Trust Extension for Commodity Computers

Bryan Parno

1. INTRODUCTION

As society rushes to digitize sensitive information and services, it is imperative to adopt adequate security protections. However, such protections fundamentally conflict with the benefits we expect from commodity computers. Consumers and businesses value commodity computers because they provide good performance and an abundance of features at relatively low cost, but attempts to construct secure systems "from the ground up" have proven expensive, time-consuming, and unable to keep pace with the changing demands of the marketplace [2, 9, 12, 13]. For example, the VAX VMM security kernel was developed over the course of eight years of considerable effort, but in the end, the project failed, and the kernel was never deployed. This failure was due, in part, to the absence of support for Ethernet – a feature considered critical by the time the kernel was completed, but not anticipated when it was initially designed [12].

Rather than building secure systems from scratch, we argue that we can resolve the tension between security and features by extending the trust a user has in one device to enable her to securely use another commodity device or service, without sacrificing the performance and features expected of commodity systems [22]. Note that we concentrate on average users and commodity systems, rather than on advanced users, special-purpose computers, or highly constrained environments (such as those found within the military).

At a high level, we support this premise by developing techniques to allow a user to employ a small, trusted, portable device to securely learn what code is executing on her local computer. Rather than entrusting her data to the mountain of buggy code likely running on her computer, we construct an on-demand secure execution environment which can perform security-sensitive tasks and handle private data in complete isolation from all other software (and most hardware) on the system. Meanwhile, non-security-sensitive software retains the same abundance of features and performance it enjoys today.

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

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

Below, we provide an overview of technologies we employ (Section 2), and define trust extension (Section 3). We then describe how to extend trust in a special-purpose mobile device to verify the security hardware in the user's local machine (Section 4), how to extend that trust in a meaningful way to software on the local machine (Section 5), how to extend trust in that software to network elements (Section 6), and finally how to extend that trust to remote computers in which neither the software nor the hardware is trusted (Section 7). Space constraints restrict the amount of related work we can cover here; for a more detailed comparison with other work in this space, please see my dissertation [22] or our book on the large body of work examining the bootstrapping of trust in commodity computers [24].

2. BACKGROUND

While other instantiations are possible, several of the techniques in this article are described in terms of features of Trusted Platform Modules (TPMs) and recent CPUs from AMD and Intel; we summarize these features below.

Measurement.

When a TPM-equipped platform first boots, platform hardware takes a measurement (a SHA-1 hash) of the BIOS and records the measurement in one of the TPM's Platform Configuration Registers (PCR) [28, 31]. The BIOS is then responsible for measuring the next piece of software (e.g., the bootloader) and any associated data files. The BIOS records the measurement in a PCR before executing the software. If each subsequent piece of software performs these steps (measure, record, execute), then the TPM holds a set of measurements of all code executed on the platform [28].

Attestation.

To securely convey measurements to an external verifier, the TPM creates attestations: with a verifier-supplied nonce, the TPM uses a private key that is never accessible outside the TPM to generate a TPM *quote*, i.e., a digital signature over the nonce and the contents of the PCRs. The nonce assures the verifier that the attestation is fresh and not from a previous boot cycle.

To ensure the attestation comes from a real hardware TPM (rather than a software emulation), the TPM comes with an endorsement keypair and an endorsement certificate for the public key from the platform's manufacturer declaring that it does indeed belong to a real TPM. To preserve user privacy, the TPM uses Attestation-Identity Keys, anonymously bound to the endorsement key, to sign attestations.

Secure Storage.

The TPM also includes a limited amount of nonvolatile RAM (NVRAM). Reading and writing to NVRAM can be restricted based on the contents of the PCRs, so an NVRAM location can be made accessible only to a particular set of software. The most straightforward way to store data securely is to define an NVRAM location large enough to hold a symmetric key and use the PCR-based restrictions to prevent other software from reading or writing the key. The symmetric key can then be used to encrypt and MAC bulk data.

Dynamic Root of Trust for Measurement.

To protect the launch of a Virtual Machine Monitor (VMM), recent CPUs from AMD and Intel extend the x86 instruction set to support a *dynamic root of trust for measurement* (DRTM) operation [1, 11].

Essentially, DRTM provides many of the security benefits of rebooting the computer (e.g., starting from a clean-slate), while bypassing the overhead of a full reboot, i.e., devices remain enabled, the BIOS and bootloader are not invoked, memory contents remain intact, etc. DRTM is implemented via a new *SKINIT* instruction on AMD (or *GETSEC[SENTER]* on Intel), which can launch a VMM at an arbitrary time (hence the colloquialism *late launch*) with built-in protection against software-based attacks. When a DRTM is invoked, the CPU's state is reset, and direct memory access (DMA) protections for a region of memory are enabled. The CPU hashes the contents (e.g., data and executable code) of the memory region, extends the measurement into the TPM's PCR 17 (and 18 on Intel), and begins executing the code.

Trust Assumptions.

Relying on a TPM's guarantees requires trusting the TPM manufacturer, the platform vendor, and the PKI linking them to the TPM's keys. The CPU, RAM, and chipset must be trusted as well. The TPM is designed to withstand software based attacks, and limited hardware attacks, such as rebooting the machine. It is not expected to resist sophisticated physical attacks. TPM-based systems also do not typically aim to prevent attacks on the machine's availability.

3. WHAT IS TRUST EXTENSION?

Creating trustworthy systems requires advances on many fronts: better programming languages, better operating systems, better network protocols, and better definitions of security. More fundamentally, however, we must enable both computers and users to make accurate, informed trust decisions. After all, even if software does improve, we must be able to determine *which* systems employ the new and improved software! In particular, it is critical that a user be able to judge whether a system (either local or remote) should be trusted before she hands over her sensitive data. Similarly, if a network element (e.g., a router) can trust information from an endhost, then numerous protocol optimizations become possible.

As a result, this article describes techniques that provide firm evidence on which to base such trust decisions. In particular, once the user decides to trust a particular device (e.g., her cellphone), we show how to use that device to verify, on her behalf, the trustworthiness of other devices and services (e.g., her laptop or a cloud-based computation). The user's trust in the original device allows her to trust its assessment of the new device or service. If the assessment is positive, then the user has essentially *extended* the trust she has in her first device to include the new device or service.

Informally speaking, we use the following definition of trust: to trust an entity X with her private data (or with a security-sensitive task), a user 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.

4. BOOTSTRAPPING TRUST IN A COMMODITY COMPUTER

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

If the user's computer is physically secure, then we can make use of special-purpose secure hardware to monitor and report on the software state of the platform. Given the software state, the user (or an agent acting on the user's behalf) can decide whether the platform should be trusted. While a full-blown secure coprocessor, such as the IBM 4758 [30], might be appealing, cost considerations limit deployment. However, for the last few years, over 350 million [34] commodity computers have been sold with a special-purpose security chip called the TPM [31]. With appropriate software support, the TPM can be used to report information about the software executing on the computer (see Section 2). The resulting attestation can be verified by a user's trusted device, such as a cellphone or a special-purpose USB fob [33]. Thus, the user's trust in her device can be extended, via the TPM, to establish trust in the software on a machine. For example, Garriss et al. use a cellphone to verify a virtual machine monitor on a kiosk computer [6], though a kiosk may not satisfy the hardware security assumptions discussed above.

However, even given a TPM-equipped machine, the question remains: How do we bootstrap trust in the TPM itself? Surprisingly, neither the TPM specifications nor the academic literature considered this problem. Instead, it was assumed that the user magically possesses the TPM's public key. Unfortunately, any straightforward approach to trusting the TPM falls victim to a *cuckoo attack*[1] [21, 22]. In this attack, the adversary convinces the user that a TPM the adversary physically controls in fact resides in the user's own local computer. Figure 1(a) illustrates one possible implementation of the cuckoo attack. Malware on the user's local machine proxies the user's TPM-related messages to a remote, TPM-enabled machine controlled by the attacker. Any credential the user requests from her local TPM (e.g., an Endorsement Certificate) can also be provided by the attacker's TPM and relayed to the malware on the user's machine.

To analyze the cuckoo attack formally, we develop a model [21, 22], using predicate logic, for bootstrapping trust in a local computer equipped with secure hardware. The model defines a set of predicates that describe the trustworthiness of people, computers, and secure hardware (e.g., TPMs), and a set of trust axioms. For example, if a trusted person P says computer C is secure, then we can conclude that C is physically secure; this encodes our previous assumption that people are reasonably effective at assessing and maintaining physical security.

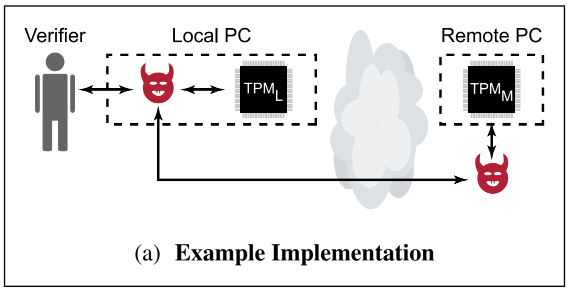
If we try to use this model to reason about the security of the user's local machine, we arrive at a logical contradiction, namely that the local machine is both trusted and untrusted. This contradiction captures the essence of the cuckoo attack: the user cannot decide whether to trust the local machine.

From Figure 1(a), it may seem that we can prevent the cuckoo attack by severing the connection between the local malware and the adversary's remote PC. The assumption is that, without a remote TPM to provide the correct responses, the infected machine must either refuse to respond or allow the true TPM to communicate with the user's device (thus, revealing the presence of the malware).

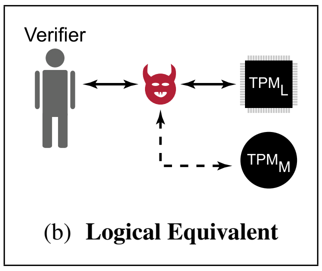
However, regardless of the implementation, cutting off network access fundamentally fails to prevent the cuckoo attack. As shown in Figure 1(b), the cuckoo attack is possible because the malware on the local machine has access to a "TPM oracle" that provides TPM-like answers without providing TPM security guarantees. If the local malware can access this oracle without network access, then cutting off network access is insufficient to prevent the cuckoo attack. In particular, since the adversary has physical possession of TPM_M, he can extract its private keys and credentials and include this data with the malware. Thus provisioned, the malware on the local machine can emulate TPM_M, even without network access.

Instead, we argue that the right way to prevent the cuckoo attack

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



(a) **Example Implementation**



(b) **Logical Equivalent**

Figure 1: The Cuckoo Attack. ***In one implementation of the cuckoo attack (a), malware on the user's local machine sends messages intended for the local TPM (TPM$_L$) to a remote attacker who feeds the messages to a TPM (TPM$_M$) inside a machine the attacker physically controls. Given physical control of TPM$_M$, the attacker can violate its security guarantees via hardware attacks. Thus, at a logical level (b), the attacker controls all communication between the verifier and the local TPM, while having access to an oracle that provides all of the answers a normal TPM would, without providing the security properties expected of a TPM.***

is to introduce a secure channel to the TPM. This channel can be instantiated either via a physically hardwired channel allowing the user to connect directly to the TPM on the local machine, or via an approach that allows the user to learn some authentic cryptographic information about the local TPM and hence establish a cryptographic secure channel.

Using both our formal model and a usability assessment, we analyze a dozen instantiations of these various approaches [21, 22], including adding a special-purpose port or button to the computer, reusing existing interfaces (e.g., USB or Firewire), employing software-only attestation, and encoding cryptographic material as a 2-D barcode or serial number printed on the computer. Each has its share of advantages and disadvantages. In the short term, placing an alphanumeric hash of the TPM's public key on the exterior of the computer seems to offer the best tradeoff between security, usability, and deployability. In the long term, the strongest security would come from a special-purpose hardware interface directly wired to the machine's secure hardware (e.g., the TPM) and designed so that the user cannot inadvertently connect the verifying device to another interface. This solution removes almost every opportunity for user error, does not require the preservation of secrets, and does not require software updates.

5. SECURELY EXECUTING CODE ON A COMMODITY COMPUTER

Unfortunately, merely establishing a secure connection between the user and the security hardware on her computer does not suffice to provide a full-featured, trustworthy execution environment. Security hardware tends to be either resource-impoverished or special-purpose (or both). Hence, we need to extend the user's trust in the security hardware to trust in the user's entire computer.

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

Previous work attempted to deal with this problem by running a *persistent security layer* in the computer's most privileged mode [5, 9, 12, 13, 29]. This layer has been variously dubbed a security kernel, a virtual machine monitor (VMM), or a hypervisor. This layer is responsible for creating isolation domains for ordinary, untrusted code and for the security-sensitive code. Unfortunately, this approach has a number of inherent drawbacks. The security layer's need to interpose on hardware accesses leads to performance degradation for ordinary code, and often requires eliminating access to devices that are too complicated to emulate (e.g., a 3D graphics card) [4]. Furthermore, the need to run both untrusted and trusted code simultaneously can lead to security vulnerabilities (e.g., side-channel attacks [26]), as well as code bloat in the security layer; the initial implementation of the Xen VMM required 42K lines of code [4] and within a few years almost doubled to 83K lines [14].

To avoid these drawbacks, we develop the Flicker architecture [15, 17–19, 22], which is designed to satisfy the need for features *and* security. Indeed, Flicker shows that these conflicting needs can both be satisfied by constructing a secure execution environment *on demand*. When invoked for secure code execution (e.g., signing a certificate or authenticating to a website), Flicker creates an isolated environment such that none of the software executing before Flicker begins can monitor or interfere with Flicker code execution, and all traces of Flicker code execution can be eliminated before regular software execution resumes. For example, a Certificate Authority (CA) could sign certificates with its private key and keep the key secret, even from an adversary who controls the BIOS, OS, and DMA-enabled devices.

5.1 Secure Code Execution

We describe Flicker, as implemented on an AMD CPU, in more detail; the sequence, shown in Figure 2, is similar on Intel.

To employ Flicker, application developers must provide the security-sensitive code (known as the Piece of Application Logic or PAL) selected for Flicker protection as an x86 binary and define its interface with the remainder of their application; we have developed tools that aid in automating this process. To create an SLB (the Secure Loader Block supplied as an argument to the DRTM instruction, *SKINIT*), the application developer links her PAL against a code module we have developed that performs the steps necessary to set up and tear down the Flicker session.

Since *SKINIT* is a privileged instruction, to execute the resulting SLB, the application passes it to a kernel module we have developed. It allocates memory, initializes various values in the SLB, and handles untrusted setup and tear-down operations. The kernel module is not included in the Trusted Computing Base (TCB) of the application, since its actions are verified.

Flicker achieves its properties using the DRTM capabilities summarized in Section 2. Instead of launching a VMM, Flicker pauses the current execution environment (e.g., the untrusted OS). In particular, Flicker saves information about the kernel's page tables to memory, along with the contents of key registers, e.g., `CR0`, `CR3`, `GDTR`, `IDTR`, and `TR`. On a multi-core machine, execution must be halted on all but the Boot Strap Processor, and the other cores must be sent an INIT Inter-Processor Interrupt (IPI) so that they respond correctly to a handshaking synchronization step performed during the execution of *SKINIT*.

The actual Flicker session begins with the execution of the *SKINIT* instruction, which receives the SLB selected for Flicker protection

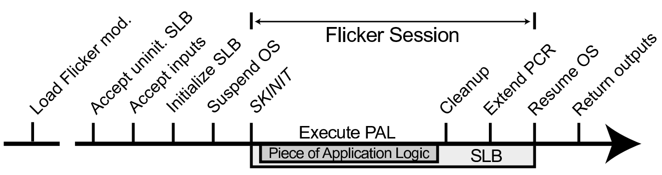


Figure 2: PAL Execution. ***Timeline showing the steps necessary to execute a*** PAL***. The Secure Loader Block (SLB) includes the*** PAL ***and the code necessary to initialize and terminate the Flicker session. The gap in the time axis indicates that the Flicker kernel module is loaded only once.***

as an argument. As described in Section 2, *SKINIT* resets CPU state, adds entries to the Device Exclusion Vector to disable Direct Memory Access (DMA) to the memory region containing the SLB, disables interrupts (including System Management Interrupts) to prevent the previously executing code from regaining control, disables debugging support, even for hardware debuggers, and extends a hash of the SLB into a PCR in the TPM. Finally, it hands control to Flicker's initialization routine. To simplify execution of PAL code, Flicker (i) loads the Global Descriptor Table (GDT), (ii) loads the `CS`, `DS`, and `SS` registers, (iii) transitions to ring 3 (user space) execution, and (iv) calls the PAL, providing the address of PAL inputs as a parameter. When PAL execution terminates, Flicker cleans up any traces of the security-sensitive code's execution by clearing out memory and register contents. Finally, it resumes the previous execution environment. This entails returning to ring 0 (kernel) execution, restoring the saved OS state, reenabling paging, reenabling interrupts, and jumping back to kernel code.

Since we only deploy Flicker's protections on demand, Flicker induces no performance overhead or feature reduction during regular computer use. Limiting Flicker's persistence also strengthens Flicker's security guarantees, since it avoids the complexity (and hence potential vulnerability) of solutions based on a VMM or a security kernel. Hence, we were able to implement the core Flicker system with a tiny TCB of 250 lines of code.

Naturally, however, non-persistence poses its own set of challenges. For instance, to enable more complex applications, we must leverage TPM-based secure storage (Section 2) to maintain state across Flicker sessions. For example, a Flicker-based application may wish to interact with a remote entity over the network. Rather than include an entire network stack and device driver in the PAL (and hence the TCB), we can invoke Flicker upon the arrival of each message, using secure storage to protect sensitive state between invocations. Preventing a variety of subtle attacks on this saved state requires developing additional protocols [23].

A platform using Flicker can convince remote parties that a Flicker session executed with a particular PAL. Our approach builds on the TPM attestation process described in Section 2. As part of Flicker's execution, the *SKINIT* instruction resets the value of PCR 17 to 0 and then extends it with the hash of the SLB (which contains the application-specific PAL). Thus, PCR 17 will take on the value $SHA(0\text{x}00^{20}||SHA(P))$, where P represents the SLB's code. The properties of the TPM, chipset, and CPU guarantee that no other operation can cause PCR 17 to take on this value. Thus, an attestation of the value of PCR 17 will convince a remote party that the PAL was executed using Flicker's protection.

To provide result integrity, after PAL execution terminates, Flicker extends PCR 17 with hashes of the PAL's input and output parameters. As another important security procedure, after extending the PAL's results into PCR 17, Flicker extends PCR 17 with a fixed public constant. This provides two powerful security properties: (i) it prevents any other software from extending values into PCR 17 and attributing them to the PAL (the fact that *SKINIT* resets PCR 17 to 0 prevents malicious software from extending values before the Flicker session); and (ii) it revokes access to any secrets kept in the TPM's secure storage which may have been available during PAL execution; specifically, the TPM's secure storage facilities (see Section 2) only allow access to the PAL's state when the value in PCR 17 matches that after the *SKINIT* operation. Hence, extending data into PCR 17 will cause subsequent attempts to access the secure storage to fail.

Combining the techniques above, a PAL can communicate securely (i.e., with both secrecy and integrity protections) with a remote party, via a cryptographic secure channel. Specifically, the PAL generates an asymmetric keypair $\{K_{\text{PAL}}, K_{\text{PAL}}^{-1}\}$ within its secure execution environment. It securely stores the private key K_{PAL}^{-1} under the value of PCR 17 so that only the identical PAL invoked in the secure execution environment can access it. As with all output parameters, the public key K_{PAL} is extended into PCR 17 before it is output to the application running on the untrusted host. The application generates a TPM quote over PCR 17 based on the nonce from the remote party. The quote allows the remote party to determine that the public key K_{PAL} was generated by a PAL running in the secure execution environment. The remote party uses the public key to create a secure channel to future invocations of the PAL.

We implement Flicker for both 32-bit Windows and Linux running on AMD and Intel CPUs. The implementation includes a number of useful code modules that PALs can choose to include, such as modules for memory management, TPM operations, cryptographic operations, and secure channels. Using these tools, we applied Flicker to several broad classes of applications. To illustrate stateless applications, we use Flicker to provide verifiable isolated execution of a kernel rootkit detector on a remote machine. To illustrate applications for which integrity protection of application state suffices, we use Flicker to verify the execution of a distributed computing application based on the BOINC framework [3]. BOINC allows scientists to develop computational modules that operate on public data; hence, we can run the module inside Flicker to provide stronger guarantees about the integrity of the results clients return. Finally, to illustrate applications whose state requires secrecy and integrity, we use Flicker to protect computations performed with: the private key for a Certificate Authority, the passwords for an SSH server and for a full-disk encryption utility, and the contents of a differentially private database.

To summarize our evaluation, with various optimizations, invoking the *SKINIT* instruction requires ∼48 ms, while secure storage operations take ∼22 ms, meaning that a typical Flicker session requires ∼70 ms, plus the time needed for the application-specific PAL to execute. Generating a TPM quote for the Flicker session is by far the slowest operation; depending on the TPM, it can take ∼362-756 ms. Fortunately, the quote can be computed in parallel with untrusted code, since it is performed outside of the Flicker environment. In general, Flicker has little impact on the performance of untrusted code, and the OS remains stable. Frequent, brief Flicker sessions have a negligible effect on system performance. Longer running Flicker sessions may produce input lag or dropped network packets, but our experiments indicate that Flicker does not corrupt data transfers (e.g., between USB and disk).

5.2 Suggested Architectural Improvements

Overall, our experiments reveal two significant performance bottlenecks for minimal TCB execution on current CPU architectures: (i) the inability to execute PALs and untrusted code simultaneously on different cores of a multi-core machine, and (ii) the use of slow TPM operations to protect PAL state during a context switch between secure and insecure execution.

To alleviate these issues, we make a number of hardware recommendations [15, 19, 22] to support an execution model like that shown in Figure 3. The key recommendations are for (1) a hardware mechanism for memory isolation that isolates the memory pages belonging to a PAL from all other code, and (2) a hardware context switch mechanism that can efficiently suspend and resume PALs, without exposing a PAL's execution state to other code.

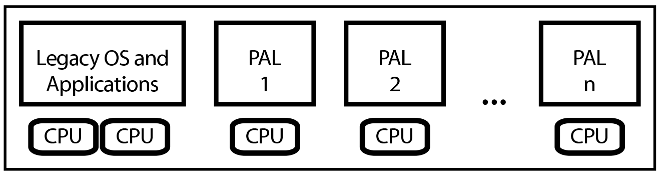


Figure 3: Goal of Our Architectural Recommendations. ***Physical platform running a legacy OS and applications along with some number of*** **PAL*s*.**

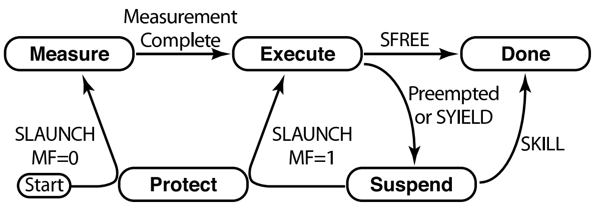


Figure 4: Life Cycle of a PAL. ***MF stands for*** **Measured Flag*. Note that these states are for illustrative purposes and need not be represented explicitly in the system.***

We define a *Secure Execution Control Block* (SECB) as a structure that holds PAL state and resource allocations, both for the purposes of launching a PAL and for storing the state of a PAL when it is not executing. The PAL and SECB should be contiguous in memory to facilitate memory isolation mechanisms. The SECB entry for allocated memory should consist of a list of physical memory pages allocated to the PAL.

Figure 4 illustrates the new life cycle of a PAL. To begin execution of a PAL described by a newly allocated SECB, we propose the addition of a new CPU instruction, *Secure Launch* (*SLAUNCH*), that takes as its argument the starting physical address of an SECB. This launch mechanism combines the hardware virtual machine management data structures on AMD and Intel with the security functionality of a DRTM operation.

To isolate memory and facilitate rapid context switches, we propose that the memory controller maintain an access control table with one entry per physical page, where each entry specifies which CPUs (if any) have access to the physical page. Memory pages are by default marked `ALL` to indicate that they are accessible by all CPUs and DMA-capable devices.

When PAL execution is started using *SLAUNCH*, the memory controller updates its access control table so that each page allocated to the PAL (as specified by the list of memory pages in the SECB) is accessible only to the CPU executing the PAL. When the PAL is subsequently suspended, the state of its memory pages transitions to `NONE`, indicating that nothing currently executing on the platform is allowed to read or write to those pages. Note that the memory allocated to a PAL includes space for data, and is a superset of the pages containing the PAL binary. Since this data is ephemeral, the PAL must take steps, such as utilizing the TPM's secure storage facilities, to ensure that its data persists across reboots [23].

When PAL execution terminates, a well-behaved PAL calls *SFREE*, which clears the memory allocated to the PAL, as well as any microarchitectural state that might contain sensitive data, and marks the memory pages with `ALL`. Similarly, the untrusted OS may kill a PAL via *SKILL*, which clears the PAL's state before freeing the associated memory pages.

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

The untrusted OS can resume a PAL by executing an *SLAUNCH* on the desired CPU, parameterized with the physical address of the PAL's SECB. The PAL's *Measured Flag* indicates to the CPU that the PAL has already been measured and is only being resumed, not started for the first time. Note that the *Measured Flag* is honored only if the SECB's memory page is set to `NONE`. This prevents the untrusted OS from invoking a PAL unless it has been measured by *SLAUNCH*. During PAL resume, the *SLAUNCH* instruction will signal the memory controller that the PAL's state should be accessible to the CPU on which the PAL is now executing. Note that the PAL may execute on a different CPU each time it is resumed. To enable multi-core PALs, if a CPU attempts to resume an already executing PAL, that CPU will be added to the pool of CPUs available to the PAL.

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

In summary, Flicker provides a solid foundation for constructing secure systems that operate in conjunction with standard software; the developer of a security-sensitive code module need only trust her own code, plus as few as 250 lines of Flicker code, for the secrecy and integrity of her code's execution. Flicker guarantees these properties even if the BIOS, OS, and DMA-enabled devices are all malicious. Our current implementation offers reasonable performance for many applications, and our hardware recommendations would enable many more.

6. LEVERAGING SECURE CODE EXECUTION IN NETWORK PROTOCOLS

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

For instance, suppose a mail server wants to improve the accuracy of its spam identification using host-based information. A recent study indicates that the average and standard deviation of the size of emails sent in the last 24 hours are two of the best indicators of whether any given email is spam [10]. This data is easy for an endhost to collect, but hard for any single mail recipient to obtain.

Thus, as an initial exploration of how endhost hardware security features can be used to improve the network, we design a general architecture named Assayer [22, 25]. While Assayer may not represent the optimal way to convey this information, we see it as a valuable first step to highlight the various issues involved. For example, can we provide useful host-based information while also protecting user privacy? Which cryptographic primitives are needed to verify this information in a secure and efficient manner? Our initial findings suggest that improved endhost security can improve the security and efficiency of the network, while simultaneously reducing the complexity of in-network elements.

Naively, we might simply use hardware-based attestation (Section 2) for each piece of information sent to the network. Returning to our earlier example, the mail server might ask each client to

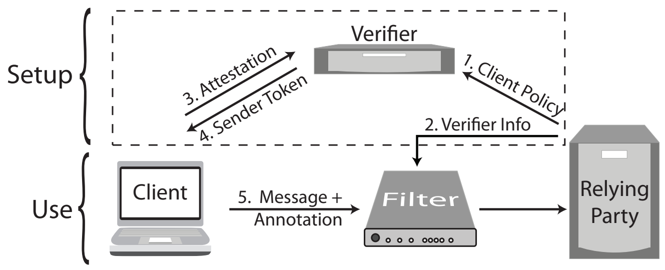


Figure 5: Assayer Components. ***The relying party (e.g., a mail server or an ISP) delegates the task of inspecting clients to one or more verifiers. It also configures one or more filters with information about the verifiers. The filters check the client's annotation and act on the information in the annotation. For example, a filter might drop the message or forward it at a higher priority to the relying party.***

include an attestation in every email. The mail server's spam filter could verify the attestation and then incorporate the host-provided information into its classification algorithm. Any legacy traffic arriving without an attestation could simply be processed by the existing algorithms. Unfortunately, checking attestations is time-consuming and requires interaction with the client. Even if this were feasible for an email filter, it would be unacceptable for other applications, such as Distributed Denial-of-Service (DDoS) mitigation, which require per-packet checks at line rates.

Similarly, labeling outbound traffic with mandatory access control labels, as proposed by McCune et al. [16], works well in tightly controlled enterprise networks, but is unsuited to the heterogeneity of the Internet.

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

As a result, the Assayer architecture employs two distinct phases: an infrequent setup phase in which the *relying party* (e.g., the mail server) establishes trust in the client, and the more frequent usage phase in which the client generates authenticated annotations on outbound messages (Figure 5).

The relying party delegates the task of inspecting clients to one or more off-path *verifier* machines. Every T days, the client convinces a verifier that it has securely installed a trustworthy code *module* that will keep track of network-relevant information, such as the number of emails recently sent, or the amount of bandwidth recently used.

The various code modules execute atop a minimal hypervisor that implements Flicker-like functionality with performance closer to what Flicker would provide with the hardware changes suggested in Section 5.2. To protect user privacy, code modules do not have visibility into the client's software state (e.g., a client module for web browsing cannot determine which web browser the client is using). Instead, we employ application-specific incentives (e.g., by giving priority to traffic carrying annotations) to convince the commodity software to submit outbound traffic to the client modules.

Having established the trustworthiness of the client, the verifier issues a limited-duration *Sender Token* that is bound to the client's code module. During the usage phase, the client submits outbound messages to its code module, which uses the Sender Token to authenticate the message annotations it generates. These annotations are then checked by one or more fast-path *filter* middleboxes, which verify the annotations and react accordingly. For instance, a relying party trying to identify spam might feed the authenticated information from the filter into its existing spam classification algorithms. Alternatively, a web service might contract with its ISP to deploy filters on its ingress links to mitigate DDoS attacks by prioritizing legitimate traffic. If the traffic does not contain annotations, then the filter treats it as legacy traffic (e.g., DDoS filters give annotated traffic priority over legacy traffic). To instantiate the sender tokens and message annotations, we designed two protocols: an efficient symmetric-key-based protocol, and a less efficient asymmetric-key-based protocol that offers additional security properties.

To evaluate the usefulness of trustworthy host-based information, we consider the application of Assayer to three case studies: spam identification, DDoS mitigation, and super-spreader worm detection. We find that Assayer is well-suited to aid in combating spam and can mitigate many (though certainly not all) network-level DDoS attacks. In these two applications, Assayer can be deployed incrementally, since victims (e.g., email hosts or DDoS victims) can deploy Assayer filters in conjunction with existing defenses. Legitimate senders who install Assayer will then see improved performance (e.g., fewer emails marked as spam, or higher success in reaching a server under DDoS attack). Legacy traffic is not dropped but is processed at a lower priority, encouraging, but not requiring, additional legitimate senders to install Assayer. Surprisingly, we find that while it is technically feasible to use Assayer to combat super-spreader worms, such use would face challenges when it comes to deployment incentives. Specifically, non-annotated traffic must be significantly delayed or dropped to have a credible chance of slowing or stopping worm propagation. However, in a legacy environment, ISPs cannot slow or drop legacy traffic until most users have started annotating their traffic, but users will not annotate their traffic unless motivated to do so by the ISPs. A non-technical approach would be to hold users liable for any damage done by non-annotated packets, thus incentivizing legitimate users to annotate their packets. This obviously raises both legal and technical issues.

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

7. SECURE CODE EXECUTION DESPITE UNTRUSTED SOFTWARE & HARDWARE

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

While various forms of homomorphic encryption can provide data secrecy [8, 32], we demonstrate that we can efficiently *verify* the results of arbitrary tasks (abstracted as function evaluations) on a computational service (e.g., in the cloud) without trusting *any* hardware or software on that system [7, 22]. This contrasts with previous approaches that were inefficient or that could only verify the results of restricted function families.

To formalize secure computational outsourcing, we introduce the notion of *verifiable computing* [7, 22]. Abstractly, a client wishes to evaluate a function F (e.g., sign a document or manipulate a photograph) over various, dynamically selected inputs $x_1, \ldots, x_k$ on one or more untrusted computers, and then verify that the values returned are indeed the result of applying F to the given inputs. The critical requirement, which precludes the use of previous solutions, is that the client's effort to generate and verify work instances must be *substantially less* than that required to perform the computation on her own.

Our definition is non-interactive: the client sends a single message to the worker and vice versa. Our definition also uses an amortized notion of complexity for the client: she can perform some expensive pre-processing, but after this stage, she is required to run very efficiently. By introducing a one-time preprocessing stage (and the resulting amortized notion of complexity), we circumvent the result of Rothblum and Vadhan [27], which indicated that efficient verifiable computation requires the use of probabilistically checkable proof (PCP) constructions. In other words, unless a substantial improvement in the efficiency of PCP constructions is achieved, our model allows much simpler and more efficient constructions.

Drawing on techniques from multi-party secure computation, we present the first protocol for verifiable computing. It provably provides computational integrity for work done by an untrusted party; it also provides provable secrecy for the computation's inputs and outputs. This privacy feature is bundled into the protocol and comes at no additional cost. This is a critical feature for many real-life outsourcing scenarios in which a function is computed over highly sensitive data (e.g., medical records or trade secrets).

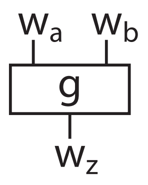
Moreover, the protocol provides asymptotically optimal performance (amortized over multiple inputs). Specifically, the protocol requires a one-time pre-processing stage which takes $O(|C|)$ time, where C is the smallest known Boolean circuit computing F. For each work instance, the client performs $O(|m|)$ work to prepare an m-bit input, the worker performs $O(|C|)$ work to compute the results, and the client performs $O(|n|)$ work to verify the n-bit result.

This result shows that we can outsource arbitrary computations to untrusted workers, preserve the secrecy of the data, and efficiently verify that the computations were done correctly. Thus, verifiable computing could be used, for instance, to outsource work to a cloud-based service, or in a distributed computing project like Folding@home [20], which outsources protein-folding simulations to millions of Internet users. To prevent cheating, these projects often assign the same work unit to multiple clients and compare the results; verifiable computing would eliminate these redundant computations and provide strong cryptographic protections against colluding workers, though there are some subtleties involved when workers are caught cheating [7, 22].

Our construction is based on the crucial (and somewhat surprising) observation that Yao's Garbled Circuit Construction [35], in addition to providing secure two-party computation, also provides a "one-time" verifiable computation. In other words, we can adapt Yao's construction to allow a client to outsource the computation of a function on a single input. Figure 6 shows an example with a single gate, but the construction extends naturally to circuit sizes polynomial in the security parameter.

At a high level, in the preprocessing stage the client garbles the circuit C according to Yao's construction. Then in the "input preparation" stage, the client reveals the random labels associated with the input bits of x in the garbling. This allows the worker to compute the random labels associated with the output bits, and from them the client will reconstruct $F(x)$. If the bit labels are sufficiently long and random, the worker will not be able to guess the labels for an incorrect output, and therefore the client is assured that $F(x)$ is the correct output.

Unfortunately, reusing the circuit for a second input x' is insecure, since once the output labels of $F(x)$ are revealed, nothing can stop the



(a)

Wa	Wb	Wz
0	0	g(0,0)
0	1	g(0,1)
1	0	g(1,0)
1	1	g(1,1)

(b)

Wa	Wb	Wz
k_a^0	k_b^0	$k_z^{g(0,0)}$
k_a^0	k_b^1	$k_z^{g(0,1)}$
k_a^1	k_b^0	$k_z^{g(1,0)}$
k_a^1	k_b^1	$k_z^{g(1,1)}$

(c)

Wa	Wb	Wz
k_a^0	k_b^0	$E_{k_a^0}(E_{k_b^0}(k_z^{g(0,0)}))$
k_a^0	k_b^1	$E_{k_a^0}(E_{k_b^1}(k_z^{g(0,1)}))$
k_a^1	k_b^0	$E_{k_a^1}(E_{k_b^0}(k_z^{g(1,0)}))$
k_a^1	k_b^1	$E_{k_a^1}(E_{k_b^1}(k_z^{g(1,1)}))$

(d)

Figure 6: Yao's Garbled Circuits for Verifiable Computation. ***The original binary gate* (a) *can be represented by a standard truth table* (b). *We then replace the 0 and 1 values with the corresponding randomly chosen λ-bit values* (c). *Finally, we use the values for w_a and w_b to encrypt the values for the output wire w_z* (d). *The random permutation of these ciphertexts is the garbled representation of gate g, which will be shipped to the worker. To compute $g(10)$, the client sends the worker (k_a^1, k_b^0). The worker tries to use these values to decrypt all four ciphertexts. This will succeed only for the third ciphertext, yielding $\hat{y} \leftarrow k_z^{g(1,0)}$, which the worker returns to the client. If $\hat{z} = k_z^0$, the client concludes the output is $z = 0$, else if $\hat{z} = k_z^1$, $z = 1$. Otherwise, if $\hat{z} \neq k_z^0$ and $\hat{z} \neq k_z^1$, then the client concludes the worker is cheating.***

worker from presenting those labels as correct for $F(x')$. Creating a new garbled circuit requires as much work as if the client computed the function itself, so on its own, Yao's Circuits do not provide an efficient method for outsourcing computation.

The second crucial idea is to combine Yao's Garbled Circuit with a fully homomorphic encryption system (e.g., Gentry's recent proposal [8]) to be able to safely reuse the garbled circuit for multiple inputs. More specifically, instead of revealing the labels associated with the bits of input x, the client will encrypt those labels under the public key of a fully homomorphic scheme. A new public key is generated for every input in order to prevent information from one execution from being useful for later executions. The worker then uses the homomorphic property to compute an encryption of the output labels and provide them to the client, who decrypts them and reconstructs $F(x)$.

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

8. CONCLUSION

Motivated by the trend of entrusting sensitive data and services to insecure computers, we develop techniques that allow a user to extend her trust in one device to another device or service. Thus, starting from trust in a simple USB device, we enable a user to trust the reports from secure hardware on her computer, and then securely execute code on that computer, despite the presence of a mountain of untrustworthy code; the untrusted code continues to enjoy the performance and features it does today. Efficiently extending the trust in this secure environment into the network improves the security of a range of network protocols, including spam detection and DDoS mitigation. Finally, we extend the user's trust to the verification of computations performed by remote computers, even though they may be arbitrarily malicious.

Building on these techniques, we aim to enable average computer users to easily and securely use their computers to perform sensitive tasks (e.g., paying bills, shopping online, or accessing medical records), while still retaining the flexibility and performance expected of modern computers. For example, we are currently designing systems that use Flicker-like protections to provide personalized online services, such as location-based services, targeted advertising, or personalized search, while still preserving user privacy. We are also developing new application security models, so that opening an email attachment or clicking on a web link will not endanger the user's computer; as part of this design, applications will have only the minimum amount of access to data and resources necessary to act on user requests, without relying on the manifests and prompts used on modern smartphones.

In short, today many people who use computers and online services operate under the illusion that their data and privacy will be protected. In the long run, the ability to bootstrap trust in these computers and services will help replace this illusion with the actual foundation of security that users expect and deserve.

Acknowledgements

This article is based on my Ph.D. dissertation [22], though much of the research it describes was conducted in collaboration with my advisor Adrian Perrig at Carnegie Mellon University, as well as with Hiroshi Isozaki, Jonathan McCune, Mike Reiter, and Arvind Seshadri [17–19]; Zongwei Zhou [25]; and Craig Gentry and Rosario Gennaro [7]. I am extremely grateful for their contributions. This article benefited from feedback from Jay Lorch, Jonathan McCune, Diana Parno, and Adrian Perrig.

References

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

[2] S. R. Ames, Jr. Security kernels: A solution or a problem? In *Proc. of the IEEE Symposium on Security and Privacy*, 1981.

[3] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the IEEE/ACM Workshop on Grid Computing*, Nov. 2004.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, E. Kotsovinos, A. Madhavapeddy, R. Neugebauer, I. Pratt, and A. Warfield. Xen 2002. Technical Report UCAM-CL-TR-553, University of Cambridge, Jan. 2003.

[5] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of ACM ASPLOS*, Mar. 2008.

[6] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Trustworthy and personalized computing on public kiosks. In *Proceedings of the Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2008.

[7] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computation: Outsourcing computation to untrusted workers. In *Proceedings of CRYPTO*, Aug. 2010.

[8] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of ACM STOC*, 2009.

[9] B. D. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in retrospect. In *Proc. of IEEE Symp. on Security & Privacy*, 1984.

[10] S. Hao, N. A. Syed, N. Feamster, A. G. Gray, and S. Krasser. Detecting spammers with SNARE: Spatio-temporal network-level automatic reputation engine. In *Proceedings of the USENIX Security Symposium*, 2009.

[11] Intel Corporation. Intel trusted execution technology – measured launched environment developer's guide. Document number 315168-005, June 2008.

[12] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, Nov. 1991.

[13] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[14] D. Magenheimer. Xen/IA64 code size stats. Xen developer's mailing list: http://lists.xensource.com/, 2005.

[15] J. M. McCune. *Reducing the Trusted Computing Base for Applications on Commodity Systems*. PhD thesis, Carnegie Mellon University, Jan. 2009.

[16] J. M. McCune, S. Berger, R. Cáceres, T. Jaeger, and R. Sailer. Shamon: A system for distributed mandatory access control. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2006.

[17] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of ACM EuroSys*, Apr. 2008.

[18] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. Minimal TCB code execution (extended abstract). In *Proc. of the IEEE Symposium on Security & Privacy*, 2007.

[19] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. How low can you go? Recommendations for hardware-supported minimal TCB code execution. In *Proceedings of ACM ASPLOS*, Mar. 2008.

[20] Pande Lab. The folding@home project. Stanford University, http://folding.stanford.edu/.

[21] B. Parno. Bootstrapping trust in a "trusted" platform. In *USENIX Workshop on Hot Topics in Security*, July 2008.

[22] B. Parno. *Trust Extension as a Mechanism for Secure Code Execution on Commodity Computers*. PhD thesis, Carnegie Mellon University, May 2010.

[23] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Proc. of IEEE Symposium on Security & Privacy*, 2011.

[24] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[25] B. Parno, Z. Zhou, and A. Perrig. Help me help you: Using trustworthy host-based information in the network. Technical Report CMU-CyLab-09-016, Carnegie Mellon University, Cylab, Nov. 2009.

[26] C. Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005.

[27] G. Rothblum and S. Vadhan. Are PCPs inherent in efficient arguments? In *Proc. of Computational Complexity*, 2009.

[28] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proc. of USENIX Security Symposium*, 2004.

[29] L. Singaravelu, C. Pu, H. Haertig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of ACM EuroSys*, 2006.

[30] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8), Apr. 1999.

[31] Trusted Computing Group. Trusted Platform Module Main Specification. Version 1.2, Revision 103, 2007.

[32] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proceedings of EuroCrypt*, June 2010.

[33] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig. Lockdown: A safe and practical environment for security applications. Technical Report CMU-CyLab-09-011, Carnegie Mellon University, Cylab, July 2009.

[34] Wave Systems Corp. Trusted Computing: An already deployed, cost effective, ISO standard, highly secure solution for improving Cybersecurity. http://www.nist.gov/itl/upload/Wave-Systems_Cybersecurity-NOI-Comments_9-13-10.pdf, 2010.

[35] A. Yao. Protocols for secure computations. In *Proc. of the IEEE Symposium on Foundations of Computer Science*, 1982.