Minimal TCB Code Execution (Extended Abstract)*

Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri
Carnegie Mellon University

Abstract

We propose an architecture that allows code to execute in complete isolation from other software while trusting only a tiny software base that is orders of magnitude smaller than even minimalist virtual machine monitors. Our technique also enables more meaningful attestation than previous proposals, since only measurements of the security-sensitive portions of an application need to be included. We achieve these guarantees by leveraging hardware support provided by commodity processors from AMD and Intel that are shipping today.

1 Introduction

The large size and huge complexity of modern operating systems makes them difficult to analyze and vulnerable to attack. The Linux kernel currently (as of version 2.6) consists of nearly 5 million lines of code [23], while Microsoft's Windows Vista includes over 50 million lines of code. 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 [4] and within a few years almost doubled to approximately 83K lines [13]. Application code depends on all of this code for its security, thus swelling the size of its Trusted Computing Base (TCB) far beyond the application code itself. As a result, even security-conscious application developers can make few guarantees about the security their applications provide. As we discuss in more detail in Section 6, such a large TCB also prevents current proposals for system-wide code attestation [2, 14, 18] from providing meaningful security information.

*This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and grants CT-0433540 and CCF-0424422 from the National Science Foundation, by the iCAST project, National Science Council, Taiwan under the Grants No. (NSC95-main) and No. (NSC95-org), and by a gift from AMD. Bryan Parno is supported in part by an NDSEG Fellowship, which is sponsored by the Department of Defense. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AMD, ARO, CMU, NSF, or the U.S. Government or any of its agencies.

In this work, we describe a Secure Execution Architecture (SEA) that allows security-sensitive code to execute in complete isolation from all other software (including the operating system and VMM, if present) and hardware devices. This dramatic reduction in the size of the TCB for an application (see Figure 1) enables meaningful software attestation and facilitates formal security analysis of the software remaining in the TCB. Our SEA provides these guarantees without requiring a reboot.

Our architecture leverages hardware support for secure virtualization provided by AMD's Secure Virtual Machine (SVM) architecture [1] or Intel's Trusted Execution Technology (TXT) [10]. 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 (SK) without requiring a reboot [1, 9]. 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.

Many security-sensitive applications can benefit from our architecture. For example, when users log in to a server using SSH, their passwords are transmitted to the server over an encrypted and authenticated channel. At the server, however, the passwords are decrypted and exist "in the clear" in the SSH server's memory, where a malicious root user, OS, kernel module, or device can readily obtain them. By encrypting the user's password so that only code executed with our SEA can decrypt it, the user can safely transmit a password to the server without worrying about the security of the server's operating system or any other software the server might be executing. The server can use attestation to convince the client system (and hence, the user) that these additional protections are in place without revealing any additional information about the configuration of the server itself.

Likewise, a server could use SEA to improve the security of its SSL keys. Our architecture can also help secure e-commerce applications, e-voting, online auctions, or medical databases, for example. It is particularly well-suited to handling sensitive data such as cryptographic keys.

0-7695-2848-1/07 $25.00 © 2007 IEEE

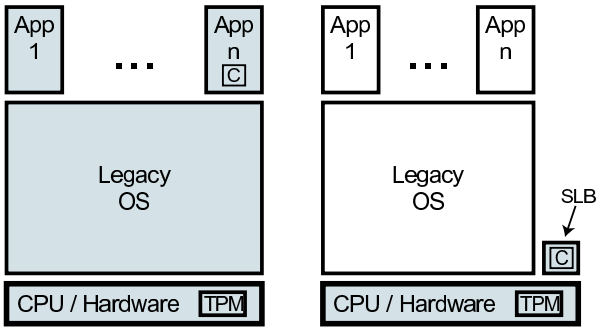

Figure 1. *The figure on the left illustrates a traditional architecture, with an application that executes a segment of sensitive code (C). The figure on the right demonstrates the use of our architecture to execute the sensitive code. The shaded portions represent the components that must be trusted in each scenario. The secure loader block (SLB) consists of the security-sensitive code plus a tiny amount of shim code.*

2 Background

Our architecture requires hardware security features which have recently come to market. We provide a brief introduction to the relevant hardware features.

2.1 Late Launch

Processors with AMD's SVM or Intel's TXT include the ability to use a *late launch* command to create a dynamic root of trust. In this work, we will focus on AMD's SVM technology, but Intel's TXT-enabled processors behave similarly. SVM includes a processor instruction, *SKINIT*, that takes a *secure loader block* (SLB) defined by a physical start address and a length up to 64KB as an argument. The *SKINIT* instruction enables various hardware-level protections for the SLB, transmits a copy of the SLB to the system's TPM so that it can be measured (hashed) into PCR 17, and then begins to execute the SLB [1]. The hardware protections disable interrupts, prevent DMA access to the SLB, and even prohibit access by hardware debuggers attached to the motherboard.

2.2 Resettable PCRs and TPM v1.2

The specification for version 1.2 TPMs [21] includes several new features. The most important feature for our work is the inclusion of dynamically resettable platform configuration registers (PCRs). On v1.1b TPMs, the value in a PCR can be reset only by rebooting the computer. With v1.2 TPMs, certain registers (currently PCRs 17–22) can be reset under carefully controlled circumstances. For example, the *SKINIT* instruction will reset the value of PCR 17 to zero before extending it with the measurement of the SLB. Note that PCR 17 assumes a default value of twenty bytes of 0xff when a system first boots. This allows a remote verifier to distinguish between code run using *SKINIT* and code run immediately after a reboot, which is necessary since any code that can access the TPM can extend PCRs. Thus, a system using a v1.2 TPM can attest to the fact that the *SKINIT* instruction was executed with a particular SLB.

2.3 Attestation

Systems equipped with a TPM can generate attestations, which are digitally signed aggregates of a TPM's PCR values. The TPM uses a private Attestation Identity Key (AIK) to produce signatures, which remote parties can verify using the corresponding public AIK.

3 Problem Definition

In this section, we summarize the goals for our architecture and describe the adversaries we seek to thwart.

3.1 Goals

Below, we enumerate the goals for our architecture:

- **Isolation.** Isolate security-sensitive code execution from all other code and devices on the system, including the operating system.
- **Provable Protection.** Convince a remote party that the security-sensitive code executed with the proper protections.
- **Meaningful Attestation.** Provide meaningful attestations that include measurements of exactly the security-sensitive code and its inputs and outputs, and nothing else. This provides the dual advantages of giving the verifier a tractable verification task (in the sense of actually deriving meaning from the measurements, as opposed to learning only that millions of lines of code were executed), and leaking as little information as possible about the remaining software state on the attesting system (since it is not security-relevant).
- **Minimal Mandatory TCB.** Minimize the amount of software that must be trusted. While a particular application may need additional functionality added to its TCB, e.g., to display an image on the screen, the amount of code that every application must include in its TCB should be minimized.

The original design for AMD and Intel's new technology called for the initiation of a "secure" VMM following the *SKINIT* instruction [9]. While this approach achieves our first two goals, it only partially achieves the goal of Meaningful Attestation, and it fails to provide a Minimal Mandatory TCB. All applications must completely trust the VMM which increases the size of the TCB and reduces the usefulness of software attestation.

3.2 Adversary Model

In designing our architecture, we allow the adversary the ability to run arbitrary code on the targeted computer. The adversary can control the operating system, devices that use DMA (direct memory access), and invoke *SKINIT* on SLBs of its choosing. Like the Trusted Computing Group's specification for the TPM [21], we allow the adversary to launch simple hardware attacks, such as power cycling the machine at arbitrary times, but we assume she cannot use highly sophisticated attacks, e.g., monitoring and modifying communications on the high-speed bus between the CPU and main memory. Note that the comparatively low-speed bus between the CPU and the TPM employs a special protocol designed to thwart snooping [21].

4 Secure Execution Architecture

Below, we describe the design for our architecture. Due to space constraints, we limit the discussion to a high-level overview. We also describe various extensions that can enhance the basic functionality.

4.1 High-level Design

Using AMD SVM and a v1.2 TPM, we show how to execute a small piece of code, which we call a Piece of Application Logic (PAL). The PAL is executed with much stronger isolation guarantees than modern operating systems can provide, while minimizing the amount of additional code that must be trusted. The PAL is protected from all software running on the system, from all of the peripherals installed on the PC, and even from hardware debuggers attached directly to the motherboard. At present, the application programmer must provide all of the code that will compose the PAL, though, as we discuss in Section 6, a variety of techniques exist to automate this process. The operating system must carefully consider which PALs it wishes to execute (for example, by performing its own measurement before invoking *SKINIT*), since by default a PAL has considerable power. We discuss techniques for limiting this power in Section 6.

In our system, instead of using the *SKINIT* instruction to launch a new VMM or SK and wipe out all previous execution state (cf., Sec. 15.26.6 of [1]), we preserve the current execution environment, invoke the *SKINIT* instruction with the PAL as a parameter, and then resume the legacy OS once the PAL terminates. Below, we describe this process in more detail.

Invoking the PAL. In order to execute a PAL with our enhanced protection guarantees, code operating at ring 0 (e.g., a kernel module) must first save the state of the current execution environment to a well-known location. This includes the base address of the page tables, global and local descriptor tables (if present), interrupt descriptor tables, the task register contents, extended features register (EFER) contents, and certain bits in the EFLAGS register. On a multi-CPU system, *SKINIT* must be executed by the bootstrap processor (BSP). First, however, the OS must deschedule all application processors (APs) and send each one an INIT inter-processor-interrupt (IPI) so that they enter a halted state [1]. The binary for the PAL is then passed as a parameter to the invocation of the *SKINIT* instruction. Note that although the OS invokes the PAL, the OS need not be trusted for the PAL to execute securely. Once the PAL starts to execute, the OS cannot tamper with the execution environment or monitor it in any way. The CPU and TPM guarantee that a legitimate PAL's secrets cannot be read by a modified PAL.

The Secure Execution Environment. The invocation of the *SKINIT* instruction automatically resets the TPM's PCRs 17–22 and extends PCR 17 with the hash of the PAL. A tiny shim layer of system code (consisting of a few hundred lines of code) extends PCR 18 with the input parameters to the PAL and jumps to the beginning of the PAL. Note that the PAL and the shim combine to form the SLB, and thus both are included in the measurement performed by the *SKINIT* instruction. While the PAL executes, it enjoys all of the protections described in Section 2.1: protection from DMA, protection from software executing on other processors, and even protection from hardware debuggers. When the PAL terminates, it jumps back to the tiny code shim to begin resuming the OS.

Resuming the OS. After the execution of the PAL completes, the tiny shim of code erases all traces of the PAL's execution. It overwrites any memory used, clears values stored in the registers and flushes the processor's caches. The shim also extends PCR 18 with the output values from the PAL, and then extends both PCR 17 and 18 with a known public value to signal the termination of the PAL in subsequent attestations. This prevents untrusted code from claiming that any values it extends into PCRs 17 or 18 were actually generated by the PAL. The shim then restores the state of the original OS from the standard location at which it was stored before the invocation of *SKINIT*. Finally, it resumes execution of the original OS. On a multi-CPU system, the OS sends each application processor (AP) a Startup IPI and reschedules it.

4.2 Extensions

While we have described the basic processes for achieving strong isolation, we also suggest a number of extensions to this basic functionality.

Attestation. In many scenarios, the computer (or *attestor*) performing the security-sensitive operations would like to convince a remote *verifier* that the operation was performed using our SEA. For example, in our SSH-password example, the server would like to convince the client that her password will be handled by a specific piece of trusted code executing with the protections offered by our architecture.

To provide such an attestation, the attestor executes the PAL as described above. When the legacy OS resumes, it can request a quote of PCRs 17 and 18 from the TPM. It must also provide the TPM with a nonce from the verifier, which provides freshness and replay prevention. The TPM will produce a signature over the nonce and the values stored in the PCRs. Using the TPM's AIK (Attestation Identity Key), the verifier can check the authenticity of the quote, and use its knowledge of the PAL and its inputs and outputs to verify that the values in PCRs 17 and 18 correspond to their expected values. Thus, the verifier can be satisfied that the PAL ran with the appropriate protections, even though the quote itself was requested from the TPM by the untrusted OS.

Multiple Invocations. While some PALs may only require one invocation (e.g., generating a user's SSH keypair), many applications may require multiple invocations. For example, an SSL server might wish to use a PAL that creates a public keypair and then on future invocations uses that keypair to establish an SSL connection. Multiple invocations can also be used to break a long-running PAL into shorter pieces, thus achieving a rough form of cooperative multi-tasking with the OS.

A PAL can secure data between invocations by using the TPM to seal its data under the value of PCR 17 (ensuring the data will be available only when PCR 17 contains this value). Since PCR 17 is reset by the *SKINIT* instruction and then immediately extended with the hash of the PAL, only a future invocation of the same PAL using *SKINIT* can produce the same value for PCR 17. Thus, no other PALs will be able to access its secrets.

A PAL can even choose to seal its secrets so that a different PAL can access them. For instance, a key-generation PAL might seal the resulting keys so that a separate key-usage PAL could access them. This can be accomplished by having the first PAL seal its secrets under the value of PCR 17 that would result from resetting the PCR and then extending it with a measurement (hash) of the second PAL.

Secure Communication. Remote parties may wish to communicate securely with a PAL executing on another machine. By creating a secure channel between the PAL and the remote party, the secrecy and integrity of information passed between them can be protected, even if all of the other software on the host has been compromised. We need not include communication software (such as network drivers) in the PAL's TCB, since we can use multiple invocations of a PAL to process data from the remote party while letting the untrusted OS manage the encrypted network packets.

Figure 2 illustrates a protocol for securely conveying a public key from the PAL to a remote party. This protocol is similar to one developed at IBM for linking remote attestation to secure tunnel endpoints [8]. The PAL generates a keypair $\{K_{\mathrm{PAL}}, K_{\mathrm{PAL}}^{-1}\}$ within its secure execution environment. It seals the private key K_{PAL}^{-1} under the value of PCR 17 so that only the identical PAL invoked in the secure execution environment can access it. Note that the PAL developer may extend other application-dependent data into PCR 17 before sealing the private key. This ensures the key will be released only if that application-dependent data is present.

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

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

5 Open Problems

While our architecture meets the goals from Section 3.1, a number of challenges remain.

Malicious or Malfunctioning PALs. In this work, we have primarily focused on a scenario in which the PAL is trusted, but the OS or other applications may be subverted. However, a malicious (or malfunctioning) PAL poses a threat to a legitimate OS, since by default, the PAL has access to the entire memory contents of the system and need not return control to the OS. Thus, without further protections in place, a legitimate OS should launch only PALs it trusts.

Several methods exist by which a legitimate OS could gain confidence in the trustworthiness of a PAL. For example, since each PAL should be relatively small, it may be possible to apply various formal analysis techniques [6] to gain confidence in it. Alternately, the OS could require each PAL to be accompanied by a proof of its safety [15].

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 $\{K_{\mathrm{PAL}}, K_{\mathrm{PAL}}^{-1}\}$ extend($PCR18, h(K_{\mathrm{PAL}})$) seal($PCR17, K_{\mathrm{PAL}}^{-1}$) 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, $nonce$) $\vee\ q.PCR17 \neq h(h(0\|\|\hat{H})\|\|\perp)$ $\vee\ q.PCR18 \neq$ $h(h(h(0\|\|nonce)\|\|h(K_{\mathrm{PAL}}))\|\|\perp)$) then abort
RP:	has authentic K_{PAL} knows server ran SEA

Figure 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 SEA.*

At the cost of a slight expansion in the TCB, we could implement protections to constrain a PAL dynamically and limit the damage it can cause. These controls might take the form of running the PAL in CPU privilege ring 3 (only the shim would execute in ring 0) and using segmentation and/or page table permissions to constrain its memory accesses or employing various forms of software fault isolation [22].

Of course, a malicious PAL could be invoked by an already-compromised OS, potentially bypassing the protections described above. However, since the *SKINIT* instruction is privileged, only code operating at ring 0 can launch a PAL. Since code at that level already controls the entire system, malicious code at ring 0 need not launch a malicious PAL to conduct an attack.

Slow PALs. While we envision PALs as small pieces of code that rapidly execute and return, one can imagine the need for longer running PALs. Since we leave the OS suspended while the PAL executes, a long-running PAL may cause the OS to miss large chunks of time. While we have not yet determined all of the effects this might have, it could potentially interfere with I/O or scheduling code in the OS. As discussed in Section 4.2, using multiple *SKINIT* invocations to break a long-running PAL into several shorter PALs may alleviate this problem. Determining the modifications necessary to allow the OS to adapt to long-running PALs is a direction for future work.

Program Separation. Ideally, the PAL should consist of the minimal amount of code necessary to carry out a security-sensitive task. Rather than including an entire application in the PAL, we would like to separate out only the security-sensitive portion. In the SSH example, we would include the password handling routines, but exclude the portions that encrypt and decrypt network packets. Such program separation can be performed manually [11, 12, 14, 16, 20], but researchers have also developed techniques for automatically decomposing a program into a security-sensitive portion and a less sensitive remainder [3,5,25]. Fortunately, security-sensitive code often involves cryptographic computation that does not rely on sophisticated operating system services and hence it can easily be packaged into a PAL.

User Interaction. While much of our early design focuses on a scenario in which a server uses SEA to perform security-sensitive operations with a client computer serving as a remote verifier, SEA could also significantly improve the security of client computers. For example, our architecture would enable an application that allows a user to securely enter her password regardless of what other software or malware might be resident on the PC. However, secure entry is not enough; the user must also be careful not to enter her password into an insecure application. For example, malware might try to convince the user that the secure password application had been launched and thereby capture her password. Thus, we plan to explore techniques for constructing a secure path from a PAL to the user, i.e., convince the user to enter her password or other sensitive information if and only if the secure password application is running.

6 Related Work

Researchers previously achieved some of the properties provided by our architecture using specialized secure coprocessors [11, 24]. While our work does not achieve the same level of physical tamper-resistance, it provides the same strong software guarantees using modern commodity hardware.

Early schemes for attesting to a platform's software state include the entire software stack (e.g., BIOS, bootloader, OS) [2, 14, 18], making it difficult to extract meaningful guarantees from the resulting attestations. Property-based attestation has been proposed [17] as a mechanism for providing meaningful attestations; unfortunately, evaluating software for the various properties of interest remains an open problem.

Other researchers have leveraged VMMs to execute security-sensitive code in isolation [19, 20]. Garriss et al. employ the new *SKINIT* instruction to eliminate the BIOS and the bootloader from their attestations and TCB [7], but as suggested in the original design [9], after the *SKINIT*, they launch a standard OS or VMM. Thus, application security depends on these large layers of code.

7 Conclusion and Future Work

In this work, we propose a Secure Execution Architecture (SEA) for executing code with strong hardware-based isolation guarantees. We also describe how to convince a remote party that protected execution occurred and how to construct secure communication channels to the security-sensitive code, but various interesting questions remain open. Compared with modern operating systems (or even VMMs), our approach adds a minuscule amount of code to an application's TCB, provides fine-grained, meaningful attestations, and allows application writers to focus on the security of their own code instead of worrying about the safety of the many layers of code beneath them.

We are continuing to explore the open problems described above, and we are in the final stages of implementing SEA and employing it for various applications.

Acknowledgments

The authors would like to thank Mark Luk, Leendert van Doorn, and Elsie Wahlig for their generous support and helpful suggestions. Michael Abd-El-Malek, Scott Garriss, James Newsome, and Diana Parno provided invaluable editing assistance. The feedback and comments from Michael Steiner and our anonymous reviewer were much appreciated.

References

[1] Advanced Micro Devices. AMD64 architecture programmer's manual: Volume 2: System programming. AMD Publication no. 24594 rev. 3.11, Dec. 2005.

[2] W. Arbaugh, D. Farber, and J. Smith. A reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, May 1997.

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

[4] 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, Jan. 2003.

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

[6] 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.

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

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

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

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

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

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

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

[14] 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.

[15] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI*, Oct. 1996.

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

[17] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *the Workshop on New Security Paradigms*, Sept. 2004.

[18] 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.

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

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

[21] Trusted Computing Group. Trusted platform module main specification. `http://www.trustedcomputinggroup.org`, Mar. 2006. Version 1.2, Revision 94.

[22] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *SOSP*, Dec. 1993.

[23] D. A. Wheeler. Linux kernel 2.6: It's worth more! Available at: `http://www.dwheeler.com/essays/linux-kernel-cost.html`, Oct. 2004.

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

[25] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, Aug. 2002.