Trust Extension as a Mechanism for Secure Code Execution on Commodity Computers

Bryan Jeffrey Parno

April 19th, 2010

School of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Prof. Adrian Perrig (Chair) . . . . . Carnegie Mellon University
Prof. David Andersen . . . . . Carnegie Mellon University
Prof. Virgil Gligor . . . . . Carnegie Mellon University
Prof. John C. Mitchell . . . . . Stanford University
Prof. Gene Tsudik . . . . . University of California, Irvine

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was supported by CyLab at Carnegie Mellon under grants CNS-0831440 and CCF-0424422 from the National Science Foundation (NSF), and by a gift from Advanced Micro Devices (AMD), Inc. The author was also supported in part by a National Defense Science and Engineering (NDSEG) Fellowship, which is sponsored by the Department of Defense, as well as by a National Science Foundation Graduate Research Fellowship.

The views and conclusions contained here are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AMD, Carnegie Mellon, NSF, or the U.S. Government or any of its agencies.

Dedicated to Diana for 101 reasons

Abstract

As society rushes to digitize sensitive information and services, it is imperative to adopt adequate security protections. However, such protections fundamentally conflict with the benefits we expect from commodity computers. In other words, consumers and businesses value commodity computers because they provide good performance and an abundance of features at relatively low costs. Meanwhile, attempts to build secure systems from the ground up typically abandon such goals, and hence are seldom adopted [8, 72, 104].

In this dissertation, I argue that we can resolve the tension between security and features by leveraging the trust a user has in one device to enable her to securely use another commodity device or service, without sacrificing the performance and features expected of commodity systems. At a high level, we support this premise by developing techniques to allow a user to employ a small, trusted, portable device to securely learn what code is executing on her local computer. Rather than entrusting her data to the mountain of buggy code likely running on her computer, we construct an on-demand secure execution environment which can perform security-sensitive tasks and handle private data in complete isolation from all other software (and most hardware) on the system. Meanwhile, non-security-sensitive software retains the same abundance of features and performance it enjoys today.

Having established an environment for secure code execution on an individual computer, we then show how to extend trust in this environment to network elements in a secure and efficient manner. This allows us to reexamine the design of network protocols and defenses, since we can now execute code on endhosts and trust the results within the network. Lastly, we extend the user's trust one more step to encompass computations performed on a remote host (e.g., in the cloud). We design, analyze, and prove secure a protocol that allows a user to outsource arbitrary computations to commodity computers run by an untrusted remote party (or parties) who may subject the computers to both software and hardware attacks. Our protocol guarantees that the user can both verify that the results returned are indeed the correct results of the specified computations on the inputs provided, and protect the secrecy of both the inputs and outputs of the computations. These guarantees are provided in a non-interactive, asymptotically optimal (with respect to CPU and bandwidth) manner.

Thus, extending a user's trust, via software, hardware, and cryptographic techniques, allows us to provide strong security protections for both local and remote computations on sensitive data, while still preserving the performance and features of commodity computers.

Acknowledgements

First, I would like to thank my adviser, Professor Adrian Perrig, for his relentless optimism and for his support of my varied research interests. Adrian's dedication, creativity, and enthusiasm are inspiring. He taught me, even as a first-year graduate student, to think big and aim high. I would also like to thank Professors Dave Andersen, Lujo Bauer, David Brumley, Virgil Gligor, and Mike Reiter for numerous discussions, extensive advice, helpful feedback, and constant support.

I am indebted to all of my coauthors over the years for all of the great ideas, hard work, late nights, and fun times. I would especially like to thank Jon McCune, a great collaborator and a great friend. I am still amazed at his ability to relentlessly track down the most obscure bugs and his talent for methodically constructing incredibly complex systems. His dedication to concrete details leavened my tendency towards abstraction and left our research greatly enriched. As mentors, Helen Wang, Ari Juels, and Rosario Gennaro expanded my horizons and fostered my growth as a researcher. I also benefited from my interactions with the members of the Parallel Data Lab, especially Mike, James, Raja, and Matthew, who introduced me to topics far from my own area of expertise and who always impressed me with the rigor of their experiments. My freedom to explore a wide range of research topics was supported by NDSEG and NSF fellowships, for which I am grateful.

Many friends have brightened my life in Pittsburgh and provided much needed distraction and entertainment. I would particularly like to thank Jim and Bonnie, Scott and Ginger, Jon and Kathleen, Ahren and Casey, Dan and Lori, James and Anne, and Mike for welcoming me into their homes and lives.

In college, I was fortunate to take graduate-level courses taught by Professors Margo Seltzer and Matt Welsh. Their classes introduced me to the world of research, inspired me to enter graduate school, and taught me many of the research skills I use to this day.

So much of who and what I am comes from my phenomenal family. I am incredibly lucky to have grown up in such a nurturing environment. My parents, in particular, never wavered in their love, confidence, or support. Their example is both humbling and inspiring.

Finally, my most heartfelt thanks go to Diana: my biggest fan and my best friend. I could not and would not have made it without her constant love and encouragement. She is a wonderful partner and a dream come true.

Contents

List of Figures

Chapter 1

Introduction

1.1 Insecure Computers in a Hostile World

Businesses and individuals are entrusting progressively greater amounts of security-sensitive data to computers, both their own and those of third parties. To be worthy of this trust, these computers must ensure that the data is handled with care (e.g., as the user expects), and protected from external threats. Unfortunately, today's computer platforms provide little assurance on either front. Most platforms still run code designed primarily for features, not security. While it is difficult to measure precisely, multiple heuristics suggest that code quality has improved relatively little with respect to security. For example, the majority of coding projects on SourceForge employ type-unsafe languages (e.g., C or C++) [92]. The National Vulnerabilities Database [150] catalogues thousands of new software vulnerability reports each year, and recent studies indicate that over 25% of US computers are infected with some form of malicious software [152].

These vulnerabilities are particularly troubling when coupled with the increasingly hostile environment to which computers (and users) are exposed. Indeed, the lucrative and difficult-to-prosecute crimes that computers facilitate have given rise to a burgeoning criminal underground in which sophisticated, financially-motivated attackers collaborate to monetize exploited computers and stolen user data [62]. These ne'er-do-wells can employ automated, turn-key packages to attack thousands of potential victims every second [187]. The victim computers are then often formed into coordinated "botnets" of tens or hundreds of thousands of machines and used to send spam or launch Distributed Denial of Service (DDoS) attacks [91, 139].

As a result, from the moment a user digitizes her data, it is under constant assault from all sides. Malicious software on the user's computer may snatch up the user's private data and ship it off to foreign lands. If the user entrusts her data or computations to a remote service, then the remote computers may be subject to all manner of physical attacks, as surveyed in Section 2.9.2. Furthermore, experience demonstrates that remote workers will attempt to return forged results even when the only payoff is an improvement of their standings in an online ranking [145]; when there is a potential to profit from such skulduggery, workers' temptations can only increase.

Sadly, these attacks on user data succeed all too often. They contribute to the over 3.6 million U.S. households that were victims of identity theft in the year 2004 alone [49]. They also undermine users' trust in electronic systems, and hence inhibit both current and future systems. For example, a 2005 *Consumer Reports* survey found that 29% of consumers had cut back on – and 25% had stopped – shopping online due to fears of fraud and identity theft [159]. Digitizing medical records could potentially reduce skyrocketing costs (studies estimate that savings from a nationwide program in the U.S. could amount to $162-346 billion annually [7]) and help save the lives of the 44,000 to 98,000 Americans killed every year as a result of medical errors [7], for example, by automatically flagging potentially dangerous prescription interactions. Nonetheless, such efforts have been hampered by legitimate fears that such digital medical records will be insecure.

1.2 A Vision for a Better World

I envision a future in which average computer users can easily and securely use their computers to perform sensitive tasks (e.g., paying bills, shopping online, or accessing medical records), while still retaining the flexibility and performance expected of modern computers. Users will regard a computer attack not as a disaster that empties bank accounts or destroys documents, but as a minor annoyance; like a blown electrical fuse, it will be easy to detect and simple to remedy. Providing security as a largely invisible default will allow the information age to finally reach its true potential: users will submit their personal information to online sites, not blindly or with trepidation, but with confidence that it cannot be stolen or misused; businesses and consumers will feel perfectly secure outsourcing work to computational services; and remote, web-based applications will provide the same level of privacy, security, and availability that native applications do.

Achieving this goal will require advances on many fronts: better programming languages, better operating systems, better network protocols, and better definitions of security.

More fundamentally, however, we must enable both computers and users to make accurate, informed trust decisions. After all, even if software does improve, we must be able to determine *which* systems employ the new and improved software! This applies both to users and to network components. In other words, it is critical that a user be able to judge whether a system (either local or remote) should be trusted before she hands over her sensitive data. Similarly, if a network element (e.g., a router) can trust information from an endhost, then numerous protocol optimizations become possible.

In this work, we focus on a definition of trust similar to the definition of a Nash Equilibrium; for example, to trust an entity X with her private data (or with a security-sensitive task), a user Alice must believe that at no point in the future will she have cause to regret having given her data (or entrusted her task) to X. As a result, this dissertation examines techniques that provide firm evidence on which to base such a belief. As an additional constraint, we concentrate on average users and commodity systems, rather than on advanced users, special-purpose computers, or highly constrained environments (such as those found within the military).

Alas, previous efforts to construct trustworthy systems "from the ground up" have proven difficult, time-consuming, and unable to keep pace with the changing demands of the marketplace [8, 72, 104, 113]. For example, the VAX VMM security kernel was developed over the course of eight years of considerable effort, but in the end, the project failed, and the kernel was never deployed. This failure was due, in part, to the absence of support for Ethernet – an emerging and highly popular feature considered critical by the time the kernel was completed, but not anticipated when it was initially designed [104]. Thus, such efforts have typically been doomed, and their methods have not been adopted into the mainstream of software development.

1.3 Overview: Building Up from a Firm Foundation

Rather than starting over, the thesis of this work is that *we can design techniques that allow users to leverage the trust they have in one device to securely use another device or service.*

As we describe in more detail below, we start from the assumption that the user has some small, portable, canonically trusted device, such as a special-purpose USB device [203] or cellphone. The question of how this initial trust is established is outside the scope of this work, though McCune et al. explore some of the related issues [136]. Given this canonically trusted device, we analyze the difficulties that arise when attempting to use it to establish trust in an ordinary computer, particularly one equipped with the latest security hardware

enhancements. However, solving this problem merely reveals that the user's computer is likely running millions of lines of potentially buggy (and hence untrustworthy) code. To enable secure functionality on such a platform, we design and implement the Flicker architecture, which provides strong security protections *on demand*, while still allowing users to enjoy the features and performance they have come to expect from general-purpose computers. Given this more secure architecture for an individual computer, we next examine the question of how we can extend that trust into the network. In other words, how can we improve the security and/or performance of network protocols if we can verify that at least some portion of the code on an endhost can be trusted? Finally, we consider the protections we can offer to outsourced computations. In other words, if the user trusts her own machine, how can she extend that trust to computations done by a remote entity (for example, as part of a cloud computing service)? In particular, what guarantees can we provide with regards to the secrecy and integrity of the computations if we trust neither the software nor the hardware of the remote party?

1.4 Bootstrapping Trust in a Commodity Computer

Initially, we focus on the problem of allowing a user to bootstrap trust in her own personal computer. This problem is fundamental, common, and should be easier than other potential scenarios. In other words, if we cannot establish trust in the user's computer, we are unlikely to be able to establish trust in a remote computer. When working with her own computer, the user can at least be reasonably certain that the computer is physically secure; i.e., an attacker has not tampered with the computer's hardware configuration. Such an assumption aligns quite naturally with standard human intuition about security: a resource (e.g., a physical key) that an individual physically controls is typically more secure than a resource she gives to someone else. Fortunately, the physical protection of valuable items has been a major focus of human ingenuity over the past several millennia.

If the user's computer is physically secure, then we can make use of special-purpose hardware to support the user's security decisions. While a full-blown secure coprocessor, such as the IBM 4758 [186], might be appealing, cost and performance considerations make deployment difficult. However, for the last few years, many commodity computers have come equipped with a Trusted Platform Module (TPM) [200] that can be used for a variety of security-related purposes, as we discuss in Chapter 2.

Unfortunately, at present, no standard mechanism exists for establishing trust in the TPM on a local machine. Indeed, any straightforward approach falls victim to a *cuckoo*

attack [154]. In this attack, the adversary extracts the private keys from a TPM under his physical control. These keys can be given to malicious software present on the user's local computer, in order to fool the user into accepting reassurances from the adversary's TPM, rather than her own.

Thus, in Chapter 3, we propose a formal model for establishing trust in a platform. The model reveals the cuckoo attack problem and suggests potential solutions. We survey the usability challenges entailed by each solution, and suggest preferred approaches to enable a user to bootstrap trust in the secure hardware on her personal computer.

1.5 Securely Executing Code on a Commodity Computer

Unfortunately, merely establishing a secure connection between the user and the security hardware on her computer does not suffice to provide a full-featured, trustworthy execution environment. As mentioned earlier, security hardware tends to be either resource-impoverished or special-purpose (or both). Hence, we desire mechanisms to leverage the user's trust in the security hardware into trust in the user's entire computer. Previous techniques [63, 66, 170] tended to overwhelm the user with extraneous information, so we focus on techniques to provide fine-grained, meaningful, and trustworthy reports (or attestations) of only the security-relevant code.

However, establishing truly secure functionality on a general-purpose computer raises a fundamental question: How can secure code execution coexist with the untrustworthy mountain of buggy yet feature-rich software that is common on modern computers? For example, how can we keep a user's keystrokes private if the operating system, the most privileged software on the computer, cannot be trusted to be free of vulnerabilities? This is made all the more challenging by the need to preserve the system's existing functionality and performance.

To address these challenges, Chapter 4 presents the Flicker architecture [129, 131, 132, 133], which is designed to satisfy the need for features *and* security. Indeed, Flicker shows that these conflicting needs can both be satisfied by constructing a secure execution environment *on demand*, using a combination of software techniques and recent commodity CPU enhancements. When invoked for a security task (e.g., signing an email or authenticating to a website), Flicker protects the execution of that code from all other software on the system, as well as from potentially malicious devices (e.g., an Ethernet card with malicious firmware). Since we only deploy Flicker's protections on demand, Flicker, unlike previous approaches discussed in Chapter 2, induces no performance overhead or feature reduction

during regular computer use. Limiting Flicker's persistence to the time necessary for the task also strengthens Flicker's security guarantees, since it avoids the complexity (and hence potential vulnerability) of solutions based on a virtual machine monitor or a security kernel. Naturally, non-persistence poses its own set of challenges, so we develop (or adapt) protocols to securely preserve state between Flicker invocations, to verifiably establish secure channels between the Flicker environment and a remote party, and to prevent various subtle attacks on Flicker's records.

As a result, Flicker provides a solid foundation for constructing secure systems that operate in conjunction with standard software; the developer of a security-sensitive code module need only trust her own code, plus as few as 250 lines of Flicker code, for the secrecy and integrity of her code's execution. Flicker guarantees these properties even if the BIOS, OS, and DMA-enabled devices are all malicious.

In Chapter 4, we demonstrate a full implementation of Flicker on an AMD platform and describe our development environment for simplifying the construction of Flicker-enabled code. We also show how Flicker can enhance the security of various classes of applications, including verifiable malware scanning, distributed computing, and SSH password handling. Since many applications require frequent, efficient trust establishment (e.g., for each client that connects), we suggest modifications to existing hardware architectures to facilitate more efficient trust establishment.

1.6 Leveraging Secure Code Execution to Improve Network Protocols

If we can provide secure code execution on endhosts, the next frontier is to examine how such trust can be used to improve the performance and efficiency of network applications. In other words, if endhosts (or at least portions of each endhost) can be trusted, then network infrastructure no longer needs to arduously and imprecisely reconstruct data already known by the endhosts.

In Chapter 5, through the design of a general-purpose architecture we call Assayer [156], we explore the issues in providing trusted host-based data, including the balance between useful information and user privacy, and the tradeoffs between security and efficiency. We also evaluate the usefulness of such information in three case studies: spam identification, distributed denial-of-service attack mitigation, and super-spreader worm detection.

To gain insight into the performance we could expect from such a system, we implement and evaluate a basic Assayer prototype. Our prototype requires fewer than 1,000 lines of

code on the endhost. Endhosts can annotate their outbound traffic in a few microseconds, and these annotations can be checked efficiently; even packet-level annotations on a gigabit link can be checked with a loss in throughput of only 3.7-18.3%, depending on packet size.

1.7 Secure Code Execution Despite Untrusted Software and Hardware

With Flicker, we assume that the user's computer is physically secure. To generalize Flicker's results, we need techniques to establish trust in code execution when even the hardware is untrustworthy. This scenario is particularly compelling as the growth of "cloud computing" and the proliferation of mobile devices contribute to the desire to outsource computing from a client device to an online service. In these applications, how can the client be assured that the secrecy of her data will be protected? Equally importantly, how can the client verify that the result returned is correct, without redoing the computation?

While various forms of homomorphic encryption can provide data secrecy [69, 202], the results in Chapter 6 demonstrate that we can efficiently *verify* the results of arbitrary tasks (abstracted as function evaluations) on a computational service (e.g., in the cloud) without trusting *any* hardware or software on that system. This contrasts with previous approaches that were inefficient or that could only verify the results of restricted function families.

To formalize secure computational outsourcing, Chapter 6 introduces the notion of *verifiable computing* [67]. Abstractly, a client wishes to evaluate a function F (e.g., sign a document or manipulate a photograph) over various, dynamically selected inputs $x_1, \ldots, x_k$ on one or more untrusted computers, and then verify that the values returned are indeed the result of applying F to the given inputs. The critical requirement, which precludes the use of previous solutions, is that the client's effort to generate and verify work instances must be *substantially less* than that required to perform the computation on her own.

Drawing on techniques from multi-party secure computation, as well as some recent developments in lattice-based cryptography, we present the first protocol for verifiable computing. It provably provides computational integrity for work done by an untrusted party; it also provides provable secrecy for the computation's inputs and outputs. Moreover, the protocol provides asymptotically optimal performance (amortized over multiple inputs). Specifically, the protocol requires a one-time pre-processing stage which takes $O(|C|)$ time, where C is the smallest known Boolean circuit computing F. For each work instance, the client performs $O(|m|)$ work to prepare an m-bit input, the worker performs $O(|C|)$ work to compute the results, and the client performs $O(|n|)$ work to verify the n-bit result.

This result shows that we can outsource arbitrary computations to untrusted workers, preserve the secrecy of the data, and efficiently verify that the computations were done correctly. Thus, verifiable computing could be used, for instance, in a distributed computing project like Folding@home [153], which outsources the simulation of protein folding to millions of Internet users. To prevent cheating, these projects often assign the same work unit to multiple clients and compare the results; verifiable computing would eliminate these redundant computations and provide strong cryptographic protections against colluding workers.

Thus, even without secure hardware, these results show that we can leverage a user's trust in one device to verify (and hence trust) the results of computations performed by an arbitrary number of remote, untrusted commodity computers.

1.8 Summary of Contributions

In the course of investigating techniques to leverage a user's existing trust in a device or service in order to securely utilize other devices or services, this dissertation makes the following high-level contributions.

1. A logical framework for analyzing the steps required to establish trust in a computer equipped with secure hardware.
2. A new approach for securely executing code on a platform that must also support untrusted legacy code. Providing security strictly on demand allows security to coexist with features and performance, and enables new properties, such as fine-grained meaningful attestations that report precisely about security-relevant actions.
3. An investigation (and development of a corresponding architecture) of the various ways in which trust in endhosts can improve the efficiency and security of network protocols.
4. A set of formal definitions for secure outsourcing of computation, as well as a protocol that satisfies those definitions; provides integrity and secrecy for the computation, the inputs, and the outputs; and achieves asymptotically optimal performance when amortized over multiple inputs.

Chapter 2

Background and Related Work in Trust Establishment

Suppose you are presented with two physically identical computers. One is running a highly-certified, formally-proven, time-tested software stack, while the other is running a commodity software stack that provides similar features, but is completely infested with highly sophisticated malware. How can you tell which computer is which? How can you decide which computer you should use to check your email, update your medical records, or access your bank account?

While the design and validation of secure software is an interesting study in its own right, we focus this chapter on a survey of existing techniques for bootstrapping trust in commodity computers, specifically by conveying information about a computer's current execution environment to an interested party. This would, for example, enable a user to verify that her computer is free of malware, or that a remote web server will handle her data responsibly.

To better highlight the research aspects of bootstrapping trust, we organize this chapter thematically, rather than chronologically. Thus, we examine mechanisms for securely collecting and storing information about the execution environment (Section 2.1), methods for using that information locally (Section 2.2), techniques for securely conveying that information to an external party (Section 2.3), and various ways to convert the resulting information into a meaningful trust decision (Section 2.4).

Bootstrapping trust requires some foundational *root of trust*, and we review various candidates in Section 2.5. We then consider how the process of bootstrapping trust can be validated (Section 2.6) and used in applications (Section 2.7). Of course, creating trust ultimately

involves human users, which creates a host of additional challenges (Section 2.8). Finally, all of the work we survey has certain fundamental limitations (Section 2.9).

Much of the research in this area falls under the heading of "Trusted Computing", the most visible aspect of which is the Trusted Platform Module (TPM), which has already been deployed on over 200 million computers [88]. In many ways, this is one of the most significant changes in hardware-supported security in commodity systems since the development of segmentation and process rings *in the 1960s*, and yet it has been met with muted interest in the security research community, perhaps due to its perceived association with Digital Rights Management (DRM) [11]. However, like any other technology, the TPM can be used for either savory or unsavory purposes. One goal of this chapter is to highlight the many ways in which it can be used to improve user security without restricting user flexibility.

While Trusted Computing is the most visible aspect of this research area, we show that many of the techniques used by Trusted Computing date back to the 1980s [66]. These ideas thus extend beyond Trusted Computing's TPM to the general concept of bootstrapping trust in commodity computers. This fact becomes all the more relevant as cellphones emerge as the next major computing platform (as of 2005, the number of cellphones worldwide was about double the number of personal computers [80, 207]). In fact, many cellphones already incorporate stronger hardware support for security than many desktop computers and use some of the techniques described in this chapter [14, 16]. Indeed, as CPU transistor counts continue to climb, CPU vendors are increasingly willing to provide hardware support for secure systems (see, for example, Intel and AMD's support for virtualization [3, 95], and Intel's new AES instructions, which provide greater efficiency and resistance to side-channel attacks [81]). Thus, research in this area can truly guide the development of new hardware-supported security features.

Contributions. In this chapter, we make the following contributions: **(1)** We draw attention to the opportunities presented by the spread of commodity hardware support for security. **(2)** We provide a unified presentation of the reasoning behind and the methods for bootstrapping trust. **(3)** We present existing research in a coherent framework, highlighting underexamined areas, and hopefully preventing the reinvention of existing techniques. While we aim to make this chapter accessible to those new to the area, we do not intend to provide a comprehensive tutorial on the various technologies; instead, we refer the interested reader to the various references, particularly those highlighted in Section 2.10, for additional details.

2.1 What Do We Need to Know? Techniques for Recording Platform State

In deciding whether to trust a platform, it is desirable to learn about its current state. In this section, we discuss why code identity is a crucial piece of platform state and how to measure it (Section 2.1.1). We then consider additional dynamic properties that may be of interest, e.g., whether the running code respects information-flow control (Section 2.1.2). Finally, we argue that establishing code identity is a more fundamental property than establishing any of the other dynamic properties discussed (Section 2.1.3). Unfortunately, the security offered by many of these techniques is still brittle, as we discuss in Section 2.9.

2.1.1 Recording Code Identity

Why Code Identity? To trust an entity X with her private data (or with a security-sensitive task), Alice must believe that at no point in the future will she have cause to regret having given her data (or entrusted her task) to X. In human interactions, we often form this belief on the basis of identity – if you know someone's identity, you can decide whether to trust them. However, while user identity suffices for some tasks (e.g., authorizing physical access), buggy software and user inexperience makes it difficult for a user to vouch for the code running on their computer. For example, when Alice attempts to connect her laptop to the corporate network, the network can verify (e.g., using a password-based protocol) that Alice is indeed at the laptop. However, *even if* Alice is considered perfectly trustworthy, this *does not* mean that Alice's laptop is free of malware, and hence it may or may not be safe to allow the laptop to connect.

Thus, to form a belief about a computer's future behavior, we need to know more than the identity of its user. One way to predict a computer's behavior is to learn its complete current state. This state will be a function of the computer's hardware configuration, as well as the code it has executed. While hardware configuration might be vouched for via a signed certificate from the computer's manufacturer, software state is more ephemeral, and hence requires us to establish *code identity* before we can make a trust decision.

Of course, the question remains: what constitutes code identity? At present, the state-of-the-art for identifying software is to compute a cryptographic hash over the software's binary, as well as any inputs, libraries, or configuration files used. The resulting hash value is often termed a *measurement*. We discuss some of the difficulties with the interpretation of this type of measurement, as well as approaches to convert such measurements into higher-level properties, in Section 2.4.

What Code Needs To Be Recorded? To bootstrap trust in a platform, we must, at the very least, record the identity of the code currently in control of the platform. More subtly, we also need to record the identity of any code that could have affected the security of the currently executing code. For example, code previously in control of the platform might have configured the environment such that the currently running code behaves unexpectedly or maliciously. In the context of the IBM 4758 secure coprocessor [185, 186], Smith analyzes in greater detail which pieces of code can affect the security of a given piece of software [183], examining issues such as previously installed versions of an application that may have had access to the currently installed application's secrets.

Who Performs the Measurements? The best time to measure a piece of software is before it starts to execute. At this point, it is in a fresh "canonical" form that is likely to be similar across many platforms [66, 127]. Once it starts executing, it will generate local state that may vary across platforms, making it difficult to evaluate the measurement. Thus, if the software currently in control of the platform is S_n, then the logical entity to measure S_n is the software that was previously in control of the platform, i.e., S_{n-1}. In other words, before executing S_n, S_{n-1} must contain code to record a measurement of S_n in its "pristine" state. This logical progression continues recursively, with each software S_i responsible for measuring software S_{i+1} before giving it control of the platform. These measurements document the *chain of trust* [200]; i.e., the party interpreting the measurements must trust each piece of software to have properly measured and recorded subsequently launched pieces of software. Of course, this leads to the question of who (or what) measures the first software (S_1) to execute on the system.

Ultimately, measuring code identity requires a hardware-based *root of trust*. After all, if we simply ask the running code to self-identify, malicious software will lie. As we discuss in Section 2.5, most research in this area uses secure hardware (e.g., secure coprocessors) for this purpose, but some recent work considers the use of general-purpose CPUs.

Thus, in a *trusted boot* (a technique first introduced by Gasser et al. [66]), a hardware-based root of trust initiates the chain of trust by measuring the initial BIOS code (see Figure 2.1). The BIOS then measures and executes the bootloader, and the bootloader, in turn, measures and executes the operating system. Note that a trusted boot *does not* mean that the software that has booted is necessarily trustworthy, merely that it must be trusted if the platform itself is to be trusted.

This process of temporal measurement collection can be extended to include additional information about less privileged code as well (i.e., code that is not in control of the platform). For example, the OS might record measurements of each application that it executes. On a

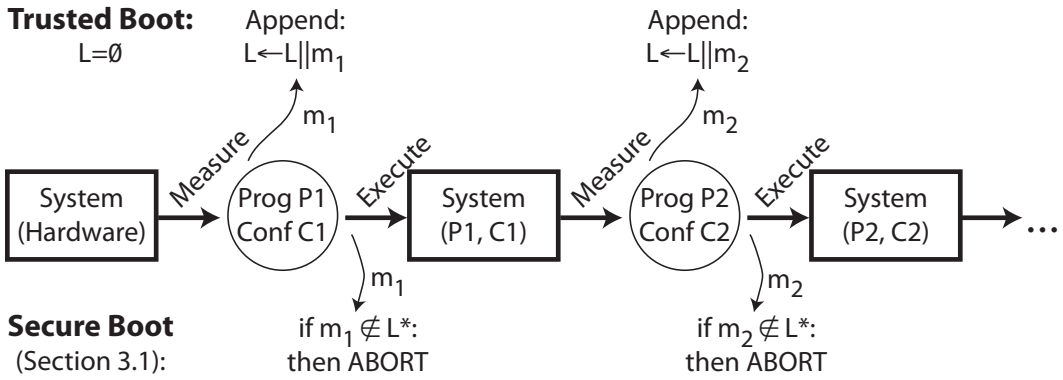


Figure 2.1: **Trusted Boot vs. Secure Boot.** *The state of a computer system changes as programs run with particular configurations. Trusted boot accumulates a list (L) of measurements for each program executed, but it does not perform any enforcement. Secure boot (Section 2.2.1) will halt the system if any attempt is made to execute a program that is not on an approved list (L*). Note that both systems must always measure programs before executing them. It is also possible to employ both types of boot simultaneously [66].*

general-purpose platform, this additional information is crucial to deciding if the platform is currently in a trustworthy state, since most modern operating systems do not, by themselves, provide enough assurance as to the security of the entire system.

On the other hand, if the software in control of the platform can be trusted to protect itself from, and maintain isolation between, less privileged code, then it may only need to record measurements of less privileged code that performs security sensitive operations. For example, the Terra project [63] observed that a trusted virtual machine monitor (VMM) can implement a trusted boot model both for itself and its virtual machines (VMs). This approach simplifies measurement, since the measurement of a single VM image can encompass an entire software stack. Furthermore, since a VMM is generally trusted to isolate itself from the VMs (and the VMs from each other), the VMM need only record measurements for the VMs that perform security-relevant actions.

Of course, virtualization can also complicate the use of secure hardware, since each VM may want or need exclusive control of it. The virtual Trusted Platform Module (vTPM) project [27] investigated how a single physical TPM can be multiplexed across multiple VMs, providing each with the illusion that it has dedicated access to a TPM.

How Can Measurements Be Secured? Of course, all of these code identity records must be secured; otherwise, malicious code might erase the record of its presence. This can happen in one of two ways (see Figure 2.2). First, in a *privilege escalation attack*, less privileged code may find an exploit in more privileged code, allowing it to access that code's secrets, erase

Chain Type	Attack Type: Privilege Escalation	Attack Type: Handoff Control to Malicious Code
Hash	Record latest value in HW	Record latest value in HW
Cert	Record latest value in HW	Prove access to latest key

Figure 2.2: **Securely Recording Code Measurements.** *Techniques for preventing attacks on the measurement record differ based on the method used to secure the record.*

the record of the malicious code's presence, or even create fake records of other software. Second, in a *handoff attack*, trusted software may inadvertently cede control of the platform to malicious software (e.g., during the boot process, the bootloader may load a malicious OS) which may attempt to erase any previously created records. Unfortunately, existing literature [63, 66, 170] tends to conflate these two types of attacks, obscuring the relative merits of techniques for securing measurements. While some research considers the design of a general-purpose, secure append-only log [173], it tends to make use of an independent logging server which may not be readily available in many environments.

Certificate Chains. Initial architecture designs for recording code identity measurements employed certificate chains [63, 66]. Before loading a new piece of software, Gasser et al. require the currently running system to generate a certificate for the new software [66]. To do so, the currently running system generates a new keypair for use by the new software and uses its private key to sign a certificate containing the new public key and a measurement of the new software. The system then erases its own secrets and loads the new software, providing the new keypair and certificate as inputs. As a result, a certificate chain connects the keypair held by the currently running software all the way back to the computer's hardware. This approach prevents handoff attacks, since by the time malicious code is loaded, the keys used to generate the certificate chain have been erased (this is an important point, often omitted in later work [63]). Thus, the only keypair the malicious code can both use (in the sense of knowing the private key) and produce a certificate chain for, is a keypair that is certified with a certificate containing the measurement of the malicious code. Thus, by requiring code to prove knowledge of a certified keypair, a remote entity can ensure that it receives an accurate measurement list.

A certificate chain, on its own, cannot prevent a privilege escalation attack from subverting the measurements. To maintain the certificate chain, privileged code must keep its private key available, and hence a privileged-escalated attacker can use that key to rewrite the chain. This attack can be prevented by recording a hash of the most recent certificate in

a more secure layer, such as secure hardware, though we are not aware of work suggesting this solution.

HASH CHAINS. Hash chains represent a potentially more efficient method of recording software measurements. A hash chain requires only a constant amount of secure memory to record an arbitrarily long, append-only list of code identities. As long as the current value of the hash chain is stored in secure memory, both privilege escalation and handoff attacks can be prevented. This is the approach adopted by the Trusted Platform Module (TPM) [200]. Several research efforts have applied this approach to the Linux kernel, and developed techniques to improve its efficiency [127, 170].

For a hardware-backed hash chain, the hardware sets aside a protected memory register that is initialized to a known value (e.g., 0) when the computer first boots. On a TPM, these protected memory registers are called Platform Configuration Registers (PCRs); current (version 1.2) TPMs are required to support at least 24 PCRs [200]. The software determining a new code module's identity I uses a hardware API to *extend* I into the log. The hardware computes a cryptographic hash over the the identity record and the current value V of the register and updates the register with the output of the hash: $V \leftarrow \texttt{Hash}(V||I)$. The software may keep an additional log of I in untrusted storage to help with the interpretation of the register's value at a future point. As long as `Hash` is collision-resistant, the register value V guarantees the integrity of the append-only log; i.e., even if malicious software gains control of the platform (via privilege escalation or a control handoff), it cannot erase its identity from the log without rebooting the platform and losing control of the machine.

Of course, without secure storage of the current value of the hash chain, a hash chain cannot protect the integrity of the log, since once malicious code gains control, it can simply replay the earlier extend operations and omit its measurement. There are no secret keys missing that would impede it.

TPM-Based Measurement Example. To make this discussion more concrete, we give an example of a TPM-based trusted boot sequence. This example is highly simplified; IBM's Integrity Measurement Architecture discusses the design and implementation of a much more complete solution for performing measurements [170]. We assume that the BIOS ($\mathcal{B}$), the bootloader ($\mathcal{L}$), and the operating system ($\mathcal{O}$) have all been modified to support measurement collection.

When the computer first boots, the TPM's PCRs are initialized to a known value (e.g., 0). ROM code then measures (computes a hash) of the BIOS ($\mathcal{B}$) and invokes `PCRExtend`

with a canonical PCR index, e.g., 5:

$$\texttt{PCRExtend}(5, \mathcal{B})$$

As a result, the TPM computes:

$$PCR_5 \leftarrow H(0||\mathcal{B})$$

The ROM code then starts executing the BIOS. The BIOS performs its usual initialization routines and extends a measurement of the bootloader ($\mathcal{L}$) into the TPM. It could choose a different PCR, but we will assume it continues to use PCR_5, so we have:

$$PCR_5 \leftarrow H(\underline{H(0||\mathcal{B})}||\mathcal{L})$$

The underlined value simply represents the previous value of PCR_5. After the `PCRExtend` operation, the BIOS can launch the bootloader. Similarly, the bootloader will extend a measurement of the OS ($\mathcal{O}$) into the TPM before staring to execute it. Finally, the OS will extend a measurement of the application ($\mathcal{A}$) into the TPM and launch the application. As a result, the value of PCR_5 is :

$$h = H(\underline{H(\underline{H(\underline{H(0||\mathcal{B})}||\mathcal{L})}||\mathcal{O})}||\mathcal{A})$$

Notice that the entire boot sequence is captured in a single hash value. Section 2.2–2.4 discuss how to use and interpret this information.

2.1.2 Recording Dynamic Properties

While code identity is an important property, it is often insufficient to guarantee security. After all, even though the system may start in a secure state, external inputs may cause it to arrive in an insecure state. Thus, before entrusting a computer with sensitive data, it might be useful to know whether the code has followed its intended control flow (i.e., that it has not been hijacked by an attack), preserved the integrity of its data structures (e.g., the stack is still intact), or maintained some form of information-flow control. We compare the merits of these dynamic properties to those of code identity in Section 2.1.3. Below, we discuss two approaches, load-time and run-time, to capturing these dynamic properties.

The simplest way to capture dynamic properties is to transform the program itself and then record the identity of the transformed program. For example, the XFI [57] and CFI [1] techniques transform a code binary by inserting inline reference monitors that enforce a va-

riety of properties, such as stack and control-flow integrity. By submitting the transformed binary to the measurement infrastructure described in Section 2.1.1, we record the fact that a program with the appropriate dynamic property enforcements built-in was loaded and executed. If the transformation is trusted to perform correctly, then we can extrapolate from the code identity that it also has the property enforced by the transformation. Of course, this approach does not protect against attacks that do not tamper with valid control flows [43]. For example, a buffer overflow attack might overwrite the Boolean variable `isAdministrator` to give the attacker unexpected privileges.

Another approach is to load some piece of code that is trusted to dynamically enforce a given security property on less-privileged code. An early example of this approach is "semantic" attestation [84], in which a language runtime (e.g., the Java or .NET virtual machine) monitors and records information about the programs it runs. For example, it might report dynamic information about the class hierarchy or that the code satisfies a particular security policy. In a similar spirit, the ReDAS system [109] loads a kernel that has been instrumented to check certain application data invariants at each system call. Trust in the kernel and the invariants that it checks can allow an external party to conclude that the applications running on the kernel have certain security-relevant properties. Again, this approach relies on a code identity infrastructure to identify that the trusted monitor was loaded.

2.1.3 Which Property is Necessary?

As discussed above, there are many code properties that are relevant to security, i.e., things we would like to know about the code on a computer before entrusting it with a security-sensitive task. However, since hardware support is expensive, we must consider what properties are fundamentally needed (as opposed to merely being more efficient in hardware).

The discussion in Section 2.1.2 suggests that many dynamic properties can be achieved (in some sense) using code identity. In other words, the identity of the code conveys the dynamic properties one can expect from it or the properties that one can expect it to enforce on other pieces of software. However, the converse does not appear to be true. That is, if a hardware primitive could report, for example, that the currently running code respected its intended control flow, then it is not clear how to use that mechanism to provide code identity. Furthermore, it clearly does not suffice to say anything meaningful about the security-relevant behavior of the code. A malicious program may happily follow its intended control-flow as it conveys the user's data to an attacker. Similar problems appear to affect other potential candidates as well. Knowing that a particular invariant has been maintained, whether it is stack integrity or information-flow control, is not particularly useful without knowing more

about the context (that is the code) in which the property is being enforced.

Thus, one can argue that code identity truly is a fundamental property for providing platform assurance, and thus a worthy candidate for hardware support. Of course, this need not preclude additional hardware support for monitoring (or enforcing) dynamic properties.

2.2 Can We Use Platform Information Locally?

We now discuss how accumulated platform information (Section 2.1) can benefit a local user. Unfortunately, these measurements cannot be used to directly provide information to local software; i.e., it does not make sense for higher-privileged software to use these measurements to convey information to less-privileged software, since the less-privileged software must already trust the higher-privileged software.

Nonetheless, in this section, we review techniques for using these measurements to convince the user that the platform has booted into a secure state, as well as to provide access control to a protected storage facility, such that secrets will only be available to a specific software configuration in the future. Such techniques tend to focus on preserving the secrecy and integrity of secrets, with less emphasis placed on availability. Indeed, using code identity for access control can make availability guarantees fragile, since a small change to the code (made for malicious or legitimate reasons) may make secret data unavailable.

2.2.1 Secure Boot

How can a user tell if her computer has booted into a secure state? One approach is to use a technique first described by Gasser et al. [66] and later dubbed "secure boot" [13].

In a computer supporting secure boot, each system component, starting with the computer's boot ROM, compares the measurement of code to be loaded to a list of measurements for authorized software (authorization is typically expressed via a signature from a trusted authority, which requires the authority's public key to be embedded in the computer's firmware) [13, 66]. Secure boot halts the boot process if there is an attempt to load unauthorized code, and thus assures the user that the platform is in an approved state simply by booting successfully.

One of the first systems to actually implement these ideas was AEGIS[1] [13]. With AEGIS, before a piece of software is allowed to execute, its identity is checked against a certificate

[1]Two relevant research efforts have used the name AEGIS. One is that of Arbaugh et al. [13] discussed in this section. The other is a design for a secure coprocessor by Suh et al. [195] and is discussed in Section 2.5.1.

from the platform's owner. The certificate identifies permitted software. Anything without a certificate will not be executed.

However, a remote party cannot easily determine that a computer has been configured for secure boot. Even if it can make this determination, it only learns that the computer has booted into some authorized state, but it does not learn any information about what specific state it happens to be in. Section 2.3 discusses the techniques needed to provide more information to a remote party.

2.2.2 Storage Access Control Based on Code Identity

Applications often require long-term protection of the secrets that they generate. Practical examples include the keys used for full disk encryption or email signatures, and a list of stored passwords for a web browser. Abstractly, we can provide this protection via an access control mechanism for cryptographic keys, where access policies consist of sets of allowed platform configurations, represented by the measurement lists described in Section 2.1. Below, we discuss two of the most prominent protected storage solutions: the IBM 4758 cryptographic co-processor and the Trusted Platform Module (TPM).

2.2.2.1 Tamper-Responding Protected Storage

The IBM 4758 family of cryptographic co-processors provides a rich set of secure storage facilities [53, 100, 185, 186]. First and foremost, it incorporates tamper-responding storage in battery-backed RAM (BBRAM). Additional FLASH memory is also available, but the contents of FLASH are always encrypted with keys maintained in BBRAM. The design intention is that any attempt to physically tamper with the device will result in it actively erasing secrets. Cryptographic keys that serve as the root for protected storage can be kept here.

The IBM 4758 enforces storage access restrictions based on the concept of software privilege layers. Layer 0 is read-only firmware. Layer 1 is, by default, the IBM-provided CP/Q++ OS. Layers 2 and 3 are for applications. Each layer can store secrets either in BBRAM or in FLASH. A hardware ratcheting lock prevents a lower-privilege layer from accessing the state of a higher-privilege layer. Thus, once an application loads at layer 2 or 3, the secrets of layer 1 are unavailable. Extensions to the OS in layer 1 could permit arbitrarily sophisticated protected storage properties, for example mirroring the TPM's sealed storage facility (discussed below) of binding secrets to a particular software configuration. The BBRAM is also ideal for storing secure counters, greatly simplifying defense against state replay attacks.

2.2.2.2 TPM-Based Sealed Storage

Despite providing much less functionality than a full-blown secure coprocessor, the TPM can also restrict storage access based on platform state. It does so by allowing software on the platform's main CPU to *seal* or *bind* secrets to a set of measurements representing some future platform state (we discuss the differences between these operations below). Both operations (seal and bind) essentially encrypt the secret value provided by the software. The TPM will refuse to perform a decryption, unless the current values in its Platform Configuration Registers (PCRs - see Section 2.1.1) match those specified during the seal or bind operation.

Full disk encryption is an example of an application that benefits from sealed storage. The disk encryption keys can be sealed to measurements representing the user's operating system. Thus, the disk can only be decrypted if the intended OS kernel has booted. (This is the basic design of Microsoft BitLocker, discussed in Section 2.7.) Connecting disk encryption with code identity prevents an attacker from modifying the boot sequence to load malware or an alternate OS kernel (e.g., an older kernel with known vulnerabilities).

To provide protected storage, both operations use encryption with 2048-bit asymmetric RSA keys. For greater efficiency, applications typically use a symmetric key for bulk data encryption and integrity protection, and then use the TPM to protect the symmetric key. The RSA keys are generated on the TPM itself,[2] and the private portions are never released in the clear. To save space in the TPM's protected storage area, the private portions are encrypted using the TPM's *Storage Root Keypair*. The private component of the keypair resides in the TPM's non-volatile RAM and never leaves the safety of the chip.

Sealing Data. With the TPM's seal operation, the RSA encryption must take place on the TPM. As a result, the TPM can produce a ciphertext that also includes the current values of any specified PCRs. When the data is later decrypted, the exact identity of the software that invoked the original seal command can be ascertained. This allows an application that unseals data to determine whether the newly unsealed data should be trusted. This may be useful, for example, during software updates.

Because sealing requires the TPM to perform the encryption, it would be much more efficient to use a symmetric encryption scheme, such as AES. The choice of RSA appears to have been an attempt to avoid adding additional complexity to the TPM's implementation, since it already requires an RSA module for other functionality.

[2] The TPM ensures that these keys are only used for encryption operations (using RSA PKCS #1v2.0 OAEP padding [200]) and never for signing.

Binding Data. In contrast to sealing, encryption using a public binding key need not take place on the TPM. This allows for greater efficiency and flexibility when performing data encryption, but it means that the resulting ciphertext does not include a record of the entity that originally invoked the bind operation, so it cannot be used to assess data integrity.

Replay Issues. Note that the above-mentioned schemes bind cryptographic keys to some representation of software identity (e.g., hashes stored in PCRs). Absent from these primitives is any form of freshness or replay-prevention. The output of a seal or bind operation is ciphertext. Decryption depends on PCR values and an optional 20-byte authorization value. It does not depend on any kind of counter or versioning system. Application developers must take care to account for versioning of important sealed state, as older ciphertext blobs can also be decrypted. An example attack scenario is when a user changes the password to their full disk encryption system. If the current password is maintained in sealed storage, and the old password is leaked, certain classes of adversaries may be able to supply an old ciphertext at boot time and successfully decrypt the disk using the old password. The TPM includes a basic monotonic counter that can be used to provide such replay protection. However, the TPM has no built-in support for combining sealed storage with the monotonic counter. Application developers must shoulder this responsibility. Section 2.7.2 discusses research on enhancing the TPM's counter facilities.

TPM-Based Sealed Storage Example. Here we continue the example begun in Section 2.1.1. Recall that we assumed that the BIOS ($\mathcal{B}$), the bootloader ($\mathcal{L}$) and the operating system ($\mathcal{O}$) have all been modified to record the appropriate code identity records in the TPM. If the OS is currently running an application ($\mathcal{A}$), then the value of PCR_5 is :

$$h = H(H(\underline{H(\underline{H(0||\mathcal{B})||\mathcal{L}})||\mathcal{O}})||\mathcal{A})$$

The application can generate secret data D_{secret} and seal it under the current value of PCR_5 by invoking:

$$\texttt{Seal}((5), D_{secret}) \rightarrow (C, MAC_{K_{root}}(5, h))$$

What benefit does this provide? If the same boot sequence is repeated (in other words, if the exact same BIOS, bootloader, OS and application are loaded in the same order) then clearly PCR_5 will take on the same value it had before. Thus a call to `Unseal` will produce D_{secret}. However, if any of these pieces of software changes, then the `Unseal` will fail. For example, suppose an attacker replaces the OS with a malicious OS ($\hat{\mathcal{O}}$). When the application

is executed, the value of PCR_5 will be:

$$\hat{h} = H(\underline{H(\underline{H(\underline{H(0||\mathcal{B})}||\mathcal{L})}||\hat{\mathcal{O}})}||\mathcal{A})$$

The properties of the hash function H guarantee that with extremely high probability $\hat{h} \neq h$, and thus if an attacker invokes `Unseal`, the TPM will refuse to decrypt C.

2.3 Can We Use Platform Information Remotely?

Section 2.1 described mechanisms for accumulating measurements of software state. In this section, we treat the issue of conveying these measurement chains to an external entity in an authentic manner. We refer to this process as *attestation*, though some works use the phrase *outbound authentication*. We also discuss privacy concerns and mitigation strategies that arise when sharing this information with third parties.

2.3.1 Prerequisites

The *secure boot* model (Section 2.2.1) does not capture enough information to securely inform a remote party about the current state of a computer, since it (at best), informs the remote party that the platform booted into some "authorized" state, but does not capture which state that happens to be, nor which values were considered during the authorization boot process.

Instead, a remote party would like to learn about the measurement of the currently executing code, as well as any code that could have affected the security of this code. Section 2.1 describes how a *trusted boot* process securely records this information in measurement chains (using either certificates or hashes).

2.3.2 Conveying Code Measurement Chains

The high-level goal is to convince a remote party (hereafter: verifier) that a particular measurement chain represents the software state of a remote device (hereafter: attestor). Only with an authentic measurement chain can the verifier make a trust decision regarding the attestor. A verifier's trust in an attestor's measurement chain builds from a hardware root of trust (Section 2.5). Thus, a prerequisite for attestation is that the verifier (1) understands the hardware configuration of the attestor and (2) is in possession of an authentic public key bound to the hardware root of trust.

The attestor's hardware configuration is likely represented by a certificate from its manufacturer, e.g., the IBM 4758's factory Layer 1 certificate [183], or the TPM's Endorsement, Platform, and Conformance Credentials [200]. Attestation-specific mechanisms for conveying public keys in an authentic way are treated with respect to privacy issues in Section 2.3.3. Otherwise, standard mechanisms (such as a Public Key Infrastructure) for distributing authentic public keys apply.

The process of actually conveying an authenticated measurement chain varies depending on the hardware root of trust. We first discuss a more general and more powerful approach to attestation used on general-purpose secure coprocessors such as the IBM 4758 family of devices. Then, given the prevalence of TPM-equipped platforms today, we discuss attestation as it applies to the TPM.

2.3.2.1 General Purpose Coprocessor-Based Attestation

Smith discusses the need for coprocessor applications to be able to authenticate themselves to remote parties [183]. This is to be distinguished from merely configuring the coprocessor as desired prior to deployment, or including a signed statement about the configuration. Rather, the code entity itself should be able to generate and maintain authenticated key pairs and communicate securely with any party on the internet. Smith details the decision to keep a private key in tamper-protected memory and have some authority generate certificates about the corresponding public key. As these coprocessors are expensive devices intended for use in high assurance applications, considerably less attention has been given to the device identity's impact on privacy.

Naming code entities on a coprocessor is itself an interesting challenge. For example, an entity may go through one or more upgrades, and it may depend on lower layer software that may also be subject to upgrades. Thus, preserving desired security properties for code and data (e.g., integrity, authenticity, and secrecy) may depend not only on the versions of software currently running on the coprocessor, but also on past and even future versions. The IBM 4758 exposes these notions as *configurations* and *epochs*, where configuration changes are secret-preserving and epoch changes wipe all secrets from the device.

During a configuration change, certificate chains incorporating historical data are maintained. For example, the chain may contain a certificate stating the version of the lowest layer software that originally shipped on the device, along with a certificate for each incremental upgrade. Thus, when a remote party interacts with one of these devices, all information is available about the software and data contained within.

This model is a relative strength of general-purpose cryptographic coprocessors. TPM-based attestations (discussed in the next section) are based on hash chains accumulated for no longer than the most recent boot cycle. The history of software that has handled a given piece of sensitive data is not automatically maintained.

Smith examines in detail the design space for attestation, some of which is specific to the IBM 4758, but much of which is more generally applicable [183]. A noteworthy contribution not discussed here is a logic-based analysis of attestation.

2.3.2.2 TPM-Based Attestation

TPM-based attestation affords less flexibility than general coprocessor-based attestation, since the TPM is not capable of general-purpose computation. During the attestation protocol (shown in Figure 2.3), software on the attestor's computer is responsible for relaying information between the remote verifier and the TPM [200]. The protocol assumes that the attestor's TPM has generated an Attestation Identity Keypair (AIK), which is an asymmetric keypair whose public component must be known to the verifier in advance, and whose private component is only accessible to the TPM. We discuss privacy issues regarding AIKs in Section 2.3.3.1.

During the protocol, the verifier supplies the attestor with a nonce to ensure freshness (i.e., to prevent replay of old attestations). The attestor then asks the TPM to generate a *Quote*. The Quote is a digital signature covering the verifier's nonce and the current measurement aggregates stored in the TPM's Platform Configuration Registers (PCRs). The attestor then sends both the quote and an accumulated measurement list to the verifier. This measurement list serves to capture sufficiently detailed metadata about measured entities to enable the verifier to make sense of them. Exactly what this list contains is implementation-specific. Marchesini et al. focus on the measurement of a long-term core (e.g., kernel) [127], while IBM's Integrity Measurement Architecture contains the hash and full path to a loaded executable, and recursively measures all dynamic library dependencies [170]. To check the accuracy of the measurement list, the verifier computes the hash aggregate that would have been generated by the measurement list and compares it to the aggregate signed by the TPM Quote. This verification process involves efficient hash function computations, so it is more efficient than performing a public-key based certificate verification for every measurement.

Preventing Reboot Attacks. A naive implementation of the above attestation protocol is susceptible to a *reboot* or *reset* attack. The basic weakness is a time-of-check to time-of-use (TOCTOU) vulnerability where the attesting platform is subject to primitive physical tampering, such as power-cycling the platform or components therein [191]. For example,

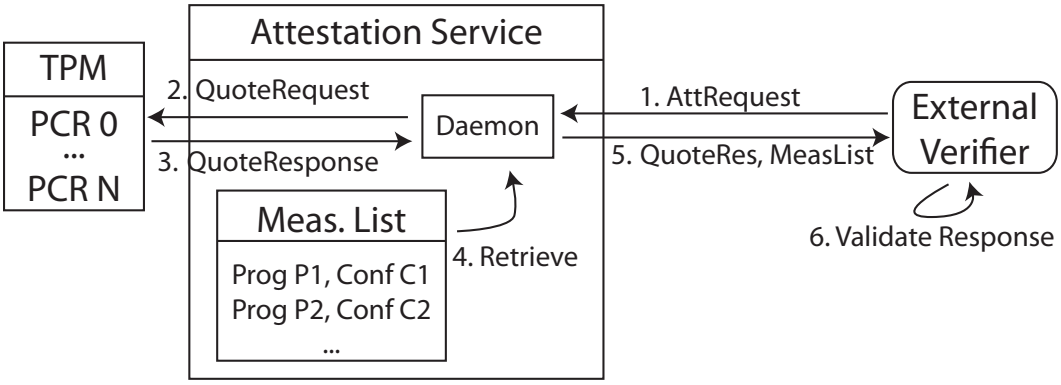


Figure 2.3: **Attestation.** *High-level summary of TPM-based attestation protocol based on signed hash chain measurements [200], e.g., as in IBM's Integrity Measurement Architecture [170]. Some protocol details are elided, e.g., the inclusion of an anti-replay nonce as part of the AttRequest message.*

the adversary may wait until the verifier has received an attestation, then reset the attestor and boot a malicious software image. Mitigating this attack requires a way to bind ephemeral session keys to the currently executing software [63, 73, 131]. These keys can then be used to establish a trusted tunnel (see below). A reboot destroys the established tunnel, thereby breaking the connection and preventing the attack.

Linking Code Identity to Secure Channels. Binding a secure channel (i.e., a channel that provides secrecy, integrity, and authenticity) to a specific code configuration on a remote host requires some care. Goldman et al. [73] consider an SSL client that connects to a server with attestation capabilities. Even if the client verifies the SSL certificate and the server's attestation, there is no linkage between the two. This enables an attack where a compromised SSL server forwards the client's attestation request to a different, trusted server. McCune et al. consider a similar challenge in establishing a secure channel between a client system and an isolated execution environment on a server [131]. Both conclude that the solution is to include a measurement of the public key used to bootstrap the secure channel in the attestation, e.g., extend the public key into one of the TPM's PCRs. Goldman et al. also discuss other more efficient solutions in the context of a virtualized environment.

2.3.3 Privacy Concerns

Participating in an attestation protocol conveys to the verifier detailed information about the software loaded for execution on a particular platform. Furthermore, the attestation often depends on a cryptographic key embedded in the secure hardware, and using the same key in multiple attestations allows those attestations to be linked together.

In some cases, this may not be a privacy concern. For example, in the military and in many enterprises, precise platform identification is desirable, and users do not have an expectation of privacy. As a result, some of the more expensive cryptographic co-processors that target these environments contain little provision for privacy.

However, in consumer-oriented applications, privacy is vital, and hence several techniques have been developed to maintain user privacy while still providing the ability to securely bootstrap trust.

2.3.3.1 Identity Certificate Authorities

One way to enhance user privacy is to employ a trusted third party to manage the relationship between a platform's true unique identity, and one or more pseudonyms that can be employed to generate attestations for different purposes. The Trusted Computing Group initially adopted this approach in the TPM [200], dubbing the trusted third party a *Privacy CA* and associating the pseudonyms with *Attestation Identity Keypairs* (AIKs). A TPM's true unique identity is represented by the *Endorsement Keypair* (EK) embedded in the TPM.[3]

At a high-level, the trusted third party validates the correctness of the user's secure hardware, and then issues a certificate declaring the user's pseudonym corresponds to legitimate secure hardware. With the TPM, the user can ask the TPM to generate an arbitrary number of AIKs. Using the TPM's EK, the user can convince the Privacy CA to issue a certificate for the public portion of an AIK, certifying that the private portion of the AIK is known only to a real, standards-compliant TPM. Of course, for many applications, it will be necessary to use a consistent pseudonym for that particular application (e.g., online banking).

The Privacy CA architecture described above has met with some real-world challenges. In reality, there is no one central authority trusted by all or even most users. Furthermore, a Privacy CA must be highly secure while also maintaining high availability, a nontrivial undertaking. To date, no commercial Privacy CAs are in operation, though a handful of experimental services have been created for research and development purposes [60].

[3] It is possible to clear a TPM's EK and generate a new one. However, once an EK is cleared, it cannot be reinstated (the private key is lost). Further, high-quality TPMs ship from the manufacturer with a certified EK. Without a certified EK, it is difficult for a Privacy CA to make a trust decision about a particular TPM. Generating one's own EK is most appropriate for security-aware enterprises with procedures in place to generate new EKs in physically controlled environments, or for highly security-conscious individuals.

2.3.3.2 Direct Anonymous Attestation

To address the limitations of Privacy CAs, a replacement protocol called Direct Anonymous Attestation (DAA) [33] was developed and incorporated into the latest TPM specification [200]. DAA is completely decentralized and achieves anonymity by combining research on group signatures and credential systems. Unlike many group signatures, it does not include a privileged group manager, so anonymity can never be revoked. However, it does allow membership to be revoked. In other words, an adversary's credentials can be invalidated without the system ever actually learning the adversary's identity.

With DAA, a TPM equipped platform can convince an *Issuer* that it possesses a legitimate TPM and obtain a membership certificate certifying this fact. However, the interaction with the Issuer is performed via zero-knowledge proofs, so that even if the Issuer colludes with a verifier, the user's anonymity is protected.

DAA also allows a user to select any desired level of privacy by employing an arbitrarily large set of pseudonyms. Thus, the user can be anonymous (by using a new pseudonym for every attestation), fully traceable (by using a single fixed pseudonym), or any level of privacy in between. These pseudonyms can be used to authorize standard TPM AIKs, so existing techniques for attestation continue to function.

In practice, however, DAA has been slow to catch on. No currently available hardware TPMs offer DAA support, due in part to the cost of implementing expensive group signature operations on the limited TPM processor. The DAA algorithm is also quite complex, since it offloads as much computation as possible to the system's (relatively) untrusted primary CPU.

Rudolph noted some weaknesses in the original DAA design that could undermine its anonymity properties [166], primarily by having the Issuer employ different long-term keys for different users. Several fixes have been proposed [121], but these attacks highlight the ability of implementation "details" to undermine the security of formally proven systems.

2.4 How Do We Make Sense of Platform State?

Knowing what code is executing on a platform does not necessarily translate into knowing whether that code can be trusted. In this section, we elaborate on this problem (2.4.1) and then review solutions that fall into two broad categories: solutions that provide only the identity of security-relevant code (2.4.2), and those that convey higher-level information (2.4.3).

2.4.1 Coping With Information Overload

At a high-level, converting code identity into security properties is simply an exercise in software engineering. If we build perfectly secure software, then knowing this bulletproof code is running on a computer suffices to assure us that the computer can be trusted. Unfortunately, developing software with strong security properties, even minimal security kernels with limited functionality, has proven to be a daunting and labor-intensive task [8, 72, 104, 113].

As a result, most computers run a large collection of buggy, unverified code. Worse, both OS and application code changes rapidly over time, making it difficult to decide whether a particular version of software, combined with dozens of other applications, libraries, drivers, etc., really constitutes a secure system.

Below, we examine techniques developed to cope with this state-space explosion.

2.4.2 Focusing on Security-Relevant Code

One way to simplify the decision as to whether a computer is trustworthy is to only record the identity of code that will impact the computer's security. Reducing the amount of security-relevant code also simplifies the verifier's workload in interpreting an attestation. To achieve this reduction, the platform must support multiple privilege layers, and the more-privileged code must be able to enforce isolation between itself and less-privileged code modules. Without isolation, privilege-escalation attacks (recall Section 2.1.1) become possible, enabling malicious code to potentially erase its tracks.

While layering is a time-honored technique for improving security and managing complexity [72, 104], we focus on the use of layering to simplify or interpret information given to an external party about the state of the system.

Privilege Layering. Marchesini et al. introduce a system [127] that uses privilege layering to simplify measurement information. It mixes the trusted boot and secure boot processes described in Section 2.1.1 and Section 2.2.1. The platform records the launch of a long-term core (an SELinux kernel in their implementation) which loads and verifies a policy file supplied by an administrator. The long-term core contains an *Enforcer* module that ensures that only applications matching the policy are allowed to execute. Thus, application execution follows the secure boot model. Secrets are bound to the long-term core, rather than specific applications, using trusted boot measurements as described in Section 2.2.2. If an external party can be convinced via remote attestation that the long-term core is trustworthy, then the only additional workload is to verify that the Enforcer is configured with an appropriate policy (i.e., one that satisfies the external party's requirements).

Virtualization. The model of attesting first to a more-privileged and presumably trustworthy core, and then to only a portion of the environment running thereupon, has been explored in great detail in the context of virtualization.

One of the early designs in this space was Microsoft's Next-Generation Secure Computing Base (NGSCB) [45, 56]. With NGSCB, security-sensitive operations are confined to one virtual machine (VM), while another VM can be used for general-purpose computing. The VMM is trusted to provide strong isolation between virtual machines (VMs), and hence an external party need only learn about the identity of the VMM and a particular VM, rather than all of the code that has executed in the other VMs. Specifically, handoff attacks (Section 2.1.1) are significant only prior to the VMM itself launching, and within the VM where an application of interest resides. Handoff attacks in other VMs are irrelevant. The challenge remains to this day, however, to construct a VMM where privilege-escalation attacks are not a serious concern.

Recording the initial VM image also provides a simple way of summarizing an entire software stack. With the advent of "virtual appliances," e.g., a dedicated banking VM provided by one's bank, this model can be quite promising. Terra generalized this approach to allow multiple "open", unrestricted VMs to run alongside "closed" or proprietary VMs [63]. sHype, from IBM, enforces mandatory access control (MAC) policies at the granularity of entire virtual machines [169]. Similar projects, including Nizza [180], Proxos [196], and Overshadow [44], have utilized virtualization to separate security-sensitive code from untrusted code. Unfortunately, the large TCB of such solutions makes strong assurance difficult.

Late Launch. A further challenge is that even VMM-based solutions include a considerable amount of non-security-relevant code, e.g., the BIOS, the boot loader, and various option ROMs. These values differ significantly across platforms, making it difficult for the recipient to assess the security of a particular software stack. Additionally, these entities are more privileged than the VMM (since they run before the VMM at the highest possible privilege level) and may be capable of undermining the VMM's ability to subsequently instantiate strong isolation between VMs.

To address these shortcomings, AMD and Intel extended the x86 instruction set to support a *late launch* operation with their respective Secure Virtual Machine (SVM) and Trusted eXecution Technology (TXT) (formerly codenamed LaGrande Technology) initiatives [3, 95]. Both AMD and Intel are shipping chips with these capabilities; they can be purchased in commodity computers. At a high level, a late launch operation essentially resets the platform to a known state, atomically measures a piece of code, and begins executing the code in a hardware-protected environment.

In more detail, the key new feature offered by the *SKINIT* instruction on AMD (or *SENTER* on Intel) is the ability to late launch a Virtual Machine Monitor (VMM) or Security Kernel at an arbitrary time with built-in protection against software-based attacks. When a late launch is invoked, the CPU's state is reset and memory protections for a region of code are enabled. The CPU measures the code in the memory region, extends the measurement into a PCR of the TPM, and begins executing the code. Essentially, a late launch provides many of the security benefits of rebooting the computer (e.g., starting from a clean-slate), while bypassing the overhead of a full reboot (i.e., devices remain enabled, the BIOS and bootloader are not invoked, memory contents remain intact, etc.).

We now describe AMD's implementation of late launch, followed by Intel's differences in terminology and technique.

AMD Secure Virtual Machine (SVM). To "late launch" a VMM with AMD SVM, software in CPU protection ring 0 (e.g., kernel-level code) invokes the *SKINIT* instruction, which takes a physical memory address as its only argument. AMD refers to the memory at this address as the Secure Loader Block (SLB). The first two words (16-bit values) of the SLB are defined to be its length and entry point (both must be between 0 and 64 KB).

To protect the SLB launch against software attacks, the processor includes a number of hardware protections. When the processor receives an *SKINIT* instruction, it disables direct memory access (DMA) to the physical memory pages composing the SLB by setting the relevant bits in the system's Device Exclusion Vector (DEV). It also disables interrupts to prevent previously executing code from regaining control. Debugging access is also disabled, even for hardware debuggers. Finally, the processor enters flat 32-bit protected mode and jumps to the provided entry point.

SVM also includes support for attesting to the proper invocation of the SLB. As part of the *SKINIT* instruction, the processor first causes the TPM to reset the values of the TPM's *dynamic* PCRs (i.e., PCRs 17–23) to zero,[4] and then transmits the (up to 64 KB) contents of the SLB to the TPM so that it can be measured (hashed) and extended into PCR 17. Note that software cannot invoke the command to reset PCR 17. The only way to reset PCR 17 is by executing another *SKINIT* instruction. Thus, future TPM attestations can include the value of PCR 17 to attest to the use of *SKINIT* and to the identity of the SLB loaded.

Intel Trusted Execution Technology (formerly LT). Intel's TXT is comprised of processor support for virtualization (VT-x) and Safer Mode Extensions (SMX) [95]. SMX provides support for the late launch of a VMM in a manner similar to AMD's SVM, so we focus

[4] A reboot of the platform sets the values of dynamic PCRs to -1, unlike with *static* PCRs, which are set to 0 during a reboot.

primarily on the differences between the two technologies. Instead of *SKINIT*, Intel introduces an instruction called *SENTER*.[5]

A late launch invoked with *SENTER* is comprised of two phases. First, an Intel-signed code module—called the Authenticated Code Module, or ACMod—must be loaded into memory. The platform's chipset verifies the signature on the ACMod using a built-in public key, extends a measurement of the ACMod into PCR 17, and finally executes the ACMod. The ACMod is then responsible for measuring the equivalent of AMD's SLB, extending the measurement into PCR 18, and then executing the code. In analogy to AMD's DEV protection, Intel protects the memory region containing the ACMod and the SLB from outside memory access using the Memory Protection Table (MPT). However, unlike the 64 KB protected by AMD's DEV, Intel's MPT covers 512 KB by default.

As the OSLO bootloader project noted [105], a late launch allows the chain of trust described in Section 2.1.1 to be significantly shortened. One promising design is to late launch a VMM, which prevents malicious platform firmware from attacking the VMM.

BIND [179] combined the late launch with secure information about the late-launched code's inputs and outputs, hence providing a more dynamic picture to a remote party. Since it predated the arrival of actual late launch hardware, it necessarily lacked an implementation.

The Flicker project (described in Chapter 4) found this approach could be extended even further to provide a secure execution environment *on demand*. It combined late launch with sealed storage (see Section 2.2.2) and a carefully engineered kernel module to allow the currently executing environment to be temporarily paused while a measured and isolated piece of code ran. Once completed, the previous environment could be resumed and run with full access to the platform's hardware (and hence performance). This reduced the code identity conveyed to a third party to a tiny Flicker-supplied shim (reported to be as little as 250 lines of code) and the security-relevant code executed with Flicker protections. However, the authors found that since the late launch primitive had not been designed to support frequent or rapid invocation, it introduced context-switch overheads on the order of tens or hundreds of milliseconds for practical security-sensitive code. Nonetheless, relatively simple changes to the hardware could dramatically improve this performance (see Section 4.6).

Finally, the TrustVisor project [130] attempts to strike a middle ground by employing a minimalist hypervisor to provide late-launch-like functionality to applications. It also provides a higher-level, simplified interface to TPM-like functionality, such as sealing secrets to code identity.

[5] Technically, Intel created a new "leaf" instruction called *GETSEC*, which can be customized to invoke various leaf operations (including *SENTER*).

2.4.3 Conveying Higher-Level Information

An orthogonal approach to interpreting code identity is to convert the information into a set of higher-level properties that facilitate trust judgements. This is typically accomplished either via code-level constraints or by outsourcing the problem to a third-party.

Code Constraints. As discussed in Section 2.1.2, multiple research efforts have studied mechanisms for applying static (e.g., via type checking [101] or inline reference monitors [57]) or dynamic (e.g., via hypervisors [174] or security kernels [109]) methods for conveying information about software. Attesting to code identity allows an external party to verify that the running code has been appropriately transformed or that the dynamic checker was loaded correctly. This in turn assures the external party that the code has the property (or properties) provided by the transformation or checker.

A related technique for providing higher-level information is to attest to a low-level policy enforcement mechanism and the policy that is being enforced. Jaeger et al. propose a policy-reduced integrity measurement architecture (PRIMA) [98] that enforces an integrity policy called Clark Wilson-Lite (CW-Lite) [177]. CW-Lite relaxes the original Clark-Wilson [47] requirements that complete, formal assurance of programs is required, and that all interfaces must have filters. Instead, only interfaces accepting low-integrity inputs must have filters. PRIMA supports the notion of trusted and untrusted subjects, and extends IBM's IMA [170] to also measure the Mandatory Access Control (MAC) policy, the set of trusted subjects, and the code-subject mapping (e.g., the active user or role when a program is run). Verification of an attestation produced on a PRIMA-capable system involves additional checks. Verification fails if any of the following occur: (1) an untrusted program executes, or (2) a low integrity flow enters a trusted program without first being filtered. PRIMA is prototyped using SELinux.

Outsourcing. Another approach is to outsource the problem of interpreting code identity to a third party. Terra [63] took an initial step in this direction, as the authors suggest that clients obtain certificates from their software providers that map hash values to software names and/or versions. By including these certificates with their attestation, the client simplifies the verifier's interpretation task (i.e., the verifier no longer needs to have its own database for mapping hash values to software packages, assuming the verifier trusts the PKI used by the software vendors). Subsequent work takes this idea much further [84, 168]. The client contacts a third-party who certifies that the client's software satisfies a much higher-level property, e.g., the client's software will never leak sensitive data. The client then presents this certificate to the verifier. Assuming the verifier trusts this third-party, it can easily conclude

that the client possesses the certified property. Unfortunately, most work in this area does not specify how the third party decides whether a particular piece of software provides a given property.

2.5 Roots of Trust

Trust in any system needs a foundation or a *root of trust*. Here, we discuss the roots of trust that have been proposed or deployed. Typically, the root of trust is based on the secrecy of a private key that is embedded in hardware; the corresponding public key is certified by the hardware's manufacturer. As we discuss, some systems further rely on a piece of code that must execute in the early boot process for their root of trust. We also discuss schemes where the root of trust is established by the properties of the physical hardware itself.

We divide this section as follows: 1) general-purpose devices with significant resistance to physical tampering, 2) general-purpose devices without significant physical defenses, 3) special-purpose minimal devices, 4) research solutions that attempt to instantiate a root of trust without custom hardware support, and 5) cryptographic techniques for evaluating specific functions, rather than creating generic roots of trust.

2.5.1 General-Purpose Tamper-Resistant and Tamper-Responding Devices

We first discuss commercial solutions available today. Relatively few products have achieved widespread commercial success, since tamper-resistant devices require costly manufacturing processes. We then discuss research projects that developed many of the design ideas manifested in today's commercial solutions. In all of these systems, the hardware stores a secret private key, and the manufacturer digitally signs a certificate of the corresponding public key. The certificate forms the root of trust that a verifier uses to establish trust in the platform.

2.5.1.1 Commercial Solutions

IBM offers a family of general-purpose cryptographic co-processors with tamper-resistant and tamper-responding properties, including the PCI-based 4758 [100, 185, 186] and the PCI-X-based 4764/PCIXCC [15, 93]. These devices include packaging for resisting and responding to physical penetration and fluctuations in power and temperature. Batteries provide power that enables an active response to detected tampering, in the form of immediate erasure of the area where internal secrets are stored and permanently disabling the device. Some

of these devices include support for online battery replacement, so that the lifetime of these devices is not constrained by the lifetime of a battery.

Smart cards are also widely deployed. A private key, typically used for authentication, resides solely in the smart card, and all private key operations take place within the card itself. In this way the cards can be used to interact with potentially untrusted terminals without risking key exposure. Gobioff et al. discuss the need for an on-card trusted path to the user, since an untrusted terminal can display one thing to the user and perform a different transaction with the card itself (e.g., doubling the amount of a transaction) [71]. Smart cards are also discussed in Section 2.8.

2.5.1.2 Research Projects

μABYSS [209] and Citadel [211] are predecessors of the modern IBM designs, placing a CPU, DRAM, FLASH ROM, and battery-backed RAM (BBRAM) within a physically tamper-resistant package. Tampering causes erasure of the BBRAM, consequently destroying the keys required to decrypt the contents of DRAM. The Dyad secure co-processor [218] also presents some design elements visible today in IBM's devices. Only signed code from a trusted entity will be executed, and bootstrapping proceeds in stages. Each stage checks its integrity by comparing against a signature stored in the device's protected non-volatile memory.

The XOM [123] and AEGIS[6] [195] designs do not trust the operating system, and include native support for partitioning cache and memory between mutually distrusting programs. The AEGIS [195] design generates secrets (for use as encryption keys) based on the physical properties of the CPU itself (e.g., logic delays). Physical tampering will impact these properties, rendering the encryption keys inaccessible.

The Cerium processor design is an attempt at providing similar properties while remaining a largely open system [41]. Cerium relies on a physically tamper-resistant CPU with a built-in private key. This key is then used to encrypt sensitive data before it is sent to memory. Cerium depends on a trusted micro-kernel to manage address space separation between mutually distrusting processes, and to manage encryption of sensitive data while it resides in untrusted DRAM.

Lee et al. propose the Secret Protected (SP) architecture for virtual secure coprocessing [120]. SP proposes hardware additions to standard CPUs in the form of a small key store, encryption capabilities at the cache-memory interface, new instructions, and platform changes to support a minimalistic trusted path. These facilities enable a Trusted Software

[6] Two relevant research efforts have used the name AEGIS. One is that of Arbaugh et al. [13] discussed in Section 2.1.1. The other is by Suh et al. [195] and is discussed in this section.

Module to execute with direct hardware protection on the platform's primary CPU. This module can provide security-relevant services to the rest of the system (e.g., emulate a TPM's functionality), or it can implement application-specific functionality. Data is encrypted and integrity protected when it leaves the CPU for main memory, with the necessary keys residing solely within the CPU itself. SP pays considerable attention to the performance as well as security characteristics of the resulting design.

2.5.2 General-Purpose Devices Without Dedicated Physical Defenses

Here we discuss devices that are designed to help increase the security of software systems, but do not incorporate explicit physical defense measures. In practice, the degree of resilience to physical compromise varies widely. For example, consider the differences in physically attacking a device 1) on a daughter card that can be readily unplugged and interposed on, 2) soldered to the motherboard, 3) integrated with the "super-IO" chip, and 4) on the same silicon as the main CPU cores. The best examples for commodity platforms today are those equipped with a Trusted Platform Module (TPM), its mobile counterpart, the Mobile Trusted Module (MTM [55, 201]), or a smart card.

TPM-equipped Platforms. The TPM chip is a hardware device, but it does not employ any specific tamper resistance. Trust in the TPM stems from three roots of trust, specifically the roots of trust for Storage, Reporting, and Measurement. Trusted storage is provided by an encryption key that permanently resides within the TPM in non-volatile RAM (see Section 2.2.2.2). The root for reporting (or communicating measurements to an external party) can be protected by the TPM's storage facilities. Finally, TPM measurement depends an immutable part of platform firmware called the Core Root of Trust for Measurement, which initializes the TPM when a platform first boots up.

MTM-equipped Platforms. For space reasons, we consider here only one *profile* from the MTM specification [201], that of the Remote Owner MTM. Trust stems from four distinct roots of trust, specifically the roots of trust for Storage, Enforcement, Reporting, and Verification. These roots of trust represent security preconditions required for the MTM to initialize successfully [55]. Unlike the TPM, an MTM may be implemented entirely in software, although a device secret must be protected so that it can be used to provide secure storage facilities. Similar to the TPM, the other roots can use keys that are protected by secure storage. The root for execution typically makes use of the isolated execution features of the platform's main CPU, e.g., ARM TrustZone [14] or TI M-Shield [16]. Boot integrity is provided using a *secure boot* model (Section 2.2.1).

Smart Cards. Smart cards and SIM cards may not have any active tamper response mechanisms; instead, they often attempt to protect a secret key through techniques such as hardware obfuscation [220]. Private key operations are performed within the card to protect the card's secrets from being exposed to untrusted terminals.

2.5.3 Special-Purpose Minimal Devices

Several research projects have considered the utility of special-purpose security hardware. In general, this minimalistic approach works for some applications, but the limited functionality will exclude many applications that depend on reporting exactly what code is currently executing. Characterizing more precisely what functionality is needed in secure hardware for various classes of applications is still an open area of research.

Chun et al. observe that much of the complexity in Byzantine-Fault-Tolerant protocols arises from an adversary's ability to lie *differently* to each legitimate participant [46]. They show that the ability to attest to an append-only log can prevent such duplicity, and hence greatly reduces the complexity and overhead of these protocols. Following up on this work, Levin et al. [122] show that the same property can be achieved with a much simpler primitive, namely the ability to attest to the value of a counter. They informally argue that this is simplest primitive that can provide this property, and they show that an attested counter can be used in a range of applications, including PeerReview and BitTorrent.

2.5.4 Research Solutions Without Hardware Support

The research community has proposed mechanisms to establish a root of trust based solely on the properties of the physical hardware, i.e., without special hardware support. The key idea in *software-based attestation* is to have code compute a checksum over itself to verify its integrity [70, 106, 175, 176, 192]. The verifier checks the result of the checksum and also measures the time taken to compute it. If an adversary attempts to interfere with the checksum computation, the interference will slow the computation, and this timing deviation can be detected by the verifier. Software-based attestation requires a number of strong assumptions, including the need for the verifier to have intimate knowledge of the hardware platform being verified, i.e., the verifier must know the platform's CPU make and model, clock speed, cache architecture, etc. In comparison with hardware-based techniques, the resulting security properties are similar to those of a late launch on a platform such as AMD SVM [3] or Intel TXT [95] (see Section 2.4.2). Secure storage remains a challenge as we discuss below.

The earliest proposal in this area is due to Spinellis [192], who proposes to use a timed self-checksumming code to establish a root of trust on a system. In the same vein, Kennel and Jamieson propose to use hardware side-effects to authenticate software [106]. Seshadri et al. implement a timed checksum function on embedded systems as well as on PCs [175, 176]. Giffin et al. propose the use of self-modifying code to strengthen self-checksumming [70].

Attacks have recently been proposed against weakened versions of software-based attestation mechanisms [38, 214]; however, these attacks are primarily based on implementation flaws, rather than fundamental limitations of the approach. Even so, additional formalism is needed to create true confidence in software-based attestation.

Long-term secure storage is also an open challenge for software-based attestation. This is because software-based attestation has no dedicated or hardware-protected storage for integrity measurements or secrets bound to integrity measurements. Thus, if such properties are desired, they must be engineered in software. However, there are fundamental limitations to the types of storage that can be protected long-term (e.g., across a power cycle) without a root of trust for storage (e.g., an encryption key available only to the trusted code that runs as part of the software-based attestation).

One line research has avoided this dependence on hardware properties by focusing on auditing the results of specific function evaluations. Audit-based solutions [26, 146] typically require the client (or randomly selected workers) to recalculate some portion of the work done by untrusted workers. This may be infeasible for resource-constrained clients and often relies on some fraction of the workers to be honest, or at least non-colluding.

2.5.5 Cryptographic Protocols

In the cryptographic community, the idea of outsourcing expensive cryptographic operations to a semi-trusted device has a long history. Chaum and Pedersen define the notion of *wallets with observers* [40], a piece of secure hardware installed by a third party, e.g. a bank, on the client's computer to "help" with expensive computations. The hardware is not trusted by the client who retains assurance that it is performing correctly by analyzing its communication with the bank. Hohenberger and Lysyanskaya formalize this model [90], and present protocols for the computation of modular exponentiations (arguably the most expensive step in public-key cryptography operations). Their protocol requires the client to interact with *two* non-colluding servers. Other work targets specific function classes, such as one-way function inversion [76].

Recent advances in fully-homomorphic encryption [69, 202] allow a worker to compute arbitrary functions over encrypted data, but they do not suffice to provide outsourceable

computing. Indeed, fully-homomorphic encryption provides *no guarantee* that the worker performed the correct computation. While our solution does employ fully-homomorphic encryption, we combine it with other techniques to provide verifiability.

The theoretical community has devoted considerable attention to the verifiable computation of arbitrary functions. *Interactive proofs* [17, 75] are a way for a powerful (e.g. super-polynomial) prover to (probabilistically) convince a weak (e.g. polynomial) verifier of the truth of statements that the verifier could not compute on its own. As it is well known, the work on interactive proofs lead to the concept of *probabilistically checkable proofs* (PCPs), where a prover can prepare a proof that the verifier can check in only very few places (in particular only a constant number of bits of the proofs needed for NP languages). Notice, however, that the PCP proof might be very long, potentially too long for the verifier to process. To avoid this complication, Kilian proposed the use of efficient arguments[7] [110, 111] in which the prover sends the verifier a short commitment to the entire proof using a Merkle tree. The prover can then interactively open the bits requested by the verifier (this requires the use of a collision-resistant hash function). A non-interactive solution can be obtained using Micali's CS Proofs [140], which remove interaction from the above argument by choosing the bits to open based on the application of a random oracle to the commitment string. Researchers have attempted to provide non-interactive, efficient protocols by combining PCP proofs with Private-Information Retrieval (PIR) schemes [4], but Dwork et al. demonstrate that these proposals are unsound and that this approach contains inherent difficulties [52]. In more recent work, which still uses some of the standard PCP machinery, Goldwasser et al. [74] show how to build an interactive proof to verify arbitrary polynomial time computations in almost linear time. They also extend the result to a non-interactive argument for a restricted class of functions.

Therefore, if we restrict our attention to non-interactive protocols, the state of the art offers either Micali's CS Proofs [140] which are arguments that can only be proven in the random oracle model, or the arguments from [74] that can only be used for a restricted class of functions.

[7] We follow the standard terminology: an *argument* is a computationally sound proof, i.e. a protocol in which the prover is assumed to be computationally bounded. In an argument, an infinitely powerful prover can convince the verifier of a false statement, as opposed to a proof where this is information-theoretically impossible or extremely unlikely.

2.6 Validating the Process

Bootstrapping trust can only be effective if we can validate the hardware, software, and protocols involved. Below we summarize the (relatively few) efforts in this direction.

From a hardware perspective, Smith and Austel discuss efforts to apply formal methods to the design of secure coprocessors [181, 185]. They also state formal security goals for such processors. Bruschi et al. use a model checker to find a replay attack in the TPM's Object Independent Authorization Protocol (OIAP) [36]. They also propose a countermeasure to address their attack, though it requires a TPM design change.

Taking a more empirical approach, Chen and Ryan identify an opportunity to perform an offline dictionary attack on weak TPM authorization data, and propose fixes [42]. Sadeghi et al. performed extensive testing on TPMs from multiple vendors to evaluate their compliance with the specification [167]. They find a variety of violations and bugs, including some that impact security. Finally, starting from the TPM specification, Gürgens et al. developed a formal automata-based model of the TPM [83]. Using an automated verification tool, they identify several inconsistencies and potential security problems.

At the software level, Kauer notes several implementation flaws in trusted computing applications [105]. These include bootloaders that fail to appropriately measure software before loading it, and BIOS software that allows flash updates without validation.

At the protocol layer, Smith defines a logic for reasoning about the information that must be included in platform measurements to allow a verifier to draw meaningful conclusions [183]. Datta et al. use the Logic of Secure Systems (LS^2) [48] to formally define and prove the code integrity and execution integrity properties of the static and dynamic TPM-based attestation protocols. The logic also helps make explicit the invariants and assumptions required for the security of the protocols.

2.7 Applications

Clearly, many applications benefit from the ability to bootstrap trust in a computer. Rather than give an exhaustive list, we focus on applications deployed in the real world, and a handful of particularly innovative projects in academia.

2.7.1 Real World

Code Access Security in Microsoft .NET. Microsoft's Code Access Security is intended to prevent unauthorized *code* from performing privileged actions [141]. The Microsoft .NET

Common Language Runtime (CLR) maintains *evidence* for *assemblies* of code and uses these to determine compliance with a security policy. One form of evidence is the cryptographic hash of the code in question. This represents one of the more widely deployed systems that supports making security-relevant decisions based purely on the identity of code as represented by a cryptographic hash of that code.

Bitlocker. One of the most widely-used applications of trust bootstrapping is BitLocker [142], Microsoft's drive encryption feature, which first appeared in the Windows Vista OS. Indeed, BitLocker's dependence on the presence of a v1.2 TPM likely helped encourage the adoption of TPMs into the commodity PC market. The keys used to encrypt and authenticate the hard-drive's contents are sealed (see Section 2.2.2) to measurements taken during the computer's initial boot sequence. This ensures that malware such as boot-sector viruses and rootkits cannot hijack the launch of the OS nor access the user's files. These protections can be supplemented with a user-supplied PIN and/or a secret key stored on a USB drive.

Trusted Network Connect (TNC). TNC is a working group with goals including strengthening network endpoints. TNC supports the use of attestation to perform Network Access Control. Thus, before a computer can connect to the network, it must pass integrity checks on its software stack, as well as perform standard user authentication checks. An explicit goal is to give non-compliant computer systems an avenue for remediation. Existing open source solutions have already been tested for interoperability with promising results [205].

Secure Boot on Mobile Phones. Mobile phones (and other embedded devices) have long benefitted from a secure boot architecture. Until recently, these devices served very specific purposes, and the degree of control afforded to mobile network operators by a secure boot architecture helped to ensure dependable service and minimize fraud. Even many modern smartphones with support for general-purpose applications employ rich capability-based secure architectures whose properties stem from secure boot. For example, Symbian Signed [55] is the interface to getting applications signed such that they can be installed and access certain capabilities on smartphones running the Symbian OS. Apple's iPhone OS employs a similar design.

2.7.2 Research Proposals

Multiple projects have considered using secure hardware to bootstrap trust in a traditional "Trusted Third Party". Examples include certifying the behavior of the auctioneer in an online auction [158], protecting the private key of a Certificate Authority [131], protecting the various private keys for a Kerberos Distribution Center [96].

Given the ever increasing importance of web-based services, multiple research efforts have studied how to bootstrap greater assurance in public web servers. In the WebALPS project, building on the IBM 4758, Jiang et al. enhanced an SSL server to provide greater security assurance to a web client [99, 182]. A "guardian" program running on the secure coprocessor provides data authenticity and secrecy, as well as safeguarding the server's private SSL keys. This approach helps protect both the web client and the web server's operator from insider attacks. In the Spork project, Moyer et al. consider the techniques needed to scale TPM-based attestation to support a high-performance web server [147]. They also implement their design by modifying the Apache web server to provide attested content and developing a Firefox extension for validating the attestations.

Of course, other network protocols can benefit from bootstrapped trust as well. For example, the Flicker project enhanced the security of SSH passwords while they are handled by the server (see Section 4.4.3.1). With a Flicker-enhanced SSH server, the client verifies an attestation that allows it to establish a secure channel to an isolated code module on the server. By submitting its password over this channel, the client can ensure that only a tiny piece of code on the server will ever see the password, even if other malware has infected the server. On a related note, the BIND project [179] observed that by binding bootstrapped code to its inputs, they could achieve a transitive trust property. For example, in the context of BGP, each router can verify that the previous router in the BGP path executed the correct code, made the correct decisions given its input, and *verified the same information about the router before it*. The last property ensures that by verifying only the previous router in the chain, the current router gains assurance about the entire BGP path.

Researchers have also investigated the use of bootstrapped trust in the network itself. Ramachandran et al. propose imbuing packets with the provenance of the hosts and applications that generated them [161]. Unfortunately, these packet markings are not secured, so the system must assume that the entire network is trusted and that all hosts have deployed the system in a tamper-proof fashion. Garfinkel et al. noted that secure hardware might help defend against network-based attacks [64].

However, the first design and implementation of this idea came from Baek and Smith, who describe an architecture for prioritizing traffic from privileged applications [18]. Using a TPM, clients attest to the use of an SELinux kernel equipped with a module that attaches Diffserv labels to outbound packets based on an administrator's network policy. This system requires a large TCB (i.e., an entire Linux kernel) and universal deployment. Gummadi et al. propose the Not-A-Bot system [82], which tries to distinguish human-generated traffic from bot-driven traffic. They attest to a small client module that tags outgoing packets generated

within one second of a keystroke or mouse click. Through trace-driven experiments, the authors show that the system can significantly reduce malicious traffic. However, the system only considers application-level attacks, i.e., the network is assumed to be uncongested. Thus, the server is responsible for verifying client attestations, which is less practical for applications such as combating network-level DDoS attacks or super-spreader worms. The system works well for human-driven application-specific scenarios, but it is difficult to adapt it to services that are not primarily human-driven, such as NTP, transaction processing, network backup, or software update servers. Rather than use a TPM, Feng and Schluessler propose, at a high-level, using Intel's Active Management Technology to provide information on the machine's state to network elements by introspecting on the main CPU's activities [59]. They do not focus on conveying this information efficiently, nor do they provide a full system design and implementation. Lastly, Dixon et al. propose pushing middle-box functionality, such as NAT and QoS to endhosts, using trusted computing as a foundation [51]. This is orthogonal, but potentially complementary, to Chapter 5's goal of conveying host-based information to network elements.

Finally, Sarmenta et al. observe that a trusted monotonic counter can be used in a wide variety of applications, including count-limited objects (e.g., keys that can only be used a fixed number of times), digital cash, and replay prevention [171]. While the TPM includes monotonic counter functionality, the specification only requires it to support a maximum of four counters, and only one such counter need be usable during a particular boot cycle. Thus, they show how to use a log-based scheme to support an arbitrary number of simultaneous counters. They also design a more efficient scheme based on Merkle trees [137], but this scheme would require modifications to the TPM, so that it could securely store the tree's root and validate updates to it.

2.8 Human Factors & Usability

Most existing work in attestation and trusted computing focuses on interactions between two computing devices. This section treats a higher goal – that of convincing the human operator of a computer that it is in a trustworthy state. These solutions sort into two categories: those where the user is in possession of an additional trustworthy device, and those based solely on the human's sensory and cognitive abilities. An additional, though somewhat orthogonal, category concerns techniques to *pair* two trustworthy devices, e.g., to establish a secure channel between two cellphones.

2.8.1 Trustworthy Verifier Device

To help a human establish trust in a computer, Itoi et al. describe a smart card-based solution called sAEGIS [97]. sAEGIS builds on the AEGIS [13] secure boot (see Section 2.2.1) but changes the source of the trusted software list. Instead of being preconfigured by a potentially untrustworthy administrator, sAEGIS allows the smart card to serve as the repository of trusted software list. Thus, a user can insert her smart card into an untrusted computer and reboot. If booting is successful, the resulting environment conforms to the policy encoded on her smart card, i.e., the executing software appears in the list stored in the smart card. Of course, the user must establish through some out-of-band mechanism that the computer indeed employs the sAEGIS system. Otherwise, it might simply ignore the user's smart card.

To help humans verify that a platform is trustworthy, Lee et al. propose the addition of a simple multi-color LED and button to computers to enable interaction with a Trusted Software Module [120]. A complete architecture and implementation for these simple interfaces remains an open problem.

The Bumpy [135] system is an architecture to provide a trusted path for sending input to web pages from a potentially malicious client-side host. A user is assumed to possess a trustworthy smartphone and an encryption-capable keyboard. Attestation is used to convince the smartphone that the user's input is being encrypted in an isolated code module.

Chapter 3 considers the challenges faced by a human user trying to learn the identity (e.g., authentic public key) of the TPM in a specific computer. This highlights the risk of a Cuckoo Attack, in which malware on the user's computer forwards the user's attestation request to another attacker-controlled system that conforms to the expected configuration. Thus, the user concludes her computer is safe, despite the presence of malware.

2.8.2 Using Your Brain to Check a Computer

Roots of trust established based on timing measurements (Section 2.5.4) can potentially be verified by humans. Franklin et al. propose personally verifiable applications as part of the PRISM architecture for human-verifiable code execution [61]. The person inputs a challenge from a physical list and then measures the length of time before the application produces the correct response (also on the list). Verification amounts to checking that the response is correct and that it arrived in a sufficiently short period of time. However, the timing-based proposals on which these solutions rest (Section 2.5.4) still face several challenges.

2.8.3 Pairing Two Trustworthy Devices

Considerable work has studied how to establish secure communication between two devices. Proposals use infrared [21], visual [134, 172], and audio channels [77, 189], as well as physical contact [193], shared acceleration [128], and even the electrical conductivity of the human body [190]. Unlike this work in which one device may be infected with malware, in these systems, the two devices are trusted, and the adversary is assumed to be an external entity.

In the realm of access control, researchers have studied a related problem known as the Chess Grandmaster Problem, Mafia Fraud, or Terrorist Fraud [5, 29], in which an adversary acts as a prover to one honest party and a verifier to another party in order to obtain access to a restricted area. Existing solutions rely on distance bounding [29, 32], which, as explained in Section 3.2.1, is ineffective for a TPM, or employ radio-frequency hopping [5] which is also infeasible for the TPM.

2.9 Limitations

When it comes to bootstrapping trust in computers, there appear to be significant limitations on the types of guarantees that can be offered against software and hardware attacks. We summarize these limitations below to alert practitioners to the dangers and to inspire more research in these areas.

2.9.1 Load-Time Versus Run-Time Guarantees

As described in Section 2.1, existing techniques measure software when it is first loaded. This is the easiest time to obtain a clean snapshot of the program, before, for example, it can create temporary and difficult to inspect local state. However, this approach is fundamentally "brittle", since it leaves open the possibility that malware will exploit the loaded software. For example, if an attacker exploits a buffer overflow in the loaded software, no measurement of this will be recorded. In other words, the information about this platform's state will match that of a platform that contains pristine software. Thus, any vulnerability in the attesting system's software potentially renders the attestation protocol meaningless. While Section 2.1.2 and Section 2.4.3 surveyed several attempts to provide more dynamic properties, the fact remains that they all depend on a static load-time guarantee. This reinforces the importance of minimizing the amount of software that must be trusted and attested to, since smaller code tends to contain fewer bugs and be more amendable to formal analysis.

2.9.2 Hardware Attacks

As discussed in Section 2.5, protection against hardware attacks has thus far been a trade-off between cost (and hence ubiquity) and resilience [12]. Even simple hardware attacks, such as connecting the TPM's reset pin to ground, can undermine the security offered by this inexpensive solution [105]. Another viable attack is to physically remove the TPM chip and interpose on the LPC bus that connects the TPM to the chipset. The low speed of the bus makes such interposition feasible and would require less than one thousand dollars in FPGA-based hardware. Tarnovsky shows how to perform a more sophisticated hardware attack [197], but this attack requires costly tools, skills, and equipment including an electron microscope. An important consequence of these attacks is that applications that rely on widespread, commodity secure hardware, such as the TPM, must align the application incentives with those of the person in direct physical control of the platform. Thus, applications such as BitLocker [142], which help a user protect her files from attackers, are far more likely to succeed than applications such as DRM, which attempt to restrict users' capabilities by keeping secrets from them.

This also makes the prospect of kiosk computing daunting. Kiosks are necessarily cost-sensitive, and hence unlikely to invest in highly-resilient security solutions. Combining virtualization with a TPM can offer a degree of trust in a kiosk computer [65], but only if the owner of the computer is trusted not to tamper with the TPM itself. Of course, other physical attacks exist, including physical-layer keyboard sniffers and screen-scrapers. The roots of trust we consider in this chapter are unlikely to ever cope with such attacks.

Similar problems plague cloud computing and electronic voting applications. When a client entrusts a cloud service with sensitive data, it can bootstrap trust in the cloud's software using the techniques we have described, but it cannot verify the security of the cloud's hardware. Thus, the client must trust the cloud provider to deploy adequate physical security. Likewise, a voter might use a trusted device to verify the software on an electronic voting machine, but she still has no guarantee regarding the machine's physical security.

Another concern is hardware attacks on secrets stored in memory. Even if a combination of hardware and software protections can protect a user's secrets while the computer is active, recent research by Halderman et al. has shown that data in RAM typically persists for a surprisingly long time (seconds or even minutes) after the computer has been powered down [85]. Hence, an attacker who has physical access to the computer may be able to read these secrets directly out of the computer's memory.

2.10 Additional Reading

With a focus on the IBM 4758, Smith's book [184] provides a thorough survey of early work in the design and use of secure coprocessors. Balacheff et al.'s book documents the early design of the TPM [19], but it is now mostly superseded by Grawrock's more recent book [79], which covers the motivation and design of the TPM, as well as Intel's late launch and virtualization support. Challener et al.'s book [39] touches on similar topics but focuses primarily on Trusted Computing from a developer's perspective, including guidance on writing TPM device drivers or applications that interface with the Trusted Software Stack (TSS). Mitchell's book [144] contains a collection of articles surveying the general field of Trusted Computing, providing useful overviews of topics such as NGSCB and DAA.

2.11 Summary

In this chapter, we organize and clarify extensive research on bootstrapping trust in commodity systems. We identify inconsistencies (e.g., in the types of attacks considered by various forms of secure and trusted boot), and commonalities (e.g., all existing attempts to capture dynamic system properties still rely in some sense on static, load-time guarantees) in previous work. We also consolidate the various types of hardware support available for bootstrapping trust. This leads us to the observation that applications based on low-cost, non-tamper-resistant hardware (e.g., the TPM), must align their incentives with those of the computer owner, suggesting that applications that help the user protect her own secrets or check her computer for malware are more likely to succeed than applications that try to hide information from her.

Chapter 3

Bootstrapping Trust in a Commodity Computer

Before entrusting a computer with a secret, a user needs some assurance that the computer can be trusted. Without such trust, many tasks are currently impossible. For example, if Alice does not trust her PC, then she cannot log in to any websites, read or edit confidential documents, or even assume that her browsing habits will remain private. While the need to bootstrap trust in a machine is most evident when traveling (and hence using computers with which the user has no prior relationship), this problem also arises more generally. For example, Alice might wish to check her email on a friend's computer. Alice may not even be able to say for sure whether she can trust her own personal computer.

One way to bootstrap trust in a computer is to use secure hardware mechanisms to monitor and report on the software state of the platform. Given the software state, the user (or an agent acting on the user's behalf) can decide whether the platform should be trusted. Due to cost considerations, most commodity computers do not include a full-blown secure coprocessor such as the IBM 4758 [186]. Instead, the move has been towards cheaper devices such as the Trusted Platform Module (TPM) [200]. The cost reduction is due in part to the decision to make the TPM secure only against software attacks. As a consequence, a TPM in the physical possession of an adversary cannot be trusted.

With appropriate software support, the TPM can be used to measure and record each piece of software loaded for execution, and to securely convey this information (via an attestation) to a remote party [170, 200]. With hardware support for a *dynamic root of trust*, included in the most recent CPUs from AMD and Intel, the attestation from the TPM can be simplified to attest to the secure, isolated execution of a particular piece of software (see Sec-

tion 2.4.1). With either approach, the resulting attestations can be verified by a user's trusted device, such as a cellphone or a special-purpose USB fob [203]. Thus, the TPM can be used to establish trust in the software on a machine.

However, the question remains: How do we bootstrap trust in the TPM itself? Surprisingly, neither the TPM specifications nor the academic literature have considered this problem. Instead, it is assumed that the user magically possesses the TPM's public key. While this assumption dispenses with the problem, it does not truly solve it, since in real life the user does not typically receive authentic public keys out of the blue. Without the TPM's public key, the user cannot determine if she is interacting with the desired local TPM or with an adversarially-controlled TPM. For example, in a cuckoo attack, malware on the local machine may forward the user's messages to a remote TPM that the adversary physically controls. Thus, the user cannot safely trust the TPM's attestation, and hence cannot trust the computer in front of her.

As a result, we need a system to allow a conscientious user to bootstrap trust in the *local* TPM, so that she can leverage that trust to establish trust in the entire platform.

Contributions. In this chapter, we make the following contributions: **(1)** We formally define (using predicate logic) the problem of bootstrapping trust in a platform. **(2)** We show how the model captures the cuckoo attack, as well as how it suggests potential solutions. **(3)** We give sample instantiations of each type of solution and discuss their advantages and disadvantages. **(4)** We recommend improvements for future platforms that aspire to be trusted.

3.1 Problem Definition

In this section, we give an informal description of the problem, followed by a more rigorous, formal definition.

In this chapter, we focus on a slightly abstracted model of the Trusted Platform Module (TPM). In particular, we model it as a security chip equipped with a public/private keypair $\{K_{\text{TPM}}, K_{\text{TPM}}^{-1}\}$ and a set of PCRs. The TPM's manufacturer provides the TPM with an Endorsement Certificate. The Endorsement Certificate certifies that the TPM is a genuine, hardware TPM and serves to authenticate the TPM's public key K_{TPM}. Chapter 2 describes the lower-level details hidden by this abstraction.

3.1.1 Informal Problem Description

Our high-level goal is to establish trust in a potentially compromised computer, so that a user can perform security-sensitive tasks. To achieve this goal, we must assume the user already trusts someone or something, and then leverage that trust to establish trust in the computer.

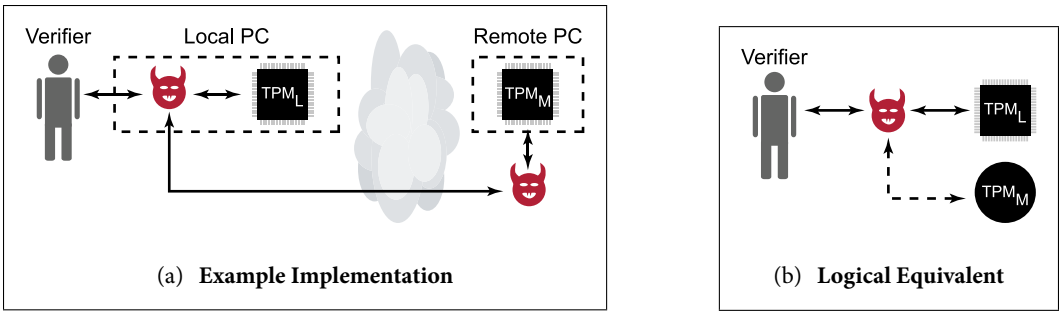


Figure 3.1: **The Cuckoo Attack.** *In one implementation of the cuckoo attack (a), malware on the user's local machine sends messages intended for the local TPM (TPM_L) to a remote attacker who feeds the messages to a TPM (TPM_M) inside a machine the attacker physically controls. Given physical control of TPM_M, the attacker can violate its security guarantees via hardware attacks. Thus, at a logical level (b), the attacker controls all communication between the verifier and the local TPM, while having access to an oracle that provides all of the answers a normal TPM would, without providing the security properties expected of a TPM.*

Specifically, we make two initial trust assumptions. First, we assume the user has a mobile, trusted device, such as a cellphone, or a special-purpose USB fob [203] that can compute and communicate with the computer. This device is assumed to be trusted in part due to its limited interface and functionality,[1] so it cannot be used for general security-sensitive tasks. We also assume the user trusts someone (potentially herself) to vouch for the physical integrity of the local machine. Without this assumption (which may not hold for kiosk computers), it is difficult to enable secure, general-purpose computing. Fortunately, humans are relatively good at protecting their physical belongings (as opposed to virtual belongings, such as passwords). Furthermore, the assumption is true relative to Internet-based attackers.

Ideally, from these two trust assumptions (a trustworthy verifier device and a physically secure local computer), we would establish trust in the secure hardware (TPM) in the local computer. Trust in the TPM could then be used to establish trust in the software on the computer. Unfortunately, there is currently no way to connect our trust assumptions to trust in the local TPM. When a user walks up to a computer, she has no reliable way of establishing the identity (public key) of the TPM inside the computer. As a result, she may fall victim to what we call a cuckoo attack.

In a *cuckoo attack*,[2] the adversary convinces the user that a TPM the adversary physically controls in fact resides in the user's own local computer. Figure 3.1(a) illustrates one possible

[1] Arguably, this assumption may not hold for current smartphones.

[2] The cuckoo bird replaces other birds' eggs with its own. The victim birds are tricked into feeding the cuckoo chick as if it were their own. Similarly, the attacker "replaces" the user's trusted TPM with his own TPM, leading the user to treat the attacker's TPM as her own.

Predicates

Predicate	Meaning
$\texttt{TrustedPerson}(p)$	User trusts person p.
$\texttt{PhysSecure}(c)$	Computer c is physically secure.
$\texttt{SaysSecure}(p, c)$	Person p says computer c is physically secure.
$\texttt{Trusted}_\texttt{C}(c)$	Computer c is trusted.
$\texttt{Trusted}_\texttt{T}(t)$	TPM t is trusted.
$\texttt{On}(t, c)$	TPM t resides on computer c.
$\texttt{CompSaysOn}(c, t)$	Computer c says TPM t is installed on computer c.

Axioms

1. $\forall p, c \;\; \texttt{TrustedPerson}(p) \wedge \texttt{SaysSecure}(p, c) \rightarrow \texttt{PhysSecure}(c)$
2. $\forall t, c \;\; \texttt{On}(t, c) \wedge \neg\, \texttt{PhysSecure}(c) \rightarrow \neg\, \texttt{Trusted}_\texttt{T}(t)$
3. $\forall t, c \;\; \texttt{On}(t, c) \wedge \texttt{PhysSecure}(c) \rightarrow \texttt{Trusted}_\texttt{T}(t)$
4. $\forall t, c \;\; \texttt{On}(t, c) \wedge \texttt{Trusted}_\texttt{T}(t) \rightarrow \texttt{Trusted}_\texttt{C}(c)$
5. $\forall t, c \;\; \texttt{On}(t, c) \wedge \neg\, \texttt{Trusted}_\texttt{T}(t) \rightarrow \neg\, \texttt{Trusted}_\texttt{C}(c)$
6. $\forall c, t \;\; \texttt{CompSaysOn}(c, t) \rightarrow \texttt{On}(t, c)$

Figure 3.2: **Trust Model.** *The predicates describe relevant properties of the system, while the axioms encode facts about the domain.*

Assumption	Encoding
1. Alice trusts herself.	$\texttt{TrustedPerson}(\text{Alice})$
2. Alice says her computer C is physically secure.	$\texttt{SaysSecure}(\text{Alice}, C)$
3. The adversary controls machine M containing TPM_M.	$\texttt{On}(\text{TPM}_M, M)$
4. M is not physically secure.	$\neg\, \texttt{PhysSecure}(M)$
5. Malware on Alice's machine C causes it to say that TPM_M is installed on C.	$\texttt{CompSaysOn}(C, \text{TPM}_M)$

Figure 3.3: **Trust Model Assumptions.** *We encode our assumptions about the situation in predicates.*

implementation of the cuckoo attack. Malware on the user's local machine proxies the user's TPM-related messages to a remote, TPM-enabled machine controlled by the attacker. The attacker's TPM_M can produce an Endorsement Certificate certifying that the TPM's public key K_{TPM_M} comes from an authentic TPM. The attacker's computer then faithfully participates in the TPM protocol, and it provides an attestation that trusted software has been loaded correctly.

(1)	$\texttt{TrustedPerson}(\text{Alice})$	Assumption 1
(2)	$\texttt{SaysSecure}(\text{Alice}, C)$	Assumption 2
(3)	$\texttt{PhysSecure}(C)$	Axiom 1: (1), (2)
(4)	$\texttt{CompSaysOn}(C, \text{TPM}_M)$	Assumption 5
(5)	$\texttt{On}(\text{TPM}_M, C)$	Axiom 6: (4)
(6)	$\texttt{Trusted}_\texttt{T}(\text{TPM}_M)$	Axiom 3: (5), (3)
(7)	$\texttt{Trusted}_\texttt{C}(C)$	Axiom 4: (5), (6)
(8)	$\texttt{On}(\text{TPM}_M, M)$	Assumption 3
(9)	$\neg\, \texttt{PhysSecure}(M)$	Assumption 4
(10)	$\neg\, \texttt{Trusted}_\texttt{T}(\text{TPM}_M)$	Axiom 2: (8), (9)
(11)	$\neg\, \texttt{Trusted}_\texttt{C}(C)$	Axiom 5: (5), (10)
(12)	$\bot$	7, 11

Figure 3.4: **Proof Failure Reveals Cuckoo Attack.** *Applying our axioms to our assumptions leads to a logical contradiction.*

As a result, the user will decide to trust the local PC. Any secrets she enters can be captured by malware and forwarded to the attacker. Even secrets protected by TPM-based guarantees (e.g., encrypted using K_{TPM_M}) will be compromised, since the TPM's specifications offer no guarantees for a TPM in the physical possession of the adversary.

Thus, it is crucial that the user be able to securely communicate with the TPM in the *local* machine before revealing any sensitive information. Note that while this attack resembles a classic Attacker-in-the-Middle attack, it differs in that the attacker controls both the local and the remote machine.

3.1.2 Formal Model

To analyze the cuckoo attack more formally, we can model the situation using predicate logic. Figure 3.2 summarizes our proposed model for establishing trust in a computer equipped with secure hardware. The first axiom encodes our assumption that trusted humans can vouch for the physical integrity of a computer. The next two axioms codify the TPM's vulnerability to hardware attacks. The second set of axioms encodes our assumption that trust in the TPM inside a computer suffices (via software attestations) to establish trust in the computer. The final axiom represents the fact that today, without the local TPM's public key, the user must accept the computer's assertion that a particular TPM resides on the computer.

To "initialize" the system, we also encode our assumptions about the concrete setting in a set of predicates (shown in Figure 3.3). By applying our set of axioms to the initial assumptions, we can reason about the trustworthiness of the local machine. Unfortunately,

as shown in Figure 3.4, such reasoning leads to a logical contradiction, namely that the local machine C is both trusted and untrusted. This contradiction captures the essence of the cuckoo attack, since it shows that the user cannot decide whether to trust the local machine.

Removing the contradiction requires revisiting our axioms or our assumptions. We explore these options below.

3.2 Potential Solutions

The cuckoo attack is possible because the attacker can convince the user to accept assurances from an untrustworthy TPM. In this section, we first show that an obvious solution, cutting off network access, addresses one instantiation of the cuckoo attack but does not solve the problem, since malware on the local machine may have enough information to perfectly emulate a TPM in software. To avoid similar missteps, we return to our formal model and consider solutions that remove an assumption, as well as solutions that fix an axiom. For each approach, we provide several concrete instantiations and an analysis of their advantages and disadvantages.

3.2.1 Removing Network Access

From Figure 3.1(a), it may seem that the cuckoo attack can be prevented by severing the connection between the local malware the adversary's remote PC. The assumption is that without a remote TPM to provide the correct responses, the infected machine must either refuse to respond or allow the true TPM to communicate with the user's device (thus, revealing the presence of the malware).

Below, we suggest how this could be implemented, and show that regardless of the implementation, this solution fundamentally does not work. We demonstrate this both with the formal model from Section 3.1.2, and with an attack.

There are several ways to remove the local malware's access to the remote TPM. We could instruct the user to sever all network connections. If the user cannot be trusted to reliably accomplish this task,[3] the verifier could jam the network connections. For example, the user's fob might include a small RJ-45 connector to plug the Ethernet jack and jam the wireless network at the logical level (by continuously sending Request-to-Send frames) or at the physical level. Finally, we could use a distance-bounding protocol [32] to prevent the adversary from making use of a remote TPM. Since the speed of light is constant [54], the verifier can re-

[3] For example, it may be difficult to tell if an infected laptop has its wireless interface enabled.

quire fast responses from the local platform and be assured that malware on the computer does not have time to receive an answer from a remote party. However, with current TPMs, identification operations take half a second or more, with considerable variance both on a single TPM and across the various TPM brands (see Section 4.5.2.3). A signal traveling at the speed of light can circle the earth about four times in the time required for an average TPM to compute a signature, making distance-bounding infeasible.

Unfortunately, removing network access is fundamentally insufficient to prevent the replay attack. One way to see this is via the formal model from Figure 3.2. Neither the predicates nor the axioms assume the local adversary has access to the remote PC. The logical flaw that allows the cuckoo attack to happen arises from Axiom 6, i.e., the local computer's ability to convince the user that a particular TPM resides on the local computer. In other words, as shown in Figure 3.1(b), the cuckoo attack is possible because the malware on the local machine has access to a "TPM oracle" that provides TPM-like answers without providing TPM security guarantees. If the local malware can access this oracle without network access, then cutting off network access is insufficient to prevent the cuckoo attack.

In particular, since the adversary has physical possession of TPM_M, he can extract its private key. He can then provide the malware on the local computer with the private key, TPM_M's Endorsement Certificate, and a list of trusted PCR values. Thus provisioned, the malware on the local machine can perfectly emulate TPM_M, even without network access.

3.2.2 Eliminating Malware

An alternate approach is to try to remove the malware on Alice's local computer. In our formal model, this equates to removing Assumption 5, which would remove the contradiction that results in the cuckoo attack. Unfortunately, this approach is both circular and hard to achieve.

First, we arrived at the cuckoo attack based on the goal of ensuring that the local machine could be trusted. In other words, the goal is to detect (and eventually remove), any malware on the machine using the TPM. Removing malware in order to communicate securely with the TPM, in order to detect and remove malware, potentially leaves us stuck in an endless loop without a base case.

In practice, there are two approaches to cutting through this circularity, but neither is satisfactory.

§1 **Trust.** The "null" solution is to simply ask the local machine for its key and trust that no malware is present.

Pros: This is clearly the simplest possible solution. Sadly, it seems to be the only viable solution available today, at least without special devices or additional hardware changes.

Cons: The assumption that the machine is not compromised will not hold for many computers. Unprotected Windows PCs are infected in minutes [2]. Even newly purchased devices may not meet this criteria [119, 188].

§2 **Timing Deviations.** Researchers have observed that certain computations can be done faster locally than malware can emulate the same computations while hiding its own presence (see Section 2.5.4). By repeating these computations, a timing gap appears between a legitimate execution of the protocol, and a malware-simulated execution. Using such a system, we could run a code module on the local computer to check for malware.

Pros: Since these approaches do not rely on special hardware, they can be employed immediately on current platforms.

Cons: Using timing deviations requires severing the PC's network access; Section 3.2.1 shows that this is non-trivial. Also, such techniques require specific hardware knowledge (e.g., about the exact CPU architecture/model, memory size, cache size, etc.) that the user is unlikely to possess.

3.2.3 Establishing a Secure Channel

Given the conclusions above, we must keep the assumptions in Figure 3.3. Thus, to find a solution, we must fix one or more of our axioms. We argue that the correct target is Axiom 6, as the others are fundamental to our problem definition.

We cannot simply remove Axiom 6, since without it, we cannot introduce the notion of a TPM being installed on a computer. Instead, establishing a secure (authentic and integrity-preserving) channel to the TPM on the local machine suffices to fix Axiom 6. Such a secure channel may be established using hardware or cryptographic techniques.

For a hardware-based approach, we would introduce a new predicate $\texttt{HwSaysOn}(t, c)$ indicating that a secure hardwired channel allowed the user to connect to the TPM on the local machine. Axiom 6 would then be written as:

$$\forall t, c \quad \texttt{HwSaysOn}(t, c) \rightarrow \texttt{On}(t, c)$$

A cryptographic approach requires the user to obtain some authentic cryptographic information about the TPM she wishes to communicate with. Based on the user's trust in the source of the information, she could then decide that the TPM was in fact inside the ma-

chine. We could encode this using the predicate $\texttt{PersonSaysOn}(p,\ t,\ c)$ indicating that a person p has claimed that TPM t is inside computer c. Axiom 6 would then be written as:

$$\forall p, t, c \quad \texttt{TrustedPerson}(p) \wedge \texttt{PersonSaysOn}(p, t, c) \rightarrow \texttt{On}(t, c)$$

3.2.3.1 Hardware-Based Secure Channels

Below, we analyze ways to implement a hardware-based modification to Axiom 6 to allow the user to establish a secure channel with the TPM on the local computer.

§3 **Special-Purpose Interface.** Add a new hardware interface to the computer that allows an external device to talk directly to the TPM. The TPM already supports differential access rights, so the external interface could be designed to allow the external verifier to guarantee that software on the machine does not interfere with the contents of the TPM while the verifier is attached.

Pros: The use of a special-purpose port reduces the chances for user error (since they cannot plug the external verifier into an incorrect port).

Cons: Introducing an entirely new interface and connector specification would require significant industry collaboration and changes from hardware manufacturers, making it an unlikely solution in the near term.

§4 **Existing Interface.** Use an existing external interface (such as Firewire or USB) to talk directly to the TPM.

Pros: This solution is much simpler to deploy, since it does not require any manufacturer changes.

Cons: Existing interfaces are not designed to support this type of communication. For example, USB devices cannot communicate with the host platform until addressed by the host. Even devices with more freedom, such as Firewire devices, can only read and write to memory addresses. While the TPM is made available via memory-mapped I/O ports, these mappings are established by the software on the machine, and hence can be changed by malware. Thus, there does not appear to be a way to reuse existing interfaces to communicate reliably with the local TPM.

§5 **External Late Launch Data.** Recent CPUs from AMD and Intel can perform a *late launch* of an arbitrary piece of code (see Section 2.4.1). During the late launch, the code to be executed is measured and the measurement is sent to the TPM. The code is then executed in a protected environment that prevents interference from any other hardware or

software on the platform. If the late launch operation also made the code's measurement code available externally, then the user's verifier could check that the invoked code was trustworthy. The code could then check the integrity of the platform or establish a secure channel from the verifier to the TPM.

Pros: Recent CPUs contain the late launch functionality needed to measure and securely execute code.

Cons: Existing interfaces (such as USB) do not allow the CPU to convey the fact that a late launch occurred nor the measurement of the executed code in an authentic fashion. Malware on the computer could claim to perform a late launch and then send a measurement of a legitimate piece of code. This attack could be prevented by creating a special-purpose interface that talks directly to the CPU, but this brings us back to §3, which is a simpler solution.

§6 **Special-Purpose Button.** Add a new button on the computer for bootstrapping trust. For example, the button can execute an authenticated code module that establishes a secure channel between the verifier (connected via USB, for example) and the TPM. Alternatively, the button could disable all network interfaces to prevent the cuckoo attack from occurring. Such a button could also be useful for taking a laptop on an airplane.

Pros: A hardware button press cannot be overridden by malware. It also provides the user with a tangible guarantee that secure bootstrapping has been initiated.

Cons: Executing an authenticated code module requires hardware not only for invoking the necessary code, but also for verifying digital signatures (similar to §9), since the code will inevitably need updates. This approach also relies on the user to push the button before connecting the verifier device, since the device cannot detect the button push. If the user plugs in the verifier before pushing the button, on the computer could fool the device with a cuckoo attack. Both versions of this solution require hardware changes.

3.2.3.2 Cryptographic Secure Channels

Establishing a cryptographically-secure channel requires the user to share a secret with the TPM or to obtain the TPM's public key. Without a prior relationship with the TPM, the user cannot establish a shared secret, so in this section we focus on public-key methods.

§7 **Seeing-is-Believing (SiB).** An approach suggested by McCune et al. [134] (and later used for kiosk computing [65]) is to have the computer's manufacturer encode a hash of the platform's identity in a 2-D barcode and attach the barcode to the platform's case. Note

that this step should be performed by the manufacturer and not, say, the current owner, since the current owner would have to establish the TPM's identity, in which case the problem would simply recurse to them. Using a camera-equipped smartphone, the user can take a picture of the 2-D barcode and use the smartphone to process the computer's attestation.

Pros: This solution is attractive, since it requires relatively little effort from the manufacturer, and most people find picture-taking simple and intuitive.

Cons: Because it requires a vendor change, this solution will not help current platforms. It also requires the user to own a relatively expensive smartphone and install the relevant software. The user must also trust that the smartphone has not been compromised. As these phones grow increasingly complex, this assumption is likely to be violated. In a kiosk setting, the 2-D barcode may be replaced or covered up by an attacker.

§8 **SiB Without a Camera.** Instead of using a 2-D barcode, the manufacturer could encode the hash as an alpha-numeric string. The user could then enter this string into a smartphone, or into a dedicated fob.

Pros: Similar to §7, except the user no longer needs a camera-equipped device.

Cons: Similar to those of §7, but it still requires non-trivial input capability on the user's device. Relies on the user to correctly enter a complicated string.

§9 **Trusted BIOS.** If the user trusts the machine's BIOS, she can reboot the machine and have the trusted BIOS output the platform's identity (either visually or via an external interface, such as USB). The trusted BIOS must be protected from malicious updates. For example, some Intel motherboards will only install BIOS updates signed by Intel [118].

Pros: This approach does not require the user to use any custom hardware.

Cons: The user must reboot the machine, which may be disruptive. It relies on the user to only insert the verifier after rebooting, since otherwise the verifier may be deceived by local malware. The larger problem is that many motherboards do not include the protections necessary to guarantee the trustworthiness of the BIOS, and there is no indicator to signal to the user that the BIOS in the local computer is trustworthy.

§10 **Trusted Third Party.** The TPM could be equipped with a certificate provided by a trusted third-party associating the TPM with a particular machine. The verifier can use the trusted third party's public key to verify the certificate and establish trust in the TPM's public key.

Pros: The verifier only needs to hold the public key for the trusted third party and perform basic certificate checks. No hardware changes are needed.

Cons: It is unclear how the verifier could communicate the TPM's location as specified in the certificate to the user in a clear and unambiguous fashion. This solution also simply moves the problem of establishing a TPM's identity to the third party, who would need to employ one of the other solutions suggested here.

3.3 Preferred Solutions

Of all the solutions presented in Section 3.2, we argue that §3 (a special-purpose hardware interface) provides the strongest security. It removes almost every opportunity for user error, does not require the preservation of secrets, and does not require software updates. Unfortunately, the cost and industry collaboration required to introduce a new interface make it unlikely to be deployed in the near future.

Of the plausibly deployable solutions, we argue in favor of §8 (an alphanumeric hash of the TPM's public key), since it allows for a simpler verification device.

Nonetheless, we recognize that these selections are open to debate, and believe that considerable room remains for additional solutions.

3.4 Summary

Trust in a local computer is necessary for a wide variety of important tasks. Ideally, we should be able to use secure hardware, such as the TPM, to leverage our trust in the physical security of the machine in order to trust the software executing on the platform. Our formal model reveals that current attempts to create this chain of trust are vulnerable to the cuckoo attack. The model is also useful for identifying solutions, and we have explored the tradeoffs inherent in such solutions.

Chapter 4

On-Demand Secure Code Execution on Commodity Computers

While the techniques from Chapter 3 allow the user to learn what code is running on her computer, today's popular operating systems run a daunting amount of code in the CPU's most privileged mode. For example, the Linux kernel (as of version 2.6) consists of nearly 5 million lines of code [210], while Microsoft's Windows Vista includes over 50 million lines of code [125]. Their large size and huge complexity makes them difficult to analyze and vulnerable to attack. Indeed, the plethora of vulnerabilities in operating system code makes the compromise of systems commonplace, and an OS's privileged status is inherited by the malware that invades it. The integrity and secrecy of every application is at risk in such an environment.

Previous work has attempted to deal with this problem by running a *persistent security layer* in the computer's most privileged mode [44, 72, 104, 113, 180, 196]. This layer has been variously dubbed a security kernel, a virtual machine monitor (VMM), or a hypervisor. This layer is responsible for creating isolation domains for ordinary, untrusted code and for the security-sensitive code. Unfortunately, this approach has a number of inherent drawbacks. The security layer's need to interpose on hardware accesses leads to performance degradation for ordinary code, and often requires eliminating access to devices that are too complicated to emulate (e.g., a 3D graphics card) [25]. Furthermore, the need to run both untrusted and trusted code simultaneously can lead to security vulnerabilities (e.g., side-channel attacks [28, 157]), as well as code bloat in the security layer; the initial implementation of the Xen VMM required 42K lines of code [24] and within a few years almost doubled to approximately 83K lines [126].

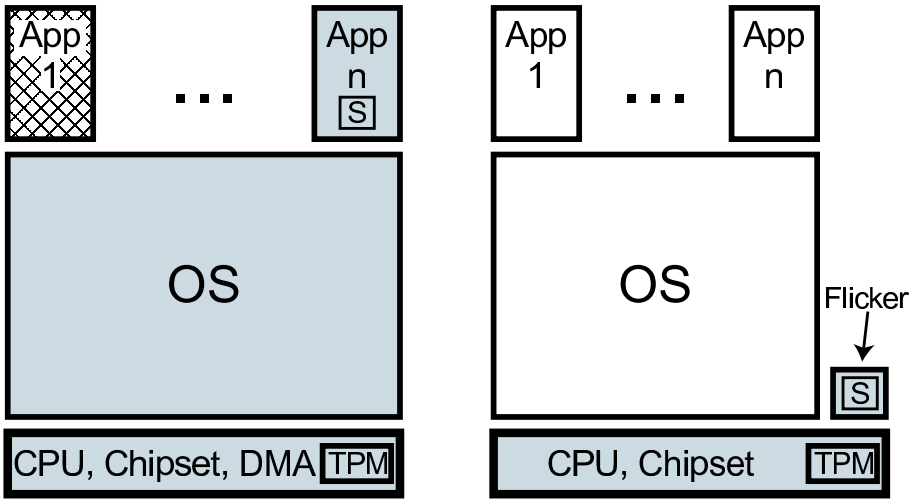


Figure 4.1: **Trusted Computing Base Comparison.** *On the left, a traditional computer is shown with an application that executes sensitive code (S). On the right, Flicker protects the execution of the sensitive code. The shaded portions represent components that must be trusted; other applications are included on the left because many applications run with superuser privileges, or with the same user privileges as the application executing sensitive code.*

To avoid these drawbacks, we propose Flicker, an architecture for providing secure code execution *on demand* with a minimal Trusted Computing Base (TCB). When invoked for secure code execution, Flicker creates an isolated environment such that none of the software executing before Flicker begins can monitor or interfere with Flicker code execution, and all traces of Flicker code execution can be eliminated before regular software execution resumes. For example, a Certificate Authority (CA) could sign certificates with its private key, even while keeping the key secret from an adversary who controls the BIOS, OS, and DMA-enabled devices (see Section 4.4.3.2). Furthermore, Flicker imposes zero overhead on ordinary code execution: when security-sensitive code is not executing, ordinary code has full access to hardware features and performance.

Flicker provides strong isolation guarantees while requiring an application to trust as few as 250 additional lines of code for its secrecy and integrity. As a result, Flicker circumvents entire layers of legacy system software and eliminates the need to rely on their correctness for security properties. As indicated in Figure 4.1, the contrast in the TCB for a sensitive operation with and without Flicker is dramatic. Once the TCB for code execution has been precisely defined and limited, formal assurance of both reliability and security properties enters the realm of possibility.

The use of Flicker, as well as the exact code executed (and its inputs and outputs), can be attested to an external party. For example, a piece of server code handling a user's password can execute in complete isolation from all other software on the server, and the server can prove to the client that the secrecy of the password was preserved (see Section 4.4.3.1). Such fine-grained attestations make a remote party's verification much simpler, since the verifier need only trust a small piece of code, instead of trusting Application X running alongside Application Y on top of OS Z with some number of device drivers installed. Also, the party using Flicker does not leak extraneous information about the system's software state, thus helping to preserve user privacy.

To achieve these properties, Flicker utilizes hardware support for late launch and attestation recently introduced in commodity processors from AMD and Intel. These processors already ship with off-the-shelf computers and will soon become ubiquitous. Although current hardware still has a high overhead, due to TPM implementations optimized for cost and a design that was meant to infrequently bootstrap a new VMM, we anticipate that future hardware performance will improve as these functions are increasingly used. Indeed, in Section 4.6, we suggest hardware modifications that can improve performance by up to six orders of magnitude. Although other researchers have proposed compelling hardware security architectures, e.g., XOM [123] or AEGIS [195], we focus on hardware modifications that tweak or slightly extend existing hardware functionality. We believe this approach offers the best chance of seeing hardware-supported security deployed in the real world.

From a programmer's perspective, the sensitive code protected by Flicker can be written from scratch or extracted from an existing program. To simplify this task, the programmer can draw on a collection of small code modules we have developed for common functions. For example, one small module protects the existing execution environment from malicious or malfunctioning sensitive code. A programmer can also apply tools we have developed to extract sensitive operations and relevant code from an application. Flicker then executes this code in complete isolation from the rest of the system. Note that applications are not restricted to defining a single Flicker module; rather, they can use arbitrarily many Flicker modules to perform various sensitive operations.

We present an implementation of Flicker using AMD's SVM technology and use it to improve the security of a variety of applications. We develop a rootkit detector that an administrator can run on a remote machine in such a way that she receives a guarantee that the detector executed correctly and returned the correct result. We also show how Flicker can improve the integrity of results for distributed computing projects, such as SETI@Home [10] or Folding@Home [153]. Finally, we use Flicker to protect a CA's private signing key and to improve an SSH server's password handling.

Contributions. In this chapter, we make the following contributions: **(1)** We design and implement an architecture that provides on-demand isolated code execution while adding a smidgen of code (orders of magnitude smaller than previous systems) to the TCB for an application's data secrecy and integrity. **(2)** We provide meaningful, fine-grained attestations of only the security-sensitive code. **(3)** We describe the development of a Flicker toolkit to simplify application development. **(4)** We provide a detailed description of a complete implementation and performance evaluation of Flicker on an AMD SVM platform with a v1.2 TPM. **(5)** We recommend modifications of commodity hardware to securely improve the performance and concurrency of Flicker. In our recommendations, we seek to minimize the changes required, thereby increasing the likelihood of their adoption.

4.1 Problem Definition

Before presenting Flicker, we define the class of adversaries we consider. We also define our goals and explain why the new hardware capabilities do not meet them on their own.

4.1.1 Adversary Model

At the software level, the adversary can subvert the operating system, so it can also compromise arbitrary applications and monitor all network traffic. Since the adversary can run code at ring 0, it can invoke the *SKINIT* instruction with arguments of its choosing. We also allow the adversary to regain control between Flicker sessions. In other words, the Flicker code run by *SKINIT* will eventually yield control of the system back to the compromised operating system. We do not consider Denial-of-Service attacks, since a malicious OS can always simply power down the machine or otherwise halt execution to deny service.

At the hardware level, we make the same assumptions as does the Trusted Computing Group with regard to the TPM [200]. In essence, the attacker can launch simple hardware attacks, such as opening the case, power cycling the computer, or attaching a hardware debugger. The attacker can also compromise expansion hardware such as a DMA-capable Ethernet card with access to the PCI bus. However, the attacker cannot launch sophisticated hardware attacks, such as monitoring the high-speed bus that links the CPU and memory.

4.1.2 Goals

We describe the goals for isolated execution and explain why SVM alone does not meet them.

Isolation. Provide complete isolation of security-sensitive code from all other software (including the OS) and devices in the system. Protect the secrecy and integrity of the code's data after it exits the isolated execution environment.

Provable Protection. After executing security-sensitive code, convince a remote party that the intended code was executed with the proper protections in place. Provide assurance that a remote party's sensitive data will be handled only by the intended code.

Meaningful Attestation. Allow the creation of attestations that include measurements of exactly the code executed, its inputs and outputs, and nothing else. This property gives the verifier a tractable task, instead of learning only that untold millions of lines of code were executed, and leaks as little information as possible about the attestor's software state.

Minimal Mandatory TCB. Minimize the amount of software that security-sensitive code must trust. Individual applications may need to include additional functionality in their TCBs, e.g., to process user input, but the amount of code that must be included in every application's TCB must be minimized.

On their own, AMD's SVM and Intel's TXT technologies only meet two of the above goals. While both provide Isolation and Provable Protection, they were both designed with the intention that the *SKINIT* instruction would be used to launch a secure kernel or secure VMM [79]. Either mechanism will significantly increase the size of an application's TCB and dilute the meaning of future attestations. For example, a system using the Xen [25] hypervisor with *SKINIT* would add almost $50,000$ lines of code[1] to an application's TCB, not including the Domain 0 OS, which potentially adds millions of additional lines of code to the TCB.

In contrast, Flicker takes a bottom-up approach to the challenge of managing TCB size. Flicker starts with fewer than 250 lines of code in the software TCB. The programmer can then add only the code necessary to support her particular application into the TCB.

4.2 Flicker Architecture

Flicker provides complete, hardware-supported isolation of security-sensitive code from all other software and devices on a platform (even including hardware debuggers and DMA-enabled devices). Hence, the programmer can include exactly the software needed for a particular sensitive operation and exclude all other software on the system. For example, the programmer can include the code that decrypts and checks a user's password but exclude the portion of the application that processes network packets, the OS, and all other software on the system.

[1] http://xen.xensource.com/

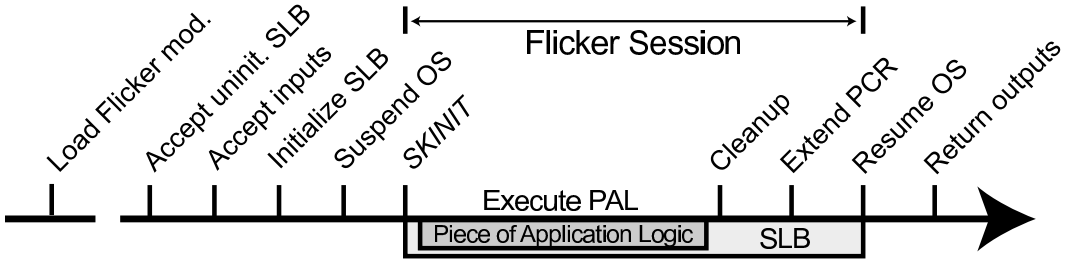


Figure 4.2: PAL **Execution.** *Timeline showing the steps necessary to execute a* PAL. *The SLB includes the* PAL*, as well as the code necessary to initialize and terminate the Flicker session. The gap in the time axis indicates that the flicker-module is only loaded once.*

4.2.1 Flicker Overview

Flicker achieves its properties using the late launch capabilities described in Section 2.4.2. Instead of launching a VMM, Flicker pauses the current execution environment (e.g., the untrusted OS), executes a small piece of code using the *SKINIT* instruction, and then resumes operation of the previous execution environment. The security-sensitive code selected for Flicker protection is the Piece of Application Logic (PAL). The protected environment of a Flicker session starts with the execution of *SKINIT* and ends with the resumption of the previous execution environment. Figure 4.2 illustrates this sequence.

Application developers must provide the PAL and define its interface with the remainder of their application (we discuss this process, as well as our work on automating it, in Section 4.3). To create an SLB (the Secure Loader Block supplied as an argument to *SKINIT*), the application developer links her PAL against an uninitialized code module we have developed called the SLB Core. The SLB Core performs the steps necessary to set up and tear down the Flicker session. Figure 4.4 shows the SLB's memory layout.

To execute the resulting SLB, the application passes it to a Linux kernel module we have developed, *flicker-module*. It initializes the SLB Core and handles untrusted setup and teardown operations. The *flicker-module* is not included in the TCB of the application, since its actions are verified.

4.2.2 Isolated Execution

We provide a simplified discussion of the operation of a Flicker session by following the timeline in Figure 4.2.

Accept Uninitialized SLB and Inputs. *SKINIT* is a privileged instruction, so an application uses the *flicker-module*'s interface to invoke a Flicker session. In the sysfs,[2] the *flicker-module* makes four entries available: `control`, `inputs`, `outputs`, and `slb`. Applications interact with the *flicker-module* via these filesystem entries. An application first writes to the `slb` entry an uninitialized SLB containing its PAL code. The *flicker-module* allocates kernel memory in which to store the SLB; we refer to the physical address at which it is allocated as `slb_base`. The application writes any inputs for its PAL to the `inputs` sysfs entry; the inputs are made available at a well-known address once execution of the PAL begins (the parameters are at the top of Figure 4.4). The application initiates the Flicker session by writing to the `control` entry in the sysfs.

Initialize the SLB. When the application developer links her PAL against the SLB Core, the SLB Core contains several entries that must be initialized before the resulting SLB can be executed. The *flicker-module* updates these values by patching the SLB.

When the *SKINIT* instruction executes, it puts the CPU into flat 32-bit protected mode with paging disabled, and begins executing at the entry point of the SLB. By default, the PAL is not built as position independent code, so it assumes that it starts at address 0, whereas the actual SLB may start anywhere within the kernel's address space. The SLB Core addresses this issue by enabling the processor's segmentation support and creating segments that start at the base of the PAL code. During the build process, the starting address of the PAL code is unknown, so the SLB Core includes a skeleton Global Descriptor Table (GDT) and Task State Segment (TSS). Once the *flicker-module* allocates memory for the SLB, it can compute the starting address of the PAL code, and hence it can fill in the appropriate entries in the SLB Core.

Suspend OS. *SKINIT* does not save existing state when it executes. However, we want to resume the untrusted OS following the Flicker session, so appropriate state must be saved. This is complicated by the fact that the majority of systems available with AMD SVM support are multi-core. On a multi-CPU system, the *SKINIT* instruction has additional requirements which must be met for secure initialization. In particular, *SKINIT* can only be run on the Boot Strap Processor (BSP), and all Application Processors (APs) must successfully receive an INIT Inter-Processor Interrupt (IPI) so that they respond correctly to a handshaking synchronization step performed during the execution of *SKINIT*. However, the BSP cannot simply send an INIT IPI to the APs if they are executing processes. Our solution is to use the CPU Hotplug support available in recent Linux kernels (starting with version 2.6.19) to deschedule all APs. Once the APs are idle, the *flicker-module* sends an INIT IPI by writing to

[2] A virtual file system that exposes kernel state.

Register	Description
EBP	Stack base pointer
ESP	Stack pointer
EFER	Extended feature register
EFLAGS	Flags register
CR0	Processor control register
CR3	Page table base address
CR4	Model-specific extensions
GDTR	Global descriptor table register
IDTR	Interrupt descriptor table register
TR	Task register

Figure 4.3: **Saved Execution State.** *Registers with state which must be saved during Flicker sessions.*

the system's Advanced Programmable Interrupt Controller. At this point, the BSP is prepared to execute *SKINIT*, and the OS state needs to be saved. In particular, we save information about the Linux kernel's page tables so the SLB Core can restore paging and resume the OS after the PAL exits. Figure 4.3 provides additional details about the system registers with values that must be saved.

While the v1.2 TPM is capable of being shared by multiple layers of system software, the current Linux driver (the `tpm_tis` module) does not expect to share the TPM. To allow *SKINIT* to operate successfully, and to permit the OS to maintain its ability to interact with the TPM after a Flicker session, we modified the `tpm_tis` driver by exposing an interface to its functions for gaining and releasing control of the TPM. (the alternative was to unload the entire driver, and thus any software which depends upon it, before the Flicker session and then reload it afterwards).

***SKINIT* and the SLB Core.** The *SKINIT* instruction enables hardware protections and then begins to execute the SLB Core, which prepares the environment for PAL execution. Executing *SKINIT* enables the hardware protections described in Section 2.4.2. In brief, the processor adds entries to the Device Exclusion Vector (DEV) to disable DMA to the memory region containing the SLB, disables interrupts to prevent the previously executing code from regaining control, and disables debugging support, even for hardware debuggers. By default, these protections are offered to 64 KB of memory, but they can be extended to larger memory regions. If this is done, preparatory code in the first 64 KB must add this additional memory to the DEV, and extend measurements of the contents of this additional memory into the TPM's PCR 17 after the hardware protections are enabled, but before transferring control to any code in these upper memory regions.

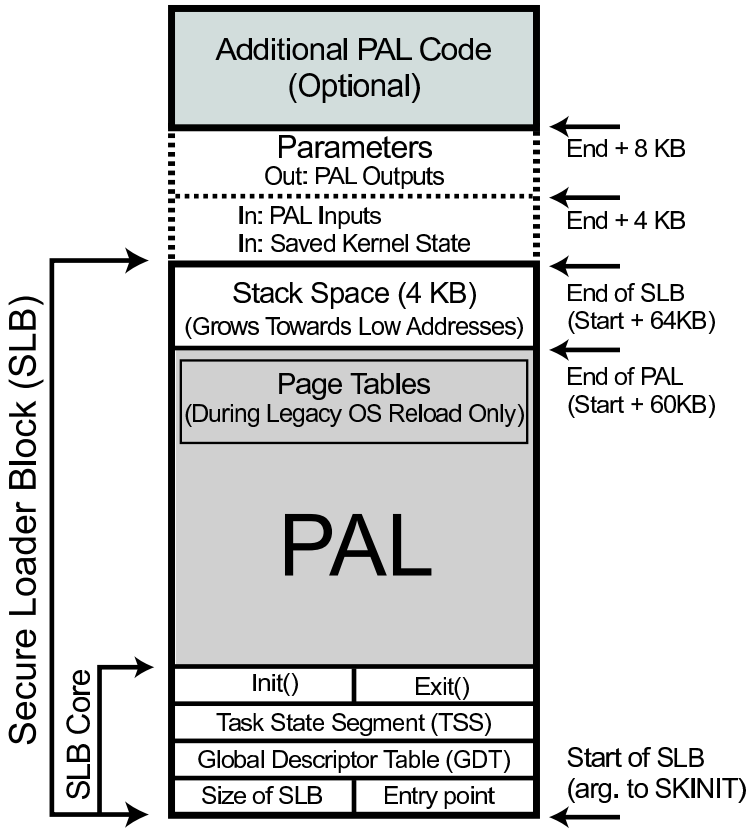


Figure 4.4: **Memory Layout of the SLB.** *The shaded region indicates memory containing executable* PAL *code. The dotted lines indicates memory used to transfer data into and out of the SLB. After the* PAL *has executed and erased its secrets, memory that previously contained executable code is used for the skeleton page tables needed to reload the OS.*

To enable straightforward execution of PAL code, it is useful to (re)define the code, data, and stack segments such that they begin at the physical base address allocated by the *flicker-module*. When the *flicker-module* prepares the PAL for use as an SLB, it writes this base address into GDT and TSS entries in the SLB Core. To minimize GDT size, the entries describing data segments are also used for stack segments. The GDT in our SLB Core contains six entries: a code and data descriptor with base address 0, a code and data descriptor with base address `slb_base`, and a call gate and task state selector. We defer discussion of the last two entries until Section 4.3, where we consider CPU privilege level changes.

The Initialization operations performed by the SLB Core once *SKINIT* gives it control are: (i) load the GDT, (ii) load the CS, DS, and SS registers and (iii) call the PAL, providing the address of PAL inputs as a parameter.

Execute PAL. Once the environment has been prepared, the PAL executes its specified application logic. To keep the TCB small, the default SLB Core includes no support for heaps, memory management, or virtual memory. Thus, it is up to the PAL developer to include the functionality necessary for her particular application. Section 4.3 describes some of our existing modules that can optionally be included to provide additional functionality. We have also developed a module that can restrict the actions of a PAL, since by default (i.e., without the module), a PAL can access the machine's entire physical memory and execute arbitrary instructions (see Section 4.3.1.2 for more details).

During PAL execution, output parameters are written to a well-known location beyond the end of the SLB. When the PAL exits, the SLB Core regains control.

Cleanup. The PAL's exit triggers the cleanup and exit code at the end of the SLB Core. The cleanup code erases any sensitive data left in memory by the PAL.

Extend PCR. To signal the completion of the SLB, the SLB Core extends a well known value into PCR 17. As we discuss in Section 4.2.4.1, this allows a remote party to distinguish between values generated by the PAL (trusted), and those produced after the OS resumes (untrusted).

Resume OS. Linux operates with paging enabled and segment descriptors set to cover all of memory, but the SLB executes in protected mode with segment descriptors starting at `slb_base`. We transition between these two states in two phases. First, we reload the segment descriptors with GDT entries that cover all of memory, and second, we enable paged memory mode.

We use a call gate in the SLB Core's GDT as a well-known point for resuming the untrusted OS. It is used to reload the code segment descriptor register with a descriptor covering all of memory.

After reloading the data and stack segments, we re-enable paged memory mode. This requires the creation of a skeleton of page tables to map the SLB Core's memory pages to the virtual addresses where the Linux kernel believes they reside. The procedure resembles that executed by the Linux kernel when it first initializes. The page tables must contain a unity mapping for the memory location of the next instruction, allowing paging to be enabled. Finally, the kernel's page tables are restored by rewriting CR3 (the page table base address register) with the value saved during the Suspend OS phase. Next, the kernel's GDT is reloaded, and control is transferred back to the *flicker-module*.

The *flicker-module* restores the execution state saved during the Suspend OS phase and fully restores control to the Linux kernel by re-enabling interrupts. On a multi-CPU system, it also sends an IPI signal to reenable the Application Processors. If the PAL outputs any values, the *flicker-module* makes them available through the sysfs `outputs` entry.

4.2.3 Multiple Flicker Sessions

PALs can leverage TPM-based sealed storage to maintain state across Flicker sessions, enabling more complex applications. For example, a Flicker-based application may wish to interact with a remote entity over the network. Rather than include an entire network stack and device driver in the PAL (and hence the TCB), we can invoke Flicker more than once (upon the arrival of each message), using secure storage to protect sensitive state between invocations.

Flicker-based secure storage can also be used by applications that wish to share data between PALs. The first PAL can store secrets so that only the second PAL can read them, thus protecting the secrets even when control reverts to the untrusted OS. Finally, Flicker-based secure storage can improve the performance of long-running PAL jobs. Since Flicker execution pauses the rest of the system, an application may prefer to break up a long work segment into multiple Flicker sessions to allow the rest of the system time to operate, essentially multitasking with the OS. We first present the use of TPM Sealed Storage and then describe extensions necessary to protect multiple versions of the same object from a replay attack against sealed storage.

4.2.3.1 TPM Sealed Storage

To save state across Flicker sessions, a PAL uses the TPM to seal the data under the measurement of the PAL that should have access to its secrets. More precisely, suppose PAL P, operating in a Flicker session, wishes to securely store data so that only PAL P', also operating under Flicker protection, can read the data.[3] P' could be a later invocation of P, or it could be a completely different PAL. Either way, while it is executing within the Flicker session, PAL P uses the TPM's Seal command to secure the sensitive data. As an argument, P specifies that PCR 17 must have the value $V \leftarrow H(0\text{x}00^{20}||H(P'))$ before the data can be unsealed. Only an *SKINIT* instruction can reset the value of PCR 17, so PCR 17 will have value V only after PAL P' has been invoked using *SKINIT*. Thus, the sealed data can be unsealed if and only if P' executes under Flicker's protection. This allows PAL code to store persistent data such that it is only available to a particular PAL in a future Flicker session.

[3] For brevity, we will assume that PALs operate with Flicker protection. Similarly, a measurement of the PAL consists of a hash of the SLB containing the PAL.

Seal(d):	**Unseal**(c):
IncrementCounter()	$d\|\|j' \leftarrow$ TPM_Unseal(c)
$j \leftarrow$ ReadCounter()	$j \leftarrow$ ReadCounter()
$c \leftarrow$ TPM_Seal($d\|\|j$,PCR_List)	if ($j' \neq j$) Output($\perp$)
Output(c)	else Output(d)

Figure 4.5: **Protocols for Replay Protection.** *Replay protection for sealed storage based on a secure counter. Ciphertext c is created when data d is sealed.*

4.2.3.2 Replay Prevention for Sealed Storage

TPM-based sealed storage prevents other code from directly learning or modifying a PAL's secrets. However, TPM Seal outputs ciphertext c (for data d) that is handled by untrusted code: $c \leftarrow TPM\_Seal(d, PCR\_list)$. The untrusted code is capable of performing a replay attack where an older ciphertext c' is provided to a PAL. For example, consider a password database that is maintained in sealed storage and a user who changes her password because it is publicized. To change a user's password, version i of the database is unsealed, updated with the new password, and then sealed again as version $i + 1$. An attacker who can cause the system to operate on version i of the password database can gain unauthorized access using the publicized password. To summarize, TPM Unseal ensures that the plaintext of c' is accessible only to the intended PAL, but it does not guarantee that c' is the most recent sealed version of data d.

Replay attacks against sealed storage can be prevented if a secure counter is available, as illustrated in Figure 4.5. To seal an updated data object, the secure counter should be incremented, and the data object should be sealed along with the new counter value. When a data object is unsealed, the counter value included in the data object at seal time should be the same as the current value of the secure counter. If the values do not match, either the counter was tampered with, or the unsealed data object is a stale version and should be discarded.

Options for realizing a secure counter with Flicker include a trusted third party, and the *Monotonic Counter* and *Non-volatile Storage* facilities of v1.2 TPMs [200]. We provide a sketch of how to implement replay protection for sealed storage with Flicker using the TPM's Non-volatile Storage facility, though a complete solution is outside the scope of this paper. In particular, we do not treat recovery after a power failure or system crash during the counter-increment and sealed storage ciphertext-output. In these scenarios, the secure counter can become out-of-sync with the latest sealed-storage ciphertext maintained by the OS. An appropriate mechanism to detect such events is also necessary.

The TPM's *Non-volatile Storage* facility exposes interfaces to *Define Space*, and *Read* and *Write* values to defined spaces. Space definition is authorized by demonstrating possession of the 20-byte TPM *Owner Authorization Data*, which can be provided to a Flicker session using the protocol we present in Section 4.2.4. A defined space can be configured to restrict access based on the contents of specified PCRs. Setting the PCR requirements to match those specified during the TPM Seal command creates an environment where a counter value stored in non-volatile storage is only available to the desired PAL. Values placed in non-volatile storage are maintained in the TPM, so there is no dependence on the untrusted OS to store a ciphertext. This, combined with the PCR-based access control, is sufficient to protect a counter value against attacks from the OS.

4.2.4 Interaction With a Remote Party

Since neither SVM nor TXT include any visual indication that a secure session has been initiated via a late launch, a remote party must be used to bootstrap trust in a platform running Flicker. Below, we describe how a platform attests to the PAL executed, the use of Flicker, and any inputs or outputs provided. We also demonstrate how a remote party can establish a secure channel to a PAL running within the protection of a Flicker session.

4.2.4.1 Attestation and Result Integrity

A platform using Flicker can convince remote parties that a Flicker session executed with a particular PAL. Our approach builds on the TPM attestation process described in Section 2.3.2.2. Below, we refer to the party executing Flicker as the *challenged party*, and the remote party as the *verifier*.

To create an attestation, the challenged party accepts a random nonce from the verifier to provide freshness and replay protection. The challenged party then uses Flicker to execute a particular PAL as described in Section 4.2.2. As part of Flicker's execution, the *SKINIT* instruction resets the value of PCR 17 to 0 and then extends it with the measurement of the PAL. Thus, PCR 17 will take on the value $V \leftarrow H(0\text{x}00^{20}||H(P))$, where P represents the PAL code. The properties of the TPM, chipset, and CPU guarantee that no other operation can cause PCR 17 to take on this value. Thus, an attestation of the value of PCR 17 will convince a remote party that the PAL was executed using Flicker's protection.

After Flicker terminates, the OS causes the TPM to load its AIK, invokes the TPM's Quote command with the nonce provided by the verifier, and specifies the inclusion of PCR 17 in the quote.

To verify the use of Flicker, the verifier must know both the measurement of the PAL, and the public key corresponding to the platform's AIK. These components allow the verifier to authenticate the attestation from the platform. The verifier uses the platform's public AIK to verify the signature from the TPM. It then computes the expected measurement of the PAL, as well as the hash of the input and output parameters. If these values match those extended into PCR 17 and signed by the TPM, the verifier accepts the attestation as valid.

To provide result integrity, after PAL execution terminates, the SLB Core extends PCR 17 with measurements of the PAL's input and output parameters. By verifying the quote (which includes the value of PCR 17), the verifier also verifies the integrity of the inputs and results returned by the challenged party, and hence knows that it has received the exact results produced by the PAL. The nonce provided by the remote party is also extended into PCR 17 to guarantee the freshness of the outputs.

As another important security procedure, after extending the PAL's results into PCR 17, the SLB Core extends PCR 17 with a fixed public constant. This provides several powerful security properties: (i) it prevents any other software from extending values into PCR 17 and attributing them to the PAL; and (ii) it revokes access to any secrets kept in the TPM's sealed storage which may have been available during PAL execution.

4.2.4.2 Establishing a Secure Channel

The techniques described above ensure the integrity of the PAL's input and output, but to communicate securely (i.e., with both secrecy and integrity protections) with a remote party, the PAL and the remote party must establish a secure channel. Fortunately, we need not include communication software (such as network drivers) in the PAL's TCB, since we can use multiple invocations of a PAL to process data from the remote party while letting the untrusted OS manage the encrypted network packets.

Figure 4.6 illustrates a protocol for securely conveying a public key from the PAL to a remote party. This protocol is similar to one developed at IBM for linking remote attestation to secure tunnel endpoints [73]. The PAL generates an asymmetric keypair $\{K_{\mathrm{PAL}}, K_{\mathrm{PAL}}^{-1}\}$ within its secure execution environment. It seals the private key K_{PAL}^{-1} under the value of PCR 17 so that only the identical PAL invoked in the secure execution environment can access it. Note that the PAL developer may extend other application-dependent data into PCR 17 before sealing the private key. This ensures the key will be released only if that application-dependent data is present.

The nonce value sent by the remote party for the TPM quote operation is also provided as an input to the PAL for extension into PCR 18. This provides the remote party with a

Remote Party (RP):	has AIK_{server}, expected hash(PAL || shim) = $\hat{H}$
RP:	generate $nonce$
RP → App:	$nonce$
App → PAL:	$nonce$
PAL:	extend($PCR_{18}, nonce$) generate $\{K_{\mathrm{PAL}}, K_{\mathrm{PAL}}^{-1}\}$ extend($PCR_{18}, h(K_{\mathrm{PAL}})$) $sdata \leftarrow$ seal($PCR_{17}, K_{\mathrm{PAL}}^{-1}$) extend($PCR_{17}, \perp$) extend($PCR_{18}, \perp$)
PAL → App:	$K_{\mathrm{PAL}}, sdata$
App:	$q \leftarrow$ quote($nonce, \{17, 18\}$)
App → RP:	q, K_{PAL}
RP:	if ($\neg$Verify($AIK_{server}, q, nonce$) $\vee\ q.PCR_{17} \neq h(h(0\|\|\hat{H})\|\|\perp)$ $\vee\ q.PCR_{18} \neq$ $h(h(h(0\|\|nonce)\|\|h(K_{\mathrm{PAL}}))\|\|\perp)$) then abort
RP:	has authentic K_{PAL} knows server ran Flicker
App:	saves $sdata$

Figure 4.6: **Establishing a Secure Channel.** *A protocol to generate a cryptographic keypair and convey the public key K_{PAL} to a remote party (RP). The messages sent between the remote party and the* PAL *can safely travel through the untrusted portion of the application (App) and the OS kernel. $\perp$ denotes a well-known value which signals the end of extensions performed within the Flicker session.*

different freshness guarantee: that the PAL was invoked in response to the remote party's request. Otherwise, a malicious OS may be able to fool multiple remote parties into accepting the same public key.

As with all output parameters, the public key K_{PAL} is extended into PCR 18 before it is output to the application running on the untrusted host. The application generates a TPM quote over PCRs 17 and 18 based on the nonce from the remote party. The quote allows the remote party to determine that the public key was indeed generated by a PAL running in the secure execution environment. The remote party can use the public key to create a secure channel [86] to future invocations of the PAL.

```
#include "slbcore.h"
const char* msg = "Hello, world";
void pal_enter(void *inputs) {
 for(int i=0;i<13;i++)
    PAL_OUT[i] = msg[i];     }
```

Figure 4.7: **An Example** PAL**.** *A simple* PAL *that ignores its inputs, and outputs "Hello, world." `PAL_OUT` is defined in `slbcore.h`.*

Our implementation of Flicker makes the above protocol available as a module that developers can include with their PAL. We discuss this further in Section 4.3.

4.3 Developer's Perspective

Below, we describe the process of creating a PAL from the perspective of an application developer. Then, we discuss techniques for automating the extraction of sensitive portions of an existing application for inclusion in a PAL.

4.3.1 Creating a PAL

We have developed Flicker primarily in C, with some of the core functionality written in x86 assembly. However, any language supported by GNU binutils and that can be linked against the core Flicker components is viable for inclusion in a PAL.

4.3.1.1 A "Hello, World" Example PAL

As an example, Figure 4.7 illustrates a simple PAL that ignores its inputs, and outputs the classic message, "Hello, world." Essentially, the PAL copies the contents of the global `msg` variable to the well-known PAL output parameter location (defined in the `slbcore` header file). Our convention is to use the second 4-KB page above the 64-KB SLB. The PAL code, when built using the process described below, can be executed with Flicker protections. Its message will be available from the `outputs` entry in the *flicker-module* sysfs location. Thus the application can simply use `open` and `read` to obtain the PAL's results. Note that the `outputs` entry is a fixed-size 4 KB binary file, and it is up to the application developer to interpret it appropriately. In this case, it would be read as a 4096 byte file containing the characters "Hello, world" followed by 4084 NULLs.

Module	Properties	LOC	Size (KB)
SLB Core	Prepare environment, execute PAL, clean environment, resume OS	94	0.312
OS Protection	Memory protection, ring 3 PAL execution	5	0.046
TPM Driver	Communication with the TPM	216	0.825
TPM Utilities	Performs TPM operations, e.g., Seal, Unseal, GetRand, PCR Extend	889	9.427
Crypto	General purpose cryptographic operations, RSA, SHA-1, SHA-512 etc.	2262	31.380
Memory Management	Implementation of malloc/free/realloc	657	12.511
Secure Channel	Generates a keypair, seals private key, returns public key	292	2.021

Figure 4.8: **Existing Flicker Modules.** *Modules that can be included in the* PAL*. Only the SLB Core is mandatory. Each adds some number of lines of code (LOC) to the* PAL*'s TCB and contributes to the overall size of the SLB binary.*

4.3.1.2 Building a PAL

To convert the code from Figure 4.7 into a PAL, we link it against the object file representing Flicker's core functionality (described as SLB Core below) using the Flicker linker script. The linker script specifies that the skeleton data structures and code from the SLB Core should come first in the resulting binary, and that the resulting output format should be binary (as opposed to an ELF executable). The application then provides this binary blob to the *flicker-module* for execution under Flicker's protection.

Application developers depend on a variety of libraries. There is no reason this should be any different just because the target executable is a PAL, except that it is desirable to modularize the libraries further than is traditionally done to help minimize the amount of code included in the PAL's TCB. We have developed several small libraries in the course of applying Flicker to the applications described in Section 4.4. The following paragraphs provide a brief description of the libraries listed in Figure 4.8.

SLB Core. The SLB Core module provides the minimal functionality needed to support a PAL. Section 4.2.2 describes this functionality in detail. In brief, the SLB Core contains space for the SLB's entry point, length, GDT, TSS, and code to manage segment descriptors and page tables. The SLB Core transfers control to the PAL code, which performs application-specific work. When the PAL terminates, it transfers control back to the SLB Core for cleanup and resumption of the OS.

OS Protection. Thus far, Flicker has focused on protecting a security-sensitive PAL from all of the other software on the system. However, we have also developed a module to protect a legitimate OS from a malicious or malfunctioning PAL. It is important to note that since *SKINIT* is a privileged instruction, only code executing at CPU protection ring 0 (recall that x86 has 4 privilege rings, with 0 being most privileged) can invoke a Flicker session. Thus, the OS ultimately decides which PALs to run, and presumably it will only run PALs that it trusts or has verified in some manner, e.g., using proof carrying code [148]. Nonetheless, the OS may desire additional guarantees. The OS Protection module restricts a PAL's memory accesses to the exact memory region allocated by the OS, thus preventing it from intentionally or inadvertently reading or overwriting the code and/or data of other software on the system. We are also investigating techniques to limit a PAL's execution time using timer interrupts in the SLB Core. These timing restrictions must be chosen carefully, however, since a PAL may need some minimal amount of time to allow TPM operations to complete before the PAL can accomplish any meaningful work.

To restrict the memory accessed by a PAL, we use segmentation and run the PAL in CPU protection ring 3. Essentially, the SLB Core creates segment descriptors for the PAL that have a base address set at the beginning of the PAL and a limit placed at the end of the memory region allocated by the OS. The SLB Core then runs the PAL in ring 3 to prevent it from modifying or otherwise circumventing these protections. When the PAL exits, it transitions back to the SLB Core running in ring 0. The SLB Core can then cleanse the memory region used and reload the OS.

In more detail, we transition from the SLB Core running in ring 0 to the PAL running in ring 3 using the *IRET* instruction which loads the `slb_base`-offset segment descriptors before the PAL executes. Executing the PAL in ring 3 only requires two additional *PUSH* instructions in the SLB Core. Returning execution to ring 0 once the PAL terminates involves the use of the call gate and task state segment (TSS) in the GDT. This mechanism is invoked with a single (far) call instruction in the SLB Core.

TPM Driver and Utilities. The TPM is a memory-mapped I/O device. As such, it needs a small amount of driver functionality to keep it in an appropriate state and to ensure that its buffers never over- or underflow. This driver code is necessary before any TPM operations can be performed, and it is also necessary to release control of the TPM when the Flicker session is ready to exit, so that the Linux TPM driver can regain access to the TPM.

The TPM Utilities allow other PAL code to perform useful TPM operations. Currently supported operations include GetCapability, PCR Read, PCR Extend, GetRandom, Seal, Unseal, and the OIAP and OSAP sessions necessary to authorize Seal and Unseal [200].

Crypto. We have developed a small library of cryptographic functions based partially on code from version 0.58 of the Putty SSH client[4] and partially on code from the PolarSSL library.[5] Supported operations include a multi-precision integer library, RSA key generation, RSA encryption and decryption, SHA-1, SHA-512, MD5, AES, and RC4.

Memory Management. We have implemented a small version of malloc/free/realloc for use by applications. The memory region used as the heap is simply a large global buffer.

Secure Channel. We have implemented the protocol described in Section 4.2.4 for creating a secure channel into a PAL from a remote party. It relies on all of the other modules we have developed (except the OS Protection module which the developer may add).

4.3.2 Automation

Ideally, we envision each PAL containing only the security-sensitive portion of each application, rather than the application in its entirety. Minimizing the PAL makes it easier to ensure that the required functionality is performed correctly and securely, facilitating a remote party's verification task. Previous research indicates that many applications can be readily split into a privileged and an unprivileged component. Such privilege separation can be performed manually [100, 112, 160, 196], or automatically [20, 35, 219].

While each PAL is necessarily application-specific, we have developed a tool using the source-code analysis tool CIL [149] to help extract functionality from existing programs. Since CIL can replace the C compiler (e.g., the programmer can simply run "`CC=cil make`" using an existing Makefile), our tool can operate even on large programs with complex build dependencies.

The programmer supplies our tool with the name of a target function within a larger program (e.g., `rsa_keygen()`). The tool then parses the program's call graph and extracts any functions that the target depends on, along with relevant type definitions, etc., to create a standalone C program. The tool also indicates which additional functions from standard libraries must be eliminated or replaced. For example, by default, a PAL cannot call `printf` or `malloc`. Since `printf` usually does not make sense for a PAL, the programmer can simply eliminate the call. For `malloc`, the programmer can convert the code to use statically allocated variables or link against our memory management library (described above). While the process is clearly not completely automated, the tool does automate a large portion of PAL creation and eases the programmer's burden, and we continue to work on increasing

[4] http://www.putty.nl/
[5] http://www.polarssl.org/

the degree of automation provided. We found the tool useful in our application of Flicker to the applications described next.

4.4 Flicker Applications

In this section, we demonstrate the versatility of the Flicker platform by showing how Flicker can be applied to several broad classes of applications. We consider applications that do not require local state, applications that require local state whose integrity must be protected, and finally, applications with state that must be both secret and integrity-protected. Within each class, we describe our implementation of one or more applications and show how Flicker significantly enhances security in each case. In Section 4.5, we evaluate the performance of the applications, as well as the general Flicker platform.

We have implemented Flicker for AMD SVM on a 32-bit Linux kernel v2.6.20, including the various modules described in Section 4.3. Each application described below utilizes precisely the modules needed (and some application-specific logic) and nothing else. On the untrusted OS, the *flicker-module* loadable kernel module is responsible for invoking the PAL and facilitating delivery of inputs and reception of outputs from the Flicker session. Further, it manages the suspension and resumption of the untrusted OS before and after the Flicker session. We also developed a TPM Quote Daemon (the *tqd*) on top of the TrouSerS[6] TCG Software Stack that runs on the untrusted OS and provides an attestation service.

4.4.1 Stateless Applications

Many applications do not require long-term state to operate effectively. For these applications, the primary overhead of using Flicker is the time required for the *SKINIT* instruction, since the attestation can be generated by the untrusted OS (see Section 4.2.4.1). As a concrete example, we use Flicker to provide verifiable isolated execution of a kernel rootkit detector on a remote machine.

For this application, we assume a network administrator wishes to run a rootkit detector on remote hosts that are potentially compromised. For instance, a corporation may wish to verify that employee laptops have not been compromised before allowing them to connect to the corporate Virtual Private Network (VPN).

We implement our rootkit detector for version 2.6.20 of the Linux kernel as a PAL. After the SLB Core hands control to the rootkit detector PAL, it computes a SHA-1 hash of the

[6] http://trousers.sourceforge.net/

kernel text segment, system call table, and loaded kernel modules. The detector then extends the resulting hash value into PCR 17 and copies it to the standard output memory location. Once the PAL terminates, the untrusted OS resumes operation and the *tqd* provides an attestation to the network administrator. Since the attestation contains the TPM's signature on the current PCR values, the administrator knows that the correct rootkit detector ran with Flicker protections in place and can verify that the untrusted OS returns the correct value. Finally, the administrator can compare the hash value returned against known-good values for that particular kernel.

4.4.2 Integrity-Protected State

Some applications may require multiple Flicker sessions, and hence a means of preserving state across sessions. For some, simple integrity protection of this state will suffice (we consider those that also require secrecy in Section 4.4.3). To illustrate this class of applications, we apply Flicker to a distributed computing application.

Applications such as SETI@Home [10] divide a task into smaller work units and distribute these units to hosts with spare computation capacity. When the hosts are untrusted, the application must take measures to detect erroneous results. A common approach distributes the same work unit to multiple hosts and compares the results. Unfortunately, this wastes significant amounts of computation, and does not provide any tangible correctness guarantees [145]. With Flicker, the clients can process their work units inside a Flicker session and attest the results to the server. The server then has a high degree of confidence in the results and need not waste computation on redundant work units.

In our implementation, we apply Flicker to the BOINC framework [9], which is a generic framework for distributed computing applications. It is currently used by several dozen projects.[7] By targeting BOINC, rather than a specific application, we can allow all of these applications to take advantage of Flicker's security properties (though some amount of application-specific modifications are still required). As an illustration, we developed a simple distributed application using the BOINC framework that attempts to factor a large number by naively asking clients to test a range of numbers for potential divisors.

In this application, our modified BOINC client contacts the server to obtain a work unit. It then invokes a Flicker session to perform application specific work. Since the PAL may have to compute for an extended period of time, it periodically returns control to the untrusted OS. This allows the OS to process interrupts (including a user's return to the computer) and multitask with other programs.

[7] http://boinc.berkeley.edu/projects.php

Since many distributed computing applications care primarily about the integrity of the result, rather than the secrecy of the intermediate state, our implementation focuses on maintaining the integrity of the PAL's state while the untrusted OS operates. To do so, the very first invocation of the BOINC PAL generates a 160-bit symmetric key based on randomness obtained from the TPM and uses the TPM to seal the key so that no other code can access it. It then performs application specific work.

Before yielding control back to the untrusted OS, the PAL computes a cryptographic MAC (HMAC) over its current state (for the factoring application, the state is simply the current prospective divisor and any successful divisors found thus far). Each subsequent invocation of the PAL unseals the symmetric key and checks the MAC on its state before beginning application-specific work. When the PAL finally finishes its work unit, it extends the results into PCR 17 and exits. Our modified BOINC client then returns the results to the server, along with an attestation. The attestation demonstrates that the correct BOINC PAL executed with Flicker protections in place and that the returned result was truly generated by the BOINC PAL. Thus, the application writer can trust the result.

4.4.3 Secret and Integrity-Protected State

Finally, we consider applications that need to maintain both the secrecy and the integrity of their state between Flicker invocations. To evaluate this class of applications, we developed two additional applications. The first uses Flicker to protect SSH passwords, and the second uses Flicker to protect a Certificate Authority's private signing key.

4.4.3.1 SSH Password Authentication

We have applied Flicker to password-based authentication with SSH. Since people tend to use the same password for multiple independent computer systems, a compromise on one system may yield access to other systems. Our primary goal is to prevent any malicious code on the server from learning the user's password, even if the server's OS is compromised. Our secondary goal is to convince the client system (and hence, the user) that the secrecy of the password has been preserved. Flicker is well suited to these goals, as it makes it possible to restrict access to the user's cleartext password on the server to a tiny TCB (the PAL), and to attest to the client that this indeed was enforced. While other techniques (e.g., PwdHash [163]) exist to ensure varied user passwords across servers, SSH provides a useful illustration of Flicker's properties when applied to a real-world system.

Client:	has K_{PAL}
Server:	has $sdata$, $salt$, $hashed\_passwd$
	generates $nonce$
Server → Client:	$nonce$
Client:	user inputs $password$
	$c \leftarrow$encrypt$_{K_{\mathrm{PAL}}}(\{password, nonce\})$
Client → Server:	c
Server → PAL:	c, $salt$, $sdata$, $nonce$
PAL:	$K_{\mathrm{PAL}}^{-1} \leftarrow$ unseal($sdata$)
	$\{password, nonce'\} \leftarrow$ decrypt$_{K_{\mathrm{PAL}}^{-1}}(c)$
PAL:	if ($nonce' \neq nonce$)
	then abort
	$hash \leftarrow$ md5crypt($salt$, $password$)
	extend($PCR17, \perp$)
PAL → Server:	$hash$
Server:	if ($hash = hashed\_passwd$)
	then allow_login
	else abort

Figure 4.9: **SSH Password Checking Protocol.** *The protocol surrounding the second Flicker session for our SSH implementation. $sdata$ contains the sealed private key, K_{PAL}^{-1}. Variables $salt$ and $hashed\_passwd$ are components of the entry in the system's `/etc/passwd` file for the user attempting to log in. The $nonce$ serves to prevent replay attacks against a well-behaved server.*

Our implementation is built upon the basic components we have described in the preceding sections, and consists of five main software components. A modified SSH client runs on the client system. The client system does not need hardware support for Flicker, but a compromise of the client may leak the user's password. We are investigating techniques for utilizing Flicker on the client side. We add a new client authentication method, *flicker-password*, to OpenSSH version 4.3p2. The *flicker-password* module establishes a secure channel to the PAL on the server using the protocol described in Section 4.2.4.2 and implements the client portion of the protocol shown in Figure 4.9.

The other four components, a modified SSH server daemon, the *flicker-module* kernel module, the *tqd*, and the SSH PAL, all run on the server system. Below, we describe the two Flicker sessions used to protect the user's password on the server.

First Flicker Session (Setup). The first session uses our Secure Channel module to provide the client's computer with a secure channel for sending the user's password to the second Flicker session, which will perform the password check.

In more detail, the Secure Channel module conveys a public key K_{PAL} to the client in such a way that the client is convinced that the corresponding private key is accessible only to the same PAL in a subsequent Flicker session. Thus, by verifying the attestation from the first Flicker session, the client is convinced that the correct PAL executed, that the legitimate PAL created a fresh keypair, and that the SLB Core erased all secrets before returning control to the untrusted OS. Using its authentic copy of K_{PAL}, the client encrypts the user's password for transmission to the second Flicker session on the server. We use PKCS1 encryption which is chosen-ciphertext-secure and nonmalleable [103]. The end-to-end encryption of the user's password, from the client system all the way into the PAL, protects the user's password in the event that any of the server's software, potentially including the OS, the sshd server software or the *flicker-module* kernel module, is malicious.

Second Flicker Session (Login). The second Flicker session processes the user's encrypted password and outputs a hash of the (unencrypted) password for comparison with the user's login information in the server's password file (see Figure 4.9).

When the second session begins, the PAL uses TPM Unseal to retrieve its private key K_{PAL}^{-1} from $sdata$. It then uses the key to decrypt the user's password. Finally, the PAL computes the hash of the user's password and salt[8] and outputs the result for comparison with the server's password file. The end result is that the user's unencrypted password only exists on the server during a Flicker session.

No attestation is necessary after the second Flicker session because, thanks to the properties of Flicker and sealed storage, the client knows that K_{PAL}^{-1} is inaccessible unless the correct PAL is executing within a Flicker session.

Instead of outputting the hash of the password, an alternative implementation could keep the entire password file in sealed storage between Flicker sessions. This would prevent dictionary attacks, but make the password file incompatible with local logins.

An obvious optimization of the authentication procedure described above is to only create a new keypair the first time a user connects to the server. Between logins, the sealed private key can be kept at the server, or it could even be given to the user to be provided during the next login attempt. If the user loses this data (e.g., if she uses a different client machine) or provides invalid data, the PAL can simply create a new keypair, at the cost of some additional latency for the user.

[8] Most *nix systems compute the hash of the user's password concatenated with a "salt" value and store the resulting hash value in an authentication file (e.g., `/etc/passwd`).

4.4.3.2 Certificate Authority

Our final application, a Flicker-enhanced Certificate Authority (CA), is similar to the SSH application but focuses on protecting the CA's private signing key. The benefit of using Flicker is that only a tiny piece of code ever has access to the CA's private signing key. Thus, the key will remain secure, even if all of the other software on the machine is compromised. Of course, malevolent code on the server may submit malicious certificates to the signing PAL. However, the PAL can implement arbitrary access control policies on certificate creation and can log those creations. Once the compromise is discovered, any certificates incorrectly created can be revoked. In contrast, revoking a CA's public key, as would be necessary if the private key were compromised, is a more heavyweight proposition in many settings.

In our implementation, one PAL session generates a 1024-bit RSA keypair using randomness from the TPM and seals the private key under PCR 17. The public key is made generally available. The second PAL session takes in a certificate signing request (CSR). It uses TPM Unseal to obtain its private key and certificate database. If the access control policy supplied by an administrator approves the CSR, then the PAL signs the certificate, updates the certificate database, reseals it, and outputs the signed certificate.

4.5 Performance Evaluation

Below, we describe our experimental setup and evaluate the performance of the Flicker platform via microbenchmarks, as well as via macrobenchmarks of the various application classes described in Section 4.4. We also measure the impact of Flicker sessions on the rest of the system, e.g., the untrusted OS and applications.

While the overhead for several applications is significant, Section 4.6 identifies several hardware modifications that can potentially improve performance by up to six orders of magnitude. Thus, it is reasonable to expect significantly improved performance in future versions of this technology.

4.5.1 Experimental Setup

Our primary test machine is an HP dc5750 which contains an AMD Athlon64 X2 Dual Core 4200+ processor running at 2.2 GHz, and a v1.2 Broadcom BCM0102 TPM. In experiments requiring a remote verifier, we use a generic PC with a CPU running at 1.6 GHz. The remote verifier is 12 hops away (determined using traceroute) with minimum, maximum, and average ping times of 9.33 ms, 10.10 ms, and 9.45 ms over 50 trials.

TPM	CPU Vendor	System Configuration	PAL Size					
			0 KB	4 KB	8 KB	16 KB	32 KB	64 KB
Yes	AMD	HP dc5750 Avg (ms):	0.00	11.94	22.98	45.05	89.21	177.52
No		Tyan n3600R Avg (ms):	0.01	0.56	1.11	2.21	4.41	8.82
Yes	Intel	TEP Avg (ms):	26.39	26.88	27.38	28.37	30.46	34.35

Figure 4.10: ***SKINIT* and *SENTER* Benchmarks.** *We run SKINIT benchmarks on AMD systems with and without a TPM to isolate the overhead of the SKINIT instruction from the overhead induced by the TPM. We also run SENTER benchmarks on an Intel machine with a TPM.*

All of our timing measurements were performed using the *RDTSC* instruction to count CPU cycles. We converted cycles to milliseconds based on each machine's CPU speed, obtained by reading `/proc/cpuinfo`.

4.5.2 Microbenchmarks

We begin by performing a number of microbenchmarks to measure the time needed by late launch and various TPM operations on two AMD machines and one Intel machine.

In addition to the AMD HP dc5750 described above, we employ a second AMD test machine based on a Tyan n3600R server motherboard with two 1.8 GHz dual-core Opteron processors. This second machine is not equipped with a TPM, but it does support execution of *SKINIT*. This allows us to isolate the performance of *SKINIT* without the potential bottleneck of a TPM. Our Intel test machine is an MPC ClientPro Advantage 385 TXT Technology Enabling Platform (TEP), which contains a 2.66 GHz Core 2 Duo processor, an Atmel v1.2 TPM, and the DQ965CO motherboard.

Since we have observed that the performance of different TPM implementations varies considerably, we also evaluate the TPM performance of two other machines with a v1.2 TPM: a Lenovo T60 laptop with an Atmel TPM, and an AMD workstation with an Infineon TPM.

4.5.2.1 Late Launch with an AMD Processor

AMD SVM supports late launch via the *SKINIT* instruction. The overhead of the *SKINIT* instruction can be broken down into three parts: (1) the time to place the CPU in an appropriate state with protections enabled, (2) the time to transfer the PAL to the TPM across the low pin count (LPC) bus, and (3) the time for the TPM to hash the PAL and extend the hash into PCR 17. To investigate the breakdown of the instruction's performance overhead, we ran the *SKINIT* instruction on the HP dc5750 (with TPM) and the Tyan n3600R (without TPM) with PALs of various sizes. We invoke *RDTSC* before executing *SKINIT* and invoke it a second time as soon as code from the PAL can begin executing.

Figure 4.10 summarizes the timing results. The measurements for the empty (0 KB) PAL indicate that placing the CPU in an appropriate state introduces relatively little overhead (less than 10 μs). The Tyan n3600R (without TPM) allows us to measure the time needed to transfer the PAL across the LPC bus. The maximum LPC bandwidth is 16.67 MB/s, so the fastest possible transfer of 64 KB is 3.8 ms [94]. Our measurements agree with this prediction, indicating that it takes about 8.8 ms to transfer a 64 KB PAL, with the time varying linearly for smaller PALs.

Unfortunately, our results for the HP dc5750 indicate that the TPM introduces a significant delay to the *SKINIT* operation. We investigated the cause of this overhead and identified the TPM as causing a reduction in throughput on the LPC bus. The TPM slows down *SKINIT* runtime by causing *long wait cycles* on the LPC bus. *SKINIT* sends the contents of the PAL to a TPM to be hashed using the following TPM command sequence: `TPM_HASH_START`, zero or more invocations of `TPM_HASH_DATA` (each sends one to four bytes of the PAL to the TPM), and finally `TPM_HASH_END`. The TPM specification states that each of these commands may take up the entire *long wait cycle* of the control flow mechanism built into the LPC bus that connects the TPM [199]. Our results suggest that the TPM is indeed utilizing most of the *long wait cycle* for each of the commands, and as a result, the TPM contributes almost 170 ms of overhead. This may be either a result of the TPM's low clock rate or an inefficient implementation, and is not surprising given the low-cost nature of today's TPM chips. The 8.82 ms taken by the Tyan n3600R may be representative of the performance of future TPMs which are able to operate at maximum bus speed.

4.5.2.2 Late Launch with an Intel Processor

Recall from Section 2.4.2 that Intel's late launch consists of two phases. In the first phase, the ACMod is extended into PCR 17 using the same `TPM_HASH_START`, `TPM_HASH_DATA`, and `TPM_HASH_END` command sequence used by AMD's *SKINIT*. In the second phase, the ACMod hashes the PAL on the main CPU and uses an ordinary `TPM_Extend` operation to record the PAL's identity in PCR 18. Thus, only the 20 byte hash of the PAL is passed across the LPC to the TPM in the second phase.

The last row in Figure 4.10 presents experimental results from invoking *SENTER* on our Intel TEP. Interestingly, the overhead of *SENTER* is initially quite high, and it grows linearly but slowly. The large initial overhead (26.39 ms) results from two factors. First, even for a 0 KB PAL, the Intel platform must transmit the entire ACMod to the TPM and wait for the TPM to hash it. The ACMod is just over 10 KB, which matches nicely with the fact that the initial overhead falls in between the overhead for an *SKINIT* with PALs of size 8 KB

(22.98 ms) and 16 KB (45.05 ms). The overhead for *SENTER* also includes the time necessary to verify the signature on the ACMod.

The slow increase in the overhead of *SENTER* relative to the size of the PAL is a result of where the PAL is hashed. On an Intel platform, the ACMod hashes the PAL on the main CPU and hence sends only a constant amount of data across the LPC bus. In contrast, an AMD system must send the entire PAL to the TPM and wait for the TPM to do the hashing.[9] Figure 4.10 suggests that for large PALs, Intel's implementation decision pays off. Further reducing the size of the ACMod would improve Intel's performance even more. The gradual increase in *SENTER*'s runtime with increase in PAL size is most likely attributable to the hash operation performed by the ACMod.

On an Intel TXT platform, the ACMod verifies that system configuration is acceptable, enables chipset protections such as the initial memory protections for the PAL, and then measures and launches the PAL [78]. On AMD SVM systems, microcode likely performs similar operations, but we do not have complete information about AMD CPUs. Since Intel TXT measures the ACMod into a PCR, an Intel TXT attestation to an external verifier may contain more information about the challenged platform and may allow an external verifier to make better trust decisions.

4.5.2.3 Trusted Platform Module (TPM) Operations

Though Intel and AMD send different modules of code to the TPM using the `TPM_HASH_*` command sequence, this command sequence is responsible for the majority of late launch overhead. More significant to overall PAL overhead, however, is Flicker's use of the TPM's sealed storage capabilities to protect PAL state during a context switch. To better understand these overheads, we perform TPM benchmarks on four different TPMs. Two of these are the TPMs in our already-introduced HP dc5750 and Intel TEP. The other two TPMs are an Atmel TPM (a different model than that included in our Intel TEP) in an IBM T60 laptop, and an Infineon TPM in an AMD system.

We evaluate the time needed for relevant operations across several different TPMs. These operations are: PCR Extend, Seal, Unseal, Quote, and GetRandom. Figure 4.11 shows the results of our TPM microbenchmarks. The results show that different TPM implementations optimize different operations. The Broadcom TPM in our primary test machine is the

[9] There is no technical reason why a PAL for an AMD system cannot be written in two parts: one that is measured as part of *SKINIT* and another that is measured by the first part before it receives control. This will enable a PAL on AMD systems to achieve improved performance, and suggests that AMD's mechanism is more flexible than Intel's.

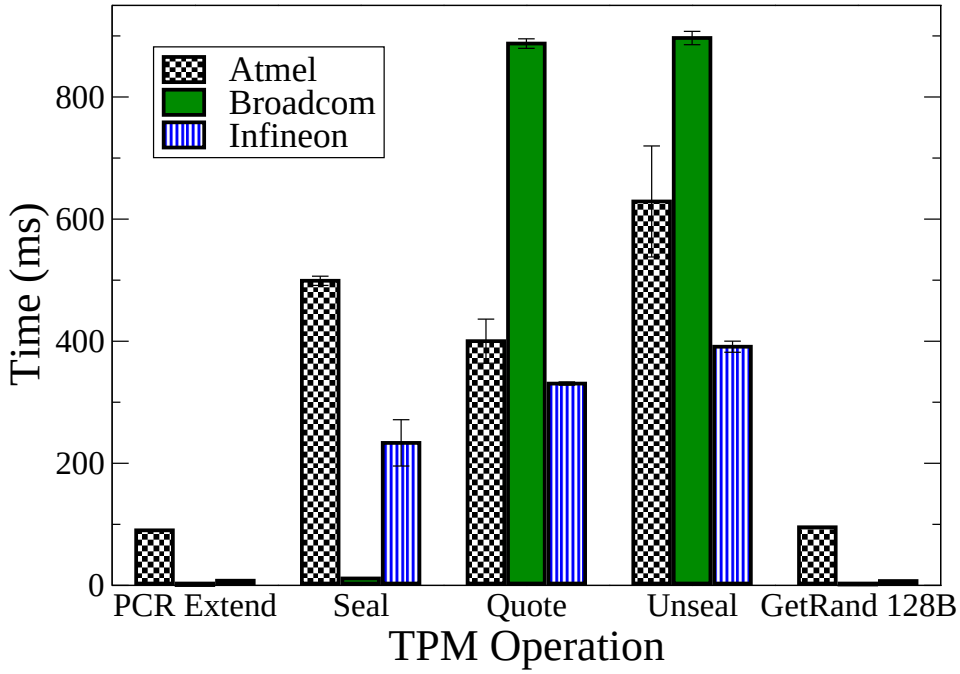


Figure 4.11: **TPM Microbenchmarks.** *TPM benchmarks run against the Atmel v1.2 TPM in a Lenovo T60 laptop, the Broadcom v1.2 TPM in an HP dc5750, the Infineon v1.2 TPM in an AMD machine, and the Atmel v1.2 TPM (note that this is not the same as the Atmel TPM in the Lenovo T60 laptop) in the Intel TEP. Error bars indicate the standard deviation over 20 trials (not all error bars are visible).*

slowest for Quote and Unseal. Switching to the Infineon TPM (which has the best average performance across the relevant operations) would reduce the TPM-induced overhead for a combined Quote and Unseal by 1132 ms, although it would also add 213 ms of Seal overhead. Even if we choose the best performing TPM for each operation (which is not necessarily technically feasible, since a speedup on one operation may entail a slowdown in another), a PAL Gen would still require almost 200 ms (177 ms for *SKINIT* and 20.01 ms for the Broadcom Seal), and a PAL Use could require at least 579.37 ms (177 ms for *SKINIT*, 390.98 ms for the Infineon Unseal, and 11.39 ms for the Broadcom Seal). These values indicate that TPM-based context-switching is extremely heavy-weight.

4.5.3 Stateless Applications

We evaluate the performance of the rootkit detector by measuring the total time required to execute a detection query. We perform additional experiments to break down the various components of the overhead involved. Finally, we measure the impact of regular runs of the rootkit detector on overall system performance.

Operation	Time (ms)
SKINIT	15.4
PCR Extend	1.2
Hash of Kernel	22.0
TPM Quote	972.7
Total Query Latency	1022.7

Figure 4.12: **Breakdown of Rootkit Detector Overhead.** *The first three operations occur during the Flicker session, while the TPM Quote is generated by the OS. The standard deviation was negligible for all operations.*

End-to-End Performance. We begin by evaluating the total time required for an administrator to run our rootkit detector on a remote machine. Our first experiment measures the total time between the time the administrator initiates the rootkit query on the remote verifier and the time the response returns from the AMD test machine. Over 25 experiments, the average query time was 1.02 seconds, with a standard deviation of less than 1.4 ms. This relatively small latency suggests that it would be reasonable to run the rootkit detector on remote machines before allowing them to connect to the corporate VPN, for example.

Microbenchmarks. To better understand the overhead of the rootkit detector, we performed additional microbenchmarks to determine the most expensive operations involved (see Figure 4.12). The results indicate that the highest overhead comes from the TPM Quote operation. This performance is TPM-specific. Other TPMs contain faster implementations (see Figure 4.11); for example, an Infineon TPM can generate a quote in under 331 ms. To reduce the overhead of *SKINIT*, we developed the following optimization.

***SKINIT* Optimization.** Short of changing the speed of the TPM and the bus through which the CPU communicates with the TPM, Figure 4.10 indicates that the best opportunity for improving the performance of *SKINIT* is to reduce the size of the SLB. To maintain the security properties provided by *SKINIT*, however, code in the SLB must be measured before it is executed. Note that *SKINIT* enables the Device Exclusion Vector for the entire 64 KB of memory starting from the base of the SLB, even if the SLB's length is less than 64 KB. One viable optimization is to create a PAL that only includes a cryptographic hash function and enough TPM support to perform a PCR Extend. This PAL can then measure and extend the application-specific PAL. A PAL constructed in this way offloads most of the burden of computing code measurement to the system's main CPU. We have constructed such a PAL in 4736 bytes. When this PAL runs, it measures the entire 64 KB and extends the resulting measurement into PCR 17. Thus, when *SKINIT* executes, it only needs to transfer 4736 bytes to the TPM. In 50 trials, we found the average *SKINIT* time to be 14 ms. While only a small

Detection Period [m:s]	Benchmark Time [m:s]	Standard Deviation [s]
No Detection	7:22.6	2.6
5:00	7:21.4	1.1
3:00	7:21.4	0.9
2:00	7:21.8	1.0
1:00	7:21.9	1.1
0:30	7:22.6	1.7

Figure 4.13: **Impact of the Rootkit Detector.** *Kernel build time when run with no detection and with rootkit detection run periodically. Note that the detection does not actually speed up the build time; rather the small performance impact it does have is lost in experimental noise.*

savings for the rootkit detector, it saves 164 ms of the 176 ms *SKINIT* requires with a 64-KB SLB. We use this optimization in the rest of our applications.

System Impact. As a final experiment, we evaluate the rootkit detector's impact on the system by measuring the time required to build the 2.6.20 Linux kernel while also running the rootkit detector periodically. Figure 4.13 summarizes our results. Essentially, our results suggest that even frequent execution of the rootkit detector (e.g., once every 30 seconds) has negligible impact on the system's overall performance.

4.5.4 Integrity-Protected State

At present, our distributed computing PAL periodically exits to check whether the main system has work to perform. The frequency of these checks represents a tradeoff between low latency in responding to system events (such as a user returning to the computer) and efficiency of computation (the percentage of time performing useful, application-specific computation), since the Flicker-induced overhead is experienced every time the application resumes its work.

In our experiments, we evaluate the amount of Flicker-imposed overhead by measuring the time required to start performing useful application work, specifically, between the time the OS executes *SKINIT*, and the time at which the PAL begins to perform application-specific work.

Figure 4.14 shows the resulting overhead, as well as its most expensive constituent operations, in particular, the time for the *SKINIT*, and the time to unseal and verify the PAL's previous state.[10] The table demonstrates how the application's efficiency improves as we al-

[10] As described in Section 4.4.2, the initial PAL must also generate a symmetric key and seal it under PCR 17. We discuss this overhead in more detail in Section 4.5.5.

Operation	Time (ms)			
Application Work	1000	2000	4000	8000
SKINIT	14.3	14.3	14.3	14.3
Unseal	898.3	898.3	898.3	898.3
Flicker Overhead	**47%**	**30%**	**18%**	**10%**

Figure 4.14: **Operations for Distributed Computing.** *This table indicates the significant expense of the Unseal operation, as well as the tradeoff between efficiency and latency.*

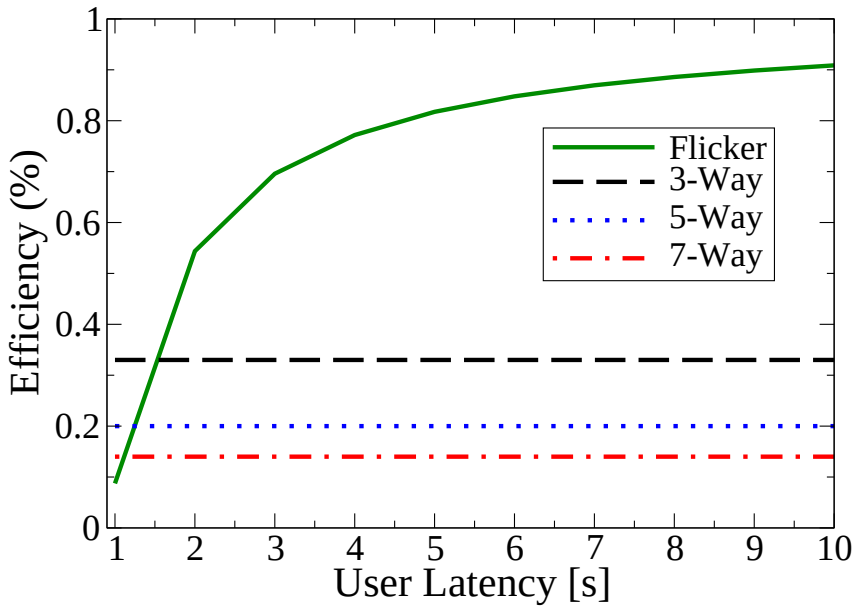


Figure 4.15: **Flicker vs. Replication Efficiency.** *Replicating to a given number of machines represents a constant loss in efficiency. Flicker gains efficiency as the length of the periods during which application work is performed increases.*

low the PAL to run for longer periods of time before exiting back to the untrusted OS. For example, if the application runs for one second before returning to the OS, only 53% of the Flicker session is spent on application work; the remaining 47% is consumed by Flicker's setup time. However, if we allow the application to run to two or four seconds at a time, then Flicker's overhead drops to only 30% or 18%, respectively. Figure 4.14 also indicates that the vast majority of the overhead arises from the TPM's Unseal operation. Again, a faster TPM, such as the Infineon, can unseal in under 400 ms.

While Flicker adds additional overhead on a single client, the true savings come from the higher degree of trust the application writer can place in the results returned. Figure 4.15 illustrates this savings by comparing the efficiency of Flicker-enhanced distributed comput-

Operation	Time (ms)
SKINIT	14.3
Key Gen	185.7
Seal	10.2
Total Time	217.1

(a) **PAL 1**

Operation	Time (ms)
SKINIT	14.3
Unseal	905.4
Decrypt	4.6
Total Time	937.6

(b) **PAL 2**

Figure 4.16: **SSH Performance Overhead.** *Average server side performance over 100 trials, including a breakdown of time spent inside each* PAL. *The standard error on all measurements is under 1%, except key generation at 14%.*

ing with the standard solution of using redundancy. With our current implementation, a two second user latency allows a more efficient distributed application than replicating to three or more machines. As the performance of this new hardware improves, the efficiency of using Flicker will only increase.

4.5.5 Secret and Integrity-Protected State

Since both SSH and the CA perform similar activities, we focus on the modified SSH implementation and then highlight places where the CA differs.

4.5.5.1 SSH Password Authentication

Our first set of experiments measures the total time required for each PAL on the server. The quote generation, seal and unseal operations are performed on the TPM using 2048-bit asymmetric keys, while the key generation and the password decryption are performed by the CPU using 1024-bit RSA keys.

Figure 4.16 presents these results, as well as a breakdown of the most expensive operations that execute on the SSH server. The total time elapsed on the client between the establishment of the TCP connection with the server, and the display of the password prompt for the user is 1221 ms (this includes the overhead of the first PAL, as well as 949 ms for the TPM Quote operation), compared with 210 ms for an unmodified server. Similarly, the time elapsed beginning immediately after password entry on the client, and ending just before the client system presents the interactive session to the user is approximately 940 ms while the unmodified server only requires 10 ms. The primary source of overhead is clearly the TPM. As these devices have just been introduced by hardware vendors and have not yet proven themselves in the market, it is not surprising that their performance is poor. Nonetheless,

current performance suffices for lightly-loaded servers, or for less time-critical applications, such as the CA.

During the first PAL, the 1024-bit key generation clearly imposes the largest overhead. This cost could be mitigated by choosing a different public key algorithm with faster key generation, such as ElGamal, and is readily parallelized. Both Seal and *SKINIT* contribute overhead, but compared to the key generation, they are relatively insignificant. We also make one call to TPM GetRandom to obtain 128 bytes of random data (it is used to seed a pseudorandom number generator), which averages 1.3 ms. The performance of PCR Extend is similarly quick and takes less than 1 ms on the Broadcom TPM.

Quote is an expensive TPM operation, averaging 949 ms, but it is performed while the untrusted OS has control. Thus, it is experienced as a latency only for the SSH client. It does not impact the performance of other processes running on the SSH server, as long as they do not require access to the TPM.

The second PAL's main overhead comes from the TPM Unseal. As mentioned above, the Unseal overhead is TPM-specific. An Infineon TPM can Unseal in 391 ms.

4.5.5.2 Certificate Authority

For the CA, we measure the total time required to sign a certificate request. In 100 trials, the total time averaged 906.2 ms (again, mainly due to the TPM's Unseal). Fortunately, the latency of the signature operation is far less critical than the latency in the SSH example. The components of the overhead are almost identical to the SSH server's, though in the second PAL, the CA replaces the RSA decrypt operation with an RSA signature operation. This requires approximately 4.7 ms.

4.5.6 Impact on Suspended Operating System

Flicker runs with the legacy OS suspended and interrupts disabled. We have presented Flicker sessions that run for more than one second, e.g., in the context of a distributed computing application (Figure 4.14). While these are long times to keep the OS suspended and interrupts disabled, we have observed relatively few problems in practice. We relate some of our experience with Flicker, and then describe the options available today to reduce Flicker's impact on the suspended system. Finally, we introduce some recommendations to modify today's hardware architecture to better support Flicker.

While a Flicker session runs, the user will perceive a hang on the machine. Keyboard and mouse input during the Flicker session may be lost. Such responsiveness glitches sometimes

occur even without Flicker, and while unpleasant, they do not put valuable data at risk. Likewise, network packets are sometimes lost even without Flicker, and today's network-aware applications can and do recover. The most significant risk to a system during a Flicker session is lost data in a transfer involving a block device, such as a hard drive, CD-ROM drive, or USB flash drive.

We have performed experiments on our HP dc5750 copying large files while the distributed computing application runs repeatedly. Each run lasts an average of 8.3 seconds, and the legacy OS runs for an average of 37 ms in between. We copy files from the CD-ROM drive to the hard drive, from the CD-ROM drive to the USB drive, from the hard drive to the USB drive, and from the USB drive to the hard drive. Between file copies, we reboot the system to ensure cold caches. We use a 1-GB file created from `/dev/urandom` for the hard drive to/from USB drive experiments, and a CD-ROM containing five 50-200-MB Audio-Video Interleave (AVI) files for the CD-ROM to hard drive / USB drive experiments. During each Flicker session, the distributed computing application performs a TPM Unseal and then performs division on 1,500,000 possible factors of a 384-bit prime number. In these experiments, the kernel did not report any I/O errors, and integrity checks with `md5sum` confirmed that the integrity of all files remained intact.

To provide stronger guarantees for the integrity of device transfers on a system that supports Flicker, these transfers should be scheduled such that they do not occur during a Flicker session. This requires OS awareness of Flicker sessions so that it can quiesce devices appropriately. Modern devices already support suspension in the form of ACPI power events [89], although this is sub-optimal since power will remain available to devices. The best solution is to modify device drivers to be Flicker-aware, so that minimal work is required to prepare for a Flicker session. We plan to further investigate Flicker-aware device drivers and OS extensions, but the best solution may be an architectural change for next-generation hardware.

4.5.7 Major Performance Problems

Our experiments reveal two significant performance bottlenecks for minimal TCB execution on current CPU architectures: (1) on a multi-CPU machine, the inability to execute PALs and untrusted code simultaneously on different CPUs, and (2) the use of TPM Seal and Unseal to protect PAL state during a context switch between secure and insecure execution.

The first issue exacerbates the second, since the TPM-based overheads apply to the entire platform, and not only to the running PAL, or even only to the CPU on which the PAL runs. With TPM-induced delays of over a second, this results in significant overhead. While this overhead may be acceptable for a system dedicated to a particular security-sensitive application, it is not generally acceptable in a multiprogramming environment.

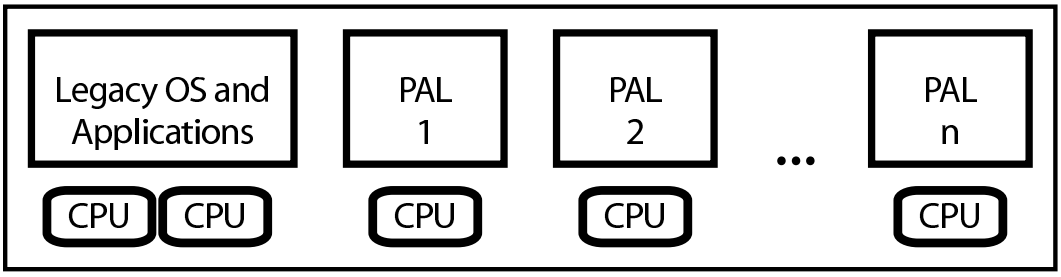


Figure 4.17: **Goal of Our Architectural Recommendations.** *Physical platform running a legacy OS and applications along with some number of* PALs.

Below, we suggest modifications to the hardware that will overcome these limitations.

4.6 Architectural Recommendations

In this section, we make hardware recommendations to alleviate the performance issues we summarize in Section 4.5.7, while maintaining the security properties of Flicker. Specifically, the goal of these recommendations is twofold: (1) to enable the concurrent execution of an arbitrary number of mutually-untrusting PALs alongside an untrusted legacy OS and legacy applications, and (2) to enable performant context switching of individual PALs. A system achieving these goals supports multiprogramming with PALs, so that there can be more PALs executing than there are physical CPUs in a system. It also enables efficient use of the execution resources available on today's multicore computing platforms. Figure 4.17 shows an example of our desired execution model. Note that we assume that a PAL only executes on one CPU core at a time, but Section 4.6.8 discusses extension to multiple cores.

We have two requirements for the recommendations we make. First, our recommendations must make minimal modifications to the architecture of today's trusted computing technologies: AMD SVM and Intel TXT. Admittedly, such a requirement narrows the scope of our creativity. However, we believe that by keeping our modifications minimal, our recommendations are more likely to be implemented by hardware vendors. Second, in order to keep our execution architecture as close to today's systems architectures as possible, we require that the untrusted OS retain the role of the resource manager. With this requirement, we open up the possibility that the untrusted OS could perform denial-of-service attacks against the PALs. However, we believe this risk is unavoidable, as the untrusted OS can always simply power down or otherwise crash the system.

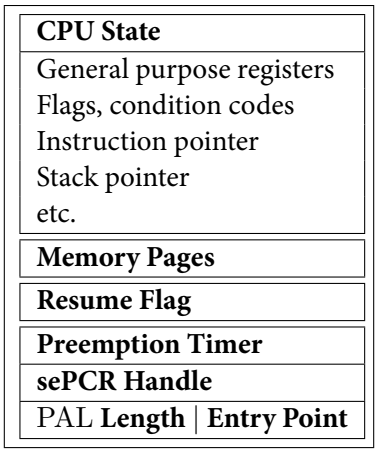


Figure 4.18: **SECB Structure.**

There are two new hardware mechanisms required to achieve our desired execution model (Figure 4.17) while simultaneously satisfying the two requirements mentioned in the previous paragraph. The first is a hardware mechanism for memory isolation that isolates the memory pages belonging to a PAL from all other code. The second is a hardware context switch mechanism that can efficiently suspend and resume PALs, without exposing a PAL's execution state to other PALs or the untrusted OS. In addition to these two mechanisms, we also require modifications to the TPM to allow external verification via attestation when multiple PALs execute concurrently.

In the rest of this section, we first describe PAL launch (Section 4.6.1), followed by our proposed hardware memory isolation mechanism (Section 4.6.2). Section 4.6.3 talks about the hardware context switch mechanism we propose. In Section 4.6.4 we describe changes to the TPM chip to enable external verification. We describe PAL termination in Section 4.6.5. Section 4.6.6 ties these recommendations together and presents the life-cycle of a PAL. Finally, Section 4.6.7 summarizes the expected performance improvement of our recommendations, and Section 4.6.8 suggests some natural extensions.

4.6.1 Launching a PAL

We propose a new mechanism for securely launching a PAL.

4.6.1.1 Recommendation

First, we recommend that the untrusted OS allocate resources for a PAL. Resources include execution time on a CPU and a region of memory to store the PAL's code and data. We define a *Secure Execution Control Block* (SECB, Figure 4.18) as a structure to hold PAL state and resource allocations, both for the purposes of launching a PAL and for storing the state of a PAL when it is not executing. The PAL and SECB should be contiguous in memory to facilitate memory isolation mechanisms. The SECB entry for allocated memory should consist of a list of physical memory pages allocated to the PAL.

To begin execution of a PAL described by a newly allocated SECB, we propose the addition of a new CPU instruction, *Secure Launch* (*SLAUNCH*), that takes as its argument the starting physical address of a SECB. Upon execution, *SLAUNCH*:

1. reinitializes the CPU on which it executes to a well-known trusted state,
2. enables hardware memory isolation (described in Section 4.6.2) for the memory region defined in the SECB and for the SECB itself,
3. transmits the PAL to the TPM to be measured (described in Section 4.6.4),
4. disables interrupts on the CPU executing *SLAUNCH*,
5. initializes the stack pointer to the top of the memory region defined in the SECB (allowing the PAL to confirm the size of its data memory region),
6. sets the *Measured Flag* in the SECB to indicate that this PAL has been measured, and
7. jumps to the PAL's entry point as defined in the SECB.

4.6.1.2 Suggested Implementation Based On Existing Hardware

We can modify the existing hardware virtual machine management data structures of AMD and Intel to realize the SECB. Both AMD and Intel use an in-memory data structure to maintain guest state.[11] The functionality of *SLAUNCH* when used to begin execution of a PAL is designed to give the same security properties as today's *SKINIT* and *SENTER* instructions.

4.6.2 Hardware Memory Isolation

To securely execute a PAL using a minimal TCB, we need a hardware mechanism to isolate its memory state from all devices and from all code executing on other CPUs (including other PALs and the untrusted OS and applications).

[11] These structures are the Virtual Machine Control Block (VMCB) and Virtual Machine Control Structure (VMCS) for AMD and Intel, respectively.

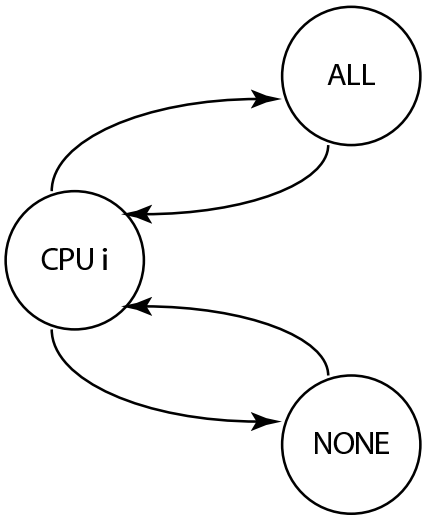


Figure 4.19: **Memory Page States.** *State machine for the possible states of a memory page in our proposed memory controller modification. The states correspond to which CPUs can access an individual memory page.*

4.6.2.1 Recommendation

We propose that the memory controller maintain an access control table with one entry per physical page, where each entry specifies which CPUs (if any) have access to the physical page. The size of this table will be $M \times N$, where M is the number of physical pages present on the platform and N is the maximum number of CPUs. Other multiprocessor designs use a similar partitioning system to protect memory from other processors [117]. To use the access control table, the memory controller must be able to determine which CPU initiates a given memory request.

Figure 4.19 presents the state machine detailing the possible states of an entry in the access control table as context switches (described in Section 4.6.3) occur. Memory pages are by default marked `ALL` to indicate that they are accessible by all CPUs and DMA-capable devices. The other states are described below. Note that the SECB for a PAL must reside within the PAL's memory region so as to benefit from its own memory protections.

When PAL execution is started using *SLAUNCH*, the memory controller updates its access control table so that each page allocated to the PAL (as specified by the list of memory pages in the SECB) is accessible only to the CPU executing the PAL. When the PAL is subsequently suspended, the state of its memory pages transitions to `NONE`, indicating that nothing currently executing on the platform is allowed to read or write to those pages. Note

that the memory allocated to a PAL includes space for data, and is a superset of the pages containing the PAL binary.

4.6.2.2 Suggested Implementation Based On Existing Hardware

We can realize hardware memory isolation as an extension to existing DMA protection mechanisms. As noted in Section 2.4.2, AMD SVM and Intel TXT already support DMA protections for physical memory pages.[12] In both protection systems, the memory controller maintains a bit vector with one bit per physical page. The value of the bit indicates whether the corresponding page can be accessed (read or written) using a DMA operation. One implementation strategy for our recommendations may be to increase the size of each entry in this protection table to include a bit per CPU on the system.

Existing memory access and cache coherence mechanisms can be used to provide the necessary information to enforce memory isolation. Identifying the CPU from which memory requests originate is straightforward, since memory reads and writes on different CPUs already operate correctly today. For example, every memory request from a CPU in an Intel system includes an *agent ID* that uniquely identifies the requesting CPU to the memory controller [178].

The untrusted OS will be unable to access the physical memory pages that it allocates to the PALs, and so supporting the execution of PALs requires the OS to cope with discontiguous physical memory. Modern OSes support discontiguous physical memory for structures like the AGP graphics aperture, which require the OS to relinquish certain memory pages to hardware. These mechanisms can be modified to tolerate the allocation of memory to PALs.

4.6.3 Hardware Context Switch

To enable multiplexing of CPUs between multiple PALs and the untrusted OS, a secure context switch mechanism is required. Our mechanism retains the legacy OS as the primary resource manager on a system, allowing it to specify on which CPU and for how long a PAL can execute.

4.6.3.1 Recommendation

We first treat the mechanism required to cause an executing PAL to yield, and then detail how a suspended PAL is resumed.

[12] The protection mechanisms are the Device Exclusion Vector (DEV) and the Memory Protection Table (MPT) for AMD and Intel, respectively.

PAL Yield. We recommend the inclusion of a PAL preemption timer in the CPU that can be configured by the untrusted OS. When the timer expires, or a PAL voluntarily yields, the PAL's CPU state should be automatically and securely written to its SECB by hardware, and control should be transferred to an appropriate handler in the untrusted OS. To enable a PAL to voluntarily yield, we propose the addition of a new CPU instruction, *Secure Yield* (*SYIELD*). Part of writing the PAL's state to its SECB includes signaling the memory controller that the PAL and its state should be inaccessible to all entities on the system. Note that any microarchitectural state that may persist long enough to leak the secrets of a PAL must be cleared upon PAL yield.

PAL Resume. The untrusted OS can resume a PAL by executing an *SLAUNCH* on the desired CPU, parameterized with the physical address of the PAL's SECB. The PAL's *Measured Flag* indicates to the CPU that the PAL has already been measured and is only being resumed, not started for the first time. Note that the *Measured Flag* is honored only if the SECB's memory page is set to `NONE`. This prevents the untrusted OS from invoking a PAL without it being measured by the TPM. During PAL resume, the *SLAUNCH* instruction will signal the memory controller that the PAL's state should be accessible to the CPU on which the PAL is now executing. Note that the PAL may execute on a different CPU each time it is resumed. Once a PAL is executing on a CPU, any other CPU that tries to resume the same PAL will fail, as that PAL's memory is inaccessible to the other CPUs.

4.6.3.2 Suggested Implementation Based On Existing Hardware

We achieve significant performance improvements by eliminating the use of TPM sealed storage as a protection mechanism for PAL state during context switches. Existing hardware virtualization extensions of AMD and Intel support suspending and resuming guest VMs.[13] We can enhance these mechanisms to provide secure context switch by extending the memory controller to isolate a PAL's state while it is executing, even from an OS. Table 4.20 shows that with current hardware, VM entry and exit overheads are on the order of half a microsecond. Reducing the context switch overhead of between approximately 200 ms and a full second for the TPM sealed storage-based context switch mechanism (recall Figure 4.11) to essentially the overhead of a VM exit or entry would be a pronounced improvement.

[13] A guest yields by executing VMMCALL / VMCALL. A VMM resumes a guest by executing VMRUN / VMRESUME for AMD and Intel, respectively.

Operation	AMD SVM		Intel TXT	
	Avg (μs)	Stdev	Avg (μs)	Stdev
VM Enter	0.5580	0.0028	0.4457	0.0029
VM Exit	0.5193	0.0036	0.4491	0.0015

Figure 4.20: **VM Entry and Exit Performance.** *Benchmarks showing the average runtime of VM Entry and VM Exit on the Tyan n3600R with a 1.8 GHz AMD Opteron and the MPC ClientPro 385 with a 2.66 GHz Intel Core 2 Duo.*

4.6.4 Improved TPM Support for Flicker

Thus far, our focus has been on recommendations to alleviate the two performance bottlenecks identified in Section 4.5.7. Unfortunately, the functionality of today's TPMs is insufficient to provide measurements, sealed storage, and attestations for multiple, concurrently executing PALs. These features are essential to provide external verification.

As implemented with today's hardware, Flicker always uses PCR 17 (and 18 on Intel systems) to store a PAL's measurement. The addition of the *SLAUNCH* instruction introduces the possibility of concurrent PAL execution. When executing multiple PALs concurrently, today's TPMs do not have enough PCR registers to securely store the PALs' measurements. Further, since PALs may be context switched in and out, there can be many more PALs executing than there exist CPUs on the system.

Ideally, the TPM should maintain a separate measurement chain for each executing PAL, and the measurement chain should indicate that the PAL began execution via the *SLAUNCH* instruction. These are the same properties that late launch provides for one PAL today.

We propose the inclusion of additional *secure execution* PCRs (sePCRs) that can be bound to a PAL during *SLAUNCH*. The number of sePCRs present in a TPM establishes the limit for the number of concurrently executing PALs, as measurements of additional PALs do not have a secure place to reside. The PAL must also learn the identity of its sePCR so that it can output a sePCR handle usable by untrusted software to generate a TPM Quote once execution is complete.

However, the addition of sePCRs introduces several challenges:

1. A PAL must be bound to a unique sePCR (Section 4.6.4.1).
2. A PAL's sePCR must be inaccessible to all other code until the PAL terminates (Section 4.6.4.2).
3. TPM Quote must be able to address the sePCRs when invoked from untrusted code (Section 4.6.4.3).

4. A PAL that used TPM Seal to seal secrets to one sePCR must be able to unseal its secrets in the future, even if that PAL terminates and is assigned a different sePCR on its next invocation (Section 4.6.4.4).
5. A hardware mechanism is required to arbitrate TPM access from multiple CPUs (Section 4.6.4.5).

Below, we present additional details for each of these challenges and propose solutions.

4.6.4.1 sePCR Assignment and Communication

Challenge 1 specifies that a PAL must be bound to a unique sePCR while it executes. The binding of the sePCR to the PAL must prevent other code (PALs or the untrusted OS) from extending or reading the sePCR until the PAL has terminated. We describe how the TPM and CPU communicate to assign a sePCR to a PAL during *SLAUNCH*.

As part of *SLAUNCH*, the contents of the PAL are sent from the CPU to the TPM to be measured. The arrival of these messages signals the TPM that a new PAL is starting, and the TPM assigns a free sePCR to the PAL being launched. The sePCR is reset to zero and extended with a measurement of the PAL. If no sePCR is available, *SLAUNCH* must return a failure code.

As part of *SLAUNCH*, the TPM returns the allocated sePCR's handle to the CPU executing the PAL. This handle becomes part of the PAL's state, residing in the CPU while the PAL is executing and written to the PAL's SECB when the PAL is suspended.[14] The handle is also made available to the executing PAL. One implementation strategy is to make the handle available in one of the CPU's general purpose registers when the PAL first gets control.

TPM Extend, Seal, and Unseal must be extended to optionally accept a PAL's sePCR as an argument, but only when invoked from within that PAL. The CPU, memory controller, and TPM must prevent other code from invoking TPM Extend, Seal, or Unseal with a PAL's sePCR. Enforcement can be performed by the CPU or memory controller using the CPU's copy of the PAL's sePCR handle. These restrictions do not apply to TPM Quote, as untrusted code will eventually need the PAL's sePCR handle to generate a TPM Quote. We describe its use in more detail in Section 4.6.4.3.

Note that the TPM in today's machines is a memory-mapped device, and access to the TPM involves the memory controller. The exact architectural details are chipset-specific,

[14] This is similar to the handling of Machine Status Registers (MSRs) by AMD SVM and Intel TXT for virtualized CPU state today.

but it may be necessary to enable the memory controller to cache the sePCR handles during *SLAUNCH* to enable enforcement of the PAL-to-sePCR binding and avoid excessive communication between the CPU and memory controller during TPM operations.

4.6.4.2 sePCR Access Control

Challenge 2 is to render a PAL's sePCR inaccessible to all other code. This includes concurrently executing PALs and the untrusted OS. This condition must hold whether the PAL is actively running on a CPU or context switched out.

The binding between a PAL and its sePCR is maintained in hardware by the CPU and TPM. Thus, a PAL's sePCR handle need not be secret, as other code attempting any TPM commands with the PAL's sePCR handle will fail. PAL code is able to access its own sePCR to invoke TPM Extend to measure its inputs, or TPM Seal or Unseal to protect secrets, as described in the previous section.

A PAL needs exclusive access to its sePCR for the TPM Extend, Seal, and Unseal operations. Allowing, e.g., a TPM PCR Read by other code does not introduce a security vulnerability for a PAL. However, we cannot think of a scenario where it is beneficial, and allowing sePCR access from other code for selected commands may unnecessarily complicate the access control mechanism.

4.6.4.3 sePCR States and Attestation

The previous section describes techniques that give a PAL exclusive access to its sePCR. However, Challenge 3 states our aim to allow TPM Quote to be invoked from untrusted code. To enable these semantics, sePCRs exist in one of three states: `Exclusive`, `Quote`, and `Free`. While a PAL is executing or context-switched out, its sePCR is in the `Exclusive` state. No other code on the system can read, extend, reset, or otherwise modify the contents of the sePCR.

When the PAL terminates, untrusted code is tasked with generating an attestation of the PAL's execution. The purpose of the `Quote` state is to grant the necessary access to the untrusted code. Thus, as part of PAL termination, the CPU must signal the TPM to transition this PAL's sePCR from the `Exclusive` to the `Quote` state.

To generate the quote, the untrusted code must be able to specify the handle of the sePCR to use. It is the responsibility of the PAL to include its sePCR handle as an output. The TPM Quote command must be extended to optionally accept a sePCR handle instead of (or in addition to) a list of regular PCR registers to include in the quote.

After a TPM Quote is generated, the TPM transitions the sePCR to the `Free` state, where it is eligible for use by another PAL via *SLAUNCH*. This can be realized as a new TPM command, `TPM_SEPCR_Free`, executable from untrusted code. We treat the case where a PAL does not terminate cleanly in Section 4.6.5.

4.6.4.4 Sealing Data Under a sePCR

TPM Seal can be used to encrypt data such that it can only be decrypted (using TPM Unseal) if the platform is in a particular software configuration, as defined by the TPM's PCRs. TPM Seal and Unseal must be enhanced to work with our proposed sePCRs.

A PAL is assigned a free sePCR by the TPM when *SLAUNCH* is executed on a CPU. However, the PAL does not have control over *which* sePCR it is assigned. This breaks the traditional semantics of TPM Seal and Unseal, where the index of the PCR(s) that must contain particular values for TPM Unseal are known at seal-time. To meet Challenge 4, we must ensure that a PAL that uses TPM Seal to seal secrets to its assigned sePCR will be able to unseal its secrets in the future, even if that PAL terminates and is assigned a different sePCR when it executes next.

We propose that TPM Seal and Unseal accept a boolean flag that indicates whether to use a sePCR. The sePCR to use is specified implicitly by the sePCR handle stored inside of the PAL's SECB.

4.6.4.5 TPM Arbitration

Today's TPM-to-CPU communication architecture assumes the use of software locking to prevent multiple CPUs from trying to access the TPM concurrently. With the introduction of *SLAUNCH*, we require a hardware mechanism to arbitrate TPM access from PALs executing on multiple CPUs. A simple arbitration mechanism is hardware locking, where a CPU requests a lock for the TPM and obtains the lock if it is available. All other CPUs learn that the TPM lock is set and wait until the TPM is free to attempt communication.

4.6.5 PAL Exit

When a PAL finishes executing, its resources must be returned to the untrusted OS so that they can be allocated to another PAL or legacy application that is ready to execute. We first describe this process for a well-behaved PAL, and then discuss what must happen for a PAL that crashes or otherwise exits abnormally.

Normal Exit. The memory pages for a PAL that are inaccessible to the remainder of the system must be freed when that PAL completes execution. It is the PAL's responsibility to erase any secrets that it created or accessed before freeing its memory. To free this memory, we propose the addition of a new CPU instruction, *Secure Free* (*SFREE*). *SFREE* is parameterized with the address of the PAL's SECB, and communicates to the memory controller that these pages no longer require protection. The memory controller then updates its access control table to mark these pages as `ALL` so that the untrusted OS can allocate them elsewhere. Note that *SFREE* executed by other code must fail. This can be detected by verifying that the *SFREE* instruction resides at a physical memory address inside the PAL's memory region. As part of *SFREE*, the CPU also sends a message to the TPM to cause the terminating PAL's sePCR to transition from the `Exclusive` state to the `Quote` state.

Abnormal Exit. The code in a PAL may contain bugs or exploitable flaws that cause it to deviate from the intended termination sequence. For example, it may become stuck in an infinite loop. The preemption timer discussed in Section 4.6.3 can preempt the misbehaving PAL, but the memory allocated to that PAL remains in the `NONE` state, and the sePCR allocated to that PAL remains in the `Exclusive` state. These resources must be freed without exposing any of the PAL's secrets to other entities on the system.

We propose the addition of a new CPU instruction, *Secure Kill* (*SKILL*), to kill a misbehaving PAL. Its operations are as follows:

1. Erase all memory pages associated with the PAL.
2. Mark the PAL's memory pages as available to `ALL`.
3. Extend the PAL's sePCR with a well known constant that indicates that *SKILL* was executed.
4. Transition the PAL's sePCR to the `Free` state.

Depending on low-level implementation details, *SKILL* may be merged with *SFREE*. One possibility is that *SFREE* behaves identically to *SKILL* whenever it executes outside of a PAL.

4.6.6 PAL Life Cycle

Figure 4.21 summarizes the life cycle of a PAL on a system with our recommendations. To provide a better intuition for the ordering of events, we step through each state in detail. We also provide pseudocode for *SLAUNCH*, and indicate which states of a PAL's life cycle correspond to portions of the *SLAUNCH* pseudocode (Figure 4.22).

Launch: Protect and Measure. The untrusted OS is responsible for creating the necessary SECB structure for a PAL so that the PAL can be executed. The OS allocates memory pages

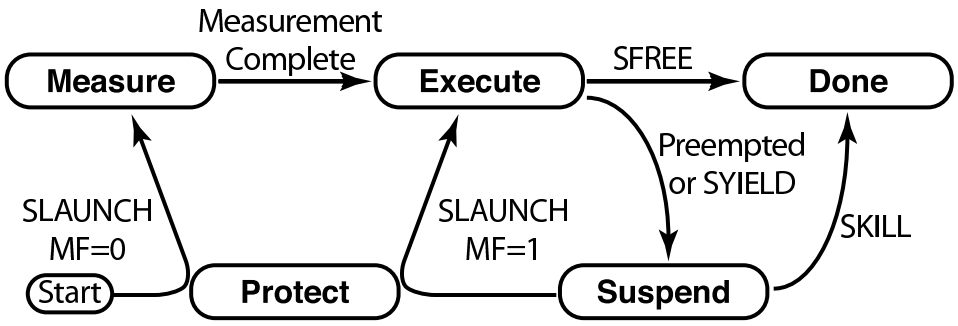


Figure 4.21: **Life Cycle of a PAL.** *MF stands for* Measured Flag. *Note that these states are for illustrative purposes and need not be represented in the system.*

Start:
OS: Allocate pages for SECB S and PAL P
OS: Initialize SECB.pages
OS: Initialize SECB.timer

Protect:
CPU$_i$: *SLAUNCH* (S)
CPU$_i$: Reinitialize to trusted state
CPU$_i$: Disable interrupts
CPU$_i$ to MC: SECB.pages
MC: if($\exists p \in$ SECB.pages s.t. $p.accessible =$ `NONE`) FAIL
MC: $\forall p \in$ SECB.pages: $p.accessible =$ CPU$_i$
MC to CPU$_i$: done
CPU$_i$: ESP=SECB.pages.top

Measure:
if($\neg$SECB.MeasuredFlag)
  CPU$_i$: send PAL to TPM
  TPM: Allocate sePCR ℓ
  MC: if($\neg\exists \ell \in$ sePCRs s.t. sePCR [ℓ].state $=$ `Quote`) FAIL
  TPM: $h =$ SHA-1(PAL)
  TPM: sePCR [ℓ] $= 0$
  TPM: sePCR [ℓ] $=$ SHA-1(sePCR [ℓ]$||h$)
  TPM to CPU$_i$: done
  CPU$_i$: SECB.MeasuredFlag $= 1$

Execute:
CPU$_i$: EIP=SECB.pages.eip
CPU$_i$: Begin executing

Figure 4.22: ***SLAUNCH* Pseudocode.**

for the PAL and sets the PAL's preemption timer. The OS then invokes the *SLAUNCH* CPU instruction with the address of the SECB, initiating the transition from the `Start` state to the `Protect` state in Figure 4.21. This causes the CPU to signal the memory controller with the address of the SECB. The memory controller updates its access control table (recall Section 4.6.2) to mark the memory pages associated with the SECB as being accessible only by the CPU which executed the *SLAUNCH* instruction. If the memory controller discovers that another PAL is already using any of these memory pages, it signals the CPU that *SLAUNCH* must return a failure code. Once the memory protections are in place, the memory controller signals the CPU. The CPU inspects the *Measured Flag* and begins the measurement process since it is clear. The *Measured Flag* in the SECB (Figure 4.18) is used to distinguish between a PAL that is being executed for the first time and a PAL that is being resumed. This completes the transition from the `Protect` state to the `Measure` state.

The CPU then begins sending the contents of the PAL to the TPM to be hashed. When the first message arrives at the TPM, the TPM attempts to allocate a sePCR for this PAL. A free sePCR is allocated, reset, and then extended with a measurement of the contents of the PAL. The TPM returns a handle to the allocated sePCR to the CPU, where it is maintained as part of the SECB. If there is no sePCR available, the TPM returns a failure code to the CPU. The CPU signals the memory controller to return the SECB's pages to the `ALL` state, and *SLAUNCH* returns a failure code. Upon reception of the sePCR handle, the CPU sets the *Measured Flag* for the PAL to indicate that it has been measured. The completion of measurement causes a transition from the `Measure` state to the `Execute` state.

Execute. The PAL is now executing with full hardware protections. It is free to complete whatever application-specific task it was designed to do. If it requires data from an external source (e.g., network or disk), it may yield by executing *SYIELD*. If it has been running for too long, the CPU may preempt it. These events affect transitions to the `Suspend` state. If the PAL is ready to exit, it can transition directly to the `Done` state by executing *SFREE*.

Suspend: Preempted or *SYIELD*. The PAL is no longer executing, and it must transition securely to the `Suspend` state. The CPU signals the memory controller that this PAL is suspending, and the memory controller updates its access control table for that PAL's memory pages to `NONE`, indicating that those pages should be unavailable to all processors and devices until the PAL resumes. Once the protections are in place, the memory controller signals the CPU, and the CPU completes the secure state clear (e.g., it may be necessary to clear microarchitectural state such as cache lines). At this point, the PAL is suspended. If the OS has reason to believe that this PAL is malfunctioning, it can terminate the PAL using the *SKILL* instruction. *SKILL* causes a transition directly to the `Done` state.

Resume. The untrusted OS invokes the *SLAUNCH* instruction on the desired CPU to resume a PAL, again with the address of the PAL's SECB. The causes a transition from the `Suspend` state to the `Protect` state. The CPU signals the memory controller with the SECB's address, just as when `Protect` was reached from the initial `Start` state. The memory controller enables access to the PAL's memory pages by removing the `NONE` status on the PAL's memory pages, setting them as accessible only to the CPU executing the PAL. The memory controller signals an error if these pages were in use by another CPU. The memory controller then signals the CPU that protections are in place. The *Measured Flag* is set, indicating that the PAL has already been measured, so the CPU reloads the suspended architectural state of the PAL and directly resumes executing the PAL's instruction stream, causing a transition from the `Protect` to the `Execute` state.

Exit. While executing, the PAL can signal that it has completed execution with *SFREE*. This causes the CPU to send a message to the TPM indicating that the PAL's sePCR should transition to the `Quote` state. It is assumed that the PAL has already completed an application-level state clear. The CPU then performs a secure state clear of architectural and microarchitectural state, and signals to the memory controller that this PAL has exited. The memory controller marks the relevant pages as available to the remainder of the system by transitioning them to the `ALL` state. This CPU is now finished executing PAL code, as indicated by the transition to the `Done` state. It becomes available to the untrusted OS for use elsewhere.

4.6.7 Expected Impact

Here, we summarize the impact we expect our recommendations to have on Flicker application performance. First, the improved memory isolation of PAL state allows truly concurrent execution of secure and legacy code, even on multicore systems. Thus, PAL execution no longer requires the entire system to grind to a halt.

With Flicker on existing hardware, a PAL yields by simply transferring control back to the untrusted OS. Resume is achieved by executing late launch again. It is the responsibility of the PAL to protect its own state before yielding, and to reconstruct the necessary state from its inputs upon resume. Protecting state requires the use of the TPM Seal and Unseal commands. An *SKINIT* on AMD hardware can take up to 177.52 ms (Figure 4.10), while Seal requires 20-500 ms and Unseal requires 290-900 ms (Figure 4.11). Thus, context switching into a PAL (which requires unsealing prior data) can take over 1000 ms, while context switching out (which requires sealing the PAL's state) can require 20-500 ms. Further, existing hardware has no facility for guaranteeing that a PAL can be preempted (to prevent it from compromising system availability).

With our recommendations, we eliminate the use of TPM Seal and Unseal during context switches and only require that the TPM measure the PAL once (instead of on every context switch). We expect that an implementation of our recommendations can achieve PAL context switch times on the order of those possible today using hardware virtualization support, i.e., 0.6 μs on current hardware (Figure 4.20). This reduces the overhead of context switches by six orders of magnitude (from 200-1000 ms on current hardware) and hence makes it significantly more practical to switch in and out of a PAL.

Taken together, these improvements help make on-demand minimal TCB code execution with Flicker a practical and effective way to achieve secure computation on commodity systems, while only requiring relatively minor changes in existing technology.

As an alternative to our recommended hardware modifications, we could instead consider increasing the speed of the TPM and the bus through which it communicates with the CPU. As shown in Section 4.5, the TPM is a major bottleneck for efficient Flicker applications on current hardware. Increasing the TPM's speed could potentially reduce the cost of using the TPM to protect PAL state during a context switch, and similarly reduce the penalty of using *SKINIT* during every context switch. However, achieving sub-microsecond overhead comparable to our recommendations would require significant hardware engineering of the TPM, since many of its operations use a 2048-bit RSA keypair. Even with hardware support to make the operations performant, the power consumed by such operations is wasteful, since we can achieve superior performance with less power-intensive modifications.

4.6.8 Extensions

We discuss issues that our recommendations do not address, but that may be desirable in future systems.

Multicore PALs. As presented, we offer no mechanism for allocating more than one CPU to a single PAL. First, it should be noted that a single application-level function that will benefit from multicore PALs can be implemented as multiple single-CPU PALs. However, applications that require frequent communication between code running on different CPUs (e.g., for locks) may suffer from PAL launch, termination and context switching overheads. To address this, a mechanism is needed to join a CPU to an existing PAL. The join operation adds the new CPU to the memory controller's access control table for the PAL's pages.

sePCR Sets. As presented, we propose a one-to-one relationship between sePCRs and PALs. It is a straightforward extension to group sePCRs into sets and bind a set of sePCRs to each PAL. The TPM operations that accept an sePCR as an argument will need to be modified appropriately. Some will be indexed by the sePCR set itself (e.g., *SLAUNCH* will need to

cause all sePCRs in a set to reset), some by a subset of the sePCRs in a set (e.g., TPM Quote), and others by the individual sePCRs inside a set (e.g., TPM Extend).

PAL Interrupt Handling. As presented, interrupts are disabled on the CPU executing a PAL (expiry of the preemption timer does not cause a software-observable interrupt to the PAL). We believe that a PAL's purpose should be to perform an application-specific security-sensitive operation. As such, we recommend that a PAL not accept interrupts. However, there may still be situations where it is necessary to receive an interrupt, e.g., in future systems where a PAL requires human input from the keyboard. Thus, a PAL should be able to configure an Interrupt Descriptor Table to receive interrupts. However, this may result in the PAL receiving extraneous interrupts. Routing only the interrupts the PAL is interested in requires the CPU to reprogram the interrupt routing logic every time a PAL is scheduled, which may create undesirable overhead or design complexity.

4.7 Summary

Flicker allows code to verifiably execute with hardware-enforced isolation, while adding as few as 250 lines of code to the application's TCB. Given the correlation between code size and bugs in the code, Flicker significantly improves the security of the code it executes. New desktop machines already contain the hardware support necessary for Flicker, so widespread Flicker-based applications can soon become a reality. We have also recommended changes to the CPU, memory controller, and TPM that alleviate today's dependence on computationally expensive TPM operations to protect application state during context switches, and that allow concurrent execution of secure and insecure code. As a result, our research brings a Flicker of hope for securing commodity computers.

Chapter 5

Using Trustworthy Host-Based Information in the Network

Why is it difficult to improve network security? One culprit is the fact that network elements cannot trust information provided by the endhosts. Indeed, network elements often waste significant resources painstakingly reconstructing information that *endhosts already know*. For example, a network-level Denial-of-Service (DoS) filter must keep track of how many packets each host recently sent, in order to throttle excessive sending. Researchers have developed many sophisticated algorithms to trade accuracy for reduced storage overhead [58, 204, 208], but they all amount to *approximating* information that can be *precisely and cheaply tracked* by the sender! In other words, the filter's task could be greatly simplified if each sender could be trusted to include its current outbound bandwidth usage in each packet it sent.

Of course, the question remains: Is it possible to trust endhosts? Chapters 2-4 imply that we can leverage the widespread deployment of commodity computers equipped with hardware-based security enhancements to allow the network to trust *some* host-based information. As such security features become ubiquitous, it is natural to ask if we can leverage them to improve network security and efficiency.

As an initial exploration of how endhost hardware security features can be used to improve the network, we have designed a general architecture named Assayer. While Assayer may not represent the optimal way to convey this information, we see it as a valuable first step to highlight the various issues involved. For example, can we provide useful host-based information while also protecting user privacy? Which cryptographic primitives are needed to verify this information in a secure and efficient manner? Our initial findings suggest that

improved endhost security can improve the security and efficiency of the network, while simultaneously reducing the complexity of in-network elements.

In the Assayer architecture, senders employ secure hardware to convince an off-path *verifier* that they have installed a small code module that maintains network-relevant information. A small protection layer enforces mutual isolation between the code module and the rest of the sender's software, ensuring both security and privacy. Once authorized by a verifier, the code module can insert cryptographically-secured information into outbound traffic. This information is checked and acted on by in-path *filters*.

To evaluate the usefulness of trustworthy host-based information, we consider the application of Assayer to three case studies: spam identification, Distributed Denial-of-Service (DDoS) mitigation, and super-spreader worm detection. We find that Assayer is well-suited to aid in combating spam and can mitigate many (though certainly not all) network-level DDoS attacks. In these two applications, Assayer can be deployed incrementally, since victims (e.g., email hosts or DDoS victims) can deploy Assayer filters in conjunction with existing defenses. Legitimate senders who install Assayer will then see improved performance (e.g., fewer emails marked as spam, or higher success in reaching a server under DDoS attack). Legacy traffic is not dropped but is processed at a lower priority, encouraging, but not requiring, additional legitimate senders to install Assayer. Surprisingly, we find that while it is technically feasible to use Assayer to combat super-spreader worms, such use would face challenges when it comes to deployment incentives.

To better understand the performance implications of conveying host-based information to the network, we implement a full Assayer prototype, including multiple protocol implementations. Our prototype employs trusted computing hardware (a TPM) that is readily available in commodity hardware [88]. The size of the protection layer on the client that protects code modules from the endhost (and vice versa) is minuscule (it requires 841 lines of code), and the individual modules are even smaller. Our verifier prototype can sustain 3300 client verifications per second and can handle bursts of up to 5700 clients/second. Generating and verifying annotations for outbound traffic requires only a few microseconds for our most efficient scheme, and these annotations can be checked efficiently. Even on a gigabit link, we can check the annotations with a reasonable throughput cost of 3.7-18.3%, depending on packet size.

Contributions. In this chapter, we make the following contributions: **(1)** We explore the design of network mechanisms to leverage endhost-based computation and state to improve application efficiency, accuracy, and security. **(2)** We design an architecture to provide such information securely and efficiently. **(3)** We implement and evaluate the architecture.

5.1 Problem Definition

5.1.1 Architectural Goals

We aim to design an architecture to allow endhosts to share information with the network in a trustworthy and efficient manner. This requires the following critical properties:

Annotation Integrity. Malicious endhosts or network elements should be unable to alter or forge the data contained in message annotations.

Stateless In-Network Processing. To ensure the scalability of network elements that rely on endhost information, we seek to avoid keeping per-host or per-flow state on these devices. If per-flow state becomes feasible, we can use it to cache authentication information carried in packets.

Privacy Preservation. We aim to leak no more user information than is already leaked in present systems. In other words, we do not aim to protect the privacy of a user who visits a website and enters personal information. However, some applications may require small losses of user privacy. For example, annotating outbound emails with the average length of emails the user sent in the last 24 hours leaks a small amount of personal information, but it can significantly decrease the probability that a legitimate sender's email is marked as spam [87]. We can provide additional privacy by only specifying this information at a coarse granularity, e.g., "short", "medium", and "long". Further research will be necessary to determine whether people accept this tradeoff.

Incremental Deployability. While we believe that trustworthy endhost information would be useful in future networks, we strive for a system that can bring immediate benefit to those who deploy it, regardless of others' adoption status.

Efficiency. To be adopted, the architecture must not unduly degrade client-server network performance. Furthermore, to prevent DoS attacks on the architecture's components, they must be capable of acting efficiently.

5.1.2 Assumptions

Since we assume that our trusted software and hardware components behave correctly, we aim to minimize the size and complexity of our trusted components, since software vulnerabilities are correlated with code size [143], and smaller code is more amenable to formal analysis. We assume that clients can perform hardware-based attestations. In this work, we focus on TCG-based attestations, since TPMs are becoming ubiquitous in commodity PCs [88]; however, other types of secure hardware are also viable (see Section 2.5). Finally,

we make the assumption that secure-hardware-based protections can only be violated with local hardware attacks. We assume remote attackers cannot induce users to perform physical attacks on their own hardware.

5.2 The Assayer Architecture

With Assayer, we hope to explore the intriguing possibilities offered by the advent of improved hardware security in endhosts. If endhosts can be trusted, how can we simplify and improve the network? What techniques are needed to extend host-based hardware assurance into the network? Can trust be verified without significantly reducing network performance? We examine these issues and more below.

Initially, we focus on the qualities needed to build a generic architecture for conveying host-based information to the network, and hence our discussion is necessarily quite general. However, we explore application-specific details, including deployment incentives in Section 5.4.

5.2.1 Overview

Suppose a mail server wants to improve the accuracy of its spam identification using host-based information. For example, a recent study indicates that the average and standard deviation of the size of emails sent in the last 24 hours are two of the best indicators of whether any given email is spam [87]. These statistics are easy for an endhost to collect, but hard for any single mail recipient to obtain.

However, the mail server is faced with the question: how can it decide whether host-provided information is trustworthy? Naively, the mail server might ask each client to include a hardware-based attestation (see Section 2.3) of its information in every email. The mail server's spam filter could verify the attestation and then incorporate the host-provided information into its classification algorithm. Any legacy traffic arriving without an attestation could simply be processed by the existing algorithms. Unfortunately, checking attestations is time-consuming and requires interaction with the client. Even if this were feasible for an email filter, it would be unacceptable for other applications, such as DDoS mitigation, which require per-packet checks at line rates.

Thus, the question becomes: how can we make the average case fast and non-interactive? The natural approach is to cryptographically extend the trust established during a single hardware-based attestation over multiple outbound messages. Thus, the cost of the initial verification is amortized over subsequent messages.

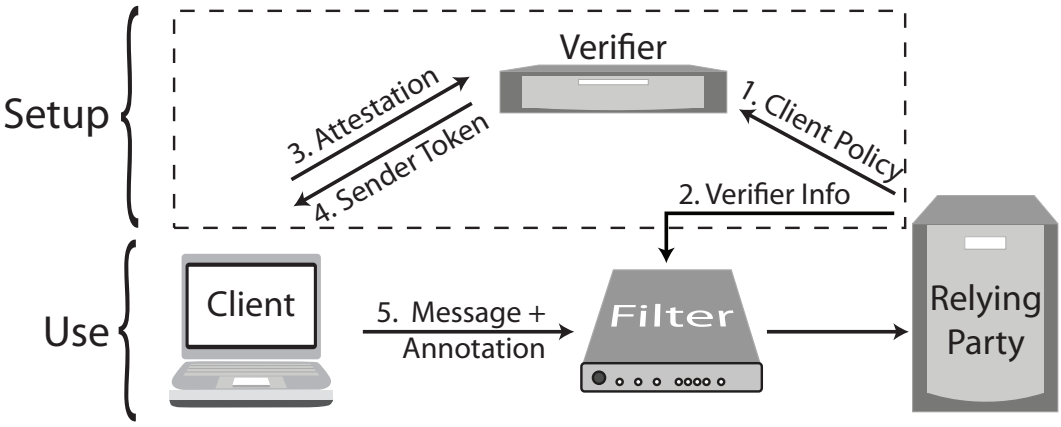


Figure 5.1: **System Components.** *The relying party (e.g., a mail server or an ISP) delegates the task of inspecting clients to one or more verifiers. It also configures one or more filters with information about the verifiers. Every T days, the client convinces a verifier via an attestation that its network measurement modules satisfy the relying party's policy. The verifier issues a Sender Token that remains valid for the next T days. The client can use the Sender Token to annotate its outbound messages (e.g., an annotation for each email, flow, or packet). The filter verifies the client's annotation and acts on the information in the annotation. For example, the filter might drop the message or forward it at a higher priority to the relying party.*

As a result, the Assayer architecture employs two distinct phases: an infrequent setup phase in which the *relying party* (e.g., the mail server) establishes trust in the client, and the more frequent usage phase in which the client generates authenticated annotations on outbound messages (Figure 5.1).

The relying party delegates the task of inspecting clients to one or more off-path *verifier* machines. Every T days, the client convinces a verifier that it has securely installed a trustworthy code *module* that will keep track of network-relevant information (Figure 5.2), such as the number of emails recently sent, or the amount of bandwidth recently used. Section 5.2.2.1 considers how we can secure this information while still allowing the user to employ a commodity OS and preserving user privacy. Having established the trustworthiness of the client, the verifier issues a limited-duration *Sender Token* that is bound to the client's code module.

During the usage phase, the client submits outbound messages to its code module, which uses the Sender Token to authenticate the message annotations it generates. These annotations are then checked by one or more fast-path *filter* middleboxes, which verify the annotations and react accordingly. For instance, a relying party trying to identify spam might feed the authenticated information from the filter into its existing spam classification algo-

rithms. Alternatively, a web service might contract with its ISP to deploy filters on its ingress links to mitigate DDoS attacks by prioritizing legitimate traffic. If the traffic does not contain annotations, then the filter treats it as legacy traffic (e.g., DDoS filters give annotated traffic priority over legacy traffic).

5.2.2 Assayer Components

We present the design decisions for each of Assayer's components, saving the protocol details for Section 5.2.3.

5.2.2.1 Clients

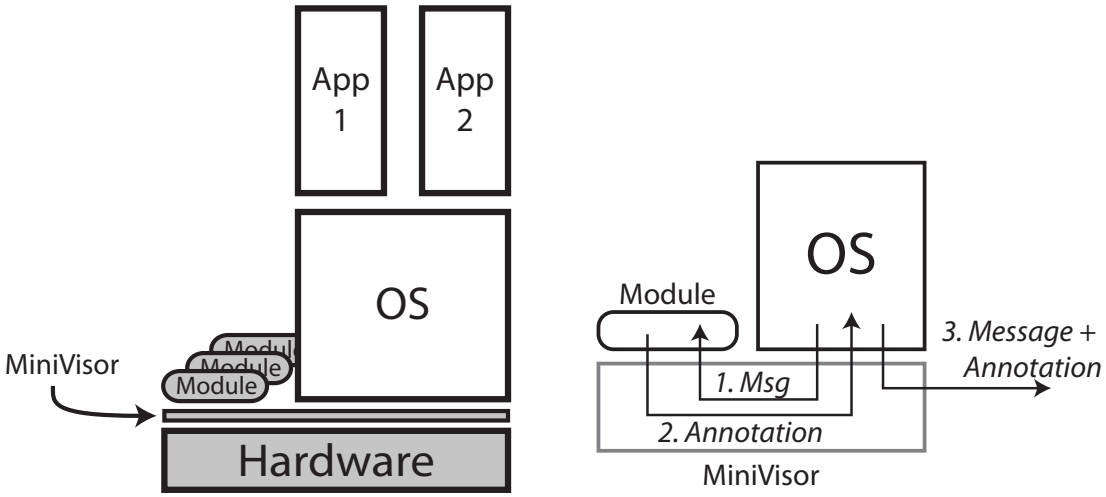
In this section, we consider the generic requirements for allowing clients to transmit trustworthy information to the network. We explore application-specific functionality and client deployment incentives in our case studies (Section 5.4).

Client Architecture. At a high-level, we aim to collect trustworthy data on the client, despite the presence of (potentially compromised) commodity software. To accomplish this, a client who wishes to obtain the benefits of Assayer can install a protective layer that isolates the application-specific *client modules* from the rest of the client's software (Figure 5.2(a)). These client modules could be simple counters (e.g., tracking the number or size of emails sent) or more complex algorithms, such as Bayesian spam filters. The protective layer preserves the secrecy and integrity of the module's state, as well as its execution integrity. It also protects the client's other software from a malicious or malfunctioning module. Untrusted code can submit outbound traffic to a module in order to obtain an authenticated annotation (see Figure 5.2(b)).

How can a client convince the verifier that it has installed an appropriate protective layer and client module? With Assayer, the client can employ hardware-based *attestation* to prove exactly that. When the verifier returns a Sender Token, the protective layer invokes *sealed storage* to bind the Sender Token to the attested software state. This can be combined with the TPM's monotonic counters to prevent state-replay attacks. Thus, any change in the protective layer or client module will make the Sender Token inaccessible and require a new attestation.

Designing the protective layer raises several questions. How much functionality should it provide? Should the client modules be able to learn about the client's other software? Should the protective layer control the entire platform, or only enough to provide basic isolation?

With Assayer, we chose to implement the protective layer as a minimal hypervisor (dubbed MiniVisor) that contains only the functionality needed to protect the client modules from



(a) **Client Configuration** (b) **Annotation Generation**

Figure 5.2: **Client Operations. (a)** *The client attests to the presence of a protective layer (MiniVisor) that isolates modules from untrusted code (and vice versa) and from each other. This allows the client to attest to multiple relying parties without a reboot.* **(b)** *Untrusted code submits messages (e.g., email or packets) to the modules to obtain authenticated annotations.*

the client's other software (and vice versa). This approach sacrifices visibility into the client's software state (e.g., a client module for web browsing cannot determine which web browser the client is using), but protects user privacy from overly inquisitive client modules. Using MiniVisor makes the Trusted Computing Base tiny (potentially fewer than 1,000 lines of code – see Section 5.5) and reduces the performance impact on the client's other software.

In developing MiniVisor, we reject the use of a full-fledged OS [170] or even a VMM [45, 56] as protective layers. Such an approach would leak an excessive amount of information about the client's platform, and assuring the security of these larger code bases would be difficult.

Initially, it is tempting to give MiniVisor full control over the client's network card, in order to ensure that all traffic can be examined by the client modules. However, this approach would significantly increase MiniVisor's complexity, and it would be difficult to ensure full control over *all* outbound traffic on *all* interfaces. Instead, we advocate the use of application-specific incentives to convince the commodity software to submit outbound traffic to the client modules. Since the resulting annotations are cryptographically protected for network transmission, these protections will also suffice while the annotations are handled by the untrusted software. In Section 5.4, we explore whether sufficient client incentives exist for a variety of applications.

Client Modules. As mentioned above (and explored in more detail in Section 5.4), we expect Assayer to support a wide variety of client modules. While we have initially focused on relatively simple modules, e.g., modules to track the size and frequency of emails sent, they could potentially expand to perform more complex operations (e.g., Bayesian spam filtering) of interest to relying parties.

For *global vantage* applications, such as recording statistics on email volume, we expect a single client module would be standardized and accepted by multiple relying parties. MiniVisor's configuration would ensure that the client only runs a single copy of the module to prevent state-splitting attacks. The module could be obtained and installed along with the client's application software (e.g., email client). The application (or a plugin to the application) would then know to submit outbound emails to the module via MiniVisor.

For *single relying party* applications, such as DDoS prevention, a relying party may only care about tracking statistics specific to itself. In this case, the relying party may itself provide an appropriate client module, for example, when the client first contacts the verifier. This model highlights the importance of preserving user privacy, as we emphasized above. The client's OS can submit packets to this module if and only if they are destined to the particular relying party that supplied the client module.

5.2.2.2 Verifiers

Verifiers are responsible for checking that clients have installed a suitable version of MiniVisor and client module and for issuing Sender Tokens. The exact deployment of verifiers is application and relying-party specific. We envision three primary deployment strategies. First, a relying party could deploy its own verifiers within its domain. Second, a trusted third party, such as VeriSign or Akamai could offer a verification service to many relying parties. Finally, a coalition of associated relying parties, such as a group of ISPs, might create a federation of verifiers, such that each relying party deploys a verifier and trusts the verifiers deployed by the other relying parties.

In the last two of these scenarios, the verifiers operate outside of the relying party's direct administrative domain. Even in the first scenario, the relying party may worry about the security of its verifier. To assuage these concerns, the relying party periodically requests a hardware-based attestation from the verifier. Assuming the attestation is correct, the relying party establishes the key material necessary to create an authenticated channel between the verifiers and the filters. On the verifier, this key material is bound (using sealed storage) to the correct verifier software configuration. The limited duration of the verifier's key material is a performance optimization that bounds the length of revocation lists that must be maintained to track misbehaving verifiers.

The use of multiple verifiers, as well as the fact that clients only renew their client tokens infrequently (every T days), makes the verifiers unlikely targets for Denial-of-Service (DoS) attacks, since a DoS attacker would need to flood many verifiers over an extended time (e.g., a week or a month) to prevent clients from obtaining tokens.

Any distributed and well-provisioned set of servers could enable clients to locate the verifiers for a given relying party. While a content distribution network is a viable choice, we propose a simpler, DNS-based approach to ease adoption. Initially, each domain can configure a well-known subdomain to point to the appropriate verifiers. For example, the DNS records for `company.com` would include a pointer to a verifier domain name, e.g., `verifier.company.com`. That domain name would then resolve to a distributed set of IP addresses representing the server's verifier machines. While the DNS servers may themselves become victims of DoS attacks, the static listing of verifier machines is relatively easy to replicate, cache, and serve. Furthermore, if Assayer becomes ubiquitous, the global top-level domain (gTLD) servers could be extended to store a verifier record (in addition to the standard name server record) for each domain. The gTLD servers are already well-provisioned, since a successful attack on them would make many services unavailable.

5.2.2.3 Filters

Filters are middleboxes deployed on behalf of the relying party to act on the annotations provided by the client. For instance, a spam filter might give a lower spam score to an email from a sender who has generated very little email recently. These filters must be able to verify client annotations efficiently to prevent the filters themselves from becoming bottlenecks. In addition, to prevent an attacker from reusing old annotations, each filter must only accept a given annotation once. Section 5.6 shows that filters can perform all of these duties at reasonable speeds.

Filter deployment will be dictated by the application (discussed in more detail in Section 5.4), as well as by the relying party's needs and business relationships. For example, a mail server might simply deploy a single filter as part of an existing spam classification tool chain, whereas a web hosting company may contract with its ISP to deploy DDoS filters at the ISP's ingress links.

To enable filters to perform duplicate detection, the client modules include a unique nonce as part of the authenticated information in each annotation. Filters can insert these unique values into a rotating Bloom Filter [30] to avoid duplication. Section 5.3 discusses the effectiveness of this approach against replay attacks.

5.2.2.4 Relying Party

The relying party (e.g., the operator of the mail or web server) must arrange for an appropriate deployment of verifiers and filters. It may also periodically verify the correctness of its verifiers and issue fresh key material to them. If it detects that a verifier is misbehaving, the relying party can refuse to renew its key material. In Section 5.2.3, we also describe additional provisions to allow the relying party to actively revoke rogue verifiers.

5.2.3 Protocol Details

Below, we enumerate desirable properties for the authorization scheme used to delegate verifying power to verifiers, as well as that used by clients to annotate their outbound traffic. We then describe a scheme based on asymmetric cryptographic operations that achieves all of these properties. Since asymmetric primitives often prove inefficient, we show how to modify the protocols to use efficient symmetric cryptography, though at the cost of two properties. Hybrid approaches of these two schemes are possible, but we focus on these two to explore the extremes of the design space. In Section 5.6, we quantify their performance trade-offs.

5.2.3.1 Desirable Properties

1. **Limited Token Validity.** Verifier key material is only valid for a limited time period and is accessible only to valid verifier software. Sender Tokens should have similar restrictions.
2. **Verifier Accountability.** Verifiers should be held accountable for the clients they approve. Thus one verifier should not be able to generate Sender Tokens that appear to originate from another verifier.
3. **Scalability in Filter Count.** The verifier's work, as well as the size of the Sender Token, should be independent of the number of filters.
4. **Topology Independence.** Neither the verifier nor the sender should need to know which filter(s) will see the client's traffic. In many applications, more than one filter may handle the client's traffic, and the number may change over time. Thus, the sender's token must be valid at any filter operated by the same relying party. We feel the benefits of this approach outweigh the potential for an adversary to use a single Sender Token on multiple disparate paths to the server. Such duplicated packets will be detected when the paths converge.

V	Knows K_{RP}
V	Launches software $Code_V$. $Code_V$ recorded in PCRs.
$Code_V$	Generates $\{K_V, K_V^{-1}\}$. Seals K_V^{-1} to $Code_V$.
V	Extends K_V into a PCR.
$RP \rightarrow V$	Attestation request and a random nonce n
$V \rightarrow RP$	K_V, TPM_Quote = $PCRs, Sign_{K_{AIK}^{-1}}(PCRs\|\|n), C_{AIK}$
RP	Check cert, sig, n, $PCRs$ represent $Code_V$ and K_V
$RP \rightarrow V$	$Policy, Sign_{K_{RP}^{-1}}(Policy)$
$RP \xrightarrow{*} F_i$	$K_V, Sign_{K_{RP}^{-1}}(K_V)$

Figure 5.3: **Verifier Attestation.** *V is the verifier, RP the relying party, F_i the filters, and C_{AIK} a certificate for the verifier's AIK. Section 2.3 has additional background on attestation.*

5. **Filter Independence.** A filter should not be able to generate Sender Tokens that are valid at other filters. This prevents a rogue filter from subverting other filters.

6. **Client and Filter Accountability.** The relying party should be able to distinguish between traffic generated by a malicious client and that generated by a malicious filter. Otherwise, a rogue filter can impersonate a sender.

5.2.3.2 Protocol Specifications

At a high-level, after verifying the trustworthiness of a verifier, the relying party installs the verifier's public key in each of the filters. The verifier, in turn, assesses the trustworthiness of clients. If a verification is successful, the verifier signs the client's public key to create a Sender Token. The client includes this token in each annotated message, and the client module generates annotations by signing its information (e.g., count of emails sent) using the client's private key. Below, we describe these interactions in more detail.

Verifier Attestation. Before giving a verifier the power to authorize client annotations, the relying party must ascertain that the verifier is in a correct, trusted state (Figure 5.3). It does so via an attestation (Section 2.3). The attestation convinces the relying party that the verifier is running trusted code, that only the trusted code has access to the verifier's private key, and that the keypair is freshly generated. Since the verifier token's validity is limited, the relying party periodically rechecks the verifier's correctness by rerunning the attestation protocol.

To prepare for an attestation, the verifier launches trusted verifier code. This code is measured by the platform, and the measurement is stored in the TPM's Platform Configuration Registers (PCRs). In practice (see Section 5.5), we use a late launch operation to measure and execute a minimal kernel and the code necessary to implement the verifier. The verifier code

C	Launches software $Code_C$. $Code_C$ recorded in PCRs.
$Code_C$	Generates $\{K_C, K_C^{-1}\}$. Seals K_C^{-1} to $Code_C$.
C	Extends K_C into a PCR.
$C \rightarrow V$	Token request
$V \rightarrow C$	Attestation request and a random nonce n
$C \rightarrow V$	K_c, TPM_Quote = $PCRs, Sign_{K_{AIK}^{-1}}(PCRs\|\|n), C_{AIK}$
V	Check cert, sig, n, $PCRs$ represent $Code_C$ and K_C
$V \rightarrow C$	$Token_C = [ID_V, K_C, expire_C, H(C_{AIK}),$ $Sign_{K_V^{-1}}(V\|\|K_C\|\|expire_C\|\|H(C_{AIK}))]$

Figure 5.4: **Client Attestation.** *C is the client, $Code_C$ is the MiniVisor protection layer from Section 5.2.2.1. V is the verifier, $expire_C$ is an expiration date for the sender's token, and H is a cryptographic hash function.*

$C \rightarrow Code_C$	Traffic contents p.
$Code_C$	Processes p to produce digest d.
$Code_C$	Generates a random nonce m.
$Code_C \rightarrow C$	$Annote_C = (m, d, Sign_{K_C^{-1}}(m\|\|d))$
$C \rightarrow RP$	$p, Token_C, Annote_C$

Figure 5.5: **Traffic Annotation.** *C is the client, $Code_C$ is the client module from Section 5.2.2.1 and RP is the relying party. The digest d represents the module's summary of network-relevant information about the client and/or traffic. The client sends the traffic to the relying party, but it will be processed along the way by one or more filters.*

generates a new public/private keypair and uses the TPM to seal the private key to the current software configuration. Thus, any change in the verifier's software will make the private key inaccessible.

After checking the verifier's attestation, the relying party instructs its filters to accept the verifier's new public key when processing annotated traffic. Since the filter is run by (or acts on behalf of) the relying party, it can be configured with the relying party's public key, and thus verify the authenticity of such updates.

Client Attestation. A similar process takes place when a client requests a Sender Token from a verifier (Figure 5.4). The client's MiniVisor generates a keypair and attests to the verifier that the private key is bound to the client module and was generated recently. If the client's attestation verifies correctly, the verifier returns a Sender Token consisting of the verifier's ID, the client's public key, an expiration date, and the verifier's signature.

Traffic Annotation. To annotate outbound traffic (e.g., an email or a packet), untrusted code on the client asks the client module to produce an annotation (Figure 5.5). The untrusted code passes the traffic's contents to the client module. The client module uses its internal state

```
1: if p contains Token_C, Annote_C then
2:    (ID_V, K_C, expire_C, H, Sig_V) ← Token_C
3:    Verify Sig_V using K_V.
4:    Use expire_C to check that Token_C has not expired.
5:    (m, d, Sig_C) ← Annote_C
6:    Verify Sig_C using K_C.
7:    Check that pair (K_C, m) is unique.
8:    Insert (K_C, m) into Bloom Filter.
9:    if All verifications succeed then
10:      Accept d as an authentic annotation of p
11:   else
12:      Drop p
13:   end if
14: end if
```

Figure 5.6: **Filtering Annotations.** *Processing of traffic contents p at a filter.*

to generate a digest d containing network relevant information about the traffic and/or client; i.e., it may indicate the average bandwidth used by the host, or the number of emails sent. Note that for most applications, the digest will include a hash of the traffic's contents to bind the annotation to a particular piece of traffic. Finally, the module produces an annotation that consists of a unique nonce, the digest, and the client's signature. Untrusted code can then add the client's Sender Token and annotation to the outbound traffic and send it to the relying party.

Annotation Checking. Filters that receive annotated traffic can verify its validity using the filtering algorithm shown in Figure 5.6. The filter uses the verifier's ID to look up the corresponding public key provided by the relying party. It uses the key to verify the authenticity and freshness of the client's Sender Token. The filter may optionally decide to cache these results to speed future processing. It then checks the authenticity and uniqueness of the annotation. It stores a record of the nonce to prevent duplication and accepts the validity of the annotation's digest if it passes all verification checks. However, if an annotation's verification checks fail, the filter drops the traffic. Legitimately generated traffic will only fail to verify if an on-path adversary modifies the traffic. Such an adversary can also drop or alter the traffic, so dropping malformed traffic does not increase the adversary's ability to harm the client.

5.2.3.3 A Symmetric Alternative

The protocols shown above possess all of the properties described in Section 5.2.3.1. Unfortunately, they require the client to compute a public-key signature for each item of traffic sent and the filter to verify two public-key signatures per annotation. The challenge is to improve the efficiency while retaining as many of the properties from Section 5.2.3.1 as possible.

At a high-level, instead of giving the verifier's public key to the filters, we establish a shared symmetric key between each verifier and all of the filters. Similarly, the client uses a symmetric, rather than a public, key to authenticate its annotations. The verifier provides the client with this key, which is calculated based on the symmetric key the verifier shares with the filters, as well as the information in the client's Sender Token. This makes it unnecessary for the verifier to MAC the client's token, since any changes to the token will cause the filters to generate an incorrect symmetric key, and hence to reject client's annotations. We describe these changes in more detail below.

Verifier Attestation. The last step of the protocol in Figure 5.3 is the only one that changes. Instead of sending the verifier's public key to all of the filters, the relying party generates a new symmetric key K_{VF}. The relying party encrypts the key using the verifier's newly generated public key and sends the verifier the resulting ciphertext ($Encrypt_{K_V}(K_{VF})$). Since the corresponding private key is sealed to the verifier's trusted code, the relying party guarantees that the symmetric key is protected. The relying party also encrypts the key and sends it to each of the filters, establishing a shared secret between the verifier and the filters.

Client Attestation. The protocol shown in Figure 5.4 remains the same, except for two changes. First, when the client sends its token request, it includes a randomly chosen client identifier ID_C. Second, to create the client's Sender Token, the verifier first computes a symmetric key that the client uses to authorize annotations:

$$K_{CF} = PRF_{K_{VF}}(V||ID_C||expire_C), \tag{5.1}$$

where PRF is a secure pseudo-random function. The verifier then sends the client:

$$Encrypt_{K_C}(K_{CF}), Token = (V, ID_C, expire_C).$$

The attestation convinces the verifier that K_C^{-1} is bound to trusted code, i.e., only trusted code can obtain K_{CF}. Without knowing K_{VF}, no one can produce K_{CF}.

Traffic Annotation. Traffic annotation is the same as before, except that instead of producing a signature over the traffic's contents, the code module produces a Message Authentication Code (MAC) using K_{CF}, an operation that is orders of magnitude faster.

Annotation Checking. The algorithm for checking annotations remains similar. Instead of checking the verifier's signature, the filter regenerates K_{CF} using Equation 5.1 and its knowledge of K_{VF}. Instead of verifying the client's signature on the annotation, the filter uses K_{CF} to verify the MAC. As a result, instead of verifying two public key signatures, the filter calculates one PRF application and one MAC, operations that are three orders of magnitude faster.

This scheme achieves the first four properties listed in Section 5.2.3.1, but it does not provide properties 5 and 6. Since each verifier shares a single symmetric key with all filters, a rogue filter can convince other filters to accept bogus annotations. We could prevent this attack by having the relying party establish a unique key for each verifier-filter pair, but this would violate another property. Either the verifier would have to MAC the client's Sender Token using all of the keys it shares with the filters (violating the fourth property), or the verifier would have to guess which filters would see the client's traffic, violating our topology-independence property.

Similarly, since the client and the filter share a symmetric key, the relying party cannot distinguish between malicious filters and malicious clients. Nonetheless, since the relying party's operator controls the filters, such risks should be acceptable in many applications, given the dramatic performance benefits offered by the symmetric scheme.

5.2.4 User Privacy and Client Revocation

To encourage adoption, Assayer must preserve user privacy, while still limiting clients to one identity per machine and allowing the relying party to revoke misbehaving clients. The Direct Anonymous Attestation (DAA) protocol [33] was designed to provide exactly these properties. However, as mentioned in Section 2.3.3, available TPMs do not yet implement this protocol, so until DAA becomes available on TPMs (or whatever secure hardware forms the basis for Assayer), Assayer must imperfectly approximate it using structured AIK certificates. We emphasize that this is a *temporary* engineering hack, not a fundamental limitation of Assayer, since DAA demonstrates that we can achieve both privacy and accountability.

Recall from Section 2.3.3 that TPM-equipped clients sign attestations using randomly generated attestation identity keys (AIKs). A Privacy CA issues a limited-duration certificate that vouches for the binding between an AIK and the original TPM Endorsement Key (EK). With Assayer, clients obtain AIK certificates that specify that the AIK is intended for communicating with a specific relying party. Using a different AIK for each relying party prevents the relying parties from tracking the client across sites. However, since all of the AIKs are certified by the same EK, they can all be bound to a single installation of MiniVisor, preventing an attacker from using a separate MiniVisor for each destination.

Of course, similar to issuing multiple DNS lookups using the same source IP address, this approach allows the Privacy CA to learn that *some* client intends to visit a particular set of relying parties. The DAA protocol eliminates both this linkage and the reliance on the Privacy CA.

To preserve user privacy with respect to a single relying party, the client can generate a new AIK and request a new certificate from the Privacy CA. However, Privacy CAs may only simultaneously issue one AIK certificate per relying party per TPM EK. Thus, a client could obtain a 1-day certificate for an AIK, but it could not obtain another certificate for the same relying party until the first certificate expires. This prevents a client from generating multiple *simultaneous* identities for communicating with a particular relying party.

Since each client token contains a hash of the client's AIK certificate, if the relying party decides a client is misbehaving, it can provide the hash to the Privacy CA and request that the Privacy CA cease providing relying party-specific AIK certificates to the EK associated with that particular AIK. This would prevent the client from obtaining new AIKs for communication with this particular relying party, though not for other relying parties. Similarly, the relying party can instruct its verifiers and filters to cease accepting attestations and annotations from that AIK.

5.3 Potential Attacks

In this section, we analyze potential attacks on the generic Assayer architecture and show how Assayer defends against them. We consider application-specific attacks in Section 5.4.

5.3.1 Exploited Clients

Code Replacement. An attacker may exploit code on remote, legitimate client machines. If the attacker replaces MiniVisor or the client module with malware, the TPM will refuse to unseal the client's private key, and hence the malware cannot produce authentic annotations. Without physical access to the client's machine, the attacker cannot violate these hardware-based guarantees.

Code Exploits. An adversary who finds an exploit in trusted code (i.e., in MiniVisor or a client module) can violate Assayer's security. This supports our argument that trusted client code should be minimized as much as possible. Exploits of untrusted code are less problematic. The relying party trusts MiniVisor to protect the client module, and it trusts the client module to provide accurate annotations. Thus, the trusted client module will continue to function, regardless of how the adversary exploits the untrusted code.

Flooding Attacks. Since an attacker cannot subvert the annotations, she might instead choose to flood Assayer components with traffic. As we explained in Section 5.2.2.2, the verifiers are designed to withstand DoS attacks, so flooding them will be unproductive. Since filters must already check annotations efficiently to prevent bottlenecks, flooding the filters (with missing, invalid, or even valid annotations) will not hurt legitimate traffic throughput.

Annotation Duplication. Since MiniVisor does not maintain control of the client's network interface, an attacker could ask the client module to generate an annotation and then repeatedly send the same annotation, either to the same filter or to multiple filters. Because each authorized annotation contains a unique nonce, duplicate annotations sent to the same filter will be dropped. Duplicates sent to different filters will be dropped as soon as the traffic converges at a single filter downstream. Section 5.5.4 discusses our Bloom Filter implementation for duplicate detection. Furthermore, duplicates are more likely to cause congestion problems close to the victim (since those links tend to be smaller), but paths also tend to converge close to the victim, minimizing the advantage of sending duplicate annotations.

Bypassing the Client Module. Since MiniVisor does not control the client's network interface, malicious software need not submit outbound messages to the client module. Instead of trying to lockdown the client's communication channels, we rely on application-specific incentives (see Section 5.4) to encourage both legitimate and malicious senders to obtain proper annotations. For example, in some applications, filters will drop messages that do not contain annotations. In others (such as DDoS mitigation), simply prioritizing annotated messages suffices to improve the performance of legitimate clients, even if malicious software sends non-annotated traffic.

5.3.2 Malicious Clients

Beyond the above attacks, an attacker might use hardware-based attacks to subvert the secure hardware on machines she physically controls. For example, the adversary could physically attack the TPM in her machine and extract its private keys. This would allow her to create a fake attestation, i.e., convince the verifier that the adversary's machine is running trusted Assayer code, when it is not.

However, the adversary can only extract N TPM keys, where N is the number of machines in her physical possession. This limits the attacker to N unique identities. Contacting multiple verifiers does not help, since sender identities are tracked based on their AIKs, not on their Sender Tokens. As discussed in Section 5.2.4, at any moment, each TPM key corresponds to exactly one AIK for a given relying party. Furthermore, to obtain a Sender Token from the verifier, the attacker must commit to a specific relying-party-approved client mod-

ule. If the attacker's traffic deviates from the annotations it contains, it can be detected, and the attacker's AIK for communicating with that relying party will be revoked. For example, if the attacker's annotations claim she has only sent X packets, and the relying party detects that the attacker has sent more than X packets, then the relying party knows that the client is misbehaving and will revoke the AIK the client uses to communicate with this relying party (see Section 5.2.4). Since the Privacy CA will not give the attacker a second AIK for the same relying party, this AIK can only be replaced by purchasing a new TPM-equipped machine, making this an expensive and unsustainable attack.

5.3.3 Rogue Verifiers

A rogue verifier can authorize arbitrary clients to create arbitrary annotations. However, the verifier's relatively simple task makes its code small and easy to analyze. The attestation protocol shown in Figure 5.3 guarantees that the relying party only approves verifiers running the correct code. Since verifiers are owned by the relying party or by someone with whom the relying party has a contractual relationship, local hardware exploits should not be a concern. Furthermore, since verifiers cannot imitate each other (even in the symmetric authentication scheme), a relying party that detects unusual or incorrect traffic coming from clients approved by a verifier can revoke that verifier. Revocation can be performed by refusing to renew the verifier's key material, or by actively informing the filters that they should discard the rogue verifier's public key.

5.3.4 Rogue Filters

A rogue filter can discard or mangle annotated traffic, or give priority to attack traffic. However, since it sits on the path from the client to the relying party, a rogue filter can already drop or alter traffic arbitrarily. In the asymmetric scheme (Section 5.2.3.2), a rogue filter cannot convince correct filters to elevate attack traffic, since it cannot generate correct verifier signatures necessary for the Sender Tokens. Similarly, a rogue filter cannot frame a client, since it cannot sign packets using the client's private key. The symmetric scheme trades off these properties in favor of greater efficiency. Fortunately, since the filters are directly administered by the relying party and perform a relatively simple task, rogue filters should rarely be a problem.

		Policy Creator	
		Recipient	**Network**
Focus	**Concentrated**	Spam	DDoS
	Diffuse	Spam	Super Spreaders

Figure 5.7: **Case Studies.** *Our case studies can be divided based on who determines acceptable policies and how focused the attack traffic is. For example, spammers can send a large amount of spam to one recipient or a few spam messages to many recipients.*

5.4 Case Studies

To evaluate the power of trustworthy host-based information, we consider the usefulness of Assayer in three diverse applications (see Figure 5.7). For each application, we motivate why Assayer can help, explain how to instantiate and deploy Assayer's components, and consider the motivations for clients to deploy the system. While we present evidence that Assayer-provided information can help in each application, it is beyond the scope of this work to determine the optimal statistics Assayer should provide or the optimal thresholds that relying parties should set.

5.4.1 Spam Identification

Motivation. Numerous studies [34, 87, 162] suggest that spam can be distinguished from legitimate email based on the sender's behavior. For example, one study found that the time between sending two emails was typically on the order of minutes [213], whereas the average email virus or spammer generates mail at a much higher rate (e.g., in the Storm botnet's spam campaigns, the average sending rate was 152 spam messages per minute, per bot [115]). Another study [87] found that the average and standard deviation of the size of emails sent over the last 24 hours were two of the best indicators of whether any individual email was spam.

These statistics can be difficult to collect on an Internet-scale, especially since spammers may adopt stealthy strategies that send only a few spam messages to each domain [162], but still send a large amount of spam in aggregate. However, the host generating the email is in an ideal position to collect these statistics.

Of course, host-based statistics are not sufficient to definitively identify spam, and they may falsely suggest that legitimate bulk email senders are spammers. However, by combining these statistics with existing spam classification techniques [108, 162], we expect that spam identification can be significantly improved for most senders.

Instantiation. To aid in spam identification, we can design a client module for annotating outbound email with relevant statistics, e.g., the average size of all emails generated during the last 24 hours. Each time the client generates a new email, it submits the email to the client module. The client module creates an authenticated annotation for the email with the relevant statistics, which the untrusted client email software can then add as an additional email header.

The relying party in this case would be the email recipient's mail provider, which would designate a set of verifiers. The filter(s) could simply be added as an additional stage in the existing spam identification infrastructure. In other words, the filter verifies the email's annotation and confirms the statistics it contains. These statistics can then be used to assess the probability that the message is spam. For example, an email from a sender who has not generated any email in the last 24 hours may be less likely to be marked as spam.

Client Incentives. Legitimate clients with normal sending behavior will have an incentive to deploy this system, since it will decrease the chance their email is marked as spam. Some mail domains may even require endhosts to employ a system like Assayer before accepting email from them.

Malicious clients may deploy Assayer, but either they will continue to send email at a high rate, which will be indicated by the emails' annotations, or they will be forced to reduce their sending behavior to that of legitimate senders. While not ideal, this may represent a substantial reduction in spam volume. Finally, malicious clients may send non-annotated spam, but as more legitimate senders adopt Assayer, this will make spam more likely to be identified as such.

5.4.2 Distributed Denial-of-Service (DDoS) Mitigation

Motivation. Like spam, DDoS is an attack in which adversaries typically behave quite differently from benign users. To maximize the effectiveness of a network-level attack, malicious clients need to generate as much traffic as possible, whereas a recent study indicates that legitimate clients generate much less traffic [31]. To confirm this, we analyzed eight days (135 GB of data representing about 1.27 billion different connections) of flow-level network traces from a university serving approximately 9,000 users. If we focus, for example, on web traffic, we find that 99.08% of source IP addresses never open more than 6 simultaneous connections to a given destination, and 99.64% never open more than 10. Similarly, 99.56% of source IP addresses send less than 10 KBps of aggregate traffic to any destination. This is far less than the client links permit (10-1000 Mbps). Since the traces only contain flow-level information, it is difficult to determine whether the outliers represent legitimate or malicious

traffic. However, other application traffic, such as for SSL or email, shows similar trends; the vast majority of users generate very little outbound traffic. This suggests that it is indeed possible to set relatively low thresholds to mitigate DDoS activity while leaving virtually all legitimate users unaffected.

Assayer enables legitimate hosts to annotate their traffic with additional information to indicate that their traffic is benign. For example, the client module might annotate each packet to show the rate at which the client is generating traffic. By prioritizing packets with low-rate annotations, filters ensure that, *during DDoS attacks*, legitimate traffic will be more likely to reach the server. A few non-standard legitimate clients may be hurt by this policy, but the vast majority will benefit. In the absence of a perfect solution, a utilitarian calculus suggests this approach is worthwhile.

Of course, this approach will only prevent attackers from sending large floods of traffic from each machine they control. They can still have each machine send a low rate of traffic, and, if they control enough machines, the aggregate may be enough to overwhelm the victim. Nonetheless, this will reduce the amount of traffic existing botnets can use for attack purposes and/or require attackers to build much larger botnets to have the same level of effect on the victim.

Numerous other systems have been proposed to fight DDoS. Packet capability systems, such as TVA [215] could use Assayer to decide whether to provide a client with a capability. Resource-based systems, such as Portcullis [155] or speak-up [206], attempt to rate-limit clients or enforce equal behavior amongst clients using checkable resource consumption, such as CPU or bandwidth. Assayer provides a more direct view into endhost behavior, but relies on secure hardware to bootstrap its guarantees. Overlay systems such as SOS [107] and Phalanx [50] use a large system of nodes to absorb and redirect attack traffic. This approach is largely orthogonal to Assayer, though combining these two approaches could be promising. For example, Assayer could keep track of overlay-relevant statistics on the host, or help provide per-client fair queuing within the overlay.

Instantiation. On the client, we modify the untrusted network stack to submit outbound packets to the Assayer client module. The client module generates an annotation indicating the number or size of packets generated recently. In this case, the relying party is the server the client is attempting to access (e.g., a website or software update server). The filters can prioritize annotated packets with "legitimate-looking" statistics over other traffic. To avoid hurting flows destined to other servers, filters give preference to annotated packets *relative* to other packets destined to that same server. This could be implemented via per-destination fair-queuing, with the Assayer-enabled server's queue giving priority to approved packets.

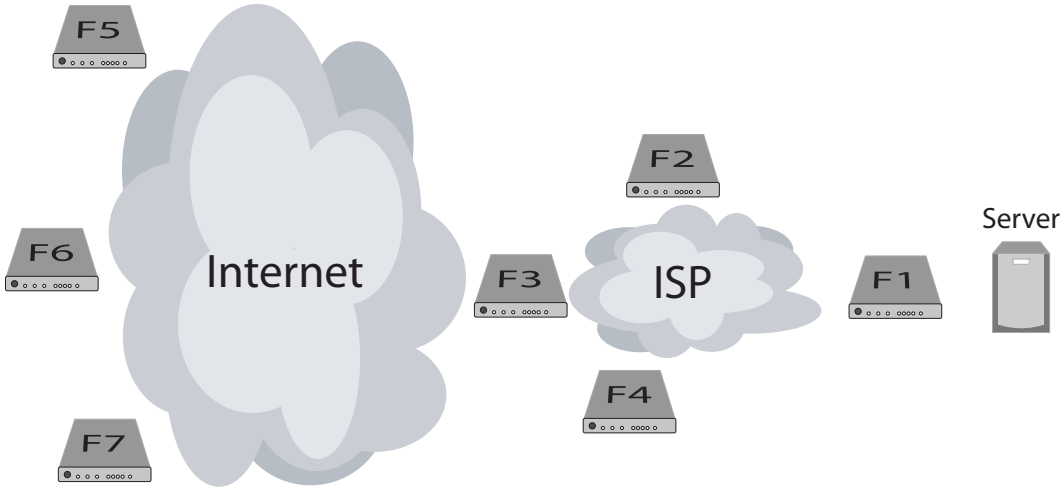


Figure 5.8: **Filter Deployment.** *Initially, a server is likely to deploy a single filter (F1) on its access link. For a fee, an ISP may decide to deploy filters (F2-F4) at its access links. The server may also be able to leverage filters scattered around the Internet (F5-F7) if it has business relations with these entities.*

To combat network-level DDoS attacks, we need to prioritize annotated packets as early as possible, before they reach the server's bottleneck. The filters must also be able to verify packet annotations efficiently at line rates to prevent the filters themselves from becoming bottlenecks. Section 5.6 shows that filters can perform these duties at reasonable speeds.

We envision filter deployment occurring in phases (see Figure 5.8), dictated by the server operator's needs and business relationships. Initially, to combat application-level attacks, the server operator may simply deploy a single filter in front of (or as a part of) the server. However, to combat network-level attacks, the server's operator may contract with its ISP to deploy filters at the ISP's ingress links. Similar arrangements could be made with other organizations around the network, depending on the business relationships available. In the long run, filters will likely become standardized, shared infrastructure that is deployed ubiquitously. However, as we show in Section 5.6.4, partial deployment can provide significant protection from attacks.

Client Incentives. From the client's perspective, the client proves that it is generating traffic at a moderate rate in exchange for elevated service from the network and server. This makes it more likely that the client can access the web services it desires, even during DDoS attacks.

5.4.3 Super-Spreader Worm Detection

Motivation. A super-spreader worm exploits a host and then begins rapidly scanning hundreds or even thousands of additional hosts for potential vulnerabilities. Again, studies show that rapidly contacting multiple hosts, as such worms do, is quite unlike typical user behavior [212]. While detecting such traffic is relatively straightforward on an enterprise scale (using tools such as NetFlow), detecting this behavior on an Internet scale is much more challenging, since any individual monitor has only a limited view of an endhost's behavior, and the amount of traffic sent to each host is so small (often no more than a single packet) that it can be missed by sampling algorithms.

These observations have lead to the development of multiple algorithms [204, 208] for trading detection accuracy for reduced state and processing overhead. However, ultimately, the host generating the scanning traffic is in the best position to detect this behavior, since it can keep a perfectly accurate local count of how many hosts it has contacted, assuming that the network can indeed trust the host's count. While middleboxes might perform this service in enterprise networks, home networks and small businesses are less likely to install such additional hardware. An Assayer approach would also allow remote destinations to verify that incoming packets do not constitute worm propagation. Nonetheless, deployment issues remain, as we discuss below.

Instantiation. For this application, the client software would again submit packets to the client module, which would produce annotations indicating the number of destinations contacted in the last X minutes. The relying party could be a backbone ISP hoping to avoid worm-based congestion, or a stub ISP protecting its clients from worm infections. In-network filters can verify the authenticity of these annotations and drop or delay packets from hosts that have contacted too many destinations recently. Like in the DDoS application, filters would need to be deployed at the edges of the relying party's administrative domain and would need to be able to verify annotations at line rate.

Client Incentives. While it is interesting to see that Assayer enables super-spreader worm detection from a technical perspective, there are several challenges in incentivizing clients and in dealing with non-annotated traffic. These challenges suggest that, despite its technical feasibility, Assayer may not be an optimal choice for this application.

While a next-generation clean-slate network could simply mandate the use of Assayer to detect super-spreader worms, deploying Assayer in a legacy environment faces a bootstrapping challenge. Unlike in the DDoS application, it is not sufficient to simply prioritize annotated traffic over non-annotated traffic, since lower-priority traffic will still spread the worm.

Instead, non-annotated traffic must be significantly delayed or dropped to have a credible chance of slowing or stopping worm propagation. However, in a legacy environment, ISPs cannot slow or drop legacy traffic until most users have started annotating their traffic, but users will not annotate their traffic unless motivated to do so by the ISPs. A non-technical approach would be to hold users liable for any damage done by non-annotated packets, thus incentivizing legitimate users to annotate their packets. This obviously raises both legal and technical issues.

5.5 Implementation

To evaluate the effectiveness and performance of Assayer, we have developed a basic prototype system. Because these are prototypes, they give rough upper-bounds on Assayer's performance impact, but considerable room for optimization remains.

We implemented MiniVisor along with appropriate client modules for our applications. Our client software can attest to MiniVisor's presence, and these attestations are checked by our verifier prototype. We have implemented a basic filter to verify annotations. To evaluate Assayer's performance at the packet level, we incorporated the filter's functionality into the Click router [114]. We present our performance results in Section 5.6.

5.5.1 Client Architecture

We implemented the client configuration shown in Figure 5.2 employing a tiny hypervisor called MiniVisor that we developed using hardware-virtualization support available from both AMD and Intel. We also evaluated using the Flicker architecture (see Chapter 4) to protect the client module with an even smaller TCB, but we found that context switching into and out of Flicker increased annotation generation time by 1-2 orders of magnitude, and hence we decided it is not yet practical for performance critical applications. As discussed in Section 4.6, hardware improvements are expected to make Flicker viable in these scenarios.

Since MiniVisor does not interact with any devices, we were able to implement it in 841 lines of code and still (as shown in Section 5.6) offer excellent performance. It supports a single hypercall that allows untrusted code to submit traffic to a client module and receive an annotation in return. MiniVisor's implementation is similar in spirit to that of SecVisor [174] and TrustVisor [130]. However, SecVisor focuses on kernel integrity protection, while TrustVisor provides a much more general interface than MiniVisor.

We employ a late launch operation (recall Section 2.4.2) to simplify client attestations by removing the early boot code (e.g., the BIOS and bootloader) from the set of trusted

code. When MiniVisor is late launched, it uses shadow page tables to isolate its own private memory area and then boots the Linux kernel for normal client usage (since we employ hardware virtualization, MiniVisor could equally well launch Windows instead). The client attestations consist of the protection layer (MiniVisor), a client module, and the fact that the protection layer is configured to properly isolate the module.

For packet-based applications (e.g., DDoS mitigation and super-spreader detection), we use Linux's TUN/TAP interface[1] to re-route outbound packets to a user-space program. The user-space program invokes MiniVisor's hypercall to obtain an annotation and then routes the packet back to the physical interface. This configuration simplified development, but it is less than optimal from a performance standpoint, since packets are passed across the user-kernel space divide multiple times. Intercepting packets inside of a network driver or kernel module would improve our performance. For these applications, adding annotations to packets could potentially cause considerable packet fragmentation. To prevent this, we reduce the MTU on the network interface facing untrusted code by the number of bytes in a standard annotation. Of course, untrusted code can increase the MTU, but that will merely hurt performance, not security.

5.5.2 Client Verification

Regardless of the application, the client must be able to create an attestation that can be checked by a verifier. Thus, we developed generic client software to produce the attestations, as well as a verifier server program to check the attestations and produce client tokens. Together, they implement the protocol shown in Figure 5.4. Since the code that allows the relying party to check verifier attestations (Figure 5.3) is similar (and less performance-sensitive), we describe and evaluate only the client attestation and verification implementations.

Client Attestations. Before it can create an attestation, our client code first generates an AIK and obtains an AIK certificate from a Privacy CA. To create an attestation, the client contacts the verifier and requests a nonce. Given the verifier's nonce, the client invokes a TPM_Quote operation. It sends the verifier the public key created by its code module, the contents of the PCRs, the list of the code described by the PCRs, the TPM's signature and the AIK certificate. The verifier checks the validity of the certificate, verifies the TPM's signature, checks that the nonce value is the same one it sent, and finally checks to make sure the PCR values reflect an appropriate version and configuration of MiniVisor. Assuming these checks pass, it returns an appropriate Sender Token (we discuss the token components in more detail below).

[1] http://vtun.sourceforge.net/tun/

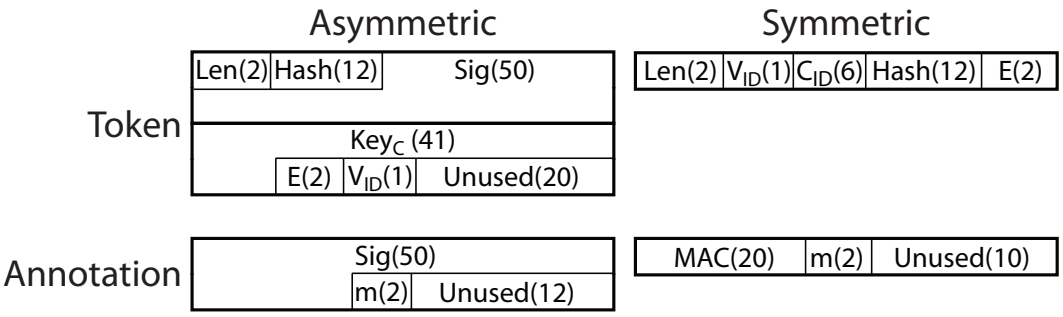


Figure 5.9: **Token and Annotation Layout.** *Byte-level layout for Sender Tokens and traffic annotations. The two are shown separately for clarity, but in practice, would be packed together. E is an expiration date, and m is a randomly-chosen nonce.*

Verifier Implementation. Our verifier prototype is implemented as a simple user-space server program. The implementation is based on a Unix/Linux preforked server library (spprocpool)[2], and the client and the verifier communicate using UDP. The verifier pre-forks several worker processes and waits for client connections. When it receives a connection, the verifier passes this connection to an idle worker process. The worker process chooses a random nonce for the client and verifies the resulting attestation. A more sophisticated server architecture would undoubtedly improve our system's performance, but this simple prototype gives us a lower bound on a verifier's potential performance.

5.5.3 Traffic Annotation

To evaluate their relative performance, we implemented both the asymmetric and symmetric protocols for generating annotations (Section 5.2.3). Figure 5.9 illustrates the layout of the Sender Tokens and traffic annotations for each scheme.

With both schemes, we add the Sender Token and the annotation to the payload itself, and then adjust the appropriate header fields (length, checksum, etc.). This provides compatibility with legacy network devices. The traffic recipient needs to remove this information before handing the payload to applications, but this is simple to implement. Of course, legacy traffic will not contain annotations, and hence is easy to identify and handle in an application-specific manner.

With the asymmetric scheme, we use elliptic curve cryptography to minimize the size of the client's public key, since it is included in the client's Sender Token and hence requires space in every packet. We use the secp160k1 curve [194], which provides approximately 80

[2] http://code.google.com/p/spprocpool/

bits of cryptographic strength. The verifier uses the elliptic curve version of the digital signature algorithm (ECDSA) to sign the client's token, and the client also uses ECDSA to sign the contents of authorized packets. In sum, the client's token takes 108 bytes, and the annotation requires 52 bytes.

With the symmetric scheme, if we use a 160-bit key with SHA1-HMAC (which remains secure, despite recent collision attacks on SHA1), then the client's token only requires 23 bytes, and the annotation requires 22 bytes.

5.5.4 Filter

We implemented the filter's functionality (Figure 5.6) both in userspace (for applications such as spam filtering), and as a module for the Click router [114] (for applications such as DDoS mitigation and super-spreader detection). Both implementations check traffic (either email or packets) for an Assayer flag in the header fields. If present, the filter checks the Sender Token and the annotation, following the algorithm in Figure 5.6. If all checks succeed, the packet is added to a priority queue.

With the asymmetric scheme, the filter needs to verify both the verifier's ECDSA signature in the client's token and the client's ECDSA signature in the annotation. With the symmetric scheme, the filter needs to generate the shared symmetric key and verify the client's SHA1-HMAC in the annotation.

To detect duplicate annotations, we use a Bloom Filter [30]. We only insert an annotation into the Bloom Filter after verifying the Sender Token and the annotation. The Bloom Filter ensures that a valid annotation is unique in a given time period t with a bounded false positive probability γ.

To illustrate this, suppose that the filter receives N valid annotations/second, each accompanied by a valid Sender Token. If we use K different hash functions, and N different annotations are added into a Bloom Filter of M bits, then γ is approximately $(1-e^{\frac{-KN}{M}})^K$ [30]. For network applications, suppose the filter operates on a 1 Gbps link. In the worst case, the filter would receive n packets/sec, where n is the link's capacity divided by the minimum packet size, and all of these packets carry valid annotations. In the symmetric scheme, $n = 1,262,626$ packets/second. Thus, to limit the false positive probability to less than $\frac{1}{10^6}$ per annotation, we need a 2MB Bloom Filter with 20 hash functions. Similar calculations can be applied to less performance-intensive applications, such as spam identification.

If we use public hash functions in our Bloom Filter, an adversary could use carefully chosen inputs to pollute the Bloom Filter, i.e., use a few specially-crafted annotations to set nearly all the bits in the Bloom Filter to 1. This attack would dramatically increase the false

positive rate and break the duplicate detection. Thus, we use a pseudorandom function (AES) with a secret key known only to the filter to randomize the input to the Bloom Filter. Thus, attackers cannot pollute the Bloom Filter with chosen input.

5.6 Evaluation

To identify potential performance bottlenecks in the Assayer architecture, we evaluated the performance of each prototype component and compared our two authentication schemes. In the interest of space, and since our spam detection application is far less latency-sensitive than our packet-level applications (DDoS and worm mitigation), we focus our evaluation on Assayer's packet-level performance. We also developed an Internet-scale simulator to evaluate how Assayer performs against DDoS attacks by large botnets.

We find that, as expected, the symmetric authentication scheme outperforms the asymmetric scheme by 1–2 orders of magnitude. Using the symmetric scheme, MiniVisor's performance is quite close to native, with network overheads ranging from 0-11%. Our verifier can sustain about 3300 verifications/second, and the filter can validate Assayer traffic with only a 3.7-18.3% decrease in throughput (depending on packet size). Finally, our simulations indicate that even sparse deployments (e.g., at the victim's ISP) of Assayer offer strong DDoS mitigation during large-scale attacks.

In our experiments, our clients and verifier run on Dell Optiplex 755s, each equipped with a 3 GHz Intel Core2 Duo and 2 GB of RAM. The filter has one 2.4 GHz Intel(R) Pentium(R) 4 with 512 MB of memory. All hosts are connected via 1 Gbps links.

5.6.1 Client Verification

We measure the time it takes a single client to generate an attestation and obtain a Sender Token from a verifier. We also evaluate how many simultaneous clients our verifier supports.

5.6.1.1 Client Latency

Since clients request new Sender Tokens infrequently (e.g., once a week), the latency of the request is unlikely to be noticed during normal operation. Nonetheless, for completeness, we measured this time using our prototype client and verifier and found that the client takes an average of 795.3 ms to obtain a Sender Token. The vast majority (99.7%) of the time is spent obtaining a quote from the TPM, since the quote requires the calculation of a 2048-bit RSA signature on a resource-impoverished TPM processor. The verifier only spends a total

Annotation Size	Symmetric	Asymmetric
10 B	2.11	1166.40
100 B	3.15	1156.75
1,000 B	5.00	1154.20
10,000 B	27.20	1180.45
100,000 B	247.55	1396.40
1,000,000 B	2452.15	3597.95
10,000,000 B	24696.25	25819.75

Figure 5.10: **Annotation Generation.** *Average time required to generate annotations using our symmetric and asymmetric protocols. All times are in microseconds.*

of 1.75 ms processing the client's request using the symmetric scheme and 3.58 ms using the asymmetric scheme.

5.6.1.2 Verifier Throughput

To test the throughput of the verifier, we developed a minimal client program that requests a nonce and responds with a pre-generated attestation as soon as the verifier responds. The client employs a simple timeout-based retransmission protocol. We launch X clients per second and measure the time it takes each client to receive its Sender Token. In our tests, each of our 50 test machines simulates 20-500 clients.

In 10 trials, we found that a single verifier using the symmetric scheme can serve a burst of up to 5700 clients without any UDP retransmission, and can sustain an average rate of approximately 3300 clients/second. With the asymmetric scheme, a verifier can serve 3800 clients in a burst, and can sustain about 1600 clients/second. This implies that our simple, unoptimized verifier prototype could, in a day, serve approximately 285 million clients with the symmetric scheme and 138 million clients with the asymmetric scheme.

5.6.2 Client Annotations

With Assayer, clients must compute a signature or MAC for each annotation. Annotating traffic adds computational latency and reduces effective bandwidth, since each traffic item (e.g., email or packet) carries fewer bytes of application data.

Figure 5.10 summarizes the results of our microbenchmarks examining the latency of generating annotations of various sizes. All results are the average of 20 trials. We expect applications such as DDoS mitigation and super-spreader worm detection to require small annotations (since the annotation must fit in a packet), while spam identification may require larger annotations. The symmetric scheme's performance is dominated by the cost of hashing

the annotation. With the asymmetric scheme, the performance for smaller annotations is dominated by the time required to generate the ECDSA signature. Thanks to MiniVisor's use of hardware support for virtualization, context switching to the client module is extremely fast (approximately 0.5 μs).

For macrobenchmarks, since email is designed to be delay tolerant, we focus on quantifying Assayer's effect on packet-level traffic. Thus, we evaluate the effect of annotating each outbound packet with the number of packets sent, along with a hash of the packet's contents. We first ping a local host (**Ping L**), as well as a host across the country (**Ping R**). This quantifies the computational latency, since each ping only uses a single packet and bandwidth is not an issue. We then fetch a static web page (8 KB) (**Req L/R**) and download a large (5 MB) file from a local web server and from a web server across the country (**Down L/R**). These tests indicate the performance impact a user would experience during an average web session. They require our client module to annotate the initial TCP handshake packets, the web request, and the outbound acknowledgements. To quantify the impact of Assayer's bandwidth reduction, we also measure the time to upload a large (5 MB) file (**Up L/R**). This test significantly increases the number of packets the client module must annotate.

We performed the above experiments using both the asymmetric and the symmetric schemes described in Section 5.2.3. Figure 5.11 summarizes our results. These results confirm our suspicion that the symmetric scheme offers significantly better performance than the asymmetric scheme. The symmetric scheme adds less than 12% overhead, even in the worst-case tests that involve uploading a large file. In many cases, the difference between the symmetric scheme and native Linux is statistically insignificant. The asymmetric scheme, on the other hand, adds significant overhead, though the effects are mitigated for remote hosts, since round-trip times occupy a large portion of the test. We could reduce the overhead by selecting a scheme that allows more efficient signing, but this would increase the burden on the filters.

5.6.3 Filter Throughput

In order to evaluate the filter's throughput inspecting packet-level annotations, we use the Netperf tools[3] running on a client machine to saturate the filter's inbound link with annotated packets. To compare our various schemes, we launch the Netperf TCP_STREAM test using 512-byte packets, which is close to the average packet size on the Internet [198]. We then experiment with varying packet sizes.

[3] http://www.netperf.org

	Native Linux	Assayer Symmetric	Assayer Asymmetric
Ping L	0.817±0.32	0.811±0.13 (-0.1%)	2.104±0.31 (+157.5%)
Ping R	11.91 ±1.90	11.99 ±3.24 (+0.1%)	14.03 ±3.67 (+17.8%)
Req L	3.129±0.03	3.48 ±0.26 (+11.3%)	12.27 ±4.18 (+292.1%)
Req R	45.83 ±12.3	44.07 ±6.93 (-0.4%)	51.35 ±12.1 (+12.0%)
Down L	1339. ±348	1427. ±382 (+6.6%)	2634. ±114 (+96.7%)
Down R	5874. ±1000	5884. ±990 (+0.2%)	6631. ±721 (+12.8%)
Up L	706.5 ±61.4	777.4 ±153 (+10.0%)	5147. ±177 (+628.5%)
Up R	3040. ±568	3078. ±1001 (+0.1%)	6234. ±961 (+105.1%)

Figure 5.11: **Performance of Client Annotations: Symmetric vs. Asymmetric.** *L represents a local request, and R represents a remote request. All times are shown in milliseconds rounded to four significant figures. Values in parentheses represent the change versus the native configuration.*

	Throughput (Mbps)	% of Click
Basic Click (user)	124	-
Sym Filter (user)	87	70.1%
AsymFilter (user)	2	1.7%
Basic Click (kernel)	225	-
Sym Filter (kernel)	154	68.4%
Sym Filter (kernel, no dup)	169	75.1%
Sym Filter (kernel, UMAC)	204	90.7%

Figure 5.12: **Packet Filtering Performance.** *"User" and "kernel" denote user-level and kernel-level mode. "Sym" and "asym" denote the symmetric scheme and the asymmetric scheme. "Basic Click" is the basic click router which simply forwards each packet. "no dup" means no duplicate detection operations are performed. All tests employ 512-byte packets.*

In our experiments (Figure 5.12), we found that a user-level basic Click router, which simply forwards all packets, could sustain a throughput of approximately 124 Mbps. A user-level filter implementing our symmetric annotation scheme has about 87 Mbps throughput, while a filter using the asymmetric scheme can only sustain approximately 2 Mbps.

By implementing the filter as a Click kernel module, we improve the performance of the filter using the symmetric scheme to about 154 Mbps, while the performance of a kernel-level basic Click router is approximately 225 Mbps. The filter using the symmetric scheme performs two major operations: verifying packet annotations and detecting duplicate annotations (see Section 5.2.3.3). Eliminating the duplicate detection operation only slightly improves the filter's throughput (up to 169 Mbps), which indicates that verifying the packet annotation is the significant performance bottleneck.

To confirm this, we modify our packet annotation implementation to use UMAC [116] instead of SHA1-HMAC. UMAC is much faster than SHA1-HMAC if the key has been set up, but with Assayer, a key is generated from the client's token and set up for every packet. This would make UMAC slower than SHA1-HMAC. To improve UMAC's performance, we implement a key cache mechanism that only generates and sets up a UMAC key for the first packet of every network flow, since all of the packets in a network flow will have the same Sender Token. Measurements indicate that the average Internet flow consists of approximately 20 packets [198]. Using this measurement as a rough estimate of our key cache's effectiveness, our filter's performance improves to 204 Mbps. This represents a 9.3% performance loss relative to a kernel-level basic Click router.

Finally, we vary the packet size used in our experiments. We find that our UMAC-based symmetric filter undergoes a 18.3% performance loss relative to Click when using 100 byte packets, whereas it comes within 3.7% of Click when using 1500 byte packets.

5.6.4 Internet-Scale Simulation

Finally, to evaluate Assayer's effectiveness for DDoS mitigation, we developed an Internet-scale simulator. The simulation's topology was developed from the CAIDA Skitter probes of router-level topology [37]. The Skitter map forms a rooted tree at the trace source and spans out to over 174,000 endpoints scattered largely uniformly around the Internet. We make the trace source the victim of the DDoS attack and then randomly select 1,000 endpoints to represent legitimate senders and 100,000 endpoints to represent attackers. We assume that legitimate senders have obtained Sender Tokens, whereas the attackers simply flood (since flooding with Sender Tokens will result in the revocation of the attacker's keys – see Section 5.2.4).

Since the Skitter map does not include bandwidth measurements, we use a simple bandwidth model in which endhost uplinks have one tenth the capacity of the victim's network connection, while the rest of the links have ten times that capacity. Thus, each endhost has a small uplink that connects it to a well-provisioned core that narrows down when it reaches the victim. To make these values concrete, senders have 10 Mbps connections, the victim has a 100 Mbps link, and the links in the middle of the network operate at 1 Gbps. As a result, ten attackers can saturate the victim's network connection. In our experiments, legitimate senders make one request every 10 ms, while attackers flood the victim with requests at their maximum uplink capacity (i.e., 10 Mbps). Attackers are allowed to start sending until the network is saturated (so that legitimate senders face the full brunt of the DDoS attack), and then we measure how long it takes legitimate senders to contact the server.

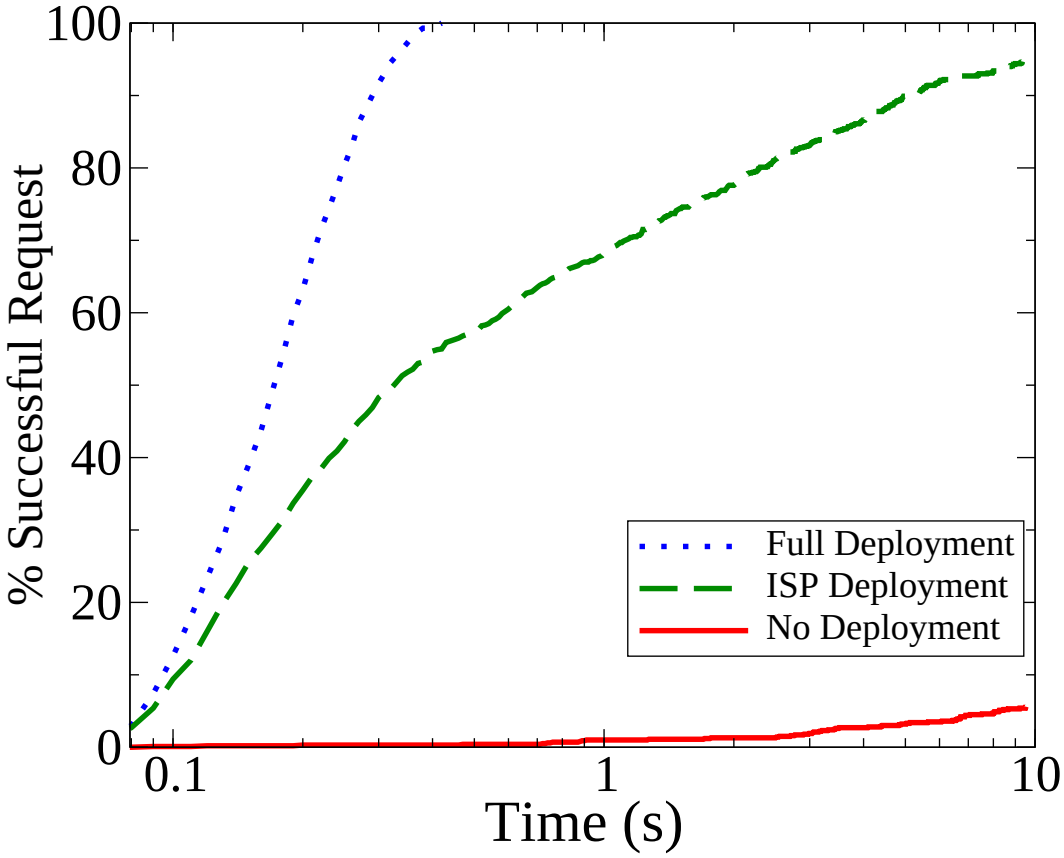


Figure 5.13: **Internet-Scale Simulations.** *Time for 1,000 senders to contact the server in the presence of 100,000 attackers. Note that the X-axis is on a logarithmic scale.*

We run our simulations with no Assayer deployment, with Assayer filters deployed at the victim's ISP, and with ubiquitous (full) Assayer deployment. Figure 5.13 shows the amount of time it takes legitimate senders to contact the server. With no deployment, less than 6% of legitimate clients can contact the server, even after 10 seconds. With a full deployment of Assayer, most clients contact the server within one RTT, which is unsurprising given that legitimate traffic enjoys priority over the attack traffic throughout the network. However, even with partial deployment at the victim's ISP, more than 68% of legitimate clients succeed in less than a second, and 95% succeed within 10 seconds, even in the face of a DDoS attack by 100,000 endhosts.

5.7 Potential Objections

In this section, we consider potential objections to the notion of conveying endhost information to the network.

5.7.1 Why Not Collect Information on the Local Router?

One might imagine that, instead of collecting statistics on the host, we could collect them on the local router, which has almost as good a view of host behavior as the host itself. However, this would misalign resource and incentives. Compared to endhosts, routers tend to be resource-impoverished and lack the secure hardware to convey information to remote parties. Even if they had such hardware, it is much harder to decide how much traffic a router should be allowed to send. For example, a spam filter might be able to develop a reasonable model of how much email a normal user (represented by a single TPM-equipped machine) might send [87], but it seems more challenging to model how much email a TPM-equipped router might be expected to forward. Furthermore, a sender's ISP has relatively little incentive to help a remote destination filter out spam, whereas the user has a strong incentive to ensure her email is not marked as spam. Nonetheless, as we note in Section 5.8, we consider the development of an ISP-based proxy to be an interesting direction for future work.

5.7.2 Is This Really Deployable Incrementally?

Yes, in the sense that for many applications, once a single server deploys an Assayer filter, individual senders can upgrade to Assayer and immediately see benefits. For example, a server concerned about DDoS can install an Assayer filter. Any client that upgrades will see its traffic prioritized during an attack, whereas legacy clients (easily distinguished by the lack of annotations) will notice the same degradation they would today, potentially incentivizing them to upgrade as well. As discussed in Section 5.4.3, not all applications are structured this way; one contribution of this work is to identify which applications will immediately benefit from an Assayer approach, and which face deployment challenges.

5.8 Summary

Many interesting and useful host-based properties are difficult or expensive to calculate external to the host, but simple to obtain on the host itself. In this chapter, we show that the growing ubiquity of trusted hardware on endhosts offers a powerful opportunity: a small

amount of software on the endhosts can be trusted to provide useful information to receivers and network elements. Even local malware cannot interfere with the information provided. This approach enables fundamentally different network security mechanisms for confronting network attacks such as spam, DDoS, and worms, since network filters can make decisions based on the host-supplied data.

Chapter 6

Verifiable Computing: Secure Code Execution Despite Untrusted Software and Hardware

Several trends are contributing to a growing desire to "outsource" computing from a (relatively) weak computational device to a more powerful computation service. For years, a variety of projects, including SETI@Home [10], Folding@Home [153], and the Mersenne prime search [138], have distributed computations to millions of clients around the Internet to take advantage of their idle cycles. A perennial problem is dishonest clients: end users who modify their client software to return plausible results without performing any actual work [145]. Users commit such fraud even when the only incentive is to increase their relative ranking on a website listing. Many projects cope with such fraud via redundancy: the same work unit is sent to several clients and the results are compared for consistency. Apart from wasting resources, this provides little defense against colluding users.

A related fear plagues cloud computing, where businesses buy computing time from a service, rather than purchasing and maintaining their own computing resources [6, 151]. Sometimes the applications outsourced to the cloud are so critical that it is imperative to rule out accidental errors during the computation. Moreover, in such arrangements, the business providing the computing services may have a strong financial incentive to return incorrect answers, if such answers require less work and are unlikely to be detected by the client.

The proliferation of mobile devices, such as smart phones and netbooks, provides yet another venue in which a computationally weak device would like to be able to outsource a computation, e.g., a cryptographic operation or a photo manipulation, to a third party and yet obtain a strong assurance that the result returned is correct.

In all of these scenarios, the computations are performed on a remote machine over which the user does not have physical control. Unable to guarantee the physical security of these machines, the user cannot assume that they have not been physically attacked or tampered with. Thus, solutions such as Flicker (Chapter 4) will not suffice, since they rely on commodity components (such as the CPU, RAM, and TPM) that are not designed to be tamper-proof (see Section 2.9.2). As a result, we require techniques that will be resilient to malicious software *and* hardware.

A key constraint is that the amount of work performed by the client to generate and verify work instances must be substantially cheaper than performing the computation on its own. It is also desirable to keep the work performed by the workers as close as possible to the amount of work needed to compute the original function. Otherwise, the worker may be unable to complete the task in a reasonable amount of time, or the cost to the client may become prohibitive.

Contributions. In this chapter, we make the following contributions: **(1)** We formally define the notion of *Verifiable Computing* to capture the outsourcing of computation to a remote party. **(2)** We design a protocol that provably meets these definitions, while providing asymptotically optimal performance (with regard to CPU and bandwidth) when amortized over multiple function inputs. **(3)** We provide the first protocol that provides secrecy of inputs and outputs in addition to verifiability of the outputs.

6.1 Overview

We define the notion of a *Verifiable Computation Scheme* as a protocol between two polynomial-time parties, a *client* and a *worker*, to collaborate on the computation of a function $F : \{0,1\}^n \rightarrow \{0,1\}^m$. Our definition uses an amortized notion of complexity for the client: he can perform some expensive pre-processing, but after this stage, he is required to run very efficiently. Since the preprocessing stage happens only once, it is important to stress that it can be performed in a trusted environment where the weak client, who does not have the computational power to perform it, outsources it to a trusted party (think of a military application in which the client loads the result of the preprocessing stage performed inside the military base by a trusted server, and then goes off into the field where outsourcing servers may not be trusted anymore – or think of the preprocessing phase as being executed on the client's home machine and then used by his portable device in the field).

By introducing a one-time preprocessing stage (and the resulting amortized notion of complexity), we can circumvent the result of Rothblum and Vadhan [165], which indicated

that efficient verifiable computation requires the use of probabilistically checkable proof (PCP) constructions. In other words, unless some a substantial improvement in the efficiency of PCP constructions is achieved, our model potentially allows much simpler and more efficient constructions than those possible in previous models.

More specifically, a verifiable computation scheme consists of three phases:

Preprocessing A one-time stage in which the client computes some auxiliary (public and private) information associated with F. This phase can take time comparable to computing the function from scratch, but it is performed only once, and its cost is amortized over all the future executions.

Input Preparation When the client wants the worker to compute $F(x)$, it prepares some auxiliary (public and private) information about x. The public information is sent to the worker.

Output Computation and Verification Once the worker has the public information associated with F and x, it computes a string π_x which encodes the value $F(x)$ and returns it to the client. From the value π_x, the client can compute the value $F(x)$ and verify its correctness.

Notice that this is inherently a non-interactive protocol: the client sends a single message to the worker and vice versa. The crucial efficiency requirement is that Input Preparation and Output Verification must take less time than computing F from scratch (ideally linear time, $O(n + m)$). Also, the Output Computation stage should take roughly the same amount of computation as F.

After formally defining the notion of verifiable computation, we present a verifiable computation scheme for *any* computable function. Assume that the function F is described by a Boolean circuit C. Then the Preprocessing stage of our protocol takes time $O(|C|)$, i.e., time linear in the size of the circuit C that the client would have used to compute the function on its own. Apart from that, the client runs in linear time, as Input Preparation takes $O(n)$ time and Output Verification takes $O(m)$ time. Finally the worker takes time $O(|C|)$ to compute the function for the client. Our protocol also provides the client with input and output privacy, a property not considered in previous work

The computational assumptions underlying the security of our scheme are the security of block ciphers (i.e., the existence of one-way functions) and the existence of a secure fully homomorphic encryption scheme [68, 69] (more details below). Thus, the security of our scheme can be proven in the standard model (as opposed to Micali's proofs [140] which require the random oracle model).

Dynamic and Adaptive Input Choice. We note that in this amortized model of computation, Goldwasser et al.'s protocol [74] can be modified using Kalai and Raz's transformation [102] to achieve a non-interactive scheme (see [164]). However an important feature of our scheme, that is not enjoyed by Goldwasser et al.'s protocol [74], is that the inputs to the computation of F can be chosen in a dynamic and adaptive fashion throughout the execution of the protocol (as opposed to [74] where they must be fixed and known in advance).

Privacy. We also note that our construction has the added benefit of providing input and output privacy for the client, meaning that the worker does not learn any information about x or $F(x)$ (details below). This privacy feature is bundled into the protocol and comes at no additional cost. This is a critical feature for many real-life outsourcing scenarios in which a function is computed over highly sensitive data (e.g., medical records or trade secrets). Our work therefore is the first to provide a weak client with the ability to efficiently and verifiably offload computation to an untrusted server in such a way that the input remains secret.

Our Solution in a Nutshell. Our work is based on the crucial (and somewhat surprising) observation that Yao's Garbled Circuit Construction [216, 217], in addition to providing secure two-party computation, also provides a "one-time" verifiable computation. In other words, we can adapt Yao's construction to allow a client to outsource the computation of a function on a single input. More specifically, in the preprocessing stage the client garbles the circuit C according to Yao's construction. Then in the "input preparation" stage, the client reveals the random labels associated with the input bits of x in the garbling. This allows the worker to compute the random labels associated with the output bits, and from them the client will reconstruct $F(x)$. If the output bit labels are sufficiently long and random, the worker will not be able to guess the labels for an incorrect output, and therefore the client is assured that $F(x)$ is the correct output.

Unfortunately, reusing the circuit for a second input x' is insecure, since once the output labels of $F(x)$ are revealed, nothing can stop the worker from presenting those labels as correct for $F(x')$. Creating a new garbled circuit requires as much work as if the client computed the function itself, so on its own, Yao's Circuits do not provide an efficient method for outsourcing computation.

The second crucial idea of the paper is to combine Yao's Garbled Circuit with a fully homomorphic encryption system (e.g., Gentry's recent proposal [69]) to be able to safely reuse the garbled circuit for multiple inputs. More specifically, instead of revealing the labels associated with the bits of input x, the client will encrypt those labels under the public key of a fully homomorphic scheme. A new public key is generated for every input in order to prevent information from one execution from being useful for later executions. The worker

can then use the homomorphic property to compute an encryption of the output labels and provide them to the client, who decrypts them and reconstructs $F(x)$.

Since we use the fully-homomorphic encryption scheme in a black-box fashion, we anticipate that any performance improvements in future schemes will directly result in similar performance gains for our protocol as well.

One Pre-Processing Step for Many Workers. Note that the pre-processing stage is independent of the worker, since it simply produces a Yao-garbled version of the circuit C. Therefore, in addition to being reused many times, this garbled circuit can also be sent to many different workers, which is the usage scenario for applications like Folding@Home [153], which employ a multitude of workers across the Internet.

How to Handle Malicious Workers. In our scheme, if we assume that the worker learns whether or not the client accepts the proof π_x, then for every execution, a malicious worker potentially learns a bit of information about the labels of the Yao-garbled circuit. For example, the worker could try to guess one of the labels, encrypt it with the homomorphic encryption and see if the client accepts. In a sense, the output of the client at the end of the execution can be seen as a very restricted "decryption oracle" for the homomorphic encryption scheme (which is, by definition, not CCA secure). Because of this one-bit leakage, we are unable to prove security in this case.

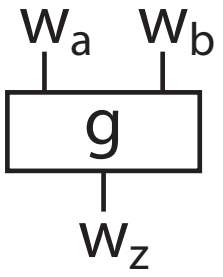
There are two ways to deal with this. One is to assume that the verification output bit by the client remains private. The other is to repeat the pre-processing stage, i.e. the Yao garbling of the circuit, every time a verification fails. In this case, in order to preserve a good amortized complexity, we must assume that failures do not happen very often. This is indeed the case in the previous scenario, where the same garbled circuit is used with several workers, under the assumption that only a small fraction of workers will be malicious. Details appear in Section 6.5.

6.2 Cryptographic Background

6.2.1 Yao's Garbled Circuit Construction

We summarize Yao's protocol for two-party private computation [216, 217]. For more details, we refer the interested reader to Lindell and Pinkas' excellent description [124].

We assume two parties, Alice and Bob, wish to compute a function F over their private inputs a and b. For simplicity, we focus on polynomial-time deterministic functions, but the generalization to stochastic functions is straightforward.



(a)

$\mathbf{w_a}$	$\mathbf{w_b}$	$\mathbf{w_z}$
0	0	g(0,0)
0	1	g(0,1)
1	0	g(1,0)
1	1	g(1,1)

(b)

$\mathbf{w_a}$	$\mathbf{w_b}$	$\mathbf{w_z}$
k_a^0	k_b^0	$k_z^{g(0,0)}$
k_a^0	k_b^1	$k_z^{g(0,1)}$
k_a^1	k_b^0	$k_z^{g(1,0)}$
k_a^1	k_b^1	$k_z^{g(1,1)}$

(c)

$\mathbf{w_a}$	$\mathbf{w_b}$	$\mathbf{w_z}$
k_a^0	k_b^0	$E_{k_a^0}(E_{k_b^0}(k_z^{g(0,0)}))$
k_a^0	k_b^1	$E_{k_a^0}(E_{k_b^1}(k_z^{g(0,1)}))$
k_a^1	k_b^0	$E_{k_a^1}(E_{k_b^0}(k_z^{g(1,0)}))$
k_a^1	k_b^1	$E_{k_a^1}(E_{k_b^1}(k_z^{g(1,1)}))$

(d)

Figure 6.1: **Yao's Garbled Circuits.** *The original binary gate* **(a)** *can be represented by a standard truth table* **(b)**. *We then replace the 0 and 1 values with the corresponding randomly chosen λ-bit values* **(c)**. *Finally, we use the values for w_a and w_b to encrypt the values for the output wire w_z* **(d)**. *The random permutation of these ciphertexts is the garbled representation of gate g.*

At a high-level, Alice converts F into a boolean circuit C. She prepares a garbled version of the circuit, $G(C)$, and sends it to Bob, along with a garbled version, $G(a)$, of her input. Alice and Bob then engage in a series of oblivious transfers so that Bob obtains $G(b)$ without Alice learning anything about b. Bob then applies the garbled circuit to the two garbled outputs to derive a garbled version of the output: $G(F(a, b))$. Alice can then translate this into the actual output and share the result with Bob. Note that this protocol assumes an honest-but-curious adversary model.

In more detail, Alice constructs the garbled version of the circuit as follows. For each wire w in the circuit, Alice chooses two random values $k_w^0, k_w^1 \xleftarrow{R} \{0,1\}^\lambda$ to represent the bit values of 0 or 1 on that wire. Once she has chosen wire values for every wire in the circuit, Alice constructs a garbled version of each gate g (see Figure 6.1). Let g be a gate with input wires w_a and w_b, and output wire w_z. We define the garbled version $G(g)$ of g to be the

following four ciphertexts:

$$\gamma_{00} = E_{k_a^0}(E_{k_b^0}(k_z^{g(0,0)})) \quad (6.1)$$

$$\gamma_{01} = E_{k_a^0}(E_{k_b^1}(k_z^{g(0,1)})) \quad (6.2)$$

$$\gamma_{10} = E_{k_a^1}(E_{k_b^0}(k_z^{g(1,0)})) \quad (6.3)$$

$$\gamma_{11} = E_{k_a^1}(E_{k_b^1}(k_z^{g(1,1)})), \quad (6.4)$$

where E is an secure symmetric encryption scheme with an "elusive range" (more details below). The order of the ciphertexts is randomly permuted to hide the structure of the circuit (i.e., we shuffle the ciphertexts, so that the first ciphertext does not necessarily encode the output for $(0, 0)$).

We refer to w_z^0 and w_z^1 as the "acceptable" outputs for gate g, since they are the only two values that represent valid bit-values for the output wire. Given input keys k_a^x, k_b^y, we will refer to $w_z^{g(x,y)}$ as the "legitimate" output, and $w_z^{1-g(x,y)}$ as the "illegitimate" output.

In Yao's protocol, Alice transfers all of the ciphertexts to Bob, along with the wire values corresponding to the bit-level representation of her input. In other words, she transfers either k_a^0 if her input bit is 0 or k_a^1 if her input bit is 1. Since these are randomly chosen values, Bob learns nothing about Alice's input. Alice and Bob then engage in an oblivious transfer so that Bob can obtain the wire values corresponding to his inputs (e.g., k_b^0 or k_b^1). Bob learns exactly one value for each wire, and Alice learns nothing about his input. Bob can then use the wire values to recursively decrypt the gate ciphertexts, until he arrives at the final output wire values. When he transmits these to Alice, she can map them back to 0 or 1 values and hence obtain the result of the function computation.

6.2.2 The Security of Yao's Protocol

Lindell and Pinkas prove [124] that Yao is a secure two-party computation protocol under some specific assumptions on the encryption scheme E used to garble the circuit. More specifically the encryption function E needs:

- *Indistinguishable ciphertexts for multiple messages:* For every two vectors of messages $\bar{x}$ and $\bar{y}$, no polynomial time adversary can distinguish between an encryption of $\bar{x}$ and an encryption of $\bar{y}$. Notice that because we require security for multiple messages, we cannot use a one-time pad.
- *An elusive range:* Encryptions under different keys fall into different ranges of the ciphertext space (at least with high probability).

- *An efficiently verifiable range:* Given the key k, it is possible to decide efficiently if a given ciphertext falls into the range of encryptions under k.

We give a formal definition of these properties. Recall that a private encryption scheme is a pair of algorithms (E, D), the encryption and decryption algorithms respectively, that run on input the security parameter λ, a random λ-bit key k, and λ-bit strings (the plaintext and ciphertext, respectively). We use $\texttt{negli}()$ to denote a negligible function of its input.

Definition 1 *We say that a private encryption scheme (E, D) is* Yao-secure *if the following properties are satisfied. Assume $k \leftarrow \{0,1\}^\lambda$:*

- Indistinguishability of ciphertexts for multiple messages: *For every efficient adversary A, and every two vectors of ciphertexts $[x_1, ..., x_\ell]$ and $[y_1, ..., y_\ell]$ (with $\ell = poly(\lambda)$), and $u_i = E_k(x_i)$, $z_i = E_k(y_i)$, we have that*

$$|Prob[A[u_1, \ldots, u_\ell] = 1] - Prob[A[z_1, \ldots, z_\ell] = 1]| < \texttt{negli}(\lambda)$$

- Elusive Range: *Let* $\mathsf{Range}_\lambda(k) = \{E_k(x)\}_{x \in \{0,1\}^\lambda}$. *For every efficient adversary A we require:*

$$Prob[A(1^\lambda) \in \mathsf{Range}_\lambda(k)] < \texttt{negli}(\lambda)$$

- Efficiently Verifiable Range: *There exists an efficient machine M such that $M(k, c) = 1$ if and only if $c \in \mathsf{Range}_\lambda(k)$.*

Lindell and Pinkas show [124] that Yao's garbled circuit technique, combined with a secure oblivious transfer protocol, is a secure two-party computation protocol (for semi-honest parties) if E is Yao-secure. They also show how to build Yao-secure encryption schemes based on one-way functions.

6.2.3 Fully Homomorphic Encryption

A fully-homomorphic encryption scheme $\mathcal{E}$ is defined by four algorithms: the standard encryption functions $\mathbf{KeyGen}_\mathcal{E}$, $\mathbf{Encrypt}_\mathcal{E}$, and $\mathbf{Decrypt}_\mathcal{E}$, as well as a fourth function $\mathbf{Evaluate}_\mathcal{E}$. $\mathbf{Evaluate}_\mathcal{E}$ takes in a circuit C and a tuple of ciphertexts and outputs a ciphertext that decrypts to the result of applying C to the plaintexts. A nontrivial scheme requires that $\mathbf{Encrypt}_\mathcal{E}$and $\mathbf{Decrypt}_\mathcal{E}$ operate in time independent of C [68, 69]. More precisely, the time needed to generate a ciphertext for an input wire of C, or decrypt a ciphertext for an output wire, is polynomial in the security parameter of the scheme (independent of C). Note that this implies that the length of the ciphertexts for the output wires is bounded by some polynomial in the security parameter (independent of C).

Gentry recently proposed a scheme, based on ideal lattices, that satisfies these requirements for arbitrary circuits [68, 69]. The complexity of $\mathbf{KeyGen}_{\mathcal{E}}$ in his initial *leveled* fully homomorphic encryption scheme grows linearly with the depth of C. However, under the assumption that his encryption scheme is *circular secure* – i.e., roughly, that it is "safe" to reveal an encryption of a secret key under its associated public key – the complexity of $\mathbf{KeyGen}_{\mathcal{E}}$ is independent of C. See [23, 68, 69] for more discussion on circular-security (and, more generally, key-dependent-message security) as it relates to fully homomorphic encryption.

In this paper, we use fully homomorphic encryption as a black box, and therefore do not discuss the details of any specific scheme.

6.3 Problem Definition

At a high-level, a verifiable computation scheme is a two-party protocol in which a *client* chooses a function and then provides an encoding of the function and inputs to the function to a *worker*. The worker is expected to evaluate the function on the input and respond with the output. The client then verifies that the output provided by the worker is indeed the output of the function computed on the input provided.

6.3.1 Basic Requirements

A *verifiable computation scheme* $\mathcal{VC} = (\mathbf{KeyGen}, \mathbf{ProbGen}, \mathbf{Compute}, \mathbf{Verify})$ consists of the four algorithms defined below.

1. $\mathbf{KeyGen}(F, \lambda) \rightarrow (PK, SK)$: Based on the security parameter λ, the randomized *key generation* algorithm generates a public key that encodes the target function F, which is used by the worker to compute F. It also computes a matching secret key, which is kept private by the client.
2. $\mathbf{ProbGen}_{SK}(x) \rightarrow (\sigma_x, \tau_x)$: The *problem generation* algorithm uses the secret key SK to encode the function input x as a public value σ_x which is given to the worker to compute with, and a secret value τ_x which is kept private by the client.
3. $\mathbf{Compute}_{PK}(\sigma_x) \rightarrow \sigma_y$: Using the client's public key and the encoded input, the worker *computes* an encoded version of the function's output $y = F(x)$.
4. $\mathbf{Verify}_{SK}(\tau_x, \sigma_y) \rightarrow y \cup \perp$: Using the secret key SK and the secret "decoding" τ_x, the *verification* algorithm converts the worker's encoded output into the output of the

function, e.g., $y = F(x)$ or outputs $\perp$ indicating that σ_y does not represent the valid output of F on x.

A verifiable computation scheme should be both correct and secure. A scheme is correct if the problem generation algorithm produces values that allows an honest worker to compute values that will verify successfully and correspond to the evaluation of F on those inputs. More formally:

Definition 2 (Correctness) *A verifiable computation scheme $\mathcal{VC}$ is* correct *if for any choice of function F, the key generation algorithm produces a keypair $(PK, SK) \leftarrow \mathbf{KeyGen}(F, \lambda)$ such that, $\forall x \in \texttt{Domain}(F)$, if $(\sigma_x, \tau_x) \leftarrow \mathbf{ProbGen}_{SK}(x)$ and $\sigma_y \leftarrow \mathbf{Compute}_{PK}(\sigma_x)$ then $y = F(x) \leftarrow \mathbf{Verify}_{SK}(\tau_x, \sigma_y)$.*

Intuitively, a verifiable computation scheme is secure if a malicious worker cannot persuade the verification algorithm to accept an incorrect output. In other words, for a given function F and input x, a malicious worker should not be able to convince the verification algorithm to output $\hat{y}$ such that $F(x) \neq \hat{y}$. Below, we formalize this intuition with an experiment, where $poly(\cdot)$ is a polynomial.

Experiment $\mathbf{Exp}_A^{Verif}[\mathcal{VC}, F, \lambda]$
  $(PK, SK) \xleftarrow{R} \mathbf{KeyGen}(F, \lambda)$;
  For $i = 1, \ldots, \ell = poly(\lambda)$;
    $x_i \leftarrow A(PK, x_1, \sigma_1, \ldots, x_i, \sigma_i)$;
    $(\sigma_i, \tau_i) \leftarrow \mathbf{ProbGen}_{SK}(x_i)$;
  $(i, \hat{\sigma}_y) \leftarrow A(PK, x_1, \sigma_1, \ldots, x_\ell, \sigma_\ell)$;
  $\hat{y} \leftarrow \mathbf{Verify}_{SK}(\tau_i, \hat{\sigma}_y)$
  If $\hat{y} \neq \perp$ and $\hat{y} \neq F(x_i)$, output '1', else '0';

Essentially, the adversary is given oracle access to generate the encoding of multiple problem instances. The adversary succeeds if it produces an output that convinces the verification algorithm to accept on the wrong output value for a given input value. We can now define the security of the system based on the adversary's success in the above experiment.

Definition 3 (Security) *For a verifiable computation scheme $\mathcal{VC}$, we define the advantage of an adversary A in the experiment above as:*

$$Adv_A^{Verif}(\mathcal{VC}, F, \lambda) = Prob[\mathbf{Exp}_A^{Verif}[\mathcal{VC}, F, \lambda] = 1] \tag{6.5}$$

A verifiable computation scheme $\mathcal{VC}$ is secure *for a function F, if for any adversary A*

running in probabilistic polynomial time,

$$Adv_A^{Verif}(\mathcal{VC}, F, \lambda) \leq \mathtt{negli}(\lambda) \tag{6.6}$$

where $\mathtt{negli}()$ *is a negligible function of its input.*

In the above definition, we could have also allowed the adversary to select the function F. However, our protocol is a verifiable computation scheme that is secure for *all* F, so the above definition suffices.

6.3.2 Input and Output Privacy

While the basic definition of a verifiable computation protects the integrity of the computation, it is also desirable that the scheme protect the secrecy of the input given to the worker(s). We define input privacy based on a typical indistinguishability argument that guarantees that *no* information about the inputs is leaked. Input privacy, of course, immediately yields output privacy.

Intuitively, a verifiable computation scheme is *private* when the public outputs of the problem generation algorithm $\mathbf{ProbGen}$ over two different inputs are indistinguishable; i.e., nobody can decide which encoding is the correct one for a given input. More formally consider the following experiment: the adversary is given the public key for the scheme and selects two inputs x_0, x_1. He is then given the encoding of a randomly selected one of the two inputs and must guess which one was encoded. During this process the adversary is allowed to request the encoding of any input he desires. The experiment is described below. The oracle $\mathbf{PubProbGen}_{SK}(x)$ calls $\mathbf{ProbGen}_{SK}(x)$ to obtain (σ_x, τ_x) and returns only the public part σ_x.

Experiment $\mathbf{Exp}_A^{Priv}[\mathcal{VC}, F, \lambda]$
$(PK, SK) \xleftarrow{R} \mathbf{KeyGen}(F, \lambda)$;
$(x_0, x_1) \leftarrow A^{\mathbf{PubProbGen}_{SK}(\cdot)}(PK)$
$(\sigma_0, \tau_0) \leftarrow \mathbf{ProbGen}_{SK}(x_0)$;
$(\sigma_1, \tau_1) \leftarrow \mathbf{ProbGen}_{SK}(x_1)$;
$b \xleftarrow{R} \{0, 1\}$;
$\hat{b} \leftarrow A^{\mathbf{PubProbGen}_{SK}(\cdot)}(PK, x_0, x_1, \sigma_b)$
If $\hat{b} = b$, output '1', else '0';

Definition 4 (Privacy) *For a verifiable computation scheme $\mathcal{VC}$, we define the advantage of an adversary A in the experiment above as:*

$$Adv_A^{Priv}(\mathcal{VC}, F, \lambda) = Prob[\mathbf{Exp}_A^{Priv}[\mathcal{VC}, F, \lambda] = 1] \tag{6.7}$$

A verifiable computation scheme $\mathcal{VC}$ is private *for a function F, if for any adversary A running in probabilistic polynomial time,*

$$Adv_A^{Priv}(\mathcal{VC}, F, \lambda) \leq \mathtt{negli}(\lambda) \tag{6.8}$$

where $\mathtt{negli}()$ is a negligible function of its input.

An immediate consequence of the above definition is that in a private scheme, the encoding of the input must be probabilistic (since the adversary can always query x_0, x_1 to the $\mathbf{PubProbGen}$ oracle, and if the answer were deterministic, he could decide which input is encoded in σ_b).

A similar definition can be made for output privacy.

6.3.3 Efficiency

The final condition we require from a verifiable computation scheme is that the time to encode the input and verify the output must be smaller than the time to compute the function from scratch.

Definition 5 (Outsourceable) *A $\mathcal{VC}$ can be outsourced if it permits efficient generation and efficient verification. This implies that for any x and any σ_y, the time required for $\mathbf{ProbGen}_{SK}(x)$ plus the time required for $\mathbf{Verify}(\sigma_y)$ is $o(T)$, where T is the time required to compute $F(x)$.*

Some functions are naturally outsourceable (i.e., they can be outsourced with no additional mechanisms), but many are not. For example, it is cheaper to verify the result of sorting a list of numbers than to perform the sort itself. Similarly, it is cheaper to verify a factorization than to find a factorization. However, a function that asks whether the factors of a number fall in a particular range is not naturally outsourceable.

Notice that we are not including the time to compute the key generation algorithm (i.e., the encoding of the function itself). Therefore, the above definition captures the idea of an outsourceable verifiable computation scheme which is more efficient than computing the function in an *amortized* sense, since the cost of encoding the function can be amortized over many input computations.

6.4 An Efficient Verifiable-Computation Scheme with Input and Output Privacy

6.4.1 Protocol Definition

We are now ready to describe our scheme. Informally, our protocol works as follows. The key generation algorithm consists of running Yao's garbling procedure over a Boolean circuit computing the function F: the public key is the collection of ciphertexts representing the garbled circuit, and the secret key consists of all the random wire labels. The input is encoded in two steps: first a fresh public/secret key pair for a homomorphic encryption scheme is generated, and then the labels of the correct input wires are encrypted with it. These ciphertexts constitute the public encoding of the input, while the secret key is kept private by the client. Using the homomorphic properties of the encryption scheme, the worker performs the computation steps of Yao's protocol, but working over ciphertexts (i.e., for every gate, given the encrypted labels for the correct input wires, obtain an encryption of the correct output wire, by applying the homomorphic encryption over the circuit that computes the "double decryption" in Yao's protocol). At the end, the worker will hold the encryption of the labels of the correct output wires. He returns these ciphertexts to the client who decrypts them and then computes the output from them. We give a detailed description below.

Protocol $\mathcal{VC}$.

1. $\mathbf{KeyGen}(F, \lambda) \rightarrow (PK, SK)$: Represent F as a circuit C. Following Yao's Circuit Construction (see Section 6.2.1), choose two values, $w_i^0, w_i^1 \xleftarrow{R} \{0,1\}^\lambda$ for each wire w_i. For each gate g, compute the four "garbled" ciphertexts $(\gamma_{00}^g, \gamma_{01}^g, \gamma_{10}^g, \gamma_{11}^g)$ described in Equations 6.1-6.4. The public key PK will be the full set of ciphertexts, i.e., $PK \leftarrow \cup_g(\gamma_{00}^g, \gamma_{01}^g, \gamma_{10}^g, \gamma_{11}^g)$, while the secret key will be the wire values chosen: $SK \leftarrow \cup_i(w_i^0, w_i^1)$.
2. $\mathbf{ProbGen}_{SK}(x) \rightarrow \sigma_x$: Run the doubly-homomorphic encryption scheme's key generation algorithm to create a new keypair: $(PK_\mathcal{E}, SK_\mathcal{E}) \leftarrow \mathbf{KeyGen}_\mathcal{E}(\lambda)$. Let $w_i \subset SK$ be the wire values representing the binary expression of x. Set $\sigma_x \leftarrow (PK_\varepsilon, \mathbf{Encrypt}_\mathcal{E}(PK_\mathcal{E}, w_i))$ and $\tau_x \leftarrow SK_\mathcal{E}$.
3. $\mathbf{Compute}_{PK}(\sigma_x) \rightarrow \sigma_y$: Calculate $\mathbf{Encrypt}_\mathcal{E}(PK_\mathcal{E}, \gamma_i)$. Construct a circuit Δ that on input w, w', γ outputs $D_w(D_{w'}(\gamma))$, where D is the decryption algorithm corresponding to the encryption E used in Yao's garbling (thus, Δ computes the appropriate decryption in Yao's construction). Calculate $\mathbf{Evaluate}_\mathcal{E}(\Delta, \mathbf{Encrypt}_\mathcal{E}(PK_\mathcal{E}, w_i), \mathbf{Encrypt}_\mathcal{E}(PK_\mathcal{E}, \gamma_i))$ repeatedly, to decrypt your way through the ciphertexts, just

as in the evaluation of Yao's garbled circuit. The result is $\sigma_y \leftarrow \mathbf{Encrypt}_{\mathcal{E}}(PK_{\varepsilon}, \bar{w}_i)$, where $\bar{w}_i$ are the wire values representing $y = F(x)$ in binary.

4. $\mathbf{Verify}_{SK}(\sigma_y) \rightarrow y \cup \perp$: Use $SK_{\mathcal{E}}$ to decrypt $\mathbf{Encrypt}_{\mathcal{E}}(PK_{\varepsilon}, \bar{w}_i)$, obtaining $\bar{w}_i$. Use SK to map the wire values to an output y. If the decryption or mapping fails, output $\perp$.

Remark: *On verifying ciphertext ranges in an encrypted form.* Recall that Yao's scheme requires the encryption scheme E to have an *efficiently verifiable range*: Given the key k, it is possible to decide efficiently if a given ciphertext falls into the range of encryptions under k. In other words, there exists an efficient machine M such that $M(k, \gamma) = 1$ if and only if $\gamma \in \mathsf{Range}_{\lambda}(k)$. This is necessary to "recognize" which ciphertext to pick among the four ciphertexts associated with each gate.

In our verifiable computation scheme $\mathcal{VC}$, we need to perform this check using an encrypted form of the key $c = \mathbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, k)$. When applying the homomorphic properties of $\mathcal{E}$ to the range testing machine M, the worker obtains an encryption of 1 for the correct ciphertext, and an encryption of 0 for the others. Of course he is not able to distinguish which one is the correct one.

The worker then proceeds as follows: for the four ciphertexts $\gamma_1, \gamma_2, \gamma_3, \gamma_4$ associated with a gate g, he first computes $c_i = \mathbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, M(k, \gamma_i))$ using the homomorphic properties of $\mathcal{E}$ over the circuit describing M. Note that only one of these ciphertexts encrypts a 1, exactly the one corresponding to the correct γ_i. Then the worker computes $d_i = \mathbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, D_k(\gamma_i))$ using the homomorphic properties of $\mathcal{E}$ over the decryption circuit Δ. Note that $k' = \Sigma_i M(k, \gamma_i) D_k(\gamma_i)$ is the correct label for the output wire. Therefore, the worker can use the homomorphic properties of $\mathcal{E}$ to compute

$$c = \mathbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, k') = \mathbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, \Sigma_i M(k, \gamma_i) D_k(\gamma_i))$$

from c_i, d_i, as desired.

6.4.2 Proof of Security

The main result of our paper is the following.

Theorem 1 *Let E be a Yao-secure symmetric encryption scheme and $\mathcal{E}$ be a semantically secure homomorphic encryption scheme. Then protocol $\mathcal{VC}$ is a secure, outsourceable and private verifiable computation scheme.*

The proof of Theorem 1 requires two high-level steps. First, we show that Yao's garbled circuit scheme is a one-time secure verifiable computation scheme, i.e. a scheme that can be used to compute F securely on one input. Then, by using the semantic security of the homomorphic encryption scheme, we reduce the security of our scheme (with multiple executions) to the security of a single execution where we expect the adversary to cheat.

6.4.2.1 Proof Sketch of Yao's Security for One Execution

Consider the verifiable computation scheme $\mathcal{VC}_{Yao}$ defined as follows:

Protocol $\mathcal{VC}_{Yao}$.

1. $\mathbf{KeyGen}(F, \lambda) \rightarrow (PK, SK)$: Represent F as a circuit C. Following Yao's Circuit Construction (see Section 6.2.1), choose two values, $w_i^0, w_i^1 \xleftarrow{R} \{0,1\}^\lambda$ for each wire w_i. For each gate g, compute the four ciphertexts $(\gamma_{00}^g, \gamma_{01}^g, \gamma_{10}^g, \gamma_{11}^g)$ described in Equations 6.1-6.4. The public key PK will be the full set of ciphertexts, i.e, $PK \leftarrow \cup_g(\gamma_{00}^g, \gamma_{01}^g, \gamma_{10}^g, \gamma_{11}^g)$, while the secret key will be the wire values chosen: $SK \leftarrow \cup_i(w_i^0, w_i^1)$.
2. $\mathbf{ProbGen}_{SK}(x) \rightarrow \sigma_x$: Reveal the labels of the input wires associated with x. In other words, let $w_i \subset SK$ be the wire values representing the binary expression of x, and set $\sigma_x \leftarrow (PK_\varepsilon, w_i)$. τ_x is the empty string.
3. $\mathbf{Compute}_{PK}(\sigma_x) \rightarrow \sigma_y$: Compute the decryptions in Yao's protocol to obtain the labels of the correct output wires. Set σ_y to be these labels.
4. $\mathbf{Verify}_{SK}(\sigma_y) \rightarrow y \cup \perp$: Use SK to map the wire values in σ_y to the binary representation of the output y. If the mapping fails, output $\perp$.

Theorem 2 *$\mathcal{VC}_{Yao}$ is a correct verifiable computation scheme.*

Proof of Theorem 2: The proof of correctness follows directly from the proof of correctness for Yao's garbled circuit construction [124]. Using C and $\tilde{x}$ will produce a $\tilde{y}$ that represents the correct evaluation of $F(x)$. ■

We prove that $\mathcal{VC}_{Yao}$ is a *one-time* secure verifiable computation scheme. The definition of *one-time secure* is the same as Definition 3 except that in experiment $\mathbf{Exp}_A^{Verif}$, the adversary is allowed to query the oracle $\mathbf{ProbGen}_{SK}(\cdot)$ only once (i.e., $\ell = 1$) and must cheat on that input.

Intuitively, an adversary who violates the security of this scheme must either guess the "incorrect" random value $k_w^{1-y_i}$ for one of the output bit values representing y, or he must break the encryption scheme used to encode the "incorrect" wire values in the circuit. The

former happens with probability $\leq \frac{1}{2^\lambda}$, i.e., negligible in λ. The latter violates our security assumptions about the encryption scheme. We formalize this intuition below using an hybrid argument similar to the one used in [124].

Theorem 3 *Let E be a Yao-secure symmetric encryption scheme. Then $\mathcal{VC}_{Yao}$ is a one-time secure verifiable computation scheme.*

Proof of Theorem 3: Assume w.l.o.g. that the function F outputs a single bit (at the end of the proof we show how to deal with the case of multiple-bit outputs). Assume a canonical order on the gates in the circuit computing F, and let m be the number of such gates. Let PK be the garbled circuit obtained by running $\mathbf{KeyGen}(F, \lambda)$.

Fix any adversary A; we show that for A, the probability of successfully cheating is negligible in λ, if the encryption scheme E is Yao-secure. We do this by defining a series of *hybrid* experiments where we change the setting in which A is run, but in a controlled way: each experiment in the series will be *computationally indistinguishable* from the previous one, if the security of the encryption scheme holds. The first experiment in the series is $\mathbf{Exp}_A^{Verif}$. In the last experiment, we will show that information-theoretically A can cheat only with negligible probability, therefore proving that in order to cheat in the original experiment, A must distinguish between two experiments in the series, and thus break the encryption scheme.

We denote with $H_A^i[\mathcal{VC}, F, \lambda]$, the i^{th} hybrid experiment, run with an adversary A, verifiable computation scheme $\mathcal{VC}$, function F and security parameter λ. All experiments output a Boolean value, and therefore we can define $Adv_A^i(\mathcal{VC}, F, \lambda) = Prob[H_A^i[\mathcal{VC}, F, \lambda] = 1]$.

Define

$$p_b = Prob[A \text{ in } \mathbf{Exp}_A^{Verif}[\mathcal{VC}_{Yao}, F, \lambda] \text{ outputs } x \text{ s.t. } F(x) = b]$$

Note that we can estimate these probabilities by running the experiment many times. Set β to be the bit such that $p_\beta \geq p_{\bar{\beta}}$. Notice that $p_\beta \geq 1/2$.

Experiment $H_A^0[\mathcal{VC}_{Yao}, F, \lambda]$: This experiment is exactly like $\mathbf{Exp}_A^{Verif}[\mathcal{VC}_{Yao}, F, \lambda]$ except that when A queries $\mathbf{ProbGen}$ on the input x (recall that we are considering the case where the adversary only submits a single input value and must cheat on that input), the oracle selects a random[1] x' such that $F(x') = \beta$ and returns $\sigma_{x'}$, where $(\sigma_{x'}, \tau_{x'}) \leftarrow \mathbf{ProbGen}_{SK}(x')$. The experiment's output bit is set to 1 if A manages to cheat over input x', i.e. produces a valid proof for $\bar{\beta}$ (and to 0 otherwise).

[1] Since F is a Boolean function, w.l.o.g. we can assume that we can efficiently sample x' such that $F(x') = b$.

Lemma 1 *If E is a Yao-secure encryption scheme, then for all efficient adversaries A we have* $|Adv_A^0(\mathcal{VC}_{Yao}, F, \lambda) - Adv_A^{Verif}(\mathcal{VC}_{Yao}, F, \lambda)| \leq \mathtt{negli}(\lambda)$.

Proof of Lemma 1: The Lemma follows from the security of Yao's two-party computation protocol [124].

Recall that in Yao's protocol, two parties P_1 and P_2 want to compute a function F over inputs x and y privately held respectively by P_1 and P_2, without revealing any information about their inputs except the value $F(x, y)$. The protocol goes as follows: P_1 garbles a circuit computing the function F, and gives to P_2 the labels of his input x. Moreover, P_1 and P_2 engage in OT protocols to give P_2 the labels of her input y, without revealing this input to P_1. Then P_2 executes the circuit on his own and sends the output label to P_1, who reveals the output of the function $F(x, y)$. Note that P_1 sends his input labels in the clear to P_2. The intuition is that P_1's input remains private since P_2 can't associate the labels with the bit values they represent. This intuition is formalized in the proof in [124].

Therefore, we reduce the indistinguishability of the initial hybrid experiment $H_A^0[\mathcal{VC}_{Yao}, F, \lambda]$ and $\mathbf{Exp}_{A^*}^{Verif}[\mathcal{VC}_{Yao}, F, \lambda]$ to the security of Yao's protocol. In other words, we show that if there exists A such that

$$|Adv_A^0(\mathcal{VC}_{Yao}, F, \lambda) - Adv_A^{Verif}(\mathcal{VC}_{Yao}, F, \lambda)| > \epsilon$$

with non-negligible ϵ, then we can learn some information about P_1's input with roughly the same advantage.

Suppose we run Yao's two-party protocol between P_1 and P_2 with the function F computed over just P_1's input x'. We assume that P_1's input is chosen with the right distribution[2] (i.e. $F(x') = \beta$). For any two values x, x', with $F(x) = F(x')$, the security of Yao's protocol implies that no efficient player P_2 can distinguish if x or x' was used.

We build a simulator S that plays the role of P_2 and distinguishes between the two input cases, with probability $p_\beta\epsilon$, thus creating a contradiction.

The protocol starts with P_1 sending the garbled circuit PK and the encoding of his input $\sigma_{x'}$. The simulator computes the label ℓ associated with the output $F(x')$. At this point the simulator engages A over the input PK, and A requests the encoding of an input x. If $F(x) \neq \beta$ the simulator tosses a random coin, and outputs the resulting bit. Notice however that with probability p_β, $F(x) = \beta = F(x')$. In this case, the simulator provides A with the encoding $\sigma_{x'}$, and returns as its output the experiment bit.

[2] We can assume this since the security of Yao's protocol is for all inputs, so in particular for this distribution.

Notice that if $x = x'$ we are running $\mathbf{Exp}_A^{Verif}[\mathcal{VC}_{Yao}, F, \lambda]$, while if $x \neq x'$ we are running $H_{A^*}^0[\mathcal{VC}_{Yao}, F, \lambda]$. Therefore the simulator distinguishes between the two input values exactly with probability $p_\beta \epsilon$, therefore creating a contradiction. ■

Experiment $H_A^i[\mathcal{VC}_{Yao}, F, \lambda]$ **for** $i = 1, \ldots, m$**:** During the i^{th} experiment the $\mathbf{ProbGen}$ oracle still chooses a random value x' to answer A's query as in $H_A^0[\mathcal{VC}_{Yao}, F, \lambda]$. This value x' defines $0/1$ values for all the wires in the circuit. We say that a label w^b for wire w is *active* if the value of wire w when the circuit is computed over x' is b. We now define a family of fake garbled circuits PK_{fake}^i for $i = 0, \ldots, m$, as follows. For gate g_j with $j \leq i$, if w^b is the active label associated with its output wire w, then *all* four ciphertexts associated with g_j encrypt w^b. For gate g_j, with $j > i$, the four ciphertexts are computed correctly as in Yao's garbling technique, where the value encrypted depends on the keys used to encrypt it. Notice that $PK_{fake}^0 = PK$ since for all of the gates, the ciphertexts are computed correctly. The experiment's output bit is still set to 1 if A manages to cheat over input x', i.e. produces a valid proof for $\bar{\beta}$ (and to 0 otherwise).

Lemma 2 *If E is a Yao-secure encryption scheme, then for all efficient adversaries A we have* $|Adv_A^i(\mathcal{VC}_{Yao}, F, \lambda) - Adv_A^{i-1}(\mathcal{VC}_{Yao}, F, \lambda)| \leq \mathtt{negli}(\lambda)$.

This lemma is proven in [124], and we refer the reader to it for a full proof. Intuitively, the lemma follows from the ciphertext indistinguishability of the encryption scheme E.

Lemma 3 $Adv_A^m(\mathcal{VC}_{Yao}, F, \lambda) = 2^{-\lambda}$

Proof of Lemma 3: Recall that $Adv_A^m(\mathcal{VC}_{Yao}, F, \lambda)$ is the probability that A manages to cheat over input x', i.e., to provide the incorrect output label. However, the view of A is information-theoretically independent of that label, since the incorrect output label is inactive and has not been encrypted in the garbled circuit PK_{fake}^m. Since labels are chosen as random λ-bit strings, the probability of guessing the incorrect output label is exactly $2^{-\lambda}$.

■

This completes the proof of Theorem 3. ■

Remark: This proof does not readily extend to the case of a function F with multiple output bits, because in that case it might not be possible to sample an x which produces a specific output (think of a one-way function F for example). However, notice that if the output is n bits, then the value y computed by a successful cheating adversary must be different from $F(x)$ in at least one bit. Thus, at the beginning of the simulation, we can try to guess the bit on which the adversary will cheat and then run the proof for the 1-bit case. Our guess will be right with probability $1/n$.

6.4.2.2 Proof of Theorem 1

The proof of Theorem 1 follows from Theorem 2 and the semantic security of the homomorphic encryption scheme. More precisely, we show that if the homomorphic encryption scheme is semantically secure, then we can transform (via a simulation) a successful adversary against the full verifiable computation scheme $\mathcal{VC}$ into an attacker for the one-time secure protocol $\mathcal{VC}_{Yao}$. The intuition is that for each query, the labels in the circuit are encrypted with a semantically-secure encryption scheme (the homomorphic scheme), so multiple queries do not help the adversary to learn about the labels, and hence if he cheats, he must be able to cheat in the one-time case as well.

Proof of Theorem 1: Let us assume for the sake of contradiction that there is an adversary A such that $Adv_A^{Verif}(\mathcal{VC}, F, \lambda) \geq \epsilon$, where ϵ is non-negligible in λ. We use A to build another adversary A' which queries the $\mathbf{ProbGen}$ oracle only once, and for which $Adv_{A'}^{Verif}(\mathcal{VC}_{Yao}, F, \lambda) \geq \epsilon'$, where ϵ' is close to ϵ. The details of A' follow.

A' receives as input the garbled circuit PK. It activates A with the same input. Let ℓ be an upper bound on the number of queries that A makes to its $\mathbf{ProbGen}$ oracle. The adversary A' chooses an index i at random between 1 and ℓ and continues as follows. For the j^{th} query by A, with $j \neq i$, A' will respond by (i) choosing a random private/public key pair for the homomorphic encryption scheme $(PK_{\mathcal{E}}^j, SK_{\mathcal{E}}^j)$ and (ii) encrypting random λ-bit strings under $PK_{\mathcal{E}}^j$. For the i^{th} query, x, the adversary A' gives x to its own $\mathbf{ProbGen}$ oracle and receives σ_x, the collection of active input labels corresponding to x. It then generates a random private/public key pair for the homomorphic encryption scheme $(PK_{\mathcal{E}}^i, SK_{\mathcal{E}}^i)$, and it encrypts σ_x (label by label) under $PK_{\mathcal{E}}^i$.

Once we prove the Lemma 4 below, we have our contradiction and the proof of Theorem 1 is complete ■

Lemma 4 *$Adv_{A'}^{Verif}(\mathcal{VC}_{Yao}, F, \lambda) \geq \epsilon'$ where ϵ' is non-negligible in λ.*

Proof of Lemma 4: This proof also proceeds by defining, for any adversary A, a set of hybrid experiments $\mathcal{H}_A^k(\mathcal{VC}, F, \lambda)$ for $k = 0, \ldots, \ell - 1$. We define the experiments below. Let i be an index randomly selected between 1 and ℓ as in the proof above.

Experiment $\mathcal{H}_A^k(\mathcal{VC}, F, \lambda) = 1$]: In this experiment, we change the way the oracle $\mathbf{ProbGen}$ computes its answers. For the j^{th} query:

- *$j \leq k$ and $j \neq i$:* The oracle will respond by (i) choosing a random private/public key pair for the homomorphic encryption scheme $(PK_{\mathcal{E}}^j, SK_{\mathcal{E}}^j)$ and (ii) encrypting random λ-bit strings under $PK_{\mathcal{E}}^j$.

- $j > k$ *or* $j = i$*:* The oracle will respond exactly as in $\mathcal{VC}$, i.e. by (i) choosing a random private/public key pair for the homomorphic encryption scheme $(PK_{\mathcal{E}}^{j}, SK_{\mathcal{E}}^{j})$ and (ii) encrypting the correct input labels in Yao's garbled circuit under $PK_{\mathcal{E}}^{j}$.

In the end, the bit output by the experiment $\mathcal{H}_A^k$ is 1 if A successfully cheats on the i^{th} input and otherwise is 0. We denote with $Adv_A^k(\mathcal{VC}, F, \lambda) = Prob[\mathcal{H}_A^k(\mathcal{VC}, F, \lambda) = 1]$. Note that

- $\mathcal{H}_A^0(\mathcal{VC}, F, \lambda)$ is identical to the experiment $\mathbf{Exp}_A^{Verif}[\mathcal{VC}, F, \lambda]$, except for the way the bit is computed at the end. Since the index i is selected at random between 1 and ℓ, we have that

$$Adv_A^0(\mathcal{VC}, F, \lambda) = \frac{Adv_A^{Verif}(\mathcal{VC}, F, \lambda)}{\ell} \geq \frac{\epsilon}{\ell}$$

- $\mathcal{H}_A^{\ell-1}(\mathcal{VC}, F, \lambda)$ is equal to the simulation conducted by A' above, so

$$Adv_A^{\ell-1}(\mathcal{VC}, F, \lambda) = Adv_{A'}^{Verif}(\mathcal{VC}_{Yao}, F, \lambda)$$

If we prove for $k = 0, \ldots, \ell - 1$ that experiments $\mathcal{H}_A^k(\mathcal{VC}, F, \lambda)$ and $\mathcal{H}_A^{k-1}(\mathcal{VC}, F, \lambda)$ are computationally indistinguishable, that is for every A

$$|Adv_A^k(\mathcal{VC}, F, \lambda) - Adv_A^{k-1}(\mathcal{VC}, F, \lambda)| \leq \mathtt{negli}(\lambda) \tag{6.9}$$

we are done, since that implies that

$$Adv_{A'}^{Verif}(\mathcal{VC}_{Yao}, F, \lambda) \geq \frac{\epsilon}{\ell} - \ell \cdot \mathtt{negli}(\lambda)$$

which is the desired non-negligible ϵ'.

But Eq. 6.9 easily follows from the semantic security of the homomorphic encryption scheme. Indeed assume that we could distinguish between $\mathcal{H}_A^k$ and $\mathcal{H}_A^{k-1}$, then we can decide the following problem, which is easily reducible to the semantic security of $\mathcal{E}$:

Security of $\mathcal{E}$ with respect to Yao Garbled Circuits: *Given a Yao-garbled circuit PK_{Yao}, an input x for it, a random public key $PK_{\mathcal{E}}$ for the homomorphic encryption scheme, a set of ciphertexts $c_1, \ldots, c_n$ where n is the size of x, decide if for all i, $c_i = \mathbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, w_i^{x_i})$, where w_i is the i^{th} input wire and x_i is the i^{th} input bit of x, or c_i is the encryption of a random value.*

Now run experiment $\mathcal{H}_A^{k-1}$ with the following modification: at the k^{th} query, instead of choosing a fresh random key for $\mathcal{E}$ and encrypting random labels, answer with $PK_{\mathcal{E}}$ and

the ciphertexts $c_1, \ldots, c_n$ defined by the problem above. If c_i is the encryption of a random value, then we are still running experiment $\mathcal{H}_A^{k-1}$, but if $c_i = \mathbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, w_i^{x_i})$, then we are actually running experiment $\mathcal{H}_A^k$. Therefore we can decide the Security of $\mathcal{E}$ with respect to Yao Garbled Circuits with the same advantage with which we can distinguish between $\mathcal{H}_A^k$ and $\mathcal{H}_A^{k-1}$.

The reduction of the Security of $\mathcal{E}$ with respect to Yao Garbled Circuits to the basic semantic security of $\mathcal{E}$ is an easy exercise, and details will appear in the final version. ■

6.4.3 Proof of Input and Output Privacy

Note that for each oracle query the input and the output are encrypted under the homomorphic encryption scheme $\mathcal{E}$. It is not hard to see that the proof of correctness above, easily implies the proof of input and output privacy. For the one-time case, it obviously follows from the security of Yao's two-party protocol. For the general case, it follows from the semantic security of $\mathcal{E}$, and the proof relies on the same hybrid arguments described above.

6.4.4 Efficiency

The protocol we have described meets the efficiency goals outlined in Section 6.3.3. During the preprocessing stage, the client performs $O(|C|)$ work to prepare the Garbled Yao circuit. For each invocation of $\mathbf{ProbGen}$, the client generates a new keypair and encrypts one Yao label for each bit of the input, which requires $O(n)$ effort. The worker computes its way through the circuit by performing a constant amount of work per gate, so the worker takes time linear in the time to evaluate the original circuit, namely $O(|C|)$. Finally, to verify the worker's response, the client performs a single decryption and comparison operation for each bit of the output, for a total effort of $O(m)$. Thus, amortized over many inputs, the client performs $O(n + m)$ work to prepare and verify each input and result.

6.5 How to Handle Cheating Workers

Our definition of security (Definition 3) assumes that the adversary does not see the output of the $\mathbf{Verify}$ procedure run by the client on the value σ returned by the adversary. Theorem 1 is proven under the same assumption. In practice this means that our protocol $\mathcal{VC}$ is secure if the client keeps the result of the computation private.

In practice, there might be circumstances where this is not feasible, as the behavior of the client will change depending on the result of the evaluation (e.g., the client might refuse

to pay the worker). Intuitively, and we prove this formally below, seeing the result of **Verify** on proofs the adversary correctly **Compute**s using the output of **PubProbGen** does not help the adversary (since it already knows the result based on the inputs it supplied to **PubProbGen**). But what if the worker returns a malformed response – i.e., something for which **Verify** outputs $\perp$. How does the client respond, if at all? One option is for the client to ask the worker to perform the computation again. But this repeated request informs the worker that its response was malformed, which is an additional bit of information that a cheating worker might exploit in its effort to generate forgeries. Is our scheme secure in this setting? In this section, we prove that our scheme remains secure as long as the client terminates after detecting a malformed response. We also consider the interesting question of whether our scheme is secure if the client terminates only after detecting $k > 1$ malformed responses, but we are unable to provide a proof of security in this setting.

Note that there is a real attack on the scheme in this setting if the client does not terminate. Specifically, for concreteness, suppose that each ciphertext output by $\mathbf{Encrypt}_{\mathcal{E}}$ encrypts a single bit of a label for an input wire of the garbled circuit, and that the adversary wants to determine the first bit $w_{11}^{b_1}$ of the first label (where that label stands in for unknown input $b_1 \in \{0,1\}$). To do this, the adversary runs **Compute** as before, obtaining ciphertexts that encrypt the bits $\bar{w}_i$ of a label for the output wire. Using the homomorphism of the encryption scheme $\mathcal{E}$, it XORs $w_{11}^{b_1}$ with the first bit of $\bar{w}_i$ to obtain $\bar{w}_i'$, and it sends (the encryption of) $\bar{w}_i'$ as its response. If **Verify** outputs $\perp$, then $w_{11}^{b_1}$ must have been a 1; otherwise, it is a 0 with overwhelming probability. The adversary can thereby learn the labels of the garbled circuit one bit at a time – in particular, it can learn the labels of the output wire, and thereafter generate a verifiable response without actually performing the computation.

Intuitively, one might think that if the client terminates after detecting k malformed responses, then the adversary should only be able to obtain about k bits of information about the garbled circuit before the client terminates (using standard entropy arguments), and therefore it should still be hard for the adversary to output the entire "wrong" label for the output wire as long as λ is sufficiently larger than k. However, we are unable to make this argument go through. In particular, the difficulty is with the hybrid argument in the proof of Theorem 1, where we gradually transition to an experiment in which the simulator is encrypting the same Yao input labels in every round. This experiment must be indistinguishable from the real world experiment, which permits different inputs in different rounds. When we don't give the adversary information about whether or not its response was well-formed or not, the hybrid argument is straightforward – it simply depends on the semantic security of the FHE scheme.

However, if we do give the adversary that information, then the adversary can distinguish rounds with the same input from rounds with random inputs. To do so, it chooses some "random" predicate P over the input labels, such that $P(w^1_{b_1}, w^2_{b_2}, \ldots) = P(w^1_{b'_1}, w^2_{b'_2}, \ldots)$ with probability $1/2$ if $(b_1, b_2, \ldots) \neq (b'_1, b'_2, \ldots)$. Given the encryptions of $w^1_{b_1}, w^2_{b_2}, \ldots$, the adversary runs **Compute** as in the scheme, obtaining ciphertexts that encrypt the bits $\bar{w}_i$ of a label for the output wire, XORs (using the homomorphism) $P(w^1_{b_1}, w^2_{b_2}, \ldots)$ with the first bit of $\bar{w}_i$, and sends (an encryption of) the result $\bar{w}'_i$ as its response. If the client is making the same query in every round – i.e., the Yao input labels are the same every time – then, the predicate always outputs the same bit, and thus the adversary gets the same response (well-formed or malformed) in every round. Otherwise, the responses will tend to vary.

One could try to make the adversary's distinguishing attack more difficult by (for example) trying to hide which ciphertexts encrypt the bits of which labels – i.e., via some form of obfuscation. However, the adversary may define its predicate in such a way that it "analyzes" this obfuscated circuit, determines whether two ostensibly different inputs in fact represent the same set of Yao input labels, and outputs the same bit if they do. (It performs this analysis on the encrypted inputs, using the homomorphism.) We do not know of any way to prevent this distinguishing attack, and suspect that preventing it may be rather difficult in light of Barak et al.'s result that there is no general obfuscator [22].

Security with Verification Access. We say that a verifiable computation scheme is secure *with verification access* if the adversary is allowed to see the result of **Verify** over the queries x_i he has made to the **ProbGen** oracle in $\mathbf{Exp}_A^{Verif}$ (see Definition 3).

Let $\mathcal{VC}^\dagger$ be like $\mathcal{VC}$, except that the client terminates if it receives a malformed response from the worker. Below, we show that $\mathcal{VC}^\dagger$ is secure with verification access. In other words, it is secure to provide the worker with verification access (indicating whether a response was well-formed or not), until the worker gives a malformed response. Let $\mathbf{Exp}_A^{Verif^\dagger}\left[\mathcal{VC}^\dagger, F, \lambda\right]$ denote the experiment described in Section 6.3.1, with the obvious modifications.

Theorem 4 *If $\mathcal{VC}$ is a secure outsourceable verifiable computation scheme, then $\mathcal{VC}^\dagger$ is a secure outsourceable verifiable computation scheme with verification access. If $\mathcal{VC}$ is private, then so is $\mathcal{VC}^\dagger$.*

Proof of Theorem 4: Consider two games between a challenger and an adversary A. In the real world game for $\mathcal{VC}^\dagger$, Game 0, the interactions between the challenger and A are exactly like those between the client and a worker in the real world – in particular, if A's response was well-formed, the challenger tells A so, but the challenger immediately aborts if A's response is malformed. Game 1 is identical to Game 0, except that when A queries

Verify, the challenger always answers with the correct y, whether A's response was well-formed or not, and the challenger never aborts. Let ϵ_i be A's success probability in Game i.

First, we show that if $\mathcal{VC}$ is secure, then ϵ_1 must be negligible. The intuition is simple: since the challenger always responds with the correct y, there is actually no information in these responses, since A could have computed y on its own. More formally, there is an algorithm B that breaks $\mathcal{VC}$ with probability ϵ_1 by using A as a sub-routine. B simply forwards communications between the challenger (now a challenger for the $\mathcal{VC}$ game) and A, except that B tells A the correct y w.r.t. all of A's responses. B forwards A's forgery along to the challenger.

Now, we show that $\epsilon_0 \leq \epsilon_1$, from which the result follows. Let E_{mal} be the event that A makes a malformed response, and let E_f be the event that A successfully outputs a forgery – i.e., where $\mathbf{Exp}_A^{Verif^\dagger}[\mathcal{VC}^\dagger, F, \lambda]$ outputs '1'. A's success probability, in either Game 0 or Game 1, is:

$$Prob[E_f] = Prob[E_f|E_{mal}] \cdot Prob[E_{mal}] + Prob[E_f|\neg E_{mal}] \cdot Prob[\neg E_{mal}] \quad (6.10)$$

If A does not make a malformed response, then Games 0 and 1 are indistinguishable to A; therefore, the second term above has the same value in Games 0 and 1. In Game 0, $Prob[E_f|E_{mal}] = 0$, since the challenger aborts. Therefore, $\epsilon_0 \leq \epsilon_1$. ■

In practice Theorem 4 implies that every time a malformed response is received, the client must re-garble the circuit (or, as we said above, make sure that the results of the verification procedure remain secret). Therefore the amortized efficiency of the client holds only if we assume that malformed responses do not happen very frequently.

In some settings, it is not necessary to inform the worker that its response is malformed, at least not immediately. For example, in the Folding@Home application [153], suppose the client generates a new garbled circuit each morning for its many workers. At the end of the day, the client stops accepting computations using this garbled circuit, and it (optionally) gives the workers information about the well-formedness of their responses. (Indeed, the client may reveal all of its secrets for that day.) In this setting, our previous security proof clearly holds even if there are arbitrarily many malformed responses.

6.6 Summary

We introduced the notion of Verifiable Computation as a natural formulation for the increasingly common phenomenon of outsourcing computational tasks to untrusted workers. In this environment, neither the software nor the hardware can be trusted. We describe a scheme that combines Yao's Garbled Circuits with a fully-homomorphic encryption scheme to provide extremely efficient outsourcing, even in the presence of an adaptive adversary. As an additional benefit, our scheme maintains the privacy of the client's inputs and outputs.

Chapter 7

Conclusion

Motivated by the trend of entrusting sensitive data and services to insecure computers, we develop techniques that allow a user to extend her trust in one device in order to securely utilize another device or service. A key constraint is our focus on commodity computers, particularly the need to preserve the performance and features expected of such platforms.

Using a logical framework, we analyze the perils of establishing trust in a local computer equipped with commodity (i.e., low-cost) security hardware and provide guidance on selecting a mechanism that preserves security while minimizing changes to existing computer designs. To make the results of such an interaction meaningful, we develop the Flicker architecture for providing an on-demand, secure execution environment for security-sensitive code. Building on recent improvements in commodity CPUs, Flicker provides strong isolation, reporting, and state preservation for security sensitive code. Because it runs only on demand, Flicker imposes zero overhead on non-security-sensitive code.

Given the ability to construct a secure environment on an individual endhost, we design and evaluate Assayer, an architecture for efficiently extending trust in such an environment to elements in the network. We show how protocols for a wide variety of applications, including spam identification, DDoS mitigation, and worm suppression, can benefit from such trusted, host-based information.

Finally, we show how a user can use a trusted local host to verify the results of computations entrusted to one or more remote workers who employ completely untrusted commodity hardware and software. To formalize this situation, we define the notion of *Verifiable Computing* and design the first protocol that supports the efficient, non-interactive outsourcing of arbitrary functions to such workers while guaranteeing the secrecy of the data and the integrity of the results.

Collectively, these techniques provide a secure foundation on which to build trusted systems worthy of handling security-sensitive data and services, without abandoning the performance and features that initially drove the adoption of commodity computers.

Bibliography

All of the URLs listed here are valid as of April, 2010.

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 340–353, 2005. (Referenced on page 29.)

[2] B. Acohido and J. Swartz. Unprotected PCs can be hijacked in minutes. *USA Today*, Nov. 2004. (Referenced on page 67.)

[3] Advanced Micro Devices. AMD64 architecture programmer's manual. AMD Publication no. 24593 rev. 3.14, 2007. (Referenced on pages 23, 42, and 49.)

[4] W. Aiello, S. N. Bhatt, R. Ostrovsky, and S. Rajagopalan. Fast verification of any remote procedure call: Short witness-indistinguishable one-round proofs for NP. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 463–474, 2000. (Referenced on page 51.)

[5] A. Alkassar, C. Stüble, and A.-R. Sadeghi. Secure object identification or: Solving the chess grandmaster problem. In *Proceedings of the New Security Paradigm Workshow (NSPW)*, 2003. (Referenced on page 57.)

[6] Amazon Web Services LLC. Amazon Elastic Compute Cloud. Online at `http://aws.amazon.com/ec2`. (Referenced on page 158.)

[7] American Electronics Association. eHealth 101: Electronic medical records reduce costs, improve care, and save lives. `http://www.aeanet.org/publications/AeA_CS_eHealth_EMRs.asp`, Dec. 2006. (Referenced on page 15.)

[8] S. R. Ames, Jr. Security kernels: A solution or a problem? In *Proceedings of the IEEE Symposium on Security and Privacy*, 1981. (Referenced on pages 4, 16, and 41.)

[9] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the IEEE/ACM Workshop on Grid Computing*, Nov. 2004. (Referenced on page 92.)

[10] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@Home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002. (Referenced on pages 74, 92, and 158.)

[11] R. Anderson. Cryptography and competition policy - issues with "Trusted Computing". In *Proceedings of the Workshop on Economics and Information Security*, May 2003. (Referenced on page 23.)

[12] R. Anderson and M. Kuhn. Tamper resistance – a cautionary note. In *Proceedings of the USENIX Workshop on Electronic Commerce*, pages 1–11, July 1995. (Referenced on page 58.)

[13] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 65–71, May 1997. (Referenced on pages 31, 47, and 56.)

[14] ARM. ARM security technology. PRD29-GENC-009492C, 2009. (Referenced on pages 23 and 48.)

[15] T. Arnold and L. van Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48(3), 2004. (Referenced on page 46.)

[16] J. Azema and G. Fayad. M-Shield mobile security technology: making wireless secure. Texas Instruments Whitepaper. Available at `http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf`, Feb. 2008. (Referenced on pages 23 and 48.)

[17] L. Babai. Trading group theory for randomness. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 421–429, 1985. (Referenced on page 51.)

[18] K.-H. Baek and S. Smith. Preventing theft of quality of service on open platforms. In *Proceedings of the IEEE/CREATE-NET Workshop on Security and QoS in Communication Networks*, Sept. 2005. (Referenced on page 54.)

[19] B. Balacheff, L. Chen, S. Pearson, D. Plaquin, and G. Proudler. *Trusted Computing Platforms – TCPA Technology in Context*. Prentice Hall, 2003. (Referenced on page 59.)

[20] D. Balfanz. *Access Control for Ad-hoc Collaboration*. PhD thesis, Princeton University, 2001. (Referenced on page 90.)

[21] D. Balfanz, D. Smetters, P. Stewart, and H. C. Wong. Talking to strangers: Authentication in ad-hoc wireless networks. In *Proceedings of the ISOC Symposium on Network*

and Distributed System Security (NDSS), Feb. 2002. (Referenced on page 57.)

[22] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahay, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Proceedings of CRYPTO*, pages 1–18, 2001. (Referenced on page 180.)

[23] B. Barak, I. Haitner, D. Hofheinz, and Y. Ishai. Bounded key-dependent message security. In *Proceedings of EuroCrypt*, 2010. (Referenced on page 166.)

[24] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, E. Kotsovinos, A. Madhavapeddy, R. Neugebauer, I. Pratt, and A. Warfield. Xen 2002. Technical Report UCAM-CL-TR-553, University of Cambridge, Jan. 2003. (Referenced on page 72.)

[25] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2003. (Referenced on pages 72 and 76.)

[26] M. Belenkiy, M. Chase, C. C. Erway, J. Jannotti, A. Küpçü, and A. Lysyanskaya. Incentivizing outsourced computation. In *Proceedings of the Workshop on Economics of Networked Systems (NetEcon)*, pages 85–90, 2008. (Referenced on page 50.)

[27] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of the USENIX Security Symposium*, 2006. (Referenced on page 26.)

[28] D. J. Bernstein. Cache-timing attacks on AES. Online at `http://cr.yp.to/papers.html#cachetiming`, Apr. 2005. (Referenced on page 72.)

[29] T. Beth and Y. Desmedt. Identification tokens - or: Solving the chess grandmaster problem. In *Proceedings of CRYPTO*, 1991. (Referenced on page 57.)

[30] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 1970. (Referenced on pages 131 and 149.)

[31] K. Borders and A. Prakash. Web tap: Detecting covert web traffic. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2004. (Referenced on page 142.)

[32] S. Brands and D. Chaum. Distance-bounding protocols. In *Proceedings of EuroCrypt*, 1994. (Referenced on pages 57 and 65.)

[33] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2004. (Referenced on pages 40 and 137.)

[34] A. Brodsky and D. Brodsky. A distributed content-independent method for spam detection. In *Proceedings of the Workshop on Hot Topics in Understanding Botnets*, 2007. (Referenced on page 141.)

[35] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the USENIX Security Symposium*, 2004. (Referenced on page 90.)

[36] D. Bruschi, L. Cavallaro, A. Lanzi, and M. Monga. Replay attack in TCG specification and solution. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2005. (Referenced on page 52.)

[37] CAIDA. Skitter. `http://www.caida.org/tools/measurement/skitter/`. (Referenced on page 154.)

[38] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2009. (Referenced on page 50.)

[39] D. Challener, J. Hoff, R. Catherman, D. Safford, and L. van Doorn. *Practical Guide to Trusted Computing*. Prentice Hall, Dec. 2007. (Referenced on page 59.)

[40] D. Chaum and T. Pedersen. Wallet databases with observers. In *Proceedings of CRYPTO*, 1992. (Referenced on page 50.)

[41] B. Chen and R. Morris. Certifying program execution with secure procesors. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2003. (Referenced on page 47.)

[42] L. Chen and M. D. Ryan. Offline dictionary attack on TCG TPM weak authorisation data, and solution. In *Proceedings of the Conference on Future of Trust in Computing*, 2008. (Referenced on page 52.)

[43] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the USENIX Security Symposium*, Aug. 2005. (Referenced on page 30.)

[44] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2008. (Referenced on pages 42 and 72.)

[45] Y. Chen, P. England, M. Peinado, and B. Willman. High assurance computing on open hardware architectures. Technical Report MSR-TR-2003-20, Microsoft Research, Mar. 2003. (Referenced on pages 42 and 129.)

[46] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2007. (Referenced on page 49.)

[47] D. D. Clark and D. R. Wilson. A comparison of commercial and military security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987. (Referenced on page 45.)

[48] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009. (Referenced on page 52.)

[49] Department of Justice, Bureau of Statistics. Press release: Identity theft 2004. `http://bjs.ojp.usdoj.gov/content/pub/press/it04pr.cfm`, Apr. 2006. (Referenced on page 15.)

[50] C. Dixon, T. Anderson, and A. Krishnamurthy. Phalanx: Withstanding multimillion-node botnets. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2008. (Referenced on page 143.)

[51] C. Dixon, A. Krishnamurthy, and T. Anderson. An end to the middle. In *Hot Topics in Operating Systems (HotOS)*, 2009. (Referenced on page 55.)

[52] C. Dwork, K. Nissim, M. Naor, M. Langberg, and O. Reingold. Succinct proofs for NP and spooky interactions. Online at `http://www.cs.bgu.ac.il/~kobbi/papers/spooky_sub_crypto.pdf`, Dec. 2004. (Referenced on page 51.)

[53] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 2001. (Referenced on page 32.)

[54] A. Einstein. On the electrodynamics of moving bodies. *Annalen der Physik*, 17:891–921, 1905. (Referenced on page 65.)

[55] J.-E. Ekberg and M. Kylänpää. Mobile trusted module (MTM) - an introduction. Technical Report NRC-TR-2007-015, Nokia Research Center, 2007. (Referenced on pages 48 and 53.)

[56] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open

platform. *IEEE Computer*, 36(7):55–62, July 2003. (Referenced on pages 42 and 129.)

[57] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2006. (Referenced on pages 29 and 45.)

[58] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *ACM Transactions on Computer Systems*, 21(3), 2003. (Referenced on page 123.)

[59] W.-C. Feng and T. Schluessler. The case for network witnesses. In *Proceedings of the IEEE Workshop on Secure Network Protocols*, Oct. 2008. (Referenced on page 55.)

[60] H. Finney. PrivacyCA. `http://privacyca.com`. (Referenced on page 39.)

[61] J. Franklin, M. Luk, A. Seshadri, and A. Perrig. PRISM: Enabling personal verification of code integrity, untampered execution, and trusted I/O or human-verifiable code execution. Technical Report CMU-CyLab-07-010, Carnegie Mellon University, Cylab, Feb. 2007. (Referenced on page 56.)

[62] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An inquiry into the nature and causes of the wealth of internet miscreants. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2007. (Referenced on page 14.)

[63] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, 2003. (Referenced on pages 18, 26, 27, 38, 42, and 45.)

[64] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS support and applications for Trusted Computing. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, May 2003. (Referenced on page 54.)

[65] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Trustworthy and personalized computing on public kiosks. In *Proceedings of the Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2008. (Referenced on pages 58 and 69.)

[66] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proceedings of the National Computer Security Conference*, 1989. (Referenced on pages 18, 23, 25, 26, 27, and 31.)

[67] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computation: Outsourcing computation to untrusted workers. In *Proceedings of CRYPTO*, Aug. 2010. (Referenced on page 20.)

[68] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. (Referenced on pages 160, 165, and 166.)

[69] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, 2009. (Referenced on pages 20, 50, 160, 161, 165, and 166.)

[70] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2005. (Referenced on pages 49 and 50.)

[71] H. Gobioff, S. Smith, J. Tygar, and B. Yee. Smart cards in hostile environments. In *Proceedings of the USENIX Workshop on Electronic Commerce*, July 1995. (Referenced on page 47.)

[72] B. D. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in retrospect. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1984. (Referenced on pages 4, 16, 41, and 72.)

[73] K. Goldman, R. Perez, and R. Sailer. Linking remote attestation to secure tunnel endpoints. In *Proceedings of the ACM Workshop on Scalable Trusted Computing (STC)*, 2006. (Referenced on pages 38 and 85.)

[74] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: interactive proofs for muggles. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, 2008. (Referenced on pages 51 and 161.)

[75] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. *SIAM Journal on Computing*, 18(1):186–208, 1989. (Referenced on page 51.)

[76] P. Golle and I. Mironov. Uncheatable distributed computations. In *Proceedings of the RSA Conference*, 2001. (Referenced on page 50.)

[77] M. T. Goodrich, M. Sirivianos, J. Solis, G. Tsudik, and E. Uzun. Loud and clear: Human-verifiable authentication based on audio. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2006. (Referenced on page 57.)

[78] D. Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006. (Referenced on page 99.)

[79] D. Grawrock. *Dynamics of a Trusted Platform*. Intel Press, 2008. (Referenced on pages 59 and 76.)

[80] GSM Association. GSM mobile phone technology adds another billion connections in just 30 months. GSM World Press Release, June 2006. (Referenced on page 23.)

[81] S. Gueron and M. E. Kounavis. New processor instructions for accelerating encryption and authentication algorithms. *Intel Technology Journal*, 13(2), 2009. (Referenced on page 23.)

[82] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-bot: Improving service availability in the face of botnet attacks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009. (Referenced on page 54.)

[83] S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga. Security evaluation of scenarios based on the TCG's TPM specification. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2007. (Referenced on page 52.)

[84] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *Proceedings of the Conference on Virtual Machine Research*, 2004. (Referenced on pages 30 and 45.)

[85] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the USENIX Security Symposium*, 2008. (Referenced on page 58.)

[86] S. Halevi and H. Krawczyk. Public-key cryptography and password protocols. *ACM Transactions Information and System Security*, 2(3), 1999. (Referenced on page 86.)

[87] S. Hao, N. A. Syed, N. Feamster, A. G. Gray, and S. Krasser. Detecting spammers with SNARE: Spatio-temporal network-level automatic reputation engine. In *Proceedings of the USENIX Security Symposium*, 2009. (Referenced on pages 125, 126, 141, and 156.)

[88] T. Hardjono and G. Kazmierczak. Overview of the TPM key management standard. TCG Presentations: `http://www.trustedcomputinggroup.org/resources/overview_of_the_tpm_key_management_standard`, Sept. 2008. (Referenced on pages 23, 124, and 125.)

[89] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification. Revision 3.0b, Oct. 2006. (Referenced on page 106.)

[90] S. Hohenberger and A. Lysyanskaya. How to securely outsource cryptographic computations. In *Proceedings of the IACR Theory of Cryptography Conference (TCC)*, 2005. (Referenced on page 50.)

[91] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Apr. 2008. (Referenced on page 14.)

[92] J. Howell, J. R. Douceur, J. Elson, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2008. (Referenced on page 14.)

[93] IBM. CCA basic services reference and guide for the IBM 4758 PCI and IBM 4764 PCI-X cryptographic coprocessors. 19th Ed., 2008. (Referenced on page 46.)

[94] Intel Corporation. Intel low pin count (LPC) interface specification. Revision 1.1, Aug. 2002. (Referenced on page 98.)

[95] Intel Corporation. Intel trusted execution technology – measured launched environment developer's guide. Document number 315168-005, June 2008. (Referenced on pages 23, 42, 43, and 49.)

[96] N. Itoi. Secure coprocessor integration with Kerberos V5. In *Proceedings of the USENIX Security Symposium*, 2000. (Referenced on page 53.)

[97] N. Itoi, W. A. Arbaugh, S. J. Pollack, and D. M. Reeves. Personal secure booting. In *Proceedings of the Australasian Conference on Information Security and Privacy (ACISP)*, pages 130–144, July 2000. (Referenced on page 56.)

[98] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *Proceedings of the ACM Symposium on Access Control Models And Technologies (SACMAT)*, 2006. (Referenced on page 45.)

[99] S. Jiang. WebALPS implementation and performance analysis. Master's thesis, Dartmouth College, 2001. (Referenced on page 54.)

[100] S. Jiang, S. Smith, and K. Minami. Securing web servers against insider attack. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2001. (Referenced on pages 32, 46, and 90.)

[101] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In

Proceedings of the USENIX Security Symposium, 2004. (Referenced on page 45.)

[102] Y. T. Kalai and R. Raz. Probabilistically checkable arguments. In *Proceedings of CRYPTO*, 2009. (Referenced on page 161.)

[103] B. Kaliski and J. Staddon. PKCS #1: RSA cryptography specifications. RFC 2437, 1998. (Referenced on page 95.)

[104] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, Nov. 1991. (Referenced on pages 4, 16, 41, and 72.)

[105] B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of the USENIX Security Symposium*, Aug. 2007. (Referenced on pages 44, 52, and 58.)

[106] R. Kennell and L. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the USENIX Security Symposium*, 2003. (Referenced on pages 49 and 50.)

[107] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *Proceedings of ACM SIGCOMM*, 2002. (Referenced on page 143.)

[108] A. Khorsi. An overview of content-based spam filtering techniques. *Informatica*, 31:269–277, 2007. (Referenced on page 141.)

[109] C. Kil, E. C. Sezer, A. Azab, P. Ning, and X. Zhang. Remote attestation to dynamic system properties. In *Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2009. (Referenced on pages 30 and 45.)

[110] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the ACM Symposium on Theory of computing (STOC)*, pages 723–732, 1992. (Referenced on page 51.)

[111] J. Kilian. Improved efficient arguments (preliminary version). In *Proceedings of CRYPTO*, pages 311–324, 1995. (Referenced on page 51.)

[112] D. Kilpatrick. Privman: A library for partitioning applications. In *Proceedings of the USENIX Annual Technical Conference*, 2003. (Referenced on page 90.)

[113] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009. (Referenced on pages 16, 41, and 72.)

[114] E. Kohler. *The Click modular router*. PhD thesis, MIT, Nov. 2000. (Referenced on

pages 146 and 149.)

[115] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. On the spam campaign trail. In *Proceedings of the Workshop on Large-Scale Exploits and Emergent Threats*, 2008. (Referenced on page 141.)

[116] E. T. Krovetz. UMAC: Message authentication code using universal hashing. RFC 4418, Mar. 2006. (Referenced on page 154.)

[117] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the Symposium on Computer Architecture*, Apr. 1994. (Referenced on page 110.)

[118] P. Lang. Flash the Intel BIOS with confidence. *Intel Developer UPDATE Magazine*, Mar. 2002. (Referenced on page 70.)

[119] J. LeClaire. Apple ships iPods with Windows virus. *Mac News World*, Oct. 2006. (Referenced on page 67.)

[120] R. B. Lee, P. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2005. (Referenced on pages 47 and 56.)

[121] A. Leung, L. Chen, and C. J. Mitchell. On a possible privacy flaw in direct anonymous attestation (DAA). In *Proceedings of the Conference on Trust*, 2008. (Referenced on page 40.)

[122] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009. (Referenced on page 49.)

[123] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000. (Referenced on pages 47 and 74.)

[124] Y. Lindell and B. Pinkas. A proof of Yao's protocol for secure two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009. (Referenced on pages 162, 164, 165, 172, 173, 174, and 175.)

[125] S. Lohr and J. Markoff. Windows is so slow, but why? *The New York Times*, Mar. 2006. (Referenced on page 72.)

[126] D. Magenheimer. Xen/IA64 code size stats. Xen developer's mailing list: `http://lists.xensource.com/`, Sept. 2005. (Referenced on page 72.)

[127] J. Marchesini, S. W. Smith, O. Wild, J. Stabiner, and A. Barsamian. Open-source applications of TCPA hardware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2004. (Referenced on pages 25, 28, 37, and 41.)

[128] R. Mayrhofer and H. Gellersen. Shake well before use: Intuitive and secure pairing of mobile devices. *IEEE Transactions on Mobile Computing*, 8(6):792–806, 2009. (Referenced on page 57.)

[129] J. M. McCune. *Reducing the Trusted Computing Base for Applications on Commodity Systems*. PhD thesis, Carnegie Mellon University, Jan. 2009. (Referenced on page 18.)

[130] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010. (Referenced on pages 44 and 146.)

[131] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2008. (Referenced on pages 18, 38, and 53.)

[132] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. Minimal TCB code execution (extended abstract). In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2007. (Referenced on page 18.)

[133] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. How low can you go? Recommendations for hardware-supported minimal TCB code execution. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2008. (Referenced on page 18.)

[134] J. M. McCune, A. Perrig, and M. K. Reiter. Seeing-is-believing: Using camera phones for human-verifiable authentication. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005. (Referenced on pages 57 and 69.)

[135] J. M. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *Proceedings of the ISOC Symposium on Network and Distributed System Security (NDSS)*, Feb. 2009. (Referenced on page 56.)

[136] J. M. McCune, A. Perrig, A. Sheshadri, and L. Doom. Turtles all the way down: Research challenges in user-based attestation. In *Proceedings of the Workshop on Hot Topics in Security*, 2007. (Referenced on page 16.)

[137] R. C. Merkle. A certified digital signature. In *Proceedings of CRYPTO*, pages 218–238, 1989. (Referenced on page 55.)

[138] Mersenne Research, Inc. The great internet Mersenne prime search. http://www.mersenne.org/prime.htm. (Referenced on page 158.)

[139] E. Messmer. Downadup/conflicker worm: When will the next shoe fall? Network World, Jan. 2009. (Referenced on page 14.)

[140] S. Micali. CS proofs (extended abstract). In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 1994. (Referenced on pages 51 and 160.)

[141] Microsoft Corporation. Code access security. MSDN .NET Framework Developer's Guide – Visual Studio .NET Framework 3.5, 2008. (Referenced on page 52.)

[142] Microsoft Corporation. Full volume encryption using Windows BitLocker drive encryption. Microsoft Services Datasheet, 2008. (Referenced on pages 53 and 58.)

[143] S. C. Misra and V. C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *Proceedings of the Conference on Computational Science and Its Applications (CCSIA)*, Jan. 2003. (Referenced on page 125.)

[144] C. Mitchell, editor. *Trusted Computing*. The Institution of Electrical Engineers, 2005. (Referenced on page 59.)

[145] D. Molnar. The SETI@Home problem. *ACM Crossroads*, 7.1, 2000. (Referenced on pages 15, 92, and 158.)

[146] F. Monrose, P. Wyckoff, and A. Rubin. Distributed execution with remote audit. In *Proceedings of ISOC Network and Distributed System Security Symposium (NDSS '99)*, Feb. 1999. (Referenced on page 50.)

[147] T. Moyer, K. Butler, J. Schiffman, P. McDaniel, and T. Jaeger. Scalable web content attestation. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2009. (Referenced on page 54.)

[148] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1998. (Referenced on page 89.)

[149] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the Conference on Compilier Construction*, 2002. (Referenced on page 90.)

[150] NIST Computer Security Resource Center (CSRC). National vulnerability database. http://nvd.nist.gov/home.cfm. (Referenced on page 14.)

[151] Oracle Corporation. Sun Utility Computing. Online at http://www.sun.com/service/sungrid/index.jsp. (Referenced on page 158.)

[152] Organization for Economic Co-operation and Development. Malicious software (malware): a security threat to the internet economy. http://www.oecd.org/dataoecd/53/34/40724457.pdf. (Referenced on page 14.)

[153] Pande Lab. The folding@home project. Stanford University, http://folding.stanford.edu/. (Referenced on pages 21, 74, 158, 162, and 181.)

[154] B. Parno. Bootstrapping trust in a "trusted" platform. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)*, July 2008. (Referenced on page 18.)

[155] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. Maggs, and Y.-C. Hu. Portcullis: Protecting connection setup from denial-of-capability attacks. In *Proceedings of ACM SIGCOMM*, Aug. 2007. (Referenced on page 143.)

[156] B. Parno, Z. Zhou, and A. Perrig. Help me help you: Using trustworthy host-based information in the network. Technical Report CMU-CyLab-09-016, Carnegie Mellon University, Cylab, Nov. 2009. In submission. (Referenced on page 19.)

[157] C. Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005. (Referenced on page 72.)

[158] A. Perrig, S. Smith, D. Song, and J. Tygar. SAM: A flexible and secure auction architecture using trusted hardware. *E-Commerce Tools and Applications*, 1, Jan. 2002. (Referenced on page 53.)

[159] Princeton Survey Research Associates International. Leap of Faith: Using the Internet Despite the Dangers (Results of a National Survey of Internet Users for Consumer Reports WebWatch). http://www.consumerwebwatch.org/pdfs/princeton.pdf, Oct. 2005. (Referenced on page 15.)

[160] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the USENIX Security Symposium*, Aug. 2003. (Referenced on page 90.)

[161] A. Ramachandran, K. Bhandankar, M. B. Tariq, and N. Feamster. Packets with provenance. Technical Report GT-CS-08-02, Georgia Tech, 2008. (Referenced on page 54.)

[162] A. Ramachandran and N. Feamster. Understanding the network-level behavior of

spammers. In *Proceedings of ACM SIGCOMM*, 2006. (Referenced on page 141.)

[163] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the USENIX Security Symposium*, Aug. 2005. (Referenced on page 93.)

[164] G. Rothblum. *Delegating Computation Reliably: Paradigms and Constructions*. PhD thesis, Massachusetts Institute of Technology, 2009. (Referenced on page 161.)

[165] G. Rothblum and S. Vadhan. Are PCPs inherent in efficient arguments? In *Proceedings of Computational Complexity (CCC'09)*, 2009. (Referenced on page 159.)

[166] C. Rudolph. Covert identity information in direct anonymous attestation (DAA). In *Proceedings of the IFIP Information Security Conference*, May 2007. (Referenced on page 40.)

[167] A.-R. Sadeghi, M. Selhorst, C. Stüble, C. Wachsmann, and M. Winandy. TCG inside? - A note on TPM specification compliance. In *Proceedings of the ACM Workshop on Scalable Trusted Computing (STC)*, 2006. (Referenced on page 52.)

[168] A.-R. Sadeghi and C. Stueble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Proceedings of the Workshop on New Security Paradigms (NSPW)*, 2004. (Referenced on page 45.)

[169] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype: Secure hypervisor approach to trusted virtualized systems. Technical Report RC23511, IBM Research, Feb. 2005. (Referenced on page 42.)

[170] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the USENIX Security Symposium*, 2004. (Referenced on pages 18, 27, 28, 37, 38, 45, 60, and 129.)

[171] L. Sarmenta, M. van Dijk, C. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS (extended version). Technical Report MIT-CSAIL-2006-064, Massachusetts Institute of Technology, 2006. (Referenced on page 55.)

[172] N. Saxena, J.-E. Ekberg, K. Kostiainen, and N. Asokan. Secure device pairing based on a visual channel (short paper). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006. (Referenced on page 57.)

[173] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the USENIX Security Symposium*, 1998. (Referenced on page 27.)

[174] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the ACM Conference on Operating Systems Principles (SOSP)*, 2007. (Referenced on pages 45 and 146.)

[175] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005. (Referenced on pages 49 and 50.)

[176] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004. (Referenced on pages 49 and 50.)

[177] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of the ISOC Symposium on Network and Distributed System Security (NDSS)*, 2006. (Referenced on page 45.)

[178] T. Shanley. *The Unabridged Pentium 4*. Addison Wesley, first edition, August 2004. (Referenced on page 111.)

[179] E. Shi, A. Perrig, and L. van Doorn. BIND: A time-of-use attestation service for secure distributed systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005. (Referenced on pages 44 and 54.)

[180] L. Singaravelu, C. Pu, H. Haertig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, 2006. (Referenced on pages 42 and 72.)

[181] S. Smith and V. Austel. Trusting trusted hardware: Towards a formal model for programmable secure coprocessors. In *Proceedings of the USENIX Workshop on Electronic Commerce*, 1998. (Referenced on page 52.)

[182] S. W. Smith. WebALPS: Using trusted co-servers to enhance privacy and security of web transactions. IBM Research Report RC-21851, October 2000. (Referenced on page 54.)

[183] S. W. Smith. Outbound authentication for programmable secure coprocessors. *Journal of Information Security*, 3:28–41, 2004. (Referenced on pages 25, 36, 37, and 52.)

[184] S. W. Smith. *Trusted Computing Platforms: Design and Applications*. Springer, 2005. (Referenced on page 59.)

[185] S. W. Smith, R. Perez, S. H. Weingart, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *Proceedings of the National Information Systems Security Conference*, Oct. 1999. (Referenced on pages 25, 32, 46, and 52.)

[186] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8), Apr. 1999. (Referenced on pages 17, 25, 32, 46, and 60.)

[187] Sophos. Do-it-yourself phishing kits found on the Internet, reveals Sophos. http://www.sophos.com/spaminfo/articles/diyphishing.html. (Referenced on page 14.)

[188] Sophos. Best Buy digital photo frames ship with computer virus. http://www.sophos.com/pressoffice/news/articles/2008/01/photo-frame.html, Jan. 2008. (Referenced on page 67.)

[189] C. Soriente, G. Tsudik, and E. Uzun. HAPADEP: Human-assisted pure audio device pairing. In *Proceedings of the International Information Security Conference (ISC)*, Sept. 2008. (Referenced on page 57.)

[190] C. Soriente, G. Tsudik, and E. Uzun. Secure pairing of interface constrained devices. *International Journal on Security and Networks*, 4(1), 2009. (Referenced on page 57.)

[191] E. R. Sparks. A security assessment of trusted platform modules. Technical Report TR2007-597, Dartmouth College, 2007. (Referenced on page 37.)

[192] D. Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security*, 3(1), 2000. (Referenced on pages 49 and 50.)

[193] F. Stajano and R. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Proceedings of the Security Protocols Workshop*, 1999. (Referenced on page 57.)

[194] Standards for Efficient Cryptography Group. SEC 2: Recommended elliptic curve domain parameters, 2000. (Referenced on page 148.)

[195] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the International Conference on Supercomputing*, 2003. (Referenced on pages 31, 47, and 74.)

[196] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the Symposium on Oper-*

ating Systems Design and Implementation (OSDI), 2006. (Referenced on pages 42, 72, and 90.)

[197] C. Tarnovsky. Security failures in secure devices. In *Black Hat DC Presentation*, Feb. 2008. (Referenced on page 58.)

[198] K. Thompson, G. J. Miller, and R. Wilder. Wide-area Internet traffic patterns and characteristics. *IEEE Network*, 11, 1997. (Referenced on pages 152 and 154.)

[199] Trusted Computing Group. PC client specific TPM interface specification (TIS). Version 1.2, Revision 1.00, July 2005. (Referenced on page 98.)

[200] Trusted Computing Group. Trusted Platform Module Main Specification. Version 1.2, Revision 103, 2007. (Referenced on pages 17, 25, 28, 33, 36, 37, 38, 39, 40, 60, 75, 83, and 89.)

[201] Trusted Computing Group. TCG mobile trusted module specification. Version 1.0, Revision 6, 2008. (Referenced on page 48.)

[202] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proceedings of EuroCrypt*, June 2010. (Referenced on pages 20 and 50.)

[203] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig. Lockdown: A safe and practical environment for security applications. Technical Report CMU-CyLab-09-011, Carnegie Mellon University, Cylab, July 2009. (Referenced on pages 16, 61, and 62.)

[204] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *Proceedings of the ISOC Symposium on Network and Distributed System Security (NDSS)*, 2005. (Referenced on pages 123 and 145.)

[205] J. von Helden, I. Bente, and J. Vieweg. Trusted network connect (TNC). European Trusted Infrastructure Summer School, 2009. (Referenced on page 53.)

[206] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *Proceedings of ACM SIGCOMM*, Sept. 2006. (Referenced on page 143.)

[207] C. Wallace. Worldwide PC market to double by 2010. Forrester Research, Inc. Press Release, Dec. 2004. (Referenced on page 23.)

[208] N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *Proceedings of the USENIX Security Symposium*, 2004. (Referenced on pages 123 and 145.)

[209] S. Weingart. Physical security for the μABYSS system. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987. (Referenced on page 47.)

[210] D. A. Wheeler. Linux kernel 2.6: It's worth more! Available at: http://www.dwheeler.com/essays/linux-kernel-cost.html, Oct. 2004. (Referenced on page 72.)

[211] S. White, S. Weingart, W. Arnold, and E. Palmer. Introduction to the Citadel architecture: Security in physically exposed environments. Technical Report RC16672, IBM T. J. Watson Research Center, 1991. (Referenced on page 47.)

[212] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2002. (Referenced on page 145.)

[213] M. M. Williamson. Design, implementation and test of an email virus throttle. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2003. (Referenced on page 141.)

[214] G. Wurster, P. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005. (Referenced on page 50.)

[215] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *Proceedings of ACM SIGCOMM*, Aug. 2005. (Referenced on page 143.)

[216] A. Yao. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 160–164, 1982. (Referenced on pages 161 and 162.)

[217] A. Yao. How to generate and exchange secrets. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986. (Referenced on pages 161 and 162.)

[218] B. S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994. (Referenced on page 47.)

[219] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3), Aug. 2002. (Referenced on page 90.)

[220] X. Zhuang, T. Zhang, H. Lee, and S. Pande. Hardware assisted control flow obfuscation for embedded processors. In *Proceedings of the Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2004. (Referenced on page 49.)