Bootstrapping Trust in Commodity Computers

Bryan Parno Jonathan M. McCune Adrian Perrig
CyLab, Carnegie Mellon University

Abstract

Trusting a computer for a security-sensitive task (such as checking email or banking online) requires the user to know something about the computer's state. We examine research on securely capturing a computer's state, and consider the utility of this information both for improving security on the local computer (e.g., to convince the user that her computer is not infected with malware) and for communicating a remote computer's state (e.g., to enable the user to check that a web server will adequately protect her data). Although the recent "Trusted Computing" initiative has drawn both positive and negative attention to this area, we consider the older and broader topic of bootstrapping trust in a computer. We cover issues ranging from the wide collection of secure hardware that can serve as a foundation for trust, to the usability issues that arise when trying to convey computer state information to humans. This approach unifies disparate research efforts and highlights opportunities for additional work that can guide real-world improvements in computer security.

1 Introduction

Suppose you are presented with two 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 may be completely infested with highly sophisticated malware. How can you tell which computer is which? How can you tell which computer you should use to check your email, update your medical records, or access your bank account?

As businesses and individuals entrust progressively greater amounts of security-sensitive data to computer platforms, it becomes increasingly important to inform them whether their trust is warranted. While the design and validation of secure software is an interesting study in its own right, we focus this survey on how trust can be bootstrapped 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 survey thematically, rather than chronologically. Thus, we examine mechanisms for securely collecting and storing information about the execution environment (§2), how to make use of that information locally (§3), techniques for securely conveying that information to an external party (§4), and various ways to convert the resulting information into a meaningful trust decision (§5).

Bootstrapping trust requires some foundational *root of trust*, and we review various candidates in §6. We then consider how the process of bootstrapping trust can be validated (§7) and used in applications (§8). Of course, creating trust ultimately involves human users, which creates a host of additional challenges (§9). Finally, all of the work we survey has certain fundamental limitations (§10), which leaves many interesting questions open for future work in this area (§11).

Much of this research 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 [45]. 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) [4]. However, like any other technology, the TPM can be used for either savory or unsavory purposes. One goal of this work is to highlight the many ways (and inspire the study of new ways) in which it can be used to improve user security without restricting user flexibility.

While Trusted Computing may be the most visible aspect of this research area, we show that many of the techniques used by Trusted Computing date back to the 1980s [33]. Hence these ideas extend beyond Trusted Computing's TPM to the general concept of bootstrapping trust in commodity computers. This 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 [39, 98]). In fact, many cellphones already incorporate stronger hardware support for security than many desktop computers and use some of the techniques described in this survey [8, 10]. Indeed, as CPU transistor counts continue to climb, CPU and computer vendors are increasingly willing to provide hardware support for secure systems (see, for example, Intel and AMD's support for virtualization [2, 47], and Intel's new AES instructions, which provide greater efficiency and resistance to side channels [40]). Thus, research in this area can truly guide the development of new hardware-supported security features.

Contributions. In this work, we **(1)** draw attention to the opportunities presented by the spread of commodity hardware support for security, **(2)** provide a unified presentation of the reasoning behind and the methods for bootstrapping trust, and **(3)** present existing research in a coherent framework, highlighting underexamined areas, and hopefully preventing the reinvention of existing techniques. While we aim to make this survey 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 for additional details (see also §12).

2 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 (§2.1). We then consider additional dynamic properties that may be of interest, e.g., whether the running code respects information-flow control (§2.2). Finally, we argue that establishing code identity is a more fundamental property than establishing any of the other dynamic properties discussed (§2.3). Unfortunately, the security offered by many of these techniques is still brittle, as we discuss in §10.

2.1 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 §5.

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

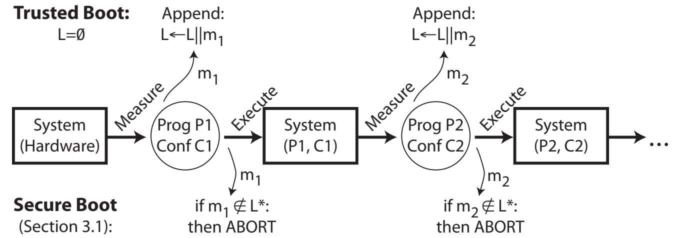

Figure 1: **Trusted Boot vs. Secure Boot.** *The state of a computer system changes as programs run with particular configurations. Trusted boot accumulates a list (L) of measurements for each program executed, but it does not perform any enforcement. Secure boot (§3.1) will halt the system if any attempt 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 [33].*

the IBM 4758 secure coprocessor [89, 90], Smith analyzes in greater detail which pieces of code can affect the security of a given piece of software [87], 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 [33, 63]. 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* [95]; 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 §6, 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. [33]), a hardware-based root of trust initiates the chain of trust by measuring the initial BIOS code (see Figure 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.

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

Figure 2: **Securely Recording Code Measurements.** *Techniques for preventing attacks on the measurement record differ based on the method used to secure the record.*

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 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 [30] 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 [13] 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). 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 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 [30, 33, 77] 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 [79], 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 [30, 33]. Before loading a new piece of software, Gasser et al. require the currently running system to generate a certificate for the new software [33]. 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 [30]). 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) [95]. Several research efforts have applied this approach to the Linux kernel, and developed techniques to improve its efficiency [63, 77].

For a hardware-backed hash chain, the hardware sets aside a protected memory register (called a Platform Configuration Register on the TPM [95]) that is initialized to a known value (e.g., 0) when the computer first boots. 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.

2.2 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 §2.3. 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 [27] and CFI [1] techniques transform a code binary by inserting inline reference monitors that enforce a variety of properties, such as stack and control-flow integrity. By submitting the transformed binary to the measurement infrastructure described in §2.1, 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 [20]. For example, a buffer overflow attack might overwrite the `isAdministrator` variable 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 [43], 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 [57] 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.

2.3 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 §2.2 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.

3 Can We Use Platform Information Locally?

We now discuss how accumulated platform information (§2) 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.

3.1 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. [33] and later dubbed "secure boot" [6].

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 com-

puter's firmware) [6, 33]. 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] [6]. With AEGIS, before a piece of software is allowed to execute, its identity is checked against a certificate 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. §4 discusses the techniques needed to provide more information to a remote party.

3.2 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 §2. Below, we discuss two of the most prominent protected storage solutions: the IBM 4758 cryptographic coprocessor and the Trusted Platform Module (TPM).

3.2.1 Tamper-Responding Protected Storage

The IBM 4758 family of cryptographic co-processors provides a rich set of secure storage facilities [24, 52, 89, 90]. 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. 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.

[1]Two relevant research efforts have used the name AEGIS. One is that of Arbaugh et al. [6] discussed in this section. The other is a design for a secure coprocessor by Suh et al. [93] and is discussed in §6.1.

3.2.2 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 §2.1) 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 §8.) 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.

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.

[2]The TPM ensures that these keys are only used for encryption operations (using RSA PKCS #1v2.0 OAEP padding [95]) and never for signing.

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. §8.2 discusses research on enhancing the TPM's counter facilities.

4 Can We Use Platform Info. Remotely?

§2 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.

4.1 Prerequisites

The *secure boot* model (§3.1) 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 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. §2 describes how a *trusted boot* process securely records this information in measurement chains (using either certificates or hashes).

4.2 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 (§6). 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 [87], or the TPM's Endorsement, Platform, and Conformance Credentials [95]. Attestation-specific mechanisms for conveying public keys in an authentic way are treated with respect to privacy issues in §4.3. 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.

4.2.1 General Purpose Coprocessor-based Attestation

Smith discusses the need for coprocessor applications to be able to authenticate themselves to remote parties [87]. 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 [87]. A noteworthy contribution not discussed here is a logic-based analysis of attestation.

4.2.2 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 3), software on the attestor's computer is responsible for relaying information between the remote verifier and the TPM [95]. 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. We discuss privacy issues regarding AIKs in §4.3.1.

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) [63], while IBM's Integrity Measurement Architecture contains the hash and full path to a loaded executable, and recursively measures all dynamic library dependencies [77]. 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 [91]. For example, 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 [30, 37, 64]. 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. [37] 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.

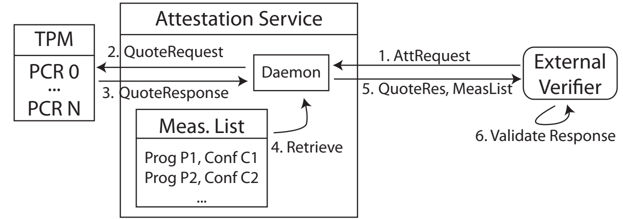

Figure 3: **Attestation.** *High-level summary of TPM-based attestation protocol based on signed hash chain measurements [95], e.g., as in IBM's Integrity Measurement Architecture [77]. Some protocol details are elided, e.g., the inclusion of an anti-replay nonce as part of the AttRequest message.*

McCune et al. consider a similar challenge in establishing a secure channel between a client system and an isolated execution environment on a server [64]. 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.

4.3 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.

4.3.1 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 [95], 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]

[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

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 [28].

4.3.2 Direct Anonymous Attestation

To address the limitations of Privacy CAs, a replacement protocol called Direct Anonymous Attestation (DAA) [14] was developed and incorporated into the latest TPM specification [95]. 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.

In practice, however, 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 [73], primarily by having the Issuer employ different long-term keys for different users. Several fixes have been proposed [60], but these attacks highlight the ability of implementation "details" to undermine the security of formally proven systems.

controlled environments, or for highly security-conscious individuals.

5 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 (5.1) and then review solutions that fall into two broad categories: solutions that provide only the identity of security-relevant code (5.2), and those that convey higher-level information (5.3).

5.1 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 that 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 [3, 36, 54, 58].

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 that have been developed to cope with this state-space explosion.

5.2 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 §2.1) become possible, enabling malicious code to potentially erase its tracks.

While layering is a time-honored technique for improving security and managing complexity [36, 54], 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 [63] that uses privilege layering to simplify measurement information. It mixes the trusted boot and secure boot processes described in §2.1 and §3.1. 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 §3.2. 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.

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) [26]. 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 (§2.1) 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 [30]. sHype, from IBM, enforces mandatory access control (MAC) policies at the granularity of entire virtual machines [76].

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) initiatives [2, 47]. 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.

As the OSLO bootloader project noted [55], a late launch allows the chain of trust described in §2.1 to be significantly shortened. One promising design is to late launch a VMM, thereby eliminating the possibility of malicious platform firmware attacking the VMM.

BIND [84] 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 found this approach could be extended even further [64] to provide a secure execution environment *on demand*. It combined late launch with sealed storage (see §3.2) 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.

5.3 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 §2.2, multiple research efforts have studied mechanisms for applying static (e.g., via type checking [53] or inline reference monitors [27]) or dynamic (e.g., via hypervisors [80] or security kernels [57]) 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) [50] that enforces an integrity policy called Clark Wilson-Lite (CW-Lite) [83]. CW-Lite relaxes the original Clark-Wilson [22] 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 [77] 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 [30] 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 [43, 75]. 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.

6 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, and 4) research solutions that attempt to instantiate a root of trust without custom hardware support.

6.1 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.

6.1.1 Commercial Solutions

IBM offers a family of general-purpose cryptographic co-processors with tamper-resistant and tamper-responding properties, including the PCI-based 4758 [52, 89, 90] and the PCI-X-based 4764/PCIXCC [9, 46]. 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) [35]. Smart cards are also discussed in §9.

6.1.2 Research Projects

μABYSS [99] and Citadel [100] 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 [102] 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 [62] and AEGIS[4] [93] designs do not trust the operating system, and include native support for partitioning cache and memory between mutually distrusting programs. The AEGIS [93] 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 [18]. 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 [59]. 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 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.

[4]Two relevant research efforts have used the name AEGIS. One is that of Arbaugh et al. [6] discussed in §2.1. The other is by Suh et al. [93] and is discussed in this section.

6.2 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 [25, 96]), or a smart card.

TPM-equipped Platforms. 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 §3.2.2). 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 [96], 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 [25]. 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 [8] or TI M-Shield [10]. Boot integrity is provided using a *secure boot* model (§3.1).

Smart Cards. Smart cards and SIM cards do not have any active tamper response mechanisms; instead, they attempt to protect a secret key through techniques such as hardware obfuscation [103]. Private key operations are performed within the card to protect the card's secrets from being exposed to untrusted terminals.

6.3 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 [21]. 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. [61] 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.

6.4 Research Solutions – No 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 [34, 56, 81, 82, 92]. 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 [2] or Intel TXT [47] (see §5.2). Secure storage remains a challenge as we discuss below.

The earliest proposal is due to Spinellis [92], 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 [56]. Seshadri et al. implement a timed checksum function on embedded systems as well as on PCs [81, 82]. Giffin et al. propose the use of self-modifying code to strengthen self-checksumming [34].

Attacks have recently been proposed against weakened versions of software-based attestation mechanisms [16, 101]; however, these attacks are primarily based on implementation flaws, rather than fundamental limitations of the approach. Nonetheless, 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).

7 Validating the Process

Bootstrapping trust can only be effective if we can validate the hardware and protocols involved. From a hardware perspective, Smith and Austel discuss efforts to apply formal methods to the design of secure coprocessors [85, 89]. 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) [15]. 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 [19]. Sadeghi et al. performed extensive testing on TPMs from multiple vendors to evaluate their compliance with the specification [74]. 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 [42]. Using an automated verification tool, they identify several inconsistencies and potential security problems.

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 [87]. Datta et al. use the Logic of Secure Systems (LS^2) [23] 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.

8 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.

8.1 Real World

Code Access Security in Microsoft .NET. Microsoft's Code Access Security is intended to prevent unauthorized *code* from performing privileged actions [67]. 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 [68], 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 harddrive's contents are sealed (see §3.2) 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 [97].

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 [25] 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.

8.2 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 [72], protecting the private key of a Certificate Authority [64], protecting the various private keys for a Kerberos Distribution Center [48].

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 [51, 86]. 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 [70]. 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 [64]. 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 sub-

mitting 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 [84] 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. Garfinkel et al. noted that secure hardware might help defend against network-based attacks [31]. However, the first design and implementation of this idea came from Baek and Smith, who describe an architecture for prioritizing traffic from privileged applications [11]. 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. Gummadi et al. propose the Not-A-Bot system [41], 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.

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 [78]. 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 [66], but this scheme would require modifications to the TPM, so that it could securely store the tree's root and validate updates to it.

9 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.

9.1 Trustworthy Verifier Device

To help a human establish trust in a computer, Itoi et al. describe a smart card-based solution called sAEGIS [49]. sAEGIS builds on the AEGIS [6] secure boot (see §3.1) 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 [59]. A complete architecture and implementation for these simple interfaces remains an open problem.

The Bumpy [65] 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.

Parno considers the challenges faced by a human user trying to learn the identity (e.g., authentic public key) of the TPM in the specific computer in front of her [71]. He 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. Parno also considers the relative merits of potential solutions.

9.2 Using Your Brain to Check a Computer

Roots of trust established based on timing measurements (§6.4) can potentially be verified by humans. Franklin et al. propose personally verifiable applications as part of the PRISM architecture for human-verifiable code execution [29]. 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 (§6.4) still face several challenges.

10 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.

10.1 Load-Time Versus Run-Time Guarantees

As described in §2, 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 §2.2 and §5.3 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.

10.2 Hardware Attacks

As discussed in §6, protection against hardware attacks has thus far been a tradeoff between cost (and hence ubiquity) and resilience [5]. Even simple hardware attacks, such as connecting the TPM's reset pin to ground, can undermine the security offered by this inexpensive solution [55]. 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 [94], 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 [68], 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 [32], 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 survey 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 [44]. Hence, an attacker who has physical access to the computer may be able to read these secrets directly out of the computer's memory.

11 Future Directions and Open Questions

Developing techniques to report the current state of the platform without relying on the assumption that previously loaded software does not contain vulnerabilities represents a promising direction for future work. This might take the form of hardware support for recording the fact that the loaded software has never deviated from its intended control flow (though even these protections do not wholly suffice [20]), or that all memory accesses have respected type safety. Similarly, existing efforts to convert code identity information into meaningful, security-relevant properties remains an open problem, despite the initial work discussed in §5.

In the area of cloud computing, a careful study is needed to determine what type of root of trust a cloud provider needs in order to provide adequate assurance to clients. Low-cost solutions (e.g., the TPM) that do not provide tamper-resistance may not suffice without additional legal protections.

Another area that has not been examined in detail is how trust bootstrapping can coexist with backups and recovery. How can a secure coprocessor's keys be backed up if they cannot leave the coprocessor? What happens if the TPM malfunctions? If an attestation convinces the user that her computer cannot be trusted, what remedy does she have? How can she revert to a previous, trustworthy state? Again, initial work in this area [7, 95] does not provide complete solutions.

Finally, the existing work on involving humans in attestations (§9) provides more questions and challenges than answers, and yet the end goal of trust bootstrapping is to help users make better security decisions.

12 Additional Reading

With a focus on the IBM 4758, Smith's book [88] 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 [12], but it is now mostly superseded by Grawrock's more recent book [38], which covers the motivation and design of the TPM, as well as Intel's late launch and virtualization support. Challener et al.'s book [17] 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 [69] contains a collection of articles surveying the general field of Trusted Computing, providing useful overviews of topics such as NGSCB and DAA.

13 Conclusions

In this survey, we organize and clarify extensive research on bootstrapping trust in commodity systems. We identify inconsistencies (e.g., attacks prevented 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.

We conclude that bootstrapping trust holds promise as a powerful primitive for building secure systems. We hope that this paper will help to clarify this sometimes controversial topic, and inspire other researchers to work in this area.

14 Acknowledgements

The authors are grateful to Virgil Gligor for stimulating discussions, and to Reiner Sailer, Ron Perez, and the anonymous reviewers for their insightful comments.

This research was supported in part by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 and MURI W 911 NF 0710287 from the Army Research Office, and grants CNS-0831440 and CCF-0424422 from the National Science Foundation. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, CMU, CyLab, NSF, or the U.S. Government or any of its agencies.

References

[1] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] Advanced Micro Devices. AMD64 architecture programmer's manual. AMD Publication no. 24593 rev. 3.14, 2007.

[3] S. R. Ames, Jr. Security kernels: A solution or a problem? In *IEEE S&P*, 1981.

[4] R. Anderson. Cryptography and competition policy - issues with "Trusted Computing". In *Proceedings of the Workshop on Economics and Information Security*, May 2003.

[5] R. Anderson and M. Kuhn. Tamper resistance – a cautionary note. In *Proc. of the USENIX Workshop on Electronic Commerce*, 1996.

[6] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2007.

[7] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, and J. M. Smith. Automated recovery in a secure bootstrap process. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 1998.

[8] ARM. ARM security technology. PRD29-GENC-009492C, 2009.

[9] T. Arnold and L. P. Van Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48(3), 2004.

[10] J. Azema and G. Fayad. M-Shield mobile security technology: making wireless secure. Texas Instruments Whitepaper.

[11] K.-H. Baek and S. Smith. Preventing theft of quality of service on open platforms. In *The Workshop on Security and QoS in Communication Networks*, 2005.

[12] B. Balacheff, L. Chen, S. Pearson, D. Plaquin, and G. Proudler. *Trusted Computing Platforms*. Prentice Hall, 2003.

[13] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of USENIX Security Symposium*, 2006.

[14] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *ACM CCS*, 2004.

[15] 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.

[16] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *ACM CCS*, 2009.

[17] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn. *Practical Guide to Trusted Computing*. Prentice Hall, Dec. 2007.

[18] B. Chen and R. Morris. Certifying program execution with secure procesors. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2003.

[19] 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.

[20] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of USENIX Security Symposium*, 2005.

[21] 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.

[22] D. D. Clark and D. R. Wilson. A comparison of commercial and military security policies. In *IEEE S&P*, 1987.

[23] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *IEEE S&P*, 2009.

[24] 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.

[25] J.-E. Ekberg and M. Kylänpää. Mobile trusted module (MTM) - an introduction. Technical Report NRC-TR-2007-015, Nokia Research Center, 2007.

[26] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *IEEE Computer*, 36(7):55–62, July 2003.

[27] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proc. of the Symposium on Operating Systems Design and Implementation*, 2006.

[28] H. Finney. PrivacyCA. `http://privacyca.com`.

[29] 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, CMU Cylab, Feb. 2007.

[30] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of ACM SOSP*, 2003.

[31] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS support and applications for Trusted Computing. In *Proceedings of HotOS*, 2003.

[32] 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)*, 2008.

[33] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proceedings of the National Computer Security Conference*, 1989.

[34] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *ACSAC*, 2005.

[35] H. Gobioff, S. Smith, J. Tygar, and B. Yee. Smart cards in hostile environments. In *Proceedings of the USENIX Workshop on Electronic Commerce*, 1996.

[36] B. D. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in retrospect. In *IEEE S&P*, 1984.

[37] 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.

[38] D. Grawrock. *Dynamics of a Trusted Platform*. Intel Press, 2008.

[39] GSM Association. GSM mobile phone technology adds another billion connections in just 30 months. GSM World Press Release, 2006.

[40] S. Gueron and M. E. Kounavis. New processor instructions for accelerating encryption and authentication algorithms. *Intel Technology Journal*, 13(2), 2009.

[41] 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.

[42] 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.
[43] 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.
[44] 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.
[45] T. Hardjono and G. Kazmierczak. Overview of the TPM key management standard. TCG Presentations: `https://www.trustedcomputinggroup.org/news/`, Sept. 2008.
[46] IBM. CCA basic services reference and guide for the IBM 4758 PCI and IBM 4764 PCI-X cryptographic coprocessors. 19th Ed., 2008.
[47] Intel Corporation. Intel trusted execution technology – measured launched environment developer's guide. Document number 315168-005, 2008.
[48] N. Itoi. Secure coprocessor integration with Kerberos V5. In *Proceedings of the USENIX Security Symposium*, 2000.
[49] 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)*, 2001.
[50] 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.
[51] S. Jiang. WebALPS implementation and performance analysis. Master's thesis, Dartmouth College, 2001.
[52] S. Jiang, S. Smith, and K. Minami. Securing web servers against insider attack. In *ACSAC*, 2001.
[53] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the USENIX Security Symposium*, 2004.
[54] 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), 1991.
[55] B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of the USENIX Security Symposium*, 2007.
[56] R. Kennell and L. Jamieson. Establishing the genuinity of remote computer systems. In *Proc. of USENIX Security Symposium*, 2003.
[57] 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.
[58] 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 ACM SOSP*, 2009.
[59] R. B. Lee, P. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, 2005.
[60] 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.
[61] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of USENIX NSDI*, 2009.
[62] 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 *The Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.
[63] J. Marchesini, S. W. Smith, O. Wild, J. Stabiner, and A. Barsamian. Open-source applications of TCPA hardware. In *ACSAC*, 2004.
[64] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proc. of the ACM European Conference on Computer Systems*, 2008.
[65] J. M. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *Proceedings of NDSS*, Feb. 2009.
[66] R. C. Merkle. A certified digital signature. In *Advances in Cryptology (CRYPTO)*, 1989.
[67] Microsoft Corporation. Code access security. MSDN .NET Framework Developer's Guide – Visual Studio .NET Framework 3.5, 2008.
[68] Microsoft Corporation. Full volume encryption using Windows BitLocker drive encryption. Microsoft Services Datasheet, 2008.
[69] C. Mitchell, editor. *Trusted Computing*. The Institution of Electrical Engineers, 2005.
[70] T. Moyer, K. Butler, J. Schiffman, P. McDaniel, and T. Jaeger. Scalable web content attestation. In *ACSAC*, 2009.
[71] B. Parno. Bootstrapping trust in a "trusted" platform. In *Proceedings of the USENIX Workshop on Hot Topics in Security*, July 2008.
[72] A. Perrig, S. Smith, D. Song, and J. Tygar. SAM: A flexible and secure auction architecture using trusted hardware. *E-Commerce Tools and Applications*, 2002.
[73] C. Rudolph. Covert identity information in direct anonymous attestation (DAA). In *Proc. of the IFIP Info. Security Conference*, 2007.
[74] A.-R. Sadeghi, M. Selhorst, C. Stüble, C. Wachsmann, and M. Winandy. TCG inside? - A note on TPM specification compliance. In *Proceedings of ACM STC*, 2006.
[75] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Proceedings of the Workshop on New Security Paradigms (NSPW)*, 2004.
[76] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen open-source hypervisor. In *ACSAC*, Dec. 2005.
[77] 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.
[78] 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.
[79] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proc. of the USENIX Security Symposium*, 1998.
[80] 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 ACM SOSP*, 2007.
[81] 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 SOSP*, Oct. 2005.
[82] A. Seshadri, A. Perrig, L. vDoorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE S&P*, 2004.
[83] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of NDSS*, 2006.
[84] E. Shi, A. Perrig, and L. van Doorn. BIND: A time-of-use attestation service for secure distributed systems. In *IEEE S&P*, 2005.
[85] S. Smith and V. Austel. Trusting trusted hardware: Towards a formal model for programmable secure coprocessors. In *USENIX Workshop on Electronic Commerce*, 1998.
[86] S. W. Smith. WebALPS: Using trusted co-servers to enhance privacy and security of web transactions. IBM Res. Rep. RC-21851, 2000.
[87] S. W. Smith. Outbound authentication for programmable secure coprocessors. *Journal of Information Security*, 3:28–41, 2004.
[88] S. W. Smith. *Trusted Computing Platforms: Design and Applications*. Springer, 2005.
[89] S. W. Smith, R. Perez, S. H. Weingart, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *National Information Systems Security Conference*, 1999.
[90] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8), 1999.
[91] E. R. Sparks. A security assessment of trusted platform modules. Technical Report TR2007-597, Dartmouth College, 2007.
[92] D. Spinellis. Reflection as a mechanism for software integrity verification. *ACM Trans. on Information and System Security*, 3(1), 2000.
[93] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proc. of the Conference on Supercomputing*, 2003.
[94] C. Tarnovsky. Security failures in secure devices. In *Black Hat DC Presentation*, Feb. 2008.
[95] Trusted Computing Group. Trusted Platform Module Main Specification. Version 1.2, Revision 103, 2007.
[96] Trusted Computing Group. TCG mobile trusted module specification. Version 1.0, Revision 6, 2008.
[97] J. von Helden, I. Bente, and J. Vieweg. Trusted network connect (TNC). European Trusted Infrastructure Summer School, 2009.
[98] C. Wallace. Worldwide PC market to double by 2010. Forrester Research, Inc. Press Release, Dec. 2004.
[99] S. Weingart. Physical security for the μABYSS system. In *IEEE S&P*, 1987.
[100] 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.
[101] G. Wurster, P. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *IEEE S&P*, 2005.
[102] B. S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.
[103] X. Zhuang, T. Zhang, H. Lee, and S. Pande. Hardware assisted control flow obfuscation for embedded processors. In *Proc. of the Conference on Compilers, Architecture & Synthesis for Embedded Systems*, 2004.