Reducing the Trusted Computing Base for Applications on Commodity Systems

Jonathan M. McCune

January 13, 2009

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

Thesis Committee:

Prof. Adrian Perrig, Co-Chair (Carnegie Mellon University)
Prof. Michael K. Reiter, Co-Chair (University of North Carolina)
Prof. Gregory R. Ganger (Carnegie Mellon University)
Dr. Leendert van Doorn (Advanced Micro Devices)

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

This research was supported in part by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 and MURI W 911 NF 0710287 from the Army Research Office, and grants CCF-0424422, CNS-0509004, CNS-0627357, CT-0433540 and CT-0756998 from the National Science Foundation, and by the iCAST project, National Science Council, Taiwan under the Grants No. (NSC95-main) and No. (NSC95-org), and by gifts from AMD and Intel.

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

Keywords: Software Security, Trusted Computing Base, Dynamic Root of Trust for Measurement, Remote Attestation, Trusted Path, User Input, Trusted Platform Module, Human-Verifiable, Authentication, Visual Channel, Camera Phone, Trusted Mobile Device

To honor the memory and character of Judge Barron Patterson McCune.

Abstract

Today we have powerful, feature-rich computer systems plagued by powerful, feature-rich malware. Current malware exploit the vulnerabilities that are endemic to the huge computing base that needs to be trusted to secure our private information. This thesis presents an architecture called Flicker that alleviates security-conscious developers from the burden of making sense out of this code base, allowing them to concentrate on the security of their own code. Since today's legacy operating systems will likely be used for the foreseeable future, we design Flicker to coexist with these systems.

Flicker allows code to execute in complete isolation from other software while trusting as few as 250 lines of additional code – orders of magnitude smaller than even minimalist virtual machine monitors. Flicker also enables more meaningful attestation of the code executed and its inputs and outputs than previous proposals, since only measurements of the security-sensitive portions of an application need to be included. Flicker leverages hardware support provided by commodity processors from AMD and Intel that are widely available today, and does not require a new OS or a VMM. Flicker's properties hold even if the BIOS, OS and DMA-enabled devices are all malicious. We evaluate a full implementation of Flicker on an AMD system and apply Flicker to four server-side applications.

We also perform a detailed case study of the use of Flicker to reduce the trusted computing base to which users' input events are exposed on their own computers, circumventing entire classes of malware such as keyloggers and screen scrapers. This case study involves the development of a system called Bumpy that allows the user to specify strings of input as sensitive when she enters them, and ensures that these inputs reach the desired endpoint in a protected state. The inputs are processed in a Flicker-isolated code module on the user's system, where they can be encrypted or otherwise processed for a remote webserver. A trusted mobile device can provide feedback to the user that her inputs are bound for the intended destination. We describe the design, implementation, and evaluation of Bumpy, with emphasis on both usability and security issues.

Flicker depends on attestations composed of cryptographic hashes and digital signatures to allow a remote verifier to ascertain the identity of code that executes with Flicker's protections. We propose a mechanism called Seeing-is-Believing to allow the computer's owner to authenticate the physical identity of her computer, in addition to its digital identity represented in the attestation. This rules out the possibility of successful man-in-the-middle or proxy attacks, and reduces the need for trusted third parties that are unavailable today.

Attestation technologies potentially pose a risk to users' privacy. Flicker protects users' privacy by including only the code executed during a Flicker session in an attestation, instead of providing information about all software loaded for execution during the current boot cycle.

Motivated by our experience with Flicker on today's hardware, we offer suggestions to improve Flicker's performance that leverage existing processor technology, retain security, and improve performance.

Acknowledgements

Though my name is printed on the cover of this thesis, the word "I" does not appear within its chapters. I do this to pay tribute to the myriad contributions of my advisors and collaborators, and the support of my family and friends.

Kathleen, thank you for believing in me and for your patience during so many late nights, many of them unanticipated. To my parents: Thank you for giving me the freedom and opportunity to pursue my own interests, even when they appeared incomprehensible or dangerous. I also thank my grandfather for teaching me that there is no substitute for honest hard work.

Mike and Adrian, the research environment you created for me is without match. Your dedication, encouragement, support, and hard work constantly inspired me to better myself and aim higher. I am at a loss for words.

Bryan and Arvind, Flicker would not have happened without you. Bryan, I am particularly grateful for your analytical and writing skills, and for letting me spend so much time buried in the details of these systems. Arvind, each lunch at Phipps Conservatory left me with new ideas and fresh energy.

To Reiner Sailer, Stefan Berger, Ramón Cáceres, Trent Jaeger, Ron Perez, Leendert van Doorn, and the rest of the IBM System Security Group circa 2005: The technical skills I learned during my summer in your research group made it possible for me to implement this thesis. Leendert van Doorn and Elsie Wahlig provided additional support for AMD SVM, without which I could not have implemented Flicker.

Lujo Bauer offered much advice and encouragement that was a great source of comfort to a new graduate student. Lujo's encouragement to persist with Mobile phone programming was my first foray into alpha release systems – an activity which has been very fruitful for this thesis.

Many of my fellow students and collaborators took time away from their own over-extended lives to provide feedback on this thesis, and for this I am extremely grateful. I hesitate to include such a list, as I will surely omit some who deserve mention. Unrestrained comments from many anonymous reviewers provided equal parts useful feedback, entertainment, and disbelief.

As an undergraduate at UVA, Dave Evans enthusiastically shared his passion for academia, provided me with support and encouragement, and helped me make the decision to begin the journey that has lead to this thesis.

To CMU Crew: My heart was and shall always remain in the work. HOTO and LiveStrong are on my calendar indefinitely.

To Josh, Andy, and Lenny: Thanks for helping me learn what is possible by leaving one's comfort zone, and for being ready for anything. To the WashPa crowd: I could not do the things I have done without your loyalty and friendship.

Finally, I want to thank all of my friends who made life in Pittsburgh enjoyable and sociable. Mike A., I will always aspire to match your level of optimism, clarity of purpose, and general zest for life. Scott, Jim and Bonnie, Bryan and Diana, Dave and Natalie, Ahren and Casey, and Ellery and Alexa, thanks for all the good times.

All errors and limitations remaining in this thesis are mine alone.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The size and complexity of modern operating systems makes them difficult to analyze and vulnerable to attack. Applications built on these OSes inherit their vulnerabilities, as the hierarchical privilege structure of these OSes yields total control of applications to the OS. Thus, even the simplest application function operates with a trusted computing base (TCB) consisting of the union of all operating system and device driver code. Other applications running with super-user privileges can readily modify the OS on which they run, thus adding their codebase to the TCB. If the OS is running on top of a virtual machine monitor, then it too must be trusted.

Regardless of how a computer system is designed, the security of a given application remains intimately tied to the user who is operating it. Security is a holistic property, and user errors can render even the best-designed security systems useless [93]. Security remains a secondary objective for most users, even if they are interested in protecting sensitive data such as their bank account numbers. Unfortunately, a security-conscious user who *wants* to dedicate some of her scarce time to verify that her input is not observed by malicious code during a sensitive online financial transaction faces an impasse. *Keyloggers* can capture a user's typed input and *screen scrapers* can process the content displayed to the user to obtain sensitive information such as credit card numbers [68, 119, 145, 170]. We lack the technology to give the user a definitive indication that her input is safe.

There are numerous ongoing projects to build a new "secure" OS [66, 153, 159], and there have been many attempts in the past [16, 86, 148, 149]. While we applaud these efforts, the reality is that computer systems are not always chosen for their technical merits. The availability of essential applications, network effects, and economies of scale apply tremendous pressure to retain legacy software. The size of the installed base of today's popular commodity OSes renders a clean-slate approach infeasible. Rather, we must coexist with these systems, while adhering to the axioms of isolation, small code size, and user-friendly design.

Trusted Computing technologies based on a Trusted Platform Module (TPM) security chip such as remote attestation and sealed storage have been proposed with the goal of improving the resilience of commodity computing platforms against software-based attacks [172]. These technologies have received criticism for their ability to erode users' privacy because (1) they require trusted third parties that uniquely identify the user's platform and then possibly the user through product registration information [7]. Additionally, (2) integrity measurement and remote attestation based on a static root of trust [143] leak information about *all* software loaded during the current boot cycle, even if a remote challenger is only interested in a single application on the user's computing platform. This thesis presents mechanisms that eliminate concern (2) and partially relieve concern (1).

Designing new systems while maintaining compatibility with old systems has been standard practice since the computing industry burgeoned in the 1970s [89]. Today, however, there exists a mountain of software that is riddled with security vulnerabilities. Unfortunately, the number of vulnerabilities tends to be proportional to code size [117]. In this thesis, we will show how to execute sensitive code in isolation despite the presence of untrusted legacy code, thereby opening up the opportunity for security-conscious developers to create applications without including the entire software stack in each application's TCB.

Thesis Statement. *Security-sensitive code can be verifiably executed in isolation on commodity hardware without a persistent layer of trusted system software, and without breaking compatibility with legacy operating*

systems and applications. This architecture can be realized with a software trusted computing base that is orders of magnitude smaller than even today's minimalist virtual machine monitors.

We now provide an overview of each technical chapter of this thesis. Background material is presented in Chapter 2, and additional related work is discussed in Chapter 7 following the primary technical content. The results of this thesis have been documented in a number of publications [107, 108, 109, 111, 112, 113, 114, 115].

1.1 TCB Minimization Infrastructure

We develop *Flicker*, a secure execution architecture that allows security-sensitive code to execute in complete isolation from all other software (including the operating system and VMM, if present). Flicker's properties hold even if the BIOS, OS and DMA-enabled devices are all malicious. This dramatic reduction in the size of the TCB for an application enables meaningful software attestation and facilitates formal security analysis of the software remaining in the TCB. Flicker provides these guarantees without requiring a reboot, a change of OS, or a VMM. Indeed, in its most minimal configuration Flicker adds fewer than 250 lines of code to the mandatory software TCB. Yet Flicker's isolation also allows more complex configurations to include more code without forcing its inclusion in the TCB for minimized configurations. Flicker coexists with existing systems by imposing no computational overhead at all when Flicker is inactive.

Flicker leverages hardware support for secure virtualization provided by AMD's Secure Virtual Machine (SVM) architecture [3] or Intel's Trusted Execution Technology (TXT) [75]. These technologies provide a hardware-based dynamic root of trust, as well as new forms of memory protection. They are designed to atomically measure and launch a VMM or security kernel without requiring a reboot [61]. In contrast, we propose using this technology to securely execute sensitive application code in complete isolation and then return to the user's legacy operating system. By doing so, we eliminate the OS from the application's TCB. Furthermore, our architecture

can be deployed today, and need not await the development of a perfectly secure VMM. SVM- and TXT-equipped processors are currently shipping in commodity servers and PCs.

Remote attestation technology based on the TPM [172] can be used to convince a remote party that precisely this code module and nothing else executed during a Flicker *session*. This design protects the user's privacy and relieves the verifier from having to make sense out of the user's entire software stack. Flicker supports protocols for establishing authentic communication between a PAL and a remote entity, and it is architected such that the code that generates attestations need not be trusted.

Flicker allows application developers to focus on the security of *their* code without blindly trusting an unverifiable quantity of code executing below. We describe the design, implementation, and analysis of Flicker on commodity hardware in Chapter 3. Chapter 3 also shows how we use Flicker to improve the security of four different server applications.

1.2 TCB Reduction for Sensitive User Input

We have already described how a vulnerability in any part of a system's OS renders users' sensitive data insecure regardless of what application they may be running. On top of this untrustworthy OS sits a complex and monolithic web browser, which faces protection and assurance challenges similar to those of the OS [33]. It is not surprising that trusting this software stack for the protection of private data in web transactions often leads to data compromise.

In Chapter 4, we provide a detailed case study where we develop *Bumpy*, a system that builds on Flicker for protecting user input to web pages. This case study is more thorough than our server-side Flicker applications, and illustrates many of the challenges arising from the use of Flicker to secure client-side applications.

Bumpy leverages encryption-capable input devices to process all user input in a Flicker-protected environment. We employ the secure attention sequence `@@` to enable users to specify forthcoming input as sensitive, so

that far more than just passwords can be protected. A trusted mobile device can also receive updates from Bumpy's Flicker-protected module to provide definitive feedback to the user as to the status of input protections on her system. Bumpy enables the remote webserver to specify how sensitive input is processed once it has been entered, with TPM-based remote attestation to convince the webserver that the desired processing is active. Thanks to Flicker, the attestation covers only such input-processing code, and protects the users' privacy since no information is leaked about other (security-*irrelevant*) code. We explore end-to-end encryption and password hashing [52, 51, 138] as two possibilities for processing input.

1.3 Human-Verifiable Authentication

Trusted computing technology depends on a third party to certify a computing platform and its TPM as complying with the relevant specifications without leaking the exact identity of the platform. Unfortunately, the resulting certificates do not provide strong *physical* identification of the relevant computing platform to the platform owner. This limitation is particularly egregious when a user wants to verify the security properties of a public computer, e.g., in an Internet cafe. In Chapter 5, we develop *Seeing is Believing* (SiB), a technique leveraging two-dimensional barcodes and camera phones to create a *visual channel* that provides *demonstrative identification* of communicating devices to the human user(s). SiB works even when devices share no prior context, or when the prerequisite of a trusted third party or public key infrastructure (PKI) may undesirably inflate the user's TCB, if such an authority exists at all.

These devices may be components in one person's computing environment, such as her keyboard, display, mobile phone, laptop or digital camera, or they may be devices belonging to two different people. The most relevant use of SiB for this thesis is to ascertain the identity of the TPM in a user's computer. This enables one to bind a third-party certificate to the physical identity of a particular platform, thereby allowing users or administrators who take an interest in security to effectively manage their own systems.

1.4 Architectural Recommendations

One result of our performance evaluation of Flicker is that we use the available secure-virtualization hardware features far more frequently than intended during their original design. In Chapter 6, we summarize the primary sources of overhead for Flicker and make architectural recommendations for next-generation hardware to better support Flicker.

Chapter 2

Background on Trusted Computing Technology

This thesis leverages a suite of technologies known as *trusted computing* and specified by the Trusted Computing Group (TCG). The TCG is an organization that promotes open standards to strengthen computing platforms against software-based attacks [172]. The purpose of this chapter is to provide background information on TCG technologies, as leveraged by this thesis. We discuss the static and dynamic roots of trust for measurement (Sections 2.1 and 2.2), remote attestation (Section 2.3), and sealed storage (Section 2.4). Both AMD and Intel are shipping chips with these capabilities; they can be purchased in commodity computers.

2.1 Integrity Measurement and Static Root of Trust

The v1.2 Trusted Platform Module (TPM) chip contains an array of 24 or more Platform Configuration Registers (PCRs), each capable of storing a 160-bit hash. These PCRs can be *Extended* with a *Measurement* (cryptographic hash) of data, such as a program binary. Given a measurement `m` $\leftarrow$ `SHA-1(data)`, extend performs: $\texttt{PCR}_{new} \leftarrow \texttt{SHA-1}(\texttt{PCR}_{old}||\texttt{m})$.

TPMs include two kinds of PCRs: static and dynamic. Static PCRs reset to 0^{160} when the TPM itself resets (generally during a full platform

reset or power-cycle, although physical TPM-reset attacks have been demonstrated [88, 94, 140]), and can only have their value updated via an Extend operation. This is known as a *Static* Root of Trust because it persists for an entire boot cycle. These PCRs can be used to keep a record of measurements for all software loaded since the last reboot, as in IBM's Integrity Measurement Architecture [143]. Dynamic PCRs are present in v1.2 TPMs, and are relevant when the platform supports Dynamic Root of Trust.

Once measurements have accumulated in the PCRs, they can be *attested* to a remote party to demonstrate what software has been loaded on the platform. They can also be used to *seal* data to a particular platform configuration. We discuss each of these in Sections 2.3 and 2.4, respectively.

2.2 Late Launch / Dynamic Root of Trust

AMD's Secure Virtual Machine (SVM) extensions [3] and Intel's Trusted eXecution Technology (TXT, formerly LaGrande Technology (LT)) [75] provide support for a *late launch* operation to bootstrap a *Dynamic* Root of Trust. This primary difference is that a dynamic root of trust can be established without requiring a full platform reset.

The TPM v1.2 specification [171] 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 0^{160} without a reboot. A reboot sets the value of PCRs 17–23 to 1^{160}, 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 executing the *late launch* instruction to bootstrap a *Dynamic* Root of Trust (*SKINIT* on AMD systems and *SENTER* on Intel systems). Thus, software cannot reset PCR 17, though PCR 17 can be read and extended by software before calling *SKINIT* or after *SKINIT* has completed.

In addition to resetting the dynamic PCRs, late launch resets the CPU to a known trusted state without rebooting the rest of the system. This includes configuring the system's memory controller to prevent access to the

launching code from DMA-capable devices. One of the newly reset dynamic PCRs is then automatically extended with a measurement of the software that will get control following the late launch [3]. This enables software to bootstrap without including the BIOS or any system peripherals in the TCB. The Open Secure LOader (OSLO) performs a late launch on AMD systems to remove the BIOS from the TCB of a Linux system [88]. Trusted Boot[1] from Intel performs similarly for Intel hardware, though it adds the ability to enforce a Launch Control Policy (LCP). The Flicker system (Chapter 3) uses late launch to briefly interrupt the execution of a legacy OS and execute a special-purpose code module in isolation from all other software and devices on the platform, before returning control to the legacy OS.

2.2.1 AMD Secure Virtual Machine (SVM)

To perform a late launch on a system with AMD SVM, 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)

[1] `http://sourceforge.net/projects/tboot`

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.

2.2.2 Intel Trusted Execution Technology (TXT)

Intel's TXT is comprised of processor support for virtualization (VT-x) and Safer Mode Extensions (SMX) [75]. SMX provides support for the late launch of a VMM in a manner similar to AMD's SVM, so we focus primarily on the differences between the two technologies. Instead of *SKINIT*, Intel introduces a new "leaf" instruction called *GETSEC*, which can invoke various leaf operations, including *SENTER*.

A late launch invoked with *SENTER* is comprised of two phases. First, an Intel-signed code module—called the Authenticated Code Module, or ACMod—must be loaded into memory. The platform's chipset verifies the signature on the ACMod using a built-in public key, extends a measurement of the ACMod into PCR 17, and finally executes the ACMod. The ACMod is then responsible for measuring the equivalent of AMD's SLB, extending the measurement into PCR 18, and then executing the code. Analogous to AMD's DEV, Intel protects the memory region containing the ACMod and the SLB from outside memory access using a DMA Remapping table provided by their Virtualization Technology for Directed I/O (VT-d) [76].

2.3 Attestation

A computing platform containing a Trusted Platform Module (TPM) can provide an *attestation* of 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. The log is maintained by an *integrity measurement architecture* (e.g., IBM IMA [143]). Each event is reduced to a *measurement*, m, using the SHA-1 cryptographic hash function. For example, program `a.out` is reduced to a measurement by hashing its binary

executable: $m \leftarrow$SHA-1(`a.out`). Each measurement is *extended* into one of the TPM's PCRs.

The attestation process involves a challenge-response protocol, where the challenger sends a cryptographic nonce (for replay protection) and a list of PCR indexes, and requests a *TPM Quote* over the listed PCRs. A Quote is a digital signature computed over an aggregate of the listed PCRs using an *Attestation Identity Key* (AIK). An AIK is an asymmetric signing keypair generated on the TPM. We discuss certification of AIKs shortly. The messages exchanged between a challenger C and an untrusted system U to perform an attestation are:

$C \rightarrow U$: `nonce, PCRindexes`

$U \rightarrow C$: `PCRvals,` $\{$`PCRvals, nonce`$\}_{AIK^{-1}}$

Once the challenger receives the attestation response, it must (1) verify its nonce is part of the reply, (2) check the signature with the public AIK obtained via an authentic channel, (3) verify that the list of PCR values received corresponds to those in the digital signature, and (4) verify that the PCR values themselves represent an acceptable set of loaded software. Note that since the sensitive operations for a TPM Quote take place entirely within the TPM chip, the TPM Quote operation can safely be invoked from untrusted software. The only attack available to malicious software is denial-of-service. In the context of the Flicker system (Chapter 3), this removes the code that causes the TPM Quote to be generated from the system's TCB.

2.3.1 Certifying Platform Identity

The Attestation Identity Keypair (AIK) used to perform the TPM Quote effectively represents the identity of the attesting host. While we discuss the use of Seeing-is-Believing to authenticate a system's AIK in Chapter 5, this is only useful for learning the AIK of physically nearby systems. We now discuss options for certifying an AIK (i.e., authenticating the public AIK for a particular physical host) that are viable across the Internet.

Multiple credentials are provided by TPM and host manufacturers that are intended to convince a remote party that they are communicating with

a valid TPM installed in a host in conformance with the relevant specifications [172]. These are the TPM's Endorsement Key (EK) Credential, Platform Credential, and Conformance Credential. One option is to use these credentials directly as the host's identity, but the user's privacy may be violated. Motivated by privacy concerns, the TCG has specified Privacy Certificate Authorities (Privacy CAs). Privacy CAs are responsible for certifying that an AIK generated by a TPM comes from a TPM and host with valid Endorsement Key, Platform, and Conformance Credentials.

Attestation Identity Keys and Privacy CAs were proposed in v1.1b of the TCG specification [172]. Direct Anonymous Attestation (DAA) has also been proposed as an alternative to Privacy CAs for protecting platform identity [23, 26]. To the best of our knowledge, no systems are available today that include TPMs supporting DAA. Note that the design of Flicker (Chapter 3) is orthogonal to the choice of authentication method.

2.4 TPM-Based Sealed Storage

All TPMs generate a 2048-bit RSA Storage Root Key (SRK) that will never leave the chip. The SRK enables *sealed storage*, whereby data leaving the TPM chip is encrypted under the SRK for storage on another medium. Data can be sealed with respect to the values of certain PCR registers, so that the unsealing process will fail unless the required values are present. TPM Seal outputs a ciphertext, which contains the sealed data and information about the platform configuration required for its release. Software is responsible for keeping it on a non-volatile storage medium. There is no limit on the use of sealed storage, but the data is encrypted using (relatively slow) asymmetric algorithms inside the TPM. Thus, it is common to encrypt and MAC the data to be sealed using (relatively fast) symmetric algorithms on the platform's main CPU, and then keep the symmetric encryption and MAC keys in sealed storage. The TPM includes a random number generator that can be used for key generation.

An alternative is to use the TPM's Non-Volatile RAM (NV-RAM) facility. NV-RAM can be configured with similar properties to sealed storage,

in that a region of NV-RAM can be made inaccessible unless the PCR values match those specified when the region was defined. NV-RAM has a limited number of write cycles during the TPM's lifetime, but the use of a symmetric master key that is only read from NV-RAM in the common case can greatly extend its life. Flicker (Chapter 3) can use TPM sealed storage or NV-RAM to protect long-term state that is manipulated during Flicker sessions.

Chapter 3

Flicker: An Execution Infrastructure for Minimizing the Trusted Computing Base using a Dynamic Root of Trust

The large size and huge complexity of today's popular operating systems makes them difficult to analyze and vulnerable to attack. These systems run a daunting amount of code in the CPU's most privileged mode. Version 2.6 of the Linux *kernel* alone consists of nearly 5 million lines of code [179], while Microsoft's Windows Server 2003 includes over 50 million lines of code [105]. Even virtual machine monitors (VMMs), often touted as smaller and more secure than commodity operating systems, include substantial amounts of code that tend to grow over time. For example, the initial implementation of the Xen VMM required 42K lines of code [13] and within a few years almost doubled to approximately 83K lines [104], excluding the size of the privileged OS running in the management VM. The inevitability 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 because the Trusted Computing Base (TCB) for a given application extends far beyond the application code itself. Unfortunately, today's formal methods for proving software correct do not scale to the size of today's operating systems and applications. As a result, even security-conscious application developers can make few guarantees about the security properties of their applications. Thus, an effective way to improve the security of applications today is to reduce the size of the relevant Trusted Computing Base (TCB). If the TCB for code execution can be precisely defined and limited, formal assurance of both reliability and security properties enters the realm of possibility [28].

An additional challenge for secure software execution is to learn that the desired code was actually loaded for execution. Remote attestation is one way to convince an external entity that a particular program or set of programs was loaded for execution. However, the security properties attainable via attestation also suffer when the TCB is bloated, since such a large TCB prevents current proposals for system-wide code attestation [8, 106, 143] from providing meaningful security information. The only guarantee is that exploitable vulnerabilities do exist.

Attestation is more valuable in a system that provides strong isolation of application code, so that only relevant code needs to be included in attestations. Such *fine-grained* attestations also 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 smaller attestation does not leak extraneous information about the system's software state.

Thus, we need a mechanism to execute security-sensitive code in isolation without bloating the TCB and to attest to the correct invocation of this code. To achieve these goals, we propose *Flicker*, an architecture to enable code execution with a trusted computing base (TCB) that is orders of magnitude smaller than even minimalist hypervisors or security kernels. 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. The use of Flicker, as well as the exact code executed (and its inputs and outputs), can be attested to an external party. Flicker can operate at any time and does not require a new OS or even a VMM, so the platform for non-sensitive operations is unchanged.

For example, a Certificate Authority (CA) can sign certificates with its private key, even while keeping the key secret from an adversary that controls the BIOS, OS, and DMA-enabled devices. Or, server code handling a user's password can execute in complete isolation from all other software on the server, and an attestation from the server can convince the client that the secrecy of the password is preserved.

To achieve these properties, Flicker utilizes hardware support for establishing a dynamic root of trust (also known as 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. Flicker provides strong isolation guarantees while requiring the application to trust as few as 250 additional lines of code for its secrecy and integrity, thereby circumventing entire layers of legacy system software and eliminating reliance on their correctness for security properties (Figure 3.1).

Chapter 2 contains background information on the Trusted Computing primitives on which Flicker is constructed. Although current hardware still has a high overhead, we anticipate that future hardware performance will improve as these functions are increasingly used. Indeed, in Chapter 6, we suggest hardware modifications that can improve performance by up to six orders of magnitude. Finally, many applications perform security-sensitive operations where the speed of the operations is not the first priority.

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, such as cryptographic operations or protecting the existing execution environment from malicious or malfunctioning PALs.

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

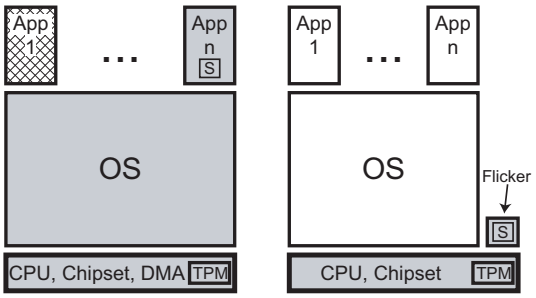


Figure 3.1: On the left, a traditional computer with an application that executes sensitive code (S). On the right, Flicker protects the execution of the sensitive code. The shaded portions represent components that must be trusted; other applications are included on the left because many applications run with or can be exploited to acquire superuser privileges.

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.

3.1 Problem Definition

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

3.1.1 Adversary Model

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

At the hardware level, we make the same assumptions as does the Trusted Computing Group with regard to the TPM [172]. In essence, the attacker can launch simple hardware attacks, such as opening the case, power cycling the computer, or attaching a hardware debugger. The attacker can also compromise expansion hardware such as a DMA-capable Ethernet card with access to the PCI bus. However, the attacker cannot launch sophisticated hardware attacks, such as monitoring the high-speed bus that links the CPU and memory.

3.1.2 Goals

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

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

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

Meaningful Attestation. Allow the creation of attestations that include measurements of exactly the code executed, its inputs and outputs, and nothing else. This property provides the dual advantages of giving the verifier a tractable task (instead of learning only that untold millions of lines of code were executed), and leaking as little information as possible about the attestor's software state (instead of sharing the identity of all software executed since reboot).

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 [61]. 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 [12] hypervisor with *SKINIT* would add almost $50,000$ lines of code[1] to an application's TCB, not including the Domain 0 OS, which potentially adds millions of additional lines of code to the TCB.

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

3.2 Flicker Architecture

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

3.2.1 Flicker Overview

Flicker achieves its properties using the late launch capabilities described in Chapter 2.2. Instead of launching a VMM, Flicker pauses the current exe-

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

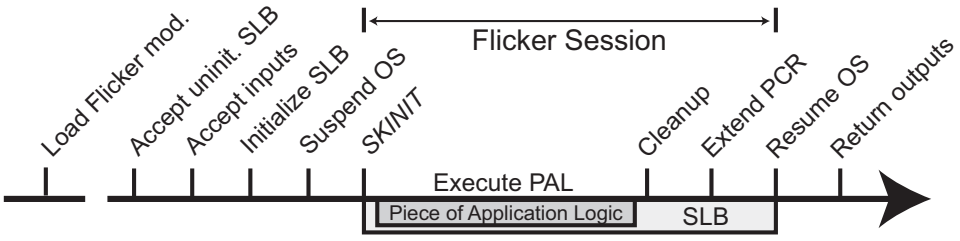


Figure 3.2: Timeline showing the steps necessary to execute a PAL. The SLB includes the PAL, as well as the code necessary to initialize and terminate the Flicker session. The gap in the time axis indicates that the *flicker-module* is only loaded once.

cution environment (e.g., the untrusted OS), executes a small piece of code using the *SKINIT* instruction, and then resumes operation of the previous execution environment. The security-sensitive code selected for Flicker protection is the Piece of Application Logic (PAL). The protected environment of a Flicker session starts with the execution of *SKINIT* and ends with the resumption of the previous execution environment. Figure 3.2 illustrates this sequence.

Application developers must provide the PAL and define its interface with the remainder of their application (we discuss this process, as well as our work on automating it, in Section 3.3). To create an SLB (the Secure Loader Block supplied as an argument to *SKINIT*), the application developer links her PAL against an uninitialized code module we have developed called the SLB Core. The SLB Core performs the steps necessary to set up and tear down the Flicker session. Figure 3.3 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.

3.2.2 Isolated Execution

We provide a simplified discussion of the operation of a Flicker session by following the timeline in Figure 3.2.

Accept Uninitialized SLB and Inputs. *SKINIT* is a privileged instruction, so an application uses the *flicker-module*'s interface to invoke a Flicker session. In the sysfs,[2] the *flicker-module* makes four entries available: `control`, `inputs`, `outputs`, and `slb`. Applications interact with the *flicker-module* via these file-system entries. An application first writes to the `slb` entry an uninitialized SLB containing its PAL code. The *flicker-module* allocates kernel memory in which to store the SLB; we refer to the physical address at which it is allocated as `slb_base`. The application writes any inputs for its PAL to the `inputs` sysfs entry; the inputs are made available at a well-known address once execution of the PAL begins (the parameters are at the top of Figure 3.3). The application initiates the Flicker session by writing to the `control` entry in the sysfs.

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

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

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

starting address of the PAL code, and hence it can fill in the appropriate entries in the SLB Core.

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

***SKINIT* and the SLB Core.** The *SKINIT* instruction enables hardware protections and then begins to execute the SLB Core, which prepares the environment for PAL execution. Executing *SKINIT* enables the hardware protections described in Section 2.2. In brief, the processor adds entries to the Device Exclusion Vector (DEV) to disable DMA to the memory region containing the SLB, disables interrupts to prevent the previously executing code from regaining control, and disables debugging support, even for hardware debuggers. By default, these protections are offered to 64 KB of memory, but they can be extended to larger memory regions. If this is done, preparatory code in the first 64 KB must add this additional memory to the DEV, and extend measurements of the contents of this additional

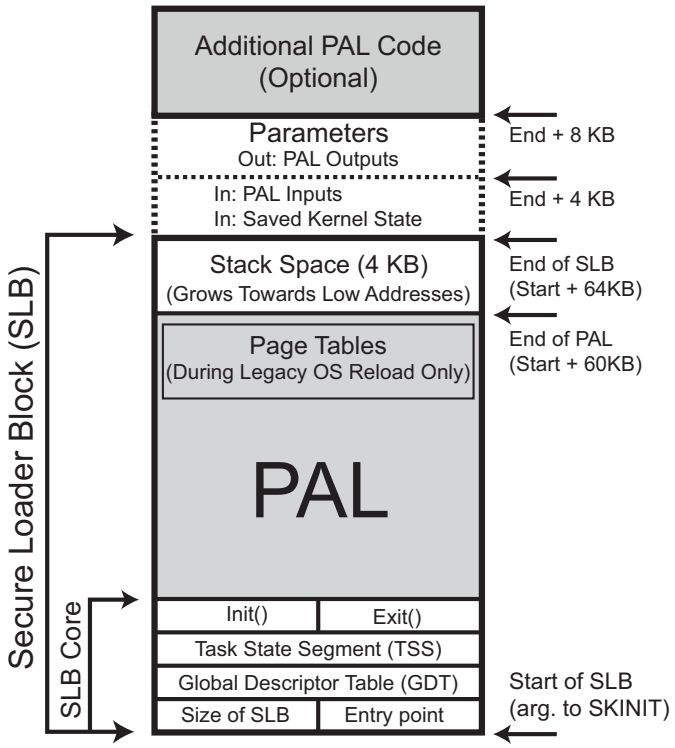


Figure 3.3: Memory layout of the SLB. The shaded region indicates memory containing executable PAL code. The dotted lines indicates memory used to transfer data into and out of the SLB. After the PAL has executed and erased its secrets, memory that previously contained executable code is used for the skeleton page tables needed to reload the OS.

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

Execute PAL. Once the environment has been prepared, the PAL executes its 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 PAL developer to include the functionality necessary for her particular application. Section 3.3 describes some of our existing modules that can optionally be included to provide additional functionality. We have also developed a module that can restrict the actions of a PAL, since by default (i.e., without the module), a PAL can access the machine's entire physical memory and execute arbitrary instructions (see Section 3.3.1.2 for more details).

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

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

Extend PCR. To signal the completion of the SLB, the SLB Core extends a well known value into PCR 17. As we discuss in Section 3.2.4.1, this allows a remote party to distinguish between values generated by the PAL (trusted), and those produced after the OS resumes (untrusted).

Resume OS. Linux operates with paging enabled and segment descriptors set to cover all of memory, but the SLB executes in protected mode with segment descriptors starting at `slb_base`. We transition between these

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 PAL outputs any values, the *flicker-module* makes them available through the sysfs `outputs` entry.

3.2.3 Multiple Flicker Sessions

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

Flicker-based secure storage can also be used by applications that wish to share data between PALs. The first PAL can store secrets so that only the second PAL can read them, thus protecting the secrets even when control reverts to the untrusted OS. Finally, Flicker-based secure storage can

improve the performance of long-running PAL jobs. Since Flicker execution pauses the rest of the system, an application may prefer to break up a long work segment into multiple Flicker sessions to allow the rest of the system time to operate, essentially multitasking with the OS. We first present the use of TPM Sealed Storage and then describe extensions necessary to protect multiple versions of the same object from a replay attack against sealed storage.

3.2.3.1 TPM Sealed Storage

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

3.2.3.2 Replay Prevention for Sealed Storage

TPM-based sealed storage prevents other code from directly learning or modifying a PAL's secrets. However, TPM Seal outputs ciphertext c (for data d) that is handled by untrusted code: $c \leftarrow TPM\_Seal(d, PCR\_list)$. The untrusted code is capable of performing a replay attack where an older ciphertext c' is provided to a PAL. For example, consider a password

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

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

Table 3.1: Replay protection for sealed storage based on a secure counter. Ciphertext c is created when data d is sealed.

database that is maintained in sealed storage and a user who changes her password because it is publicized. To change a user's password, version i of the database is unsealed, updated with the new password, and then sealed again as version $i+1$. An attacker who can cause the system to operate on version i of the password database can gain unauthorized access using the publicized password. To summarize, TPM Unseal ensures that the plaintext of c' is accessible only to the intended PAL, but it does not guarantee that c' is the most recent sealed version of data d.

Replay attacks against sealed storage can be prevented if a secure counter is available, as illustrated in Table 3.1. To seal an updated data object, the secure counter should be incremented, and the data object should be sealed along with the new counter value. When a data object is unsealed, the counter value included in the data object at seal time should be the same as the current value of the secure counter. If the values do not match, either the counter was tampered with, or the unsealed data object is a stale version. In both cases, the data object is untrustworthy and should be discarded.

Options for realizing a secure counter with Flicker include a trusted third party, and the *Monotonic Counter* and *Non-volatile Storage* facilities of v1.2 TPMs [172]. We provide a sketch of how to implement replay protection for sealed storage with Flicker using the TPM's Non-volatile Storage facility.In particular, we do not treat recovery after a power failure or system crash during the counter-increment and sealed storage ciphertext-output. In these scenarios, the secure counter can become out-of-sync with the latest sealed-storage ciphertext maintained by the OS. An appropriate mechanism to detect such events is also necessary.

The TPM's *Non-volatile Storage* facility exposes interfaces to *Define Space*, and *Read* and *Write* values to defined spaces. Space definition is authorized by demonstrating possession of the 20-byte TPM *Owner Authorization Data*, which can be provided to a Flicker session using the protocol we present in Section 3.2.4. A defined space can be configured to restrict access based on the contents of specified PCRs. Setting the PCR requirements to match those specified during the TPM Seal command creates an environment where a counter value stored in non-volatile storage is only available to the desired PAL. Values placed in non-volatile storage are maintained in the TPM, so there is no dependence on the untrusted OS to store a ciphertext. This, combined with the PCR-based access control, is sufficient to protect a counter value against attacks from the OS.

3.2.4 Interaction With a Remote Party

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

3.2.4.1 Attestation and Result Integrity

A platform using Flicker can convince remote parties that a Flicker session executed with a particular PAL. Our approach builds on the TPM attestation process described in Chapter 2.3. Below, we refer to the party executing Flicker as the *challenged party*, and the remote party as the *verifier*.

To create an attestation, the challenged party accepts a random nonce from the verifier to provide freshness and replay protection. The challenged party then uses Flicker to execute a particular PAL as described in Section 3.2.2. As part of Flicker's execution, the *SKINIT* instruction resets the value of PCR 17 to 0 and then extends it with the measurement of the PAL. Thus, PCR 17 will take on the value $V \leftarrow H(0\text{x}00^{20}||H(P))$, where

P represents the PAL code. The properties of the TPM, chipset, and CPU guarantee that no other operation can cause PCR 17 to take on this value. Thus, an attestation of the value of PCR 17 will convince a remote party that the PAL was executed using Flicker's protection.

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

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

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

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

3.2.4.2 Establishing a Secure Channel

The techniques described above ensure the integrity of the PAL's input and output, but to communicate securely (i.e., with both secrecy and integrity

protections) with a remote party, the PAL and the remote party must establish a secure channel. We create a secure channel by combining multiple Flicker sessions, the attestation capabilities just described, and some additional cryptographic techniques (described below).

We first describe the intuition behind the protocol, and then describe it in detail. The PAL 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 PAL using the technique described above. An attestation convinces the remote party that the PAL ran with Flicker's protections and that the public key was a legitimate output of the PAL. Finally, the remote party can use the PAL's public key to create a secure channel [63] to the PAL. We need not include communication software (such as network drivers) in the PAL's TCB, since we can use multiple invocations of a PAL to process data from the remote party while letting the untrusted OS manage the encrypted network packets.

We now describe the protocol (Table 3.2) for securely conveying a public key from the PAL to a remote party for use in establishing a secure channel. This protocol is similar to one developed at IBM for linking remote attestation to secure tunnel endpoints [59].

The PAL uses randomness from the TPM to generate an asymmetric keypair $\{K_{\text{PAL}}, K_{\text{PAL}}^{-1}\}$ within the protection of a Flicker session. The private key is then sealed for a future invocation of the same PAL using the technique described in Section 3.2.3. The TPM guarantees that no other code (not even a different PAL) can access the private key. Note that the PAL developer may extend other application-dependent data into PCR 17 before sealing the private key. This ensures the key will be released only if that application-dependent data is present.

The public portion of the key is extended into PCR 17 as an output parameter from the Flicker session. The OS sends the public key to the remote party along with an attestation proving that the PAL ran with Flicker's protections and that the public key was a legitimate output of the PAL that ran. The remote party uses the PAL's public key to create a secure channel [63] to the PAL. Since only the PAL executed with Flicker's protections can ac-

cess the private key, only the PAL can decrypt communications from the remote party. Since the PAL executes with Flicker's protections, no other code on the system can access the contents of the remote party's messages.

The nonce value sent by the remote party for the TPM quote operation is also provided as an input to the PAL for extension into PCR 18. This provides the remote party with a different freshness guarantee: that the PAL was invoked in response to the remote party's request. Otherwise, a malicious OS may be able to fool multiple remote parties into accepting the same public key.

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

Our implementation of Flicker makes the above protocol available as a module that developers can include with their PAL. We discuss this further in Section 3.3. We make use of this module in our SSH application, and thus we will revisit this protocol in Section 3.4.3.1.

3.3 Developer's Perspective

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

3.3.1 Creating a PAL

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

Remote Party (RP):	has AIK_{server}, expected hash(PAL || shim) = $\hat{H}$
RP:	generate *nonce*
RP → App:	*nonce*
App → PAL:	*nonce*
PAL:	extend($PCR18, nonce$) generate $\{K_{\text{PAL}}, K_{\text{PAL}}^{-1}\}$ extend($PCR18, h(K_{\text{PAL}})$) seal($PCR17, K_{\text{PAL}}^{-1}$) extend($PCR17, \perp$) extend($PCR18, \perp$)
PAL → App:	K_{PAL}
App:	$q \leftarrow$ quote($nonce, \{17, 18\}$)
App → RP:	q, K_{PAL}
RP:	if ($\neg$Verify($AIK_{server}, q, nonce$) $\vee\ q.PCR17 \neq h(h(0\|\|\hat{H})\|\|\perp)$ $\vee\ q.PCR18 \neq$ $h(h(h(0\|\|nonce)\|\|h(K_{\text{PAL}}))\|\|\perp)$) then abort
RP:	has authentic K_{PAL} knows server ran Flicker

Table 3.2: Protocol to generate and convey the public key K_{PAL} to a remote party (RP). Note that the messages between the application (App) and the PAL can safely travel through the untrusted portion of the application and the OS kernel. $\perp$ denotes a well-known value which signals the end of extensions performed within the Flicker session.

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

Table 3.3: A simple PAL that ignores its inputs, and outputs "Hello, world." `PAL_OUT` is defined in `slbcore.h`.

Module	Properties	LOC	Size (KB)
SLB Core	Prepare env., exec. PAL, clean env., resume OS	94	0.312
OS Protection	Memory protection, ring 3 PAL execution	5	0.046
TPM Driver	Low-level TPM communication	216	0.825
TPM Utils	TPM operations, e.g., Seal, GetRand, Extend	889	9.427
Crypto	Cryptographic library, e.g., RSA, SHA-1, etc.	2262	31.380
Memory Utils	Implementation of malloc/free/realloc	657	12.511
Sec. Chan.	Gen. keypair, seal priv. key, return pub. key	292	2.021

Table 3.4: Modules that can be included in the PAL. Only the SLB Core is mandatory. Each adds some number of lines of code (LOC) to the PAL's TCB and contributes to the overall size of the SLB binary.

3.3.1.1 A "Hello, World" Example PAL

As an example, Table 3.3 illustrates a simple PAL that ignores its inputs, and outputs the classic message, "Hello, world." Essentially, the PAL copies the contents of the global `msg` variable to the well-known PAL output parameter location (defined in the `slbcore` header file). Our convention is to use the second 4-KB page above the 64-KB SLB. The PAL code, when built using the process described below, can be executed with Flicker protections. Its message will be available from the `outputs` entry in the *flicker-module* sysfs location. Thus the application can simply use `open` and `read` to obtain the PAL's results.

3.3.1.2 Building a PAL

To convert the code from Table 3.3 into a PAL, we link it against the object file representing Flicker's core functionality (described as SLB Core below) using the Flicker linker script. The linker script specifies that the skeleton data structures and code from the SLB Core should come first in the resulting binary, and that the resulting output format should be binary (as opposed to an ELF executable). The application then provides this binary blob to the *flicker-module* for execution under Flicker's protection.

Application developers depend on a variety of libraries. There is no reason this should be any different just because the target executable is

a PAL, except that it is desirable to modularize the libraries further than is traditionally done to help minimize the amount of code included in the PAL's TCB. We have developed several small libraries in the course of applying Flicker to the applications described in Section 3.4. The following paragraphs provide a brief description of the libraries listed in Table 3.4.

SLB Core. The SLB Core module provides the minimal functionality needed to support a PAL. Section 3.2.2 describes this functionality in detail. In brief, the SLB Core contains space for the SLB's entry point, length, GDT, TSS, and code to manage segment descriptors and page tables. The SLB Core transfers control to the PAL code, which performs application-specific work. When the PAL terminates, it transfers control back to the SLB Core for cleanup and resumption of the OS.

OS Protection. Thus far, Flicker has focused on protecting a security-sensitive PAL from all of the other software on the system. However, we have also developed a module to protect a legitimate OS from a malicious or malfunctioning PAL. It is important to note that since *SKINIT* is a privileged instruction, only code executing at CPU protection ring 0 (recall that x86 has 4 privilege rings, with 0 being most privileged) can invoke a Flicker session. Thus, the OS ultimately decides which PALs to run, and presumably it will only run PALs that it trusts or has verified in some manner, e.g., using proof carrying code [121]. Nonetheless, the OS may desire additional guarantees. The OS Protection module restricts a PAL's memory accesses to the exact memory region allocated by the OS, thus preventing it from intentionally or inadvertently reading or overwriting the code and/or data of other software on the system. We are also investigating techniques to limit a PAL's execution time using timer interrupts in the SLB Core.

Aside from availability concerns, a malicious PAL that is allowed to run forever can tamper with the system's System Management Mode handlers and potentially set the system's cooling fans to their lowest setting. The malicious PAL may then perform CPU-intensive work, risking hardware

damage. These timing restrictions must be chosen carefully, however, since a PAL may need some minimal amount of time to allow TPM operations to complete before the PAL can accomplish any meaningful work. Alternatively, a malicious PAL may install malicious SMM handlers and then exit, assuming that the normal course of system activity will eventually involve CPU-intensive work. To prevent this attack, the OS should either inspect PAL code to ensure that it does not install a new SMM handler, or only execute PALs from a trusted source (that presumably does not include an SMM attack in their PALs).

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

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

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

The TPM Utilities allow other PAL code to perform useful TPM operations. Currently supported operations include GetCapability, PCR Read,

PCR Extend, GetRandom, Seal, Unseal, and the OIAP and OSAP sessions necessary to authorize Seal and Unseal [172].

Crypto. We have developed a small library of cryptographic functions. Supported operations include a multi-precision integer library, RSA key generation and encryption, 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 3.2.4 for creating a secure channel into a PAL from a remote party. It relies on all of the other modules we have developed (except the OS Protection module which the developer may add).

3.3.2 Automation

Ideally, we envision each PAL containing only the security-sensitive portion of each application, rather than the application in its entirety. Minimizing the PAL makes it easier to ensure that the required functionality is performed correctly and securely, facilitating a remote party's verification task. Previous research indicates that many applications can be readily split into a privileged and an unprivileged component. Such privilege separation can be performed manually [90, 133, 168, 81], or automatically [24, 11, 186].

While each PAL is necessarily application-specific, we have developed a tool using the source-code analysis tool CIL [122] to help extract functionality from existing programs. Since CIL can replace the C compiler (e.g., the programmer can simply run "`CC=cil make`" using an existing Makefile), our tool can operate even on large programs with complex build dependencies.

The programmer supplies our tool with the name of a target function within a larger program (e.g., `rsa_keygen()`). The tool then parses the program's call graph and extracts any functions that the target depends on, along with relevant type definitions, etc., to create a standalone C program.

The tool also indicates which additional functions from standard libraries must be eliminated or replaced. For example, by default, a PAL cannot call `printf` or `malloc`. Since `printf` usually does not make sense for a PAL, the programmer can simply eliminate the call. For `malloc`, the programmer can convert the code to use statically allocated variables or link against our memory management library (described above). While the process is clearly not completely automated, the tool does automate a large portion of PAL creation and eases the programmer's burden, and we continue to work on increasing the degree of automation provided. We found the tool useful in our application of Flicker to the applications described next.

3.4 Flicker Applications

In this section, we demonstrate the versatility of the Flicker platform by showing how Flicker can be applied to several broad classes of applications. Within each class, we describe our implementation of one or more applications and show how Flicker significantly enhances security in each case. In Section 3.5, we evaluate the performance of the applications, as well as the general Flicker platform.

We have implemented Flicker for AMD SVM on a 32-bit Linux kernel v2.6.20, including the various modules described in Section 3.3. Each application described below utilizes precisely the modules needed (and some application-specific logic) and nothing else. On the untrusted OS, the *flicker-module* loadable kernel module is responsible for invoking the PAL and facilitating delivery of inputs and reception of outputs from the Flicker session. Further, it manages the suspension and resumption of the untrusted OS before and after the Flicker session. We also developed a TPM Quote Daemon (the *tqd*) on top of the TrouSerS[4] TCG Software Stack that runs on the untrusted OS and provides an attestation service.

[4] `http://trousers.sourceforge.net/`

3.4.1 Stateless Applications

Many applications do not require long-term state to operate effectively. For these applications, the primary overhead of using Flicker is the time required for the *SKINIT* instruction, since the attestation can be generated by the untrusted OS (see Section 3.2.4.1). As a concrete example, we use Flicker to provide verifiable isolated execution of a kernel rootkit detector on a remote machine.

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

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

3.4.2 Integrity-Protected State

Some applications may require multiple Flicker sessions, and hence a means of preserving state across sessions. For some, simple integrity protection of this state will suffice (we consider those that also require secrecy in Section 3.4.3). To illustrate this class of applications, we apply Flicker to a distributed computing application.

Applications such as SETI@Home [6] 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 [118]. 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 [5], which is a generic framework for distributed computing applications. It is currently used by several dozen projects.[5] By targeting BOINC, rather than a specific application, we can allow all of these applications to take advantage of Flicker's security properties (though some amount of application-specific modifications are still required). As an illustration, we developed a simple distributed application using the BOINC framework that attempts to factor a large number by naively asking clients to test a range of numbers for potential divisors.

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

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

Before yielding control back to the untrusted OS, the PAL computes a

[5] `http://boinc.berkeley.edu/projects.php`

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

3.4.3 Secret and Integrity-Protected State

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

3.4.3.1 SSH Password Authentication

We have applied Flicker to password-based authentication with SSH. Since people tend to use the same password for multiple independent computer systems, a compromise on one system may yield access to other systems. Our primary goal is to prevent any malicious code on the server from learning the user's password, even if the server's OS is compromised. Our secondary goal is to convince the client system (and hence, the user) that the secrecy of the password has been preserved. Flicker is well suited to these goals, as it makes it possible to restrict access to the user's cleartext password on the server to a tiny TCB (the PAL), and to attest to the client that this indeed was enforced. While other techniques (e.g., PwdHash [138]) 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 de-

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

Table 3.5: The protocol surrounding the second Flicker session for our SSH implementation. *sdata* contains the sealed private key, K_{PAL}^{-1}. Variables *salt* and *hashed_passwd* are components of the entry in the system's `/etc/passwd` file for the user attempting to log in. The *nonce* serves to prevent replay attacks against a well-behaved server.

scribed in the preceding sections, and consists of five main software components. A modified SSH client runs on the client system. The client system does not need hardware support for Flicker, but a compromise of the client may leak the user's password. We are investigating techniques for utilizing Flicker on the client side. We add a new client authentication method, *flicker-password*, to OpenSSH version 4.3p2. The *flicker-password* module establishes a secure channel to the PAL on the server using the protocol described in Section 3.2.4.2 and implements the client portion of the protocol shown in Table 3.5.

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

First Flicker Session (Setup). The first session uses our Secure Channel module to provide the client 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_{PAL} to the client in such a way that the client is convinced that the corresponding private key is accessible only to the same PAL in a subsequent Flicker session. Thus, by verifying the attestation from the first Flicker session, the client is convinced that the correct PAL executed, that the legitimate PAL created a fresh keypair, and that the SLB Core erased all secrets before returning control to the untrusted OS. Using its authentic copy of K_{PAL}, the client encrypts the user's password for transmission to the second Flicker session on the server. We use PKCS1 encryption which is chosen-ciphertext-secure and nonmalleable [85]. The end-to-end encryption of the user's password, from the client system all the way into the PAL, protects the user's password in the event that any of the server's software is malicious.

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

password file (see Table 3.5).

When the second session begins, the PAL uses TPM Unseal to retrieve its private key K_{PAL}^{-1} from *sdata*. It then uses the key to decrypt the user's password. Finally, the PAL computes the hash of the user's password and salt[6] and outputs the result for comparison with the server's password file. The end result is that the user's unencrypted password only exists on the server during a Flicker session.

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

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

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

3.4.3.2 Certificate Authority

Our final application, a Flicker-enhanced Certificate Authority (CA), is similar to the SSH application but focuses on protecting the CA's private signing key. The benefit of using Flicker is that only a tiny piece of code ever has access to the CA's private signing key. Thus, the key will remain secure, even if all of the other software on the machine is compromised. Of course, malevolent code on the server may submit malicious certificates to the signing PAL. However, the PAL can implement arbitrary access control policies

[6] 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`).

on certificate creation and can log those creations. Once the compromise is discovered, any certificates incorrectly created can be revoked. In contrast, revoking a CA's public key, as would be necessary if the private key were compromised, is a more heavyweight proposition in many settings.

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

3.5 Performance Evaluation

Below, we describe our experimental setup and evaluate the performance of the Flicker platform. We begin with macrobenchmarks of the various applications described in Section 3.4, and then perform microbenchmarks to better identify the most significant sources of overhead.

While the overhead for several applications is significant, we have identified several hardware modifications that improve performance by up to six orders of magnitude. These modifications are discussed in Section 6. Thus, it is reasonable to expect significantly improved performance in future versions of this technology.

Finally, we evaluate the impact of Flicker sessions on the rest of the system, e.g., the untrusted OS and applications.

3.5.1 Experimental Setup

Our primary test machine is an HP dc5750 which contains an AMD Athlon64 X2 Dual Core 4200+ processor running at 2.2 GHz, and a v1.2 Broadcom BCM0102 TPM. In experiments requiring a remote verifier, we use a generic PC with an AMD Opteron CPU running at 1.6 GHz. The remote verifier is

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

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`.

3.6 End-to-End Application Macrobenchmarks

Here, we evaluate the high-level performance overhead of the four applications that we have extended with Flicker (Section 3.4). We divide our analysis between those applications maintaining no long-term state, integrity-protected state, and secret state. We then summarize the common overheads among all of the applications.

3.6.1 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 ma-

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

Table 3.7: Performance impact of the Rootkit Detector. Kernel build time when run with no detection and with rootkit detection run periodically. Note that the detection does not actually speed up the build time; rather the small performance impact it does have is lost in experimental noise.

chine. 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 instrumentation to determine the most expensive operations involved (Table 3.6). The results indicate that the highest overhead comes from the TPM Quote operation. This performance is TPM-specific. We discuss the performance of other TPMs in Section 3.7.3.

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 3.7 summarizes our results. Essentially, our results suggest that even frequent execution of the rootkit detector (e.g., once every 30 seconds) has negligible impact on the system's overall performance.

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 3.8: Operations for Distributed Computing. This table indicates the significant expense of the Unseal operation, as well as the tradeoff between efficiency and latency. We achieve this *SKINIT* time using an optimization presented in Section 3.7.1.

3.6.2 Integrity-Protected State

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

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

Table 3.8 shows the resulting overhead, as well as its most expensive constituent operations, in particular, the time for the *SKINIT*, and the time to unseal and verify the PAL's previous state.[7] The table demonstrates how the application's efficiency improves as we allow the PAL to run for longer periods of time before exiting back to the untrusted OS. For example, if the application runs for one second before returning to the OS, only 53% of the Flicker session is spent on application work; the remaining 47% is consumed by Flicker's setup time. However, if we allow the application to run for two or four seconds at a time, then Flicker's overhead drops to only 30% or 18%,

[7] As described in Section 3.4.2, the initial PAL must also generate a symmetric key and seal it under PCR 17. We discuss this overhead in more detail in Section 3.6.3.

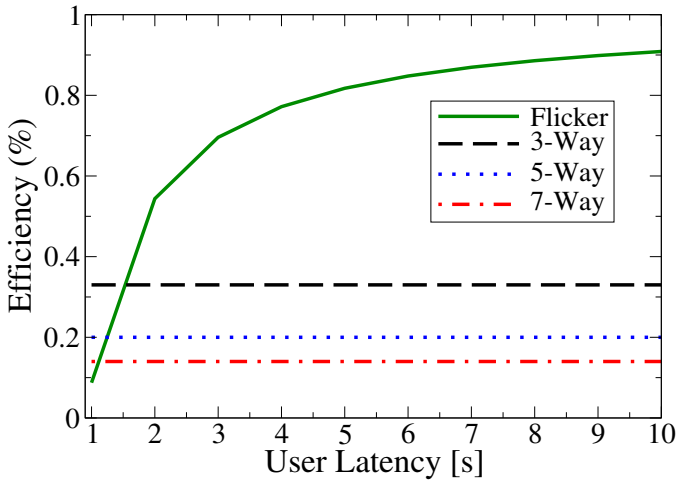


Figure 3.4: Flicker vs. Replication Efficiency. Replicating to a given number of machines represents a constant loss in efficiency. Flicker gains efficiency as the length of the periods during which application work is performed increases.

respectively. Table 3.8 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 3.4 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.

3.6.3 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.

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

(a) **PAL 1**

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

(b) **PAL 2**

Figure 3.5: SSH Overhead. Average server side performance over 100 trials, including a breakdown of time spent inside each PAL. The standard error on all measurements is under 1%, except key generation at 14%. We achieve this *SKINIT* time using an optimization presented in Section 3.7.1.

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

Figure 3.5 presents these results, as well as a breakdown of the most expensive operations that execute on the SSH server. The total time elapsed on the client between the establishment of the TCP connection with the server, and the display of the password prompt for the user is 1221 ms (this includes the overhead of the first PAL, as well as 949 ms for the TPM Quote operation), compared with 210 ms for an unmodified server. Similarly, the time elapsed beginning immediately after password entry on the client, and ending just before the client system presents the interactive session to the user is approximately 940 ms while the unmodified server only requires 10 ms. The primary source of overhead is clearly the TPM. As these devices have just been introduced by hardware vendors and have not yet proven themselves in the market, it is not surprising that their performance is poor. Nonetheless, current performance suffices for lightly-loaded servers, or for less time-critical applications, such as the CA.

During the first PAL, the 1024-bit key generation clearly imposes the largest overhead. This cost could be mitigated by choosing a different public key algorithm with faster key generation, such as ElGamal, and is readily

parallelized. Both Seal and *SKINIT* contribute overhead, but compared to the key generation, they are relatively insignificant. We also make one call to TPM GetRandom to obtain 128 bytes of random data (it is used to seed a pseudorandom number generator), which averages 1.3 ms. The performance of PCR Extend is similarly quick and takes less than 1 ms on the Broadcom TPM.

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

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

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

3.6.4 Summary of High-Level Flicker Overheads

Here, we categorize the overheads incurred by our Flicker-enabled applications, so that we can establish a broader baseline for what kind of performance is available today using microbenchmarks in Section 3.7. We focus on the performance of two generic PALs. The first PAL (PAL *Gen*) launches, generates application-specific data, seals the data using the TPM's sealed storage capability, and exits. The second PAL (PAL *Use*) launches, unseals data sealed during a previous session, operates on that data, reseals the data, and exits.

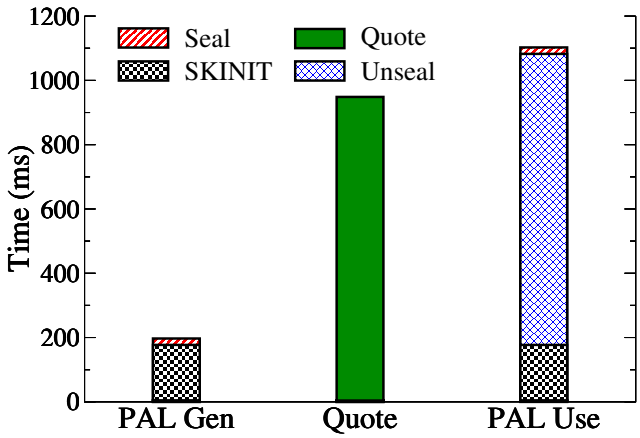


Figure 3.6: Breakdown of overheads that will be incurred by generic applications implemented with Flicker. Measurements were taken using an HP dc5750 containing an AMD processor and a Broadcom TPM. PAL Gen represents the overhead for an application that generates data and seals it for later use. PAL Use unseals previous state, modifies it, and reseals it.

Figure 3.6 summarizes our results (taken over 100 runs with negligible variance) and indicates both the total time taken by each PAL, as well as the breakdown of the overhead for each. Note that these numbers represent pure overhead—the time necessary for application-specific work is added on top of these measurements. We also include the time required to perform a TPM Quote operation, since this operation is needed to create an attestation that will convince an external party that a PAL was executed successfully.

Looking at the breakdown of the execution time, each PAL requires a late launch, represented by the *SKINIT* region (the PAL uses the full 64 KB supported by AMD). The PAL Gen session experiences the additional overhead of sealing data using the TPM's 2048-bit RSA Storage Root Key. The PAL Use session must perform a TPM Unseal, and may also perform a Seal operation before exiting. Both TPM Quote and TPM Unseal perform a private RSA operation (digital signature and decrypt, respectively), which is their dominant source of overhead.

Our results indicate that the TPM's role in protecting PAL state during a context-switch creates significant amounts of overhead. Storing data for

later use requires approximately 200 ms (PAL Gen), but accessing, modifying, and then storing state (PAL Use) requires over a second. Note also that this experiment was run on the Broadcom TPM, which had the fastest seal operation of all TPMs that we tested, as we discuss with our microbenchmarks in Section 3.7.

The above overheads are exacerbated by the constraint that no other code can execute during PAL execution. Thus, while a PAL Use module executes, all other operations on the computer will be suspended for over a second. This overhead is particularly egregious on a multi-processor machine, as the late launch operation requires all but one of the processors to be in a special idle state. As a result, most of the computer's processing power and responsiveness vanish for over a second during PAL execution.

3.6.5 Impact on Suspended Operating System

Flicker runs with the legacy OS suspended and interrupts disabled. We have presented Flicker sessions that run for more than one second, e.g., in the context of a distributed computing application (Table 3.8). While these are long times to keep the OS suspended and interrupts disabled, we have observed relatively few problems in practice. We relate some of our experience with Flicker and describe the options available today to reduce Flicker's impact on the suspended system.

While a Flicker session runs, the user will perceive a hang on the machine. Keyboard and mouse input during the Flicker session may be lost. Such responsiveness glitches sometimes occur even without Flicker, and while unpleasant, they do not put valuable data at risk. Likewise, network packets are sometimes lost even without Flicker, and today's network-aware applications can and do recover. The most significant risk to a system during a Flicker session is lost data in a transfer involving a block device, such as a hard drive, CD-ROM drive, or USB flash drive.

We have performed experiments on our HP dc5750 copying large files while the distributed computing application runs repeatedly. Each run lasts an average of 8.3 seconds, and the legacy OS runs for an average of 37 ms in

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

To provide stronger guarantees for the integrity of device transfers on a system that supports Flicker, these transfers should be scheduled such that they do not occur during a Flicker session. This requires OS awareness of Flicker sessions so that it can quiesce devices appropriately. Modern devices already support suspension in the form of ACPI power events [69], although this is sub-optimal since power will remain available to devices. The best solution is to modify the relevant OS schedulers to be Flicker-aware, so that minimal work is required to prepare for a Flicker session. We plan to further investigate Flicker-aware device drivers and OS extensions, but the best solution may be an architectural change for next-generation hardware (Section 6).

3.7 Microbenchmarks

To determine if the application overheads described in the preceding section are representative of current hardware, we perform a number of microbenchmarks to measure the time needed by late launch and various TPM operations on two AMD machines and one Intel machine.

In addition to the AMD HP dc5750 described above, we employ a second AMD test machine based on a Tyan n3600R server motherboard with two 1.8 GHz dual-core Opteron processors. This second machine is not equipped with a TPM, but it does support execution of *SKINIT*. This allows us to

TPM	CPU Vendor	Overhead in ms for various PAL sizes					
		0 KB	4 KB	8 KB	16 KB	32 KB	64 KB
Yes	**AMD**	0.00	11.94	22.98	45.05	89.21	177.52
No		0.01	0.56	1.11	2.21	4.41	8.82
Yes	**Intel**	26.39	26.88	27.38	28.37	30.46	34.35

Table 3.9: *SKINIT* and *SENTER* benchmarks. We run *SKINIT* benchmarks on AMD systems with (an HP dc5750) and without (a Tyan n3600R) a TPM to isolate the overhead of the *SKINIT* instruction from the overhead induced by the TPM. We also run *SENTER* benchmarks on an Intel Technology Enabling Platform (TEP) with a TPM.

isolate the performance of *SKINIT* without the potential bottleneck of a TPM. Our Intel test machine is an MPC ClientPro Advantage 385 TXT Technology Enabling Platform (TEP), which contains a 2.66 GHz Core 2 Duo processor, an Atmel v1.2 TPM, and the DQ965CO motherboard.

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

3.7.1 Late Launch with an AMD Processor

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

Table 3.9 summarizes the timing results. The measurements for the empty (0 KB) PAL indicate that placing the CPU in an appropriate state

introduces relatively little overhead (less than 10 μs). The Tyan n3600R (without TPM) allows us to measure the time needed to transfer the PAL across the LPC bus. The maximum LPC bandwidth is 16.67 MB/s, so the fastest possible transfer of 64 KB is 3.8 ms [74]. Our measurements agree with this prediction, indicating that it takes about 8.8 ms to transfer a 64 KB PAL, with the time varying linearly for smaller PALs.

Unfortunately, our results for the HP dc5750 indicate that the TPM introduces a significant delay to the *SKINIT* operation. We investigated the cause of this overhead and identified the TPM as causing a reduction in throughput on the LPC bus. The TPM slows down *SKINIT* runtime by causing *long wait cycles* on the LPC bus. *SKINIT* sends the contents of the PAL to a TPM to be hashed using the following TPM command sequence: `TPM_HASH_START`, zero or more invocations of `TPM_HASH_DATA` (each sends one to four bytes of the PAL to the TPM), and finally `TPM_HASH_END`. The TPM specification states that each of these commands may take up the entire *long wait cycle* of the control flow mechanism built into the LPC bus that connects the TPM [171]. Our results suggest that the TPM is indeed utilizing most of the *long wait cycle* for each of the commands, and as a result, the TPM contributes almost 170 ms of overhead. This may be either a result of the TPM's low clock rate or an inefficient implementation, and is not surprising given the low-cost nature of today's TPM chips. The 8.82 ms taken by the Tyan n3600R may be representative of the performance of future TPMs which are able to operate at maximum bus speed.

***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 the size of the SLB. To maintain the security properties provided by *SKINIT*, however, code in the SLB must be measured before it is executed. Note that *SKINIT* enables the Device Exclusion Vector for the entire 64 KB of memory starting from the base of the SLB, even if the SLB's length is less than 64 KB. One viable optimization is to create a PAL that only includes a cryptographic hash function and enough TPM support to perform a PCR

Extend. This PAL can then measure and extend the application-specific PAL. A PAL constructed in this way offloads most of the burden of computing code measurement to the system's main CPU. We have constructed such a PAL in 4736 bytes. When this PAL runs, it measures the entire 64 KB and extends the resulting measurement into PCR 17. Thus, when *SKINIT* executes, it only needs to transfer 4736 bytes to the TPM. In 50 trials, we found the average *SKINIT* time to be 14 ms. While only a small savings for the rootkit detector, it saves 164 ms of the 176 ms *SKINIT* requires with a 64-KB SLB. We used this optimization in our applications in Section 3.6.

3.7.2 Late Launch with an Intel Processor

Chapter 2.2 provides an introduction to Intel's late launch capabilities. Intel's late launch consists of two phases. First, the ACMod is extended into PCR 17 using the same `TPM_HASH_START`, `TPM_HASH_DATA`, and `TPM_HASH_END` command sequence used by AMD's *SKINIT*. The ACMod then hashes the PAL on the main CPU and uses an ordinary `TPM_Extend` operation to record the PAL's identity in PCR 18. Thus, only the 20 byte hash of the PAL is passed across the LPC to the TPM in the second phase.

The last row in Table 3.9 presents experimental results from invoking *SENTER* on our Intel TEP. Interestingly, the overhead of *SENTER* is initially quite high, and it grows linearly but slowly. The large initial overhead (26.39 ms) results from two factors. First, even for a 0 KB PAL, the Intel platform must transmit the entire ACMod to the TPM and wait for the TPM to hash it. The ACMod is just over 10 KB, which matches nicely with the fact that the initial overhead falls in between the overhead for an *SKINIT* with PALs of size 8 KB (22.98 ms) and 16 KB (45.05 ms). The overhead for *SENTER* also includes the time necessary to verify the signature on the ACMod.

The slow increase in the overhead of *SENTER* relative to the size of the PAL is a result of where the PAL is hashed. On an Intel platform, the ACMod hashes the PAL on the main CPU and hence sends only a constant amount of data across the LPC bus. In contrast, an AMD system must

send the entire PAL to the TPM and wait for the TPM to do the hashing. However, in practice, there is no need to incur *SKINIT* overhead beyond that required for a PAL that measures itself, as in Section 3.7.1. Table 3.9 suggests that for large PALs, Intel's implementation decision pays off. Further reducing the size of the ACMod would improve Intel's performance even more. The gradual increase in *SENTER*'s runtime with increase in PAL size is most likely attributable to the hash operation performed by the ACMod.

On an Intel TXT platform, the ACMod verifies that system configuration is acceptable, enables chipset protections such as the initial memory protections for the PAL, and then measures and launches the PAL [61]. On AMD SVM systems, microcode likely performs similar operations, but we do not have complete information about AMD CPUs. Since Intel TXT measures the ACMod into a PCR, an Intel TXT attestation to an external verifier may contain more information about the challenged platform and may allow an external verifier to make better trust decisions.

3.7.3 Trusted Platform Module (TPM) Operations

Though Intel and AMD send different modules of code to the TPM using the `TPM_HASH_*` command sequence, this command sequence is responsible for the majority of late launch overhead. More significant to overall PAL overhead, however, is Flicker's use of the TPM's sealed storage capabilities to protect PAL state during a context switch. To understand whether the generic overheads from Figure 3.6 are representative, we perform TPM benchmarks on four different TPMs. Two of these are the TPMs in our already-introduced HP dc5750 and Intel TEP. The other two TPMs are an Atmel TPM (a different model than that included in our Intel TEP) in an IBM T60 laptop, and an Infineon TPM in an AMD system.

We evaluate the time needed for relevant operations across several different TPMs. These operations are: PCR Extend, Seal, Unseal, Quote, and GetRandom. Figure 3.7 shows the results of our TPM microbenchmarks. The results show that different TPM implementations optimize different operations. The Broadcom TPM in our primary test machine is the slow-

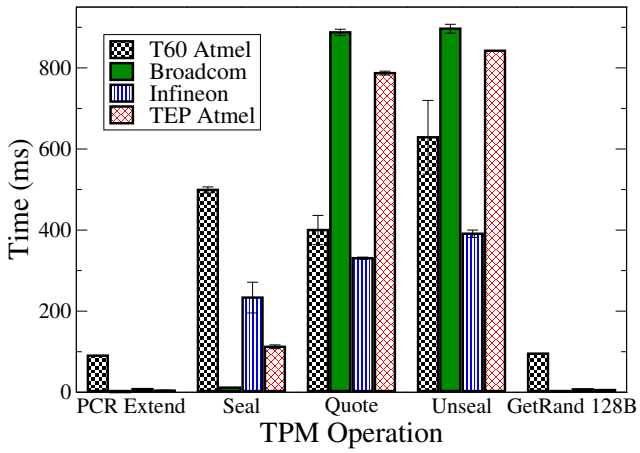


Figure 3.7: TPM benchmarks run against the Atmel v1.2 TPM in a Lenovo T60 laptop, the Broadcom v1.2 TPM in an HP dc5750, the Infineon v1.2 TPM in an AMD machine, and the Atmel v1.2 TPM (note that this is not the same as the Atmel TPM in the Lenovo T60 laptop) in the Intel TEP. Error bars indicate the standard deviation over 20 trials (not all error bars are visible).

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

3.7.4 Major Sources of Performance Problems

Our experiments reveal two significant performance bottlenecks for minimal TCB execution on current CPU architectures: (1) on a multi-CPU machine,

the inability to execute PALs and untrusted code simultaneously on different CPUs, and (2) the use of TPM Seal and Unseal to protect PAL state during a context switch between secure and insecure execution.

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

3.8 Summary

Flicker allows code to verifiably execute with hardware-enforced isolation. Flicker itself adds as few as 250 lines of code to the application's TCB. Given the correlation between code size and bugs in the code, Flicker significantly improves the security and reliability of the code it executes. New desktop and laptop machines already contain the hardware support necessary for Flicker, so widespread Flicker-based applications can soon become a reality.

With Flicker, application developers finally have the opportunity to write secure applications without relying on the security of layer upon layer of legacy software, and without breaking compatibility with today's commodity systems.

Chapter 4

Protecting Sensitive User Input with a Reduced Trusted Computing Base

Today's commodity web browsers and operating systems do not provide any facility by which a user can be sure that her sensitive information is reaching its intended destination unmolested. The Secure Socket Layer (SSL) provides cryptographic protection for data while it is in-transit on the network, but the prevalence of host-based malware such as keyloggers and screen scrapers suggests that a more complete mechanism is needed. It is also desirable to enable the user to confirm that such a mechanism is active at any moment. Indeed, we argue that the user should have the ability to control which of her inputs are deemed sensitive.

Ideally, we would construct a system that guarantees that all user input is directed exclusively to its intended destination, irrespective of the presence of malware on the user's system. We would like the destination to be able to distinguish between input arriving via a secure mechanism from input provided via today's legacy channels.

Flicker provides us with an isolation mechanism that can process data without exposing it to malware on the user's computer. On top of this we develop Bumpy, a system for protecting a user's sensitive input intended for

a webserver from a compromised client OS or compromised web browser. We consider a user who desires to provide strings of information (e.g., a credit card number or mailing address) to a remote webserver (e.g., her bank) by entering it via her web browser. We limit ourselves to a case study of user input to web pages, although our techniques can also be applied to local applications. Bumpy is able to protect this sensitive user input by reducing the requisite trusted computing base to exclude the legacy OS and applications without requiring a hypervisor or VMM.

Bumpy separates the process of accepting user input into trusted and untrusted parts, and thus can be viewed as implementing a type of privilege separation [144]. Bumpy employs two primary mechanisms. First, the initial handling of all keystrokes is performed in a special-purpose code module that is isolated from the legacy OS using the Flicker system (Chapter 3). Second, we establish the convention that sensitive input begin with the *secure attention sequence* `@@`, so that a user can indicate to this module that the data she is about to type is sensitive. These sensitive inputs are released to the legacy platform only after being encrypted for the end webserver or otherwise processed to protect user privacy [52, 51, 138].

Bumpy allows the remote webserver to configure the nature of the processing performed on user input before it is transmitted to the webserver, and automatically isolates the configurations and data-handling for mutually distrusting webservers. The webserver for which the user's current input will be processed can receive a TCG-style attestation that the desired input protections are in-place, potentially allowing the webserver to offer additional services to users with improved input security.

In order for the user to determine the website for which her input will be encrypted, she requires some trusted display to which the input-handling module can send this information. Since the client computer display cannot be trusted in our threat model, we explore the use of a separate user device, or Trusted Monitor, that receives such indicators from the input-handling module, authenticates them (using digital signatures) and displays them to the user.

Our prototype implementation of Bumpy demonstrates both the practi-

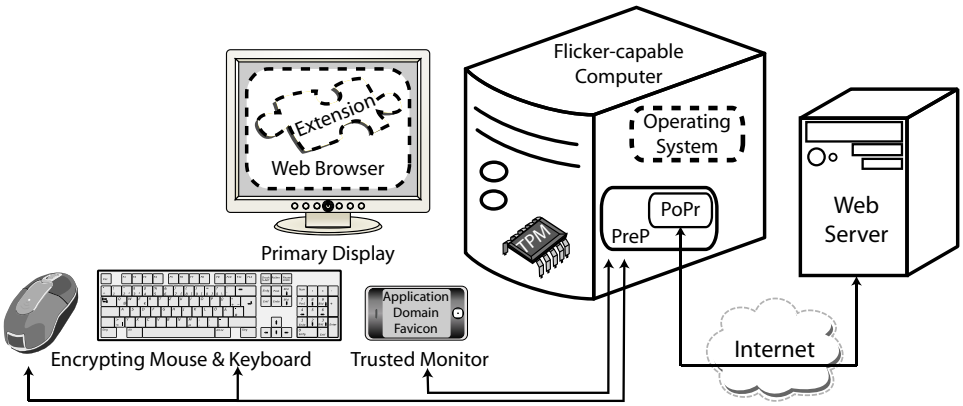


Figure 4.1: Logical flow through the major components of the Bumpy system. The OS, web browser, and browser extension are untrusted.

cality of our approach and the fact that commodity hardware already offers nearly the full set of functionality needed to achieve these protections. In fact, the only compromise we make in our implementation is using an embedded Linux system as an encrypting USB Interposer, as we have been unable to locate keyboards or mice offering programmable encryption. We also leverage a smartphone as a Trusted Monitor for the user. However, we emphasize that the emergence of encrypting keyboards and far simpler devices to serve as a Trusted Monitor would suffice to remove any bloat from Bumpy's TCB. Bumpy is achievable without any client-side trusted software of complexity even close to that of a general-purpose OS, VMM, or hypervisor.

4.1 Overview

We detail our goals and assumptions, introduce the user experience, and provide an overview of our design and the major system components of Bumpy (Figure 4.1).

4.1.1 Goals and Assumptions

Goals. Our goals are to protect keystrokes from a potentially malicious legacy input system while retaining a seamless user experience, and to offer assurance to both the remote webserver and the user herself that input is protected. To the remote webserver, we provide an attestation that the user's input was protected with Bumpy, including the presence of encryption-capable input devices. To the user, we provide an indicator of whether it is safe to enter sensitive input. Bumpy achieves this without breaking compatibility with existing operating systems and without requiring a hypervisor or VMM.

Assumptions and Threat Model. We consider the user's OS and applications (including the web browser and its extensions) to be malicious. We assume the user has a trustworthy mobile device to serve as a Trusted Monitor and input devices (keyboard and mouse) capable of encryption. We also assume the remote webserver to which the user wishes to direct her input is uncompromised, and that the certificate authority (CA) that issues the webserver's SSL certificate is similarly uncompromised.

We leverage the Flicker system to protect sensitive code executing on the user's computer (Chapter 3). As such, the user's computer must meet the hardware requirements for Flicker: a version 1.2 TPM, and a CPU and chipset capable of establishing a *Dynamic Root of Trust*, also known as *late launch*. Chapter 2 provides additional background on the relevant technologies, which are widely available today.

Physical attacks such as "shoulder surfing" and keyboard emanation attacks [187] are beyond the scope of Bumpy. Thus, we do not discuss them further.

4.1.2 User Experience

We are striving to make Bumpy usable by non-experts to protect sensitive input. Our mechanism employs a convention for entering sensitive information, and a trustworthy indication of the destination for that information.

This indication is conveyed via an external display, called the Trusted Monitor (Figure 4.1). It is our intention that the Trusted Monitor will help to alleviate some of the usability problems (e.g., a lack of feedback) identified for password managers such as PwdHash [32], although we leave a formal usability study as future work.

In the common case, the user experience with Bumpy follows this sequence:

1. The user signals that she is about to enter sensitive information by pressing `@@`. Note that this can be thought of as a convention, e.g., "my passwords should always start with `@@`."
2. The Trusted Monitor beeps to acknowledge the reception of `@@` in the PreP, and updates its display to show the destination of the user's upcoming sensitive input.
3. The user types her sensitive data. Bumpy does not change this step from the user's perspective.
4. The user performs an action that signals the end of sensitive input (e.g., presses Tab or Enter, or clicks the mouse). Bumpy does not change this step from the user's perspective.

While users are accustomed to typing their passwords without seeing the actual characters on-screen (e.g., the characters appear as asterisks), most other sensitive data is displayed following entry. Given our desire to remove the legacy OS from the input TCB and the threat of malicious screen scrapers, this echoing to the main display must be prevented by Bumpy. The usability of entering relatively short sequences of characters (e.g., credit card numbers) under these conditions may remain acceptable to concerned users, but it is not ideal. We perceive this as the price one must pay for secure input with an untrusted OS.

For those users employing a Trusted Monitor of sufficient capability, sensitive keystrokes can be echoed there for validation by the user. While this partially eliminates the challenge of entering input "blind," a minimal Trusted Monitor would still make it impractical to compose lengthy messages.

4.1.3 Technical Overview

We now summarize the main components of Bumpy. In Figure 4.1, solid arrows represent logical communication through encrypted tunnels. Bumpy is built around encryption-capable input devices sending input events directly into a Pre-Processor (PreP) protected by the Flicker system on the user's computer. Bumpy allows the remote webserver to control (within certain limits) how users' sensitive input is processed after it is entered with Bumpy. We term this Post-Processing, and enable it by allowing the webserver to provide a post-processor (PoPr) along with web content. Bumpy tracks and isolates PoPrs from different webservers, as well as supports standardized PoPrs that may be used across many websites. Leveraging the Flicker system (Chapter 3), the PreP and PoPrs execute in isolation from each other and from the legacy OS.

Encryption and password-hashing are two desirable forms of post-processing of user input. Site-specific hashing of passwords (as in PwdHash [138]) can provide password diversity across multiple websites, and prevent the webserver from ever having to handle the user's true password. Dedicated post-processing with server-supplied code can resolve issues with the PwdHash [138] algorithm producing unacceptable passwords (e.g., passwords without any punctuation characters that violate the site's password requirements) or passwords from a reduced namespace, since the webserver itself provides the algorithm. Encrypting input directly within the Bumpy environment to the remote webserver dramatically reduces the client-side TCB for sensitive user input.

4.2 Identifying and Isolating Sensitive Input

In this section, we focus on acquiring input from the user in the PreP, and storing sensitive input such that it is protected from the legacy OS. Section 4.3 treats the post-processing and delivery of this input to approved remote servers. We identify three requirements for protecting user input against a potentially malicious legacy OS:

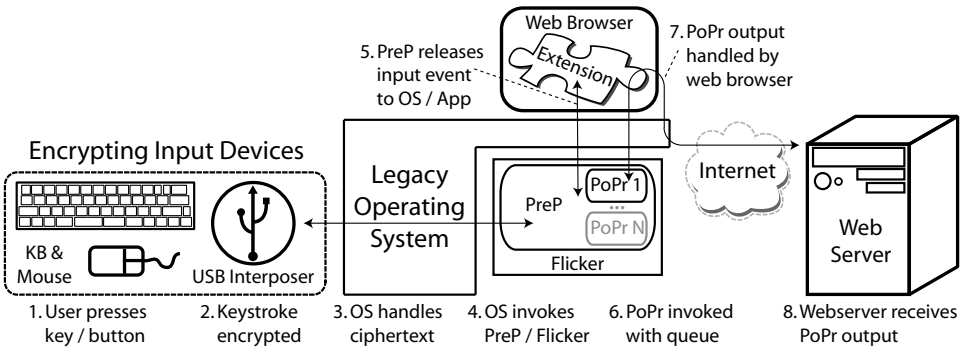


Figure 4.2: Acquiring user input with Bumpy. Steps 1–5 (described in Section 4.2) occur for every keystroke or mouse click performed by the user. Steps 6–8 (described in Section 4.3) occur only in response to a keystroke or mouse click that the PreP detects will cause a *blur* event (in the web browser GUI sense) while the user is entering sensitive data. We revisit this figure in Section 4.6.3 while describing the life of a keystroke within our implementation.

R1 All input must be captured and isolated.

R2 Sensitive input must be distinguishable from non-sensitive input.

R3 The final destination for sensitive input must be identifiable.

Requirement R1 for protecting user input is to acquire the input without exposing it to the legacy OS. The challenge here is that we wish to avoid dependence on a VMM or hypervisor and retain the OS in charge of device I/O. We propose to use encryption-capable input devices to send opaque input events through the untrusted OS to a special-purpose Piece of Application Logic (PAL) that is protected by the Flicker system (Steps 1–4 in Figure 4.2). This PAL is architected in two components. The first is specifically designed to Pre-Process encrypted input events from the input devices, and we call it the PreP. The PreP achieves requirement R2 by monitoring the user's input stream for the secure attention sequence "`@@`" introduced in Section 4.1.2, and then taking appropriate action (which affects what input event is released in Step 5 of Figure 4.2). The PreP serves as the source of input events for post-processing by a destination-specific Post-Processor

(PoPr). The process of authenticating a PoPr serves to identify the final destination for sensitive input (requirement R3). The PoPr encrypts or otherwise processes the received input for the remote server (Steps 6–8 in Figure 4.2).

These components are separated so that the PreP's sensitive state information can be kept isolated from the PoPr, as Bumpy supports multiple, mutually distrusting PoPrs that accept input events from the same PreP. The PreP's state information includes the cryptographic state associated with the encrypting input devices, the currently active PoPr, and a queue of buffered input events. The PreP's state is protected by encrypting it under a master key that is maintained on the user's TPM chip. The properties of Flicker (Chapter 3) guarantee that no code other than the exact PreP can access it. For the following sections we encourage readers not intimately familiar with trusted computing technology to read Chapter 2 before proceeding.

We defer discussion of the one-time setup of the cryptographic state associated with the encrypting input device(s) until Section 4.2.2. We proceed assuming that the setup has already been completed.

4.2.1 Steady-State User Input Protection

We describe the actions taken by the PreP in response to user input events and events from the web browser. The state machine in Figure 4.3 summarizes these actions.

Every event e is processed in a distinct Flicker session, i.e., the PreP only accepts a single event as an input parameter. We design Bumpy this way out of necessity, due to two conflicting desires. The first is to avoid trusting the OS, and the second is to remain responsive to the user as she provides input to her system. One consequence of this design is that every Flicker session (i.e., PreP invocation) begins and ends with the decryption and encryption of the PreP's sensitive state information, respectively.

The legacy OS provides arguments for each invocation of the PreP: the event e to be processed, the SSL certificate for the active website, the PoPr

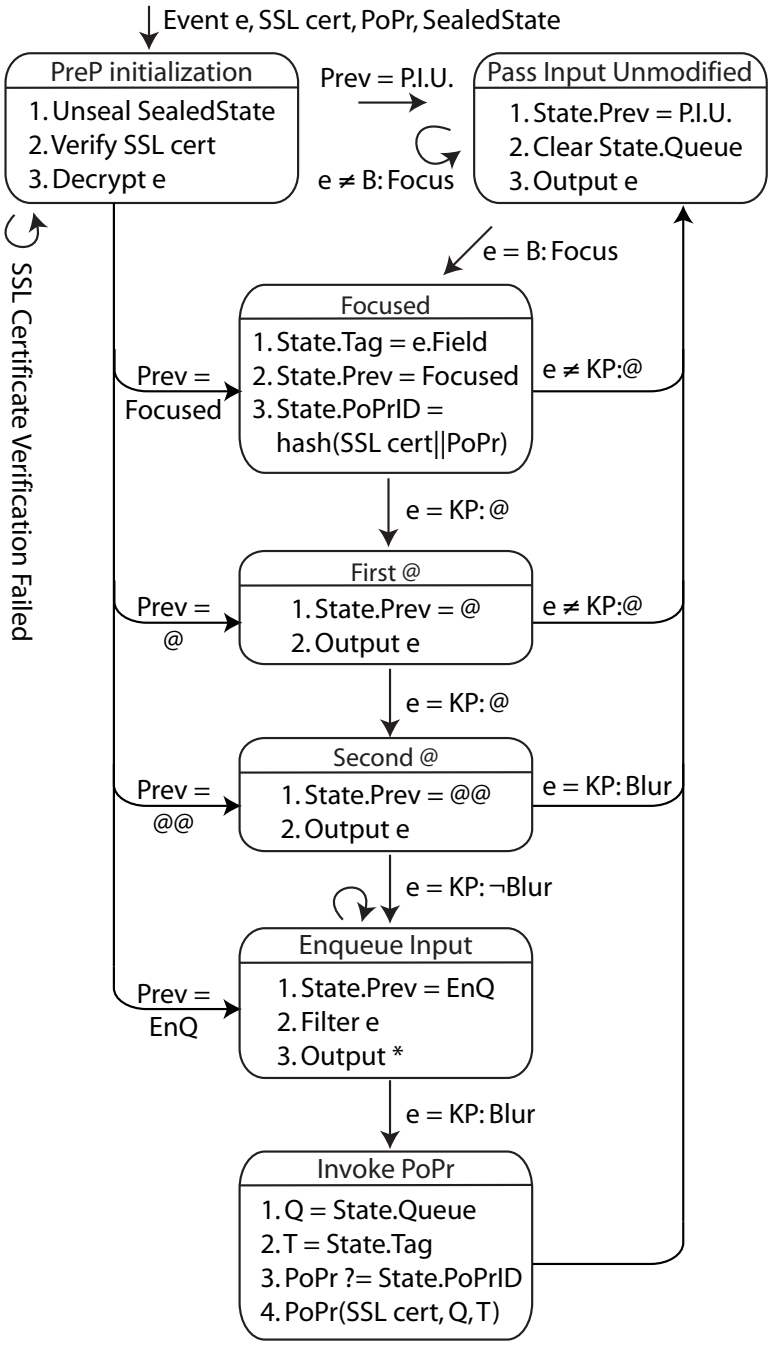


Figure 4.3: States of the PreP. KP = keypress or mouse click. B:Focus = browser GUI *focus* event. Blur indicates action taken in response to events on the encrypted input channel that cause a GUI *blur* event (e.g., Tab, Shift+Tab, Enter, or mouse click).

associated with the active website, and the PreP's encrypted state. Each event e can be an encrypted keystroke or mouse click, or it can be a *focus* event[1] from the browser. All other event types from the browser are ignored. The PreP maintains in its state the necessary cryptographic information to decrypt and integrity-check input events from the input device(s). The master keys used to protect the secrecy and integrity of the PreP's state are TPM-protected based on the identity of the PreP. We describe these protocols in greater detail as part of our implementation in Section 4.6.

During each run of the PreP (i.e., during each Flicker session in Step 4 of Figure 4.2), the state machine (Figure 4.3) begins in PreP Initialization and transitions to the state where the previous PreP invocation ended (maintained as State.Prev in Figure 4.3), where the current event then causes a single transition. Actions listed in a state are performed when an event causes arrival into that state (as opposed to returning to a state because of the value of State.Prev). If there is no action for a particular event in a particular state, then that event is ignored. For example, browser *focus* events are ignored in the Second ©, Enqueue Input, and Invoke PoPr states.

PreP Initialization. Regardless of the previous state of the PreP, it always performs an initialization step. The PreP first decrypts and integrity-checks its own long-term state, verifies that the provided SSL certificate is valid using its own list of trusted certificate authorities (which we define as being part of the PreP itself), and verifies that the provided PoPr is signed by the provided SSL certificate. (If any of these verification steps fail, the current event is dropped.) Next, the incoming event e is processed. If it is an encrypted input event from the input device(s), then it is decrypted, integrity-checked, and verified to be in-sequence (using cryptographic keys and a sequence number maintained in the PreP's state). If any of the steps involving synchronization with the input device(s) fail, then input events can no longer be received. We discuss options for recovery in Section 4.6.2.3.

[1] A *focus* event is an event in the web browser's graphical user interface where a new component such as an HTML text input field becomes active. This generally follows a *blur* event caused by the previously focused component becoming inactive. These events fire in response to user actions, such as clicking the mouse.

The PreP then transitions to State.Prev where e will cause one additional state transition. During the very first invocation of a PreP, it transitions to Pass Input Unmodified. The following paragraphs describe the actions taken upon entry to a state caused by an event, not by State.Prev. At the end of each of these states, the PreP's sensitive long-term state is *sealed*[2] using the TPM-protected master key, and then cleared (set to zero) before the PreP terminates and produces output. Untrusted code running on the OS maintains the ciphertext that makes up the PreP's sealed state and provides it as input to the PreP's next invocation.

Pass Input Unmodified. The common case for user input is that it is non-sensitive, and does not require special protection by the PreP. Any existing queue of sensitive events is discarded upon entry to this state. Unless e is a browser *focus* event, State.Prev is set to remain in the Pass Input Unmodified state. The current input event is not considered sensitive, and it is provided as an output when the PreP exits. The legacy OS then interprets this input event just as it does today.

Focused. A browser *focus* event contains the name of the field that has just received focus (e.Field). The PreP saves the cryptographic hash of the current PoPr and its SSL certificate (which was validated during PreP Initialization) as the *PoPrID* . It is necessary to track the PoPrID to ensure that the PoPr is not maliciously exchanged while the user is typing sensitive input. When invoked with a keystroke, a PreP in the Focused state checks whether the keystroke is the `@` character. If so, the PreP transitions to the First `@` state. Otherwise, the PreP transitions back to the Pass Input Unmodified state. The keystroke is output to the legacy OS for processing. Note that the `@` keystroke is not secret; it serves only to signify that the user may be about to enter something she considers sensitive.

[2] *Sealed* means that the state is encrypted and integrity-protected (by computing a MAC) for subsequent decryption and integrity-verification. This use of *sealed* is consistent with the TPM's *sealed storage* facility, which we describe in Chapter 2.

First `@`. We have defined the secure attention sequence for Bumpy to be `@@` in the input stream immediately following a browser *focus* event. This state serves to keep track of the `@` characters that the user enters. It is possible that the user is in the process of initiating sensitive input. When invoked with a keystroke that is the `@` character, the system transitions to the Second `@` state. Otherwise, the system transitions back to the Pass Input Unmodified state. The keystroke is output to the legacy OS.

Second `@`. When in the Second `@` state, the user has successfully indicated that she is about to provide some sensitive input using Bumpy. From this point forward, the only way for the user to terminate the process of entering sensitive input is to perform an action that will cause a *blur* event in the current input field. A *blur* event is an event in the graphical user interface that indicates that a particular component is becoming inactive, generally because the focus is now elsewhere. Relevant actions include clicking the mouse or pressing Tab, Shift+Tab, Alt+Tab, or Enter. Note that we explicitly *do not* listen for *blur* events from the web browser, as a malicious browser would be able to terminate secure input prematurely. If the user's input does not represent a *blur* event, then the system transitions to the Enqueue Input state.

Enqueue Input. For each PreP invocation in the Enqueue Input state, the decrypted keystroke is filtered before being appended to State.Queue. The filter identifies and drops illegal password characters that would not cause a *blur* event (e.g., meta-characters used for editing, such as arrows, Backspace, and Delete). We discuss editable sensitive input in Section 4.8. This process continues until the user causes a *blur* event, e.g., presses Tab, Shift+Tab, Alt+Tab, Enter, or clicks the mouse. A decoy input event is released to the legacy OS: an asterisk. From the perspective of the legacy OS, when the user types her sensitive input, she appears to be typing asterisks. This has the convenient property of mimicking the usual functionality of input to password fields even if the current field is a normal text field: all keystrokes appear as asterisks. Note that these asterisks will eventually be

discarded, as the PoPr provides the remote webserver with the true input for the protected fields. When the user causes a *blur* event, the system transitions to the Invoke PoPr state.

Invoke PoPr. When the PreP state machine transitions to the Invoke PoPr state, it means that the user successfully directed the web browser's focus to a particular field, entered `@@` to signal the start of sensitive input, provided some sensitive input, and then caused a *blur* event that signals the end of sensitive input. This input will now be handed to the PoPr for destination-specific processing (Section 4.3). First, however, it is necessary to check that the PoPr provided as an input to the PreP during the current Flicker session is the same PoPr that was provided during the *focus* event that initiated this sensitive input. If the PoPr is changed, malicious code may be trying to fool the user. In this case, the system transitions directly back to Pass Input Unmodified where the queue of sensitive input events is discarded. If the PoPr is confirmed to be the same, it is invoked. The PoPr also runs with Flicker's protections, but it is less trusted than the PreP. Thus, PreP state is sealed and cleared *before* invoking the PoPr with the queue of input events. This is essential, as the PreP state includes the cryptographic secrets involved in communication with the user's physical input device(s). If a PoPr could compromise this information, it might be able to collude with a malicious OS and capture all subsequent input on the user's platform. Once the PoPr has executed, its output is handed to the web browser via the legacy OS for submission to the webserver from which the PoPr originated. If the PoPr considers these input events to be secret, then they should be encrypted such that the legacy OS will be unable to learn their values.

4.2.2 Associating the PreP and Input Device(s)

Bumpy depends on input devices capable of encrypting input events generated by the user, so that the legacy OS can pass the opaque events to the PreP without learning their value. We now describe the process of estab-

lishing the necessary cryptographic keys when input device(s) and a PreP are first associated.

The PreP is invoked with the command to establish a new association with an input device. During this process, the PreP:

1. Generates symmetric encryption and MAC keys for the protection of its own long-term state, K_{lt_enc}, K_{lt_mac}, if they do not already exist.
2. Generates an asymmetric keypair, $K_{input}, K_{input}^{-1}$, for bootstrapping communication with the encrypting input device, and adds the private key to its long-term state. Note that no other software (not even a PoPr) will ever be allowed to access K_{input}^{-1}.

The public key K_{input} is then conveyed to the input device. Given the complexity of equipping input devices with root CA keys to verify certificates, we use a trust-on-first-use model where the input device simply accepts K_{input} and then prevents it from changing unexpectedly. Since encryption-capable input devices like we require do not currently exist, we specify how the input device enters a state where it is willing to accept a public encryption key. The greatest challenge is to prevent key re-establishment from being initiated in the presence of malicious code, whether through a design or implementation failure or a social engineering attack. One promising design may be a physical switch that must be placed into the "Establish Keys" position before key establishment can commence. This way, most users will establish keys once and forget about it. Another alternative is to use a location-limited channel (Chapter 5) [10, 166]. We describe input device key establishment for our USB Interposer implementation in Section 4.6.

In the future, more sophisticated input devices may verify an attestation that the public key came from a known-good PreP. Another promising alternative is that input device manufacturers imprint their devices with a certificate establishing them as approved encrypting input devices, though it can be challenging to establish that a certificate corresponds to a particular physical device [125]. The PreP can then verify the origin of input events as trustworthy, provided that the PreP's list of trusted CAs covers

manufacturers' signing keys.

Irrespective of the method of public-key exchange, symmetric keys for encryption and integrity protection should be established to maximize performance. Additionally, we require the use of sequence numbers so that the PreP can detect if any keystrokes are dropped or reordered by the legacy OS.

4.2.3 PreP State Freshness

The PreP must have the ability to protect its own state when the legacy OS has control (i.e., across Flicker sessions). The secrecy and integrity of the PreP's state is ensured by encrypting it under a symmetric master key kept in PCR-protected non-volatile RAM on the TPM chip. It is PCR-protected under the measurement of the PreP itself, so that no other code can ever unseal the master key. However, protecting the freshness of the PCR-protected state is more challenging. The risk is a state roll-back or replay attack where, e.g., an attacker may try to keep the PreP in the Pass Input Unmodified (Figure 4.3) state by perpetually providing the same ciphertext of the PreP's state as an input to the PreP. The standard solution to this problem is to employ a secure counter or other versioning scheme that can track the latest version of the PreP's state. While the TPM does implement a monotonic counter [172], its specification dictates that the counter need only support incrementing every five seconds. This is clearly insufficient to keep up with per-keystroke events. Our solution is to leverage the sequence numbers associated with input events coming from the input devices (Section 4.6.2.3).

4.3 Input Post-Processing and Attestation

We now detail the actions taken by the PoPr to Post-Process sensitive input events enqueued by the PreP (Step 6 in Figure 4.2). Then, we describe the attestation process employed to convince the remote webserver (which is the PoPr provider) that it is receiving inputs protected by Bumpy.

4.3.1 Post-Processing Sensitive Input

The final destination for sensitive input protected by Bumpy is the webserver from which the current web page originated. In Section 4.2.1, we describe how the PoPr is invoked by the PreP when the user has completed entering sensitive input for a particular field. The PoPr is provided by the webserver hosting the current web page in the user's browser. The sensitive input is tagged with the name of the field that had focus when the input was entered, and this {*inputString, tag*} pair is what the PoPr receives. The PoPr can perform arbitrary, site-specific transformations on the sensitive input before handing the (potentially opaque) result to the web browser for transmission to the remote webserver when the page is submitted.

4.3.1.1 Example Forms of Post-Processing

We consider two example forms of post-processing that we believe to be widely useful on the web today, though there may be many others. The first is encryption of user input such that only the webserver can process the raw input, and the second is destination-specific hashing (with the PwdHash algorithm [138]) so that, e.g., passwords cannot be reused at multiple websites.

End-to-End Encryption. Encrypting sensitive inputs for the webserver completely removes the *web browser and OS* from the TCB of the input path for the field accepting the sensitive input, though it requires the *webserver* to be Bumpy-aware. This capability is achieved on the user's system by embedding a webserver-generated public encryption key in the PoPr. The PreP will automatically check that the PoPr (and hence its encryption key) is certified by the webserver from which it originated.

Note that it may seem tempting to have the input device encrypt the user's input all the way to the remote webserver, removing the need for the PreP or PoPr. We prefer the flexibility afforded by the Flicker architecture to process input in a PoPr on the user's system in whatever way is appropriate for a given application or remote system. It is not the duty of the input

device manufacturer to foresee all of these possible applications. Indeed, incorporating too much programmability into the input device itself is sure to make it a promising target for attack. Rather, the input device is tasked solely with getting keystrokes securely to the PreP, and websites concerned about the size of the input TCB can supply their own minimized PoPr.

Destination-Specific Hashing with PwdHash. A second form of post-processing – contained entirely within the user's system – is to perform a site-specific transformation of certain fields before they are released to the web browser for transmission to the webserver. For example, usernames or passwords can be hashed along with the webserver's domain name, thereby providing the user with additional protection when she employs the same username or password at multiple websites. The domain name is obtained from the webserver's SSL certificate, which is verified by the PreP before the PoPr begins executing. In Section 4.6, we describe our implementation of the PwdHash [52, 51, 138] algorithm within Bumpy.

4.3.1.2 Activating a PoPr

It is possible that malicious browser or OS code will intentionally load the wrong PoPr for the current site. If the PoPr is well-behaved (i.e., provided by a reputable webserver), then it is unlikely to expose the user's sensitive input. However, attackers may intentionally use a PoPr from a compromised server with a valid SSL certificate. In this case, our defense is the Trusted Monitor, as it will display the domain name and favicon[3] of the website that has certified the current PoPr. It is the user's responsibility to see that the information on the Trusted Monitor corresponds to the web page that she is browsing. We explain the detailed operation of the Trusted Monitor, including the users' responsibilities, and how secure communication between the PreP and the Trusted Monitor is established, in Section 4.4.

[3] A *favicon* is an image associated with a particular website or web page, installed by the web designer. It is commonly displayed alongside the address bar and alongside a tab's title for browsers that support tabs.

4.3.2 Attestation and Verifying Input Protections

Flicker enables the computer using Bumpy to attest to the PreP and PoPr that have run most recently. This attestation can be verified by a remote entity to ascertain whether the user's input received the intended protection. Though there are no technical limitations governing which device (or devices) perform the verification, we proceed from the perspective of the remote webserver as the verifier. Institutions such as banks employ professional administrators who are better suited than the average consumer to make trust decisions in response to which PreP and PoPr are in operation on the user's computer. For certain types of transactions (e.g., online banking), the webserver may be willing to expose more services to a user whose computer can provide this assurance that the user's input is being protected. However, the user still must behave responsibly.

If verification by the remote webserver succeeds, then the requested web page can be served normally. However, if verification fails, then the software state of the user's system cannot be trusted, and the webserver should prevent access to sensitive services. One option is to serve a web page with an explanation of the error, though there is no guarantee that the malicious (or unknown) software will display the error. We discuss an extension to create a trusted path between the Trusted Monitor and webserver for conveyance of such error notifications in Section 4.8.

4.3.2.1 Establishing Platform Identity

TPM-based remote attestation is used to convince the webserver that the user's input is protected with Bumpy. However, the remote webserver must first have a notion of the identity of the user's computer system. We use an Attestation Identity Key (AIK) generated by the TPM in the user's computer. Chapter 2 discusses known techniques for certifying an AIK, any of which can be applied to Bumpy. Here, Bumpy benefits from the property of the Flicker system (Chapter 3) that causes attestations to cover *only* the PreP and PoPr code that was executed, and no other software at all.

4.3.2.2 The Attestation Protocol

Here, we describe the protocol between a user's system with Bumpy and a Bumpy-aware webserver when they connect for the first time. As an example, we consider a user who is trying to login to a webserver's SSL-protected login page. The user's browser sends a normal HTTPS request for the login page.

In response, the webserver participates in an SSL connection and delivers the login page. Embedded within the page (e.g., in a hidden input element) are several Bumpy-specific pieces of information, which must be signed by the webserver's private SSL key:

- *nonce* – A nonce to provide replay protection for the ensuing attestation.
- *hash* – The cryptographic hash of the PoPr. The PoPr itself can be obtained with another HTTP request and verified to match this hash.
- *favicon* – The favicon corresponding to the webserver.
- $Cert_{ws\_enc}$ – A public encryption key signed by the webserver's private SSL key.

A well-behaved browser then passes the newly received PoPr, embedded information, and the webserver's public SSL certificate to the untrusted code module that manages the invocation of Flicker sessions with the PreP. During subsequent Flicker sessions, these data are provided as input to the PreP. The PreP verifies the webserver's SSL certificate using its own list of trusted CAs, and verifies that the other input parameters are properly signed.

If all verifications succeed, an output message is prepared for the webserver. This message requires the generation of an asymmetric keypair within the PoPr that will serve to authenticate future encrypted strings of completed input as having originated within this PoPr. This key is generated and its private component is protected in accordance with the Flicker external communication protocol (Chapter 3). Only this PoPr will ever be able to access the private key.

When key generation completes, the newly generated public signing key (K_{PoPr_sig}) is extended into a TPM Platform Configuration Register (PCR) and output from the PoPr. Untrusted code running on the legacy OS then passes this key back to the web browser, along with the user's system's public identity (e.g., an Attestation Identity Key, or the set of Endorsement Key, Platform, and Conformance Credentials) and a TPM attestation covering the relevant PCRs. These tasks can be left to untrusted code because the properties of the PCRs in the TPM chip prevent untrusted code from undetectably tampering with their values.

In steady-state, the PoPr will encrypt user input using the public key in the webserver-provided $Cert_{ws\_enc}$, and sign it with $K^{-1}_{PoPr\_sig}$ to authenticate that it came from the PoPr running on the user's computer. There is no need to perform an attestation during future communication between this PoPr and webserver.

4.3.2.3 Processing Attestation Results

Remote entities need to have knowledge that a set of attested measurements represents a PreP and PoPr that keep the user's input and PreP state safe (encrypted when untrusted code runs, which may include Flicker sessions with other, distrusted PALs). Prominent institutions (e.g., banks) may develop and provide their own PoPrs for protecting user input to their websites. In these cases, the institution's webserver can easily be configured with the expected PoPr measurements, since it provided the PoPr in the first place. If one PoPr proves to be sufficient for a wide variety of websites, then its measurement may become a standard which can be widely deployed.

The webserver must also have knowledge of existing PrePs in order to make a trust decision based on the attestation result. We expect the number of PrePs to be reasonably small in practice, as most input devices adhere to a well-known (and simple) protocol.

4.4 The Trusted Monitor

Bumpy's input protections by themselves are of limited value unless the user can ascertain whether the protections are active when she enters sensitive data. The primary usability criticism [32] of PwdHash [138] is that it provides insufficient feedback to the user as to the state of input protections. Thus, it is of utmost importance that the user is aware of the transition between protected and unprotected input. With Bumpy, the Trusted Monitor serves as a trusted output device that provides feedback to the user concerning the state of input protections on her computer.

4.4.1 Feedback for the User

When input protections are active, the Trusted Monitor displays the final destination (e.g., website) whose PoPr will receive her next sensitive input. We represent this using the domain name and favicon of the currently active PoPr, as reported by the PreP. When input protections are disabled, the Trusted Monitor displays a warning that input is unprotected and that users should use `@@` to initiate sensitive input. Figure 4.4 shows screen shots from our implementation. In addition to changing the information on its display, the Trusted Monitor uses distinctive beeps to signal when protections become enabled or disabled.

The Trusted Monitor works in concert with the properties of the PreP's Second `@` and Enqueue Input states (Figure 4.3): when in these states, the PoPr is *locked in* and cannot change until after the sensitive input to a single field is processed by this PoPr (in the Invoke PoPr state). As such, the PoPr represented by the domain name and favicon that are displayed by the Trusted Monitor will remain the active PoPr until input to the current field is complete. Thus, there is no need for the user to worry about a malicious PoPr change in the middle of a string of sensitive input. However, the user must be diligent between fields. She must ensure that the Trusted Monitor responds to each unique `@@` sequence that she types (i.e., that the Trusted Monitor beeps and shows that protection is enabled) before proceeding to input her sensitive data. This is because the untrusted OS may affect the

delivery of encrypted keystrokes to the PreP and PreP messages to the Trusted Monitor.

The risk is that malicious code may try to confuse the user such that she misinterprets the Trusted Monitor's display for one input field as indicating that her input is secure for additional input fields. One such attack works as follows. Malcode allows keystrokes and Trusted Monitor updates to proceed normally until the user begins typing sensitive input for one input field on a web page. The Trusted Monitor beeps and updates its display to indicate that protections are active. At this point, the malcode begins to suppress Trusted Monitor updates, but the Trusted Monitor cannot immediately distinguish between suppressed updates and a distracted user who has turned away from her computer. A user who finishes typing this secret and then transitions to another input field and proceeds to enter another secret — even after entering `@@` and glancing at the Trusted Monitor, but without waiting for confirmation of the receipt of the new `@@` by the PreP— renders the second secret vulnerable to disclosure. To expose this secret, the malicious OS plays the user's encrypted inputs to the PreP after the user is finished typing the second secret, but provides a malicious PoPr to the PreP when transitioning to the Focused and Invoke PoPr states for the second input. That is, the user provided the second secret presuming it was protected in the same way as the first, but since she did not confirm that the second `@@` was received by the PreP before she typed the second secret, it is vulnerable to disclosure to a malicious PoPr.

To help users avoid such pitfalls, it may be desirable for the Trusted Monitor to emit an audible "tick" per sensitive keystroke received by the PreP, in addition to the preceding beep when the `@@` is received. This way, the absence of ticks might be another warning to the user.

4.4.2 Protocol Details

To facilitate the exchange of information regarding the active PoPr, a cryptographic association is needed between the PreP and the Trusted Monitor. To establish this association, the Trusted Monitor engages in a one-time

initialization protocol with the PreP, whereby cryptographic keys are established for secure (authentic) communication between the PreP and the Trusted Monitor. The protocol is quite similar to that used between the PreP and input device(s) in Section 4.2.2.

The initialization process for PreP-to-Trusted Monitor communication is an infrequent event (i.e., only when the user gets a new Trusted Monitor or input device). Thus, a trust-on-first-use approach is reasonable, where the Trusted Monitor simply accepts the public key claimed for the PreP. Any of a range of more secure (but more manual or more infrastructure-dependent) approaches can be employed, including ones that allow the Trusted Monitor to validate an attestation from the TPM on the user's computer as to the correct operation of the PreP and to the value of its public key (a capability offered by Flicker). The PreP can save its private key in PCR-protected storage on the user's computer, and so will be available only to this PreP in the future (as in Section 4.2).

The Trusted Monitor need not be a very complex device. Its responsibilities are to receive notifications from the user's computer via wired or wireless communication, and to authenticate and display those notifications. While our implementation employs a smartphone for a Trusted Monitor (Section 4.6), this is far more capable than is necessary (and more capable than we would recommend).

With a smartphone serving as the Trusted Monitor, there is no reason why the user's Trusted Monitor cannot perform the full gamut of verification tasks we have described as being in the webserver's purview. In fact, technically savvy and privacy-conscious users may prefer this model of operation, and it becomes significantly easier to adopt if a small number of PrePs and PoPrs become standardized across many websites. These users can learn that their input is being handled by precisely the PreP and PoPr that they have configured for their system, and that opaque third-party code is never invoked with their input.

4.5 Security Analysis

We discuss Bumpy's TCB, the implications of a compromised web browser, phishing attacks, and usability.

4.5.1 Trusted Computing Base

One of the primary strengths of Bumpy is the reduction in the TCB to which input is exposed on the user's computer. Always in the TCB are the encrypting input device and the PreP that decrypts and processes the encrypted input events on the user's computer. The PoPr associated with each website is also in the TCB for the user's interaction with that website, but the PreP isolates each PoPr from both the PreP's sensitive state and the OS (thereby preventing a malicious PoPr from harming a well-behaved OS). The encrypting input device is a dedicated, special-purpose hardware device, and the PreP is a dedicated, special-purpose software module that executes with Flicker's isolation. A compromise of either of these components is fatal for Bumpy, but their small size dramatically reduces their attack surface with respect to alternatives available today, and may make them amenable to formal verification. The PoPr may be specific to the destination website, and may be considered a local extension of the remote server. It does not make sense to send protected input to a remote server that the user is unwilling to trust. Additionally, the PoPr's functionality is well-defined, leading to small code size.

Also in the TCB is the Trusted Monitor that displays authenticated status updates from the PreP, i.e., the domain name and favicon for the active PoPr. The Trusted Monitor never handles the user's sensitive input, so compromising it alone is insufficient to obtain the user's input. However, if the Trusted Monitor indicates that all is well when in fact it is not, then a phishing attack may be possible (Section 4.5.3).

4.5.2 Compromised Browser

If the user's browser or OS is compromised, then malicious code can invoke the PreP with input of its choosing. Bumpy can still keep the user's sensitive input safe provided that she adheres to the convention of starting sensitive input with `@@` and pays attention to the security indicator on her Trusted Monitor.

The cryptographic tunnel between the input device and PreP prevents malicious code from directly reading any keystrokes, and prevents the malicious code from injecting spurious keystrokes. Thus, a compromised browser's options are restricted to providing spurious inputs to the PreP, including SSL certificates, PoPrs, and browser *focus* events. None of these are sufficient to violate the security properties of Bumpy, but they can put the user's diligence in referring to the Trusted Monitor to the test.

Malicious SSL Certificates. The PreP is equipped with a list of trusted certificate authorities (CAs). Any SSL certificate that cannot be verified is rejected, causing sensitive keystrokes to be dropped. Thus, an attacker's best option is to compromise an existing site's SSL certificate (thereby reducing the incentive to attack the user's computer), or to employ a phishing attack by registering a similar domain name to that which the user expects (e.g., hotmai1.com, instead of hotmail.com) and using an identical favicon.

Malicious PoPr. The PreP will not accept a PoPr unless it can be verified with the current SSL certificate, thereby reducing this attack to an attack on the SSL certificate (as described in the previous paragraph) or webserver.

Malicious Browser Focus Events. A malicious browser may generate spurious or modified *focus* events in an attempt to confuse the PreP with respect to which field is currently active. However, regardless of which field is active, the user controls whether the current input events are considered sensitive. When they are sensitive, input to a field is always encrypted (initially by the input device(s), and subsequently within the PreP) and tagged with the field's name until input to the field completes, at which

point they are released to the PoPr. A malicious *focus* event may only cause ciphertext to be tagged with the wrong field name. Provided that the PoPr is written to protect *all* processed input events when they are released to the legacy OS for transmission to the webserver, malicious focus events are only a threat to availability. However, we already consider an adversary which controls the OS on the user's computer, and is thus already in total control of availability.

Omitted Browser Focus Events. A malicious browser may refuse to deliver any focus events. In this case, as the state machine in Figure 4.3 shows, the user's `@@` will not trigger special protections for user input. Thus, the Trusted Monitor will not beep or update its display to show that input protections are active. Currently, it is the user's responsibility to detect that her `@@` did not trigger input protections. However, it is a straightforward extension to detect `@@` from within the Pass Input Unmodified state in Figure 4.3 and actively warn the user that her browser may be compromised.

4.5.3 Phishing

If a user is fooled by a phishing attack (e.g., she confuses similar-looking domains), she may be using Bumpy's protections to enter her sensitive data directly into a phishing website. Defeating phishing attacks is not our focus here, though Bumpy should be compatible with a wide range of phishing defenses [79]. As a simple measure, Bumpy provides an indicator on the Trusted Monitor that includes the domain name and favicon of the current website. Recall from Section 4.3.2.2 that the favicon must be signed by the webserver's private SSL key. It is the subject of future work to determine whether users are more likely to fall victim to phishing attacks that use an exact duplicate of the phished site's favicon.

Though we have not solved some of the intrinsic problems with certificate authorities and SSL (e.g., users are in the habit of always accepting certificates), the PreP can enforce policies such as: only PoPrs from whitelisted webservers are eligible to receive a user's input; PoPrs from blacklisted

webservers can never receive a user's input; and self-signed certificates are never acceptable. These policies are enforceable in the PreP, and require the user to have a Trusted Monitor only to provide feedback to improve usability.

With a PoPr implementing PwdHash, only the hashed password is returned to the web browser. If a user is fooled into entering her password into a phishing site with a different domain name, the phishing site captures only a hash of the user's password, and must successfully perform an offline dictionary attack before any useful information is obtained about the user's password at other sites. Additionally, in the case where a user ignores the indicator but has established the habit of starting her password with `@@`, hashing of the user's password can restrict the impact of the user's being phished on one website to that website alone. With a compromised OS, malware on the user's system can observe the hashed password when it is released to the web browser, but this password is only valid at a single website.

4.5.4 Usability

Confusion. If users do not understand the Bumpy system, or their mental model of the system is inaccurate, then they may be fooled by a malicious web page. For example, a prompt such as the following may trick the user into believing that there is no need to prefix her password with `@@` on the current web page:

`Input your password: @@ _____`

The user may also become confused if she makes a typographical error entering `@@`, and tries to use backspace to correct it. Bumpy will not offer protections in this case, until the user changes to another input field and then comes back to the current field (i.e., causes a *blur* event and then a new *focus* event). The Trusted Monitor does indicate that protections are disabled, but it may not be obvious to the user why this is the case. We discuss editable sensitive input in Section 4.8.2.

Only a formal user study can ascertain the level of risk associated with

this kind of attack, which we plan to pursue in future work.

Extra Mouse Clicks. When a user clicks in an input field, a *focus* event is generated for the field and conveyed to the PreP. The user's next mouse click is interpreted by the PreP as a *blur* event for the current input field, disabling input protection. An attack may be possible if the user clicks the mouse in an input field after already typing part of her input into the field. This click could be interpreted as a *blur* event, and cause the rest of the user's keystrokes to be sent unencrypted. This may arise when, e.g., the user forgot her credit card number after entering the first few digits from memory, and needs to go lookup the remainder. The Trusted Monitor will beep and update its display to indicate that input protections are disabled when this *blur* event happens, but this may be a source of user confusion.

4.6 Implementation

Our implementation of Bumpy supports verification by the remote webserver with a smartphone as Trusted Monitor to provide feedback to the user. We implement two PoPrs: one encrypts sensitive input as-is for transmission to a Bumpy-aware webserver, and the other hashes passwords with the PwdHash algorithm [138] for transmission to an unmodified webserver.

We have been unable to find any commercially available keyboards or mice that enable programmable encrypted communication. However, myriad wireless keyboards do implement encrypted communication with their host adapter (e.g., encrypted Bluetooth packets are decrypted in the Bluetooth adapter's firmware, and not in software). Thus, the problem is not technical, but rather a reflection of the market's condition. Indeed, Microsoft's NGSCB was originally architected to depend on USB keyboards capable of encryption [42, 126]. In our system, we have developed a USB Interposer using a low-power system-on-a-chip. Our USB Interposer supports a USB keyboard and mouse and manages encryption for use with Bumpy.

We have implemented Bumpy using an HP dc5750 with an AMD Athlon64

X2 at 2.2 GHz and a Broadcom v1.2 TPM as the user's computer, with a USB-powered BeagleBoard [15] containing a 600 MHz ARM CPU running embedded Linux serving as the USB Interposer. We use a Nokia E51 smartphone running Symbian OS v9.2 as the Trusted Monitor. Our USB Interposer supports encryption of all keyboard events, and mouse click events. Mouse movement events (i.e., X and Y delta information) are not encrypted, since only mouse clicks trigger *blur* events in the web browser GUI.

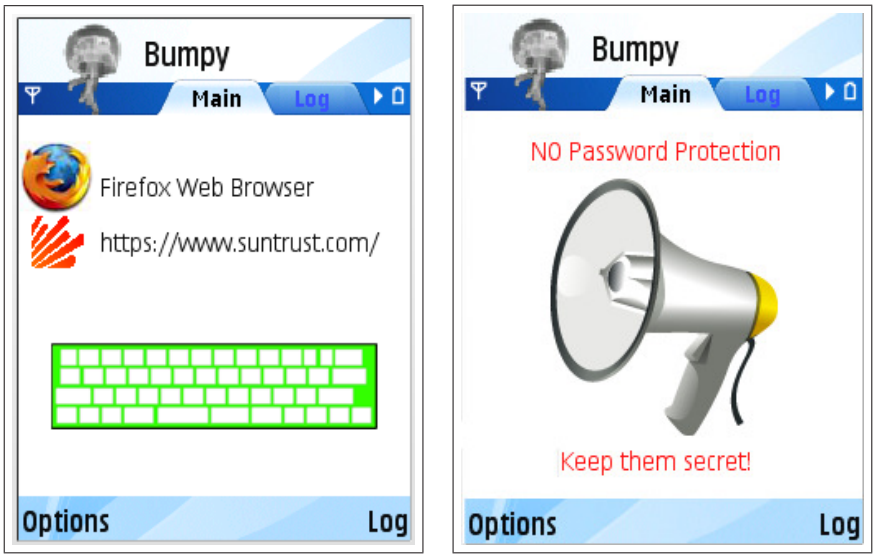
4.6.1 Bumpy Components

Our implementation includes the PreP and two PoPrs that run with Flicker's protections on the user's computer, the USB Interposer (BeagleBoard), the Trusted Monitor running on a smartphone, and an untrusted web browser extension and Perl script. We begin by describing the components that are in Bumpy's TCB, and then treat the additional untrusted components that are required for availability (which we are forced to surrender since we consider the OS as untrusted).

PreP and PoPrs. We implemented the PreP as a Piece of Application Logic that runs with the protection of the Flicker system and (1) receives encrypted keystroke events from the encrypting input device (i.e., the USB Interposer), (2) invokes one of our PoPrs to process the encrypted keystrokes for the webserver, either by re-encrypting them or performing the PwdHash [138] operation on passwords, and (3) sends encrypted messages to the Trusted Monitor that provide the favicon and domain of the active web page and PoPr. In our implementation, the PreP and both PoPrs are all part of the same PAL that runs using Flicker. An input parameter controls which PoPr is active.

USB Interposer. Our USB Interposer is built using a BeagleBoard featuring an OMAP3530 processor implementing the ARM Cortex-A8 instruction set [15], and a Prolific PL-25A1 USB-to-USB bridge [132]. We currently run embedded Debian Linux to benefit from the Linux kernel's mature support for both USB host and client operation. While this adds considerable

code-size to our TCB, the interposer executes in relative isolation with a very specific purpose. We implement a small Linux application that receives all keyboard and mouse events (using the kernel's `evdev` interface), and encrypts all keyboard and mouse click events, letting mouse movement information pass in the clear. We describe the cryptographic protocol details in Section 4.6.2.



(a) Protection enabled visiting SunTrust bank.

(b) Protection disabled.

Figure 4.4: Screen shots of the Trusted Monitor.

Trusted Monitor. We implemented a Symbian C++ application that runs on the Nokia E51 smartphone and serves as the Trusted Monitor. The Trusted Monitor updates its display in response to authenticated messages from the PreP, as described in Section 4.4. Figure 4.4 shows screen shots of the Trusted Monitor in action. When a session is active between the Trusted Monitor and PreP, the Trusted Monitor displays the domain name and favicon of the active web page's PoPr. It also displays a green keyboard (Figure 4.4(a)) as a unified indicator that protections are enabled. When input protections are disabled, it displays a warning message that

input is unprotected and that `@@` should be used for sensitive input (Figure 4.4(b)). The Trusted Monitor uses distinctive beeps whenever input protections transition between enabled and disabled.

Note that after the initial configuration of the Trusted Monitor and PreP (Section 4.6.2), no further configuration is necessary during subsequent input sessions. The long-term symmetric keys encrypted under the master key that is kept in PCR-protected TPM NV-RAM will only be accessible to the correct PreP. Thus, only the PreP will be able to send authentic messages to the Trusted Monitor.

Untrusted Components. We developed an untrusted Firefox *Browser Extension* that communicates a web page's SSL certificate and embedded PoPr, and all *focus* events to the PreP. An untrusted Perl script facilitates communication between all components, manages the invocation of Flicker sessions, injects decrypted keystrokes into the OS using the Linux kernel's Uinput driver, and provides TPM *Quotes* in response to attestation requests. Note that the Flicker architecture provides the property that the code requesting the attestation from the TPM chip need not be trusted. To convey encrypted data from the PreP to the USB Interposer, Trusted Monitor, or browser extension, the PreP must exit and release the ciphertext to the Perl script.

4.6.2 Secure Communication with the PreP

Both the USB Interposer and the Trusted Monitor require the ability to exchange secret, integrity-protected messages with the PreP. We implement the Flicker external communication protocol for both, with a trust-on-first-use model for accepting the respective public keys created in the PreP. Neither the USB Interposer nor the Trusted Monitor is pre-configured with knowledge of the identity of the TPM in the user's computer or the identity of the PreP installed on the user's computer.

We program a dedicated button on the USB Interposer to bootstrap association with a PreP, whereas the Trusted Monitor exposes a menu option

to the user to connect to her computer to perform the initial configuration. The USB Interposer communicates with the user's computer via USB, and we use the AT&T 3G cellular network or WiFi to connect the Trusted Monitor to the user's computer using a standard TCP/IP connection. An untrusted Perl script running on the user's computer handles reception of these messages and invokes Flicker sessions with the PreP so that the messages can be processed.

Both the USB Interposer and Trusted Monitor send a request to initiate an association with the PreP, passing in the command to bootstrap Flicker's external communication protocol (Chapter 3), as well as a nonce for the subsequent attestation. The PreP then uses TPM-provided randomness to generate a 1024-bit RSA keypair. In accordance with Flicker's external communication protocol, the PreP extends PCR 17 with the measurement of its newly generated public key. The public key is then output from the PreP to be sent to the Trusted Monitor, and PCR 17 is *capped* (extended with a random value) to indicate the end of the Flicker session. At this point, PCR 17 on the user's computer contains an immutable record of the PreP executed and public key generated during execution.

4.6.2.1 PreP Authentication

Our use of a trust-on-first-use model to accept the PreP's public key dictates that no further verification of the exchanged keys is necessary. However, rigorous security goals may require the USB Interposer or Trusted Monitor to verify that the user's computer is running an approved PreP. In our current prototype, the USB Interposer and Trusted Monitor request a TPM attestation from the user's computer to ascertain the machine's public *Attestation Identity Key* (AIK) that it uses to sign attestations (TPM *Quotes* [172]), and the measurement (SHA-1 hash) of the PreP that will process input events. On subsequent connections, any change in the AIK or PreP measurement is an error. This way, it is readily extensible to allow application vendors to distribute signed lists of expected measurements, to leverage a PKI, or to a community-driven system similar in spirit to that of Wendlandt

et al. (*Perspectives* [178]), and thus enable the USB Interposer and Trusted Monitor to validate the identity of the PreP themselves.

The USB Interposer and Trusted Monitor include a nonce with their initial connection requests, and expect a response that includes a TPM Quote over the nonce and PCR 17. The measurements extended into PCR 17[4] are expected to be the measurement of the PreP itself, the command to bootstrap external communication (`ExtCommCmd`), and the measurement of the public RSA key produced by the PreP. H denotes the SHA-1 hash function: `PCR17` $\leftarrow$ `H(H(H(`0^{160}`||H(PreP)) ||H(ExtCommCmd))||H(PubKey))`.

The USB Interposer and Trusted Monitor perform the same hash operations themselves using the measurement of the PreP, value of `ExtCommCmd`, and hash of the received public key. They then verify that the resulting hash matches the value of PCR 17 included in the TPM Quote.

4.6.2.2 Symmetric Key Generation for Communication with the PreP

We bootstrap secret and integrity-protected communication between the PreP and the USB Interposer or Trusted Monitor using the PreP's relevant public key to establish a shared master key K_{M1}. Separate symmetric encryption and MAC keys are derived for each direction of communication. We use AES with 128-bit keys in cipher-block chaining mode (AES-CBC) and HMAC-SHA-1 to protect the secrecy and integrity of all subsequent communication between the Trusted Monitor and the PreP. These keys form a part of the long-term state maintained by both endpoints.

$K_{aes1} \leftarrow$ `HMAC-SHA-1(`K_{M1}`,'aes128.1')`^{128}

$K_{hmac1} \leftarrow$ `HMAC-SHA-1(`K_{M1}`,'hmac-sha1.1')`

$K_{aes2} \leftarrow$ `HMAC-SHA-1(`K_{M1}`,'aes128.2')`^{128}

$K_{hmac2} \leftarrow$ `HMAC-SHA-1(`K_{M1}`,'hmac-sha1.2')`

[4]This example is specific to an AMD system. The measurements extended by Intel systems are similar.

4.6.2.3 Long-Term State Protection

The PreP must protect its state from the untrusted legacy OS while Flicker is not active. To facilitate this, the PreP generates a 20-byte master key K_{M2} using TPM-provided randomness. This master key is kept in PCR-protected non-volatile RAM (NV-RAM) on the TPM chip itself. We choose TPM NV-RAM instead of TPM Sealed Storage because of a significant performance advantage. The PCR 17 value required for access to the master key is that which is populated by the execution of the PreP using Flicker:
`PCR17` $\leftarrow$ `SHA-1(`0^{160}`||SHA-1(PreP))`.

Flicker ensures that no code other than the precise PreP that created the master key will be able to access it. Our PreP uses AES-CBC and HMAC-SHA-1 to protect the secrecy and integrity of the PreP's state while the (untrusted) legacy OS runs and stores the ciphertext. The necessary keys are derived as follows:
$K_{aes} \leftarrow$ `HMAC-SHA-1(`K_{M2}`,'aes128')`^{128},
$K_{hmac} \leftarrow$ `HMAC-SHA-1(`K_{M2}`,'hmac-sha1')`.

This is sufficient to detect malicious changes to the saved state and to protect the state's secrecy. However, a counter is still needed to protect the freshness of the state and prevent roll-back or replay attacks. The TPM does include a monotonic counter facility [172], but it is only required to support updating once every five seconds. This is insufficient to keep up with user input. Instead, we leverage the sequence numbers used to order encrypted input events coming from the USB Interposer. The PreP is constructed such that a sequence number error causes the PreP to fall back to a challenge-response protocol with the USB Interposer, where the PreP ensures that it is receiving fresh events from the USB Interposer and reinitializes its sequence numbers. Any sensitive input events that have been enqueued when a sequence number error takes place are discarded. Note that this should only happen when the system is under attack.

The USB Interposer and Trusted Monitor run on devices with ample non-volatile storage available.

4.6.3 The Life of a Keystroke

Here, we detail the path taken by keystrokes for a single sensitive web form field. It may be useful to refer back to Figures 4.2 and 4.3. At this point, symmetric cryptographic keys are established for bidirectional, secret, authenticated PreP-USB Interposer and PreP-Trusted Monitor communication. We now detail the process that handles keystroke events as the user provides input to a web page.

The user begins by directing focus to the relevant field, e.g., via a click of the mouse. On a well-behaved system, our browser extension initiates a Flicker session with the PreP, providing the name of the field, and the webserver's SSL certificate, PoPr (which includes the encryption key certificate $Cert_{ws\_enc}$), nonce, and favicon as arguments. The PreP verifies the SSL certificate using its CA list and verifies that the PoPr, nonce, and favicon are signed by the same SSL certificate. The user then types `@@` to indicate that the following input should be regarded as sensitive. The user's keystrokes travel from the keyboard to the USB Interposer, where they are encrypted for the PreP, and transmitted to the Perl script on the user's computer (Steps 1–3 in Figure 4.2). The script then initiates other Flicker sessions with the PreP, this time providing the encrypted keystrokes as input (Step 4 in Figure 4.2). The PreP decrypts these keystrokes and recognizes `@@` (Figure 4.3) as the sequence to indicate the start of sensitive input. The PreP outputs the `@` characters in plaintext and prepares a message for the Trusted Monitor to indicate the domain name and favicon of the current website and PoPr. The Trusted Monitor receives this message, beeps, and updates its display with the domain name and favicon.

Subsequent keystrokes are added to a buffer maintained as part of the PreP's long-term state. Dummy keystrokes (asterisks) are output for delivery to the legacy operating system (Step 5 in Figure 4.2) using the Uinput facility of the Linux kernel (which is also used when cleartext mouse and keyboard input events need to be injected). This enables the browser to maintain the same operational semantics and avoid unnecessary user confusion (e.g., by fewer asterisks appearing than characters that she has typed).

In the common case (after the long-term cryptographic keys are established), TPM-related overhead for one keystroke is limited to the TPM extend operations to initiate the Flicker session, and a 20-byte read from NV-RAM to obtain the master key protecting the sealed state. All other cryptographic operations are symmetric and performed by the main CPU. Section 4.7 offers a performance analysis.

When the user finishes entering sensitive input into a particular field, she switches the focus to another field. The PreP catches the relevant input event (a *Blur* in Figure 4.3) on the input stream, and prepares the sensitive input for handoff to the PoPr (Step 6 in Figure 4.2). We have implemented two PoPrs: encryption directly to the webserver, and PwdHash [138]. The PreP will then receive a *focus* event from the browser, indicating that focus has moved to another field. Note that form submission is a non-sensitive input event, so no special handling is required.

Encryption for Webserver. A widely useful PoPr encrypts the sensitive input for the remote webserver exactly as entered by the user (Steps 6–8 in Figure 4.2). This is accomplished using a public encryption key that is certified by the webserver's private SSL key. We use RSA encryption with PKCS#1v15 padding [83] to encrypt symmetric AES-CBC and HMAC-SHA-1 keys, which are used to encrypt and MAC the actual input with its corresponding field tags. The public encryption key is embedded in the PoPr.

Post-Processing as PwdHash. Another useful PoPr performs a site-specific transformation of data before submission to the webserver. We have implemented the PwdHash [138] algorithm in our PoPr. When this PoPr is active, the remote webserver need not be aware that Bumpy is in use, since the hashed password is output to the web browser as if it were the user's typed input. The PoPr manages the transformation from the user's sensitive password to a site-specific hash of the password, based on the domain name of the remote webserver.

4.6.4 The Webserver's Perspective

We now describe the process of acquiring sensitive input from the perspective of a Bumpy-enabled webserver. Prior to handling any requests, the webserver generates an asymmetric encryption keypair and signs the public key using its private SSL key (using calls to OpenSSL), resulting in $Cert_{ws\_enc}$. $Cert_{ws\_enc}$ can be used for multiple clients.

Our implementation consists of a Perl CGI script. When a request arrives at the webserver for a page that accepts user input, our CGI script is invoked to bundle $Cert_{ws\_enc}$ with a freshly generated nonce (for the upcoming attestation from the user's computer) and the hash and URL of the binary image of our direct-encryption PoPr. The ensuing bundle is then embedded into a hidden input field on the resulting web page. The hash and URL of the PoPr prevents wasting bandwidth on transferring the full PoPr unless it is the user's computer's first time employing this PoPr.

When the user submits the resulting page, the webserver expects to receive an attestation from the user's computer covering the PreP, the provided PoPr and nonce, and a public signing key (K_{PoPr_sig}) newly generated by the PoPr on the user's computer. Currently, we employ trust-on-first-use to accept the Attestation Identity Key (AIK) that the user's computer's TPM used to sign the PCR register values. We have manually configured the webserver with the expected measurement of the PreP and PoPrs, as they are part of the same binary in our implementation. If the measurements in the attestation match the expected values, then K_{PoPr_sig} is associated with $K^{-1}_{ws\_enc}$ (and the user's computer's TPM's AIK) to enable decryption and authentication of subsequent strings of sensitive input encrypted by the PoPr.

4.7 Evaluation

We discuss the size of the TCB for our implementation, the performance impact on ordinary typing, webserver overhead, and the impact of network latency on the refresh rate of the Trusted Monitor's display.

PreP and PoPrs		
Func.	**Lang.**	**SLOC**
Main	.c	1044
PwdHash	.c	99
PwdHash	.h	4
Total	.c, .h	1147

Flicker libraries		
Func.	**Lang.**	**SLOC**
Crypto	.c	3980
Crypto	.h	471
TPM	.c	1210
TPM	.h	252
Util	.c	518
Util	.h	251
Util	.S	161
Total	.c, .h, .S	6854

USB Interposer		
Func.	**Lang.**	**SLOC**
Decode, Encrypt & TX	.c	489

Webserver CGI		
Func.	**Lang.**	**SLOC**
Embed & Verify	.pl	167

Trusted Monitor		
Func.	**Lang.**	**SLOC**
Protocol	.cpp	979
Protocol	.h	286
UI	.cpp	539
UI	.h	160
Util	.cpp	50
Util	.h	34
Total	.cpp, .h	2048

Table 4.1: Lines of code for trusted Bumpy components obtained using SLOCCount [179]. The PreP and PoPrs include only the Flicker libraries in their software TCB. The USB Interposer, webserver, and Trusted Monitor also include their respective operating systems.

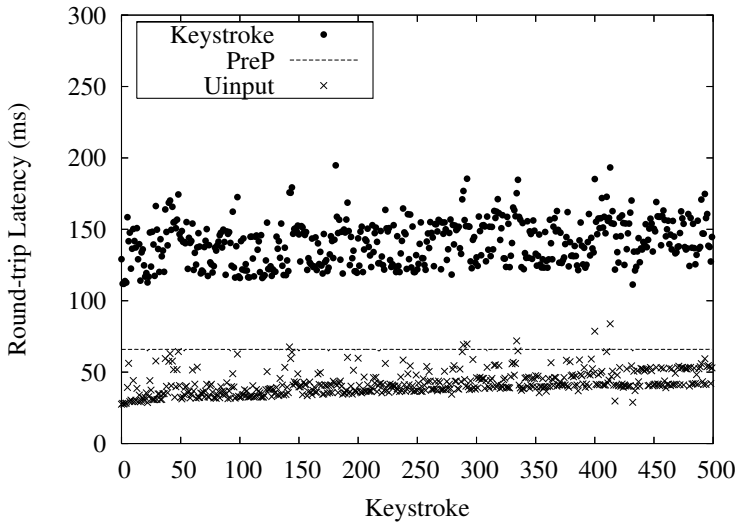


Figure 4.5: Latencies for 500 individual keystrokes. The PreP and Uinput latencies are components of the Keystroke latencies.

Code Size. Bumpy provides strong security properties in part due to its small trusted computing base (TCB). Table 4.1 shows the code size for our PreP and PoPrs, USB Interposer, webserver CGI script, and Trusted Monitor. Note that the TCB for the PreP and PoPrs includes *no additional code beyond the listed Flicker libraries* thanks to the properties of Flicker. Our current USB Interposer runs as a Linux application on a BeagleBoard; however, its only interface is the USB bridge to the user's computer, and its only function is to transmit encrypted keyboard and mouse events. Our Trusted Monitor includes Symbian OS in its TCB, as it runs as a normal smartphone application. We emphasize that the inclusion of Linux in the TCB of our USB Interposer and Symbian OS in the TCB of our Trusted Monitor is an artifact of our prototype implementation, and not a necessary consequence of our architecture.

Typing Overhead with USB Interposer. We measured the round-trip-time between reception of a keypress on the USB Interposer (from the physical keyboard) and reception of an acknowledgement from the user's

computer. This includes the time to encrypt and HMAC the keypress in the USB Interposer, send it to the user's computer via the USB-to-USB bridge, invoke the Flicker session on the user's computer with the PreP (unseal PreP state using the master key kept in PCR-protected TPM Non-Volatile RAM, decrypt and authenticate the newly arrived keypress, reseal PreP state, and release the new keypress to the OS), and send the acknowledgement back over the USB-to-USB bridge. In 500 trials, we experienced overhead of 141±15 ms (Figure 4.5). This is mildly noticeable during very fast typing, similar to an SSH session to a far-away host. It is noteworthy that the overhead consumed by Flicker (i.e., by the PreP) is 66±0.1 ms per keystroke, suggesting that more than half of the latency in our current prototype may be an artifact of the untrusted Perl script in our implementation. Indeed, the contribution of the Uinput driver used to inject keystrokes (42±8 ms) is uncharacteristically large, and grows over time. Writing to the driver from our Perl script presently involves the creation of a child process and a new virtual input device for every keystroke. The virtual input device driver was not designed to scale so far. Our script should be modified to employ the same virtual input device throughout. Ample opportunities remain for optimization, which we plan to pursue in the course of future work in preparation for a user study.

Webserver Overhead with Encryption PoPr. With our encryption PoPr enabled, the webserver must embed $Cert_{ws\_enc}$, a newly generated nonce, the hash of the desired PoPr, the URL at which the client system can obtain the PoPr, and a signature covering the favicon and all of these items into each page that may accept sensitive input. Our webserver is a Dell PowerEdge 2650 with two Intel Xeon 2.4 GHz CPUs running the Debian Linux flavor of Apache 2.2.3. In 25 trials, our CGI script induces a page-load latency of 17.0±0.4 ms, which is primarily composed of reading the cryptographic keys from disk (8.2±0.0 ms) and signing the nonce and metadata (8.6±0.5 ms). When the user submits the completed page, the webserver must verify an attestation from her platform. In 25 trials, our CGI script induces a form-submission latency of less than 2 ms to verify the

signature on the attestation. Note that symmetric keys can be established that reduce the need for the signature-verification operation to a one-time overheads. Though we have not yet implemented this optimization, the only cost is a few tens of bytes of long-term state maintained on the user's computer and the webserver.

Trusted Monitor Network Latency. Our Trusted Monitor uses a TCP connection between the Nokia E51 smartphone and the user's computer. If there is significant network latency, then the Trusted Monitor may not be displaying the correct URL and favicon when the user looks at it. The smartphone can access the Internet using either its 3G/3.5G cellular radio, or using standard 802.11b/g wireless access points. To evaluate the latency impact of using these networks, we performed a simple echo experiment with an established TCP connection, where the E51 sends a series of 4-byte requests and receives 24-byte responses (excluding TCP/IP headers) from the HP workstation. We observed an average round-trip time (RTT) of 102±82 ms using the 802.11 network, and 211±25 ms using AT&T's 3.5G network. In our experience, these latencies are imperceptible to the user as she turns her head to look away from her primary display and towards the Trusted Monitor.

4.8 Discussion

We describe the use of the Trusted Monitor as an input proxy when a USB Interposer is not available, design alternatives and other interesting features that Bumpy might be extended to offer.

4.8.1 Trusted Monitor as Input Proxy

Here we discuss the use of the Trusted Monitor as a proxy to perform encryption of keystroke events in addition to its use as a trustworthy display. We also implement and evaluate this design, which stands as an alternative to the implementation and evaluation presented in the previous sections. This

design is the most analogous to our prior Bump in the Ether work [112], and the most accessible with off-the-shelf devices.

The Trusted Monitor as an encrypting input device is convenient and usable with an external keyboard attached. We use a Bluetooth Apple Wireless Keyboard[5] to provide keystrokes to the Nokia E51 via the E51's Bluetooth Wireless Keyboard application. We have paired this keyboard with the Nokia E51 using Bluetooth's standard encryption and integrity protection.[6] Subsequent discussion treats the wireless keyboard as part of the Trusted Monitor. Our current smartphone prototype does not support protection of mouse input, thus users must change fields on a web form with the keyboard (Tab, Shift+Tab) after entering sensitive data. If the PreP allowed *blur* events from the web browser instead of directly from the input device, then a compromised browser could disable input protections by sending spurious *blur* events.

Typing Overhead with Trusted Monitor as Proxy. This experiment is identical to the experiment measuring the typing overhead of the USB Interposer, except that a Bluetooth wireless keyboard is attached to the Nokia E51, and the Nokia E51 communicates with the user's computer via a standard 802.11b/g WiFi network. We again measured the round-trip-time between reception of a keypress on the Trusted Monitor and reception of an acknowledgement from the PreP. In 50 trials, we experienced overhead of 181±39 ms. This is slower than with the USB Interposer, but it remains quite usable. Unfortunately, over 100 ms of this overhead is induced by the wireless network, and is not amenable to optimization.

Trusted Monitor Attestation Latency. The process of initially associating the PreP on the user's computer with her Trusted Monitor involves the Trusted Monitor receiving an attestation in the form of a TPM Quote from the user's computer. This one-time event only occurs as part of the

[5] `http://www.apple.com/keyboard/`

[6] We are aware of weaknesses in Bluetooth's security primitives (e.g., cracking its PIN [151]), but we placed a higher emphasis on availability of commodity components for our prototype implementation of the smartphone as a proxy for input.

	Avg (ms)	Stdev (ms)
Nonce Gen.	4.3	0.5
Quote RTT	1950.7	115.8
Quote Verif.	7.8	0.5
PCR Verif.	0.1	0.3

Table 4.2: Overhead associated with requesting, generating, receiving, and verifying a TPM Quote.

cryptographic tunnel setup when the Trusted Monitor and PreP establish their first connection, i.e., it is not on the critical path for each keystroke. Table 4.2 shows the relevant overheads. The Trusted Monitor first generates a cryptographic nonce to ensure freshness of the resulting quote. The nonce is then sent to the user's computer, where a TPM Quote is generated. The quote is returned to the Trusted Monitor, where the signature on the quote is verified with the public AIK, and the AIK and PCR values are saved as known-good values. In our current prototype, the establishment of symmetric cryptographic keys means that the attestation information will never be used again. This is a consequence of our trust-on-first-use design; the attestation information may prove valuable in the future if the Trusted Monitor is configured with approved PreP measurements and the identity of the TPM in the user's computer.

4.8.2 Bumpy Design Alternatives

`@@` at Any Time. As presented, the secure attention sequence for Bumpy is the `@@` sequence immediately following a *focus* event from the web browser GUI. There are no technical limitations to enabling a secure attention sequence at any time, regardless of where in a field the cursor may be. However, we anticipate significant usability challenges for all but the most savvy users. This may prove to be an interesting direction for future work.

Editing Bumpy-Protected Input. As presented (Section 4.2.1), Bumpy ignores non-display characters that do not cause a *blur* event in the web browser GUI while the user is entering sensitive data. Examples of such

characters are backspace and the arrow keys. Here too, there are no technical limitations to enabling the user to edit her opaque (from the browser's perspective) data. However, we are concerned about a malicious browser tampering with the cursor and confusing the user. Additional investigation is warranted to determine whether this attack amounts to anything beyond a denial-of-service attack (e.g., to get better data for a keystroke timing attack [163]).

Trusted Path Between Trusted Monitor and Webserver. There are many circumstances where the lack of a trusted path from a remote server to a user with a compromised computer can lead to the user's loss of sensitive information. For example, when a remote server checks an attestation from the user's computer and finds known malware installed, it is desirable to inform the user that her system is compromised. Other researchers have considered the use of PDAs or smartphones in such roles (e.g., Balfanz et al. [9]), but we consider this enhancement to Bumpy to be beyond the scope of the current paper.

PreP as Password Store. The direct-encryption PoPr breaks the web browser's ability to remember passwords on behalf of the user. This feature can be reenabled using the PreP or PoPr as a password store, and the Trusted Monitor as the interface to select a stored password.

4.8.3 Other Interesting Features

Password Leak Detection. A compelling feature that can readily be added to a PreP is to look for the user's password(s) in the input stream and detect whether it appears when input protections are not enabled. This may allow the system to issue a warning if, e.g., the user is about to fall victim to a phishing attack.

Hardware Keyloggers. Resistance to physical attacks is not an explicit goal of Bumpy; however, the issue warrants discussion. Bumpy's resilience to hardware keyloggers depends on the model used for associating new input

devices with the user's computer. If a simple plug-and-play architecture is allowed, then a hardware keylogger inserted between the input device and the user's computer can appear as a new input device to the computer, and a new computer to the input device. One alternative is for input devices to require manufacturer certification before the user's computer will associate with them. However, this may prove to be impractical, as users may perceive all certification errors as indicative of a broken device. The core research challenge here is the problem of key establishment between devices with no prior context (Chapter 5) [10, 166].

4.9 Summary

User input to today's commodity applications and operating systems is exposed to host-based malware. A viable alternative protects user input while simultaneously giving feedback to the user and the intended destination for her input that protections are in place. We develop such an alternative on top of the Flicker architecture, thereby demonstrating its utility in improving the security properties of real applications on today's systems.

Bumpy protects users' sensitive input from keyloggers and screen scrapers by excluding the legacy OS and software stack from the TCB for input. Bumpy allows users to dictate which input is considered sensitive, thus introducing the possibility of protecting much more than just passwords. Bumpy allows webservers to define how input that their users deem sensitive is handled, and further allows users' systems to generate attestations that input protections are in place. With a separate local device, Bumpy can provide the user with a positive indicator that her input is protected. We have implemented Bumpy and show that it is efficient and compatible with existing legacy software.

Chapter 5

Seeing-is-Believing: Using Camera Phones for Human-Verifiable Authentication

Chapters 3 and 4 depend on TPM-based attestation to convince a verifier that code has executed with hardware-enforced isolation. Fundamental to checking attestations is the verifier's knowledge of the identity of the TPM generating the attestation, which is the TPM's public Endorsement Key (EK). While there are proposals for a public key infrastructure (PKI) that enables the verifier to confirm that a given Endorsement Key Certificate corresponds to a specification-compliant TPM [172], two problems remain. First, this PKI exists only on paper. Second, even if it did exist, a digital certificate is insufficient to provide a simple human-verifiable binding between an Endorsement Key Certificate and a *physical* computer whose TPM contains this EK.

For the security-conscious user who wishes to ascertain the true identity of the TPM in her computer, the lack of a physical binding is a serious limitation. It leaves room for a form of man-in-the-middle (MITM) attack called

a proxy attack, where malicious code on the user's machine hides by proxying all TPM requests to an attacker-controlled machine. More generally, a MITM attack takes place when Alice and Bob believe they are communicating with each other, when in fact they are both communicating with Charlie, who is able to monitor, modify, inject, suppress, or otherwise tamper with Alice and Bob's intended communication without their knowledge.

An out-of-band communication channel that can provide an authentic copy of the public EK suffices to defeat proxy attacks. The challenge, then, is to construct an out-of-band channel that provides authenticity for the exchange of public keys while unambiguously showing the human user which physical device originates the public key. We also seek to achieve these properties using the interfaces present on current devices. Balfanz et al. refer to such an authentication mechanism as providing *demonstrative identification* of the communication devices [10]. We approach this problem with the premise that, in many situations, a user can *visually* identify the desired device.

We propose to use the camera on a mobile phone as a new *visual channel* to achieve demonstrative identification of communicating devices formerly unattainable with wireless communication. We term this approach Seeing-is-Believing (SiB). In SiB, one device uses its camera to take a snapshot of a barcode encoding cryptographic material identifying, e.g., the public key of another device. We term this a *visual channel*. Barcodes can be pre-configured and printed on labels attached to devices, or they can be generated on-demand and shown on a device's display.

As camera-equipped mobile phones rapidly approach ubiquity, these devices become an excellent platform for security applications that can be deployed to millions of users. Today's mobile phones increasingly feature Internet access, cameras, high-quality displays, and short-range Bluetooth wireless radios. They can perform public-key cryptographic operations in under one second.

In addition to identifying the TPM in a computer, we apply this visual channel to bootstrap authenticated key exchange between devices that share no prior context, including such devices as mobile phones, wireless

access points, and public printers. This enables users to configure two devices to communicate over a secret and authentic channel, e.g., to exchange sensitive documents or personal messages, even if the underlying communication primitive is insecure (i.e., wireless). We also use SiB to secure device configuration in the context of a smart home.

5.1 Seeing-is-Believing (SiB)

With SiB, a mobile phone's integrated camera serves as a visual channel to provide demonstrative identification of the communicating devices to the user while also providing an out-of-band mechanism for exchanging authentic information. By *demonstrative identification*, we mean the property that the user is sure her device is communicating with *that* other device. In SiB, the user identifies *that* other device visually. This serves to strongly authenticate data from the other device since the user knows precisely which devices are communicating. Thus, SiB can be used to bootstrap authentic and secret communication, thereby defeating man-in-the-middle attacks while allowing the use of convenient wireless communication. SiB also captures user intentions in an intuitive way. What better way for a user to tell device A that it should communicate securely with device B than to take a picture of device B using device A's integrated camera?

In the remainder of this section, we detail the physical realization of the visual channel with 2D barcodes. The use of the visual channel to bootstrap secure communication is then illustrated with a specific example. We end this section with a discussion on using SiB with devices that may be lacking a display or a camera, or both. Sections 5.2 and 5.3 then provide detailed usage scenarios for the demonstrative identification provided by SiB. In Section 5.4, we move on to discuss a weaker – though still valuable – property that can be provided by the visual channel, which we term *presence*.

5.1.1 2D Barcodes as a Visual Channel

We implement the visual channel with a 2D barcode (e.g., Data Matrix [77]), displayed by or affixed to one device and captured by another with its digital camera. When a user executes the SiB protocol, she must aim the camera of her mobile device at a barcode on another device (either displayed electronically or affixed to the device's housing). The act of aiming the camera at the desired device results in demonstrative identification of the targeted device. We say that the device displaying the barcode is in *Show* mode, and that the device whose camera is active is in *Find* mode.

We now present a more detailed example of the use of SiB. Suppose Alice and Bob want to set up a secure channel between their camera phones. Alice's phone generates a 2D barcode encoding appropriate public cryptographic material and *Shows* it on its screen, while Bob uses his phone's digital camera in *Find* mode to take a snapshot of Alice's screen displaying the barcode. Bob must watch his phone's LCD, acting as viewfinder, updating in real time in response to his positioning of his camera-phone. A barcode recognition algorithm processes each image in the viewfinder in real time and overlays a colored rectangle around recognized barcodes. Once a barcode is successfully recognized, the view-finding process stops and the barcode recognition and error-correcting algorithms return the data represented by the barcode. Section 5.5 presents further details of our implementation.

5.1.2 Pre-Authentication and the Visual Channel

We build on work by Balfanz et al. [10], and Stajano and Anderson [166], to secure wireless communication by leveraging an out-of-band channel for authentication. Our out-of-band channel is the visual channel. We adopt the term pre-authentication, as Balfanz et al. suggest [10], to describe the authentic data exchanged on the visual channel. Pre-authentication data is later used to authenticate one or both of the communicating parties in almost any standard public-key communication protocol over the wireless link. Eavesdropping on the visual channel gives no advantage to an attacker, provided that the underlying cryptographic primitives are secure, and that

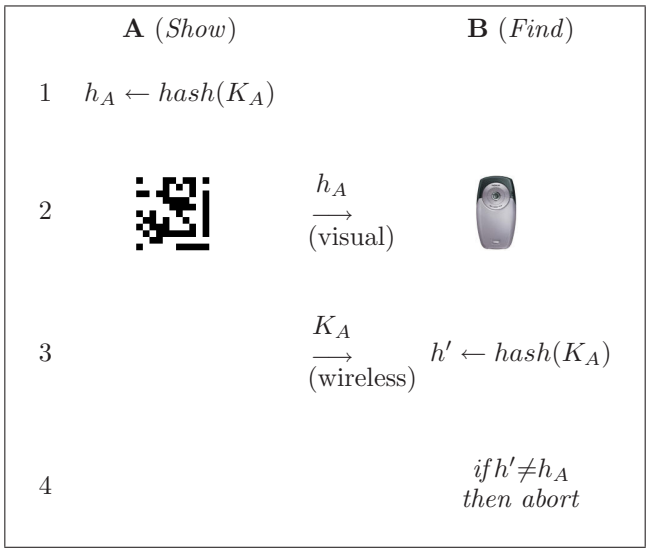


Figure 5.1: Pre-authentication over the visual channel. K_A is A's public key, which can be either long-term or ephemeral, depending on the protocol.

the mobile devices themselves have not been compromised.

Balfanz et al. discuss the use of infrared communication as a "secure side-channel" for pre-authentication between mobile devices [10]. They focus on the property that infrared is a "location-limited channel," emphasizing the difficulty an attacker faces in trying to interfere with the channel, because he must be in close physical proximity to the communicating devices. The primary advantage of SiB is that it uses a visual channel instead of an invisible channel, thus adding a direct human factor. We acknowledge that attacks against infrared are difficult to perform, but we believe that the inability of the user to actually see which devices are communicating provides dangerous opportunities to an attacker.

Figure 5.1 shows the pre-authentication phase of SiB, carried out over the visual channel. Device A *Shows* its public key by displaying a hash of its key as a barcode: $h_A \leftarrow hash(K_A)$. The user of device B then aims her device's camera at the display of device A, causing software (in *Find* mode) on device B to process the barcode from device A's display: $A \overset{visual}{\rightarrow} B : h_A$. At this point, device B has an authentic copy of the hash of A's public key. We

say that this hash was conveyed via the visual channel. Device A can then send A's full public key to device B via the untrusted wireless connection: $A \overset{wireless}{\rightarrow} B : K_A$. After receiving K_A via the untrusted wireless connection, software on device B can recompute the hash of K_A ($h' \leftarrow hash(K_A)$) and compare the computed hash with the hash received via the visual channel: $h' \overset{?}{=} h_A$. If there is any discrepancy, device B aborts.

Provided that the mobile phone has not been compromised, and that the visual channel and relevant cryptographic primitives are secure against active adversaries (Section 5.7 presents a detailed security analysis), authentication in SiB requires merely that the user confirm her camera is pointed at the intended device.

5.1.3 Device Configurations

The concepts of SiB can be applied in different ways to devices with different capabilities, each equipped with either a camera and display, a camera only, a display only, or neither. In some cases, these device configurations impose some limitations on the strength of the achievable security properties. Table 5.1 summarizes these properties.

The most flexible configuration for SiB is when both devices have both a camera and a display – these have a CD in their column or row heading in Table 5.1. These devices can be mutually authenticated, since both possess cameras. Further, each device can make use of either a long-term public key or an ephemeral public key in each exchange, since barcodes containing keys are displayed on an electronic screen (as opposed to paper or some other fixed medium).

We refer to devices equipped with no display – devices without a D in their column or row heading in Table 5.1 – as "displayless" devices. These devices can be authenticated with a long-term public key. A barcode encoding a commitment to the key, or multiple barcodes encoding the key itself, must be affixed to the device's housing (e.g., in the form of a sticker). The issue of whether to use a commitment to a key, or the key itself, is addressed in Section 5.3.

		Y			
		CD	**C**	**D**	**N**
X	**CD**	✓	✓*	✓	✓*
	C	✓	✓*	✓	✓*
	D	presence	presence	×	×
	N	×	×	×	×

Legend	
✓	Strong authentication possible
✓*	Barcode label required on housing
presence	Confirm presence only
×	No authentication possible

Table 5.1: Can a device of type X authenticate a device of type Y? We consider devices with cameras and displays (CD), cameras only (C), displays only (D), and neither (N).

Entries in Table 5.1 marked *presence* indicate that demonstrative identification of communicating devices is unattainable, but a property we term *presence* is still achievable. Presence refers to the ability to demonstrate that a device is in view of someone. We describe this property in more detail in Section 5.4.

5.2 Bidirectional Authentication

Providing mutual authentication between mobile devices that share no prior context is a difficult problem. In this section, we show how SiB can be used to intuitively capture user intentions and establish a mutually authenticated security context between precisely the devices the user wants, without a trusted authority. Examples of the established security context include authenticated exchange of public keys, and an authenticated Diffie-Hellman key exchange to establish a shared secret [38]. The device combinations we consider in this section are those where both devices have cameras.

We now walk through the use of SiB, beginning with device discovery and barcode generation. Next, we describe pre-authentication and bootstrapping

a well-known public key protocol. Then, we describe options to satisfy different security requirements and project the likely performance of SiB on emerging mobile phones.

The SiB protocol begins when Alice and Bob decide they want to communicate securely. They must decide upon whose device will *Show* initially, and whose device will *Find*. The *Showing* device computes a commitment to its public key material and generates a barcode encoding this commitment, and any necessary network information to establish a wireless connection. The key material can take the form of a user's long-term public key, or it can be an ephemeral key for use in only one key exchange. One practical example of this key material is a self-signed public key certificate extended with additional information about the key owner (e.g., name, email address, etc., similar to a vCard [36, 71]). The decision regarding what form of public key material to use is orthogonal to the authentication provided by SiB.

The pre-authentication phase now begins. The users take turns *Showing* and *Finding* (displaying and taking snapshots of) their respective barcodes. The order is not important, but it is necessary that Alice's device capture the barcode commitment to Bob's public key, and that Bob's device capture the barcode commitment to Alice's public key. This pre-authentication protocol is secure as long as an attacker cannot find a second pre-image for the commitment function, and is unable to perform an active attack on the visual channel.

After pre-authentication is complete, both devices now hold commitments to the other device's public key, and the devices can exchange public keys over the wireless link. The devices then perform the same commitment function over the other device's public key, ensuring that the result matches the commitment that was received over the visual channel. At this point, the devices have mutually authenticated one another's public keys, and Alice and Bob achieve demonstrative identification that the devices in their hands are the ones that are communicating. These authenticated public keys can then be used appropriately in any well-known public-key protocol on the wireless link (e.g., Diffie-Hellman [38], signed email, IKE [65], SSL/TLS [37]). It is imperative that in the chosen protocol, each party

verifies that the other does in fact hold the private key corresponding to its authenticated public key.

A user may desire to protect their privacy by avoiding transmission of their public key on the wireless network. For example, public key transmission may allow eavesdroppers to ascertain which devices are communicating. The user's public key can be encoded in a barcode directly, or in a sequence of barcodes if a single barcode has insufficient data capacity. The key is thereby obtained by the other device without transmitting it on the wireless medium, while retaining the demonstrative identification property with respect to the device originating the key. It is then advisable that the public key protocol that is used with SiB authentication is key-private [17].

As the processing and display capabilities of mobile phones improve, visual channel bandwidth will improve sufficiently for data transmitted over the visual channel to include network addresses for the relevant wireless interfaces (e.g., Bluetooth, 802.11) in addition to authentication data. This is more convenient for the user, since she never has to wait for discovery of neighboring devices or select a device from a list. Madhavapeddy et al. use barcodes on camera phones to speed up the Bluetooth device discovery process in this way [103].

5.3 Unidirectional Authentication

We now discuss entries from Table 5.1 where the device of type X (the authenticator) is equipped with a camera, and the device of type Y (the device being authenticated) lacks a display and a camera. It is this presence of a camera on the authenticator, and lack of a display and a camera on the device being authenticated, that are responsible for the security properties of this particular device combination. We refer to a device of type X as camera-equipped, and a device of type Y as displayless.

Displayless devices do not have the ability to display newly generated values. Still, a camera-equipped device can authenticate displayless devices and establish secure communication channels. The displayless device must be equipped with a long-term public/private keypair, and a sticker con-

taining a barcode of a commitment to its public key must be affixed to its housing. Since the displayless device is constrained to the use of a single public/private key pair for its entire lifetime, the option to generate per-interaction public keys no longer applies. Of course, devices can be reprogrammed and new stickers affixed, but we consider this to be a significant maintenance task. As in Section 5.2, there are privacy issues with using fixed public keys that might be of concern. We now discuss three scenarios where unidirectional authentication is valuable.

Trusted Platform Module. A TCG-compliant computing platform contains a Trusted Platform Module security chip to help protect the platform against software-based attack [172]. Chapter 2 contains additional background on TCG technologies. However, the current specification does not give the owner of the platform a definitive mechanism to authenticate the TPM in *her* machine. Rather, the specification provides protocols to learn that one is communicating with a "valid" (specification-compliant) TPM. Unfortunately, this leaves room for a proxy attack whereby malicious code leverages a remote, benign platform to respond to local TPM challenges. We propose the use of a barcode to encode the TPM's public Endorsement Key Credential, thereby enabling SiB-based demonstrative identification of the TPM in *this* computing platform for a physically present user. We discuss this further in Section 5.6.1.

Wireless Access Point. An 802.11 access point (AP) is one example of a class of devices where "sticker-based" authentication may be desirable. Camera-enabled devices can authenticate the AP, enabling the establishment of a secure link-level connection between the camera-enabled device and the AP. This solution also enables deployment of wireless connectivity in environments where security policies require physical presence for network access. Figure 5.2 shows the SiB application on a mobile phone scanning a barcode installed on a wireless access point.

Figure 5.2: Phone running SiB scanning a barcode on an 802.11 access point.

Public Printer Another application where demonstrative identification of communicating devices is desirable is when using a printer in a public place. Similar to the wireless access point, the printer can have a barcode affixed to its housing so that a user can use SiB to authenticate wireless communication with the printer or print server and bootstrap the establishment of a secure connection. Secure communication is important here not only to ensure the secrecy of the printed document, but to prevent a man-in-the-middle attack used to inject malicious software onto the user's computer by masquerading as a printer driver.

5.4 Presence Confirmation

A display-only device (display-equipped and cameraless) is unable to strongly authenticate other devices using SiB. Equipped with no camera, it makes no difference whether the entity the cameraless device wants to authenticate has a display, or makes use of a barcode sticker – the cameraless device cannot "see" them. However, display-only devices can obtain a property we refer to as *presence* (Table 5.1). That is, it can confirm the presence of some other device in line-of-sight with its display.

To detect the presence of a nearby device, the display-only device generates a key K for a message authentication code (MAC), encodes it in a barcode, and displays that barcode, noting the time when it was first displayed. Any nearby devices that are able to see the display and capture the barcode can send data to the display along with a MAC computed over that data: $\{data, MAC(K, data)\} \rightarrow$ *display-only device*.

When the data and MAC arrive over the wireless channel, the display-only device knows that some device has been in line-of-sight during the time since K was first displayed. We emphasize that this *presence* property is quite weak – the display-only device has no way of knowing how many devices can see its display, or whether the radio signal is from the same device that is in line-of-sight with its display. It can only verify the MAC computed over the data received via the wireless channel, and it can measure the delay between displaying the barcode and receiving the MAC on the wireless channel.

Despite the weakness of the presence property, there are still practical applications for devices capable of determining presence. For instance, the presence property is useful in the context of a smart home. It can restrict remote control access of a television to users in the same room. In general, it can serve to limit authority to control a device to users located in view of that device.

Consider the establishment of a security context between a TV and a DVD player to secure wireless communication between the two. The user can use SiB to strongly authenticate the DVD player to her phone through a barcode attached to the DVD player's housing. She can then demonstrate the DVD player's presence to the TV by sending it the public key of the DVD player, along with a MAC over the DVD player's public key:
$\{K_{DVD}, MAC(K, K_{DVD})\} \rightarrow TV$.

The TV is then configured to establish a secure, authenticated connection to the DVD player whenever the user selects the DVD player as the active input source on the TV. Taken one step further, the TV can add the DVD player to its list of trusted devices, such that the TV will automatically accept input from the DVD player whenever the user inserts a DVD.

Following the initial publication of SiB [110, 111], Saxena et al. extended the presence property to achieve authentication if users are willing to perform an integrity check [146]. They devise *Visual authentication based on Integrity Checking* (VIC), where both devices compute a common checksum on exchanged public data, and compare their results via a unidirectional SiB session on the visual channel. VIC is applicable when both devices have an electronic display, and at least one device has a camera. VIC requires user A to prompt user B as to whether B's device accepted or rejected. User A must then press a button on her device to indicate whether B's device output accept or reject. While this step requires user diligence, it is a simple binary comparison, and may be a viable option when mutual authentication with SiB is not possible or prohibitively complex.

5.5 Implementation Details

5.5.1 Series 60 Phone Application

We built SiB in C++ such that it will run on mobile phones running Symbian OS (tested with versions 6.1, 7.0s, and 8.1a) with the Nokia Series 60 User Interface. The size of the Symbian Installation System (SIS) file for SiB is only 52 KB, including a full implementation of RSA. This makes deployment feasible over even the most constrained channels, such as General Packet Radio Service (GPRS).

The Nokia N70 is our development platform today, though we initially developed SiB on the Nokia 6600 and 6620. To present a sense of the user experience with SiB, Figure 5.3 contains a photograph of SiB in action. Alice's Nokia 6620 (background), is displaying a barcode, while Bob's Nokia 6620 (foreground) is successfully decoding the data encoded in Alice's phone's barcode. In bidirectional authentication with SiB, Alice and Bob would then switch roles. Bob's phone would display a barcode, and Alice's phone would decode it.

The barcode format and image processing algorithm in our system is adapted from *Visual Codes* [136]. The data contained in the barcodes

Figure 5.3: SiB application on a Nokia 6620 with one phone scanning a barcode on the LCD of another.

for SiB is augmented with Reed-Solomon error correcting codes to provide better performance in the presence of errors in the image processing [134]. We ported Karn's implementation of Reed-Solomon codes to Symbian OS [87]. The SHA-1 cryptographic hash function is used for all hashing operations [82], and all wireless communication occurs via Bluetooth [62].

It is worth pointing out that the last three years have seen a great deal of development in barcode processing on mobile phones. A Java standard for mobile devices has been published that specifies the use of 2D barcodes [84]. While phones adhering to this Java specification are not yet available, phones are available today that include native barcode processing support, such as the Nokia N95. We plan to update our implementation to take advantage of this support.

To enable users to perform a key exchange between two camera phones, our application generates and maintains an RSA keypair representing the user's identity. We use the XySSL[1] library for RSA operations, including key generation, encryption, decryption, signing, and verification. Ephemeral

[1] `http://xyssl.org/`

Diffie-Hellman key exchange can also be used to establish a shared secret between the two devices [38], or users can upload their own key files from an existing application.

Bluetooth device discovery is a time consuming and error-prone process, since there is no user-friendly way to distinguish between two devices with the same Bluetooth Device Name. We eliminate the Bluetooth discovery process by including the Bluetooth MAC address in the barcode displayed by the first *Showing* device.

Thus, for a secure and usable SiB exchange, the device that *Shows* first needs to convey 48 bits of Bluetooth address and 160 bits of SHA-1 output (a total of 208 bits) in its barcode. Unfortunately, each *Visual Code* barcode has a useful data capacity of only 68 bits [136], since 15 of the 83 total bits in the raw barcode format are reserved for Reed-Solomon codes. We now describe how we use multiple barcodes to increase the effective bandwidth of the visual channel.

5.5.2 Visual Channel Bandwidth

The visual channel bandwidth between two devices can be increased by choosing a barcode format with a higher data capacity or by using multiple barcodes of a given capacity. There are two basic approaches to using multiple barcodes: cycle through the barcodes one-at-time, or tile the barcodes side-by-side. Cycling is necessary on an electronic screen that is two small to display tiled barcodes, such as the screen on a mobile phone. Tiling is necessary when when cycling is not feasible, due to barcodes being printed on a label instead of displayed electronically. Tiling is also an option on larger electronic displays. In both cases, the Reed-Solomon codes embedded in each barcode indicate whether a processed code is valid or invalid, enabling fully automated scanning of multiple barcodes.

5.5.2.1 Cycling Multiple Barcodes

Barcodes can be cycled as fast as the camera and recognition algorithm on the other device can process them. On the Nokia N70, we achieve good

results displaying each barcode for one seventh of a second on the *Showing* device, and configuring the camera on the *Finding* device to send bitmap images with a resolution of 160x120 to the recognition algorithm.

Encoding a 48-bit Bluetooth address and 160-bit SHA-1 output requires a total of four barcodes, including necessary sequencing information to allow the scanning device to properly reorder the scanned barcodes. The device that *Shows* first must include its Bluetooth address to enable the *Finding* device to initiate the Bluetooth connection between devices. Three barcodes suffice after the devices switch roles, since the Bluetooth connection is already established and only the output of SHA-1 and the sequencing information need to be encoded.

There is a limit to the corrective capability of the Reed-Solomon codes, and barcode scans with significant reading errors can cause the Reed-Solomon codes to report corrected errors when the data remains corrupted. We add an additional checksum to the data payload spread across multiple barcodes to detect and suppress these errors.

We performed timing analysis on our implementation of bidirectional authenticated RSA public key exchange between two Nokia N70s when operated by an experienced user (Table 5.2). We instrumented the application to track the length of time to generate the user's public key initially, though this is a one-time cost and may be unnecessary if the user has an existing public key on her desktop platform that can be copied to her mobile phone. We measured the length of time the user spends aiming her device before the four (first *Show*) or three (second *Show*) cycling barcodes are successfully recognized, as well as the length of time devices spend cycling their barcodes on-screen. Role-switch is automated via the Bluetooth connection between devices, so we present the latency involved in establishing the Bluetooth connection and causing each device to switch roles.

5.5.2.2 Tiling Multiple Barcodes

When scanning tiled barcodes, we configure the camera on the *Finding* device to return higher resolution 640x480 bitmaps. We have successfully

Operation	**Avg (s)**	**Stdev**
Generate Public Key	10.145	4.928
Recognize Barcodes	5.609	2.079
Establish BT Conn.	1.266	0.490
Display Barcodes	6.456	1.598
Save Public Key	0.016	0.000
Total Key Exchange	10.452	4.452

Table 5.2: Latency of mutual authenticated key exchange with SiB using our barcode-cycling implementation on Nokia N70s, including user-induced, computational, and Bluetooth overheads. The total is less than the sum of the components because operations may overlap, e.g., Alice's device may have completed barcode recognition and started establishing the Bluetooth connection while Bob's device is still displaying barcodes. RSA operations used 1024-bit keys.

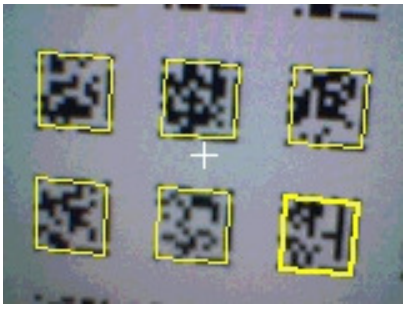


Figure 5.4: Mobile phone screen shot showing the SiB application on a Nokia 6620 recognizing multiple tiled barcodes displayed on an LCD screen.

scanned six tiled barcodes from a laptop's LCD in a single frame at this resolution on the Nokia 6620, as shown in Figure 5.4. The Nokia N70 supports image sizes up to 1600x1200, but the recognition algorithm imposes sufficient processing overhead to degrade the user's view-finding experience at higher resolutions. We conclude that scanning tiled barcodes for a single logical item is a viable implementation strategy.

5.6 Applications of Seeing-is-Believing

We initially developed SiB in 2004 [110]. Since then, we have gained some practical experience with its use in various circumstances, which we relate here.

5.6.1 Applications in Trusted Computing

The Trusted Computing Group (TCG) has specified a Trusted Platform Module (TPM), which is a dedicated security chip designed to increase the resilience of a computing platform to software-based attacks [172]. We provide background on the TCG and TPM in Chapter 2. The ability of SiB to demonstratively identify a computing device is useful in the context of trusted computing.

A TCG-style attestation is a digitally signed list of a particular set of programs. However, if the origin of the public key used to verify the signature cannot be confirmed, then it does not truly identify *which* platform loaded these programs. This enables a proxy attack, whereby a compromised machine that is challenged to attest its software state forwards the attestation request to a machine known to be in a benign state, and then forwards the resulting attestation back to the original challenger.

In many situations, it is imperative that the machine in front of the user is the one generating the attestation. For example, consider an extension to our Bumpy system (Chapter 4) that enables the Bumpy Trusted Monitor to verify an attestation from the user's platform. If equipped with the true identity of the TPM in the user's machine, a mobile device can perform the necessary computation to process an attestation from the user's machine.

A solution to the above problem is to enable the user to definitively identify the TPM in her machine, so that the true origin of any attestations purported to be from the user's machine can be verified. We propose that TPM-equipped machines include a barcode commitment to their Endorsement Key Certificate somewhere on the case, so that SiB can be used to ascertain the true identity of the TPM in that machine.

Note that an attacker with direct access to the computing platform can

subvert the TPM by physical means. Thus, the use of SiB enhances security under the assumption of software-only attacks and requires an attacker to have physical access to the computing platform, ruling out all remote attacks.

5.6.2 Seeing-is-Believing and the Grey Project

SiB has been in use at Carnegie Mellon for several years as part of the Grey Project [14]. Grey is an access control system with mobile phones as the primary development platform, and is currently in use by 25 people to access 35 office and laboratory doors. SiB is used in Grey to allow two users to exchange contact information, including users' public keys, in an authenticated manner. However, the implementation of SiB used by Grey is written in Java, and the performance impact of Java on the Nokia N70 is noticeable. We expect these problems to diminish with the next generation of mobile phones, where we hope to employ native barcode recognition in accordance with JSR-257 [84].

5.6.3 Group Key Establishment

Secure group communication requires the distribution of authentic information to group members' devices. We consider this problem in a context where members' devices share no prior context. SiB has proven to be quite usable for one-on-one exchange of information, such as between two people, or between one person and a device. However, as part of ongoing research on group key establishment, we have encountered some human-factors challenges when a large number of people try to perform SiB multiple times and in close proximity to one another. Recall that mutual authentication with SiB between two people requires a role switch, where the person whose device was initially in *Show* mode changes to *Find*, and vice versa.

In a group key establishment scenario, we have found that people often make one particular mistake. After performing the first half of a mutual exchange, they look for another person to exchange with, instead of performing the second half. For example, consider Alice, Bob, and Charlie,

where all three would like to establish a group key. Alice may photograph Bob's device, and then, when her device switches to *Show* mode, she may allow Charlie to photograph it, instead of Bob. This opportunity for confusion has proved a major obstacle for the development of a usable group key-establishment protocol.

In Section 5.1, we introduced the property that SiB should either establish authentic communication, or fail. This property applied to SiB in a single direction only, and we achieve mutual authentication by repeating the unidirectional exchange in the other direction. In a group scenario, we require a binding between both unidirectional exchanges. That is, Alice authenticates Bob's key, and then Bob authenticates Alice's key, or else the exchange fails. To make the scheme usable, failure should be an infrequent occurrence. Given our experience with group key establishment, it is worth considering how to achieve strong mutual authentication between two people with only a single unidirectional SiB step. In Section 5.4, we showed that unidirectional SiB can only provide *presence* to the display-only device, but that an extension to use *Visual authentication based on Integrity Checking* (VIC [146]) can provide mutual authentication if users are willing to press a button on one device based on whether the other device *accepts* or *rejects*. In particular, VIC reduces by a factor of two the number of SiB exchanges that must be done in a group scenario. It is the subject of future work to determine if this change results in fewer human errors.

5.7 Security Analysis

In addition to the security of the underlying cryptographic primitives, the security of SiB is based on the assumption that an attacker is unable to perform an active attack on the visual channel, and is unable to compromise the mobile device itself. We first discuss the employed cryptographic primitives, then the security properties of various side-channels for authentication. Finally, we discuss attacks against the visual channel.

5.7.1 Cryptography

Our implementation uses cycling barcodes that provide sufficient bandwidth to convey a full 160 bit SHA-1 hash. As discussed in Section 5.2, the hash transmitted in the barcode needs to be secure against active attacks, which we achieve through the properties of the visual channel. However, if an adversary can find a second pre-image of the value encoded in the barcode, then a passive attack on the barcode coupled with an active attack against the wireless network connection can be successful. For particularly cautious users, and as mobile phone cameras and displays increase in fidelity, the key itself can be encoded in the barcode, eliminating this dependence on a cryptographic hash function.

5.7.2 Selecting an Authentication Channel

Mutual authentication between two parties without the assistance of a trusted authority requires a channel that is secure against active attacks, such as a man-in-the-middle (MITM) attack. We analyze potential channels based on the degree to which the user's intentions are captured, and the amount of feedback that the channel provides to the user. Table 5.3 contains a summary of proposed channels and their characteristics.

Activity on channels such as infrared, ultrasound, or radio is undetectable to humans without specialized equipment. Therefore, if Alice believes her device is communicating with Bob's device via infrared, the only assurance she has that it is actually doing so is through status indicators on the two devices. She cannot see infrared radiation leaving her device and entering Bob's, and she certainly cannot see an attacker's device outputting interference patterns and affecting the data stream. Similarly, in case of ultrasound and radio, Alice and Bob need to rely on status indicators of their devices, but they are not sure that Alice's device is indeed setting up a key with Bob's device. Thus, the users' intentions are not captured well, and feedback is indirect and prone to error. Using an audible signal (marked "beeps" in Table 5.3) for data exchange is more intuitive, but this would not work well in noisy environments and is still prone to a man-in-the-middle

Channel	COTS	Resists MITM	Convenient
Ultrasound	○	○	●
Audible ("beeps")	●	◐	●
Radio	●	○	●
Physical Contact	○	●	●
Near Field Comm.	◐	◐	●
Wired Link	●	●	○
Spoken Passwords	●	●	○
Written Passwords	●	●	○
Visual Hash Verif.	●	●	◐
Infrared	●	◐	◐
Loud-and-Clear	●	●	◐
Seeing-is-Believing	●	●	●

Table 5.3: Characteristics of various channels proposed for authentication. We acknowledge that rating the convenience of a channel is subjective; however, we believe it is useful to compare various channels in this way. Section 7.5 contains a discussion of many of these alternatives. COTS indicates that the necessary hardware is already present in Commercial Off-The-Shelf products. Symbols: yes (●), partial (◐), no (○).

attack since it can be difficult for people to tell where "beeps" originate and how many devices are "beeping."

Physical contact between devices is much more intuitive for people and captures the intentions of the users – identifying the devices between which they want to establish a secure communication link [166]. Unfortunately, most current devices are not equipped with an interface for this purpose. This may change in the future, however, as Near Field Communication (NFC) interfaces have been standardized for use in mobile phones [84]. It is necessary to analyze the difficulty of performing an attack against an NFC device from a distance of several meters or more. An alternative approach is to use a wired link, for example connect both devices with a USB cable, however, this approach is not convenient to use and people would need to carry a wire with them.

Another approach is for Alice and Bob to establish a secret password, either by speaking the password aloud, or by writing passwords on paper and passing them to each other. Both Alice and Bob would then need to type in the password correctly, which the devices use to perform a secure password protocol, e.g., EKE [18]. We believe this approach is cumbersome in comparison with SiB, particularly on devices with a limited keyboard.

Finally, both devices could present a visual representation of the hash of the exchanged key material to detect a man-in-the-middle attack [39, 58, 98, 127]. Each user must then press a button to indicate whether the images on their devices are the same. These approaches, however, are not secure unless people carefully compare the output of the visual hash function. We believe SiB has an advantage here not just in ease-of-use but because strong authentication is intrinsically linked with device identification.

5.7.3 Attacks Against Seeing-is-Believing

Active attacks are extremely difficult to perform against the visual channel without being detected by the user. The user has in mind the device at which she is aiming her camera, and will be conscious of a mistake if she takes a snapshot of anything else. We believe the act of taking a picture of

that device – the one with which the user wants to communicate securely – is intuitive, and should therefore enjoy a low rate of operator error. Thus, the visual channel has the property of being resilient against active attacks (e.g., a man-in-the-middle attack), and the property that active attacks are easily detected by the user, who can then terminate wireless communication. It is ideal for authentication, providing the user with demonstrative identification of the communicating devices without burdening the user with device names or certificate management.

In Section 5.4, we discuss a presence property which requires the user to demonstrate that her device can see a display. Kuhn details some attacks which enable a malicious party to read the contents of a CRT screen without actually being in line-of-sight with it. For example, a sophisticated adversary may be able to measure emitted electromagnetic radiation [92], or to assemble the contents of the CRT by looking at reflected light from the CRT [91]. Defense against this form of attack is outside the scope of SiB.

An attacker can disrupt the lighting conditions around Alice and Bob in an attempt to disrupt SiB. However, changes of sufficient magnitude to impair SiB are easily observed by Alice, Bob, and any people in the vicinity, alerting them to some kind of unusual behavior. A more sophisticated, and subtle, attack is to use infrared radiation or a carefully aimed laser to overwhelm the CCD[2] in a phone's camera. If an attacker is able to flood an environment with sufficient infrared radiation or aim a laser directly at the camera's CCD, the CCD in a phone's camera can begin to saturate, and all attempts to take pictures will yield a picture with all pixels set at or above the intensity of the legitimate image, up to the maximum value for each pixel. Essentially, the image becomes noise. Alice will see that the image in her viewfinder is not the picture of Bob's phone that she expects, and can abort the protocol. We have experimented with an off-the-shelf red laser pointer and confirmed these claims.

Even without a user monitoring the process, the electronic-warfare-esque techniques necessary to cause the CCD to output a meaningful image other

[2]Charge Coupled Devices (CCDs) are the prevalent type of image sensor used in today's digital cameras.

than the scene in front of the camera are beyond the reach of all but the most sophisticated adversaries with current technology. We are unaware of any attacks feasible today which result in anything but noise from the camera under attack.

5.7.4 Sticker-based Attacks

We have described how devices without an electronic display may be equipped with stickers on which a barcode commitment to the device's public key is printed (Section 5.1.3). The obvious attack against these devices is to replace the sticker on the case with a new sticker encoding the public key of the attacker's device and perform a MITM attack on the wireless channel. Thus, some level of tamper-resistance may be desired for the stickers on, e.g., computers in Internet cafes.

An interesting alternative exists for devices that do have displays but may still employ sticker-based authentication, such as TPM-equipped computers. Future systems that employ secure or verifiable video subsystems may be able to enter a special mode where the barcode commitment to their public keys can be displayed electronically. If the size of the trusted computing base can be managed and verified, this may be a better option than using stickers. If nothing else, it can provide defense-in-depth in that the mobile device can photograph the barcode on the sticker and the display and check for discrepancy. The attacker will need to modify both the software running on the device and the sticker affixed to the device. Challenge-response protocols can also be employed to confirm that, e.g., the barcode on the display really does correspond to the TPM chip identified by the sticker.

5.8 Summary

This thesis develops mechanisms to verifiably execute code in isolation in Chapters 3 and 4. Here, we improve the strength of the verification by showing how to bind the cryptographic identity of the TPM in the user's computer with the physical identity of that computer. To this end, we

propose Seeing-is-Believing, a system that uses barcodes and camera phones as a visual channel for human-verifiable authentication. This channel rules out man-in-the-middle attacks against public-key-based key establishment protocols. The visual channel has the desirable property that it provides demonstrative identification of the communicating parties, providing the user with assurance that her device is communicating with *that* other device. We have also analyzed the establishment of secure, authenticated sessions between SiB-enabled devices and devices missing either a camera, a display, or both, and found that secure communication is possible in many situations.

Chapter 6

Recommendations for Hardware-Supported Minimal TCB Code Execution

In this chapter, we make hardware recommendations to alleviate the Flicker performance issues we summarize in Section 3.7.4. Our investigation reveals that by combining alterations to Flicker with hardware modifications to improve performance and concurrency, we can achieve efficient minimal TCB code execution. In other words, we can avoid today's performance issues and execute application code while trusting only the mandatory TCB. We emphasize isolation, secure initialization, and external verification.

Although other researchers have proposed compelling hardware security architectures, e.g., XOM [99] or AEGIS [167], we focus on hardware modifications that tweak or slightly extend existing hardware functionality. We believe this approach offers the best chance of seeing hardware-supported security deployed in the real world. Through a series of experiments on existing commodity hardware, we show that our recommendations promise significant performance improvements.

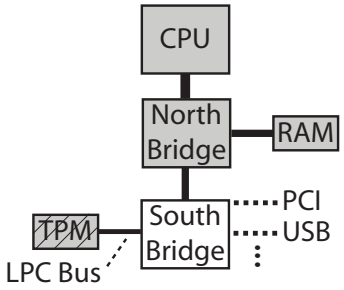


Figure 6.1: Chipset configuration for a modern x86 computer. Shaded components are part of the minimal TCB for our execution model. The TPM is shaded differently because it is included for practical reasons but is not an essential part of a stored-program computer architecture.

6.1 Security Properties

Isolation. Execution of the PAL must be protected from legacy software on the platform, as well as the hardware components not included in the TCB shown in Figure 6.1. At the same time, to maintain reasonable performance, we need to be able to execute a PAL concurrently with legacy software. On a system with a single CPU, virtual concurrency is achieved by rapidly context switching between threads of execution. This requires a secure mechanism to protect the secrecy and integrity of PAL execution state while other code executes. Given the trend towards multi-core CPUs, PAL state must also be protected *during* execution, since malicious code may be running concurrently on another CPU.

Secure Initialization. The isolation described above is only useful if PAL execution can be securely initiated. In other words, the legacy software cannot be trusted to properly initialize the protections necessary for the PAL's protection. Hence a mechanism is needed that provides a "clean slate" for PAL execution without actually rebooting the platform.

External Verification. The isolation and secure initialization properties allow a PAL to execute unmolested. However, an external party that de-

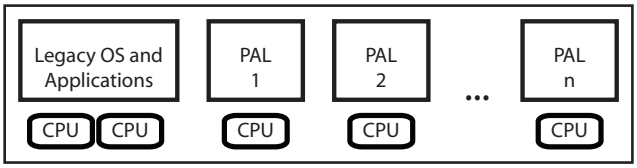


Figure 6.2: Physical platform running a legacy OS and applications along with some number of PALs.

pends on outputs from the PAL must be able to distinguish between a PAL that was executed with full hardware protections and a PAL that was executed in a malicious, e.g., virtual, environment.

6.2 Overview of Recommendations

To meet these security requirements, we establish two goals for our recommendations: (1) to enable the concurrent execution of an arbitrary number of mutually-untrusting PALs alongside an untrusted legacy OS and legacy applications, and (2) to enable performant context switching of individual PALs. A system achieving these goals supports multiprogramming with PALs, so that there can be more PALs executing than there are physical CPUs in a system. It also enables efficient use of the execution resources available on today's multi-core computing platforms. Figure 6.2 shows an example of our desired execution model. Note that we assume that a PAL only executes on one CPU core at a time, but Section 6.10 discusses extension to multiple cores.

We have two requirements for the recommendations we make. First, our recommendations must make minimal modifications to the architecture of today's trusted computing technologies: AMD SVM and Intel TXT. Admittedly, such a requirement narrows the scope of our creativity. However, we believe that by keeping our modifications minimal, our recommendations are more likely to be implemented by hardware vendors. Second, in order to keep our execution architecture as close to today's systems architectures as possible, we require that the untrusted OS retain the role of the

resource manager. With this requirement, we open up the possibility that the untrusted OS could perform denial-of-service attacks against the PALs. However, we believe this risk is unavoidable, as the untrusted OS can always simply power down or otherwise crash the system.

There are two new hardware mechanisms required to achieve our desired execution model (Figure 6.2) while simultaneously achieving the two goals stated above. The first is a hardware mechanism for memory isolation that isolates the memory pages belonging to a PAL from all other code. The second is a hardware context switch mechanism that can efficiently suspend and resume PALs, without exposing a PAL's execution state to other PALs or the untrusted OS. In addition to these two mechanisms, we also require modifications to the TPM to allow external verification via attestation when multiple PALs execute concurrently.

In the rest of this section, we first describe PAL launch (Section 6.3), and our proposed hardware memory isolation mechanism (Section 6.4). Section 6.5 talks about the hardware context switch mechanism we propose. In Section 6.6 we describe changes to the TPM chip to enable external verification. We describe PAL termination in Section 6.7. Section 6.8 ties these recommendations together and presents the life-cycle of a PAL. Finally, Section 6.9 summarizes the expected performance improvement of our recommendations.

6.3 Launching a PAL

We propose a mechanism for securely launching a PAL to achieve the *Secure Initialization* security property from Section 6.1.

6.3.1 Recommendation

First, we recommend that the untrusted OS allocate resources for a PAL. Resources include execution time on a CPU and a region of memory to store the PAL's code and data. We define a *Secure Execution Control Block* (SECB, Figure 6.3(a)) as a structure to hold PAL state and resource alloca-

tions, both for the purposes of launching a PAL and for storing the state of a PAL when it is not executing. The PAL and SECB should be contiguous in memory to facilitate memory isolation mechanisms. The SECB entry for allocated memory should consist of a list of physical memory pages allocated to the PAL.

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

1. reinitializes the CPU on which it executes to a well-known trusted state,
2. enables hardware memory isolation (described in Section 6.4) for the memory region defined in the SECB and for the SECB itself,
3. transmits the PAL to the TPM to be measured (described in Section 6.6),
4. disables interrupts on the CPU executing *SLAUNCH*,
5. initializes the stack pointer to the top of the memory region defined in the SECB (allowing the PAL to confirm the size of its data memory region),
6. sets the *Measured Flag* in the SECB to indicate that this PAL has been measured, and
7. jumps to the PAL's entry point as defined in the SECB.

6.3.2 Suggested Implementation

We can modify the existing hardware virtual machine management data structures of AMD and Intel to realize the SECB. Both AMD and Intel use an in-memory data structure to maintain guest state. These structures are the Virtual Machine Control Block (VMCB) and Virtual Machine Control Structure (VMCS) for AMD and Intel, respectively. The functionality of *SLAUNCH* when used to begin execution of a PAL is designed to give the same security properties as today's *SKINIT* and *SENTER* instructions.

6.4 Hardware Memory Isolation

To securely execute a PAL using a minimal TCB, we need a hardware mechanism to isolate its memory state from all devices and from all code executing on other CPUs (including other PALs and the untrusted OS and applications). This mechanism will achieve the *Isolation* property from Section 6.1.

6.4.1 Recommendation

We propose that the memory controller maintain an access control table where each entry specifies which CPUs (if any) have access to a range of physical pages. A naive solution is an $M \times N$ bitmap, where M is the number of physical pages present on the platform and N is the maximum number of CPUs. However, the naive solution wastes area on the CPU chip. A better solution is one where arbitrary ranges of memory can be specified. Other multiprocessor designs use a similar partitioning system to protect memory from other processors [95]. To use the access control table, the memory controller must be able to determine which CPU initiates a given memory request.

Figure 6.3(b) presents the state machine detailing the possible states of an entry in the access control table as context switches (described in Section 6.5) occur. Memory pages are by default marked `ALL` to indicate that they are accessible by all CPUs and DMA-capable devices. The other states are described below.

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.

6.4.2 Suggested Implementation

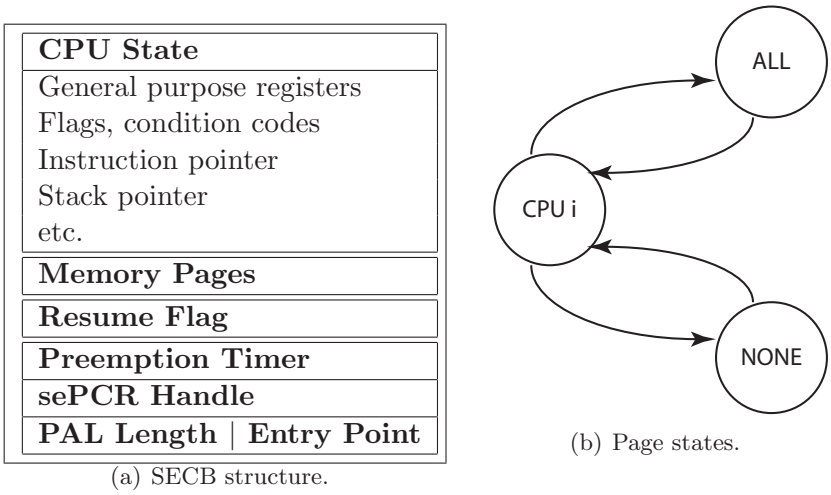
We can realize hardware memory isolation as an extension to existing DMA protection mechanisms. As noted in Chapter 2.2, AMD SVM and Intel TXT already support DMA protections for physical memory pages. In both protection systems, the memory controller maintains a bit vector with one bit per physical page. The value of the bit indicates whether the corresponding page can be accessed (read or written) using a DMA operation. One implementation strategy for our recommendations may be to increase the size of each entry in this protection table to include a bit per CPU on the system.

Existing memory access and cache coherence mechanisms can be used to provide the necessary information to enforce memory isolation. Identifying the CPU from which memory requests originate is straightforward, since memory reads and writes on different CPUs already operate correctly today. For example, every memory request from a CPU in an Intel system includes an *agent ID* that uniquely identifies the requesting CPU to the memory controller [152].

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

6.5 Hardware Context Switch

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



(a) SECB structure.

(b) Page states.

Figure 6.3: State machine for the possible states of a memory page in our proposed memory controller modification. The states correspond to which CPUs can access an individual memory page.

6.5.1 Recommendation

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

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

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

6.5.2 Suggested Implementation

We achieve significant performance improvements by eliminating the use of TPM sealed storage as a protection mechanism for PAL state during context switches. Existing hardware virtualization extensions of AMD and Intel support suspending and resuming guest VMs.[1] We can enhance these mechanisms to provide secure context switch by extending the memory controller to isolate a PAL's state while it is executing, even from an OS. Table 6.1 shows that with current hardware, VM entry and exit overheads are on the order of half a microsecond. Reducing the context switch overhead of between approximately 200 ms and a full second for the TPM sealed storage-based context switch mechanism (recall Figure 3.6) to essentially the overhead of a VM exit or entry would be a pronounced improvement.

6.6 TPM Support for Flicker

Thus far, our focus has been on recommendations to alleviate the two performance bottlenecks identified in Section 3.7.4. Unfortunately, the functional-

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

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

Table 6.1: Benchmarks showing the average runtime of VM Entry and VM Exit on the Tyan n3600R with a 1.8 GHz AMD Opteron and the MPC ClientPro 385 with a 2.66 GHz Intel Core 2 Duo.

ity of today's TPMs is insufficient to provide measurements, sealed storage, and attestations for multiple, concurrently executing PALs. These features are essential to achieve the *External Verification* property from Section 6.1.

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

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

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

However, the addition of sePCRs introduces several challenges:

1. A PAL must be bound to a unique sePCR (Section 6.6.1).
2. A PAL's sePCR must be inaccessible to all other code until the PAL

terminates (Section 6.6.2).

3. TPM Quote must be able to address the sePCRs when invoked from untrusted code (Section 6.6.3).
4. A PAL that used TPM Seal to seal secrets to one sePCR must be able to unseal its secrets in the future, even if that PAL terminates and is assigned a different sePCR on its next invocation (Section 6.6.4).
5. A hardware mechanism is required to arbitrate TPM access from multiple CPUs (Section 6.6.5).

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

6.6.1 sePCR Assignment and Communication

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

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

As part of *SLAUNCH*, the TPM returns the allocated sePCR's handle to the CPU executing the PAL. This handle becomes part of the PAL's state, residing in the CPU while the PAL is executing and written to the PAL's SECB when the PAL is suspended.[2] The handle is also made available to the executing PAL. One implementation strategy is to make the handle available in one of the CPU's general purpose registers when the PAL first gets control.

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

TPM Extend, Seal, and Unseal must be extended to optionally accept a PAL's sePCR as an argument, but only when invoked from within that PAL. The CPU, memory controller, and TPM must prevent other code from invoking TPM Extend, Seal, or Unseal with a PAL's sePCR. Enforcement can be performed by the CPU or memory controller using the CPU's copy of the PAL's sePCR handle. These restrictions do not apply to TPM Quote, as untrusted code will eventually need the PAL's sePCR handle to generate a TPM Quote. We describe its use in more detail in Section 6.6.3.

Note that the TPM in today's machines is a memory-mapped device, and access to the TPM involves the memory controller. The exact architectural details are chipset-specific, but it may be necessary to enable the memory controller to cache the sePCR handles during *SLAUNCH* to enable enforcement of the PAL-to-sePCR binding and avoid excessive communication between the CPU and memory controller during TPM operations.

6.6.2 sePCR Access Control

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

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

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

6.6.3 sePCR States and Attestation

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

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

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

After a TPM Quote is generated, the TPM transitions the sePCR to the `Free` state, where it is eligible for use by another PAL via *SLAUNCH*. This can be realized as a new TPM command, `TPM_SEPCR_Free`, executable from untrusted code. We treat the case where a PAL does not terminate cleanly in Section 6.7.

6.6.4 Sealing Data Under a sePCR

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

A PAL is assigned a free sePCR by the TPM when *SLAUNCH* is executed on a CPU. However, the PAL does not have control over *which* sePCR it is assigned. This breaks the traditional semantics of TPM Seal and Un-

seal, where the index of the PCR(s) that must contain particular values for TPM Unseal are known at seal-time. To meet Challenge 4, we must ensure that a PAL that uses TPM Seal to seal secrets to its assigned sePCR will be able to unseal its secrets in the future, even if that PAL terminates and is assigned a different sePCR when it executes next.

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

6.6.5 TPM Arbitration

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

6.7 PAL Exit

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

Normal Exit. The memory pages for a PAL that are inaccessible to the remainder of the system must be freed when that PAL completes execution. It is the PAL's responsibility to erase any secrets that it created or accessed before freeing its memory. To free this memory, we propose the addition of a new CPU instruction, *Secure Free* (*SFREE*). *SFREE* is parameterized with

the address of the PAL's SECB, and communicates to the memory controller that these pages no longer require protection. The memory controller then updates its access control table to mark these pages as `ALL` so that the untrusted OS can allocate them elsewhere. Note that *SFREE* executed by other code must fail. This can be detected by verifying that the *SFREE* instruction resides at a physical memory address inside the PAL's memory region. As part of *SFREE*, the CPU also sends a message to the TPM to cause the terminating PAL's sePCR to transition from the `Exclusive` state to the `Quote` state.

Abnormal Exit. The code in a PAL may contain bugs or exploitable flaws that cause it to deviate from the intended termination sequence. For example, it may become stuck in an infinite loop. The preemption timer discussed in Section 6.5 can preempt the misbehaving PAL, but the memory allocated to that PAL remains in the `NONE` state, and the sePCR allocated to that PAL remains in the `Exclusive` state. These resources must be freed without exposing any of the PAL's secrets to other entities on the system.

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

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

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

6.8 PAL Life Cycle

Figure 6.4 summarizes the life cycle of a PAL on a system with our recommendations. To provide a better intuition for the ordering of events, we step

through each state in detail. We also provide pseudocode for *SLAUNCH*, and indicate which states of a PAL's life cycle correspond to portions of the *SLAUNCH* pseudocode (Table 6.2).

Launch: Protect and Measure. The untrusted OS is responsible for creating the necessary SECB structure for a PAL so that the PAL can be executed. The OS allocates memory pages for the PAL and sets the PAL's preemption timer. The OS then invokes the *SLAUNCH* CPU instruction with the address of the SECB, initiating the transition from the `Start` state to the `Protect` state in Figure 6.4. This causes the CPU to signal the memory controller with the address of the SECB. The memory controller updates its access control table (recall Section 6.4) to mark the memory pages associated with the SECB as being accessible only by the CPU which executed the *SLAUNCH* instruction. If the memory controller discovers that another PAL is already using any of these memory pages, it signals the CPU that *SLAUNCH* must return a failure code. Once the memory protections are in place, the memory controller signals the CPU. The CPU inspects the *Measured Flag* and begins the measurement process since it is clear. The *Measured Flag* in the SECB (Figure 6.3(a)) is used to distinguish between a PAL that is being executed for the first time and a PAL that is being resumed. This completes the transition from the `Protect` state to the `Measure` state.

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

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

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

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

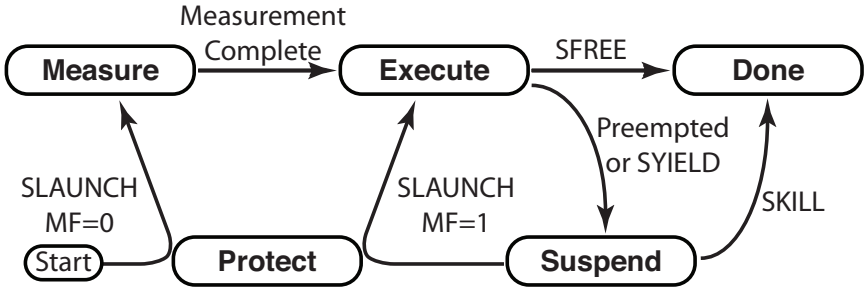


Figure 6.4: Life cycle of a PAL. MF stands for *Measured Flag*. Note that these states are for illustrative purposes and need not be represented in the system.

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

6.9 Expected Impact

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

With Flicker on existing hardware, a PAL yields by simply transferring control back to the untrusted OS. Resume is achieved by executing late launch again. It is the responsibility of the PAL to protect its own state before yielding, and to reconstruct the necessary state from its inputs upon

Start:
OS: Allocate pages for SECB S and PAL P
OS: Initialize SECB.pages
OS: Initialize SECB.timer
Protect:
CPU_i: *SLAUNCH* (S)
CPU_i: Reinitialize to trusted state
CPU_i: Disable interrupts
CPU_i to MC: SECB.pages
MC: if($\exists p \in$ SECB.pages s.t. $p.accessible =$ `NONE`) FAIL
MC: $\forall p \in$ SECB.pages: $p.accessible = \text{CPU}_i$
MC to CPU_i: done
CPU_i: ESP=SECB.pages.top
Measure:
if($\neg$SECB.MeasuredFlag)
CPU_i: send PAL to TPM
TPM: Allocate sePCR ℓ
MC: if($\neg\exists \ell \in$ sePCRs s.t. sePCR $[\ell]$.state = `Quote`) FAIL
TPM: h = SHA-1(PAL)
TPM: sePCR $[\ell] = 0$
TPM: sePCR $[\ell]$ = SHA-1(sePCR $[\ell]||h$)
TPM to CPU_i: done
CPU_i: SECB.MeasuredFlag = 1
Execute:
CPU_i: EIP=SECB.pages.eip
CPU_i: Begin executing

Table 6.2: *SLAUNCH* pseudocode.

resume. Protecting state requires the use of the TPM Seal and Unseal commands. An *SKINIT* on AMD hardware can take up to 177.52 ms (Table 3.9), while Seal requires 20-500 ms and Unseal requires 290-900 ms (Figure 3.7). Thus, context switching into a PAL (which requires unsealing prior data) can take over 1000 ms, while context switching out (which requires sealing the PAL's state) can require 20-500 ms. Further, existing hardware has no facility for guaranteeing that a PAL can be preempted (to prevent it from compromising system availability).

With our recommendations, we eliminate the use of TPM Seal and Unseal during context switches and only require that the TPM measure the PAL once (instead of on every context switch). We expect that an implementation of our recommendations can achieve PAL context switch times on the order of those possible today using hardware virtualization support, i.e., 0.6 μs on current hardware (Table 6.1). This reduces the overhead of context switches by six orders of magnitude (from 200-1000 ms on current hardware) and hence makes it significantly more practical to switch in and out of a PAL.

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

As an alternative to our recommended hardware modifications, we could instead consider increasing the speed of the TPM and the bus through which it communicates with the CPU. As shown in Section 3.7, the TPM is a major bottleneck for efficient Flicker applications on current hardware. Increasing the TPM's speed could potentially reduce the cost of using the TPM to protect PAL state during a context switch, and similarly reduce the penalty of using *SKINIT* during every context switch. However, achieving submicrosecond overhead comparable to our recommendations would require significant hardware engineering of the TPM, since many of its operations use a 2048-bit RSA keypair. Even with performant hardware, the power consumed by such operations is wasteful, since we may achieve superior performance with our less power-intensive modifications.

6.10 Extensions

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

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

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

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

interrupts. Routing only the interrupts of interest to the PAL requires the CPU to reprogram the interrupt routing logic every time a PAL is scheduled, which may create undesirable overhead or design complexity.

Chapter 7

Related Work

We now discuss related work that is not essential background for understanding this thesis (Chapter 2). We first relate systems for providing isolated execution (Section 7.1) and then discuss other research on remote attestation (Section 7.2). We also consider works on protecting user I/O (Section 7.3), web browser security (Section 7.4), and authentication without prior context (Section 7.5).

7.1 Isolation

Virtualization [131] and microkernels [1] are two technologies capable of providing strong isolation and reduced TCB size compared to commodity monolithic kernels (e.g., Microsoft Windows and Linux). Numerous works leverage these foundations.

Around 1980, VM370 [35] received a security retrofit in the form of KVM370 [57, 147]. This effort to reduce the TCB settled on the VMM as the smallest desirable virtualization layer, with further reduction purportedly yielding little benefit [56]. Flicker is able to reduce the TCB further for application-specific functionality. The goal of a tiny system-wide TCB remains elusive.

NetTop [116] uses a virtual machine monitor (VMware) and operating system with MAC support (SELinux) to enable what were traditionally

physically separate computer terminals on the desks of government employees to be consolidated onto a single system. The NetTop architecture relies extensively on the security controls of the host OS, which suffers from excessive TCB complexity.

Exokernels are designed to enable application-level and application-specific resource management, thus reducing the kernel's responsibilities to securely multiplexing the available hardware resources [43]. The Denali isolation kernel is structured much like a VMM, but does away with faithful emulation of the underlying hardware to reduce complexity and increase performance [180]. This design comes at the cost of breaking compatibility with legacy operating systems, whereas Flicker remains compatible. Both of these designs reduce the TCB for a given application, but not to the same extent as Flicker.

EROS [153] is a capability-based microkernel with performance on the order of today's commodity operating systems. Asbestos [41] is an operating system designed with labels and isolation as priorities, enabling system-wide information flow controls. Singularity [72] uses formally verified and strongly typed channels between processes. Unfortunately, all three of these systems require applications to be ported to a fundamentally different interface.

Proxos [168] defines a system call routing language and allows Linux applications running on the Xen [12] VMM to specify that certain system calls and their arguments be handled by a special-purpose *private* OS. For example, all file operations on `/etc/secrets` can be routed to the private OS such that the legacy OS never sees the file's contents. Note that applications' memory must be managed by the underlying, trusted VMM so that the legacy OS cannot simply access the secrets in the application's memory space. This architecture resembles that of Exokernels and Denali in that applications have control over many security-sensitive functions that are traditionally in the operating system's purview. Proxos has the advantage that legacy operating systems and applications can continue to run, but the extent of TCB minimization in their prototype remains orders of magnitude behind that of Flicker.

CHAOS [30], Overshadow [31], and SP^3 [184] use a trusted VMM to

protect application data from an untrusted OS. These works present compelling ideas, but implementation details and experimental results are wanting. Source code for Flicker is publicly available.

Nizza is an architecture to reduce the TCB for security-sensitive applications by executing their sensitive components as *AppCores* on top of the *L4Env* [160], which leverages the L4 microkernel [100]. L^{4}Linux [67] is used to export the Linux ABI, thereby enabling legacy applications to run unmodified. We believe this architecture holds great promise for tomorrow's systems, but it remains underutilized today. Its TCB is also orders of magnitude larger than that of Flicker. The PERSEUS [128] architecture is a precursor to Nizza that also uses L^{4}Linux to support legacy applications. Nizza extends PERSEUS by demonstrating components for real-world applications.

Enforcer [106] binds the notion of software identity to a signing keypair, with emphasis on commodity hardware and existing applications. However, enforcer leverages the static root of trust for measurement (Chapter 2.1), which suffers from excessive TCB complexity.

IBM developed the rHype research hypervisor and subsequently applied their security technology to the sHype hypervisor security architecture [142]. Their goal is to implement mandatory access controls at the hypervisor level. While compelling, their implementation is built for Xen [12], which consists of tens of thousands of lines of code for the hypervisor [104], not to mention the complete Linux kernel running in the privileged domain 0.

SELinux [161, 165] enables enforcement of extremely fine-grained security policies for the Linux kernel. However, this fine granularity has resulted in policy complexity [78] that makes analysis of the resulting security properties somewhat untenable.

AEGIS [167] is an entirely new hardware architecture designed to be secure against both physical and software attacks. Currently, only simulated performance results are available, and compatibility with existing software is dubious. XOM [99] proposes hardware changes to provide security properties for applications, even if the OS or VMM is compromised. However, XOM breaks compatibility with legacy hardware and software.

Systems employing trusted hardware have also been built, for example the Dyad HW architecture [185], the IBM 4758 [40, 81, 162] or the Cerium processor [29]. Jiang used a secure coprocessor to build an SSL co-server to process student passwords and grades [80]. These solutions are effective but add both financial and administrative costs to a system.

Flicker adds less than 250 lines of code to the TCB of a PAL, 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.

7.2 Attestation and Trusted Computing

Chapter 2 provided essential background information on attestation and trusted computing technology. Here, we discuss additional related work in trusted computing.

Ames [4], Gold et al. [56], and Tasker [169] were among the first to propose computer systems capable of cryptographically demonstrating their security properties to other systems. Today, this is called attestation.

Early schemes for attesting to a platform's software state include the entire software stack (e.g., BIOS, bootloader, OS, applications) [8, 106, 143]. Arbaugh et al. proposed secure boot, whereby each layer of the software stack checks that the integrity of the next layer matches a known-good configuration, otherwise boot is aborted [8]. This architecture does not allow a system to attest its configuration to an external party. Sailer et al. designed an integrity measurement architecture for Linux that implements trusted boot, whereby an external party can receive an attestation of all software that has been loaded since boot and make its own trust decision depending on the software configuration [143]. Unfortunately, the security of a newly executed piece of code depends on the security of all previously executed code in both of these systems. 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. Property-based attestation has been proposed [141] as a mechanism for providing meaningful attestations; unfortunately, evaluating software for the various properties of interest remains an open problem.

Terra [53] is an architecture for trusted computing that leverages a *Trusted VMM (TVMM)* to allow legacy applications to continue running in an "open-box" virtualized environment while high-security applications execute in a "closed-box" special-purpose VM. Terra provides an attestation service that allows applications running in a local VM to authenticate themselves to remote parties [96, 181]. However, Terra makes heavy use of certification from independent software vendors, which does not exist today. Further, Terra is implemented on an OS-hosted commercial VMM which suffers from an extremely large TCB.

The BIND system guarantees the safe execution of a small piece of code – BGP routing management is used as an example application – to an external party using attestation [158], but BIND relies on the security of a trusted kernel that was never implemented.

We have focused on late launch and associated trusted computing technologies such as the TPM. Seshadri et al. explore an alternate means for creating a dynamic root of trust at runtime, called Pioneer [150]. The Pioneer system provides code integrity guarantees to an external verifier. Pioneer is not a realistic alternative today as the verifier must possess intimate knowledge of the microarchitectural design of the challenged system's CPU and cannot tolerate Internet levels of network latency.

Kauer developed the Open Secure Loader (OSLO) [88], which employs *SKINIT* to eliminate the BIOS and bootloader from the TCB and establish a dynamic root of trust for trusted boot. OSLO consists of just over 1,000 lines of code, and is larger than Flicker because it executes at boot time and includes support for the Multiboot Specification [123]. OSLO also includes an implementation of SHA-1 to hash the OS kernel, whereas SHA-1 is optional with Flicker. OSLO served as a starting point for the development of our Flicker implementation. Trusted Boot[1] from Intel performs similarly for

[1] `http://sourceforge.net/projects/tboot`

Intel hardware. Garriss et al. employ the new *SKINIT* instruction to eliminate the BIOS and the bootloader from their attestations and TCB [54], but as suggested in the original design [61], after the *SKINIT*, they launch a standard OS or VMM. Thus, application security depends on these large layers of code.

7.3 Protecting User Input and Output

We review related work on protecting sensitive user I/O. Many of these works consider an untrusted system such as a kiosk in an Internet cafe or other public location, but the prevalence of malware today suggests that users must be wary of even their own machine. We discuss works leveraging a mobile device to improve I/O security (Section 7.3.1), and works considering a redesign of the graphical window manager (Section 7.3.2).

7.3.1 Mobile Devices

The most closely related work is our prior work called Bump[2] in the Ether (BitE) [112]. BitE circumvents the legacy input path by leveraging encryption by user input devices (e.g., an encrypting keyboard), just as Bumpy does. However, BitE retains the legacy OS and Window Manager in its TCB, is tailored to local applications, and performs attestations to its correct functioning based on a static root of trust. In contrast, Bumpy dramatically reduces the TCB for input by leveraging a dynamic root of trust for each input event, works for sensitive input to websites, and supports secure post-processing of sensitive input (e.g., password hashing).

Borders and Prakash propose a Trusted Input Proxy (TIP) as a module in a virtual machine architecture where users can indicate data as sensitive using a keyboard escape sequence [21]. Users are presented with a special dialog box where they can enter their sensitive data, after which it is injected into the SSL session by the TIP. Again, however, the TCB of TIP includes a VMM and OS, whereas Bumpy's TCB includes neither.

[2]We derive the name Bumpy from Bump in the Ether.

Garriss et al. employ the new *SKINIT* instruction to establish a dynamic root of trust to launch a virtual machine monitor (VMM) on unfamiliar public kiosk machines, thus eliminating the BIOS and the bootloader from their TCB [54, 55]. After using a mobile device to verify an attestation[3] that the VMM launched successfully, the user's personal virtual machine [25] can be resumed on the public machine. While compelling, application security depends on large layers of code. In Bumpy, the remote webserver verifies an attestation to the correct operation of the Flicker sessions handling user input. Bumpy is trivially extensible to allow the user's mobile device to verify attestations, and in fact we originally considered this architecture in BitE.

Balfanz and Felten explored the use of hand-held computing devices (e.g., PDAs) as smart-cards, and found some advantages because the user can interact directly with the hand-held for sensitive operations [9]. The authors generalized their work into a design paradigm they call *splitting trust*, where a smaller, trusted device performs security-sensitive operations and a large, powerful device performs other operations. Bumpy can be considered a system designed in accordance with the principles of splitting trust.

The Pebbles project attempts to let handhelds and PCs work together when both are available, as opposed to the conventional view that handhelds are used when PCs are unavailable [120]. Bumpy uses a trusted mobile device as a Trusted Monitor to help improve input security on a PC.

Ross et al. develop a framework for access to Internet services, where both the sensitivity of the information provided by the service and the capabilities of the client device are incorporated [139]. This framework depends on either a trusted proxy infrastructure or service providers running a trusted proxy. While promising, this scheme is not widely deployed today.

Oprea et al. consider the use of public terminals to access one's home PC. The public terminals are considered unsuitable for trusting with long-term secrets such as passwords [124]. A trusted mobile device (e.g., PDA) provides all input which is cryptographically tunneled to the user's home

[3]Background on the relevant trusted computing primitives is provided in Chapter 2.

PC, and the public terminal is given read-only access to serve as a display for only the applications that the user accesses on her home PC during that session. Further, these credentials are short-lived, expiring shortly after the user disconnects her mobile device from the public terminal. Bumpy differs in that it operates directly on the user's PC, is specific to providing user input to websites, and has a dramatically smaller TCB.

Sharp et al. develop a system for splitting input and output across an untrusted terminal and a trusted mobile device [155]. Applications run on a trusted server or on the mobile device itself, using VNC [135] to export video to the trusted and untrusted displays in accordance with a security policy. The user has the ability to decide on the security policy used for the untrusted keyboard, mouse, and display. An initial user study yielded encouraging results, but this technique is best described as a tool for power users. In contrast, Bumpy is designed for users who may have very little understanding of computer security.

In more recent work, Sharp et al. propose an architecture for fighting crimeware (e.g., keyloggers and screen scrapers) via split-trust web applications [156]. Web-applications are written to support an untrusted browser and a trusted mobile device with limited browsing capabilities. All security-critical decisions are confirmed on the mobile device. This architecture raises the bar for web-based attackers, but it also raises usability issues which are the subject of future work.

The Zone Trusted Information Channel (ZTIC [73]) is a recent device with a dedicated display and the ability to perform cryptographic operations. Its purpose is to confirm online banking transactions in isolation from malware on the user's computer. This device is appropriate for use as a Trusted Monitor in Bumpy.

Bumpy uses the separate Trusted Monitor as a verifier and indicator for the input framework, rather than as a platform for execution of portions of a split application or as an input device. But perhaps more importantly, the TCB of Bumpy is far smaller than in these other works, and in fact Bumpy can be viewed as extreme in this respect.

7.3.2 Secure Window Managers

Much prior work has addressed the issue of a security-conscious graphical window manager. Unfortunately, none of it is readily available for non-expert users on commodity systems today. We review related work chronologically.

Several government and military computer windowing systems have been developed with attention to security and the need to carefully isolate different grades of information (e.g., classified, secret, top secret). Early efforts to secure commercial window managers resulted in the development of *Compartmented Mode Workstations* [20, 27, 49, 130, 137, 182], where tasks with different security requirements are strictly isolated from each other. These works consider an operating environment where an employee has various tasks she needs to perform, and some of her tasks have security requirements that necessitate isolation from other tasks. For example, Picciotto et al. consider trusted cut-and-paste in the X window system [129]. Cut-and-paste is strictly confined to allow information flow from low-sensitivity to high-sensitivity applications, so that high-sensitivity information can never make its way into a low-sensitivity application. Epstein et al. performed significant work towards trusted X for military systems in the early 1990s [46, 47, 48]. While these systems are effective for employees trained in security-sensitive tasks, they are unsuitable for use by consumers.

Shapiro et al. propose the EROS Trusted Window System [154], which demonstrates that breaking an application into smaller components can greatly increase security while maintaining very powerful windowing functionality. Unfortunately, EROS is incompatible with a significant amount of legacy software, which hampers widespread adoption. In contrast, BitE works in concert with existing window managers.

Microsoft's Next-Generation Secure Computing Base (NGSCB) proposes encrypting keyboard and mouse input, and video output [42, 126]. In NGSCB, special USB keyboards encrypt keystrokes which pass through the regular operating system into the *Nexus*, where they are decrypted. Once in the Nexus, they can be sent to a trusted application running in Nexus-mode,

or they can be sent to the legacy OS. Applications running in Nexus-mode have the ability to take control of the system's primary display, which was designed to be useful for establishing a trusted tunnel.

A compelling recent example is Nitpicker [50], but it currently requires changing operating systems and porting existing legacy applications. Bumpy remains compatible with existing legacy operating systems, to the extent that they meet the requirements for Flicker (Chapter 3), i.e., it may be necessary to install a kernel module or driver.

Common to the majority of these schemes is a mechanism by which some portion of the computer's screen is trusted. That is, an area of the screen is controlled by some component of the trusted computing base (TCB) and is inaccessible to all user applications. However, if an application can use a "full-screen" mode, it may be able to spoof any trusted output. This is particularly relevant given the ubiquity of interactive and animated multimedia on today's web pages [2]. Precisely defining trusted full-screen semantics that a non-expert user can operate securely is, to the best of our knowledge, an unsolved problem. Considering the value that the user receives from being able to maximize applications, and the role of multi-media applications on today's commodity PCs, we believe the ability to run applications in full-screen mode on the system's primary display is an indispensable feature. Still, there is no effective way to establish a trusted tunnel if there is no trusted display. Due to the complexity of X and the likely confusion of untrained users, it is difficult to implement a trusted screen area in an assurable way. Bumpy uses the trusted mobile device's screen—a physically separate display—as a Trusted Monitor for output.

We emphasize that, despite the large body of work on trusted windowing systems, the majority of users do not employ any kind of trusted windowing system. Thus, we conclude that users do not want to change their windowing system.

7.4 Browser Security

Ross et al. developed PwdHash, an extension for the Firefox web browser that hashes users' typed passwords in combination with the domain serving the page to produce a unique password for every domain [138]. The PwdHash algorithm adapts earlier work by Gabber et al. on protecting users' privacy while browsing the web [52, 51]. Chiasson et al. identify usability problems with PwdHash, specifically, that it provides insufficient feedback to the user regarding the status of protections [32]. We extend this work in two ways in the context of Bumpy. First, we implement the PwdHash algorithm as one possible transformation of sensitive data in Bumpy, with a much smaller TCB than the web browser and OS that must be trusted with PwdHash. Second, we leverage a Trusted Monitor to provide feedback to the user regarding the status of her input. Validating the efficacy of our feedback mechanisms with a user study remains the subject of future work (discussed in Chapter 8.2).

7.5 Authentication without Prior Context

We study authentication between two co-located entities with no prior trust relationships. This context rules out the use of a public key infrastructure or trusted third party to perform authentication.

A common mechanism to establish a secure channel between two entities is to use Diffie-Hellman key establishment [38]. Unfortunately, a man-in-the-middle (MITM) attack is possible if the two entities do not share any established trusted information. Bellovin and Merrit propose the encrypted key exchange (EKE) protocol, which prevents the MITM attack if both parties share a secret password [18]. Several researchers have refined this approach [19, 22, 101, 183], but they all require a shared secret password between the two entities, which may be cumbersome to establish in many mobile settings.

Another approach to defeat the MITM attack is to use a secondary channel to verify that the same key is shared by two parties. An approach that

several researchers have considered is that a human can manually verify that the generated keys are identical [97, 174, 175]. Uzun et al. found usability issues with general classes of string comparison-based protocols [173]. To avoid manual comparison, researchers have devised visual metaphors that represent the hash of a key to make it easier for people to perform the comparison [58, 98, 127, 39]. Though these schemes make key comparison easier for the user, they still rely on the user to diligently compare the resulting visual key representations. With SiB, visual device identification is an integral part of establishing a connection between devices.

To defend against MITM attacks, Stajano and Anderson propose to set up keys through a link that is created through physical contact [166]. However, in many settings, devices may not have interfaces that connect for this purpose, or they may be too bulky to carry around. Balfanz et al. extend this approach to use short-range wireless infrared communication [10]. Of all these approaches, theirs is the most closely related to SiB, and we discuss it further in Section 5.1.2. Čapkun, Hubaux, and Buttyán have further extended this research direction [176]. They make use of one-hop transitive trust to enable two nodes that have never met to establish a key. SiB could leverage this technique equally well. Hoepman gives a more rigorous definition of the ephemeral pairing problem and presents ephemeral key exchange protocols for authentic channels and private channels [70].

Clarke et al. propose protocols for camera-based authentication of the screen content of public computers [34]. A camera-equipped, trusted, mobile device monitors the screen of the public computer for the duration of a transaction, verifying the presence of a nonce, a one-time password, and a MAC. The mobile device connects to the user's trusted "proxy," which is their home computer, to verify that the screen contents have not been modified. Image processing and optical character recognition are both considered as viable mechanisms for processing the screen. SiB can be used to implement this protocol, with a 2D barcode serving as the relevant image.

Following the initial publication of SiB [110, 111], Saxena et al. further explored the visual channel [146]. They consider the minimal device capabilities that can support SiB, and devise a video codec that can use a

mobile phone's camera to decode data encoded in a severely constrained visual channel – in the limit a single flashing LED. This scheme is valuable in low-cost scenarios where the only output mechanism available may be an LED, though it requires a higher level of understanding from the user.

Also following our work, Goodrich et al. developed Loud-and-Clear, a system that uses an audio channel to establish authentic keys [60]. In Loud-and-Clear, English phrases are derived from the hash of a device's public key. One device uses a text-to-speech engine to read a phrase aloud, while the other device displays a phrase on-screen. The human user is tasked with listening to one phrase and comparing it with the written phrase. HAPADEP extends this mechanism to pure audio device pairing [164].

In some cases, the visual channel bandwidth available between two devices may be insufficient for standard cryptographic techniques. For example, a single barcode in our implementation has a data payload of only 68 bits. To address issues with low-bandwidth channels, Laur et al. propose protocols based on *Manually Authenticated Strings* (MANA) conveyed across an out-of-band channel that may have low bandwidth [97]. In their case, the low bandwidth channel is that of humans performing a manual comparison. MANA IV requires users to visually compare short ℓ-bit strings displayed on their devices and push a button on each device to indicate whether the strings match, where ℓ is presumed to be shorter than the output of a cryptographic hash function. Given SHA-1 as an acceptable hash function, $\ell < 160$.

SiB can be modified to use MANA-IV as the commitment protocol when the only available visual channel is exceptionally low-bandwidth. The visual channel is used to convey the ℓ-bit strings between devices, where they can be programmatically compared. This design is compelling because it removes the users' responsibility to carefully compare the ℓ-bit strings, thereby substantially reducing the opportunity for human error [173]. However, MANA IV requires the devices to exchange three messages before the visual channel exchange takes place. This necessitates an out-of-band mechanism for discovering the network identity of the other device. Traditional Bluetooth discovery mechanisms are one option in this scenario.

7.5.1 Barcode Recognition with Camera Phones

SiB depends on a camera phone having the ability to use its camera to recognize two-dimensional (2D) barcodes. Several projects exist that seek to allow camera-equipped mobile phones to interact with physical objects through the use of 2D barcodes. Rohs and Gfeller develop their own 2D code explicitly for use with mobile phones, emphasizing their ability to be read from electronic screens and printed paper [136]. Woodside develops *semacodes*, [4] which is an implementation of the Data Matrix barcode standard for mobile phones [77]. Woodside considers the primary application of semacodes as containers for a URL which contains information about the physical location where the barcode was installed. Madhavapeddy et al. use SpotCodes to enhance human-computer interaction by using a camera-phone as a pointing and selection device [102]. Researchers working on the CoolTown[5] project at HP Labs propose tagging electronics around the house with barcodes to be read by camera phones or PDAs so that additional data about the tagged device can be easily retrieved.

Hanna considers devices with barcodes affixed to aid in the establishment of security parameters [64]. His work considers a smart home, where a user may want to establish a security context for controlling appliances or other devices in a smart-home. In Hanna's work, the barcode contains a secret which is also stored inside the device. Hanna proposes using this secret to enable the secure transmission of commands to the device from a master controller over an untrusted network. We refer to the security property discussed by Hanna as *presence*, where it is desirable that only users or devices close to some device are able to control it. We discuss the notion of *presence* further in Section 5.4.

Today, recognition of 2D barcodes with mobile phones has become accepted practice. Phones are now available that include barcode recognition software, such as the Nokia N95.[6] Further, a Java standard has been published that specifies an API for barcode recognition on mobile phones [84].

[4] `http://www.semacode.com/`
[5] `http://www.cooltown.com/`
[6] `http://mobilecodes.nokia.com/`

Chapter 8

Conclusions and Future Work

We state the conclusions of this thesis before discussing opportunities for further investigation.

8.1 Conclusions

Networked computer systems have grown in complexity to a level we can no longer control. To date, formal methods have not reached a level of maturity or scalability to prove these systems correct [44, 45, 177]. Consequently, we use TCB minimization to approach correctness for security-sensitive parts of applications [157], observing that for many applications the security-sensitive operations are relatively small. We have developed a method for isolated code execution on commodity systems with a TCB that adds as few as 250 lines of code to application-specific functionality. TPM-based attestation can be used to convince a verifier, which may be local or remote, that the application-specific code did execute with the desired protections. Our work finally gives application developers the opportunity to write secure applications without relying on the security of layer upon layer of legacy software, and without breaking compatibility with today's commodity systems.

We have extended our system for isolated code execution to minimize the TCB for sensitive user input to web pages. We allow the webserver to control the processing of user input intended for that webserver. While we require the user to be aware of our system, and to demonstrate some diligence in its operation, we are optimistic about its ability to defeat software keyloggers and screen scrapers in the wild. The use of our system can be attested to both a local verifier and a remote webserver, giving the user and/or webserver (and hence the organization that operates it, e.g., a bank) the ability to determine with higher assurance than can be obtained today that the user's inputs were protected.

We have shown that trust relationships predicated on authentic public key exchange can be established without depending on a trusted third party or public key infrastructure, further reducing the amount of trust that must be placed in entities beyond the user's control. This gives the conscientious user the opportunity to configure her own input and verification devices, and the savvy user the ability to retain total control over her own privacy-relevant inputs.

Perhaps most encouraging is the demonstration that so-called *trusted computing* technologies do have uses that can enhance users' security and privacy. The inclusion of TPMs in commodity systems was a significant gamble by platform manufacturers, and the push-back against a perceived utility as nothing more than a DRM component threatens to kill the trusted computing movement. We hope that our work will mature into a widely deployed service available on commodity systems with trusted computing technology, and that it will make a difference in the security of ordinary users' everyday computing experience.

8.2 Opportunities for Further Investigation

The technologies that have come to be known as *Trusted Computing*, such as remote attestation and the creation of a dynamic root of trust, are still in their infancy. While this thesis explores some uses of this technology, many issues exist that warrant further exploration. We consider three in partic-

ular: usability, automatic privilege separation, and user-observable verification. An extended discussion of open questions related to user-observable verification of attestations has been previously published [115].

8.2.1 User Studies

We design the Bumpy and SiB systems in this thesis with an emphasis on usability, but we have not performed a formal user study. A user study is needed to evaluate and refine the mechanisms through which users interact with Bumpy in particular, and remote attestation in general.

8.2.2 Automatic Privilege Separation

Flicker dramatically reduces the TCB for sensitive code, but modifying existing applications to work with Flicker is largely a manual process. Researchers have considered manual privilege separation in other domains [90, 133, 168, 106, 81]. However, security gains made possible by Flicker can be realized more rapidly if existing applications can be automatically separated [24, 11, 186] such that the privileged components run with Flicker's protections.

8.2.3 User-Observable Verification

In order to use their computing devices with confidence, users need to know if the software on their computing devices is infected by malware. Attestation enables a *verifier* to learn the configuration of the software on any TCG-compliant computing device, provided that infrastructure such as Flicker (Chapter 3) or IBM's Integrity Measurement Architecture [143] is in place to measure loaded software (Chapter 2). The verifier can compare this configuration to a known-good configuration to detect deviations.

Unfortunately, a successful verification of one device by another does not directly translate into a *user-observable verification*, potentially failing to give the user assurance that her devices are working as intended. The user who seeks a trustworthy system from which to verify others quickly discovers an endless loop of trust dependencies. Though we have shown

how to dramatically reduce the TCB for sensitive operations, it is an open research question to provide the user with a device that is axiomatically trustworthy, thereby breaking the dependency loop. Further, it is not clear whether a user-friendly recovery mechanism for a failed verification can ever be devised.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer USENIX Conference*, pages 93–112, June 1986. 169

[2] Adobe Systems Incorporated. Adobe flash player 9 security. White Paper, July 2008. 178

[3] Advanced Micro Devices. AMD64 architecture programmer's manual: Volume 2: System programming. AMD Publication no. 24594 rev. 3.11, December 2005. 19, 24, 25

[4] S. R. Ames, Jr. Security kernels: A solution or a problem? In *Proceedings of the IEEE Symposium on Security and Privacy*, April 1981. 172

[5] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the Workshop on Grid Computing*, November 2004. 55

[6] 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. 54

[7] W. A. Arbaugh. Improving the tcpa specification. *IEEE Computer*, 35(8):77–79, 2002. 18

[8] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Security and*

Privacy, May 1997. 31, 172

[9] D. Balfanz and E. W. Felten. Hand-held computers can be better smart cards. In *Proccedings of the USENIX Security Symposium*, August 1999. 119, 175

[10] D. Balfanz, D. Smetters, P. Stewart, and H. C. Wong. Talking to strangers: Authentication in ad-hoc wireless networks. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, February 2002. 89, 120, 122, 124, 125, 180

[11] D. Balfanz. *Access Control for Ad-hoc Collaboration*. PhD thesis, Princeton University, 2001. 52, 185

[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003. 35, 170, 171

[13] P. R. Barham, B. Dragovic, K. A. Fraser, S. M. Hand, T. L. Harris, A. C. Ho, E. Kotsovinos, A. V. Madhavapeddy, R. Neugebauer, I. A. Pratt, and A. K. Warfield. Xen 2002. Technical Report UCAM-CL-TR-553, University of Cambridge, January 2003. 30

[14] L. Bauer, S. Garriss, J. M. McCune, M. K. Reiter, J. Rouse, and P. Rutenbar. Device-enabled authorization in the Grey system. In *Proceedings of the 8th Information Security Conference (ISC)*, September 2005. 139

[15] BeagleBoard.org. BeagleBoard revision B6 system reference manual revision 0.1. BeagleBoard.org, November 2008. 104

[16] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical report, MITRE MTR-2997, March 1976. 18

[17] M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval. Key-privacy in public-key encryption. In *Proceedings of Advances in Cryptology (ASIACRYPT)*, 2001. 129

[18] S. Bellovin and M. Merrit. Augmented encrypted key exchange: a password-based protocol secure against dictionary atttacks and password file compromise. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 244–250, 1993. 143, 179

[19] S. M. Bellovin and M. Merrit. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 72–84, 1992. 179

[20] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings. Compartmented mode workstation: Prototype highlights. *Software Engineering*, 16(6):608–618, June 1990. 177

[21] K. Borders and A. Prakash. Securing network input via a trusted input proxy. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)*, August 2007. 174

[22] V. Boyko, P. MacKenzie, and S. Patel. Provably secure password authentication and key exchange using Diffie-Hellman. In *Proceedings of Advances in Cryptology (EUROCRYPT)*, pages 156–171, 2000. 179

[23] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proceedings of the ACM Conference on Computer and Communications Security (CSS)*, October 2004. 28

[24] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the USENIX Security Symposium*, August 2004. 52, 185

[25] R. Cáceres, C. Carter, C. Narayanaswami, and M. Raghunath. Reincarnating pcs with portable soulpads. In *Proceedings of the conference on Mobile systems, applications, and services (MobiSys)*, pages 65–78, June 2005. 175

[26] J. Camenisch. Better privacy for trusted computing platforms. In *Proceedings of the European Symposium On Research in Computer Security (ESORICS)*, 2004. 28

[27] M. Carson and J. Cugini. An X11-based Multilevel Window System architecture. In *Proceedings of the EUUG Technical Conference*, 1990. 177

[28] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6), 2004. 31

[29] B. Chen and R. Morris. Certifying program execution with secure procesors. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2003. 172

[30] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, P. C. Mew, and W. Mao. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. Technical Report FDUPPITR-2007-0801, Parallel Processing Institute, Fudan University, August 2007. 170

[31] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 2–13, March 2008. 170

[32] S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *Proceedings of the USENIX Security Symposium*, August 2006. 80, 96, 179

[33] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, August 2004. 20

[34] D. E. Clarke, B. Gassend, T. Kotwal, M. Burnside, M. van Dijk, S. Devadas, and R. L. Rivest. The untrusted computer problem and camera-based authentication. In *Proceedings of the International Conference on Pervasive Computing*, pages 114–124, 2002. 180

[35] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM*

Journal of Research and Development, 25(5):483–490, September 1981. 169

[36] F. Dawson and T. Howes. vCard MIME directory profile. RFC 2426, September 1998. 128

[37] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol: Version 1.1. RFC 4346, April 2006. 128

[38] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, November 1976. 127, 128, 135, 179

[39] S. Dohrmann and C. Ellison. Public key support for collaborative groups. In *Proceedings of the PKI Research Workshop*, April 2002. 143, 180

[40] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, 2001. 172

[41] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 17–30, 2005. 170

[42] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *IEEE Computer*, 36(7):55–62, July 2003. 103, 177

[43] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, December 1995. 170

[44] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Symposium on Operating System Design and Im-*

plementation (OSDI), 2000. 183

[45] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001. 183

[46] J. Epstein. A prototype for Trusted X labeling policies. In *Proceedings of the Sixth Annual Computer Security Applications Conference (ACSAC)*, December 1990. 177

[47] J. Epstein and J. Picciotto. Issues in building Trusted X Window Systems. *The X Resource*, 1(1), Fall 1991. 177

[48] J. Epstein and J. Picciotto. Trusting X: Issues in building Trusted X window systems -or- what's not trusted about X? In *Proceedings of the Annual National Computer Security Conference*, October 1991. 177

[49] G. Faden. Reconciling CMW requirements with those of X11 applications. In *Proceedings of the Annual National Computer Security Conference*, October 1991. 177

[50] N. Feske and C. Helmuth. A nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 85–94, 2005. 178

[51] E. Gabber, P. Gibbons, Y. Matias, and A. Mayer. How to make personalized web browsing simple, secure, and anonymous. In *Proceedings of Financial Cryptography*, 1997. 21, 77, 92, 179

[52] E. Gabber, P. B. Gibbons, D. M. Kristol, Y. Matias, and A. Mayer. On secure and pseudonymous client-relationships with multiple servers. *ACM Transactions on Information and System Security*, 2(4):390–415, 1999. 21, 77, 92, 179

[53] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*,

October 2003. 173

[54] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Towards trustworthy kiosk computing. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, February 2006. 174, 175

[55] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Trustworthy and personalized computing on public kiosks. In *Proceeding of the Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2008. 175

[56] B. Gold, R. Linde, and P. Cudney. KVM/370 in retrospect. In *Proceedings of the IEEE Symposium on Security and Privacy*, April 1984. 169, 172

[57] B. Gold, R. Linde, R. J. Peller, M. Schaefer, J. Scheid, and P. D. Ward. A security retrofit for VM/370. *AFIPS National Computing Conference*, 48:335–344, June 1979. 169

[58] I. Goldberg. Visual key fingerprint code. `http://www.cs.berkeley.edu/iang/visprint.c`, 1996. 143, 180

[59] K. Goldman, R. Perez, and R. Sailer. Linking remote attestation to secure tunnel endpoints. Technical Report RC23982, IBM, 2006. 46

[60] M. T. Goodrich, M. Sirivianos, J. Solis, G. Tsudik, and E. Uzun. Loud and clear: Human-verifiable authentication based on audio. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2006. 181

[61] D. Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006. 19, 35, 73, 174

[62] J. C. Haartsen. The Bluetooth radio system. *IEEE Personal Communications Magazine*, pages 28–36, 2000. 134

[63] S. Halevi and H. Krawczyk. Public-key cryptography and password protocols. *ACM Transactions on Information and System Security*,

2(3), 1999. 46

[64] S. R. Hanna. Configuring security parameters in small devices. `draft-hanna-zeroconf-seccfg-00.txt`, July 2002. 182

[65] D. Harkins and D. Carrel. The Internet key exchange (IKE). RFC 2409, November 1998. 128

[66] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of microkernel-based systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 1997. 18

[67] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1997. 171

[68] K. Hemenway and T. Calishain. *Spidering Hacks*. O'Reilly, October 2003. 17

[69] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification. Revision 3.0b, October 2006. 69

[70] J.-H. Hoepman. The ephemeral pairing problem. In *Proceedings of Financial Cryptography*, 2004. 180

[71] T. Howes and M. Smith. MIME content-type for directory information. RFC 2425, September 1998. 128

[72] G. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. D. Zill. An overview of the singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, October 2005. 170

[73] IBM Zurich Research Lab. Security on a stick. Press release, October 2008. 176

[74] Intel Corporation. Intel low pin count (LPC) interface specification. Revision 1.1, August 2002. 71

[75] Intel Corporation. Trusted eXecution Technology – preliminary architecture specification and enabling considerations. Document number 31516803, November 2006. 19, 24, 26

[76] Intel Corporation. Intel virtualization technology for directed I/O. Intel Publication no. D51397-004 rev. 1.2, September 2008. 26

[77] ISO/IEC. IS 16022:2006: Information technology — automatic identification and data capture techniques — Data Matrix bar code symbology specification. For review, International Organization for Standardization, Geneva, Switzerland., 2006. 124, 182

[78] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the USENIX Security Symposium*, pages 59–74, August 2003. 171

[79] M. Jakobsson and S. Myers. *Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft*. Wiley, December 2006. 101

[80] S. Jiang. WebALPS implementation and performance analysis. Master's thesis, Dartmouth College, 2001. 172

[81] S. Jiang, S. Smith, and K. Minami. Securing web servers against insider attack. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2001. 52, 172, 185

[82] P. Jones. RFC3174: US Secure Hash Algorithm 1 (SHA-1). `http://www.faqs.org/rfcs/rfc3174.html`, September 2001. 134

[83] J. Jonsson and B. Kaliski. PKCS #1: RSA cryptography specifications version 2.1. RFC 3447, February 2003. 111

[84] JSR-257. JSR-257: Contactless communication API. Java Community Process, October 2006. 134, 139, 143, 182

[85] B. Kaliski and J. Staddon. PKCS #1: RSA cryptography specifications. RFC 2437, 1998. 58

[86] 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, November 1991. 18

[87] P. Karn. Reed-solomon encoding/decoding. `http://www.ka9q.net/code/fec/`, 2002. 134

[88] B. Kauer. OSLO: Improving the security of trusted computing. In *Proceedings of the USENIX Security Symposium*, August 2007. 24, 25, 173

[89] T. Kidder. *The Soul of a New Machine*. Atlantic-Little, Brown, August 1981. 18

[90] D. Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference*, 2003. 52, 185

[91] M. G. Kuhn. Optical time-domain eavesdropping risks of CRT displays. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2002. 144

[92] M. G. Kuhn and R. J. Anderson. Soft tempest: Hidden data transmission using electromagnetic emanations. In *Proceedings of the Information Hiding Workshop (IHW)*, pages 124–142, April 1998. 144

[93] C. Kuo. *Reduction of End User Errors in the Design of Scalable, Secure Communication*. PhD thesis, Carnegie Mellon University, 2008. 17

[94] K. Kursawe, D. Schellekens, and B. Preneel. Analyzing trusted platform communication. In *Proceedings of the Cryptographic Advances in Secure Hardware Workshop (CRASH)*, September 2005. 24

[95] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the Symposium on Computer Architecture*, April 1994. 152

[96] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems (TOCS)*, 10(4):265–310, 1992. 173

[97] S. Laur, N. Asokan, and K. Nyberg. Efficient mutual data authentication using manually authenticated strings. Cryptology ePrint Archive, Report 2005/424, 2005. 180, 181

[98] R. Levien. PGP snowflake. Personal communication, 1996. 143, 180

[99] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems*, 2000. 147, 171

[100] J. Liedtke. On micro-kernel construction. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–250, 1995. 171

[101] P. MacKenzie, S. Patel, and R. Swaminathan. Password authenticated key exchange based on RSA. In *Proceedings of Advances in Cryptology (ASIACRYPT)*, pages 599–613, 2000. 179

[102] A. Madhavapeddy, D. Scott, R. Sharp, and E. Upton. Using camera-phones to enhance human-computer interaction. In *Proceedings of Ubiquitous Computing (Adjunct Proceedings: Demos)*, 2004. 182

[103] A. Madhavapeddy, D. Scott, R. Sharp, and E. Upton. Using visual tags to bypass Bluetooth device discovery. In *Proceedings of the ACM Mobile Computing and Communications Review (MC2R)*, January 2005. 129

[104] D. Magenheimer. Xen/IA64 code size stats. Xen developer's mailing list: `http://lists.xensource.com/`, September 2005. 30, 171

[105] V. Maraia. *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison Wesley, first edition, September 2005. 30

[106] J. Marchesini, S. W. Smith, O. Wild, J. Stabiner, and A. Barsamian. Open-source applications of TCPA hardware. In *the IEEE Computer Security Applications Conference*, 2004. 31, 171, 172, 185

[107] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki.

Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, April 2008. 19

[108] 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

[109] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. How low can you go? Recommendations for hardware-supported minimal TCB code execution. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008. 19

[110] J. M. McCune, A. Perrig, and M. K. Reiter. Seeing-is-believing: Using camera phones for human-verifiable authentication. Technical Report CMU-CS-04-174, Carnegie Mellon University, November 2004. 133, 138, 180

[111] J. M. McCune, A. Perrig, and M. K. Reiter. Seeing-is-believing: Using camera phones for human-verifiable authentication. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005. 19, 133, 180

[112] J. M. McCune, A. Perrig, and M. K. Reiter. Bump in the Ether: A framework for securing sensitive user input. In *Proceedings of USENIX Annual Technical Conference*, June 2006. 19, 117, 174

[113] J. M. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, February 2009. 19

[114] J. M. McCune, A. Perrig, and M. K. Reiter. Seeing-is-believing: Using camera phones for human-verifiable authentication. *International Journal of Security and Networks Special Issue on Secure Spontaneous Interaction*, 4(1), 2009. 19

[115] J. M. McCune, A. Perrig, A. Seshadri, and L. van Doorn. Turtles all the way down: Research challenges in user-based attestation. In

Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec), August 2007. 19, 185

[116] R. Meushaw and D. Simard. NetTop: Commercial technology in high assurance applications. *VMware Tech Trend Notes*, 9(4):1–8, 2000. 169

[117] S. C. Misra and V. C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *Proceedings of the Conference on Computational Science and Its Applications*, January 2003. 18

[118] D. Molnar. The SETI@Home problem. *ACM Crossroads*, 7.1, 2000. 55

[119] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A crawler-based study of spyware on the web. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, February 2006. 17

[120] B. A. Myers. Using handhelds and PCs together. *Communications of the ACM*, 44(11), November 2001. 175

[121] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998. 50

[122] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the Conference on Compilier Construction*, 2002. 52

[123] Y. K. Okuji, B. Ford, E. S. Boleyn, and K. Ishiguro. The multiboot specification. Version 0.6.95, 2006. 173

[124] A. Oprea, D. Balfanz, G. Durfee, and D. K. Smetters. Securing a remote terminal application with a mobile trusted device. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 438–447, 2004. 175

[125] B. Parno. Bootstrapping trust in a "trusted" platform. In *Proceedings*

of the USENIX Workshop on Hot Topics in Security (HotSec), July 2008. 89

[126] M. Peinado, Y. Chen, P. England, and J. Manferdelli. NGSCB: A trusted open system. In *Proceedings of the Australasian Conference on Information Security and Privacy (ACISP)*, July 2004. 103, 177

[127] A. Perrig and D. Song. Hash visualization: A new technique to improve real-world security. In *Proceedings of the Workshop on Cryptographic Techniques and E-Commerce (CrypTEC)*, pages 131–138, July 1999. 143, 180

[128] B. Pfitzmann, J. Riordan, C. Stüble, M. Waidner, and A. Weber. The PERSEUS system architecture. In *Proceedings of Verlässliche IT-Systeme (Dependable IT Systems)*, pages 1–17, 2001. 171

[129] J. Picciotto. Towards trusted cut and paste in the X Window System. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 1991. 177

[130] J. Picciotto and J. Epstein. A comparison of Trusted X security policies, architectures, and interoperability. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 1992. 177

[131] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17, July 1974. 169

[132] Prolific Technology Inc. PL-25A1 hi-speed USB host to host bridge controller. PL-25A1 Product Brochure, October 2006. 104

[133] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the USENIX Security Symposium*, August 2003. 52, 185

[134] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 1960. 134

[135] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998. 176

[136] M. Rohs and B. Gfeller. Using camera-equipped mobile phones for interacting with real-world objects. *Proceedings of Advances in Pervasive Computing*, pages 265–271, April 2004. 133, 135, 182

[137] D. S. H. Rosenthal. LInX—a Less INsecure X server (Sun Microsystems unpublished draft). 177

[138] 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*, August 2005. 21, 56, 77, 81, 91, 92, 96, 103, 104, 111, 179

[139] S. J. Ross, J. L. Hill, M. Y. Chen, A. D. Joseph, D. E. Culler, and E. A. Brewer. A composable framework for secure multi-modal access to Internet services from post-PC devices. *Mobile Network Applications*, 7(5):389–406, 2002. 175

[140] A.-R. Sadeghi, M. Selhorst, C. Stüble, C. Wachsmann, and M. Winandy. TCG inside? - A note on TPM specification compliance. In *Proceedings of the ACM Workshop on Scalable Trusted Computing (STC)*, 2006. 24

[141] A.-R. Sadeghi and C. Stueble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Proceedings of the Workshop on New Security Paradigms (NSPW)*, 2004. 173

[142] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype: Secure hypervisor approach to trusted virtualized systems. Technical Report RC23511, IBM Research, February 2005. 171

[143] 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. 18, 24, 26, 31,

172, 185

[144] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975. 77

[145] S. Saroiu, S. D. Gribble, and H. M. Levy. Measurement and analysis of spyware in a university environment. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004. 17

[146] N. Saxena, J.-E. Ekberg, K. Kostiainen, and N. Asokan. Secure device pairing based on a visual channel (short paper). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006. 133, 140, 180

[147] M. Schaefer and B. Gold. Program confinement in KVM/370. In *Proceedings of the Annual ACM Conference*, pages 404–410, October 1977. 169

[148] W. L. Schiller. Design of a security kernel for the PDP-11/45. Technical Report ESD-TR-73-294, MTR-2709, The MITRE Corporation, HQ Electronic Systems Division: Hanscom AFB, December 1973. 18

[149] W. L. Schiller. The design and specification of a security kernel for the PDP-11/45. Technical Report ESD-TR-75-69, The MITRE Corporation, HQ Electronic Systems Division, Hanscom AFB, MA, May 1975. Available at: `http://csrc.nist.gov/publications/history/schi75.pdf`. 18

[150] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2005. 173

[151] Y. Shaked and A. Wool. Cracking the Bluetooth PIN. In *Proceedings of the Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2005. 117

[152] T. Shanley. *The Unabridged Pentium 4*. Addison Wesley, first edition,

August 2004. 153

[153] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1999. 18, 170

[154] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *Proceedings of the USENIX Security Symposium*, 2004. 177

[155] R. Sharp, J. Scott, and A. Beresford. Secure mobile computing via public terminals. In *Proceedings of the International Conference on Pervasive Computing*, May 2006. 176

[156] R. Sharp, A. Madhavapeddy, R. Want, and T. Pering. Enhancing web browsing security on public terminals using mobile composition. In *Proceeding of the Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2008. 176

[157] V. Y. Shen, T.-J. Yu, S. M. Thebaut, and L. R. Paulsen. Identifying error-prone software an empirical study. *IEEE Transactions on Software Engineering*, 11(4):317–324, 1985. 183

[158] 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. 173

[159] A. Shieh, D. Williams, E. G. Sirer, and F. B. Schneider. Nexus: A new operating system for trustworthy computing. In *WIP Session at the ACM Symposium on Operating Systems Principles (SOSP)*, October 2005. 18

[160] L. Singaravelu, C. Pu, H. Haertig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, 2006. 171

[161] S. Smalley and P. Loscocco. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the*

FREENIX Track: USENIX Annual Technical Conference, 2001. 171

[162] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8), April 1999. 172

[163] D. X. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on SSH. In *Proceedings of the USENIX Security Symposium*, August 2001. 119

[164] C. Soriente, G. Tsudik, and E. Uzun. HAPADEP: Human-assisted pure audio device pairing. In *Proceedings of the International Information Security Conference (ISC)*, September 2008. 181

[165] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The flask security architecture: System support for diverse security policies. In *Proceedings of the USENIX Security Symposium*, 1999. 171

[166] F. Stajano and R. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Proceedings of the Security Protocols Workshop*, 1999. 89, 120, 124, 143, 180

[167] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the International Conference on Supercomputing*, 2003. 147, 171

[168] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, November 2006. 52, 170, 185

[169] P. S. Tasker. Trusted computer systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, April 1981. 172

[170] W. S. L. R. Team. First half 2005 security trends report. Websense Security Labs, 2005. 17

[171] Trusted Computing Group. PC client specific TPM interface specifi-

cation (TIS). Version 1.2, Revision 1.00, July 2005. 24, 71

[172] Trusted Computing Group. Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands, July 2007. Version 1.2, Revision 103. 18, 20, 23, 28, 34, 43, 52, 90, 107, 109, 121, 130, 138

[173] E. Uzun, K. Karvonen, and N. Asokan. Usability analysis of secure pairing methods. In *Proceedings of the Usable Security Workshop*, February 2007. 180, 181

[174] S. Vaudenay. Secure communications over insecure channels based on short authenticated strings. In *Advances in Cryptology (CRYPTO)*, volume 3621. Lecture Notes in Computer Science, 2005. 180

[175] M. Čagalj, S. Čapkun, and J.-P. Hubaux. Key agreement in peer-to-peer wireless networks. In *Proceedings of the IEEE (Special Issue on Cryptography and Security), volume 94*, pages 467–478, 2006. 180

[176] S. Čapkun, J. Hubaux, and L. Buttyán. Mobility helps security in ad hoc networks. In *Proceedings of the ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, June 2003. 180

[177] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 3–17, February 2000. 183

[178] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *Proceedings of the USENIX Annual Technical Conference*, June 2008. 108

[179] D. A. Wheeler. Linux kernel 2.6: It's worth more! Available at: `http://www.dwheeler.com/essays/linux-kernel-cost.html`, October 2004. 30, 113

[180] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the Symposium on Op-*

erating System Design and Implementation (OSDI), December 2002. 170

[181] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(1):3–32, 1994. 173

[182] J. P. L. Woodward. Security requirements for system high and compartmented mode workstations. Technical Report MTR 9992, Rev. 1, The MITRE Corporation, November 1987. 177

[183] T. Wu. The secure remote password protocol. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 1999. 179

[184] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 71–80, 2008. 170

[185] B. S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994. 172

[186] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Secure program partitioning. *ACM Transactions on Computer Systems (TOCS)*, 20(3), August 2002. 52, 185

[187] L. Zhuang, F. Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2005. 79