An Execution Infrastructure for TCB Minimization

Jonathan M. McCune, Bryan Parno, Adrian Perrig,
Michael K. Reiter, Hiroshi Isozaki

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

An Execution Infrastructure for TCB Minimization

Jonathan M. McCune[*] Bryan Parno[*] Adrian Perrig[*] Michael K. Reiter[*][†] Hiroshi Isozaki[‡]

[*] Carnegie Mellon University

[†] University of North Carolina at Chapel Hill

[‡] Toshiba Corporation

December 18, 2007

Abstract

We present Flicker, an infrastructure for executing security-sensitive code in complete isolation while trusting as few as 250 lines of additional code. Flicker can also provide meaningful, fine-grained attestation of the code executed (as well as its inputs and outputs) to a remote party. Flicker guarantees these properties even if the BIOS, OS and DMA-enabled devices are all malicious. Flicker leverages new commodity processors from AMD and Intel and does not require a new OS or VMM. We demonstrate a full implementation of Flicker on an AMD platform and describe our development environment for simplifying the construction of Flicker-enabled code.

1 Introduction

Today's popular operating systems run a daunting amount of code in the CPU's most privileged mode. The plethora of vulnerabilities in this code makes the compromise of systems commonplace, and its privileged status is inherited by the malware that invades it. The integrity and secrecy of every application is at risk in such an environment.

To address these problems, we propose Flicker, an architecture for isolating sensitive code execution using a minimal Trusted Computing Base (TCB). 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 execution resumes. For example, a Certificate Authority (CA) could sign certificates with its private key, even while keeping the key secret from an adversary that controls the BIOS, OS, and DMA-enabled devices. Flicker can operate at any time and does not require a new OS or even a VMM, so the user's platform for non-sensitive operations remains unchanged.

Flicker provides strong isolation guarantees while requiring the application to trust as few as 250 additional lines of code for its secrecy and integrity. As a result, Flicker circumvents entire layers of legacy system software and eliminates reliance on their correctness for security properties (see Figure 1). Once the TCB for code execution has been precisely defined and limited, formal assurance of both reliability and security properties enters the realm of possibility.

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

To achieve these properties, Flicker utilizes hardware support for late launch and attestation recently introduced in commodity processors from AMD and Intel. These processors already ship with off-the-shelf computers and will soon become ubiquitous. Although current hardware still has a high overhead, we anticipate that future hardware performance will improve as these functions are increasingly used. Furthermore, many applications perform security-sensitive operations where speed is not the first priority.
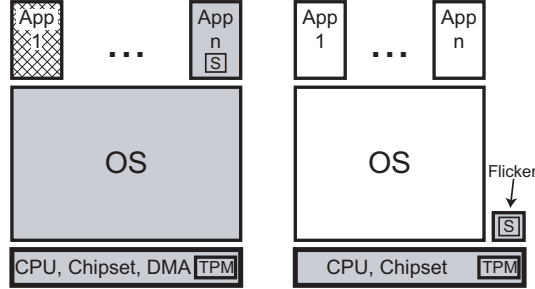
1

Figure 1: *On the left, a traditional computer with an application that executes sensitive code (S). On the right, Flicker protects the execution of the sensitive code. The shaded portions represent components that must be trusted; other applications are included on the left because many applications run with superuser privileges.*

From a programmer's perspective, the sensitive code protected by Flicker can be written from scratch or extracted from an existing program. To simplify this task, the programmer can draw on a collection of small code modules we have developed for common functions, and can apply tools we have developed to extract sensitive operations and relevant code from an application.

We present an implementation of Flicker using AMD's SVM technology and use it to improve the security of a variety of applications. We develop a rootkit detector that an administrator can run on a remote machine and receive a guarantee that the detector executed correctly and returned the correct result. We also show how Flicker can improve the integrity of results for distributed computing projects. Finally, we use Flicker to protect a CA's private signing key and to improve an SSH server's password-handling procedure.

2 Background

In this section, we provide background information on the hardware technologies leveraged by Flicker.

2.1 TPM-Based Attestation

A computing platform containing a Trusted Platform Module (TPM) can provide an *attestation* or *quote*—essentially a signature on the current platform state—to an external entity. The platform state is detailed in a log of software events, such as applications started or configuration files used. Each event is reduced to a *measurement*, m, using a cryptographic hash function, H. The hash value is stored in one of the TPM's Platform Configuration Registers (PCRs) by extending a particular PCR's current value, v_t, i.e., the PCR's value is updated as $v_{t+1} \leftarrow H(v_t || m)$, where $||$ denotes concatenation. The TPM can sign the values of the PCRs, effectively signing the event log.

To sign its PCRs, the TPM uses the private portion of an Attestation Identity Key (AIK) pair. The AIK pair is generated by the TPM, and the private AIK never leaves the TPM in a readable state. A certificate from a Privacy CA attests that the AIK corresponds to an AIK generated by a legitimate TPM.

Attestation allows an external party (or *verifier*) to make a trust decision based on the platform's software state. The verifier authenticates the public AIK by validating the AIK's certificate chain and deciding whether to trust the issuing Privacy CA. It then validates the signature on the PCR values and checks that the PCR values correspond to the events in the log by hashing the log entries and comparing the results to the PCR values in the attestation. Finally, it decides whether to trust the platform based on the events in the log. Typically, the verifier must assess a list of all software loaded since boot time (including the OS) and its configuration information.

2.2 TPM-Based Sealed Storage

TPMs also provide sealed storage, whereby some amount of data can be encrypted using an asymmetric key whose private component never leaves the TPM in unencrypted form. The sealed data can be bound to a

particular software state, as defined by the contents of various PCRs. The TPM will only unseal (decrypt) the data when the PCRs contain the values specified by the seal command.

2.3 TPM 1.2 Dynamic PCRs

The TPM v1.2 specification [30] allows for *static* and *dynamic* PCRs. Only a system reboot can reset the value in a static PCR, but under the proper conditions, the dynamic PCRs 17–23 can be reset to zero without a reboot (a reboot sets the value of PCRs 17–23 to -1, so that a remote verifier can distinguish between a reboot and a dynamic reset). Only a hardware command from the CPU can reset PCR 17, and the CPU will issue this command only after receiving an *SKINIT* instruction as described below. Thus, software cannot reset PCR 17, though PCR 17 can be read and extended by software before or after *SKINIT* has completed.

2.4 Late Launch

In this section, we discuss the capabilities offered by AMD's Secure Virtual Machine (SVM) extensions [1]. Intel offers similar capabilities with their Trusted eXecution Technology (TXT, formerly LaGrande Technology (LT)) [12]. Both AMD and Intel are shipping chips with these capabilities; they can be purchased in commodity computers. In this paper, we will focus on AMD's SVM technology, but Intel's TXT technology functions analogously.

SVM chips are designed to allow the *late launch* of a Virtual Machine Monitor (VMM) or Security Kernel at an arbitrary time with built-in protection against software-based attacks. To launch the VMM, software in CPU protection ring 0 (e.g., kernel-level code) invokes the new *SKINIT* instruction, which takes a physical memory address as its only argument. The memory at this address is known as the Secure Loader Block (SLB). The first two words (16-bit values) of the SLB are defined to be its length and entry point (both must be between 0 and 64 KB).

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

SVM also includes support for attesting to the proper invocation of the SLB. As part of the *SKINIT* instruction, the processor first causes the TPM to reset the values of PCRs 17–23 to zero, and then transmits the (up to 64 KB) contents of the SLB to the TPM so that it can be measured (hashed) and extended into PCR 17. Note that software cannot reset PCR 17 without executing another *SKINIT* instruction. Thus, future TPM attestations can include the value of PCR 17 and hence attest to the use of the *SKINIT* instruction and the identity of the SLB loaded.

3 Problem Definition

3.1 Adversary Model

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

At the hardware level, we make the same assumptions as does the Trusted Computing Group with regard to the TPM [31]. In essence, the attacker can launch simple hardware attacks, such as opening the case, power cycling the computer, or attaching a hardware debugger. The attacker can also compromise add-on hardware such as a DMA-capable Ethernet card with access to the PCI bus. However, the attacker cannot launch sophisticated hardware attacks, such as monitoring the high-speed bus linking the CPU and memory.

3.2 Goals

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

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

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

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

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

On their own, AMD's SVM and Intel's TXT technologies only meet two of the above goals. While both provide Isolation and Provable Protection, they were both designed with the intention that the *SKINIT* instruction would be used to launch a secure kernel or secure VMM [10]. Either mechanism will significantly increase the size of an application's TCB and dilute the meaning of future attestations. For example, a system using the Xen [5] hypervisor with *SKINIT* would add almost 83,000 lines of code [17] to an application's TCB, not including the Domain 0 OS, which potentially adds millions of additional lines of code to the TCB.

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

4 Flicker Architecture

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

4.1 Flicker Overview

Flicker achieves its properties using the late launch capabilities described in Section 2.4. Instead of launching a VMM, Flicker pauses the current execution environment (e.g., the untrusted OS), executes a small piece of code using the *SKINIT* instruction, and then resumes operation of the previous execution environment. The security-sensitive code selected for Flicker protection is the Chunk Of Application Logic (COAL). The protected environment of a Flicker session starts with the execution of *SKINIT* and ends with the resumption of the previous execution environment. Figure 2 illustrates this sequence.

Application developers must provide the COAL and define its interface with the remainder of their application (we discuss this process, as well as our work on automating it, in Section 5). To create an SLB (the Secure Loader Block supplied as an argument to *SKINIT*), the application developer links her COAL against an uninitialized code module we have developed called the SLB Core. The SLB Core performs the steps necessary to set up and tear down the Flicker session. Figure 3 shows the SLB's memory layout.

To execute the resulting SLB, the application passes it to a Linux kernel module we have developed, *flicker-module*. It initializes the SLB Core and handles untrusted setup and tear-down operations. The *flicker-module* is not included in the TCB of the application, since its actions are verified.
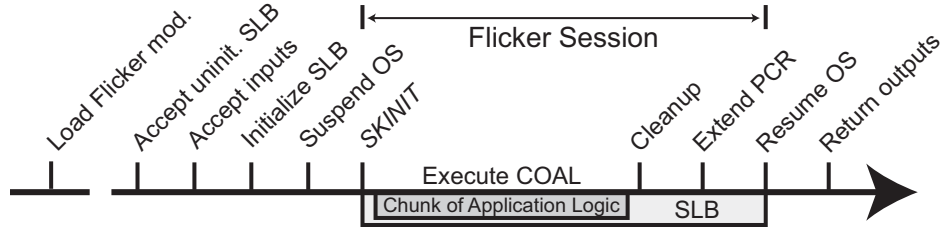
Figure 2: *Timeline showing the steps necessary to execute a* COAL*. The SLB includes the* COAL*, as well as the code necessary to initialize and terminate the Flicker session. The gap in the time axis indicates that the flicker-module is only loaded once.*

4.2 Isolated Execution

We provide a simplified discussion of the operation of a Flicker session by following the timeline in Figure 2. **Accept Uninitialized SLB and Inputs**. Because *SKINIT* is a privileged instruction, an application uses the *flicker-module*'s interface to invoke a Flicker session. In the sysfs,[1] the *flicker-module* makes four entries available: `control`, `inputs`, `outputs`, and `slb`. Applications interact with the *flicker-module* via these filesystem entries. An application first writes to the `slb` entry an uninitialized SLB containing its COAL code. The *flicker-module* allocates kernel memory in which to store the SLB; we refer to the physical address at which it is allocated as `slb_base`. The application writes any inputs for its COAL to the `inputs` sysfs entry; the inputs are made available at a well-known address once execution of the COAL begins (the parameters are at the top of Figure 3). The application initiates the Flicker session by writing to the `control` entry in the sysfs.

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

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

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

***SKINIT* and the SLB Core**. The *SKINIT* instruction enables hardware protections and then begins to execute the SLB Core, which prepares the environment for COAL execution. Executing *SKINIT* enables the hardware protections described in Section 2.4. In brief, the processor adds entries to the Device Exclusion

---

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

Additional COAL Code
(Optional)

Parameters
Out: COAL Outputs

In: COAL Inputs
In: Saved Kernel State

Stack Space (4 KB)
(Grows Towards Low Addresses)

Page Tables
(During Legacy OS Reload Only)

COAL

Init()   Exit()
Task State Segment (TSS)
Global Descriptor Table (GDT)
Size of SLB   Entry point

Secure Loader Block (SLB)

SLB Core

End + 8 KB

End + 4 KB

End of SLB
(Start + 64KB)

End of COAL
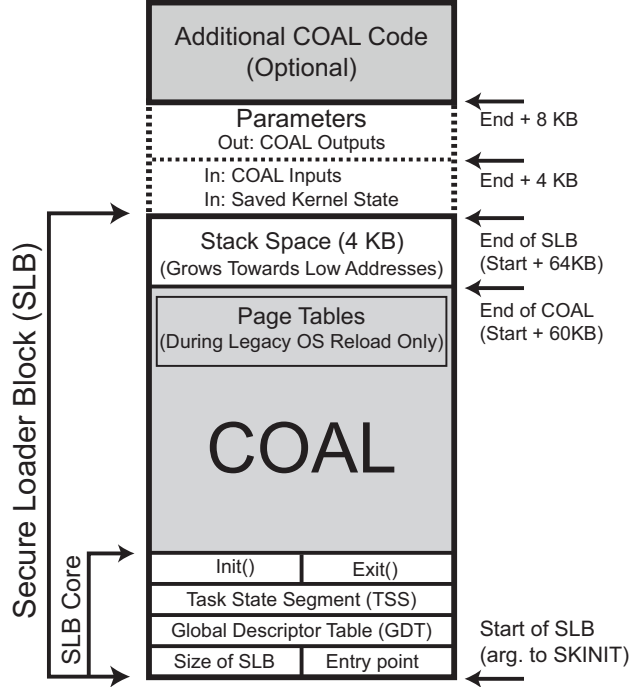(Start + 60KB)

Start of SLB
(arg. to SKINIT)

Figure 3: *Memory layout of the SLB. The shaded region indicates memory containing executable* COAL *code. The dotted lines indicates memory used to transfer data into and out of the SLB. After the* COAL *has executed and erased its secrets, memory that previously contained executable code is used for the skeleton page tables needed to reload the OS.*

Vector (DEV) to disable DMA to the memory region containing the SLB, disables interrupts to prevent the previously executing code from regaining control, and disables debugging support, even for hardware debuggers. By default, these protections are offered to 64 KB of memory, but they can be extended to larger memory regions. If this is done, preparatory code in the first 64 KB must add this additional memory to the DEV, and extend measurements of this additional memory into the TPM's PCR 17 after the hardware protections are enabled, but before transferring control to any code in these upper memory regions.

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

Execute COAL. Once the environment has been prepared, the COAL executes its application-specific logic. To keep the TCB small, the default SLB Core includes no support for heaps, memory management, or virtual memory. Thus, it is up to the COAL developer to include the functionality necessary for her particular application. Section 5 describes some of our existing modules that can optionally be included to provide additional functionality. During COAL execution, output parameters are written to a well-known location beyond the end of the SLB. When the COAL exits, the SLB Core regains control.

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

Extend PCR. To signal the completion of the SLB, the SLB Core extends a well known value into PCR 17. As we discuss in Section 4.4.1, this allows a remote party to distinguish between values generated by the COAL (trusted), and those produced after the OS resumes (untrusted).

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

We use a call gate in the SLB Core's GDT as a well-known point for resuming the untrusted OS. It is

used to reload the code segment descriptor register with a descriptor covering all of memory.

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

The *flicker-module* restores the execution state saved during the Suspend OS phase and fully restores control to the Linux kernel by re-enabling interrupts. If the COAL outputs any values, the *flicker-module* makes them available through the sysfs `outputs` entry.

4.3 Multiple Flicker Sessions

COALs can leverage TPM-based sealed storage to maintain state across Flicker sessions, enabling more complex applications. For example, a Flicker-based application may wish to interact with a remote entity over the network. Rather than include an entire network stack and device driver in the COAL (and hence the TCB), we can invoke Flicker more than once (upon the arrival of each message), using secure storage to protect sensitive state between invocations. Flicker-based secure storage can also be used by applications that wish to share data between COALs. The first COAL can store secrets so that only the second COAL can read them, thus protecting the secrets even when control reverts to the untrusted OS. Finally, Flicker-based secure storage can improve the performance of long-running COAL jobs. Since Flicker execution pauses the rest of the system, an application may prefer to break up a long work segment into multiple Flicker sessions to allow the rest of the system time to operate, essentially multitasking with the OS.

To save state across Flicker sessions, a COAL uses the TPM to seal the data under the measurement of the COAL that should have access to its secrets. More precisely, suppose COAL P, operating in a Flicker session, wishes to securely store data so that only COAL P', also operating under Flicker protection, can read the data.[2] P' could be a later invocation of P, or it could be a completely different COAL. Either way, while it is executing within the Flicker session, COAL P uses the TPM's Seal command to secure the sensitive data. As an argument, P specifies that PCR 17 must have the value $V = H(\text{0x00}^{20}||H(P'))$ before the data can be unsealed. Only an *SKINIT* instruction can reset the value of PCR 17, so PCR 17 will have value V only after COAL P' has been invoked using *SKINIT*. Thus, the sealed data can be unsealed if and only if P' executes under Flicker's protection. This allows COAL code to store persistent data such that it is only available to a particular COAL in a future Flicker session.

TPM-based sealed storage prevents other code from directly learning or modifying a COAL's secrets. However, TPM Seal outputs ciphertext c (for data d) that is handled by untrusted code: $c = TPM\_Seal(d)$. The untrusted code is capable of performing a replay attack where an older ciphertext c' is provided to a COAL. TPM Unseal ensures that c' is only accessible to the intended COAL, but it does not guarantee that c' is the most recent sealed version of data d. In fact, c is more recent in this example. This problem is not unique to Flicker; it applies anywhere that TPM-based sealed storage is used. TPM v1.2 provides a *monotonic counter* function that can be used to version instances of a particular data item in sealed storage.

4.4 Interaction With a Remote Party

A remote party may need to verify the execution of Flicker. Below, we describe how a platform attests to the COAL executed, the use of Flicker, and any inputs or outputs provided. We also demonstrate how a remote party can establish a secure channel to a COAL.

4.4.1 Attestation and Result Integrity

A platform using Flicker can convince remote parties that a Flicker session executed with a particular COAL. Our approach builds on the TPM attestation process described in Section 2.1. Below, we refer to the party

---

[2]For brevity, we will assume that COALs operate with Flicker protection. Similarly, a measurement of the COAL consists of a hash of the SLB containing the COAL.

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

Figure 4: *A simple* COAL *that ignores its inputs, and outputs "Hello, world."* `COAL_OUT` *is defined in* `slbcore.h`.

executing Flicker as the *challenged party*, and the remote party as the *verifier*.

To create an attestation, the challenged party accepts a random nonce from the verifier to provide freshness and replay protection. The challenged party then uses Flicker to execute a particular COAL as described in Section 4.2. As part of Flicker's execution, the *SKINIT* instruction resets the value of PCR 17 to 0 and then extends it with the measurement of the COAL. Thus, PCR 17 will take on the value $V = H(0x00^{20}||H(P))$, where P represents the COAL 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 COAL was executed using Flicker's protection.

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

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

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

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

4.4.2   Establishing a Secure Channel

The techniques described above ensure the integrity of the COAL's input and output, but to communicate securely (i.e., with both secrecy and integrity protections) with a remote party, the COAL and the remote party must establish a secure channel. We create a secure channel by combining multiple Flicker sessions, the attestation capabilities just described, and some additional cryptographic techniques [18]. Essentially, the COAL generates an asymmetric keypair within the protection of the Flicker session and then transmits the public key to the remote party. The private key is sealed for a future invocation of the same COAL using the technique described above. An attestation convinces the remote party that the COAL ran with Flicker's protections and that the public key was a legitimate output of the COAL. Finally, the remote party can use the COAL's public key to create a secure channel [11] to the COAL.

5   Developer's Perspective

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

Module	Properties	LOC	Size (KB)
SLB Core	Prepare environment, execute COAL, clean environment, resume OS	94	0.312
OS Protection	Memory protection, ring 3 COAL execution	5	0.046
TPM Driver	Communication with the TPM	216	0.825
TPM Utilities	Performs TPM operations, e.g., Seal, Unseal, GetRand, PCR Extend	889	9.427
Crypto	General purpose cryptographic operations, RSA, SHA1, SHA512 etc.	2262	31.380
Memory Management	Implementation of malloc/free/realloc	657	12.511
Secure Channel	Generates a keypair, seals private key, returns public key	292	2.021

Figure 5: *Modules that can be included in the* COAL. *Only the SLB Core is mandatory. Each adds some number of lines of code (LOC) to the* COAL*'s TCB and contributes to the overall size of the SLB binary.*

5.1   Creating a COAL

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

5.1.1   A "Hello, World" Example COAL

As an example, Figure 4 illustrates a simple COAL that ignores its inputs, and outputs the classic message, "Hello, world." Essentially, the COAL copies the contents of the global `msg` variable to the well-known COAL output parameter location (defined in the `slbcore` header file). Our convention is to use the second 4 KB page above the 64 KB SLB. The COAL code, when built using the process described below, can be executed with Flicker protections. Its message will be available from the `outputs` entry in the *flicker-module* sysfs location. Thus the application can simply use `open` and `read` to obtain the COAL's results.

5.1.2   Building a COAL

To convert the code from Figure 4 into a COAL, we link it against the object file representing Flicker's core functionality (described as SLB Core below) using the Flicker linker script. The linker script specifies that the skeleton data structures and code from the SLB Core should come first in the resulting binary, and that the resulting output format should be binary (as opposed to an ELF executable). The application then provides this binary blob to the *flicker-module* for execution under Flicker's protection.

Application developers depend on a variety of libraries. There is no reason this should be any different just because the target executable is a COAL, except that it is desirable to modularize the libraries further than is traditionally done to help minimize the amount of code included in the COAL's TCB. We have developed several small libraries in the course of applying Flicker to the applications described in Section 6. The following paragraphs provide a brief description of the libraries listed in Figure 5.

SLB Core. The SLB Core module provides the minimal functionality needed to support a COAL. Section 4.2 describes this functionality in detail. In brief, the SLB Core contains space for the SLB's entry point, length, GDT, TSS, and code to manage segment descriptors and page tables. The SLB Core transfers control to the COAL code, which performs application-specific work. When the COAL terminates, it transfers control back to the SLB Core for cleanup and resumption of the OS.

OS Protection. Thus far, Flicker has focused on protecting a security-sensitive COAL from all of the other software on the system. However, we have also developed a module to protect a legitimate OS from a malicious or malfunctioning COAL. It is important to note that since *SKINIT* is a privileged instruction, only code executing at CPU protection ring 0 (recall that x86 has 4 privilege rings, with 0 being most privileged) can invoke a Flicker session. Thus, the OS ultimately decides which COALs to run, and presumably it will only run COALs that it trusts or has verified in some manner, e.g., using proof carrying code [20]. Nonetheless, the OS may desire additional guarantees. The OS Protection module restricts a COAL's memory accesses to the exact memory region allocated by the OS, thus preventing it from intentionally or inadvertently reading or overwriting the code and/or data of other software on the system. We are also investigating techniques to limit a COAL's execution time.

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

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

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

The TPM Utilities allow other COAL code to perform useful TPM operations. Currently supported operations include GetCapability, PCR Read, PCR Extend, GetRandom, Seal, Unseal, and the OIAP and OSAP sessions necessary to authorize Seal and Unseal [31].

Crypto. We have developed a small library of cryptographic functions. Supported operations include a multi-precision integer library, RSA key generation, RSA encryption and decryption, SHA-1, SHA-512, MD5, AES, and RC4.

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

Secure Channel. We have implemented the protocol described in Section 4.4 for creating a secure channel into a COAL from a remote party. It relies on all of the other modules we have developed (except the OS Protection module which the developer may add).

5.2 Automation

Ideally, we envision each COAL containing only the security-sensitive portion of each application, rather than the application in its entirety. Minimizing the COAL makes it easier to ensure that the required functionality is performed correctly and securely, facilitating a remote party's verification task. Previous research indicates that many applications can be readily split into a privileged and an unprivileged component. Such privilege separation can be performed manually [14, 16, 22, 29], or automatically [4, 6, 33].

While each COAL is necessarily application-specific, we have developed a tool using the source-code analysis tool CIL [21] to help extract functionality from existing programs. Since CIL can replace the C compiler (e.g., the programmer can simply run "`CC=cil make`" using an existing Makefile), our tool can operate even on large programs with complex build dependencies.

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

6 Applications of Flicker

In this section, we demonstrate the versatility of the Flicker platform by showing how Flicker can be applied to several broad classes of applications. Within each class, we describe our implementation of one or more

applications and show how Flicker significantly enhances security in each case. In Section 7, we evaluate the performance of the applications, as well as the general Flicker platform.

We have implemented Flicker for AMD SVM on a 32-bit Linux kernel v2.6.20, including the various modules described in Section 5. Each application described below utilizes precisely the modules needed (as well as some application-specific logic) and nothing else. On the untrusted OS, the *flicker-module* loadable kernel module is responsible for invoking the COAL and facilitating delivery of inputs and reception of outputs from the Flicker session. Further, it manages the suspension and resumption of the untrusted OS before and after the Flicker session. We also developed a TPM Quote Daemon (the *tqd*) on top of the TrouSerS[3] TCG Software Stack that runs on the untrusted OS and provides an attestation service.

6.1 Non-Stateful Applications

Many applications do not require long-term state to operate effectively. For these applications, the primary overhead of using Flicker is the time required for the *SKINIT* instruction, since the attestation can be generated by the untrusted OS (see Section 4.4.1). As a concrete example, we use Flicker to provide verifiable isolated execution of a kernel rootkit detector on a remote machine.

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

We implement our rootkit detector for version 2.6.20 of the Linux kernel as a COAL. After the SLB Core hands control to the rootkit detector COAL, it computes a SHA-1 hash of the kernel text segment, system call table, and loaded kernel modules. The detector then extends the resulting hash value into PCR 17 and copies it to the standard output memory location. Once the COAL terminates, the untrusted OS resumes operation and the *tqd* provides an attestation to the network administrator. Since the attestation contains the TPM's signature on the current PCR values, the administrator knows that the correct rootkit detector ran with Flicker protections in place and can verify that the untrusted OS returns the correct value. Finally, the administrator can compare the hash value returned against known-good values for that particular kernel.

6.2 Integrity-Protected State

Some applications may require multiple Flicker sessions, and hence a means of preserving state across sessions. For some, simple integrity protection of this state will suffice (we consider those that also require secrecy in Section 6.3). To illustrate this class of applications, we apply Flicker to a distributed computing application.

Applications such as SETI-at-Home [3] divide a task into smaller work units and distribute these units to hosts with spare computation capacity. When the hosts are untrusted, the application must take measures to detect erroneous results. A common approach distributes the same work unit to multiple hosts and compares the results. Unfortunately, this wastes significant amounts of computation, and does not provide any tangible correctness guarantees [19]. With Flicker, the clients can process their work units inside a Flicker session and attest the results to the server. The server then has a high degree of confidence in the results and need not waste computation on redundant work units.

In our implementation, we apply Flicker to the BOINC framework [2], which is a generic framework for distributed computing applications. It is currently used by several dozen projects.[4] By targeting BOINC, rather than a specific application, we potentially allow all of these applications to take advantage of Flicker's security properties (though some amount of application-specific modifications are still required). As an illustration, we developed a simple distributed application that attempts to factor a large number by naively asking clients to test a range of numbers for potential divisors.

In this application, our modified BOINC client contacts the server to obtain a work unit. It then invokes a Flicker session to perform application specific work. Since the COAL may have to compute for an extended period of time, it periodically returns control to the untrusted OS.

Since many distributed computing applications care about the integrity of the result, rather than the secrecy of the intermediate state, our implementation focuses on maintaining the integrity of the COAL's state while the untrusted OS operates. To do so, the very first invocation of the BOINC COAL generates a

---

[3]http://trousers.sourceforge.net/
[4]http://boinc.berkeley.edu/projects.php

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

Figure 6: *The protocol surrounding the second Flicker session for our SSH implementation. sdata contains the sealed private key, K_{COAL}^{-1}. Variables salt and hashed_passwd are the salt and hashed-password components of the entry in the system's* `/etc/passwd` *file for the user attempting to log in. The nonce serves to prevent replay attacks against a well-behaved server.*

160-bit symmetric key based on randomness obtained from the TPM and uses the TPM to seal the key so that no other code can access it. It then performs application specific work.

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

6.3 Secret and Integrity-Protected State

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

6.3.1 SSH Password Authentication

We have applied Flicker to password-based authentication with SSH. Since people tend to use the same password for multiple independent computer systems, a compromise on one system may yield access to other systems. Our primary goal is to prevent any malicious code on the server from learning the user's password, even if the server's OS is compromised. Our secondary goal is to convince the client system (and hence, the user) that the secrecy of the password has been preserved. Flicker is well suited to these goals, as it makes it possible to restrict access to the user's cleartext password on the server to a tiny TCB (the COAL), and to attest to the client that this indeed was enforced. While other techniques (e.g., PwdHash [23]) exist to ensure varied user passwords across servers, SSH provides a useful illustration of Flicker's properties when applied to a real-world system.

Our implementation is built upon the basic components we have described in the preceding sections, and consists of five main software components. A modified SSH client runs on the client system. The client

system does not need hardware support for Flicker, but a compromise of the client may leak the user's password. We are investigating techniques for utilizing Flicker on the client side. We add a new client authentication method, *flicker-password*, to OpenSSH version 4.3p2. *flicker-password* is the realization of the combination of the protocols from Section 4.4.2 and Figure 6.

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

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

In more detail, the Secure Channel module conveys a public key K_{COAL} to the client in such a way that the client is convinced that the corresponding private key is accessible only to the same COAL in a subsequent Flicker session. Thus, by verifying the attestation from the first Flicker session, the client is convinced that the correct COAL executed, that the legitimate COAL created a fresh keypair, and that the SLB Core erased all secrets before returning control to the untrusted OS. Using its authentic copy of K_{COAL}, the client encrypts the user's password for transmission to the second Flicker session on the server. We use PKCS1 encryption which is chosen-ciphertext-secure and nonmalleable [15]. The end-to-end encryption of the user's password, from the client system all the way into the COAL, protects the user's password in the event that any of the server's software is malicious.

Second Flicker Session (Login). The second Flicker session processes the user's encrypted password (see Figure 6). The COAL performs the TPM Unseal operation to retrieve K_{COAL}^{-1} from *sdata*. It then decrypts the user's password. Finally, it computes the hash of the password and salt[5] and outputs the result for comparison with the server's password file. The end result is that the user's unencrypted password only exists on the server during a Flicker session.

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

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

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

6.3.2 Certificate Authority

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

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

---

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

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

Table 1: **Breakdown of Rootkit Detector Overhead.** *The first three are incurred during the Flicker session, while the TPM Quote is generated on the OS. The standard deviation was negligible for all operations.*

7   Performance Evaluation

Below, we describe our experimental setup and evaluate the performance of the Flicker platform, as well as the various applications described in Section 6.

7.1   Experimental Setup

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

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

7.2   Stateless Applications

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

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

Microbenchmarks.   To better understand the overhead of the rootkit detector, we performed additional microbenchmarks to determine the most expensive operations involved (see Table 1). The results indicate that the highest overhead comes from the Quote operation. This performance is TPM-specific. Other TPMs contain faster implementations; for example, an Infineon TPM can generate a quote in under 331 ms. Within the COAL, the main Flicker-related overhead arises from the *SKINIT* operation. This prompts us to further analyze the overhead of *SKINIT*.

SKINIT Overhead.   The performance overhead of *SKINIT* is divided into two parts: the time needed to place the CPU in an appropriate state with protections enabled, and the time to transfer the SLB to the TPM.

To investigate the breakdown of the *SKINIT*'s performance overhead, we ran the *SKINIT* command on our AMD test machine with SLBs of various sizes. We invoke *RDTSC* before executing *SKINIT* and invoke it a second time as soon as code from the SLB can begin executing. Figure 2 summarizes the timing results and shows that changing the CPU state requires less than 1 ms, indicating that the TPM is the cause of the remaining overhead. The runtime varies linearly with the number of bytes in the SLB, as the CPU must send the entire SLB to the TPM to be measured.

SKINIT Optimization.   Short of changing the speed of the TPM and the bus through which the CPU communicates with the TPM, the best opportunity for improving the performance of *SKINIT* is to reduce

14

SLB Size (KB)	0	4	16	32	64
Avg (ms)	0.0	11.9	45.0	89.2	177.5

Table 2: *Time required to execute the SKINIT instruction on our AMD test machine with SLBs of various sizes.*

Detection Period [min.]	Benchmark Time [m:s]	Standard Deviation [s]	Slowdown % (s)
No Detection	6:43.9	0.50	0 (0)
5.0	6:49.1	0.71	1.2 (5.2)
3.0	6:48.7	1.30	1.2 (4.8)
2.0	6:49.0	1.10	1.2 (5.1)
1.0	6:49.9	1.41	1.5 (6.0)
0.5	6:51.4	0.55	1.7 (7.5)

Table 3: **Impact of the Rootkit Detector.** *Kernel build time when run with no detection and with rootkit detection run periodically.*

the size of the SLB. To maintain the security properties provided by *SKINIT*, however, code in the SLB must be measured before it is executed. Note that *SKINIT* enables the Device Exclusion Vector for the entire 64 KB of memory starting from the base of the SLB, even if the SLB's length is less than 64 KB. One viable optimization is to include a cryptographic hash function and enough TPM support to perform a PCR Extend, in the first part of the SLB, along with the necessary code to measure and extend the remaining portions of the SLB. An SLB constructed in this way offloads most of the burden of computing its own measurement to the system's main CPU. We have constructed such an SLB in 4736 bytes. When this SLB runs, it measures the entire 64 KB and extends the resulting measurement into PCR 17. Thus, when *SKINIT* executes, it only needs to transfer 4736 bytes to the TPM. In 50 trials, we found the average *SKINIT* time to be 14 ms. While only a small savings for the rootkit detector, it saves 164 ms of the 176 ms *SKINIT* requires with a 64 KB SLB. We use this optimization in the rest of our applications.

System Impact. As a final experiment, we evaluate the rootkit detector's impact on the system by measuring the time required to build the 2.6.20 Linux kernel while also running the rootkit detector periodically. Table 3 summarizes our results. Essentially, our results suggest that even frequent execution of the rootkit detector has little impact (less than 2% slowdown) on the system's overall performance.

7.3 Integrity-Protected State

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

In our experiments, we evaluate the amount of Flicker-imposed overhead by measuring the time required to start performing useful application work, specifically, between the time the OS hands control to the BOINC client, and the time at which the COAL begins to perform application-specific work.

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

Table 4: **Operations for Distributed Computing.** *This table indicates the significant expense of the Unseal operation, as well as the tradeoff between efficiency and latency.*
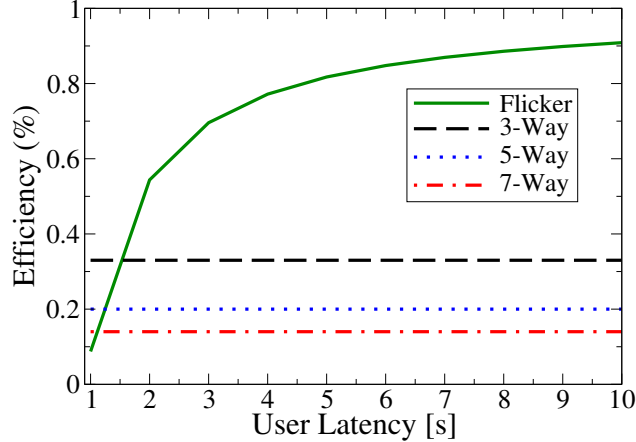
Figure 7: **Flicker vs. Replication Efficiency.** *Replicating to a given number of machines represents a constant loss in efficiency. Flicker gains efficiency as the length of the periods during which application work is performed increases.*

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

(a) **COAL 1**

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

(b) **COAL 2**

Figure 8: **SSH Overhead.** *Average server side performance over 100 trials, including a breakdown of time spent inside each COAL. The standard error on all measurements is under 1%, except key generation at 14%.*

Table 4 shows the resulting overhead, as well as its most expensive constituent operations, in particular, the time for the *SKINIT*, and the time to unseal and verify the COAL's previous state.[6] The table demonstrates how the application's efficiency improves as we allow the COAL to run for longer periods of time before exiting back to the untrusted OS. It also indicates that the vast majority of the overhead arises from the TPM's Unseal operation. Again, a faster TPM, such as the Infineon, can unseal in under 400 ms.

While Flicker adds additional overhead on a single client, the true savings come from the higher degree of trust the application writer can place in the results returned. Figure 7 illustrates this savings by comparing the efficiency of Flicker-enhanced distributed computing with the standard solution of using redundancy. With our current implementation, a two second user latency allows a more efficient distributed application than replicating to three or more machines. As the performance of this new hardware improves, the efficiency of using Flicker will only increase.

7.4 Secret and Integrity-Protected State

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

7.4.1 SSH Password Authentication

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

---

[6]As described in Section 6.2, the initial COAL must also generate a symmetric key and seal it under PCR 17. We discuss this overhead in more detail in Section 7.4.

Figure 8 presents these results, as well as a breakdown of the most expensive operations that execute on the SSH server. The total time elapsed on the client between the establishment of the TCP connection with the server, and the display of the password prompt for the user is 1221 ms (this includes the overhead of the first COAL, as well as 949 ms for the TPM Quote operation), compared with 210 ms for an unmodified server. Similarly, the time elapsed beginning immediately after password entry on the client, and ending just before the client system presents the interactive session to the user is approximately 940 ms while the unmodified server only requires 10 ms. The primary source of overhead is clearly the TPM. As these devices have just been introduced by hardware vendors and have not yet proven themselves in the market, it is not surprising that their performance is poor. Nonetheless, current performance suffices for lightly-loaded servers, or for less time-critical applications, such as the CA.

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

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

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

7.4.2 Certificate Authority

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

8 Related Work

In an earlier extended abstract [18], we described the goals and motivation for a system like Flicker, and suggested that the new processors from AMD and Intel could support such an architecture. This work expands the abstract with a detailed design, implementation, supporting tools, multiple applications, extensive evaluation results, and lessons learned. Below, we discuss closely related work in the area of code isolation, code integrity and remote attestation.

Various systems provide isolation through virtualization (e.g., Terra [8], Nizza [27] and Proxos [29]) or by running the code inside trusted hardware, for example the Dyad HW architecture [32], the IBM 4758 [14,28] or the Cerium processor [7]. Jiang used a secure coprocessor to build an SSL co-server to process student passwords and grades [13]. Flicker adds less than 250 lines of code to the TCB of a COAL, compared with tens or hundreds of thousands of lines of code for today's popular VMMs. While Flicker does not achieve the same level of physical tamper-resistance as do secure coprocessors, it provides the same strong software guarantees using modern commodity hardware.

In the area of providing code integrity guarantees, IBM's Integrity Measurement Architecture (IMA) is a realization of the trusted boot approach [24]. Unfortunately, the security of a newly executed piece of code depends on the security of all previously executed code—due to the lack of isolation, a single compromised piece of code may compromise all subsequent code. Such large attestations can be difficult to verify and leak information about the software on the attestor's platform. The BIND system guarantees the safe execution of a small piece of code to an external party [26], but BIND relies on the security of a trusted kernel that was never implemented. The Pioneer system provides code integrity guarantees to an external verifier [25]. However, the verifier in Pioneer needs to be a local host, because Pioneer cannot be used across the Internet.

Garriss et al. employ the new *SKINIT* instruction to eliminate the BIOS and bootloader from the TCB [9], but as suggested in the original design [10], after the *SKINIT*, they launch a standard OS or VMM. Thus,

application security depends on large layers of code.

9 Conclusion

Flicker allows code to verifiably execute in complete, hardware-enforced isolation. Flicker itself adds as few as 250 lines of code to the application's TCB. Given the correlation between code complexity and bugs in the code, Flicker significantly improves the security and reliability of the code it executes. New desktop machines already contain the hardware support necessary for Flicker, so widespread Flicker-based applications can soon become a reality. As a result, our research brings a Flicker of hope for securing commodity computers.

References

[1] Advanced Micro Devices. AMD64 virtualization: Secure virtual machine architecture reference manual. AMD Publication no. 33047 rev. 3.01, May 2005.

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

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@Home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[4] D. Balfanz. *Access Control for Ad-hoc Collaboration*. PhD thesis, Princeton University, 2001.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of SOSP*, 2003.

[6] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of USENIX Security Symposium*, 2004.

[7] B. Chen and R. Morris. Certifying program execution with secure procesors. In *Proceedings of HotOS*, 2003.

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

[9] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Towards trustworthy kiosk computing. In *Workshop on Mobile Computing Systems and Applications*, Feb. 2006.

[10] D. Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006.

[11] S. Halevi and H. Krawczyk. Public-key cryptography and password protocols. *ACM Trans. Information and System Security*, 2(3), 1999.

[12] Intel Corporation. LaGrande technology preliminary architecture specification. Intel Publication no. D52212, May 2006.

[13] S. Jiang. WebALPS implementation and performance analysis. Master's thesis, Dartmouth College, 2001.

[14] S. Jiang, S. Smith, and K. Minami. Securing web servers against insider attack. In *Proc. of the Computer Security Applications Conference*, 2001.

[15] B. Kaliski and J. Staddon. PKCS #1: RSA cryptography specifications. RFC 2437, 1998.

[16] D. Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference*, 2003.

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

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

[19] D. Molnar. The SETI@Home problem. *ACM Crossroads*, 7.1, 2000.

[20] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM PLDI*, 1998.

[21] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the Conference on Compiler Construction*, 2002.

[22] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *the USENIX Security Symposium*, Aug. 2003.

[23] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the USENIX Security Symposium*, Aug. 2005.

[24] 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.

[25] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. VanDoorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of SOSP*, 2005.

[26] E. Shi, A. Perrig, and L. van Doorn. BIND: A time-of-use attestation service for secure distributed systems. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2005.

[27] 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.

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

[29] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of OSDI*, 2006.

[30] Trusted Computing Group. PC client specific TPM interface specification (TIS). `http://www.trustedcomputinggroup.org`, July 2005. Version 1.2, Revision 1.00.

[31] Trusted Computing Group. Trusted platform module main specification. Version 1.2, Revision 94, 2006.

[32] B. S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.

[33] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Secure program partitioning. *ACM Trans. on Computer Systems*, 20(3), Aug. 2002.