How Low Can You Go? Recommendations for Hardware-Supported Minimal TCB Code Execution *

Jonathan M. McCune† Bryan Parno† Adrian Perrig† Michael K. Reiter† ‡ Arvind Seshadri†
† Carnegie Mellon University
‡ University of North Carolina at Chapel Hill

Abstract

We explore the extent to which newly available CPU-based security technology can reduce the Trusted Computing Base (TCB) for security-sensitive applications. We find that although this new technology represents a step in the right direction, significant performance issues remain. We offer several suggestions that leverage existing processor technology, retain security, and improve performance. Implementing these recommendations will finally allow application developers to focus exclusively on the security of their own code, enabling it to execute in isolation from the numerous vulnerabilities in the underlying layers of legacy code.

Categories and Subject Descriptors C.4 [*Performance of Systems*]; D.2.11 [*Software Architectures*]; K.6.5 [*Security and Protection*]

General Terms Measurement, Design, Security

Keywords Trusted Computing, Late Launch, Secure Execution

1. Introduction

The architecture of today's computer systems is layered, with applications forming the highest layer and the hardware forming the lowest. With the layered architecture, each application's Trusted Computing Base (TCB), and hence security, depends on many layers of code, including the system firmware (BIOS), the firmware of various peripheral devices, the bootloader, the OS kernel, and the application's own code. With the trend towards increasingly feature-rich and complex systems, the code size and complexity of each layer has grown tremendously. For example, today's OSes consist of several million lines of code and support a wide variety of hardware platforms. With the explosion in size and complexity of an application's TCB, securing applications has become a daunting task.

On a modern computing device, the minimal TCB for executing a piece of code consists of the CPU, the memory, and the interface between them. The challenge then is to develop an architecture that executes application code while relying only on this mandatory TCB, yet simultaneously maintains compatibility with the existing layered systems architecture.

In earlier work [16, 17], we proposed a Secure Execution Architecture (SEA)[1] that executes the security-sensitive code of an application while trusting only the mandatory TCB and a Trusted Platform Module (TPM). SEA achieves this property by executing an application's security-sensitive code in isolation from all other software on the system. The isolation is achieved using the CPU-based isolation technologies present in modern commodity CPUs from AMD and Intel, namely AMD's Secure Virtual Machine (SVM) technology [1] and Intel's Trusted Execution Technology (TXT) [11].

In this paper, we evaluate the performance of SEA on commodity systems. Unfortunately, SVM and TXT were designed for extremely infrequent usage, say once per boot cycle. As a result, we find that the SEA approach on current hardware suffers from performance issues that undermine its appeal. Fortunately, our investigation also reveals that by combining alterations to SEA with hardware modifications to improve performance and concurrency, we can achieve efficient minimal TCB code execution. In other words, we can execute application code while trusting only the mandatory TCB and avoid today's performance issues.

Although other researchers have proposed compelling hardware security architectures, e.g., XOM [14] or AEGIS [23], 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.

In summary, this paper makes the following contributions:

- We specify the hardware requirements for executing application code with a minimal mandatory TCB.
- Using our own implementation of primitives for minimal TCB code execution, we show that current hardware renders it impractical, e.g., paralyzing the processor for a full second to set up a trusted execution session.
- We recommend modifications of commodity hardware to securely improve the performance and concurrency of SEA. In our recommendations, we seek to minimize the changes required, thereby increasing the likelihood of their adoption.

* This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and grants CNS-0509004, 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 a National Science Foundation Graduate Research Fellowship. 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.

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.

[1] We present a list of acronyms in the appendix.

2. Background

We provide information on the hardware technologies we explore.

2.1 Trusted Platform Modules (TPMs)

The TPM is a chip designed by the Trusted Computing Group to strengthen platforms against software attack [25].

2.1.1 TPM-Based Attestation

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

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

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

2.1.2 TPM-Based Sealed Storage

TPMs also provide sealed storage, whereby data can be encrypted using an asymmetric key whose private component never leaves the TPM in unencrypted form. The sealed data can be bound to a particular software state, as defined by the contents of various PCRs. The TPM will only unseal (decrypt) the data when the PCRs contain the same values specified by the seal command. Thus, only specific software can retrieve the sealed values.

2.1.3 Dynamic (Resettable) PCRs

The TPM v1.2 specification [24] allows for *static* and *dynamic* PCRs. Only a system reboot can reset the value in a static PCR, but under the proper conditions, the dynamic PCRs 17–23 can be reset to zero without a reboot (a reboot sets the value of PCRs 17–23 to -1, so that an external 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 performing a late launch (as described below). Thus, software cannot reset PCR 17, though it can be read and extended before or after a late launch.

2.2 Late Launch

Recently, processor vendors AMD and Intel have both released CPU technology designed to eliminate several of the lower layers of software from a system's TCB. The capability of performing a *late launch* is included in AMD CPUs as part of their Secure Virtual Machine (SVM) technology [2], while Intel includes it in their Trusted Execution Technology (TXT) [10], formerly LaGrande Technology (LT). Both AMD and Intel are shipping processors with these capabilities; they can be purchased in commodity computers.

The key new feature offered by the *SKINIT* instruction on AMD (or *SENTER* on Intel) is the ability to *late launch* a Virtual Machine Monitor (VMM) or Security Kernel at an arbitrary time with built-in protection against software-based attacks. At a high-level, the CPU's state is reset and memory protections for a region of code are enabled. The CPU measures the code in the memory region, extends the measurement into a PCR of the TPM, and begins executing the code. Essentially, a late launch provides many of the security benefits of rebooting the computer (e.g., starting from a clean-slate), while bypassing the overhead of a full reboot (i.e., devices remain enabled, the BIOS and bootloader are not invoked, etc.).

We now describe AMD's implementation of late launch, followed by Intel's differences in terminology and technique.

2.2.1 AMD Secure Virtual Machine (SVM)

To "late launch" a VMM with AMD SVM, software in CPU protection ring 0 (e.g., kernel-level code) invokes the *SKINIT* instruction, which takes a physical memory address as its only argument. AMD refers to the memory at this address 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 the dynamic PCRs to zero, and then transmits the (up to 64 KB) contents of the SLB to the TPM so that it can be measured (hashed) and extended into PCR 17. Note that software cannot invoke the command to reset PCR 17. The only way to reset PCR 17 is by executing another *SKINIT* instruction. Thus, future TPM attestations can include the value of PCR 17 to attest to the use of *SKINIT* and to the identity of the SLB loaded.

2.2.2 Intel Trusted Execution Technology (formerly LT)

Intel's TXT is comprised of processor support for virtualization (VT-x) and Safer Mode Extensions (SMX) [11]. 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 an instruction called *SENTER*.[2]

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. In analogy to AMD's DEV protection, Intel protects the memory region containing the ACMod and the SLB from outside memory access using the Memory Protection Table (MPT). However, unlike the 64 KB protected by AMD's DEV, Intel's MPT covers 512 KB by default.

[2] Technically, Intel created a new "leaf" instruction called *GETSEC*, which can be customized to invoke various leaf operations (including *SENTER*).

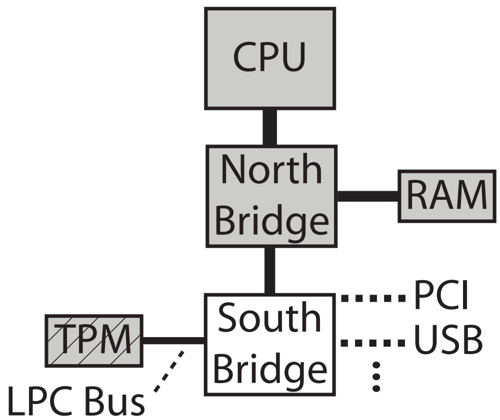


Figure 1. *Chipset configuration for a modern x86 computer. Shaded components are part of the minimal TCB for our execution model. The TPM is shaded differently because it is included for practical reasons but is not an essential part of a stored-program computer architecture.*

3. Execution and Threat Model

We define our execution model and its requirements, and detail the threat model motivating our design.

3.1 Execution Model and Requirements

We focus on an execution model designed to execute small blocks of code with the smallest possible TCB. We term each block of code a Piece of Application Logic (PAL).

Ideally, we would like to enable PAL execution while continuing to support legacy code (both operating systems and applications), but without suffering significant performance penalties.

On a modern stored-program computer, the minimal hardware TCB includes the CPU, memory (RAM), and the interface between the CPU and memory (memory controller, commonly known as the north bridge). Figure 1 shows a modern system, with shaded TCB components. To avoid expanding the TCB any further, we require the following 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 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 depends 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.

3.2 Threat Model

At the software level, the adversary can subvert all of the legacy software on the platform, including the OS or VMM. He can also compromise arbitrary applications and monitor all network traffic. Since the adversary can run code at ring 0, he can invoke the *SKINIT* or *SENTER* instruction with arguments of its choosing. We do not consider DoS attacks, since a malicious OS can always simply power down the machine or otherwise halt execution to deny service.

At the hardware level, we make the same assumptions as the Trusted Computing Group with regard to the TPM [25]. In essence, the attacker can launch simple hardware attacks, such as opening the case, power cycling the computer, or attaching a hardware debugger. The attacker can also compromise add-on hardware such as a DMA-capable Ethernet card with access to the PCI bus. However, the attacker cannot launch sophisticated hardware attacks, such as monitoring the high-speed bus linking the CPU, the north bridge, and memory. Again like the Trusted Computing Group, we omit treatment of covert channels and side-channel attacks though they would be interesting material for future work.

Of course, the PAL itself must be trusted to perform its assigned task securely. The relatively small size of the PAL may facilitate the use of formal analysis techniques to verify the code's security and correctness properties [6].

3.3 The Secure Execution Architecture (SEA)

In our SEA [16, 17], the core idea for minimizing the software TCB is to use the late launch operation to execute a Piece of Application Logic (PAL) in complete isolation from all other software on the system. To frame it in terms of our requirements from Section 3.1, the late launch operation provides *secure initialization*, since it reinitializes the CPU to a known, trusted state without clearing the contents of memory or device state.

The late launch also helps achieve *isolation*, since it sets up DMA protections in the north bridge to isolate the PAL from the unshaded hardware components shown in Figure 1. Since the late launch wipes out the previous execution state, SEA efficiently suspends the untrusted system software before launching the PAL. The suspend of the untrusted system is efficient because all necessary system state can simply remain in-place in memory, provided that the PAL is configured so as not to interfere with the state of the suspended system. Resuming the suspended system after the PAL terminates is efficient for similar reasons.

With SEA, the PAL can protect state between executions or context switches by taking advantage of the TPM's sealed storage capability. During late launch, the TPM's dynamic PCRs are reset to zero, and then PCR(s) 17 (and 18 on Intel systems) are extended with a measurement of the code that begins executing. Thanks to the properties of the CPU, chipset, and the hash function used for measurement [12], these PCR(s) represent the identity of the code loaded for execution, and they cannot be made to contain the measurements of any code without actually loading and executing it.[3] Finally, since the late launch places a measurement of the PAL in the TPM, the system executing a PAL can provide an *attestation* to an external party.

Since SEA requires the TPM for sealed storage and attestations, the TPM must be added to SEA's TCB (see Figure 1). However, the south bridge is not included in the TCB since the TPM is capable of creating a secure channel to the PAL (by engaging in secure transport sessions [25]).

[3] To be precise, this is a load-time measurement. If the code accepts input parameters and contains a vulnerability, it may be possible to overwrite some of the code after measurement and before execution completes. This is a well-known time-of-check, time-of-use problem with load-time attestation, and is not unique to our execution model [19].

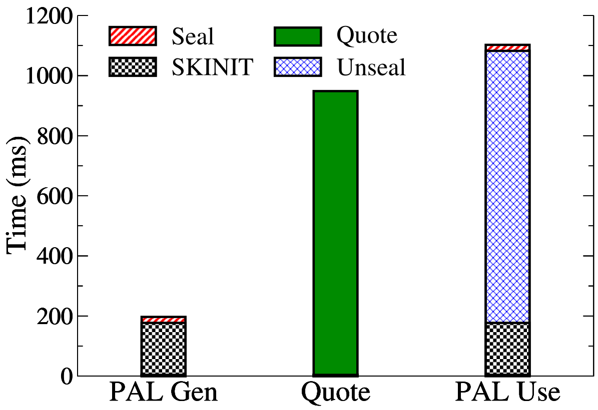


Figure 2. *Breakdown of overheads that will be incurred by generic applications implemented in the SEA model. Measurements were taken using an HP dc5750 containing a 2.2 GHz AMD processor and a Broadcom TPM. PAL Gen represents the overhead for an application that generates data and seals it for later use. PAL Use unseals previous state, modifies it, and optionally reseals it.*

4. Evaluation of Existing Hardware

Through macro and microbenchmarks of our implementation on commodity hardware, we evaluate the performance of SEA, and then conclude with a summary of the issues we identify.

4.1 Implementing SEA Applications

To evaluate the overheads with which nearly every practical application built on SEA will have to contend, we have implemented a generic framework based on our earlier abstract design [17]. To implement the SEA model, we developed a Linux kernel module that suspends the current execution environment and uses late launch to run a PAL. The PAL is then responsible for resuming the previous execution environment once it finishes its application-specific task.

On top of this architecture, we built a number of SEA-enhanced applications [16]. We implemented a kernel rootkit detector and a distributed factoring program that use our architecture to provide isolation and integrity protection. We also use the architecture to protect the confidentiality of a certificate authority's private signing key, and to secure an SSH server's password handling routines. The performance issues we encountered in creating these applications inspired the current paper.

Rather than analyze the performance of specific applications, 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. For example, our certificate authority and SSH PALs each generate a key, seal the private portion for later use, and then return the public key. The second PAL (PAL *Use*) launches, unseals data sealed during a previous session, and operates on that data. It optionally reseals the data and exits. In the certificate authority example, the PAL might unseal the private key and sign some data with it. This example would not require a subsequent seal, since the unsealed key could simply be erased. On the other hand, an application performing a distributed computing task (such as our factoring application or SETI@Home [3]) might perform a limited amount of work and then seal its intermediate state so that it can later resume its computations.

4.2 End-to-End Benchmarks

To evaluate at a macro level the amount of overhead encountered by *Use* and *Gen* PALs, we measured the performance of our implementation on an HP dc5750, which contains a 2.2 GHz AMD Athlon64 X2 Dual Core 4200+ processor and a v1.2 Broadcom TPM. In Section 4.3, we present additional microbenchmarks on other platforms to establish a broader baseline for what kind of performance is available today.

Figure 2 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 would be 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 in the next section.

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.

4.3 Microbenchmarks

To determine if the overheads described above 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 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.

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

4.3.1 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

TPM	CPU Vendor	System Configuration	PAL Size					
			0 KB	4 KB	8 KB	16 KB	32 KB	64 KB
Yes	AMD	**HP dc5750 Avg (ms):**	0.00	11.94	22.98	45.05	89.21	177.52
No		**Tyan n3600R Avg (ms):**	0.01	0.56	1.11	2.21	4.41	8.82
Yes	Intel	**TEP Avg (ms):**	26.39	26.88	27.38	28.37	30.46	34.35

Table 1. *SKINIT and SENTER benchmarks. We run SKINIT benchmarks on AMD systems with and without 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 machine with a TPM.*

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 1 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 [9]. 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 [24]. 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.

4.3.2 Late Launch with an Intel Processor

Recall from Section 2.2.2 that 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 1 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.[4] Table 1 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 [8]. 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 for better trust decisions.

4.3.3 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 SEA'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 2 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 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 slowest 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.

[4] There is no technical reason why a PAL for an AMD system cannot be written in two parts: one that is measured as part of *SKINIT* and another that is measured by the first part before it receives control. This will enable a PAL on AMD systems to achieve improved performance, and suggests that AMD's mechanism is more flexible than Intel's.

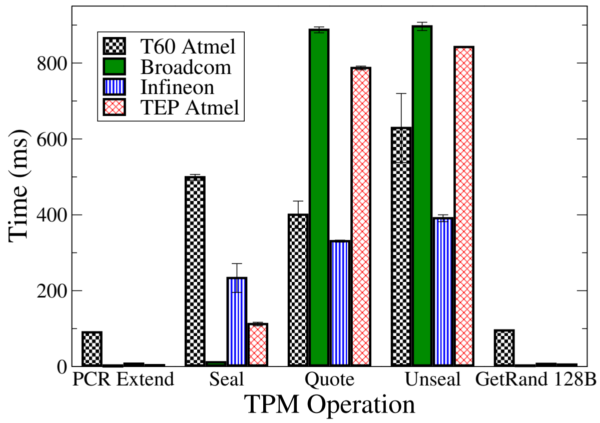


Figure 3. *TPM benchmarks run against the Atmel v1.2 TPM in a Lenovo T60 laptop, the Broadcom v1.2 TPM in an HP dc5750, the Infineon v1.2 TPM in an AMD machine, and the Atmel v1.2 TPM (note that this is not the same as the Atmel TPM in the Lenovo T60 laptop) in the Intel TEP. Error bars indicate the standard deviation over 20 trials (not all error bars are visible).*

4.4 Major 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.

5. Architectural Recommendations

In this section, we make hardware recommendations to alleviate the performance issues we summarize in Section 4.4, while maintaining the security properties of SEA. Specifically, the goal of these recommendations is twofold: (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 multicore computing platforms. Figure 4 shows an example of our desired execution model, where it is assumed that a PAL does not execute on multiple CPU cores at the same time.

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 architecture 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 4) while simultaneously satisfying the two requirements mentioned in the previous paragraph. 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 5.1), followed by our proposed hardware memory isolation mechanism (Section 5.2). Section 5.3 talks about the hardware context switch mechanism we propose. In Section 5.4 we describe changes to the TPM chip to enable external verification. We describe PAL termination in Section 5.5. Section 5.6 ties these recommendations together and presents the life-cycle of a PAL. Finally, Section 5.7 summarizes the expected performance improvement of our recommendations.

5.1 Launching a PAL

We propose a mechanism for securely launching a PAL to achieve the *Secure Initialization* security property from Section 3.1.

5.1.1 Recommendation

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 5(a)) as a structure to hold PAL state and resource allocations, both for the purposes of launching a PAL and for storing the state of a PAL when it is not executing. The PAL and SECB should be contiguous in memory to facilitate memory isolation mechanisms. The SECB entry for allocated memory should consist of a list of physical memory pages allocated to the PAL.

To begin execution of a PAL described by a newly allocated SECB, we propose the addition of a new 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 5.2) 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 5.4),
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.

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 by default. 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 the PAL is interested in requires the CPU to reprogram the interrupt routing logic every time a PAL is scheduled, which may create undesirable overhead or design complexity.

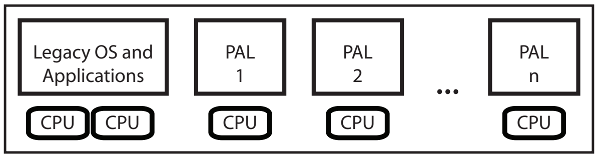
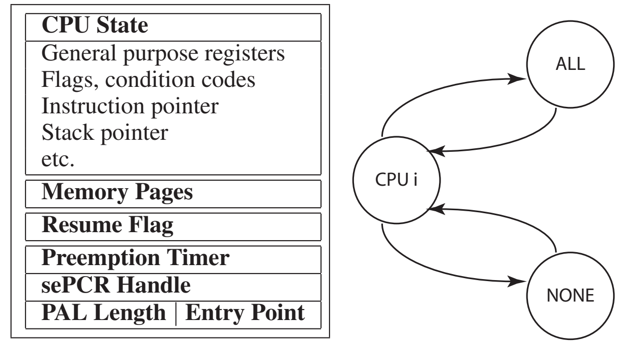


Figure 4. *Physical platform running a legacy OS and applications along with some number of PALs.*



(a) SECB structure. (b) Page states.

Figure 5. *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.*

5.1.2 Suggested Implementation based on Existing Hardware

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.[5] 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, while also improving performance.

5.2 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 3.1.

5.2.1 Recommendation

We propose that the memory controller maintain an access control table with one entry per physical page, where each entry specifies which CPUs (if any) have access to the physical page. The size of this table will be $M \times N$, where M is the number of physical pages present on the platform and N is the maximum number of CPUs. Other multiprocessor designs use a similar partitioning system to protect memory from other processors [13]. To use the access control table, the memory controller must be able to determine which CPU initiates a given memory request.

Figure 5(b) presents the state machine detailing the possible states of an entry in the access control table as context switches (described in Section 5.3) 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.

5.2.2 Suggested Implementation based on Existing Hardware

We can realize hardware memory isolation as an extension to existing DMA protection mechanisms. As noted in Section 2.2, AMD SVM and Intel TXT already support DMA protections for physical memory pages.[6] 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 [21].

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.

5.3 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.

5.3.1 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 instruction: *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 only honored if the SECB's memory page is set to `NONE`. This prevents the untrusted OS from invoking a PAL without it

[5] These structures are the Virtual Machine Control Block (VMCB) and Virtual Machine Control Structure (VMCS) for AMD and Intel, respectively.

[6] The protection mechanisms are the Device Exclusion Vector (DEV) and the Memory Protection Table (MPT) 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 2. *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 Client-Pro 385 with a 2.66 GHz Intel Core 2 Duo.*

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.

5.3.2 Suggested Implementation based on Existing Hardware

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.[7] 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 2 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 2) to essentially the overhead of a VM exit or entry would be a pronounced improvement.

5.4 TPM Support for *SLAUNCH*

Thus far, our focus has been on recommendations to alleviate the two performance bottlenecks identified in Section 4.4. Unfortunately, the functionality 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 3.1.

As implemented with today's hardware, SEA 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 a single 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 5.4.1).
2. A PAL's sePCR must be inaccessible to all other code until the PAL terminates (Section 5.4.2).
3. TPM Quote must be able to address the sePCRs when invoked from untrusted code (Section 5.4.3).
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 5.4.4).
5. A hardware mechanism is required to arbitrate TPM access from multiple CPUs (Section 5.4.5).

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

5.4.1 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.[8] 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.

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 5.4.3.

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.

5.4.2 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

[7] A guest can yield by executing VMMCALL / VMCALL and a VMM can resume a guest by executing VMRUN / VMRESUME for AMD and Intel, respectively.

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

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.

5.4.3 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 5.5.

5.4.4 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 Unseal, 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 be extended with a boolean flag that indicates whether to use an sePCR. The sePCR to use is specified implicitly by the the sePCR handle stored in the PAL's SECB.

5.4.5 TPM Arbitration

Today's TPM-to-CPU communication architecture assumes the use of locking in software 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 about the existence of the TPM lock and wait until the TPM is free to attempt communication.

5.5 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, *SFREE*. *SFREE* is parameterized with the address of an 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. 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 5.3 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 *SKILL* CPU instruction to perform a *secure kill* of 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 randomness.
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.

5.6 PAL Life Cycle

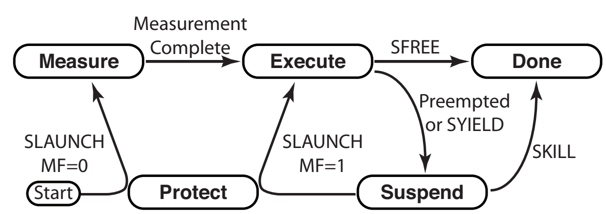


Figure 6. *Life cycle of a PAL.*

Figure 6 summarizes the life cycle of a PAL on a system extended 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 (Figure 7).

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. 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 5.2) 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

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. $\neg p.accessible$) FAIL
MC: SECB.pages = 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 [ℓ].`Quote`) FAIL
  TPM: h = SHA-1(PAL)
  TPM: sePCR [ℓ] = 0
  TPM: sePCR [ℓ] = SHA-1(sePCR [ℓ]$||h$)
  TPM to CPU$_i$: done
  CPU$_i$: SECB.MeasuredFlag = 1

Execute:
CPU$_i$: EIP=SECB.pages.eip
CPU$_i$: Begin executing

Figure 7. *SLAUNCH pseudocode.*

process since it is clear. The *Measured Flag* in the SECB (see Figure 5(a)) is used to distinguish between a PAL that is being executed for the first time and a PAL that is being resumed as part of a context switch. 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 immediately returns a failure code to the CPU, 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 successfully 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. Alternately, if the PAL is ready to exit, it can transition directly to the exit state by executing *SFREE*. These events affect transitions to either the `Suspend` or `Done` state.

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 transition from the `Protect` to the `Execute` state.

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 its own use or to execute other PALs.

5.7 Expected Impact

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

With SEA 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 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 1), while Seal requires 20-500 ms and Unseal requires 290-900 ms (Figure 3). 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 2). 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 our SEA 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 4, the TPM is a major bottleneck for efficient SEA 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 sub-microsecond 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 hardware support to make the operations performant, the power consumed by such operations is wasteful, since we can achieve similar or superior performance with our less power-intensive modifications.

6. Extensions

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

Multicore 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 multicore 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 atomically), 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).

7. Related Work

We are not aware of any other papers analyzing the performance of the new security features offered by AMD and Intel processors. However, we divide less related work into three categories: attempts to define the TCB, reduce the TCB, and invent new hardware to protect the TCB.

Defining the TCB. 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 [4]. 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 [19].

In this paper, 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 [20]. 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 moderate network latency.

Reducing the TCB. Shi et al. proposed BIND, which uses a trusted kernel with late launch technology to attest the correctness of BGP update messages [22]. Unfortunately, their secure kernel was never built.

IBM developed the rHype research hypervisor and subsequently applied their security technology to the sHype hypervisor security architecture [18]. Their goal is to implement mandatory access controls at the hypervisor level. While compelling, their implementation is built for Xen [5], which consists of tens of thousands of lines of code for the hypervisor [15], not to mention the complete Linux kernel running in the privileged domain 0.

Protecting the TCB. New hardware architectures and security coprocessors [26] have been proposed to address security problems (e.g., XOM [14], AEGIS [23], and the IBM 4758 coprocessor [7]). Unfortunately, the cost of widespread deployment of these technologies has proven to be prohibitive. We have focused on recommending changes that provide strong security and performance with a minimum amount of effort.

8. Conclusions

We have explored the extent to which today's latest commodity processors will support an execution model designed to provide applications with a minimal TCB. We have found today's offerings do make minimal TCB code execution possible. However, the current performance penalty is too severe to be practical in daily use. We have recommended changes to the CPU and memory controller that alleviate today's dependence on the TPM chip to protect application state during context switches, and that allow concurrent execution of secure and insecure code. If these changes are implemented, application developers will 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.

References

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

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

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

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

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

[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] 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.

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

[9] Intel Corporation. Intel low pin count (LPC) interface specification. Revision 1.1, Aug. 2002.

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

[11] Intel Corporation. Trusted eXecution Technology – preliminary architecture specification and enabling considerations. Document number 31516803, Nov. 2006.

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

[13] 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*, Apr. 1994.

[14] 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.

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

[16] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. An execution infrastructure for TCB minimization. Technical Report CMU-CyLab-07-018, Carnegie Mellon University, Dec. 2007.

[17] 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.

[18] 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, Feb. 2005.

[19] 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.

[20] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. VanDoorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of the Symposium on Operating Systems Principals (SOSP)*, 2005.

[21] T. Shanley. *The Unabridged Pentium 4*. Addison Wesley, first edition edition, August 2004.

[22] 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.

[23] 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.

[24] Trusted Computing Group. PC client specific TPM interface specification (TIS). Version 1.2, Revision 1.00, July 2005.

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

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

A. Acronyms Used

This table summarizes the acronyms used in this paper.

Acro.	**Expansion**	**Def.**
TCB	Trusted Computing Base	Sec. 1
SEA	Secure Execution Architecture	Sec. 1
TPM	Trusted Platform Module	Sec. 2.1
PCR	Platform Configuration Register	Sec. 2.1.1
SVM	AMD Secure Virtual Machine	Sec. 2.2
TXT	Intel Trusted Execution Technology	Sec. 2.2
SLB	Secure Loader Block	Sec. 2.2.1
PAL	Piece of Application Logic	Sec. 3.1
SECB	Secure Execution Control Block	Sec. 5.1
sePCR	Secure Execution PCR	Sec. 5.4