services, and compromise other hosts on the network. These attacks could lead to the client loading a malicious installer, installing a vulnerable or malicious disk image, or compromise of device firmware. The attacker can change the contents of the client's disk after installation and perform attacks on the running system. Reporting attacks on the system's runtime state is outside the scope of this work, but the netROTI does provide a root of trust for detecting these security violations by giving a proof of the systems initial integrity at boot time.

3.3.2 netROTI Overview

The netROTI approach is a network-based system installation method that cryptographically links the installed filesystem with the installer and source used in the installation. Figure 3.2 illustrates each phase of the installation procedure. The *preinstall phase* highlighted in green is a trusted, manual step requiring the administrator to configure the client to boot from the network and to prepare the client's Root of Trust for Measurement (RTM) (e.g., TPM) used to record and report critical code and data. Since the tasks in this phase are performed manually, we trust them axiomatically. Next, the client gathers the necessary files to install from the network in the *gather phase*, which is shown in red because it need not be run by trusted code and is unmeasured. Once collected, the system enters the subsequent blue phases, which contribute to building a *Root of Trust for Installation (ROTI) proof* linking the installer and image to the client's filesystem. The *bootstrap phase* initializes a secure execution environment for the installer after the RTM measures it. The installer downloads and measures the image to be installed in the *download phase*. Next, it configures and measures the resulting filesystem in the *configure phase*. In the final *proof phase*, the RTM generates a ROTI proof later used by the system at runtime to produce attestations, which a verifier can use to identify the installer and disk image used to configure the filesystem from which that client booted.

3.3.3 netROTI Installation Phases Detailed

Each of the netROTI installation phases has a specific goal and tasks to achieve that goal.

3.3.3.1 Preinstall Phase

The preinstall phase is a manual process carried out by the system administrator to prepare the client system for installation. This phase is needed to configure components that enable generation of ROTI proofs. To prepare the system, the administrator configures the BIOS to boot from the network and installs the RTM with the keys necessary for it to identify this client uniquely.

3.3.3.2 Gather Phase

The goal of this phase is to retrieve the installation image and installer. We need not measure this phase as we will start from a known state using only these inputs starting the next phase. First, the client machine loads the network boot firmware to obtain

network access and locate the Boot Server. It then retrieves an NBP that downloads the additional installer files, the installer kernel, a ramdisk containing the installer code, and a bootstrap program that sets up a secure environment for the installer.

3.3.3.3 Bootstrap Phase

Since the previous install phase performed unmeasured operations, there is a possibility that malicious code may have been loaded into memory. Therefore, we must establish a clean starting point for measuring subsequent operations in the installation process. The bootstrap phase achieves this through a late launch that takes a piece of code, records it in the RTM, and effectively reboots the system before executing the code in a region of protected memory. This memory protection prevents attacks from potentially malicious resident code loaded before the installer and from external devices that have direct access to memory. Once the installer kernel is launched, it measures the installer's ramdisk, unpacks it into memory, and begins the next phase. We color this phase blue to indicate the installer code and data are measured before being executed.

3.3.3.4 Download Phase

After the installer has been initialized, it enters the download phase. The goal is to retrieve and measure the disk image before installing it. First, the local system's basic networking and partition table are prepared to enable a disk image to be retrieved and written to a clean disk. The disk image is also measured into the RTM so that a verifier can later identify the trustworthiness of the downloaded disk image. This helps detect attacks on the disk image while in transit and from compromised or rogue image servers.

3.3.3.5 Configure Phase

In the configure phase, the downloaded disk image is specialized to the target system. This includes setting up networking, filesystem tables, devices, security policies, SSH host keys, etc. The installer also generates signing keys used by the RTM for generating attestations and the ROTI proof. We describe this in more detail in the next section. Next, the system's startup scripts are modified to measure the root filesystem at boot time. This filesystem manifest is included in attestations so the verifier can inspect how the filesystem has been modified.

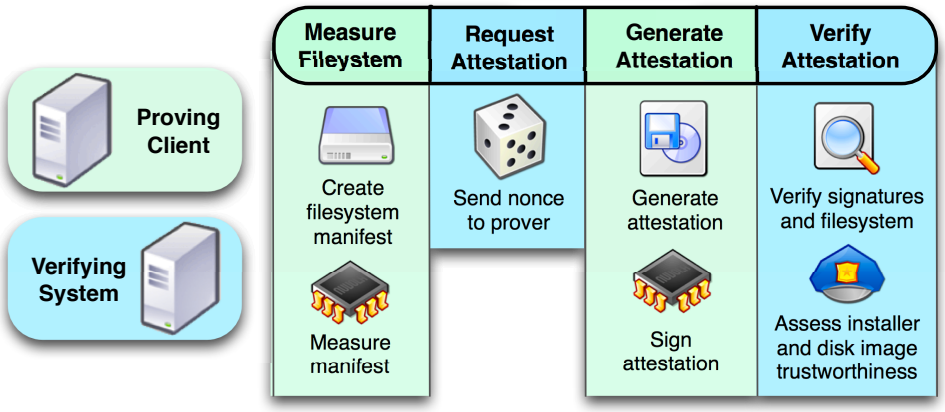

Figure 3.3. Timeline of the validation process. The filesystem on the proving client is measured to generate an attestation of the installed and current filesystems, which can be checked by the verifying system.

3.3.3.6 Proof Phase

In the final, proof phase, the installer generates a *ROTI proof* that ties the final installed filesystem to the installer and disk image for verification at runtime that the client is derived from such inputs. The ROTI proof is a signed tuple $R = \text{Sign}(F, D, I)_{K^-}$, where K^- is a private key that identifies the physical machine and is endorsed by its RTM. This tuple acts as a proof showing that a machine possessing K^- was specifically configured by I using disk image D to produce filesystem F. A verifier can inspect the ROTI proof to determine if it believes the combination of I and D results in a trustworthy installation (i.e., F). Upon completion of the ROTI proof, the system reboots into its newly installed filesystem. In the next subsection, we describe how the ROTI proof is used by a verifier to check the integrity of the client at runtime.

3.3.4 Verification

Once the installation procedure is complete, the system is ready for verification. A successful validation proves to a verifier the particular installer and disk image that was used to install the current filesystem. The verifier can then make a trust decision whether the combination of installer and disk image came from a source they trust.

Figure 3.3 illustrates the validation procedure. During boot, the initial ramdisk (initrd) is loaded and measures the root filesystem to generate a manifest of hashes for each file. Next, the system starts a network facing *attestation daemon* to handle attestation requests. When a remote verifier wishes to inspect the proving system, it sends a nonce to the attestation daemon that builds an attestation and returns it to

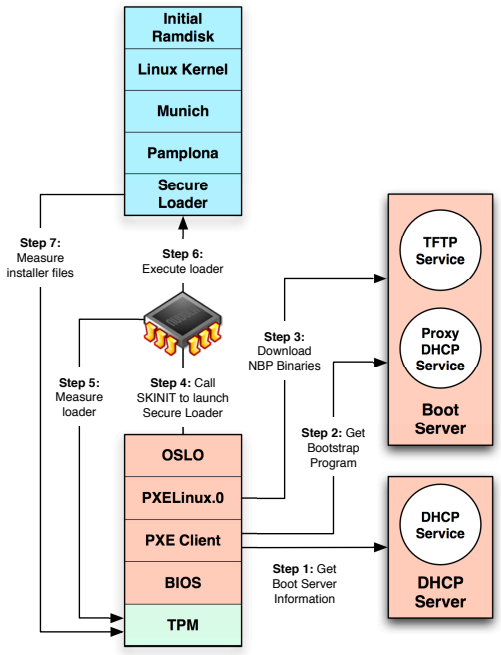

Figure 3.4. After the client follows the PXE protocol and contacts the Boot Server, it downloads the `OSLO` bootloader's three binaries, the installer kernel, and initial ramdisk. `oslo` performs the AMD late launch operation, SKINIT, which causes the CPU to clear the DRTM registers, extend PCR 17 with a hash of the Secure Loader Block (SLB), and begins executing the SLB, which measures the rest of the downloaded files and extends it to PCR 19. `pamplona` undoes the DRTM memory protection and `munich` launches the kernel to being installation.

the verifier. The attestation[1] includes in its measurement list, R the ROTI proof and F' the current filesystem manifest. The verifier checks that the signatures on both the attestation and R are from keys endorsed by the RTM of the proving system. Verifying the identities of keys is outside the scope of this work, but we assume a PKI is maintained by the administrator deploying the systems. Once it has established the identities of the signing keys, the verifier assesses whether it trusts the installer and disk image in R to produce a trustworthy installation. If it does, it then compares F' and F in R to see how they differ. If no security critical files have changed, the verifier can assume the current system booted into a filesystem that was produced by an installer and disk image the verifier trusts.

3.4 Implementing the netROTI

We now describe our proof of concept netROTI implementation.

[1]See Chapter 2 for an explanation of the attestation mechanism and format.

3.4.1 Installation

We created our netROTI as a series of scripts that automates the installation process packed into an 11 MB ext2 ramdisk downloaded along with a modified Linux 2.6.18 kernel and the OSLO [28] bootloader. OSLO is a specialized bootloader that implements the DRTM functionality in AMD processors that we use to launch the installer kernel in a secure environment. Before installation begins, the administrator configures the BIOS to boot from the PXE firmware. The TPM must also be cleared of any previous administrative passwords and keys, so the installer can create its own. This corresponds to the preinstall phase in our design. In the gather phase, the begins the PXE protocol to obtain the location of the Boot Server. The client then downloads the `pxelinux.0` NBP via TFTP, which automatically retrieves the OSLO bootloader, Linux kernel, and installer ramdisk.

The system then enters the bootstrap phase illustrated by Figure 3.4. First, the NBP constructs a multiboot header indicating the address where the the installer files are located in memory and executes the OSLO bootloader. OSLO consists of three ELF binaries that perform separate stages of the DRTM process. The first binary prepares the system for the DRTM process by shutting down all but the primary CPU core and loads the second stage binary into the secure loader block (SLB). The AMD late launch instruction, SKINIT, is invoked with the entry point address of the SLB as its only argument. The CPU then sets the DRTM PCRs in the TPM to indicate the operation was invoked, sets the device exclusion vector (DEV) to enable memory protection for the SLB, and sends a measurement of the SLB to the TPM. Finally, the CPUs jumps to the entry point in the SLB that measures the installer Linux kernel, ramdisk, and boot parameters in the multiboot header, restarts the other CPU cores, and disables the DEV protection. Finally, it launches the third stage binary that imitates a normal GRUB bootloader and launches the Linux kernel.

Once the installer has been bootstrapped, the ramdisk is unpacked into memory and the installation script is executed. This sets up basic support for devices like console and networking and starts the download phase by running a `partimage` client. This contacts a preconfigured `partimaged` server, verifies its SSL certificate against the CA certificate in the ramdisk, establishes an SSL connection, and downloads the disk image. The image is measured and then written to the hard disk.

In the configure phase, the installer scripts configures machine specific files including updating the UDEV rules for new hardware, networking configurations, firewall rules, fstab entries, creating a swap partition, regenerating SSH host keys, and so on. The

TPM is then setup with new administrative credentials and a fresh AIK is generated. The TPM also endorses the AIK by creating a certificate that signs the AIK's public key with the TPM's EK. The installer also installs a simple network-facing python service we wrote that acts as the attestation daemon, which services requests for attestations. The initial ramdisk on disk is then modified to generate a manifest of the filesystem every time it boots. This manifest contains a hash of every file and is used to create attestations.

In the proof phase, the installer measures the files on disk that have changed or been added since the disk image was written. A list of SHA1 hashes for each file is stored on the disk. The final step generates the ROTI proof by producing a TPM quote with PCRs containing every measurement taken during the installation process. This quote is signed with the newly created AIK from the configure phase. We use a hash of the system's hostname as the nonce since we are not concerned with the freshness of the quote, as we only care that the ROTI proof correctly identifies the installer and disk image for this system. We tar and gzip the quote with the file manifest and list of measurements taken during installation to create the final ROTI proof file.

3.4.2 Verification

We now describe the verification protocol between a system installed by the netROTI (the proving system) and a remote verifier. Before the protocol begins, the proving system boots into its initial ramdisk. It then executes the measurement script inserted during installation. This script generates a manifest of the entire filesystem with corresponding hashes for each file. The system resumes the boot process and starts the attestation daemon. When a verifier sends a nonce to the daemon, the daemon takes a SHA1 hash of the nonce and requests a TPM quote signed by the TPM's AIK. The quote's PCRs contains a hash of the filesystem manifest taken at boot time and the nonce. The quote is then returned to the verifier with the ROTI proof file (corresponding to R in the attestation from Section 3.3.4), the filesystem manifests taken at boot (F') and during installation (F), and the AIK's certificate (the verifier has the nonce already).

Upon receipt of the attestation, the verifier first validates the signatures of the quote and ROTI proof. It then checks that the AIK's certificate is signed by the expected TPM's EK. Next, the verifier assesses the trustworthiness of the installer and disk image by extracting them from the ROTI proof and matching them against a list of acceptable measurements. If these are found to be trustworthy, the filesystem manifests are compared to see if any files have changed since installation. If no security critical files

Type	Operation	Time (seconds)
Install	Download and Write Disk Image	64.000
Install	Configuration	18.644
	Sub-total	82.644
netROTI	netROTI Configuration	6.740
netROTI	Measure Image	1.900
netROTI	Generate TPM Quote	0.890
netROTI	Measure Modified Files	0.390
	Sub-total	9.920
Optional	TPM Setup	45.400
Optional	Generate AIK	11.220
	Sub-total	56.620
	Total Install	**149.184**

Table 3.1. Breakdown of the installation time averaged over ten installations of a Eucalyptus cloud node.

are modified, then verifier accepts the proving system as having booted into a filesystem installed by a trusted installer and disk image.

3.5 Evaluation

We constructed a proof of concept netROTI installer to assess the overall impact it has on network installation and how it address attacks on the installation process.

3.5.1 Performance

To evaluate the overhead our netROTI installer imposes on the installation procedure, we performed ten installations of a Eucalyptus cloud node's disk image across ten systems. We created an image to be installed by manually configuring an Ubuntu server cloud in our Eucalyptus on a Dell PowerEdge M605 blade with 8-core 2.3GHz Opteron CPUs and 16GB of RAM on a quiescent gigabit network. We then created a 387 MB gzipped disk image of the 1.3 GB filesystem. Table 3.1 shows the times for each operation performed during installation. Normal installation took 82.644 seconds or 55.34% of the overall time. The disk image related operations (e.g., downloading, writing, and measuring the image) are a function of the disk image's size, which can be improved through more efficient compression algorithms. In particular, we found our hardware could perform SHA1 hashes at 132 MB/s, which resulted in a 1.9 second disk image measurement time.

TPM related operations are inherently slow due to the speed of the TPM's bus (33 MHz) and its low power design. While netROTI specific operations added additional time

to the install, two operations, namely generating a new AIK and TPM setup, account for 37.95% of that overhead. We note that the function of these steps are to create keys that could be reused across multiple installations as long as the encrypted public portions of the AIK and SRK (created in the TPM setup operation) are retained during reinstallation. Thus, an administrator could copy those encrypted files and redistribute them in the installer or have the installer copy them from the local disk before overwriting it. Ultimately, we find the overhead due to the netROTI to be a fixed cost of about 8 seconds plus about a 3% overhead for measuring the image when the optional TPM setup and AIK creation steps are reused from previous installations.

3.5.2 Security Evaluation

Table 3.2 lists a comparison of several security mechanisms and their ability to handle a range of attacks on the network installation process. In addition to our netROTI design, we consider the OSLO bootloader alone, the filesystem auditing tool Tripwire [89], and the Windows Bitlocker file encryption scheme [58]. OSLO uses the DRTM process to both measure malicious installer code and defeat rootkits the system might boot into before installation, but it is unable to address attacks beyond that. Tripwire is an auditing tool that creates a digitally signed log of the installed filesystem that administrators can query to detect changes. This prevents attacks that change the disk contents after installation, but cannot guarantee anything about the filesystem during installation. Bitlocker encrypts the filesystem and optionally uses the TPM to verify that the early boot phase has not been modified before decrypting the disk. While this prevents offline attacks, modifications to the disk after decryption are not prevented. The netROTI uses OSLO for protection against rootkits and to record malicious installers. It also measures disk images before installation and uses the ROTI proof combined with boot-time filesystem measurements to detect changes. However, none of these approaches directly address attacks on the installed system at runtime.

The key advantage of the netROTI over these other approaches is its ability to provide an attestation of not only the filesystem, but the installation environment that produced it. While the other solutions in our comparison prevent attacks at various stages of the installation process, none of them can speak for the trustworthiness of the installer that produced them. By using secure hardware to measure before using each critical component during installation, a verifiable proof can be created of the filesystem's origin.

Attack Type	OSLO	Tripwire	Bitlocker	netROTI
Rootkits before install	Yes	No	No	Yes
Malicious installer code	Yes	No	No	Yes
Malicious disk image	No	No	No	Yes
Modified data after install	No	Yes	Partial	Yes
Runtime attacks	No	No	No	No

Table 3.2. A comparison of several mechanisms' ability to detect or prevent several classes of attack on an installation.

3.6 Conclusion

In this chapter, we introduced the netROTI, a method for performing network installation so that the resulting filesystem can be traced back to the exact installer and disk image that produced it. The netROTI leverages the protection of trusted computing features in modern CPUs to bootstrap a dynamic root of trust in an installer downloaded over the network and record all steps of the installation procedure to produce a ROTI proof. Using this proof, a verifier can inspect whether the guarantees of trusted distribution have been achieved. Our evaluation demonstrated the netROTI protects against a variety of attacks on the installation process and introduces only minor overhead when optimizations are taken into account.

Chapter 4

Building Integrity Verification Proxies for Virtual Machine Systems

Users are increasingly turning to online services, but are concerned for the safety of their personal data and critical business tasks. While secure communication protocols like TLS authenticate and protect connections to these services, they cannot guarantee the correctness of the endpoint system. Users would like assurance that all the remote data they receive is from systems that satisfy the users' integrity requirements. Hardware-based IM protocols have long promised such guarantees, but have failed to deliver them in practice. Their reliance on non-performant devices to generate timely attestations and ad hoc measurement frameworks limits the efficiency and completeness of remote integrity verification. In this chapter, we introduce the *integrity verification proxy* (IVP), a service that enforces integrity requirements over connections to remote systems. The IVP monitors changes to the unmodified system and immediately terminates connections to clients whose specific integrity requirements are not satisfied while *eliminating the attestation reporting bottleneck* imposed by current IM protocols. We implemented a proof-of-concept IVP that detects several classes of integrity violations on a Linux KVM system, while imposing less than 1.5% overhead on two application benchmarks and no more than 8% on I/O-bound micro-benchmarks.

4.1 Introduction

Traditionally in-house computing and storage tasks are becoming increasingly integrated with or replaced by online services. The proliferation of inexpensive cloud computing

platforms has lowered the barrier for access to cheap scalable resources, but at the cost of increased risk. Instead of just defending locally administered systems, customers must now rely on services that may be unable or unwilling to adequately secure themselves. Recent attacks on cloud platforms [90] and multinational corporations [1] have eroded the public's willingness to blindly trust these companies' ability to protect their clients' interests. As a result, the need for effective and timely verification of these services is greater than ever.

Recent advances in trusted computing hardware [20, 26, 27] and IM protocols [21] aim to achieve this goal, but current approaches are insufficient for several reasons. First, existing protocols depend on *remote attestation* to convey information about a proving system's configuration to a relying party for verification. However, an attested configuration is only valid at the time the attention was generated, and any changes to that configuration may invalidate it. Since the proving system's components may undergo changes at anytime, a relying party must continually request fresh information to detect a potential violation of system integrity. This problem is made worse by the significant delay introduced by many IM protocols' reliance on the TPM [20] to generate attestations. Since the TPM was designed for cost and not speed, it is only capable of producing roughly one attestation per second [37, 63, 91]. This renders TPM-based protocols far too inefficient for interactive applications and high demand scenarios.

Another limitation of current IM approaches is how *integrity-relevant events* are monitored on the proving system. Systems undergo numerous changes to their configurations due to events ranging from new code execution to dynamic inputs from devices. While various measurement frameworks have been developed to enable these components to report arbitrary events and its associated content (e.g., memory pages and network packets), conveying everything is impractical due to the sheer volume of data and effort placed on relying parties to reason about it. Moreover, not every event may have a meaningful effect on the system and communicating such events is a further waste. Thus, proving systems often make implicit assumptions to remove the need to collect particular measurements (e.g., programs can safely handle all network input), which may not be consistent with the trust assumptions of the relying party. This problem stems from the onus placed on the proving system's administrator to choose and configure how the various IM components will collect information without knowledge of relying party's requirements.

To improve the utility of existing IM mechanisms, we propose shifting verification from the relying party to a *verification proxy* at the proving system. Doing so eliminates

the bottleneck caused by remote attestation (and thus the TPM) from the critical path, by using traditional attestation protocols to verify the proxy and the proxy to verify the proving system's runtime integrity is maintained. Monitoring the system locally also permits the proxy to examine information relevant to the relying party's integrity requirements. Moreover, this approach supports the integration of fine-grain monitoring techniques like VMI into remote system verification that would otherwise be difficult to convey over traditional attestation protocols [10, 68, 66] or require modification to the monitored system.

In this chapter, we present the *integrity verification proxy* (IVP), an integrity monitor framework that verifies system integrity at the proving system on behalf of the relying party clients. The IVP is a service resident in a VM host that monitors the integrity of its hosted VMs for the duration of their execution through a combination of loadtime and VMI mechanisms. Client connections to the monitored VM are proxied through IVP and are maintained so long as the VM satisfies the client's supplied integrity criteria. The IVP framework is able to verify a variety of requirements through an extensible set of measurement modules that translate a client's requirements into VM-specific properties that are then tracked at runtime. When an event on the VM violates a connected client's criteria, immediate action is taken to protect that client by terminating the connection.

However, we faced several challenges in designing an IVP that can be trusted to verify the target system. First, the proxy itself must be simple to verify and able to maintain its integrity without the need for frequent attestation. We employed previous efforts in deploying static, verifiable VM hosts [92] to achieve this. Second, introspecting directly on the running VM can introduce significant performance overhead if done naively. Instead, we monitor the integrity of the VM's enforcement mechanisms by leveraging practical integrity models [38, 35, 36] to identify specific enforcement points that are critical for protecting the system's integrity. By monitoring these enforcement points, we reduce the frequency and impact of verification. Finally, managing multiple channels from the same and different clients introduces redundant criteria verification. We eliminate this redundancy by aggregating multiple connections for a single criteria.

We implement a proof-of-concept IVP for an Ubuntu VM running on a Linux Kernel-based Virtual Machine (KVM) host. We constructed both loadtime and custom CPU register-based VMI modules for monitoring VM enforcement mechanisms. We validated our proxy's ability to detect violations correctly by building and attacking a VM designed to satisfy integrity criteria based on a practical integrity models and several kernel integrity requirements. We further evaluated the performance impact the IVP imposed on

monitored VMs, finding that it introduced less than 1.5% overhead on two application-level benchmarks.

The rest of this chapter is organized as follows. Section 4.2 provides background on current IM approaches and elaborates on the limitation of current IM protocols. Section 4.3 enumerates our design goals, presents the IVP architecture broadly, and highlights the main design challenges. Section 4.4 describes of our implementation, which is followed by evaluation of functionality and performance in Section 4.5. Finally, we provide related work in Section 4.6 before concluding in Section 4.7.

4.2 Remote Integrity Verification

In this section, we present background on remote integrity verification and its building blocks: measurement and attestation. We then discuss the challenges current approaches face and show why they are insufficient for monitoring dynamic systems.

4.2.1 Integrity Verification Overview

Figure 4.1 provides a conceptual view of the remote integrity verification, where a *relying party* wants to determine whether a *proving system*'s current configuration (e.g., running code and data) satisfies the verifier's integrity criteria for a trustworthy system. The proving system measures its early boot layers, which measure the operating system code and data. This, in turn, may measure user code, data, and operations (e.g., VMs and processes). Each individual layer aims to measure the *integrity-relevant events* occurring at the layer above. The relying party *monitors* these events by requesting attestations of the measured events to evaluate satisfaction against the integrity criteria. If the proving system fails to satisfy the criteria, the monitor protects the relying party by denying access to the untrustworthy system. Thus, the monitor enforces an integrity policy (the criteria) over the communication to proving systems. Its role is similar to that of a reference monitor [93] that enforces access control policies over resources.

In order to assess system integrity accurately, the integrity monitor must observe events relevant to its integrity criteria. For example, criteria demanding enforcement of an information flow lattice might require that only trustworthy code are loaded into privileged processes and critical system files may only be written to by such processes. Thus, the monitor would require the combination of measurement mechanisms on the proving system (its *IM framework*) to record these events. We now provide a brief overview of existing measurement and attestation techniques to illustrate how an integrity monitor

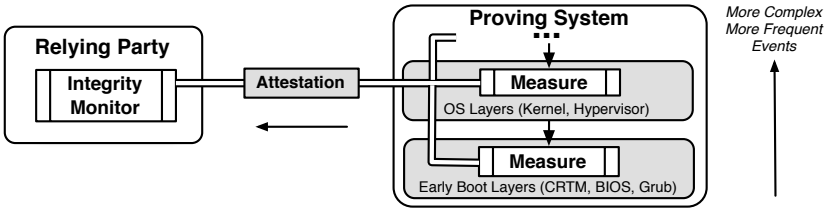

Figure 4.1. A relying party's *integrity monitor* inspects a remote system's integrity by requesting attestations of *integrity-relevant events* collected by the proving system's layers of integrity measurement.

would use them, but provide a broader review in Chapter 2 and more detail relevant to this chapter in Section 4.6.

A relying party's ability to judge system integrity is limited by which events are recorded and their detail. A framework with greater coverage of system events will be more capable of measuring the required integrity criteria for more complex configurations at higher layers. We divide these measurement techniques into two categories: (1) *loadtime* and (2) *runtime*. Loadtime measurements involve capturing changes to the system like code loading and data input *before* they occur. For example, IMA measures binaries before they are mapped into a running process [30] and Terra hashes VM disk blocks before they are paged into memory [10]. Others like Flicker [63] and TrustVisor [64] leverage hardware isolation to reduce the TCB down to a single running process. To measure other events, such as the data read and written by processes, some IM approaches measure other loadtime events. For example, PRIMA [41] measures the mandatory access control policy governing processes at loadtime.

Loadtime only frameworks assume that system integrity is maintained if all loadtime measurements are trustworthy. However, unexpected runtime events like code injection attacks or difficult to assess inputs like arbitrary network packets can subvert system integrity. To address this, runtime measurement techniques have been designed to record this class of events. Furthermore, mechanisms like Trousers [94] for userspace processes and the vTPM [32] for VMs enable these entities to report integrity-relevant events to an external IM framework.

However, mechanisms that report on a component's integrity from within run the risk of being subverted if the processes or VM is compromised. As an alternative, external approaches like VMI enables a hypervisor to observe runtime events isolated from the watched VM [95, 96, 69]. Recent VMI techniques [66, 67, 97] use hardware memory protection and trampoline code to trap execution back to the host for further inspection.

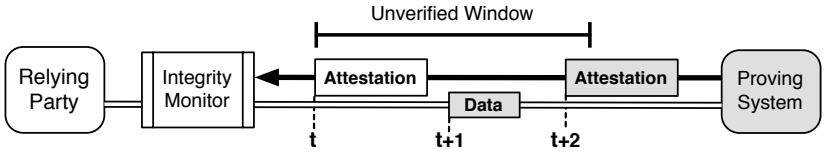

Figure 4.2. A window between each attestation exists where the integrity of the proving system is unknown.

While runtime measurement can detect changes at a finer granularity than loadtime measurements, they also introduce greater complexity. In particular, external approaches introduce a semantic gap that require domain knowledge like memory layouts to detect malicious modifications [98].

4.2.2 Integrity Monitoring Challenges

For the integrity monitor to verify system integrity accurately, its view of the proving system must be both fresh and complete. Stale or incomplete measurements limit the utility of the verification process. However, we find that current attestation-based verification models are insufficient for several reasons.

4.2.2.0.1 Stale measurements Attestation-based protocols introduce a window of uncertainty, which we illustrate in Figure 4.2. Here, the integrity monitor residing on the relying party requests an attestation at time t and finds it satisfies its integrity criteria. Since the prover is verified, the monitor permits it to send data to the relying party at $t + 1$. Later, the monitor requests a second attestation at $t + 2$ and finds the prover no longer satisfies the criteria. Because this violation could have happened at anytime between t and $t+2$, it is not clear without additional information if the data at $t+1$ was generated when the system was unacceptable. Classic attestation protocols like IMA [30] avoid this issue by buffering inputs until a later attestation is received, but this is not an option for high throughput or interactive applications.

4.2.2.0.2 Hardware bottleneck Many systems are dynamic and undergo numerous changes at any time. Thus, the monitor must continually poll for new attestations to detect changes. This problem is exacerbated by the TPM's design as a low performance device for attesting infrequent loadtime measurements like the boot process. In fact, current TPM implementations take approximately one second to generate a quote leading to major bottlenecks in any high demand scenario [37]. Designs that batch re-

mote attestations to eliminate queuing delays have been proposed [91], but still incur a significant overhead.

4.2.2.0.3 Criteria insensitive measurements A relying party's ability to assess system integrity is also limited by what events are measured. Since the proving system's administrator decides what the measurement framework will record, a remote verifier must often settle for the information provided by proving system. If that system provides only hashes of code loading operations, then a criteria requiring knowledge the possible runtime operations of those processes cannot be satisfied. However, it is difficult to know what information arbitrary clients require, which is especially challenging for public-facing services used across multiple administrative domains. On the other hand, designing an IM framework to record excessive measurements may be wasteful if they are inconsequential to the verifier's integrity criteria. Moreover, complex events occurring within an entity like may be difficult to assess. For example changes to kernel memory may indicate a rootkit, but it is hard to make that judgment without knowledge of where certain data structures are located. However, providing this context (i.e., entire memory layouts) via attestation can be impractical.

4.3 Integrity Verification Proxy

We now present the design of the IVP, an integrity monitor framework that verifies system integrity at the proving system on behalf of the relying party. By shifting a portion of the integrity monitor to the proving system, the IVP eliminates the need for continuous remote attestation and provides direct access to the system's IM framework to support a broad range of integrity criteria. We begin by describing our design goals and trust assumptions. We then give an overview of the IVP's architecture and detail how it achieves these goals.

4.3.1 Design Goals

Our aim is to extend the traditional notion of an integrity monitor into the proving system to overcome the limitations of current attestation-based verification protocols. Figure 4.3 shows the conceptual model of this approach. This model supports the following design goals.

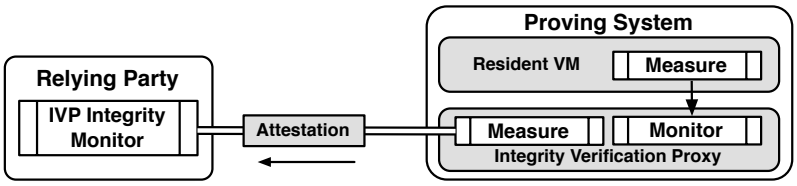

Figure 4.3. The IVP acts as an integrity monitor on the proving system that monitors the resident VM to enforce the relying party's criteria over the communication channel. The long-term integrity of the IVP and its host (i.e., layers below the resident VM) is verified by traditional loadtime attestation.

4.3.1.0.4 Enforce integrity criteria at the proving system. Monitoring system integrity remotely is insufficient because stale knowledge of the remote system's more complex events undermines the monitor's ability to correctly enforce its criteria. Instead, a relying party can establish trust in an integrity monitor on the proving system that enforces its integrity criteria. The IVP has direct access to resident VM's IM framework to eliminate the window of uncertainty caused by attestation protocols. Moreover, the IVP can terminate connections immediately when an integrity violation is detected to protect the relying party. If the relying party is also the administrator of the VM, the IVP can take further remedial measures such as rebooting the VM. However, the relying party must still monitor the IVP itself to justify such trust. Thus, the IVP must be deployed at a software layer whose integrity can be verified by the relying party without the need for continual attestation, or the purpose of moving monitoring to the proving system is defeated.

4.3.1.0.5 Criteria-relevant measurement. Traditional IM frameworks measure events irrespective of what the relying party requires. Moreover, entities on the resident VM may be implicitly trusted by the administrator and thus are not monitored. An effective IVP must support various integrity criteria that may even differ from administrator's criteria. To do this, the IVP leverages the available information about the resident VM to capture a broad set of integrity-relevant events to support differing criteria. In Figure 4.3, the IVP extracts information from both the IM framework on the proving system and additional information through external measurement techniques like VM introspection.

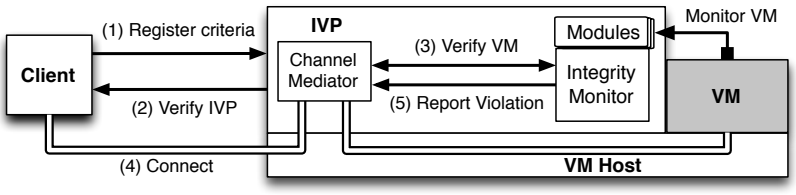

Figure 4.4. Integrity verification proxy architecture.

4.3.2 Assumptions

We make the following trust assumptions in the IVP design. First, we do not consider physical attacks on hardware, denial-of-service attacks, or weaknesses in cryptographic schemes. Next, we assume that the relying party and all the events allowed by the integrity criteria to be trustworthy. Moreover, we treat events that cannot be captured by the IM framework to be acceptable because we cannot say anything about their existence. It is important to note that such unobserved events may be harmful, but unless a mechanism can detect the degradation, it is hard to know the harm that has occurred. We consider the following threats in the IVP design. We assume a powerful external adversary who can produce external events upon the proving system that may exploit vulnerabilities. Such external events may affect both loadtime (e.g., modify files in a downloaded distribution) and runtime events (e.g., network communications). Finally, we consider attacks that modify remote storage and offline attacks on the proving system's local disk.

4.3.3 Architecture Overview

Figure 4.4 illustrates our architecture for enforcing the integrity criteria of a relying party (the client) over a network connection to an application VM. Here, the IVP is a service resident in the VM's host that verifies the integrity of the VM on behalf of the client. The client first (1) registers her integrity criteria with the IVP service. Next, (2) she establishes trust in the VM's host and IVP service by verifying their integrity through traditional attestation protocols. These components are designed to maintain their integrity at runtime, thereby enabling simple verification through loadtime measurements similar to existing protocols like IMA [30]. This verification is needed to trust the IVP to correctly enforce her criteria.

The client then requests a connection to a specific hosted VM the criteria to enforce over the channel. The IVP's *integrity monitor* is responsible for tracking the ongoing integrity of the hosted VMs relative to the client's criteria. It uses a set of *measurement*

modules to interface directly with the host's IM framework and capture integrity-relevant events, which are reported back to the monitor. If the monitor (3) determines that the VM satisfies the client's criteria, it then (4) establishes a secured network tunnel between the client and VM through the IVP's *channel mediator*. The mediator associates each tunnel with the client's criteria. If the integrity monitor detects a that a VM has violated the criteria of any connect client, it notifies the mediator to (5) terminate each associated connection.

4.3.4 Verifying the IVP Platform

The IVP verifies VM integrity on behalf of the client, thereby requiring trust in the IVP. Since our aim is to reduce client verification effort and eliminate the need for repeated remote attestation, we want an IVP that can be verified by a single attestation at channel setup unless a reboot occurs. The challenge is then building IVPs and their hosting platform in such a way that they maintain their integrity to obviate the need for remote monitoring.

This endeavor is difficult in general because systems often have large TCBs consisting of numerous components that may not be trusted. Moreover, changes to these systems at runtime like upgrades may be overlooked without frequent monitoring. However, various research projects have explored techniques for building VM hosting platforms that may be small enough to verify formally [6, 99, 100, 65, 64, 101]. While the design of a specific platform is outside the scope of this chapter, we envision a host would incorporate such approaches. As for the IVP, it only relies on a small number of services, such as networking, the introspection interface, and VM management. Research projects like Proxos [102] and work by Murray et. al. [103], have demonstrated that it is possible to build minimal VMs that depend only on the VMM and use untrusted services in other VMs securely (e.g., by encrypting and integrity-protecting the data). This would enable the IVP to function as an independent service in the host without depending on a large host VM like in Xen's Dom0. We intend to develop future IVP prototypes for various hypervisors that support this separation.

4.3.5 Channel Mediation

The IVP is responsible for mediating connections to ensure they are active only when their respective client's criteria are satisfied. The channel mediator creates an integrity association (IA) for each tunnel as the tuple (C, V, I), where C is the client, V is the VM, and I is the integrity criteria to check. Before a tunnel is brought up, the IA is

registered with the integrity monitor to verify that V continues to satisfy I. If it does, the tunnel is brought up and shutdown either voluntarily or when the integrity monitor notifies the mediator that an I has been violated.

One challenge in designing the channel mediator is proving to clients that the channel is controlled and protected by the proxy. The connection is formed as an Ethernet tunnel between the client and the VM through a virtual network managed by the mediator. This effectively places the client and VM on the same local subnet. Other mediated connections to the VM connect over the same virtual network, but are isolated from each other by the mediator using VLAN tagging. During setup, the tunnel is protected via cryptographic protocols like TLS that mutually authenticate the client and mediator. The VM is provided a certificate signed by the host's TPM at boot time to bind the platform's identity to the VM's credentials. This binding approach is similar to previous work on VM attestation [32, 60]. The client can then setup further protections directly with the VM over the tunnel. Having direct control over the network tunnel also lets the mediator tear down the connections as soon as a violation is detected.

4.3.6 Integrity Monitoring

The IVP's integrity monitor is tasked with verifying each VM's integrity against integrity criteria registered by clients connected to it. To do this, the monitor collects events from its measurement modules (see Section 4.3.7) to update its view of each VM's configuration. When the mediator registers an IA, the monitor first checks if the IA's criteria is satisfied by the current VM configuration. If so, the monitor adds a reference to the IA to a list of IAs to verify. When the VM's configuration changes, (e.g., through code loading) the integrity monitor pauses the VM and checks whether any registered IA has been violated. If so, the channel mediator is notified of the invalid IA, so it may tear down the tunnel before the VM can send data on it. The monitor then resumes execution of the VM.

In order to verify a VM's integrity, the monitor must be able to capture all integrity-relevant changes from VM creation until shutdown. To monitor loadtime events, we give the integrity monitor direct control over VM creation through the platform independent virtualization API, `libvirt`. This lets the monitor collect information about the VM's virtual hardware, initial boot parameters, kernel version, and disk image. The monitor spawns individual watcher threads for each VM and registers with the IVP's measurement modules. When the modules capture an event at runtime, the watcher is alerted with the details of the change. Since multiple IAs to the same VM may have redundant

requirements to verify, the monitor keeps a lookup table that maps IAs with the same criterion together. When a change to the VM is detected that violates one of these conditions, all IAs mapped to that criterion are invalidated by the monitor.

4.3.7 Measurement Modules

Integrity criteria consist of various loadtime and runtime requirements. The integrity monitor divides up these criteria into a set of discrete measurement modules tasked with tracking changes to specific aspects of the VM's configuration. The modules interface directly with the available IM framework to measure events in real time. For example, loadtime modules measure information like boot time parameters of the VM, while runtime modules attach a VMI to watch critical data structures. Since IM frameworks often consist of several components responsible for measuring various events, modularizing the interface allows for a more flexible design. Administrators can then write or obtain modules for the specific IM mechanism installed on the host without having to modify the monitor.

Capturing runtime events. Detecting violations at runtime requires modules to be able to capture events as they happen. The module must then notify the integrity monitor's watcher of the event before the VM continues to execute. We employ VMI to enable our modules to monitor runtime criterion. Many hypervisors now offer VMI mechanisms like xenaccess [69, 70, 104] in Xen and VMSafe [105] for VMware that enable direct access to VM resources. In addition, QEMU supports introspection through debugging tools like `gdb` and previous work has demonstrated the feasibility of VMI in KVM [67].

Each runtime module monitors a specific property on the VM. The modules actively monitor the VMs by setting *watchpoints* (e.g., locations in memory) that are triggered by integrity-relevant operations. Watchpoints can be set on sensitive data structures or regions of memory such as enforcement hooks [106], and policy vectors [3] stored in kernel memory. Other structures like function pointers and control flow variables are possible candidates [107]. Triggering a watchpoint pauses the executing VM so the module that set the watchpoint can examine the how the configuration has been altered. Pausing the VM prevents the VM from sending any data on the connection until the module can assess if the event violated an IA's criteria. After the module finishes invalidating any IAs, the VM is permitted to resume execution.

Improving efficiency. VMI gives runtime modules direct memory access, but creates a

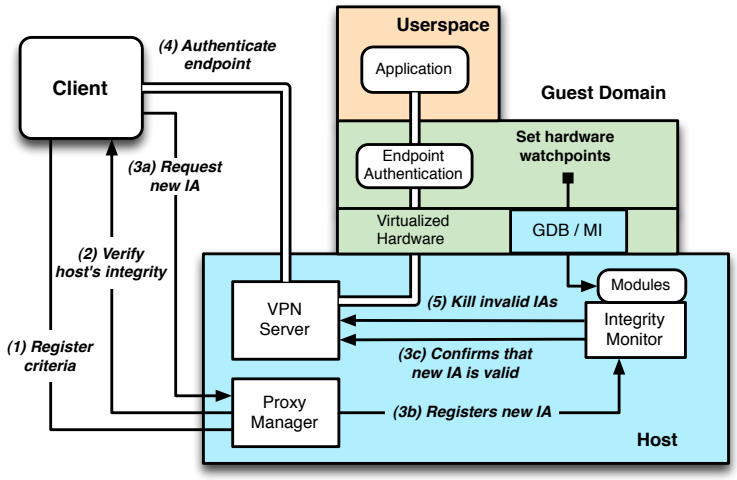

Figure 4.5. IVP implementation and protocol.

semantic gap [108] when reading directly from the VM's memory. Since the module does not have the full context of the running system, changes to complex and userspace data structures are difficult to assess. Our modules leverage the VM's extant enforcement mechanisms to report events without having to pause the VM as often. For example, instead of pausing the VM to measure every executed program, we use Linux kernel's LIM [53] framework to record hashes of every previously unseen program and executable memory-mapped file before loading them. We set a watchpoint on the in-kernel measurement list to catch each addition to it. This way, the module can avoid pausing except when LIM detects new binaries. Other in-VM monitor techniques could be leveraged to report integrity measurements to the modules to reduce the overhead of pausing the VM. Virtual devices like the vTPM [32] and co-resident monitors like SIM [67] provide potential reporting frameworks.

4.4 Implementing an IVP

We implemented a proof-of-concept IVP for a Linux KVM system. Figure 4.5 illustrates the IVP's services residing in the host. Clients interact with the IVP through a *proxy manager* to (1) register their criteria, (2) request attestations of the host's configuration, and (3) manage connections to VMs. We used a TLS-protected VPN tunnel to the VM's virtualized private network to implement the IVP's channel mediator. Initially, VMs are firewalled from the client's tunnel and all clients are isolated from each other through the VPN as well. Once the tunnel is active, a client can establish an IA with a specific

VM by first (3a) sending a request to the proxy manager and specifying which criteria previously registered should be used to mediate that connection. The proxy manager then creates the IA tuple and (3b) registers it with the integrity monitor, which in turn checks if the client's criteria are satisfied by the VM. If it is, the monitor (3c) informs the proxy manager to change the VPN firewall to allow the VM to send data to the client over the tunnel. The client can now receive data from the monitored VM as well as (4) authenticate the identity of the VM to establish an encrypted connection if desired. Finally, if at anytime the VM violates the IA's criteria, the integrity monitor (5) deletes the IA and informs the VPN server to firewall the client tunnel from the VM.

4.4.1 Verifying the Host

To verify the IVP platform's integrity, we use the ROTI approach to attest to the trusted distribution of the host [92]. We also employ the `tboot` bootloader to establish a measured launch environment (MLE) for the host using TXT [26]. The MLE establishes a DRTM through the processor that isolates, measures, and executes the kernel and initial ramdisk (initrd). This allows the boot process to be started from a trusted starting point. The initrd loads the system enforcement policies into the kernel and takes a measurement of the current filesystem before passing execution off to the root filesystem. When a client requests an attestation of the IVP platform, the ROTI proof is included with the normal attestation. The client then checks that the proof indicates no tampering with the installation has occurred and that the installer source is trusted to produce a system designed to maintain its integrity at runtime to meet the long-term integrity requirements of the IVP platform.

4.4.2 Channel Mediator

We implemented the channel mediator using OpenVPN server to manage Ethernet tunnels from remote clients to the internal virtualized network for the hosted VMs. All mediated connections from the client are aggregated through a single VPN tunnel with the individual VM endpoints permitted to transmit on that tunnel if a corresponding IA exists. VPN tunnels are established by first mutually authenticating the client's account certificate and a host certificate signed by the host's AIK. Each connection is TLS-protected and uses a Linux tap device to provide kernel supported Ethernet tunneling from the physical network interface to the virtual network bridge. Once connected, the OpenVPN server opens the firewall for traffic from the VM's virtual interface to the tunnel for each VM in the active IAs to the client. When the integrity monitor deletes

an IA, it tells the OpenVPN server to firewall the VM from the client in the deleted IA.

4.4.3 Integrity Monitor

We created the integrity monitor as a 439 source lines of code (SLOC) Python daemon that manages VM execution and monitors VM integrity. The daemon uses the hypervisor independent interface, `libvirt`, to start and stop VMs, collect information about virtual device settings, and control loadtime VM parameters. When the daemon receives a request to start a VM, it spawns a separate *watcher* thread to control the VM and monitor integrity information. When the proxy manager registers a new IA with the monitor, the monitor forwards the IA to the appropriate VM's watcher, which in turn checks that each criterion is satisfied by querying the registered measurement modules for current VM configuration. If all the modules indicate the requirements are satisfied, the watcher notifies the VPN server that the IA is valid.

The watcher registers with loadtime measurement modules to collect information about the VM before the VM is started. Next, the VM is created and the watcher attaches `gdb` to running VM process, which pauses the VM. We use `gdb` as a proof of concept VMI interface because VMs in Linux KVM run as userspace processes, making them it simple to monitor. Moreover, `gdb` can determine where kernel structures are in memory by reading debug information in the kernel or from a separate system map file that is easily obtained. The watcher then loads the runtime modules, which collect the necessary context from the paused VM and set any desired watchpoints through the `gdb` interface. After the runtime modules are registered, the VM resumes execution. When watchpoints are triggered at runtime, the VM is paused and control is passed from the watcher to the runtime module that set it. The module then introspects into the VM's memory and updates the accumulated VM configuration with any modified values detected during introspection. The module notifies the watcher if any values have changed, which checks if those changes have violated any of the registered IA's criteria. Finally, the module resumes the VM's execution.

We use hardware-assisted watchpoints in `gdb` to avoid modifying the VM code and introducing additional overhead. This raises an issue because the x86 architecture only contains 4 debug registers, which limits the number of hardware-assisted watchpoints that can be set for a process. Since software watchpoints require single stepping through the VM's execution, they are not a viable option. However, similar watchpoint functionally is feasible by using memory protection features of the KVM shadow page table for VMs as demonstrated in SIM [67]. While we did not implement this VMI approach, we

plan to explore it and further implementation options in future work.

4.5 Evaluation

We evaluated our IVP implementation in terms of functionality and performance. First, we validated the IVP's ability enforce relying party criteria correctly by performing attacks that violated various integrity requirements. We then evaluated the performance overhead imposed on the monitored VM using both micro-benchmarks and application-level benchmarks performance.

Our experimental testbed consisted of a Dell OptiPlex 980 with a 3.46GHz Intel Core i5 Dual Core Processor, 8GB of RAM, and a 500GB SATA 3.0Gb/s hard disk. The Linux KVM host ran in an Ubuntu 10.10 distribution using a custom 2.6.35 Linux kernel. Our guest VMs were allocated a single 3.46GHz vCPU without SMP, 1GB of RAM, and an 8GB QCOW2 disk image connected via virtio drivers. Each VM ran an Ubuntu Linux 10.10 server distribution with default SELinux policy and a custom LIM module.

4.5.1 Functionality

To test the IVP's functionality, we designed a target application VM running the Apache webserver. We constructed a VM image that approximates the CW-Lite [36] integrity model and designed an integrity criteria for verifying that approximation. We then had a client connect to the VM through a mediated channel associated with the CW-Lite criteria. We performed several attacks on the VM's loadtime and runtime integrity both before and after the connection was established to see if the IVP would correctly detect the violations and terminate the connection.

4.5.1.1 Building a CW-Lite Enforcing VM

We constructed an application VM that satisfies the CW-Lite integrity model. This practical integrity model differs from strict integrity models like Biba [33] and Clark-Wilson [34] by allowing for an integrity policy that identifies trusted exceptions where illegal flows are required for the system to function properly. Other practical integrity models would also be viable [38, 35]. To enforce CW-Lite, trusted processes with high integrity labels (e.g. privileged daemons) must only (1) load trustworthy code, (2) receive trustworthy inputs, and (3) handle untrusted inputs through designated filtering interfaces that can upgrade or discard low integrity data.

We configured our Apache VM with SELinux, which enforces a mandatory access control policy through Domain Type Enforcement [109]. This labels every process and system object with policy-defined types. We use the Gokyo [110] policy analysis tool to identify 79 labels from which data can flow to the Apache process and system TCB labels [36]. This included processes that access critical resources like kernel interfaces and privileged daemons. We then modified SELinux LSM to hook into the kernel's LIM [53] to receive hashes of code executed in trusted processes. The modified LSM module then denies execution of hashes that are not on a white list obtained from the Ubuntu 10.10 main repository. This secure execution monitor satisfies the first CW-Lite requirement because only trusted code from the hash list may run in trusted processes.

In addition to the identified trusted processes, several untrusted sources like the network provide necessary input to Apache. Per the third CW-Lite requirement, we must ensure untrusted inputs are only received by interfaces[1] designed to properly handle (e.g. sanitize) such input. To do this, we added additional checks to the LIM policy to whitelist only the Apache binary, designed to handle such inputs, to be loaded into the process with labels to access these interfaces. Before the interface is permitted to read data, our modified SELinux LSM checks if the interface is intended to receive untrusted data based on a CW-Lite policy and deny the read if it is not.

4.5.1.2 Specifying integrity criteria

We defined our client's integrity criteria with both loadtime and runtime requirements. For loadtime criteria, we specified hashes of a trusted VM disk image, kernel, initrd, and CW-Lite enforcement policies to match those we created above. The runtime criteria, by contrast, checks for common signs of intrusion by rootkits and unexpected modification of the VM's enforcement mechanisms and policies at runtime.

For example, previous research [98] has shown that some rootkits modify the netfilter hook in the kernel to enable remote control of the system via specially crafted network packets [111]. Other attacks replace the binary format handlers to obtain privilege escalation triggered by program execution. We specified runtime criteria that require no changes to the kernel structures located by the kernel symbols `nf_hooks` for the netfilter and `formats` for binary format handlers attacks. We also identified function pointers used to hook execution by SELinux and LIM and in-kernel policy structures that should not be modified at runtime. Furthermore, we specified that only the Ubuntu repository

[1] Interface here refers to the read-like syscalls. While programs have many interfaces, only some are intended to handle untrusted inputs.

code was to be executed in the TCB, which would catch the case where the secure execution protections were bypassed. To do this, we specified that all measurements of code loads taken by the LIM hooks should match the hash list we specified above.

4.5.1.3 Building Measurement Modules

We constructed several measurement modules to monitor various integrity requirements on the target VM. The modules were written in an inheritable base class that exposes a register function for setting watchpoints and a callback handler that is called when the watchpoint is triggered. Each module averaged 25 additional lines over the base class definition. The integrity monitor's watcher thread instantiates and registers loadtime modules before the VM is first created to measure the kernel, disk image, and enforcement policies.

Runtime modules are instantiated and set watchpoints after VM initialization. When a watchpoint is triggered, the watcher is notified and invokes the appropriate module's callback to inspect the event. We placed watchpoints on various kernel structures including SELinux, LIM, and netfilter function pointers and the binary format handler list. We also monitored the in-kernel LIM policy by set a watchpoint on the kernel's `ima_measurements` list head. This traps to the runtime module whenever a new binary is executed. The module reads the hash from the list tail and adds it to the module's list of measured code. Doing this, we can monitor all code loaded in the TCB and check for inconsistencies between the expected LIM policy and executing programs. Leveraging the LIM framework to record new code hashes lets the integrity monitor pause the VM only when new binaries are loaded.

4.5.1.4 Detecting Violations

We tested if the IVP properly mediates the CW-Lite criteria before and after connecting a client to the VM over the mediated channel. We exercised each measurement module through a series of attacks on the VM's integrity. For loadtime modules, we modified boot parameters, kernel versions, disk image contents, and policy files to values not permitted by the criteria. The modules then recorded these configuration values at VM creation. When the client initiated connection request to the IVP, the integrity monitor's watcher compared the measured values to the criteria and correctly rejected the connection. For our runtime modules, deployed attacks on the monitored data structures using attack code that exploits an x86 compatibility vulnerability in Linux kernels older than 2.6.36 [112]. This let us illegally change an unprivileged process' SELinux label

SPECINT '06 Benchmarks	Median (sec) Base	Test	Diff (%)
perlbench	403	404	0.25
bzip2	683	686	0.43
gcc	367	369	0.54
mcf	557	560	0.53
gobmk	467	467	0.00
hmmer	544	545	0.18
sjeng	575	576	0.17
libquantum	664	667	0.45
h264ref	762	763	0.13
omnetpp	494	497	0.61
astar	664	667	0.45

Table 4.1. Benchmarks with and without the IVP obtained by the median of three runs, as reported by the SPECINT 2006. The test suite does training and test runs in addition to the actual runs so the results are reproducible.

to the full privileged `kernel_t` label, thereby enabling arbitrary code execution. We used this vector to easily modify kernel memory and modify the monitored structures to violate our runtime requirements. The IVP correctly detected these changes and disconnected the connection to the VM and prevented future connection requests.

4.5.2 Performance

Next, we examined the performance impact the IVP has on monitored application VMs. We first performed a series of CPU and I/O micro-benchmarks within the monitored VM to identify any overhead in system performance indicators. We then performed macro-benchmarks with a webserver and distributed compilation VM to see the impact at the application-level.

4.5.2.1 Passive Overhead

We first evaluated the impact of runtime monitoring on the VM when integrity-relevant events are not occurring. We used three types of benchmarks to test CPU, network, and disk performance of the VM with and without the IVP active. For CPU-bound benchmarking, we used the SPECINT 2006 test suite (see Table 4.1), which performs several training runs to identify the expected standard deviation (under 1.1%) before sampling. Most tests show negligible overhead with the IVP with the largest at 0.61%.

Table 4.3 shows our results for network and disk benchmarks after 30 runs of each. We used `netperf` to evaluate network overhead. It samples maximum throughput and

Operation	Mean (± 95% CI) (ms)
Watchpoint Trigger	
VM Exit and Entry	.006 (± 0.000)
QEMU overhead	.496 (± 0.081)
GDB overhead	.327 (± 0.054)
Monitor Overhead	0.172 (± 0.028)
Runtime Modules	
Collect LIM Hash	66.76 (± 0.215)
Read kernel variable	0.132 (± 0.002)

Table 4.2. Active Overhead Micro-benchmarks of overhead incurred when watchpoint is triggered. World switches and GDB contributes 82.2% of the trigger overhead excluding modules. Collected from 100 runs.

transactions per second after saturating the network link. These tests also indicated negligible impact on networking. For disk I/O performance, we used the `dbench` benchmarking tool, which simulates a range of filesystem level operations using a configurable range of parallel processes. It presents results as the average throughput for the client processes. We found that the throughput was negatively affected as we increased the number of simultaneous clients. Our intuition for this trend was that more client processes led to more syscalls that, in turn, cause the VM process to raise signals to perform I/O through virtual devices. We profiled the VM with systrace while the benchmarks were executing and confirm this correlation. Since `gdb` uses the `ptrace` interface in the kernel to monitor processes for debug signals, every syscall incurred a small processing overhead by `gdb` to parse the signal and resume process execution. A possible solution for this would be to modify the ptrace interface to notify the `gdb` process only when debug signals are raised. Even with this overhead, our disk I/O benchmarks demonstrate overhead under 8% for 50 clients.

We also tested the effect of our IVP on two real-world applications, an Apache webserver and a `distcc` compilation VM. We initiated all of our tests from a separate computer over the TLS-protected VPN tunnel setup by the IVP. We ran 30 runs of the `ab` tool to simulate 100 concurrent clients performing 100,000 requests on the Apache VM. For the `distcc` test, we compiled Apache-2.2.19 across 3 identical VMs on separate machines with 8 concurrent threads. Again, the average of 30 such runs are taken. Our results show that the IVP introduced a 1.44% and 0.38% overhead on Apache and distcc VMs, respectively. We suspect the primary cause for the Apache overhead is the frequent network requests and disk accesses made to service the requests.

Benchmarks	Mean ± 95% CI		Diff
	Baseline	With IVP	(%)
Network: `netperf`			
TCP_STREAM (Mb/s)	268 ± 0.23	269 ± 0.22	0.2
TCP_RR (Trans/s)	1141 ± 5.65	1141 ± 1.96	0.05
Disk: `dbench`			
1 Client (Mb/s)	11.14 ± 0.02	11.12 ± 0.14	0.18
5 Clients (Mb/s)	32.64 ± 0.67	32.49 ± 0.76	0.46
10 Clients (Mb/s)	40.94 ± 1.01	40.21 ± 0.98	1.78
20 Clients (Mb/s)	47.46 ± 1.50	44.69 ± 1.12	5.83
50 Clients (Mb/s)	40.58 ± 3.09	37.41 ± 1.86	7.81

Table 4.3. Network and disk benchmarks. netperf measures throughput and transactions per second. dbench measures disk throughput. 30 runs per benchmark.

4.5.2.2 Active Overhead

Finally, we explored the delays introduced by the IVP when handling changes to monitored data structures. We profiled our measurement modules using the ftrace framework in the Linux kernel by setting markers to synchronize timings between our userspace monitor and events happening in the kernel, such as VM exits and enters. Table 4.2 shows that interrupting the VM on a tripped watchpoint introduces a 1 ms pause regardless of the measurement module involved. For simple runtime modules that read single variables, approximately 100μs additional overhead is incurred. However, more complex measurement modules take more time. For example, the LIM measurement module reads a SHA1 hash from a nested kernel list, which causes a 67 ms delay. We found the majority of this is caused by `gdb` parsing the kernel symbol table to locate the memory addresses in the VM to read. Caching these addresses when the monitor is registered would greatly speed this measurement process. Regardless, measurement modules that perform more complex measurements like reading and parsing multiple structures will increase the time the VM is paused. Moreover, watchpoints on frequently modified memory locations will result in more pauses.

4.6 Related Work

Introduction of the TPM has led to numerous IM techniques (see the comprehensive survey by Parno et. al. [21]). Initial approaches focused on TCG-style verification of the boot process, the OS kernel, modules, userspace binaries [30, 53] and system policies [41]. Application-level measurements through frameworks like Trousers [94] enable processes to pass measurements to the TPM for integrity protection and reporting.

Other techniques measured VM integrity through hypervisor support [10, 113, 79] and even virtualized the TPM for VMs [32]. More recently, Sirer et. al. [114] proposed an authorization logic supported by a custom OS kernel that enables verification using high-level statements instead of binary hashes. This approach greatly simplifies the complexity of verifying attestations and provides a richer measurement framework for both local and remote entities. However, these approaches place the verification burden on the relying party to interpret potentially stale and incomplete information. The IVP can leverage these disparate measurement techniques to verify a relying party's criteria at the proving system and supplement them with more fine-grain monitoring.

Other approaches have focused on reducing the TCB that must be verified. Bind [40], Flicker [63], and TrustVisor [64] use CPU hardware support to measure and protect the execution environment of application code and associate it with the computation's result. These approaches provide guarantees to the relying party that the result was protected from external threats during execution, but still require verification of each result's attestation.

Instead of attesting system configurations, other research has focused on maintaining runtime integrity guarantees [71, 66, 68, 67, 44, 100, 115] that remote parties can verify are being enforced. For example, Terra's Optimistic Attestation ensure certain VM disk blocks are unaltered by shutting down the VM if a modification is detected at loadtime. These approaches offer a strong foundation for monitoring runtime integrity, but do not support verifying remote verifier specified requirements. Our IVP can leverage these runtime enforcement mechanisms to maintain the IVP host's integrity. Furthermore, remote parties can use the IVP to monitor the integrity of enforcement mechanisms in the VM and their policies. Also, our design does not explicitly provide remediation like shutting down the VM because we assume the remote clients are not administrators of the VM and may have differing criteria.

IM has also been incorporated into secure communication channels. Trusted Network Connect [42] requests periodic attestations of clients before and after they join a private network and evicts systems with invalid attestations. OpenTC PET [43] uses SSL proxies in a VM host to provide attestations of the VM to the remote client. However, the proxy simply provides attestations instead of verifying the VM's integrity on behalf of the connected client. Other work [60, 61, 32, 59] has incorporated TPM attestations into public key certificates to bind integrity states to platform identities. However, the reported integrity of these approaches is only valid as long as the attested system's configuration does not change. This requires the client to continually request new certificates that

function exactly like attestations. Our IVP eliminates the need for continual polling by enforcing the client's criteria at the VM's host.

4.7 Conclusion

In this chapter, we presented the IVP, a service resident in a proving system that mediates connections on behalf of remote clients. By shifting the task of monitoring a client's integrity criteria to the proving system's host, we enable relying parties to connect to remote systems without the need for frequent attestations or further verification. We designed and implemented a proof of concept IVP for a Linux KVM host and evaluated its effectiveness and impact on performance. Our results show the IVP incurs only minor overhead for network and CPU-bound applications, but with additional delay that increases modestly as a function of I/O load. As future work, we plan to improve our VMI interface to minimize passive overhead and increase expressiveness of client's integrity criteria.

Chapter 5

Cloud Verifier: Verifiable Auditing Service for IaaS Clouds

Cloud computing has revolutionized computing by commoditizing compute, storage, and networking resources into an on-demand utility. However, adoption of cloud computing has been stymied by a lack of transparency that leaves customers unsure whether to trust these remotely administered systems. While techniques like encryption may provide confidentiality and integrity of customer data at rest, it is still not practical to process this encrypted data efficiently. Instead, trusted computing approaches aim to verify that the *integrity of the underlying system* is satisfactory for protecting user data and instances at runtime. However, current solutions are insufficient for enforcing a broad range of integrity requirements in a timely and efficient manner.

In this Chapter, we present the design and implementation of the *cloud verifier* (CV), a flexible multi-tiered framework that integrates directly into existing Infrastructure as a Service (IaaS) clouds to act as a verifiable auditing service within the cloud. The CV leverages a verification proxy within each virtual machine host to obtain a comprehensive view of each instance executing on the cloud, check that client-specified integrity invariants against the collected instance state, and disconnects clients from instances that violate such requirements. We built a proof of concept CV for the OpenStack cloud platform. Our experiments demonstrate that the CV is able to verify that numerous properties about the instances are satisfied including network configuration, disk image integrity, kernel data structure invariants, and mandatory access control policy queries. Moreover, our application benchmarks show the CV introduces minor overhead on cloud and instance performance.

5.1 Introduction

Cloud computing has revolutionized they way we consume computing resources. Instead of maintaining a locally administered data center, businesses and individuals can simply purchase compute, storage, and network resources on demand from a public cloud utility. Clouds come in a variety of models ranging from VM hosting to fully managed web services [116]. For example, IaaS clouds like Amazon's EC2 [117] and Rackspace Cloud Servers [118] provide fully customizable virtualized computing infrastructures.

While this new model has increased access to affordable resources, it comes with new and challenging security risks. By using remotely administered systems, cloud customers are no longer in able to maintain *visibility* into their computing infrastructure. Little assurance is provided that the security settings they specify are properly enforced or that their instances are running unmolested. Moreover, they cannot observe how the underlying hosting platform has been configured and whether it will protect their instances and data from threats like co-resident instances [119], malicious insiders [120, 121], and misconfigurations by data-center administrators [122, 123, 124]. The problem is only compounded when considering the risks clients of these cloud-hosted services must blindly accept when trusting both the cloud and the service provider with their sensitive data.

Cryptographic protocols have been used to protect data confidentially of data in the cloud while maintaining required functionality. This works well with services like Dropbox and Amazon's S3 key-object store that simply host data and do not require changes to the application. However, encrypted data can be problematic for services that must read plaintext to function correctly. Techniques like encrypted database queries [125, 126, 127], homomorphic cryptosystems [128, 129], and targeted data encryption [130] offer degree of functionality with encrypted data. However, these approaches are not yet sufficient for implementing general purpose cloud application as they introduce high overhead and limit the set of operations on such data.

Instead of treating the cloud as a malicious adversary, researchers have leveraged emerging trusted computing hardware to verify the *integrity* of these remote systems [21]. That is, they verify whether the software components and security policies currently in place will protect user data and instances. While initial efforts aimed for load-time verification of individual systems [30, 53, 41, 22] and isolated execution environments [63, 40, 64], recent projects have focused on verifying distributed systems [80, 79] and even cloud platforms [131, 132, 133, 134, 135]. However, these solutions are insufficient for providing a complete view of the cloud's integrity. In particular, their reliance

on inefficient attestation protocols that confer incomplete information limit their utility for monitoring cloud integrity [37, 136]. Moreover, these proposals do not support verification of arbitrary requirements that cloud customers and their clients may specify.

In this chapter, we present the design, implementation, and evaluation of the *Cloud Verifier (CV)*, a verifiable framework for monitoring integrity in IaaS clouds on behalf of its administrators, customers, and external clients. Our key observation is that by verifying integrity requirements at the cloud instead of at the relying parties, more comprehensive and efficient enforcement can be achieved. To do this, the CV leverages the cloud's hierarchical structure to build transitive trust starting in the cloud platform up to the instances themselves. Platform integrity is verified against the cloud administrator's integrity criteria, thereby preventing maliciously modified systems from executing customer VMs. From there, cloud customers and external clients specify their own integrity criteria to the CV, which distributes those requirements to an IVP[1] service on each VM host. This IVP monitors each instance's state to detect changes on the VM or its host that violate those requirements. If those requirements are violated, remediation is then performed for clients by cutting their connection and for application providers by rolling back VMs to a known good state.

The CV achieves our goals of building a more transparent cloud in several ways. It monitors the cloud platform according to an administrator policy while revealing that process to external parties. In addition, cloud customers and their clients can then use cloud-hosted services with the guarantee that their integrity requirements are satisfied. In order to provide these guarantees, we had to design the CV to address several key challenges. 1) How can remote parties establish transitive trust in an opaque hosting platform? 2) How can the CV verify a broad range of integrity requirements? Finally, 3) how can enforcement and remediation of those criteria be done in a timely and efficient manner?

Transitive Verification of the Cloud. Verification of a cloud instance is not confined to simply checking that the code and data within the VM are trustworthy. A remote verifier must also consider the integrity of the entire hosting platform as well. However, verification of the cloud is not a simple task. First, VMs may run on any arbitrary cloud node, but establishing the identity of that machine is difficult without a public PKI. Second, cloud nodes often contain custom code and data that is hard to assess and may contain secrets that should not be publicly disclosed. Finally, the cloud may migrate the VM at any time to a different node to balance resource constraints, which

[1] See Chapter 4 for more detail on the IVP.

requires the client to reestablish trust in the new hosting node. The CV reduces the complexity of verifying the platform by monitoring the cloud nodes based on the cloud administrator's verification criteria. Compromised nodes are prevented from joining the cloud at boot time and ejected from the cloud if they are compromised at runtime. Thus, remote verifiers can establish transitive trust in the nodes by verifying that the CV will correctly monitor the nodes and that the criteria it uses is acceptable.

Comprehensive Monitoring of Instance Configurations. Clients and cloud customers may have different requirements for a trustworthy instance. The CV must support these different criteria by collecting sufficient information about the VM's state to verify a broad range of them. In addition to the information reported by the VM's measurement framework (e.g., vTPM [32]), the CV leverages an extensible set of modules designed to gather information relevant to specific criteria from the available measurement interfaces. As an example, one module we designed hooks into a VM introspection interface to monitor kernel data structures for certain rootkit behavior. In practice, additional modules will add to the breadth of requirements that the CV can inspect.

Timely Remediation after Integrity Violations. Current verification approaches rely on the remote verifier to provide remediation when a violation is detected. This often introduces windows where malicious or undesirable behavior may occur because the verifier is notified too late. The CV addresses this by enforcing the client's and cloud customer's requirements at the cloud. For example, clients do not want to communicate with an instance that may be compromised and exfiltrating secrets. The CV can sever the connection to that cloud node the instant a violation of the client's criteria is detected. Ultimately, the scale of remediation is dependent on what authority the relying system has over the mediated instance like an instance owner requesting a VM reboot.

Contribution. We implemented the CV as a service integrated into the open source IaaS cloud platform, OpenStack [137]. Our prototype includes a service that verifies each cloud service's machine at boot-time before it is allowed to participate in cloud functions. Clients and cloud customers connect to the CV through a well-defined port that is redirected from the VM to the CV. They can then verify the CV through remote attestation and specify their integrity criteria for the VM they wish to monitor. Criteria are composed of invariants that are checked by the hosting node's verification service. The CV returns signed certificates indicating the specified criteria is enforced and performs further remediation when violations are detected. We implemented several measurement modules, including VM introspection, SELinux policy analysis, and VM

configuration collection that can check various criteria including detecting root-kits, VM instances tampering, and unsafe access control policies. We evaluated the performance overhead introduced by the CV's services on several real-world application VMs and found the impact to be minor.

The main contributions of this chapter are as follows:

- We present the design of the Cloud Verifier, an auditing service for IaaS clouds that enables verification of the cloud platform and its instances on behalf of cloud administrators, customers, and clients.
- Our CV facilitates transitive verification of the cloud's components based on the cloud administrator's integrity policy. This simplifies verification complexity for the client by reducing traditional remote attestation over multiple systems down to checking the validity several signatures.
- We built a proof of concept CV for OpenStack that demonstrates the feasibility of our design. We evaluated the prototype's functionality and performance through several measurement modules that verified a range of integrity requirements on real world applications.

The rest of this chapter is organized as follows. In Section 5.2, we provide an overview of the Cloud Verifier and describe the challenges involved in verifying cloud platforms. In Section 5.3 introduce the design of the Cloud Verifier and its integration into OpenStack in Section 5.4. Section 5.5 presents our evaluation of the prototype on several application VMs. We then describe related work on integrity verification and accountability in the cloud in Section 5.6 before concluding in Section 5.7.

5.2 Verifying Cloud Platforms

In this section we first provide an example of a typical an IaaS cloud and illustrate the challenges in verifying its integrity for various parties. When then provide a sketch of our proposed solution and explain how it addresses the issues we raised in our example.

5.2.1 IaaS Cloud Architecture

Clouds come in a variety of architectures with differing levels of service and features. While the definition of a "cloud" is as nebulous as its name, NIST has begun to categorize clouds based on the degree of administration and services the clouds offer to the customer [116]. To think about clouds, consider a service that cloud customer wishes to

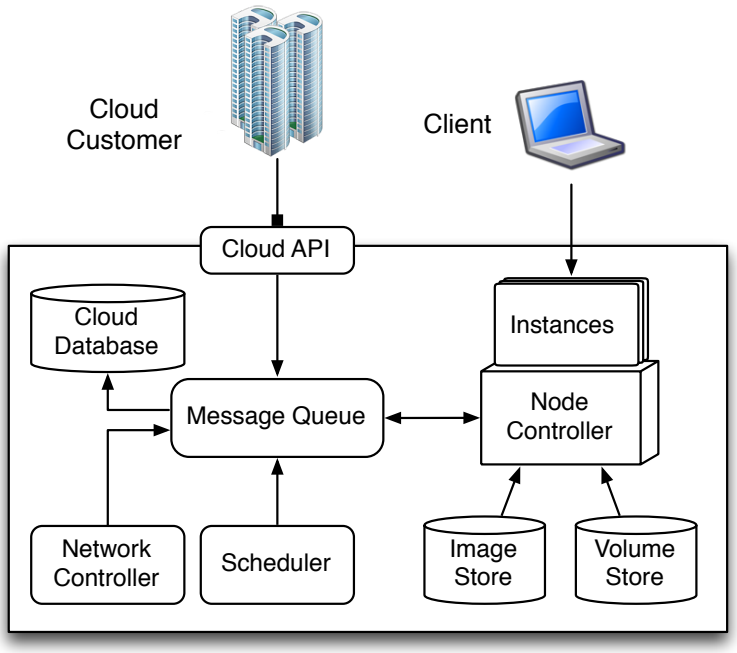

Figure 5.1. IaaS Clouds.

use. At the lowest level, IaaS clouds provide the basic infrastructure of a data center needed to implement that service. Typically this involves VM hosting, virtualized networking, and storage for VM images and disk volumes. Customers can use IaaS clouds to replace or supplement a traditional data center by hosting the service in the cloud and scaling up compute and storage requirements on-demand. Examples of IaaS clouds include Amazon's EC2 [117] and Rackspace's Cloud Servers [118]. *Platform as a Service (PaaS)* clouds like Microsoft Azure [138] and Google App Engine [139] take the IaaS architecture one step further by adding an additional layer of instance management. The customer can then run their service's code like Java, Python, or C# without the burden of managing VM images, networking, or replication. Finally, *Service as a Service (SaaS)* clouds simply provides the entire service to the customer. SaaS clouds like Salesforce.com and Google Docs provide users with a full fledged service without the hassle of managing it. In this chapter, we consider only IaaS clouds as they are the building blocks of higher level cloud abstractions. Moreover, the IaaS paradigm gives the customer more control over how cloud components manage sensitive data and code.

As an illustrative example, consider the high-level IaaS cloud architecture in Figure 5.1. The primary component is the *node controller*, a VM host for customer VM instances. Clouds are composed of thousands of these nodes, which are broken into clusters that provide a level of redundancy and can be spread out geographically based

on demand of the region. Within each cluster, a *network controller* is responsible for configuring virtual networking between instances and translating public IPs to private intra-cloud addresses. When a new instance is requested, a *scheduler* chooses a node controller to host the VM based on various scheduling policies like resource fairness. In addition, the cloud's state (e.g., which instances are running) is stored in a *cloud database* that is updated and referenced by the various components.

Instances are created from disk images stored in the *image store*. They are uploaded to the cloud by the customer or a third party vendor and remain static across reboots. Additional mutable storage is provided through additional services like a key-object stores (e.g., Amazon's S3 [117]) or network attached block storage from a *volume store*. Customers control their instances through an API endpoint, which also exposes options for configuring firewall rules, `ssh` host identity keys, and other policies. Finally, instances open to the internet can interact with clients to provide services for which they were designed.

In discussing cloud integrity, we assume the physical security of the cloud is maintained and that attacks on the hardware are prevented. We also trust the cloud at an organizational level to provide services without malicious intent. That is, we assume the cloud's components were honestly configured with the purpose of protecting the integrity and confidentiality of its customers. However, we do not trust the cloud beyond that point and accept that curious or malfeasant administrators may attempt to alter cloud systems in an untrustworthy way. Thus, we consider threats like an administrator logging in to the node controller to directly read instance memory or locally cached disk images. We also consider threats from network attackers both within the cloud intranet and externally that can snoop on, alter, or inject packets. We do not guarantee that satisfaction of an integrity criteria implies that the system will not perform undesirable behavior. We leave it up to the relying party to design integrity criteria that would ensure this property.

5.2.2 Proposed Approach

The goal of the CV is to enable multiple stakeholders (clients, administrators, etc.) to verify that their integrity requirements are satisfied while minimizing their verification effort. In other words, we want to provide the strong guarantees of integrity verification, but in a manner as simple to use as checking a certificate chain. To that end, the CV is designed to enable clients to establish trust in the cloud's auditing service that will verify the remote party's integrity criteria for them. The framework consists of several layers

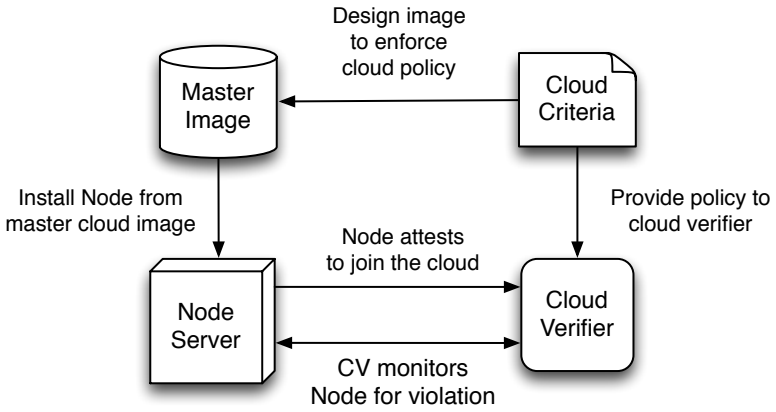

Figure 5.2. Clouds Join Cycle.

of verification that enable the client to transitively build trust in the cloud instance from the platform up. Using this layered approach, we aim to reduce effort on the client end while improving the breadth of integrity requirements that can be verified.

Enforcing Cloud Platform Integrity. Enabling remote clients to verify cloud instances through a hosting platform that verifies the client's integrity criteria requires the relying party to also believe that the cloud platform will function correctly on its behalf. To that end, the CV ensures that all cloud components satisfy a single integrity criteria specified by the cloud administrator, which clients can then compare to their own requirements. The CV does this by verifying services like the node controllers, network controllers, and API endpoints before they can join the cloud and ejects systems that violate the cloud-wide criteria. This maintains the following invariant over the cloud:

Invariant 1 (Platform Integrity)**.** *All cloud components actively participating in a CV-verified cloud satisfy the cloud administrator's integrity criteria.*

Figure 5.2 illustrates the high-level protocol for a generic IaaS cloud node. First, the cloud administrators formulate an integrity criteria for a trustworthy cloud node. Next, the administrators design a master disk image for all nodes in the cloud that satisfies the criteria. The disk image is then pushed out to all of the node machines through network installation. When a node boots, it must request to "join" the cloud in order to send / receive messages and host instances. To do this, the CV obtains a remote attestation from the node and checks it against the cloud criteria. If proving node satisfies the requirements, the CV informs the cloud that the node has joined. The CV continues the monitor the node for changes and "ejects" the cloud if violates the cloud criteria.

This protocol enforces the above invariant and ensures all monitored components

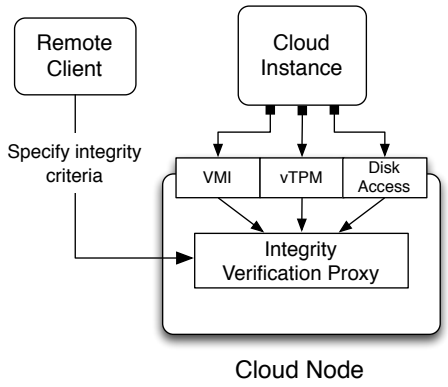

Figure 5.3. Integrity Verification Proxy Overview

are currently satisfying the cloud criteria. As a result, the CV effectively speaks for the integrity of the cloud platform. A remote client can then assess the integrity of the cloud by verifying the CV is trustworthy and then comparing the enforced criteria to the client's requirements. If they are acceptable, the client establishes a persistent connection to the CV to monitor its runtime integrity and ensure it continues to speak accurately for the cloud platform. If the CV later violate's the clients requirements, then the client can no longer trust the CV and must discontinue using it.

Monitoring VM Integrity. The second layer of the CV framework verifies cloud instance integrity on behalf of remote clients. This is done through an IVP resident in the cloud node [136]. Figure 5.3 illustrates the IVP, a modular service that collects the evolving state of instances it hosts through the available IM interfaces (e.g., VMI, vTPM, etc) used for reporting integrity-relevant events within the instance. The IVP uses this accumulated state to determine whether a client's supplied integrity criteria is satisfied by the current state of the instance. The client's criteria is sent to the IVP via the CV when the client wishes to use the instance. If the IVP finds the monitored instance satisfies the client's criteria, it generates a signed certificate vouching that the instance's configuration satisfies the client's criteria. Formally, we can say:

Definition 1 (Vouches For)**.** *For a verifier V, prover P, and criteria C, the statement $V \xrightarrow{C} P$ means V believes the configuration of P satisfies C.*

Thus, for the client's criteria C the IVP states
$IVP \xrightarrow{C} Instance$ to the remote client. This statement is only valid while the IVP continues to assert it and could be revoked at anytime. The IVP indicates the statement's revocation through disconnecting the client from the instance.

The advantage of using the IVP over traditional remote attestation is manifold. First, by continually gathering information about the instance's changing configuration, the IVP always has the most up-to-date view with which to assess instance integrity. Achieving the same level of freshness is effectively impossible given the poor performance of trusted hardware like the TPM, which takes roughly a second to generate a single attestation [37]. This is especially challenging in high demand scenarios with thousands of users.

Second, the IVP has direct access to measurement interfaces. This provides greater detail and frequency about integrity-relevant events than what would otherwise be present in attestations. This is especially true for interfaces that monitor a many complex events. For example, the VMI interface in Figure 5.3 can observe writes to a VM's kernel memory. A remote verifier could use knowledge of these writes to detect malicious behavior like that of a root-kit. However, measuring and attesting every write to memory would be prohibitively expensive if done naively. Moreover, many changes would most likely be benign or irrelevant to the remote verifier. Instead, the IVP stores these changes locally to verify client criteria.

Finally, by assessing criteria at the host, the IVP is able to verify a broader range of requirements than what may be possible with attestations. Since data presented in an attestation is formatted in a way chosen by the proving system's administrator, remote verifiers are only able to make limited inferences about the system's configuration. For example, an attestation containing the hash of configuration files is less useful than having the actual file to parse, but it would be infeasible to send every file to the remote verifier. With the IVP, a client can specify some invariants on the configuration file, which, in turn, could be checked by the IVP.

Remediation. In addition to verifying the integrity of the cloud platform and its instances, the CV enables the remote verifier to perform remediation in response to violations of its criteria. However, the degree of remediation is dependent on the authority of the remote party over the monitored component. For the cloud administrator, when the CV detects violating cloud components, it protects the cloud by ejecting that component from the cloud. This involves rebooting the machine and re-imaging it back to the master disk image. For the cloud customers like service providers that host instances on the cloud, when the CV can detect a violating instance, it can shutdown the image and rollback the attached storage volume to snapshot before the VM started. Finally, the CV automatically disconnects clients from cloud instances that violate their integrity requirements. For each of these remediations, the CV is able to perform these actions

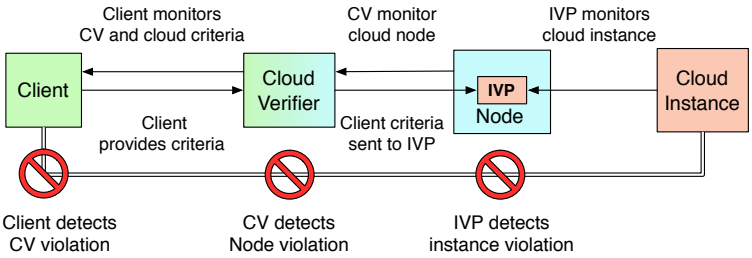

Figure 5.4. Connection Protocol Sketch

sooner than the remote party could because the delay introduced by attestation. As a result, the window for malicious behavior by the monitored system is reduced using the CV.

Protocol Sketch. We now provide a brief sketch in Figure 5.4 of the protocol used by the remote client (or instance owner) to connect to an integrity-verified cloud instance. Recall, the goal of the protocol is to connect the relying party to the instance if it (and the platform it depends on) satisfies the specified criteria. To do this, the client establishes trust in each layer of the verification framework, which will ultimately justify the instance's integrity.

In the first step, the client starts monitoring the CV's configuration. If the CV is trustworthy and its cloud criteria is at least as strict as the client's criteria, the client, in turn, provides its own integrity criteria and requests a connection to an instance. The CV then passes the client's criteria to the IVP in the node controller hosting the instance. If the IVP finds the monitored instance satisfies the client's criteria, it generates a signed *instance certificate* stating $IVP \xrightarrow{C} Instance$ and sends it back to the CV. Next, it permits the client to connect to the instance through the proxy. The CV then sends the instance certificate and a *node certificate* stating $CV \xrightarrow{Cloud} IVP$ vouching for the IVP satisfies the cloud criteria *Cloud*.

The client now has everything it needs to verify that the instance is trustworthy. First, the client can trust the CV because it is actively monitoring its configuration and its identity is certified by a public privacy CA (PCA). Next, because the client accepts the cloud's criteria, Invariant 1 states the client transitively trusts that all active nodes in the cloud satisfy *Cloud*. Thus, the client believes the node is trustworthy per the node certificate. Finally, the client believes the instance satisfies its criteria per the instance

certificate.

The last detail is how the client discovers that a violation has occurred along the chain of trust. In Figure 5.4, we can see there are three locations where the chain can be broken. The chain is rooted in the client's belief that the CV is trustworthy. If the client detects that the CV's configuration has violated its criteria, the chain is broke and the instance is no longer trusted. Next, the IVP and its node may reboot into a malicious system. The CV can detect this and deny the cloud node for rejoining. This effectively disconnects the node and prevents the client from contacting the instance. Finally, the IVP may detect a violation on the instance and will disconnect the client from the instance at the node.

5.3 Cloud Verification Architecture

We now describe in detail the architecture for our cloud verification framework. We will first explain how the CV is designed, its protocol for managing platform integrity, and its position within the cloud itself. Next, we detail how the IVP is used within the node controller and how it performs remediation for violations by the monitored instances.

5.3.1 Cloud Verifier

The cloud verifier is a public facing service within the cloud that acts as both the primary point of contact for external verifiers and is responsible for enforcing the cloud administrator's integrity criteria over the cloud platform. The CV is designed as a standalone component that is not dependent on any service within the cloud. For this reason, it should run on a separate system with a minimal TCB to simplify verification and reduce its potential for compromise. When deployed, it should function as a static service without interaction by the cloud administrator to ensure its independence from potentially hostile insiders. As the point of contact for external clients, the CV is publicly accessible over the internet through a limited interface. Its identity is certified through a public PCA to enable clients to easily obtain its credentials. While this is an extra step, it need only be done for a single CV within the CV's administrative domain. The CV also has administrative control over all components within the cloud and can disable them (e.g., turn the off) at will. For now, we consider a single CV scenario, but intend to explore how to handle larger deployments in the future. As the CV is responsible for numerous systems, we envision the CV can be replicated over each cluster of systems (e.g., cloud availability zones) and its protocols extended hierarchically for scalability.

Node $\rightarrow$ CV :	'JOIN', K_{Node}	(1)
CV $\rightarrow$ Node :	N	(2)
Node $\rightarrow$ CV :	ATT_N, ROTI	(3)
CV $\rightarrow$ Node :	'OK'|'DENY'	(4)

Figure 5.5. Join Protocol

Setup. The CV enforces the cloud administrator's criteria over the cloud platform. Before this is done, the cloud must be configured to make use of the CV. First, every component in the cloud is installed from a master disk image over the network by the cloud administrators. They are designed to block all connections on the administrative network except through an authenticated connection such as a TLS channel. Only identity keys certified by the CV are valid and all other connections are denied. The list of valid identity keys are stored in the cloud database, which we assume to be protected in a similar fashion as the CV. Each component regularly checks the cloud database (e.g., every 30 seconds) for newly signed keys to add to their local cache and revoked keys to remove. Since the CV disables systems when they are in violation of the cloud criteria, immediate dissemination of a key revocation is not required. We simple want to have invalid keys cleaned up within a short time such as a boot cycle to prevent their reuse. Moreover, each key is signed as $\{K_{Node}||MC||\text{Cloud}\}_{AIK_{CV}^{-1}}$, which states $CV \xrightarrow{\text{Cloud}}$ Node as signed by the CV's TPM at monotonic counter MC. The MC is used to ensure that all keys signed before the most recent boot of the CV are invalid.

Join Protocol. To participate in the cloud, each component must perform the join protocol in Figure 5.5 to prove to the CV that it satisfies the cloud criteria and thus have its identity key signed and disseminated via the cloud database. First, the joining component (we will refer to it as a node for now), generates a fresh identity key at boot. This key is derived from then node's TPM EK similar to an AIK. The node then (1) requests to join the cloud by sending K_{Node}, the public portion of the identity key, to the CV. The CV then replies (2) with a nonce N to request an attestation. The node responds (3) with an attestation ATT_N containing a measurement of the filesystem and boot time and the ROTI proof. If the CV finds the node satisfies the cloud criteria, it responds with (5) 'OK' and sends the signature $\{K_{Node}||MC||\text{Cloud}\}_{AIK_{CV}^{-1}}$ to the cloud database. Otherwise, it returns 'DENY'.

At the conclusion of the protocol, if the node has been admitted, the CV continues to monitor the node for reboots. It does this by polling the node for response every few seconds. The window for response should be less than the time it takes for a machine

to reboot into a malicious OS that can respond. Alternatively, an attestation could be requested that includes the MC of the node to check for a reboot. If the demand for the TPM is low, then this option is preferable, as it does not give a speedy attacker a chance trick the CV.

5.3.2 Integrity Verification Proxy

Each node controller is configured with an IVP to monitor the configuration of its hosted instances and compare that configuration to client requirements to determine if remediation is necessary. The IVP is a modular service that connects to the available measurement framework to collect the various configuration changing events. Our design uses the IVP to monitor the instances on behalf of remote clients. When the node is assigned an instance to start, its first launches an IVP monitor process. This process is in charge of collecting changes to the instance's configuration, verifying client criteria, and performing remediation. Before the instance is launched, the monitor instantiates each of its *measurement modules* to start collecting configuration events, which we will explain next. Next, when a client requests to connect to the instance, its criteria will be sent to the monitor process. The monitor registers the criteria with each module, which compare the criteria's invariants to the collect configuration. If all modules find the criteria are satisfied, the identity certificate is generated and sent to the CV. The monitor then opens a specific port that represents the satisfied criteria. The client can connect through this port to use the service on the instance it requested.

Modules. The measurement modules are either static or dynamic. Static modules collect information generated at a specific point and receive no other updates. Dynamic modules will continue to collect events for the duration of the instance's execution. Examples of a static module would be measurements of the instance's kernel, boot options, and disk image. Dynamic modules use interfaces like VMI and netfilter hooks in the Linux kernel to gather events like writers to memory and network flows respectively. When instantiated, the modules perform setup operations like registering callbacks with the interfaces and then start collecting those events.

When a criteria is register with the module, the individual requirements are compared to the collected events. If they are not satisfied, the monitor is notified. At runtime, dynamic modules will continue to receive new events. When this happens, registered criteria are rechecked to see if the new event causes a violation. If this is the case, the monitor is again notified so that remediation can be initiated.

Remediation. The primary form of remediation is to disconnect the client from the violating instance. This is simply done by closing the port assigned to represent the criteria that was violated. If the client has greater authority, additional actions can be performed. For instance owners like a service provider, the criteria can contain additional instructs that will automatically be invoked by the monitor. An example would be to shut down the instance or restart it. More complicated operations could be performed like rolling back an attached storage volume to a previous check point. One extreme example is for the cloud administrator to set a global cloud policy for instances. A requirement for no known botnet traffic to flow from an instance could be enforced by shutting down violating instances and suspending the instance owner's account.

5.3.3 Supporting Migration

Cloud platforms use migration to balance resource utilization among the cloud's nodes. When a node is overloaded, clouds migrate an instance to an underutilized node. Most commodity hypervisors support migration in either live or paused migration. In paused migration, the hypervisor suspends the instance, saves its current memory to disk, and sends the memory to the destination node. The destination node then loads the disk image from the image store, loads the memory dump into the instance's memory region, and resumes execution. The cloud must then handle the network translation to ensure traffic is buffered and redirected during migration. Live migration is similar except that the transfer of memory is done without pausing the instance. Instead, copy on write memory is used to transfer the unchanged memory until a brief pause occurs to send the remaining dirty pages over to the destination node.

Migration introduces several challenges for the CV design. First, moving an instance to another node would invalidate the instance certificate because the VM is no longer running on the node that certified it. Second, the client must know that the instance has moved to a trustworthy node. Finally, the instance must be provably bound to the new node. While we have not implemented a complete migration solution, we instead present sketches of two different approaches that could be deployed in a cloud.

Migration Reconnect. Since we make no assumption about how the instance is configured, the CV cannot rely on the instance to aid in migration. As a result, the CV must manage reestablishing trust in the instance after the migration on behalf of the client. In the *Migration Reconnect* protocol, the CV 1) reissues the connection request on behalf of the client after migration, 2) notifies the client of the migration, and 3) aids the client in reestablishing the connection with the instance. To the client, this will appear

similar to the initial connection protocol. We now give a more detailed breakdown of this approach.

When the cloud decides to move an instance, the CV only permits a destination node to be selected from the pool of verified (joined) cloud nodes. This is enforced through the issuing of node certificates that are required to communicate in the cloud administrative network. This automatically ensures the node certificate is valid for the new node. While the migration is occurring, the source IVP also sends the instance's boot-time key pair to the new node and all accumulated state about the instance. When the instance finishes migrating, the CV reissues the client's connection request, which the destination node's IVP checks. If the instance still satisfies client's criteria, a new instance node is generated by the destination node. Before the destination resumes the migrated instance, the source node's IVP is instructed to invalidate all instance certificates for the instance and disconnects all client's connected through the source node's IVP verified ports. To inform the client that a migration is occurring instead of criteria violation, the client is sent the new instance certificate with the newly assigned port number by the CV. From there, the client can reconnect to the instance.

Eliminated Interruption. One limitation of the Migration Reconnect approach is the requirement for the client to disconnect and reconnect. This could cause unacceptable delays for the client and the instance. This is caused by the overly rigid binding between the instance and the node through the instance certificate. Recent work in supporting VM migration in clouds using vTPMs [140] can be leveraged to eliminate this interruption. In this work, the binding between the vTPM's keys and the physical TPM can be migrated with the VM to the new physical host. By verifying the integrity of the new host, the binding is transitively trusted. The basic idea is to use a level of indirection to certify the integrity of the key migration process so that a remote verifier does not need to check the binding to it personally. The client must only trust that the hand-off had occurred between two trustworthy nodes. Since our CV already ensures only verified destination nodes will receive the migrated instance, the client knows that the instance will continue to run on trustworthy nodes even after migration. Thus, verifying a new instance certificate is tautological. Therefore, we can change the protocol so that the client only verifies an initial instance certificate that implies a correcting instance certificate after migration.

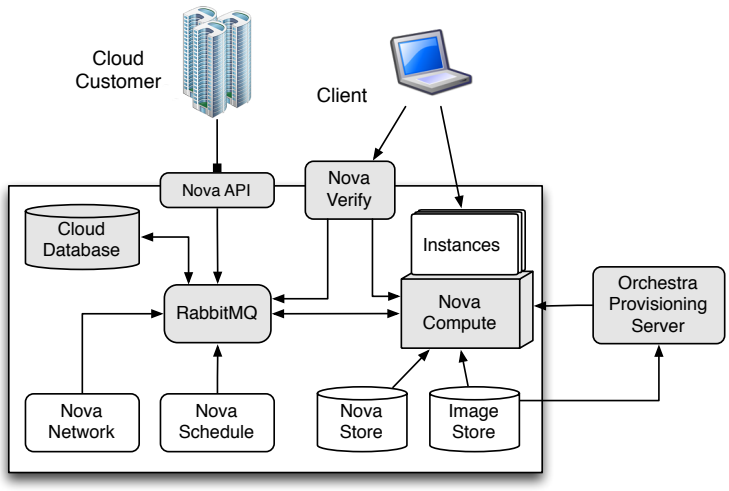

Figure 5.6. Changes to OpenStack

5.4 OpenStack Integration

We now describe our implementation of a proof-of-concept cloud verifier framework on the open source IaaS cloud platform, OpenStack [137]. First, we detail the CV and how it is integrated into the cloud architecture. This includes the installation of an instance hosting node and how the CV verifies it. Next, we illustrate our design of the changes to the IVP we had to make for the OpenStack architecture that differs from its original design. In addition, we include implementation of several measurement modules that expand the criteria options available to clients beyond those presented in Chapter 4.

5.4.1 Implementation Overview

Figure 5.6 illustrates a simplified view of the basic OpenStack infrastructure. OpenStack is composed of various 'projects' that add additional services, but we will focus on the `nova` project for our implementation because it provides the necessary components for hosting instances. The components are largely the same as those presented in the general IaaS architecture in Section 5.2, but some names have been replaced with their `nova` equivalent. The shaded components represent the changes we have made to implement the CV framework. First, the `nova-api` interface has been extended with a new commands to specify client criteria. Second, the cloud database has been updated to store node identity keys. Third, the `nova-compute` component (compute node) that hosts cloud instances was extended with our IVP implementation. We also modified the RabbitMQ message queue service to deny all messages except over an SSL channel

authenticated by a CV signed certificate. This implements our requirement that only verified components can communicate through the RabbitMQ and thus participate in the cloud. Finally, the `nova-verify` service has been added as a standalone component to implement the CV. `nova-verify` is another public facing service in addition to the `nova-api` service. Overall, the additional code added to OpenStack was roughly 2,600 SLOC in Python. Additional code for implementing modules and measurement interfaces was under 1,000 SLOC of Python.

5.4.2 Verifying Compute Nodes

The `nova-verify` service is in charge of admitting `nova-compute` systems into the cloud. This means signing their identity key and storing it in the message queue to permit the compute node to use the message queue. Since cloud operations are sent through RabbitMQ, compute nodes must follow the join protocol to do anything useful.

Registering to the cloud. When a compute node boots, it parses the locally saved configuration file to locate the RabbitMQ server and the cloud database. Normally, it then registers itself as an available compute service in the database and subscribes and publishes messages in the message queue. To mediate the registration process, we blocked all servers except those that connect through an SSL `stunnel` proxy service [141]. `stunnel` provides SSL functionality without requiring modifications to application source code. We then provided a port that redirects all non-stunnel traffic to the `nova-verify` service to begin the join protocol.

Registration. To verify the compute nodes, the `nova-verify` service listens for verification requests from compute nodes over an open port within the administrative network and places the node certificate the directory of valid certificates on the message queue if it satisfies the cloud criteria. After the compute node registers itself to the cloud, the CV monitors compute node for reboots. We implement this check by having the compute nodes periodically query the database per their normal behavior. This is done to look for pending requests they should service. This automatically updates a last accessed time for the node. However, if the status of a compute node is not updated for a certain amount[2], the CV will notice the missing update and remove the node's identity certificate from the message queue. This, in turn, invalidates the `stunnel` connection, which essentially isolates the compute node from the cloud.

[2]In our case since we believe it is unlikely for a server to fully reboot into a malicious state from a benign state in 30 seconds.

5.4.3 Building a Verifiable Compute Node

We designed the compute node to use the ROTI installation approach to simplify verification and enable the CV to trust the compute node to maintain its runtime integrity. However, large datacenters like clouds typically design a single disk image and push it out of the internal network, instead of using an optical disk in the ROTI design. Thus, we employ the netROTI approach [92] verify installation over a potentially malicious network. We give a brief description of the netROTI in our implementation, but refer the reader to Chapter 3 for further details..

We used the open source Orchestra Provisioning Server Orchestra to conveniently manage PXE-boot installations of the compute node master disk image over the network (see Figure 5.6). The disk image is modified to perform the netROTI operations that generate the ROTI proof. This includes measurements of the installed file system, the Orchestra installer, and the disk image used.

We configured the disk image to maintain runtime integrity in several ways. First, we used Advanced multi-layered Unification FileSystem (AUFS) [142] to overlays filesystem branches into a single virtual file system. This allows us to set the installed filesystem to be read only while layering a temporary file system on top. In this way, modifications to the installed filesystem will be discarded during reboot and thus the compute node can maintain an identical state across boot cycles.

However, this design requires that the runtime modifications to the temporary filesystem not to exceed its allocated size. By our observation, most modifications are relatively small (smaller than 10MB). The only exception is the OpenStack disk image cache created during instance hosting. In order to avoid copying the disk image from the image server every time, OpenStack maintains a disk image cache locally. This may cause 10GB or more of new storage per instance. Our solution is to overlay another rewritable disk partition over the cache location so that modifications to this directory will be preserved. Since we are effectively partitioning this cache from the critical compute node filesystem, we can just ignore this directory during the measuring process at boot time.

5.4.4 Client Connection Flow

We now describe how the client establishes a connection to the cloud instance. This involves verifying the various CV components and registering the client's criteria. In the next section, we provide an example of possible criteria and the modules that use them.

We designed our protocol to allow clients to connect to an instance's IVP without

requiring the client to know ahead of time whether it is hosted in a particular cloud. The only assumption we make of the client is that it knows an IVP exists for the service it wishes to use and that it has access to the public PCAs that the cloud uses. As an important note, our prototype is built under OpenStack FlatDHCP network mode, which places all instances on a private network, whose gateway is the network controller (nova-network service). Instances may be assigned public IP addresses and the network controller forwards their traffic using NAT.

Step 0: Connect to the instance's verification service. Consider a client that wishes to connect an instance with public IP $IP_{instance}$ and a service on port $P_{service}$. In addition, the client has a criteria C_{client} which should be satisfied for the duration of the connection. The client starts the protocol by first connecting to the instance's IVP port, say P_{verify}. The network controller then redirects the request to the CV, which responds with the IP address of the `nova-verify service`. This tells the client that the instance is hosted on a cloud and that the CV is available.

Step 1: Verifying cloud verifier. Like a compute node, the CV is deployed by the Orchestra Provisioning Server using a netROTI installer. The client verifies the installation and the current boot cycle before proceeding to use the cloud's services. If the verification succeeds, the client establishes a persistent unauthenticated SSL heartbeat connection to the CV. As long as the CV responds in a timely fashion, the client can trust that the CV has not rebooted.

Step 2: Request connection to instance. The client then request to connect to an instance through a new `nova-api` command `euca-register-criteria`. The arguments for the command are: 1) the public IP of the intended instance, 2) a port on the instance, and 3) the criteria the client wishes to have enforced. We added a function to the `nova-api` service to forward the request through the message queue to the compute node hosting the instance with the specified IP. We also added a special handler in the nova-compute service to respond to the criteria registration RPC call and interact with IVP.

Step 3: Register client criteria. When the IVP receives the client's request, it first registers the criteria with each measurement module. If the modules report that the criteria are satisfied, the IVP unblocks a randomly assigned port $P_{criteria}$ for $P_{instance}$ in the compute node (the VM host) firewall. This port then redirects requests to $P_{service}$ on the instance. The client accesses this port instead of the

original one to access the service with the protection of IVP. For example, a client is assigned a port 9000 for accessing a web service hosted by an instance, the compute node will add a firewall rule to redirect all traffic from port 9000 to port 80 using iptables. If a violation is later detected by IVP, this rule is deleted and all traffic to port 9000 will be denied. In this way, client's connection is bound to the criteria she specified. Note that since we used destination ports rather than source IP to represent different connections therefore no matter the client is behind a NAT firewall or proxy, its connection will always be under control of IVP. Also, considering the limited ports on compute node, we aggregate the connections based on the $\langle C_{criteria}, P_{criteria} \rangle$ tuple. If more than one client accesses the same service with the same criteria, they will be represented as the same connection.

Step 4: Connect to the service. At the completion of the Step 3, the client is returned a node certificate and an instance certificate. The first is the public key of the compute node signed by the CV. The instance certificate is the IVP signed tuple of the instance's public key generated by the IVP and the details of the connection request. The client is now free to connect over $IP_{instance}$:$P_{criteria}$ to use the requested service. It is our intention that the client uses the instance certificate to authenticate the instance and establish a TLS connection with the service. This would require the instance to be configured to use the key pair placed in the instance at boot time by the IVP to identify the system. While this provides the necessary protections at the transport level, we do not mandate that the instances use this certificate.

5.4.5 Integrity Verification Proxy

We implemented the IVP as a separate daemon spawned by the `nove-manage` service in the compute node. When an instance is requested, a new IVP object is created to monitor it. This object is called at various hooks placed throughout the instance management functions in n̂ova-manage. For example, the create instance command triggers the IVP to instantiate and initialize all of its measurement modules. The IVP is also called when a client requests to connect to the instance through the `new nova-api` command. When the IVP is spawned, it obtains handles to the available measurement interfaces. These handles are passed to the modules, which select the interface they were intended to use. If the interface is unavailable (e.g., was not installed), the module is simply destroyed. The modules hook into these interfaces and proceed to collect information from the

instance. Currently, these events are stored in the module objects, but we plan to build a more scalable solution to store information for longer running instances.

5.4.5.1 Measurement Interfaces

We used a combination of preexisting and new measurement interfaces.

- *Disk Image*: This interface provides modules with direct access to the instance's disk image before it has been started. It mounts the disk image and provides a simple interface accessing files rooted in the disk mount. This is useful for measuring the disk itself and other critical files like the SELinux policy, kernel, and its modules.
- *VMinfo*: The VMinfo interface exposes properties of the instance profile such as number of vcpus, memory size, boot parameters, network devices, etc. This information can also be used by the client to ensure the instances are given the correct resources and are not being cheated. The interface uses the libvirt Python API directly to query the VM instance object. While VMinfo can technically be polled to check for changes in the VM's properties, we currently use the interface only at instance start time. Further investigation is required to determine proper hook placement to catch updates to instance properties.
- *Policy Server*: We designed a policy measurement interface as a service for analyzing SELinux policies on an instance. The server extracts the information flow policy from a binary SELinux file into a network graph representation. Once loaded, modules can submit network flow queries to the server, which will test it against the instance's policy. A policy module can perform various queries on the policy related to its information flow graph. We have implemented several queries that we elaborate upon in Section 5.5.1.
- *Netflow:* This interface hooks into the iptables conntrack interface via `ulogd` to obtain the network flows connecting to the monitored instance. In other words, Netflow modules can track the IPs of remote systems that connect to the instance. The flows are stored in a local `sqlite3` flat file database. Client criteria can use this information to detect if unwanted hosts are using the instance, such as known botnet C&C networks. In addition, the client can use these flows to check that the firewall rules specified to the cloud are actually enforced.
- *VMI:* This proof-of-concept VMI interface lets modules check for writes to memory locations in the instance. When an instance spawns, we attach a gdb thread to it and set the hardware debug registers to watch specific kernel data structures defined in the kernel symbol table. Each module can set a hardware watchpoint

(up the register limit) and when a write is performed, the module is notified of the change. Further, the module can perform additional checks on memory when the instance is trapped.

5.4.5.2 Module Design

Each measurement module extends an abstract python base class to expose five methods. When the IVP instantiates the module, it first calls the `preinit` method to setup the internal module structures and provides a callback function to alert the IVP of any client criteria violations. The `init` method is then passed a dictionary of handles to the available measurement interfaces. The module selects the interface(s) it needs and registers any callbacks on the interfaces (e.g., memory watchpoints for the VMI interface). The IVP can then call the `register` method to pass a client's criteria to the module, which responds with `True` if the current instance state satisfies the criteria and `False` otherwise. This method also registers the criteria within the module so that it can check for violations with each update via the measurement interface. The module uses the IVP callback to alert the IVP of such violations. The `deregister` method removes a client's criteria and the `cleanup` method is called when the instance is shutdown to remove all client criteria.

5.5 Evaluation

We evaluated the functionality of our prototype by developing several measurement modules to demonstrate the types of criteria client's can specify to be verified. We also studied the performance impact of our implementation on two application-specific cloud instances and profiled each connection protocol to determine overall delay to use the CV for establishing a connection the instance. We measured the former to determine if the CV framework imposes an undue overhead on cloud applications. The latter is to discover how much our approach would inconvenience the user in using the cloud service.

Our experimental setup was an OpenStack version 2011.3-nova-milestone cloud installed on three identical Dell M610 blades. These machines have two quad core Xeon processors with 8GB of RAM and two 1Gb network cards for the private and public network. One of them serves as the compute node which hosts virtual machines. Another one serves as the Cloud Verifier as well as other necessary controlling components of the cloud like scheduler, API server and network controller. The last blade is used to simulate a normal client of the cloud and performs the benchmark program. Each

system used Ubuntu-11.10 (x86_64) with a Linux 3.0 kernel for hosts and Ubuntu-11.10 (x86_64) with kernel 3.0.1 for virtual machines.

5.5.1 Measurement Functionality

In Section 5.4.5, we presented five different measurement interfaces that the IVP can use to monitor the configuration of the cloud instance. To evaluate the usefulness of these interfaces, we built several modules to collect events from those interfaces and demonstrate the integrity invariants that can be tested.

5.5.1.1 File Hash

The file hash module uses the disk image interface, which has direct access to the instance's disk image contents, to take SHA1 hashes critical files. We used the module to collect hashes of the initial ram disk, kernel, boot loader files, kernel modules, library files, and SELinux policy files. The module accepts criteria that are black / white lists of hashes for each category of file. When the client criteria is registered, the module performs a set membership check and returns false if the invariant is violated.

5.5.1.2 Resource Verifier

We designed a module to enable clients to check that the resources given to the instance they request are actually provided. The module extracts from the hypervisor (KVM in our case) information such as the number of vCPUs, amount of RAM, kernel parameters, etc. Criteria passed to this module include minimum and maximum numbers of these values and the presence of additional consoles that an admin could use to access the instance.

5.5.1.3 Information Flow Checker

For complex configurations like an SELinux policy, it is unlikely that a remote verifier can judge the trustworthiness of such files based solely on a hash. Instead, we can use the Policy Server interface to check for unsafe information flows within the SELinux enforced system. This module uses the Disk Image interface to retrieve the binary SELinux policy module and sends it to the Policy Server interface at instance start time. When a client registers its criteria, the module issues the queries on behalf of the client to the server and checks if any of them fail.

We have implemented four basic queries so far: 1) presence of a particular set of domain labels, 2) presence of a flow between one set of labels and another, 3) presence of known unprivileged process labels (types `user_t, staff_t, guest_t`), and presence of unconfined administrators (types `sysadm_t, unconfined_t`). Unconfined administrators could perform a number of privileged tasks, such as switching off enforcement of the access control policy altogether. Clients could thus refuse to connect to servers that have these administrative processes.

We also implemented a more complex query that is composed of the previous queries to check for arbitrary disk access by process labels outside of a TCB set. Raw disk devices (e.g., `/dev/sda, /dev/lvm, /dev/md-*`, all labeled `fixed_disk_device_t` by SELinux) may be exposed to process types outside the TCB of the instance, which may not satisfy certain clients requirements. This query checks that processes that writes to raw disk devices are a subset of the system TCB. We calculate the TCB set for an SELinux policy using techniques presented by Vijayakumar et al [143].

5.5.1.4 LIM Extraction

Similar to the module in Chapter 4, we implemented a module to monitor the internal LIM measurement vector in the instance's kernel. When `init` is invoked, the `ima_measurements` symbol is registered as a new hardware watchpoint. This points to the list head of LIM records stored in the kernel every time a new binary is loaded. Once the instance launches a new executable, the interface calls the module to traverse the list and record the newly loaded executable's hash. Once the instance is resumed, the module then compares the new measurement to the criteria currently registered and invokes the IVP callback to alert it of a violation.

5.5.2 Application Benchmarks

Next, we evaluated the impact of the CV framework on application-level benchmarks. We select two of the most popular applications AMIs from the AWS marketplace, the Apache and Nginx web servers. Since we did not have access to the Amazon AMI, we installed the applications into a clean install of Ubuntu 11.10 to create our VM disk image.

We stressed each instance using Apache Benchmark (`ab`) to simulate 100,000 requests and varied the number of concurrent clients to represent different loads. We also tested the performance of simultaneous runs against two instances hosted on the same node. In

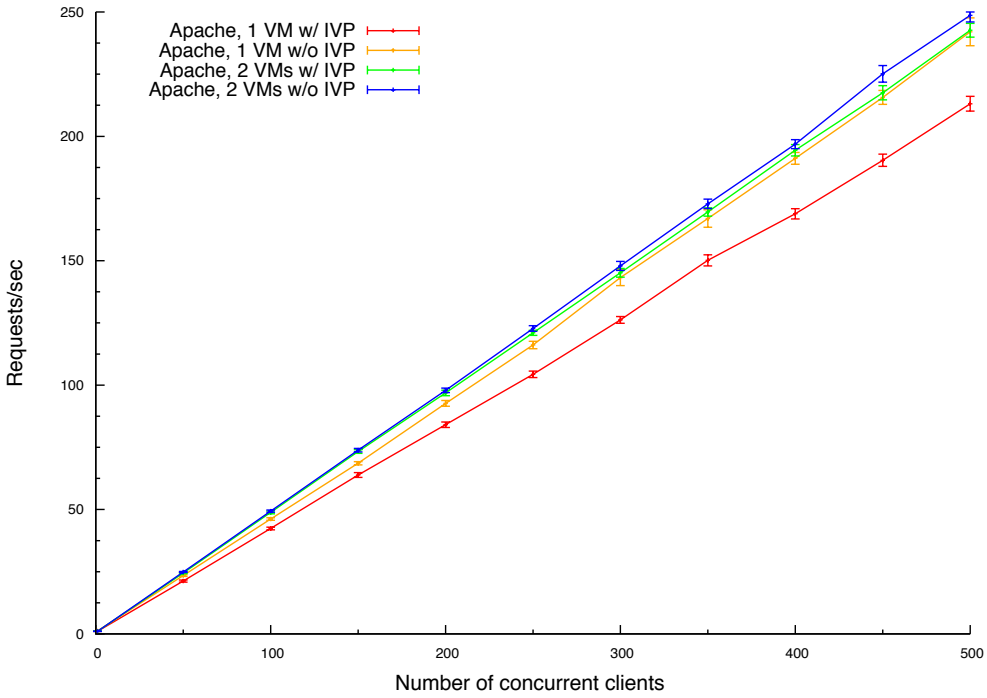

Figure 5.7. Apache requests / sec with and without IVP monitoring. Each point is the mean of 30 runs with error bars for a 95% confidence interval.

each scenario, we ran the tests with and without the verification framework for 30 runs per setting.

Figures 5.7 and 5.8 display our experimental results in terms of requests per second. Our tests show that instances on the monitored cloud incur only minor overhead, which grows with additional load. Under the single VM experiment, the IVP monitored instance begins to incur a slowdown with 200 concurrent connections for both webservers. For the two VM scenario however, the performance impact of the monitored instance is smaller and does not begin to significantly diverge from the unmonitored instance until 400 concurrent connections. As our experiments in Chapter 4 revealed, the main cause of passive monitor overhead was due to the GDB's use of the ptrace interface for monitoring VM memory. Thus, we believe a more tailored introspection interface would eliminate much of this delay.

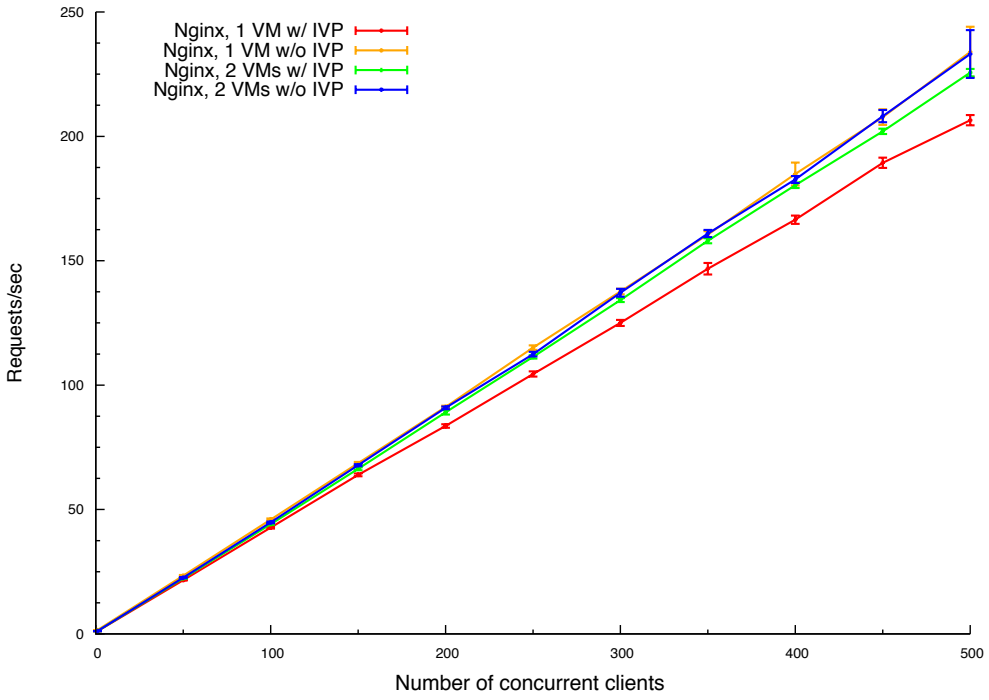

Figure 5.8. Nginx requests / sec with and without IVP monitoring. Each point is the mean of 30 runs with error bars for a 95% confidence interval.

Operation	Mean (sec)
CV Verification Protocol	1.09
Node Join Protocol	1.39
Client Criteria Registration	0.54

Table 5.1. Time to perform various CV operations. Times are averages of 30 runs. Standard error was negligible for each operation.

5.5.3 CV Operations

Table 5.1 shows the time to perform three major operations in our CV framework. The first is the time due to verifying the `nova-verify` service, which the client must do before using the cloud. The second is the compute node's cloud join protocol. In both cases, the majority of the overhead comes from the TPM quote operation (0.84 seconds), which is a common and typically required operation in verifying any remote system. The additional overhead for the CV verification operation is on par with an SSL handshake operation

in our cloud (on average 116 ms with negligible variance). We envision techniques like Asynchronous Attestation [91] can be used to reduce this delay since it will be a major bottleneck for potentially thousands of clients connecting to the cloud. For the node join protocol, this is a significantly shorter time than the boot process the node must go through and is only a one time cost per boot. Finally, the client registration operation has a overhead comparable to the time to issue a request with the API server. About 60% of the time is spent propagating the client's criteria through the cloud while the remaining time is spent at the node to verify the requirements and signed the instance certificate. It is worth noting that the total time will depend on the modules that must be check for the client's criteria. This represents the minimum time only.

5.6 Related Work

We now present a summary of work related to verifying cloud computing platforms, why they are insufficient for meeting our goals, and how the CV framework overcomes their shortcomings.

Resource Accountability. As an economic model, cloud computing is focused on delivering various resources as a commodity. Thus, there is a real desire by by the client to ensure they are not shortchanged on the purchased resources and by the cloud provider to detect theft of service. To that end, various efforts have been put forward to provide *resource accountability*, which tracks the resources given to each client's instance or account [144]. Efforts like consumer-centric resource accounting models [145] pushes for cloud designs that reveal more about the resource allocation process and let customers query that data to confirm correct provisioning. Other consumer products like VMware's vCenter Chargeback [146] and Amazon's CloudWatch [147] give customers a full accounting of their costs per requested resource, but offer few guarantees that the service is configured correctly. Many of these approaches are limited by their lack of verifiability. Sekar et al., proposed a hardware driven solution that is supported by tamper-proof components like the CPU and TPM that could, if designed, vastly improve performance and reliability of results [148]. However, these approaches are only concerned with the billable elements of the cloud and cannot detect on the functionality of the hosted instances.

Application Integrity. Another thrust of research has been to verify the integrity applications by inspecting their outputs. Early research in N-Variant systems [149] detected attacks by diversifying a system's implementation across multiple variants. Thus,

attacks that target one implementation are unlikely successfully compromise all variants. A verifier could then detect an attack by comparing the results of all variants and finding an outlier. This is similar to the concept of using redundancy to detect faults. More recent research aimed to verify workflow integrity in specialized cloud environments like IBM's System S stream processing cloud[150, 151, 152]. Here, each cloud node is a single-purpose processing unit that performs a simple task over a potentially infinite flow of data. These units are linked together in various combinations to render new information. These approaches use application level attestation protocols to enable detection of where illegal modification have occurred in the data flow. Other techniques like PeerReview [73], Accountable VMs [74], and Trinc [76] leverage hardware-based attestation to verify adherence to known distributed application protocols and detect Byzantine faults. Jana and Shmatikov designed a cloud-hosted web application framework that verifies incorrect execution based simple database accesses [134]. However, all of these approaches are limited to application protocol verification and cannot discover violations that are outside of the protocol's scope or malicious behavior like data exfiltration.

Storage Security. Many companies use cloud platforms to replace storage and hosting services, but they are concerned that such data will be lost, stolen, or modified. Cachin and Schunter[153] give a survey of threats to data secrecy on clouds and promote their TCloud initiative for more reliable cloud platforms [154, 155, 156]. In addition to encryption protocols that protect the data at rest, techniques like Proofs of Retrievability detect if the data has been lost [157]. Other research by Wei et al. aims to protect the instance disk image in the cloud by alerting the user to unauthorized modifications [158]. The Silverline project [130] uses a set of analysis tools to identify *functionally encryptable data*, data that can be processed as cipher text, to improve data confidentiality in an untrusted cloud platform. These data security techniques can certainly complement any cloud-hosted application, but they provide little guarantees for data that is used in plaintext. On the other hand, the CV framework can help detect misuse of data while it is processed in the clear.

Virtual Machine Security. Research in hardening hypervisors and protecting instances has become popular with the rise of cloud platforms. Techniques like Secure In-VM Monitoring sim, VICI [70], and SecVisor citesecvisor, and many others [68, 95, 96, 69, 104, 159, 160] introduce VMI interfaces for monitoring VM integrity. However, these approaches are designed to monitor specific integrity requirements at install-time and are not flexible to support various customer needs. Moreover, the power of these tools are unmitigated in their deployment and give administrators far more access to

sensitive VM data than should be permitted.

Other approaches have focused on improving the effectiveness of introspection to close the semantic-gap. KoP [107], Syscall interrupt tracing [161], and Virtuoso [162] enable administrators to reverse engineer instances to identify critical data structures that should be monitored. Gibraltar also helps this goal by identifying the critical invariants in the kernel that should be maintained. The CV framework can benefit from the inclusion of these VMI tools that targets more integrity-relevant memory locations.

Another area of research has been to harden the hypervisor from both co-resident VMs and insiders. Approaches like NOVA [101] and NoHype [163] minimize the VMM's attack surface by essentially eliminating all but a small resource management kernel. Other techniques like formally assured L4 microkernels [6] and privileged domain separation [102, 103] again limit the impact a compromised hypervisor can have on hosted instances. Finally, low-level monitors like LKIM [50], HyperSentry [100], and DeepSafe [164] use the protected System Management Mode memory region to monitor and enforce hypervisor integrity at runtime. The CV requires a strong guarantee that the runtime integrity of the cloud components are maintained. It can uses these approaches to ensure these claims to the remote clients.

5.7 Conclusion

In this chapter, we presented the Cloud Verifier, a framework for verifying remote client integrity requirements over cloud instances and the platform that hosts them. The CV is an independent and verifiable service in the cloud that enforces a cloud administrator's integrity criteria over the cloud components. Remote verifiers build transitive trust through the CV and the cloud node's integrity verification proxies to verify cloud instances. We constructed a proof of concept CV for the open source OpenStack cloud platform and demonstrated several measurement frameworks that the CV can leverage to verify client requirements. We further evaluated the framework on three of the most popular cloud instances used in Amazon's EC2 cloud. Our experiments show negligible overhead is incurred by the instances due to monitoring and that the connection protocols the client and nodes use impose less than a two-second delay at setup time. By adopting the CV framework into an IaaS cloud platform, cloud providers both private and public can give clients and customers the comfort of verifying the integrity of their critical business processes in the cloud.

Chapter 6

Towards a Trustworthy Future

As virtualization technologies continue to improve and the costs of higher power computing devices decreases, the already crowded market of online services will continue to grow. This trend almost certainly guarantees a future where companies and individuals will no longer have the concept of a personally administered desktop, laptop, or even mobile phone. Instead, our data, applications, and even our identities will be fused across an endless sea of server farms around the globe. As this highly connected world comes closer to realization, so too will the risks of losing all control of our sensitive information.

However, current solutions to securing our information are insufficient because the functional requirements these applications demand are already outpacing the ability of today's crypto systems. What we need is a way of controlling the release of our data and secrets based on the security and integrity we desire. Unfortunately, such control is limited at present to selecting from a list of predefined security settings with little assurance that they will be enforced.

The frameworks and tools presented in this dissertation lay the foundation for a practical approach to building more verifiable distributed systems that will provide the services of the future. Already, companies like Trusted Cloud [165] and government agencies [166] are beginning to adopt trusted computing technologies into cloud designs and techniques like the cloud verifier can ensure these platforms are accountable. Other mechanisms like the netROTI can be applied to provisioning services in general to detect compromises in data centers of all sizes.

6.1 Lessons Learned

Each chapter presented in this dissertation a methodology for augmenting different components of online services to enable better verification. These designs built on top of previous work including our own to demonstrate improvements real world systems. In doing so, we came across various challenges inherent to systems research. Throughout this dissertation, our solutions to these problems have shared a common thread that we summarize below.

6.1.1 Clear Trust Assumptions

Before we can even assess remote system integrity, we must understand what it is that we trust. Too often a solution is proposed in the literature that attempts to verify a narrow definition of system integrity without considering the reality of it deployment. For example, early TCG-style approaches like IMA [30] characterized system integrity based on a series of code hashes that were either black or white listed. However, this ignores the fact that such binaries could themselves be compromised. Defining rigorous integrity requirements should involve identifying the trusted components and assessing their weaknesses, why we are willing to trust them, and how can we detect when they have failed. In Chapter 4, we proposed using practical integrity models like CW-Lite [36], which simplifies the process of identifying the TCB for further scrutiny.

One challenge to this goal is a lack of transparency in public services like cloud platforms. Custom or proprietary code can contain bugs or misconfigurations that can lead to unintended data loss. While verification frameworks can detect these problems, the administrators may not wish to expose their code for fear of IP theft or greater security risks. One alternative would be an approach similar to server side verification proxies like the IVP and CV, which can test various client criteria against locally stored information that is not revealed outside of the administrative domain.

6.1.2 Enforcement

Verification has several drawbacks that become rapidly apparent when applied naively to complex systems. In addition to the difficulties of assessing arbitrary data, the shear numbers of possible events that can occur makes tracking the configuration of a remote system impractical. A general purpose OS without a strict MAC policy can potentially load any number of malicious processes that can compromise system integrity at will. Instead of permitting integrity-sensitive operations and later detecting whether they vi-

olated someone's integrity requirements, strong enforcement mechanisms can significant limit a system's *measurement surface*. In other words, designing a system to ensure a particular integrity policy gives remote verifiers *a priori* knowledge of that system's long-term state.

6.1.3 Performance

Performance is always critical in any high demand scenario and public utilities like clouds are no exception. Despite the compelling motivation of added security, the adoption of new technologies like trusted computing can be easily stymied by its effect on a service's utility. Thus, care must be taken to avoid placing slow components (like the TPM) on the critical path when designing these platforms for greater transparency. For example, leveraging the hierarchy of a distributed system's control backplane is a natural place to put verification mechanisms as the administrators can make use of them as well.

6.2 Future Work

This dissertation has presented the foundation of a verification framework for a particular computing platform. In this section, we discuss future avenues for this research both in terms of cloud computing platforms and verification tools in general.

6.2.1 Extending Verification Beyond IaaS Cloud

Cloud computing services have grown beyond the basic services of IaaS clouds. More powerful platforms like PaaS and SaaS cloud break many of the convenient assumptions we have leveraged in our design of the CV including a fairly minimal code base in the VM host and clear physical isolation of VMs on to a single machine. These more convoluted clouds offer greater convenience to the cloud customer because they take over even more administrative complexities like OS level configurations and resource provisioning. As a result, customer data and code may be replicated across numerous locations and executed on completely custom and proprietary code stacks.

6.2.2 Multilayer Verification

Future work should investigate approaches for verifying customer requirements at all levels of the cloud software and administration stack. This means establishing well defined integrity monitors at each level that are responsible for monitoring the integrity

of their layer and disseminating criteria up the stack to higher level monitors. One obvious challenge is translating high level criteria that a client may specify into the low level details of these layers. For example, a monitor that observes code executing in a Java VM must be able to take high level information flow requirements and detect when data from one source is leaked to a sink that violates the requirement. With a seamless translation between layers, it would no longer be necessary to push measurements from one layer to the next because they would be assessed at the layer where is it most relevant.

6.2.3 User Endpoints

In addition to verifying the service's integrity, the remote clients that provide inputs must also be accounted for. This is especially true for services that combine information from multiple sources like message boards, collaborative editors, and information aggregators. This could involve having the service mandate a minimum criteria that all clients must satisfy. Another option could be to associate the data with the integrity of the source and let users judge the trustworthiness of the information as the lowest common denominator among the sources.

6.2.4 Controlling Data Release

A major concern for users today is the general lack of control over their personal data once it is entrusted to these remote services. A ripe area for exploration would be to develop a mechanism akin to homomorphic encryption that only produces a derivative of an operation over that data if some integrity requirements are satisfied. Consider a photo sharing service that lets users specify the accounts that are permitted to look at the pictures. Normally, the user would have to trust the service to properly enforce this access control policy or encrypt the pictures and lose some functionality like rendering the pictures at different resolutions. Instead, the user could encrypt the data so that it can only be disseminated if the service was functioning in a trustworthy manner. This mechanism could enable users to ensure data security without having to maintain a persistent integrity monitor over the service or manage complicated encryption key protocols.

6.3 Concluding Remarks

Increasingly, our personal data is moving beyond our control into remotely administered domains. This new paradigm requires a change in how we manage our data and an acknowledgement of the fact we can no longer assume the services we use, even if we design them, are trustworthy. Trusted computing research has opened the door to techniques for inspecting these remote and even local services, but numerous challenges have limited and will continue to hinder its adoption. Despite this, the need to guarantee data and process integrity will continue to drive innovation in this field.

Bibliography

[1] Sony (2011), "Update on PlayStation Network and Qriocity," http://blog.us.playstation.com/2011/04/26/update-on-playstation-network-and-qriocity.

[2] Register, T. (2010), "Infamous Storm botnet rises from the grave," http://www.theregister.co.uk/2010/04/27/storm_botnet_returns.

[3] "Security-Enhanced Linux," http://www.nsa.gov/selinux.

[4] Corporation, N. D., "Tomoyo Linux," http://tomoyo.sourceforge.jp.

[5] Novell, "AppArmor Application Security for Linux," http://www.novell.com/linux/security/apparmor.

[6] Klein, G. et al. (2009) "seL4: Formal Verification of an OS Kernel," in *SOSP '09*.

[7] Myers, A. C., N. Nystrom, L. Zheng, and S. Zdancewic (2001) "Jif: Java + Information Flow," Software release. Located at http://www.cs.cornell.edu/jif.

[8] Google, "Signing Your Applications," http://developer.android.com/guide/publishing/app-signing.html.

[9] Provos, N., M. Friedl, and P. Honeyman (2003) "Preventing privilege escalation," in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, USENIX Association, Berkeley, CA, USA, pp. 16–16.
URL http://portal.acm.org/citation.cfm?id=1251353.1251369

[10] Garfinkel, T., B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh (2003) "Terra: A Virtual Machine-Based Platform for Trusted Computing," in *Proc. 19th ACM SOSP*.

[11] Watson, R. N. M., J. Anderson, B. Laurie, and K. Kennaway (2010) "Capsicum: practical capabilities for UNIX," in *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, USENIX Association, Berkeley, CA, USA, pp. 3–3.
URL http://portal.acm.org/citation.cfm?id=1929820.1929824

[12] Arbaugh, W. A., D. J. Farber, and J. M. Smith (1997) "A Secure and Reliable Bootstrap Architecture," in *Proc. IEEE SSP*.

[13] "Common Criteria for Information Technology Security Evaluation," `http://www.commoncriteriaportal.org/cc`.

[14] Dougherty, C. R., "Multiple DNS implementations vulnerable to cache poisoning," `http://www.kb.cert.org/vuls/id/800113`.

[15] Jackson, C., A. Barth, A. Bortz, W. Shao, and D. Boneh (2007) "Protecting Browsers from DNS Rebinding Attacks," in *In Proceedings of ACM CCS 07*.
URL `http:///crypto.stanford.edu/dns/dns-rebinding.pdf`

[16] Ellison, C. M. (1999) "The Nature of a Useable PKI," *Computer Networks*, **31**(8), pp. 823–830.

[17] Dierks, T. and C. Allen (1999), "The Transport Layer Security Protocol," RFC 2246.

[18] Kent, S. and K. Seo (2005), "Security Architecture for the Internet Protocol," RFC 4301 (Proposed Standard).
URL `http://www.ietf.org/rfc/rfc4301.txt`

[19] "IBM PCI Cryptographic Coprocessor," `http://www-03.ibm.com/security/cryptocards/pcicc/overview.shtml`.

[20] TCG (2005), "Trusted Platform Module," `https://www.trustedcomputinggroup.org/specs/TPM/`.

[21] Parno, B., J. M. McCune, and A. Perrig (2010) "Bootstrapping Trust in Commodity Computers," in *IEEE SP '10*.

[22] Smith, S. W. (2002) "Outbound Authentication for Programmable Secure Coprocessors," in *ESORICS*.

[23] Petroni, N. L., J. Timothy, F. Jesus, M. William, and A. Arbaugh (2004) "Copilot - A Coprocessor-based Kernel Runtime Integrity Monitor," in *In Proc. 13th USENIX Security Symposium*.

[24] Kennell, R. and L. H. Jamieson (2003) "Establishing the genuinity of remote computer systems," in *USENIX Security Symposium*.
URL `http://portal.acm.org/citation.cfm?id=1251353.1251374`

[25] Seshadri, A., M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla (2005) "Pioneer: Verifying Code Integrity And Enforcing Untampered Code Execution On Legacy Systems," in *Proceedings Of The 20th ACM SOSP*.

[26] "Trusted Execution Technology," `http://www.intel.com/technology/security/`.

[27] "Processor-Based Virtualization, AMD64 Style," `http://developer.amd.com/documentation/articles/pages/630200615.aspx`.

[28] Kauer, B. (2007) "OSLO: Improving the Security of Trusted Computing," in *16th USENIX Security Symposium*.

[29] "Trusted Boot," `http://sourceforge.net/projects/tboot`.

[30] Sailer, R., X. Zhang, T. Jaeger, and L. van Doorn (2004) "Design and Implementation of a TCG-based Integrity Measurement Architecture," in *USENIX Security Symposium*.

[31] "Trousers," `http:/http://trousers.sourceforge.net/`.

[32] Berger, S. et al. (2006) "vTPM: Virtualizing the Trusted Platform Module," in *USENIX Security Symposium*.

[33] Biba, K. J. (1975) *Integrity Considerations for Secure Computer Systems*, *Tech. Rep. MTR-3153*, MITRE.

[34] Clark, D. D. and D. R. Wilson (1987) "A Comparison of Commercial and Military Computer Security Policies," *Security and Privacy*, **00**.

[35] Sun, W., R. Sekar, G. Poothia, and T. Karandikar (2008) "Practical Proactive Integrity Preservation: A Basis for Malware Defense," in *Proc. 2008 IEEE SSP*.

[36] Shankar, U., T. Jaeger, and R. Sailer (2006) "Toward Automated Information-Flow Integrity Verification for Security-Critical Applications," in *Proc. 2006 NDSS*.

[37] Stumpf, F., A. Fuchs, S. Katzenbeisser, and C. Eckert (2008) "Improving the scalability of platform attestation," in *ACM Workshop on Scalable Trusted Computing*.

[38] Li, N., Z. Mao, and H. Chen (2007) "Usable Mandatory Integrity Protection for Operating Systems," in *Proc. IEEE SSP*.

[39] TCG (2005), "Trusted Computing Group," `http://www.trustedcomputinggroup.org/`.

[40] Shi, E., A. Perrig, and L. van Doorn (2005) "BIND: A Fine-Grained Attestation Service for Secure Distributed Systems," in *IEEE SP '05*.

[41] Jaeger, T., R. Sailer, and U. Shankar (2006) "PRIMA: Policy-Reduced Integrity Measurement Architecture," in *Proc. 11th ACM SACMAT*.

[42] TCG (2005) *Trusted Network Connect: Open Standards for Integrity-based Network Access Control*, *Technical report*, TCG, `http://www.trustedcomputinggroup.org/files/`.

[43] OpenTC, "OpenTC PET," http://www.opentc.net/publications/OpenTC_PET_prototype_documentation_v1.0.pdf.

[44] Schiffman, J., T. Moyer, C. Shal, T. Jaeger, and P. McDaniel (2009) "Justifying Integrity Using A Virtual Machine Verifier," in *Proc. 25th ACSAC (ACSAC '09)*.

[45] NIST, "SECURE HASH STANDARD," http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[46] King, S. T., P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch (2006) "SubVirt: Implementing malware with virtual machines," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, IEEE Computer Society, Washington, DC, USA, pp. 314–327.
URL http://portal.acm.org/citation.cfm?id=1130235.1130383

[47] Brickell, E., J. Camenisch, and L. Chen (2004) "Direct Anonymous Attestation," in *Proc. 11th ACM Conference on Computer and Communications Security*.

[48] Projects, T. C., "Chrome Recovery Mode," http://www.chromium.org/chromium-os/chromiumos-design-docs/recovery-mode.

[49] Ning, P., V. Atluri, S. Xu, and M. Yung (eds.) (2007) *Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing, STC 2007, Alexandria, VA, USA, November 2, 2007*, ACM.

[50] Loscocco, P. A., P. W. Wilson, J. A. Pendergrass, and C. D. McDonell (2007) "Linux Kernel Integrity Measurement Using Contextual Inspection," in *Proc. ACM workshop on Scalable trusted computing*.
URL http://doi.acm.org/10.1145/1314354.1314362

[51] "MAX6369 Watchdog Timer," http://www.maxim-ic.com/datasheet/index.mvp/id/2226.

[52] "GRUB TCG Patch to support Trusted Boot," http://trousers.sourceforge.net/grub.html.

[53] "Integrity: Linux Integrity Module(LIM)," http://lwn.net/Articles/287790/.

[54] Moyer, T., K. Butler, J. Schiffman, P. McDaniel, and T. Jaeger (2009) "Scalable asynchronous web content attestation," in *Proc. 25th ACSAC (ACSAC '09)*.

[55] Shrivastava, S. (2006) "Satem: Trusted Service Code Execution across Transactions," in *SRDS '06*.

[56] Wave, "EMBASSY Authentication Server," http://www.wave.com/products/eas.asp.

[57] Symantec, "PGP Whole Disk Encryption from Symantec," http://www.symantec.com/business/whole-disk-encryption.

[58] "BitLocker Drive Encryption: Technical Overview," http://technet.microsoft.com/en-us/windowsvista/aa906017.aspx.

[59] Gasmi, Yacine and Sadeghi, Ahmad-Reza and Stewin, Patrick and Unger, Martin and Asokan, N. (2007) "Beyond Secure Channels," in *Proc. ACM Workshop on Scalable Trusted Computing*.
URL http://doi.acm.org/10.1145/1314354.1314363

[60] Goldman, K., R. Perez, and R. Sailer (2006) "Linking Remote Attestation to Secure Tunnel Endpoints," in *Proc. First ACM Workshop on Scalable Trusted Computing*.
URL http://doi.acm.org/10.1145/1179474.1179481

[61] TCG (2005) *Infrastructure Subject Key Attestation Evidence Extension Version 1.0, Revision 5.*, *Technical report*.

[62] Latze, C., "Very First EAP-TPM Prototype," http://diuf.unifr.ch/people/latzec/prototyping/first.

[63] McCune, J. M., B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki (2008) "Flicker: An Execution Infrastructure for TCB Minimization," in *Proc. 3rd ACM SIGOPS/EuroSys*.

[64] McCune, J. M., Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig (2010) "TrustVisor: Efficient TCB Reduction and Attestation," in *Proc. IEEE SSP*.
URL http://dx.doi.org/10.1109/SP.2010.17

[65] St. Clair, L., J. Schiffman, T. Jaeger, and P. McDaniel (2007) "Establishing and Sustaining System Integrity via Root of Trust Installation," in *Proc. ACSAC*.

[66] Litty, L., H. A. Lagar-Cavilla, and D. Lie (2008) "Hypervisor Support for Identifying Covertly Executing Binaries," in *Proc. 17th Usenix Security Symposium*.

[67] Sharif, M. I., W. Lee, W. Cui, and A. Lanzi (2009) "Secure in-VM monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security*.

[68] Garfinkel, T. and M. Rosenblum (2003) "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. NDSS*.

[69] Payne, B. D., M. Carbone, and W. Lee (2007) "Secure and Flexible Monitoring of Virtual Machines," in *Proc. 23rd ACSAC*.

[70] Fraser, T., M. R. Evenson, and W. A. Arbaugh (2008) "VICI Virtual Machine Introspection for Cognitive Immunity," in *Proceedings of the 2008 ACSAC*.
URL http://dx.doi.org/10.1109/ACSAC.2008.33

[71] Seshadri, A., M. Luk, N. Qu, and A. Perrig (2007) "Secvisor: A Tiny Hypervisor To Provide Lifetime Kernel Code Integrity For Commodity Oses," in *Proceedings of twenty-first ACM SOSP*.

[72] Wang, Z., X. Jiang, W. Cui, and P. Ning (2009) "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, ACM, New York, NY, USA, pp. 545–554.
URL http://doi.acm.org/10.1145/1653662.1653728

[73] Haeberlen, A., P. Kouznetsov, and P. Druschel (2007) "PeerReview: practical accountability for distributed systems," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, ACM, New York, NY, USA, pp. 175–188.
URL http://doi.acm.org/10.1145/1294261.1294279

[74] Haeberlen, A., P. Aditya, R. Rodrigues, and P. Druschel (2010) "Accountable virtual machines," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, USENIX Association, Berkeley, CA, USA, pp. 1–16.
URL http://portal.acm.org/citation.cfm?id=1924943.1924952

[75] Dunlap, G. W., S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen (2002) "ReVirt: enabling intrusion analysis through virtual-machine logging and replay," *SIGOPS Oper. Syst. Rev.*, **36**, pp. 211–224.
URL http://doi.acm.org/10.1145/844128.844148

[76] Levin, D., J. R. Douceur, J. R. Lorch, and T. Moscibroda (2009) "TrInc: small trusted hardware for large distributed systems," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, USENIX Association, Berkeley, CA, USA, pp. 1–14.
URL http://portal.acm.org/citation.cfm?id=1558977.1558978

[77] Zeldovich, N., S. Boyd-Wickizer, and D. Mazières (2008) "Securing distributed systems with information flow control," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, USENIX Association, Berkeley, CA, USA, pp. 293–308.
URL http://portal.acm.org/citation.cfm?id=1387589.1387610

[78] Krohn, M., A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris (2007) "Information flow control for standard OS abstractions," *SIGOPS Oper. Syst. Rev.*, **41**, pp. 321–334.
URL http://doi.acm.org/10.1145/1323293.1294293

[79] Maruyama, H., F. Seliger, N. Nagaratnam, T. Ebringer, S. Munetoh, S. Yoshihama, and T. Nakamura (2004) *Trusted Platform on Demand*, *Tech. Rep. RT0564*, IBM.

[80] McCune, J., S. Berger, R. Caceres, T. Jaeger, and R. Sailer (2006) "Shamon: A System for Distributed Mandatory Access Control," in *Proc. ACSAC*.

[81] Gallagher, R. P. (1988), "A Guide to Understanding Trusted Distribution in Trusted Systems," `http://www.fas.org/irp/nsa/rainbow/tg008.htm`.

[82] Nurmi, D., R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov (2009) "The Eucalyptus Open-Source Cloud-Computing System," in *9th International Symposium on Cluster Computing and the Grid*, IEEE Computer Society, Washington, DC, USA, pp. 124–131.

[83] "Norton Ghost," `http://www.symantec.com/norton/ghost`.

[84] Acronis, "True Image," `http://www.acronis.com/homecomputing/products/trueimage/index.html`.

[85] Microsoft, "Windows Deployment Services," `http://msdn.microsoft.com/en-us/library/aa967394.aspx`.

[86] (2010), "Rocks Clusters," `http://www.rocksclusters.org/wordpress/`.

[87] "Preboot Execution Environment (PXE) Specification," `http://www.intel.com/design/archives/wfm/downloads/pxespec.htm`.

[88] "The Jedi Packet Trick takes over the Deathstar," `http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-CANSEC10-Project-Maux-III.pdf`.

[89] Kim, G. H. and E. H. Spafford (1994) "The design and implementation of tripwire: a file system integrity checker," in *Conference on Computer and Communications Security*, ACM, New York, NY, USA, pp. 18–29.

[90] BBC, "Amazon apologises for cloud fault one week on," `http://www.bbc.co.uk/news/business-13242782`.

[91] Moyer, T., K. Butler, J. Schiffman, P. McDaniel, and T. Jaeger (2009) "Scalable Asynchronous Web Content Attestation," in *ACSAC '09*.

[92] Schiffman, J., T. Moyer, T. Jaeger, and P. McDaniel (2011) "Network-based Root of Trust for Installation," *IEEE Security & Privacy*.

[93] Anderson, J. P. (1972) *Computer Security Technology Planning Study*, *Tech. Rep. ESD-TR-73-51*, The Mitre Corporation, Air Force Electronic Systems Division, Hanscom AFB, Badford, MA.

[94] "Trousers," `http://trousers.sourceforge.net/`.

[95] Payne, B. D., M. Carbone, M. Sharif, and W. Lee (2008) "Lares: An Architecture for Secure Active Monitoring Using Virtualization," in *IEEE Symposium on Security and Privacy*.

[96] Haldar, V., D. Chandra, and M. Franz (2004) "Semantic remote attestation: a virtual machine directed approach to trusted computing," in *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*.

[97] Joshi, A., S. T. King, G. W. Dunlap, and P. M. Chen (2005) "Detecting past and present intrusions through vulnerability-specific predicates," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, ACM, New York, NY, USA, pp. 91–104.
URL `http://doi.acm.org/10.1145/1095810.1095820`

[98] Baliga, A., V. Ganapathy, and L. Iftode (2008) "Automatic Inference and Enforcement of Kernel Data Structure Invariants," in *Proc. ACSAC*.
URL `http://dx.doi.org/10.1109/ACSAC.2008.29`

[99] Andronick, J., D. Greenaway, and K. Elphinstone (2010) "Towards Proving Security in the Presence of Large Untrusted Components," in *Proc. 5th Workshop on Systems Software Verification*.

[100] Azab, A. M., P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky (2010) "HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity," in *Proc. 17th ACM Conference on Computer and Communications Security*.
URL `http://doi.acm.org/10.1145/1866307.1866313`

[101] Steinberg, U. and B. Kauer (2010) "NOVA: a microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, ACM, New York, NY, USA, pp. 209–222.

[102] Ta-Min, R., L. Litty, and D. Lie (2007) "Splitting interfaces: making trust between applications and operating systems configurable," in *OSDI*, USENIX Association, Berkeley, CA, USA.

[103] Murray, D. G., G. Milos, and S. Hand (2008) "Improving Xen security through disaggregation," in *VEE*, VEE '08, ACM.

[104] Hay, B. and K. Nance (2008) "Forensics examination of volatile system data using virtual introspection," *SIGOPS Oper. Syst. Rev.*, **42**, pp. 74–82.

[105] "VMWare VMsafe," `www.vmware.com/go/vmsafe`.

[106] Smalley, S., C. Vance, and W. Salamon (2001) *Implementing SELinux as a Linux Security Module*, *Tech. Rep. 01-043*, NAI Labs.

[107] Carbone, M., W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM conference on Computer and communications security*.

[108] Chen, P. M. and B. D. Noble (2001) "When Virtual Is Better Than Real," in *Proc. HotOS*.

[109] Badger, L., D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat (1995) "Practical Domain and Type Enforcement for UNIX," in *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, SP '95, IEEE Computer Society, Washington, DC, USA, pp. 66–.
URL http://portal.acm.org/citation.cfm?id=882491.884237

[110] Jaeger, T., R. Sailer, and X. Zhang (2003) "Analyzing Integrity Protection in the SELinux Example Policy," in *Proc. 12th USENIX-SS*.

[111] "Linux Kernel Backdoors And Their Detection," http://invisiblethings.org/papers/ITUnderground2004_Linux_kernel_backdoors.ppt.

[112] "CVE-2010-3081," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3081.

[113] Santos, N., K. P. Gummadi, and R. Rodrigues (2009) "Towards Trusted Cloud Computing," in *HOTCLOUD*.

[114] Sirer, E. G., W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider (2011) "Logical attestation: an authorization architecture for trustworthy computing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, New York, NY, USA, pp. 249–264.
URL http://doi.acm.org/10.1145/2043556.2043580

[115] Srinivasan, D., Z. Wang, X. Jiang, and D. Xu (2011) "Process out-grafting: an efficient "out-of-VM" approach for fine-grained process execution monitoring," in *Proceedings of the 18th ACM conference on Computer and communications security*, New York, NY, USA, pp. 363–374.
URL http://doi.acm.org/10.1145/2046707.2046751

[116] "NIST Definition of Cloud Computing," http://csrc.nist.gov/groups/SNS/cloud-computing/.

[117] Amazon, "Amazon Web Services," http://aws.amazon.com.

[118] "Rackspace Cloud Servers," http://www.rackspace.com/cloud/.

[119] Ristenpart, T., E. Tromer, H. Shacham, and S. Savage "Hey, You, Get Off of my Cloud: Exploring Information Leakage in Third-Party Compute Clouds," in *CCS '09*, ACM.

[120] Rocha, F. and M. Correia (2011) "Lucy in the sky without diamonds: Stealing confidential data in the cloud," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, DSNW '11, IEEE Computer Society, Washington, DC, USA, pp. 129–134.
URL http://dx.doi.org/10.1109/DSNW.2011.5958798

[121] Mulazzani, M., S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl (2011) "Dark clouds on the horizon: using cloud storage as attack vector and online slack space," in *Proceedings of the 20th USENIX conference on Security*, SEC'11, USENIX Association, Berkeley, CA, USA, pp. 5–5.
URL http://dl.acm.org/citation.cfm?id=2028067.2028072

[122] Privacy Rights Clearinghouse, "Chronology of Data Breaches Security Breaches 2005 - Present," http://www.privacyrights.org/data-breach.

[123] Bugiel, S., S. Nürnberger, T. Pöppelmann, A.-R. Sadeghi, and T. Schneider (2011) "AmazonIA: when elasticity snaps back," in *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, ACM, New York, NY, USA, pp. 389–400.
URL http://doi.acm.org/10.1145/2046707.2046753

[124] Somorovsky, J., M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono (2011) "All your clouds are belong to us: security analysis of cloud management interfaces," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, ACM, New York, NY, USA, pp. 3–14.
URL http://doi.acm.org/10.1145/2046660.2046664

[125] Boneh, D. and B. Waters (2007) "Conjunctive, subset, and range queries on encrypted data," in *Proceedings of the 4th conference on Theory of cryptography*, TCC'07, Springer-Verlag, Berlin, Heidelberg, pp. 535–554.
URL http://dl.acm.org/citation.cfm?id=1760749.1760788

[126] Abdalla, M., M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. Malone-Lee, G. Neven, P. Paillier, and H. Shi (2008) "Searchable Encryption Revisited: Consistency Properties, Relation to Anonymous IBE, and Extensions," *J. Cryptol.*, **21**(3), pp. 350–391.
URL http://dx.doi.org/10.1007/s00145-007-9006-6

[127] Song, D. X., D. Wagner, and A. Perrig (2000) "Practical Techniques for Searches on Encrypted Data," in *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, IEEE Computer Society, Washington, DC, USA, pp. 44–.
URL http://dl.acm.org/citation.cfm?id=882494.884426

[128] Gentry, C. (2009) "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, ACM, New York, NY, USA, pp. 169–178.
URL http://doi.acm.org/10.1145/1536414.1536440

[129] Naehrig, M., K. Lauter, and V. Vaikuntanathan (2011) "Can homomorphic encryption be practical?" in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, ACM, New York, NY, USA, pp. 113–124.
URL http://doi.acm.org/10.1145/2046660.2046682

[130] Puttaswamy, K. P. N., C. Kruegel, and B. Y. Zhao (2011) "Silverline: toward data confidentiality in storage-intensive cloud applications," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, ACM, New York, NY, USA, pp. 10:1–10:13.
URL http://doi.acm.org/10.1145/2038916.2038926

[131] Bleikertz, S., T. Gross, and S. Mödersheim (2011) "Automated verification of virtualized infrastructures," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, ACM, New York, NY, USA, pp. 47–58.
URL http://doi.acm.org/10.1145/2046660.2046672

[132] Brown, A. and J. S. Chase (2011) "Trusted platform-as-a-service: a foundation for trustworthy cloud-hosted applications," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, ACM, New York, NY, USA, pp. 15–20.
URL http://doi.acm.org/10.1145/2046660.2046665

[133] Schiffman, J., T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel (2010) "Seeding Clouds with Trust Anchors," in *CCSW '10: Proceedings of the 2010 ACM workshop on Cloud computing security*, New York, NY, USA.

[134] Jana, S. and V. Shmatikov (2011) "EVE: Verifying Correct Execution of Cloud-Hosted Web Applications," in *3rd USENIX Workshop on Hot Topics in Cloud Computing*, Portland, OR.

[135] Wang, C. and Y. Zhou (2010) "A collaborative monitoring mechanism for making a multitenant platform accountable," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, USENIX Association, Berkeley, CA, USA, pp. 18–18.
URL http://dl.acm.org/citation.cfm?id=1863103.1863121

[136] Schiffman, J., H. Vijayakumar, and T. Jaeger (2012) "Verifying System Integrity by Proxy," in *TRUST*, pp. 179–200.

[137] "OpenStack," http://openstack.org/.

[138] Microsoft, "Windows Azure," http://www.windowsazure.com.

[139] Google, "Google App Engine," https://developers.google.com/appengine/.

[140] Danev, B., R. J. Masti, G. O. Karame, and S. Capkun (2011) "Enabling secure VM-vTPM migration in private clouds," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, ACM, New York, NY, USA, pp. 187–196.
URL http://doi.acm.org/10.1145/2076732.2076759

[141] Trojnara, M., "stunnel - multiplatform SSL tunneling proxy," http://www.stunnel.org/.

[142] Okajima, J. R., "Advanced Multi Layered Unification Filesystem," http://aufs.sourceforge.net/.

[143] Vijayakumar, H., G. Jakka, S. Rueda, J. Schiffman, and T. Jaeger (2012) "Integrity Walls: Finding Attack Surfaces from Mandatory Access Control Policies." in *Proceedings of the 7th ACM Symposium on Information, Computer, and Communications Security (AsiaCCS)*.

[144] Idziorek, J., M. Tannian, and D. Jacobson (2011) "Detecting fraudulent use of cloud resources," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, ACM, New York, NY, USA, pp. 61–72.
URL http://doi.acm.org/10.1145/2046660.2046676

[145] Mihoob, A., C. Molina-Jimenez, and S. Shrivastava (2010) "A Case for Consumer-centric Resource Accounting Models," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, IEEE Computer Society, Washington, DC, USA, pp. 506–512.
URL http://dx.doi.org/10.1109/CLOUD.2010.44

[146] "VMware vCenter Chargeback," http://www.vmware.com/products/vcenter-chargeback.

[147] "Amazon CloudWatch," http://aws.amazon.com/cloudwatch/.

[148] Sekar, V. and P. Maniatis (2011) "Verifiable resource accounting for cloud computing services," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, ACM, New York, NY, USA, pp. 21–26.
URL http://doi.acm.org/10.1145/2046660.2046666

[149] Cox, B., D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser (2006) "N-variant systems: a secretless framework for security through diversity," in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, USENIX Association, Berkeley, CA, USA.
URL http://dl.acm.org/citation.cfm?id=1267336.1267344

[150] Du, J., W. Wei, X. Gu, and T. Yu (2009) "Towards secure dataflow processing in open distributed systems," in *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, STC '09, ACM, New York, NY, USA, pp. 67–72.
URL http://doi.acm.org/10.1145/1655108.1655120

[151] ——— (2010) "RunTest: assuring integrity of dataflow processing in cloud computing infrastructures," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, ACM, New York, NY, USA, pp. 293–304.

[152] Du, J., X. Gu, and T. Yu (2010) "On verifying stateful dataflow processing services in large-scale cloud systems," in *Proceedings of the 17th ACM conference*

on Computer and communications security, CCS '10, ACM, New York, NY, USA, pp. 672–674.

[153] Cachin, C. and M. Schunter, "A Cloud You Can Trust," http://spectrum.ieee.org/computing/networks/a-cloud-you-can-trust/0.

[154] und Planungsgesellschaft mbH, T. F., "TClouds," http://www.tclouds-project.eu/.

[155] Abbadi, I. M. (2012) "Clouds Trust Anchors," in *The 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-11) (to appear)*, IEEE.

[156] Bessani, A., I. M. Abbadi, S. Bugiel, E. Cesena, M. Deng, M. Grne, N. Marnau, S. Nrnberger, M. Pasin, and N. Schirmer (2012) "TClouds: Privacy and Resilience for Internet-scale Critical Infrastructures," in *European Research Activities in Cloud Computing* (D. Petcu and J. V. Poletti, eds.), chap. 6, Cambridge Scholars Publishing, pp. 160–186.

[157] Ateniese, G., R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song (2007) "Provable data possession at untrusted stores," in *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, ACM, New York, NY, USA, pp. 598–609.
URL http://doi.acm.org/10.1145/1315245.1315318

[158] Wei, J., X. Zhang, G. Ammons, V. Bala, and P. Ning (2009) "Managing security of virtual machine images in a cloud environment," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, ACM, New York, NY, USA, pp. 91–96.
URL http://doi.acm.org/10.1145/1655008.1655021

[159] Richter, W., G. Ammons, J. Harkes, A. Goode, N. Bila, E. De Lara, V. Bala, and M. Satyanarayanan (2011) "Privacy-sensitive VM retrospection," in *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, HotCloud'11, USENIX Association, Berkeley, CA, USA, pp. 10–10.
URL http://dl.acm.org/citation.cfm?id=2170444.2170454

[160] Srinivasan, D., Z. Wang, X. Jiang, and D. Xu (2011) "Process out-grafting: an efficient "out-of-VM" approach for fine-grained process execution monitoring," in *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, ACM, New York, NY, USA, pp. 363–374.
URL http://doi.acm.org/10.1145/2046707.2046751

[161] Christodorescu, M., R. Sailer, D. L. Schales, D. Sgandurra, and D. Zamboni (2009) "Cloud security is not (just) virtualization security: a short paper," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, ACM, New York, NY, USA, pp. 97–102.
URL http://doi.acm.org/10.1145/1655008.1655022

[162] Dolan-Gavitt, B., T. Leek, M. Zhivich, J. Giffin, and W. Lee (2011) "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, IEEE Computer Society, Washington, DC, USA, pp. 297–312.
URL http://dx.doi.org/10.1109/SP.2011.11

[163] Keller, E., J. Szefer, J. Rexford, and R. B. Lee (2010) "NoHype: virtualized cloud infrastructure without the virtualization," in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, ACM, New York, NY, USA, pp. 350–361.
URL http://doi.acm.org/10.1145/1815961.1816010

[164] McAfee, "McAfee DeepSAFE," http://www.mcafee.com/us/solutions/mcafee-deepsafe.aspx.

[165] "TrustedCloud Pty Ltd," http://www.trustedcloud.com.au/.

[166] Hoover, J. N., "NSA CIO Pursues Intelligence-Sharing Architecture," http://www.informationweek.com/news/government/leadership/229401971?pgno=1.

Acronyms

AIK Attestation Identity Key. 13

AUFS Advanced multi-layered Unification FileSystem. 76

CA certificate authority. 2

CRTM core root of trust for measurement. 15

CV Cloud Verifier. 60

DAC discretionary access control. 10

DRTM dynamic root of trust for measurement. 12, 15, 48, 111

EK Endorsement Key. 13

IA integrity association. 44

IaaS Infrastructure as a Service. 58, 60, 62, 73

IDS intrusion detection system. 18

IM Integrity Measurement. 3, 4, 11

IMA Integrity Measurement Architecture. 15, 39, 40

initrd initial ramdisk. 48

IVP Integrity Verification Proxy. viii, 6, 37, 41, 60, 66

KVM Kernel-based Virtual Machine. 37, 81

LIM Linux Integrity Module. 15, 46, 82

LSM Linux Security Module. 15

MAC mandatory access control. 1

MLE measured launch environment. 48

NBP Network Boot Program. viii, 22

netROTI network-based Root of Trust for Installation. 21, 24, 75

NVRAM nonvolatile RAM. 11, 15

OA Outbound Authentication. 3, 4, 14

OSLO Open Source Loader. viii, 15, 29

PaaS Platform as a Service. 62

PCR Platform Configuration Register. 12

PKI Public Key Infrastructure. 2

PrivacyCA Privacy Certificate Authority. 13

PXE Preboot Execution Environment. 22

RNG random number generator. 11

ROTI Root of Trust for Installation. 25, 27, 48, 111

SaaS Service as a Service. 62

SLA service level agreement. 2

SLB secure loader block. 30

SLOC source lines of code. 49, 74

SRK Storage Root Key. 11

SRTM static root of trust for measurement. 12, 15, 16

TCB trusted computing base. 17

TCG Trusted Computing Group. 4, 9, 11

TCPA Trusted Computing Platform Alliance. 4

TPM Trusted Platform Module. 4, 11, 36

TSS Trusted Software Stack. 16

TXT Trusted eXecution Technology. 15, 48

VM virtual machine. 4, 16, 37, 39, 58

VMI virtual machine introspection. 18, 36, 39, 66, 71

VMM virtual machine monitor. 4, 17

vTPM virtual TPM. 17, 66

Glossary

attestation A statement generated in response to challenge that purports the current configuration of the attention system.. 3, 28

Attestation Identity Key TPM signing key whose private portion never leaves the TPM in plaintext. Can be used to identify the TPM through the use of a PrivacyCA.. 13, 107

authenticated boot A boot process designed to enable remote verifiers to detect when unexpected code entities are loaded.. 3

Cloud Verifier Framework for verifying the integrity of an IaaS cloud platform and the instances hosted on it.. 60, 107

configuration The combination of executing code and data that controls a system's behavior.. 9

Endorsement Key RSA signing key burned into the TPM. Uniquely identifies the physical platform, but cannot be used for attestation. Instead, the EK is used to certify that AIKs speak for the TPM.. 13, 107

Infrastructure as a Service A cloud model that provides VM hosting, networking, and storage resources to support virtual infrastructures.. 58, 107

integrity criteria The specification that a verifier defines for assessing the trustworthiness of remote systems.. 9, 10

Integrity Measurement Collection of events that affect a system's configuration.. 3, 107

Integrity Measurement Architecture Linux IM kernel module that records loading of binaries, kernel modules, and arbitrary files.. 15, 107

Integrity Verification Proxy Presented in Chapter 4, the IVP verifies arbitrary integrity criteria on behalf of a remote verifier by collecting integrity-relevant events from the monitored VM.. viii, 107

late launch CPU supported operation to reboot a system into a DRTM.. 12

measured launch environment Region of memory that is recorded by the CPU after a late launch operation is performed. This acts as the new starting point for the system's configuration.. 48, 108

measurement module An extensible IVP component that verifies specific integrity criteria by collecting information from the monitored VM through the available measurement framework.. 43, 71

network-based Root of Trust for Installation Installation technique that enables remote verifier to detect changes in a system's installed distribution. Binds the integrity of the distribution to its origin (installer).. 21, 108

Outbound Authentication Early attestation mechanism that produces a certificate chain of loaded code entities to prove how the current code configuration was created.. 3, 108

Platform Configuration Register Append only 160 bit registers that store arbitrary measurement as a SHA-1 hash chain. Registers are reset only at system reboot.. 12, 108

Privacy Certificate Authority CA that certifies an AIK is linked to a TPM's EK.. 13, 108

quote RSA signature over a selection of TPM PCRs signed with a key bound to the TPM (e.g., AIK).. 13

ROTI proof TPM quote that binds the installed filesystem to its installer.. 25, 27, 48

secure boot A boot process designed to prevent compromise by halting execution if unexpected code entities are loaded.. 1

trusted computing base The components (code and data) in a system that are assumed to be correct and secure.. 17, 108

Trusted Computing Group Industry group focused on developing trusted infrastructure standards.. 4, 108

Trusted eXecution Technology Intel's implementation of late launch in their CPUs. Uses the SENTER command to load an MLE.. 15, 109

Trusted Platform Module Inexpensive and low-power cryptographic co-processor on the LPC bus of many consumer electronics. Can perform cryptographic operations, generate keys for identifying the machine, and stores arbitrary measurements to generate attestations.. 4, 109

Vita

Joshua Serratelli Schiffman

Education

The Pennsylvania State University, University Park, PA. Ph.D in Computer Science and Engineering, August 2012.
The Pennsylvania State University, University Park, PA. M.S. in Computer Science and Engineering, May 2009.
The Pennsylvania State University, University Park, PA. B.S. with Honors and High Distinction in Computer Engineering, May 2006.

Awards and Honors

- ACM CCS Conference Student Travel Grant Award 2009
- ACM CCS Workshop Student Travel Grant Awards 2009, 2010
- University Graduate Fellowship Award 2008-2009
- USENIX Association Student Travel Stipend 2007-2011
- IEEE Security and Privacy Travel Grant 2009
- Penn State College of Engineering Fellowship 2006-2007
- ACM SIGMOD Undergraduate Scholarship 2006
- Admitted to the Phi Kappa Phi Honors Society 2006
- Lockheed Martin Engineering Scholars Award 2003

Professional Experience

- **Research Assistant** *Pennsylvania State University*, University Park, PA, 2006 - 2012.
  Led research projects in systems security, including trusted computing for mobile phone, virtual machine, and cloud computing platforms.
- **Co-Instructor** *Penn State University*, University Park, PA, Spring 2012
  Co-Instructor for CSE597E, a graduate seminar on systems security.
- **Research Intern** *Microsoft Research*, Redmond, WA, Summer 2011.
  Leveraged Infineon SLE secure hardware, modified Microsoft Hyper-V and designed a Windows Phone 7 application to maintain user privacy in personalized queries. Collaborated on the design and evaluated a new multi-client oblivious RAM protocol to protect user privacy in personalized data center services.
- **Research Intern** *Samsung Electronics R&D*, San Jose, CA, Summer 2009.
  Researched distributed cloud computing application security for mobile devices. Designed and implemented an access control manager for sub-delegation of the OAuth web authorization protocol in consumer electronics.
- **Research Co-op** *IBM T. J. Watson Research Center*, Hawthorne, NY, Summer 2008.
  Researched access control policies in virtual machine security and stream computing platforms.
- **Technical Intern** *Lockheed Martin*, King of Prussia, PA, Summer 2005, 2006.
  Developed web application prototypes for the Coast Guard's Deepwater program. Improved corporate web application for internal requisitions. Automated data entry for the Pennsylvania State Police ArcGIS services.

Program Committee

- **ACM Workshop on Scalable Trusted Computing (STC)**: *Member - 2012*
- **International Workshop on Security (IWSEC)**: *Member - 2012*