SeM: A CPU Architecture Extension for Secure Remote Computing

Ofir Shwartz
Electrical Engineering Dept., Technion, Israel
ofirshw@tx.technion.ac.il

Yitzhak Birk
Electrical Engineering Dept., Technion, Israel
birk@ee.technion.ac.il

ABSTRACT

In shared (multi-user) computing environments, platform software (OS, Hypervisor, VMM etc.) and most of the hardware cannot always be trusted (e.g., public clouds), so ensuring the confidentiality and integrity of a user's program (code and data) is critical. It is highly desirable to do so efficiently while accepting existing application binaries, being able to use the services of untrusted software, not modifying the OS, and with minimal intervention in the system's flow.

We present the Secure Machine (SeM), a CPU architecture extension that, unlike previous approaches, does all this. Using novel fine-grained cache and register protection managed by a CPU-resident, publicly identifiable hardware Security Management Unit (SMU), we address both software attacks and off-chip hardware attacks. SeM accepts existing application binaries, which are instrumented automatically, and only incurs negligible performance, power, and area overheads relative to an unprotected platform. SeM is extendable to parallel programs and multiple nodes.

1. INTRODUCTION

We consider a user with a trusted private computer, wishing to run his program on a remote computer such as a public cloud, which concurrently serves multiple (possibly mutually adversarial) users. The user sends a program, comprising code and data, for execution. It can be provided as files on disk or via a network, and program output is collected similarly.

The remote computer's system software may be adversarial, and the only trusted hardware is its CPU chip (with its in-chip caches). Support for identity authentication is assumed (a certificate), merely enabling the establishment of a secure virtual communication channel between the user's trusted private computer and the remote trusted CPU chip. This is done using a secret signature key stored in the trusted CPU chip by some trusted agent, possibly the chip's manufacturer.

In this setting, and without requiring OS modification, we wish to enable a user program to run conventionally on the remote machine: switch context in and out while maintaining its state, allocate memory dynamically, use I/O, and invoke system calls. All this while ensuring its confidentiality and integrity: its secrets (code, data, temporary values, and data communicated via I/O) cannot be discovered by any adversary (confidentiality); and its results, including any information that it sends to the outside world, are unaltered, or an alteration is detected (integrity). (Nonetheless, misbehaving untrusted OS services that are called by the user program can usually only be detected by a user program.)

Overshadow [2], some of whose ideas we adopt, is a comprehensive solution that provides some of the above (e.g., securely running an unchanged application); however, as it is based on a trusted virtual machine monitor (VMM), it does not address an untrusted owner (which can manipulate the VMM, or track and modify memory). Another is Intel's SGX [22]; although it addresses an untrusted owner, its key challenges are applicability to existing programs, resource efficiency, and performance. Software extensions to SGX include PANOPLY [26] (helps develop SGX applications), Haven [4] (for Windows) and Graphene [27] (Linux); these two accept unmodified applications but impose various restrictions on the programs, require a large TCB, and substantially degrade performance. SCONE [33], a small-TCB container-based solution for running unchanged applications, comes closer to achieving the goal. However: the user needs to know security-related aspects of the service to create an image from application binaries; customized support for libraries that use system calls must be developed (currently only libc); performance drops by tens of percents; and it is unclear whether it supports signals and exceptions. Sec. 8 reviews additional related work.

We present the Secure Machine (SeM), an extended CPU architecture enabling secure computing on a computer that is managed by an untrusted entity, jointly addressing the aforementioned needs without any restrictions on the flow of the system or on the untrusted OS/VMM/hypervisor. SeM can easily be integrated into any CPU architecture, and incurs only tiny performance, power, and area penalties. We use prior-art memory encryption and integrity [28], and add a novel cache and register management layer, as well as setup and termination capabilities.

In a nutshell (and ignoring setup), a secure program invokes its trusted instructions, and accesses its trusted data. When the need arises (e.g., OS kernel code for performing a context switch), it invokes an untrusted instruction, at which time the secure program's registers are immediately hidden by SeM (Sec. 3.4). The trusted cached blocks are only accessible by same-process trusted code, so the trusted memory space is protected (Sec. 3.3). When a trusted instruction is subsequently invoked (e.g., returning to user code), the registers are restored immediately. As done in previous works, memory blocks that are fetched or evicted are automatically protected by a memory encryption and validation layer.

HASP '17, June 25, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-5266-6/17/06 $15.00
http://dx.doi.org/10.1145/3092627.3092631

This work was supported in part by the Hasso Plattner Institute.

SeM's main novel protections are its ability to automatically hide register values on the first invocation of an untrusted instruction, and its ability to block untrusted memory-access instructions from accessing trusted cached blocks. SeM's main novel performance benefits are its ability to hide and restore the registers' data in a single clock cycle, and fast context switches without flushing the cache. SeM's novel applicability benefits are that the CPU is largely unchanged, the security mechanisms are hidden from the program and the unmodified OS, and the programmer need not modify the application code.

We designed and implemented *SeM-Prepare*, a tool running on the user's trusted computer for preparing existing binaries for SeM (running offline or in the course of program submission to the cloud, sometimes referred to as *application deployment*), and the *SeM-Simulate* simulator to then execute them. We then ran the SPEC CPU2006 benchmark suite [15], thereby demonstrating completeness and correct results, and showing overheads to be negligible.

This paper focuses on the core architecture and on the protection of the user's process from (software and off-chip hardware) adversaries. This work has also been extended to support multi-thread, -core and –node settings, as well as task and thread migration, making it relevant to parallel programs. These extensions are largely "orthogonal" to the core SeM architecture, and for lack of space will be reported elsewhere.

The key building blocks contributed by this work are:

- A novel hardware-maintained secure process context management, allowing efficient and secure switching between programs, and between the program and the OS.
- Secure Access, a novel method for cache access control, coupling authenticated instructions and authenticated data; this allows unencrypted code and data of adversarial programs to securely co-reside in cache.
- An automatic tool for preparing existing binaries with a small code footprint; no programming efforts.

The remainder of the paper is organized as follows. Sec. 2: our threat model; Sec. 3: SeM architecture; Sec. 4: a secure process' interaction with the OS, including attacks by a hostile OS; Sec. 5: automatic instrumentation of user binaries; Sec. 6: implementation and evaluation; Sec. 7: related work; and Sec. 8 concludes. The appendix lists the SMU instructions.

2. THREAT MODEL

The user's private computer is trusted. In the shared SeM computer ("the computer"), we assume that an adversary:

- Completely controls the OS/VMM/hypervisor, running in the most privileged ring, including the ability to change or implant code both in advance and during runtime;
- Has access to the computer's boards. It can monitor and alter board signals or emulate board-connected devices;
- Before, after and during execution, may try to read or change user code, data and results, or interfere with the OS services that the program receives;
- However, it cannot physically inspect or alter CPU chip internals. The CPU chip (HW) is assumed to be correct, and its manufacturer is trusted.

Side and covert channel attacks [29,32], as well as user application software bugs, are not addressed here, but we do not create new vulnerability in this respect. Denial of service of any kind [23] is also out of scope, as the system owner may simply shut it down. We also assume that silicon internal secrets stored during manufacturing were not leaked.

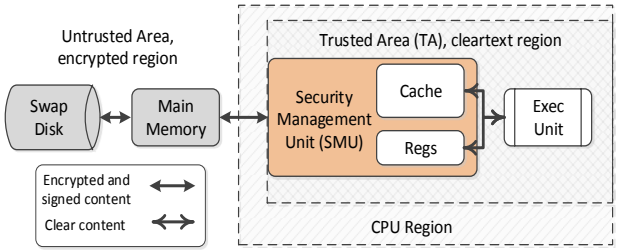

Figure 1: SeM hardware block diagram

3. THE SEM ARCHITECTURE

3.1. Overview

The basic Secure Machine (Fig. 1) comprises a single-core multi-user computer. SeM's main hardware is the Security Management Unit (SMU). It exclusively manages and controls access to the CPU registers (Sec. 3.4) and caches (Sec 3.3) in an on-chip physical domain dubbed the Trusted Area (TA), and serves as a gatekeeper between the TA and the untrusted world.

To ensure confidentiality, the user's code and data are encrypted whenever outside the TA and, for integrity, are signed using a Message Authentication Code (MAC). We use a counter mode (CM) technique for memory encryption, and a lightweight secure hash for authentication (MAC); GCM [17] is an authenticated encryption technique that provides both. We use Bonsai Merkle Tree (BMT) with a TA-resident root hash [16] to keep the integrity of the CM seeds. All these are widely used in previous works, and have been proven safe and efficient (in performance and memory footprint) [16,17]. SeM is agnostic to the memory encryption and authentication techniques, as long as these provide memory confidentiality (when desired) and integrity breech indication (mandatory).

CM encryption protects the memory at cache block granularity by assigning a seed value (commonly 64 bits) for each block's virtual address; these seeds are cached. BMT is a hash tree used for maintaining the integrity of the seeds, so that an old data block with its corresponding old seed cannot be injected into memory. The BMT blocks are also cached, so only missing BMT nodes (rather than the entire hash tree) need to validate when fetched. The performance implication of those is small [16,17], and is not unique to SeM.

Both CM encryption and BMT require memory for metadata (encryption seeds and a hash tree). This memory need not be protected, because an attacker is unlikely to inject correct values without holding the secret keys [16,17]. These small regions are allocated and zeroed at the secure program's request; if the OS fails to cooperate, an error is detected upon access. The SMU performs these operations using metadata (e.g., secret encryption and authentication keys) stored securely for each secure program during its setup. As in many other secure architectures [3,7], CPU debugging (which exposes detailed state), is disabled for a secure program.

SeM can be tailored to either physically or virtually addressed caches. When fetching a missing block, its virtual address is known in both cases. When evicting, a virtually addressed cache can store the updated seed immediately, but a physically addressed cache needs to perform a reverse TLB lookup to find the correct seed to update; this can be done in the background, without delaying the actual eviction. From here on we assume virtually addressed caches.

The rest of the computer is largely unmodified. The OS can start and stop processes, switch among them, and perform any conventional OS task, but it only accesses the cache under SMU supervision. Likewise for the hypervisor or any layer between a user program and the actual CPU hardware.

The flow (e.g., submitting to the cloud): A user program's binary (in his own trusted computer) is statically linked with shared library functions that it requires (similarly to [32]), and is then automatically instrumented with some additional instructions (explained later). Next, it is encrypted and signed, and is then sent to SeM through an untrusted medium. To execute the program, a secure connection is established between the user's computer and the SMU to securely store the program's settings (e.g., keys) in the SMU. This enables the SMU to provide each secure program with encryption, decryption and authentication services for code and data using the program's unique keys. The program is then executed, using these keys. Upon completion, the user may collect the encrypted output and validation information from the SMU.

Many previous works required attestation of the machine's cumulative state [12,28]; this state is very hard to verify, as it varies among systems and changes with system updates. (E.g., each OS update modifies OS executables, resulting in a different state hash.) In SeM, we use simple attestation to authenticate the existence of a genuine SMU, regardless of the state of the machine. This is easily doable using a publicly provable signature [11] (Sec. 1); e.g., by requesting the SMU to sign a requestor-generated random number.

SeM runs an untrusted management program for direct communication with the (possibly remote) user. Data passing through the management program is safe, as it is encrypted and the decryption keys are only known to the SMU.

3.2. Security Management Unit (SMU)

The SMU is SeM's core hardware. It resides in the CPU chip, situated between the last level on-chip trusted cache and the rest of the memory system, and between the L1 cache and the execution unit. It creates a boundary between the TA (comprising the execution unit, registers, trusted caches, etc.) and the untrusted domain (comprising optional untrusted cache levels and everything that resides off-chip). The SMU's main roles are:

- Securely store and manage cryptographic keys;
- Hide and restore register values upon switching between different modes of operation (secure / non-secure);
- Enforce the memory access control;
- Decrypt (encrypt) cache blocks upon entry into (eviction from) the TA, and maintain their integrity;

Fig. 2 depicts the SMU: on the left, it is connected to the untrusted levels of the memory hierarchy, and on the right ---

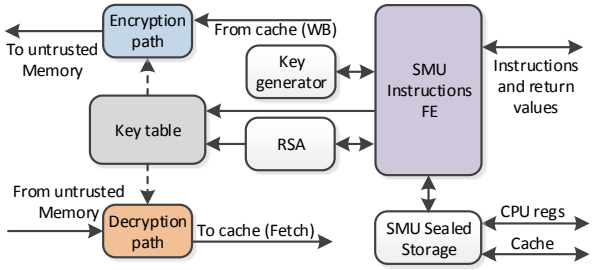

Figure 2: SMU block diagram

to the TA (execution unit and on-chip caches). The SMU comprises encryption and decryption units for both symmetric (e.g. GCM) and asymmetric (e.g. RSA [11]) ciphers, signing and signature validation units (e.g. GHASH and RSA), a key table, and a small storage, dubbed the SMU Sealed Storage (SSS), for temporary data. The asymmetric cipher is used to establish a secure connection between the user's computer and the SMU for sending the program's encryption and authentication keys to store in an SMU table.

When a program first launches, it attaches its process ID to this SMU table entry (Sec. 5). The program's code and data are encrypted using symmetric CM encryption (GCM), and are signed using a secure MAC (GHASH). As suggested in SDSM [9], whenever the seed is zero, the memory block's virtual address serves as the seed for encryption pad creation. Since the seed memory is initialized to zeros, this obviates the need to supply initial seeds with the program. Also, non-zero seeds are concatenated with '1' during encryption pad creation, so initial and runtime encryption pads are never the same. Upon a last-level cache miss in the TA, the SMU uses the symmetric decryption and authentication units to decrypt and authenticate incoming blocks. Modified cache blocks are encrypted and signed upon eviction from the TA, preserving their secrecy and integrity.

Enforcing memory access control ensures that only the (same-process) secure code can access its secure data. This is the core of SeM's protection against software attacks, which nonetheless allows blocks belonging to mutually adversarial applications to co-reside in the cache, unencrypted (Sec. 3.3). This requires that upon initiation of a new secure process, the SMU clear existing secret cache blocks of the same PID (possibly of an old secure process). Sec. 4 discusses some attacks on SeM, including an attack by an adversarial OS.

The SMU operates in two modes: *trusted* and *untrusted* (Sec. 3.4). In *trusted mode*, it expects to run only the secure program, namely run secret instructions. In *untrusted mode*, it expects to run untrusted code, such as a non-secure application or the OS, where the latter may run from within the context of the secure application (e.g., during an interrupt or a system call). The SMU uses these operating modes to provide the register access control. Switching between modes and register maintenance are discussed in Sec. 3.4.

The SMU table holds the keys and configurations for the secure programs. Each table entry contains:

PID – the process ID of the secure program using this entry.

Skey – a symmetric key for memory encryption.

Mkey – a symmetric key for memory authentication, if needed (e.g., GHASH uses Skey for both).

Root Hash – the root value of the BMT for integrity.
Process Hash – secure process hash, used to connect the secure process with its table entry.
First LEP – the address of the first secure instruction.
Sig LEP – signal handling entry point.
Error Status – holds the error code.

Upon launching of a process containing the Process Hash (more in Sec. 5), its PID is stored. If a table entry with this PID already exists, it is erased, and any secure remnants of the same-PID program are removed from the TA. The table entry must remain in the SMU throughout the execution of the secure application, even when it is not active (for cache evictions, if required), so the size of this table limits the number of concurrent secure applications. However, a typical SMU table entry is 256 bits, allowing many secure programs to run concurrently using a small in-chip memory.

The SMU executes special instructions, required for SeM's operation (see appendix), and for security reasons these are treated as fences in out-of-order CPUs. Some are automatically added to the program's code as it is submitted for execution (sparsely), and some are used by the untrusted setup application.

3.3. **Secure Access**

We now present a novel cache access management approach that allows adversarial applications' blocks to concurrently reside in cache unencrypted, while maintaining complete isolation. SeM runs multiple unrelated processes. Our encryption and authentication scheme is based on per-secure-process secret keys stored inside the SMU, which decides whether to grant a given program encryption and decryption services for any given cache block and whether to grant it access to cache (cleartext) blocks. We discuss unified caches for instructions and data, but separate ones behave similarly.

Instructions and data required for execution must be fetched into the CPU's clear cache, residing in the TA. If there is an SMU table entry containing the current process ID, the encrypted block is decrypted using the corresponding decryption key, and its MAC is checked. If correct, the **clear** block is considered *authentic* and is stored in the TA with an *Auth=True* mark; else, the **originally fetched** block is stored in the TA with *Auth=False*, and is considered *non-authentic*. This is done at cache block granularity, and only upon cache miss. Upon eviction from the TA, authentic blocks are encrypted and signed, while non-authentic ones are simply evicted (Fig. 3). Wishing to support integrity only, encryption and decryption can be bypassed. A block's *Auth* bit is reset upon cache block eviction and purging, and is propagated between the clear cache levels with the block itself. Any program, privileged as it may be, only gets decryption services by the SMU using its own private keys, if exist. Consequently, although the operating system can access any of the program's private memory outside the TA, secrecy is ensured (ciphertext).

Decrypting a memory block with GCM requires its seed, which does not exist for untrusted code that runs under the context of the secure application. Therefore, the SMU shall only perform decryption attempts for memory blocks that have the required data.

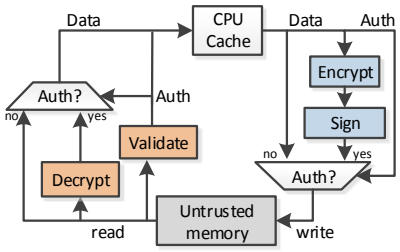

Figure 3: SMU encryption and decryption paths

The cache contains clear-text instructions and data, which may belong to unrelated processes and to different users. Each cache block's tag includes its PID, providing inter-process isolation. Attacks by a malicious OS are discussed in Sec. 4.

For data confidentiality in the clear cache, we employ *Secure Access*: authentic data blocks can only be accessed by (same-process) authenticated load/store instructions, and non-authentic blocks can only be accessed by non-authentic instructions. Upon violation of *Secure Access*, the process halts and an error is declared.

3.4. **Mode Changing and Stack Management**

We now present a novel mechanism for automatic register hiding and maintenance, using the SMU modes. Any program starts running in *untrusted* mode. If and when its secret code starts to execute, the SMU switches to *trusted* mode. Interrupts may occur, suspending the secure program. Their handlers must run in *untrusted* mode (their code is untrusted), so secret information is not leaked. Later, to resume execution, the SMU reverts to *trusted* mode.

Every program starts with a conventional non-secure stack, allocated by the OS. Secure programs also require a secure stack (protected for secrecy and integrity) for managing function calls in trusted mode, so it is allocated (by the non-secure code) and initialized (by the secure code, as soon as it starts); these instructions are automatically added into the binary. Switching modes also switches between stacks.

A fetched instruction inherits the authenticity mark of its L1 cache block, and the SMU switches modes automatically to match the mark of the invoked instruction. When changing to untrusted mode, the SMU first stores the contents of the registers (the secret context) in the SMU Sealed Storage (SSS), clears them, and changes the stack pointer to the non-secure stack). It also stores the address of the next authentic instruction to execute, dubbed the Legal Entry Point (LEP), the PID of the running process, and sets a validity mark for the content of the SSS (Pseudocode 1.) Then, the non-authentic (untrusted) code may execute safely. (In out-of-order CPUs, the untrusted instruction is delayed until the last trusted instruction fetched is committed and the registers are hidden, and the LEP is the address of the next instruction.)

Attempting to execute an authentic instruction in untrusted mode only succeeds if its address matches the process' LEP and the data in the SSS is valid and matches the PID. If so, the SMU restores the register values and the secure stack pointer (the secret context) from the SSS, and changes the process to trusted mode; else the program halts and an error is declared. In both cases, the SSS is invalidated. (Pseudocode 1.) By so doing, the SMU verifies that the secure program has resumed

```
SMU_ChageToUntrustedMode (NextLEP)
    LEP=NextLEP
    Store secret context into SSS
    Set SMU.SSS.valid = True
    Clear registers
    Mode=Untrusted

SMU_ChangeToTrustedMode (InstAddr)
   If (InstAddr==LEP) and (SMU.SSS.valid) and
     (SMU.SSS.PID ==PID)
         Restore registers from SSS
         SMU.SSS.valid = False
         Mode=Trusted
   else
         Report error and halt
```

Pseudocode 1: Operations performed by the SMU (by hardware) during an automatic mode switch

from its expected point of execution with the correct register values. (In Sec. 4 we add an LEP for handling signals.) Upon initiating a secure program, the SMU creates an empty secret context in the SSS (with the first LEP from the SMU entry); only during the first switch to *trusted mode*, the register values are preserved (not restored), but the entry point is enforced.

Switching to untrusted mode is fast: the registers may be cleared by simply switching a register window to a pending set of erased registers. The switched-out set of registers acts as the SSS, along with the LEP (which must be known for fetching the next instruction) and the PID. Switching to trusted mode is also done instantly. Verifying the validity mark and comparing the PID are simple operations. Also, register values are restored by switching a register window.

3.5. **Sharing Data with Untrusted Code**

To receive services by untrusted code, such as some OS system calls, a secure application may need to reveal some of its data. The following SMU instructions allow **only trusted code** to bypass the *Secure Access* mechanism, so these instructions must be authentic to run.

- *SMU_StoreNA(address, data)* – stores data into a memory block regardless of its *Auth* status, and sets its *Auth* bit to *False*, making it accessible to untrusted code.
- *SMU_LoadNA(address)* – loads data from a memory block regardless of its *Auth* status, for importing untrusted data by trusted code.
- *SMU_InitA(addr, size)* – stores zeros into an entire memory region of size *size* that starts at address *addr*; sets *Auth* bit to *True* for in-cache blocks; and signs and encrypts blocks that are not, used in conjunction with a write-no-allocate cache policy. Used for initializing allocated memory, so it is accessible by trusted code.

4. INTERACTIONS WITH THE OS

4.1. **Operating System Services**

SMU modes and *Secure Access* ensure that confidentiality is preserved even with unexpected invocation of untrusted code. However, the secure program runs concurrently with other (possibly adversarial) programs, so its context must be securely evicted (and later restored) on context switches. Also, although shared library functions are statically linked into the binary when prepared for SeM, system calls invoked by these functions must still be allowed to execute. Furthermore, dynamically allocated memory must get initialized to be used under *Secure Access*. Lastly, signals may be invoked and must be handled. All these are OS services, and are discussed in this section, including the required novel hardware.

Context Switch

At context switches, the OS (untrusted) modifies the page table register (e.g., CR3 in Intel Architecture). A hardware watchdog normally exists, which invokes microcode upon page table register modifications [34]. We use this mechanism to also call *SMU_EvictContext* to evict the switched-out content of the SSS from the SMU into the process' memory (cache), and *SMU_RestoreContext* to restore the secure context of the switched-in secure program from memory back into the SSS. When evicted from the TA cache, the process' memory protection is applied.

In 64-bit systems, the size of the SSS is roughly 350 bytes, similar to a thread control block, requiring ~40 cycles for these instructions, which is negligible relative to the thousands required for a context switch [10].

Resuming execution of a secure program entails attempting to execute its next authentic instruction, which causes the SMU to verify its address and the SSS content (Sec. 3.4). If the OS refrains from updating the page table register on context switch, then the SMU evict and restore calls will not be invoked. Having multiple secure programs running on the machine, the PID check upon changing to trusted mode will fail, causing the SMU to halt the secure program and to report an error. In any case, information is never leaked. *SMU_EvictContext* and *SMU_RestoreContext* are only required for secure programs; if called for a non-secure program, they finish immediately.

System Calls

Although shared library functions are statically linked into the binary during SeM preparation, system calls (untrusted code) are still used for obtaining OS services. These require passing of arguments and possibly small memory structures, and each system call has its own requirements. We obtain the system call ID by analyzing the value in the system call number register at the time of invoking the *syscall* instruction (*rax* in System V AMD64 ABI [8]); since it is always set with an immediate value, this can be done statically.

We substitute *SMU_syscall(argnum)* for each *syscall* instruction; *argnum* (e.g., 0 - 6) is the required number of arguments. The SMU will not clear these registers when next switching to untrusted (likely to occur immediately), thereby passing them to the system call. When returning to trusted mode (for the same program), it will not restore the result register value, thereby passing it to the program.

To avoid hard-coding the system call convention in hardware, we use an SMU instruction to set the convention (per-secure program) using a bitmap of registers; this must run as trusted code at the beginning of the secure program, so it is automatically embedded during instrumentation. To support system calls that require small memory structures (e.g. *sys_write* for file access), we use dedicated wrappers using *SMU_StoreNA and SMU_LoadNA*. The general argument

passing approach is similar to Overshadow [2] and SCONE [32]; yet, unlike them, we incur no overhead whenever only argument values are required.

Dynamic memory allocation

Newly allocated memory must be accessible to the secure program, and its integrity must be kept. Therefore, is must be initialized (zeros) by *SMU_InitA* to ensure its correct encryption and authentication for later use. The result: new blocks outside the TA contain zeros (signed and encrypted correctly), and cached ones contain zeros with *Auth=True*.

We do that by replacing each *malloc()* call in the original binary with *sem_malloc()* in the SeM-ready binary. *Sem_malloc()* invokes *malloc()* with its original parameters, followed by an *SMU_InitA* instruction.

The untrusted operating system must allocate the memory required for the block's metadata (CM encryption seeds and BMT hashes), if not done before. Else, an error will be raised when accessed, so wrong OS behavior will be caught.

Signal Handling

Signals may be sent to any program during execution. For lack of space, we only sketch our approach. To support program defined signal handling, SeM uses an additional LEP, at which we place a trusted signal-handling-entry function (SHEF) that identifies the signal, runs the desired signal handler, and returns (similarly to SGX). If signal handling is required, a SHEF is automatically added to the program during instrumentation, and its address is set as a *sig LEP* in the SMU entry. When a signal handler is registered (*sys_rt_sigaction* system call), we register this handler instead, and update its own mapping. We use a syscall wrapper for that (embedded automatically).

4.2. Hostile OS Attacks and SeM's Resilience

An attacker may try to read or change code or data, rerun the secure program for various purposes, or even manipulate the secure context data or the flow of execution. We have checked SeM against these, but details are omitted for lack of space. We do, however, now discuss two privileged code attacks that demonstrate SeM's main resilience mechanisms.

Forged identity attack: Consider a privileged attacker (e.g., OS) that tries using the PID of a secure program to access its clear cache. The cache natively allows same-PID cache access, but *Secure Access* allows secret (authentic) data to be accessed only by same-process authentic instructions; the attacker must therefore also properly encrypt its attacking code in order to gain access to the secret data. Not knowing the secure program's secret keys, this is impractical.

Iago attacks: These attacks use carefully chosen system call return values to manipulate library functions that are statically linked into the secure (and therefore trusted) program. [19] shows that manipulating *brk()* system call return value can cause *malloc()* to allocate an intended secure memory in a non-secure region, thus exposing data stored in it. In SeM, however, dynamically allocated memory always becomes secure memory, so data stored in it is always protected against leakage. Moreover, Iago uses return oriented programming (ROP) [31] to redirect the secure program to the attacker's code. However, if the ROP target is untrusted then automatic mode change will prevent any data leakage. Finally, Iago attacks may be defeated by checking system calls' return values [6,13,14], which can also be done in SeM.

5. SEM OPERATION

We use Secure Program Submission to the cloud, similarly to conventional cloud application submission (deployment), to prepare and send a secure program for execution. It is automatically instrumented (Sec. 4), signed and encrypted in the user's own, trusted machine. Next, the machine establishes a secure channel with SeM, and stores the program's keys into the SMU. The program can then be executed on SeM. Finally, results are prepared for the user to collect them (if needed). For lack of space, we only discuss the instrumentation of program binaries, but all the SMU instructions for the setup and finish processes are listed in the appendix.

SeM-Prepare is a small, powerful, and automatic tool for preparing previously-compiled Linux binaries (ELF) into SeM-ready Linux binaries. (Similar code for different OSs.)

First, it adds two new sections to the binary: '*nosec_init*' and '*sec_init*'. *nosec_init* is set as the new entry point instead of *main()*. It connects the secure program with its already existing SMU entry by invoking *SMU_SetPID*(*phash),* where *phash* is either chosen by SeM-Prepare or supplied by the user. (*phash* is the same value as in the SMU entry, and is not a secret.) It then allocates memory for a secure stack and for the CM and BMT metadata (Sec. 3.1), and calls *sec_init* that initializes the secure stack by *SMU_InitA*, updates the stack pointer, and calls *main()* (so the secure code begins). The address of *sec_init* is the *First LEP* in the SMU table entry.

Next, for trusted code to execute correctly on SeM, SeM-Prepare statically links shared functions, analyzes and replaces *syscall*s with *SMU_syscall*s, implants wrappers for *malloc*, required system calls, and a SHEF (if required) (Sec. 4). Finally, the resulting binary is encrypted and signed, leaving the *nosec_init* section unencrypted. The keys are either provided by the user or chosen randomly.

6. IMPLEMENTAITON AND EVALUATION

SeM-Prepare was described in detail in Sec. 5.

SeM-Simulator runs SeM-ready binaries by simulating the SMU's behavior. It was implemented using Pin [7].

SeM's overhead on program execution is in memory access (encryption/decryption), memory allocations (secure init), and system call wrappers (only when wrappers are needed). The system (OS) oriented overhead is for the context switch (merely ~1.02X the non-secure context switch, Sec. 4). All these have a negligible effect on unmodified performance-critical elements such as the cache (flushes are not required) and branch prediction, and none at all on the execution units. Also, mode changes impose no overhead (Sec. 3.4).

Evaluation. We instrumented the SPEC CPU2006 benchmark suite [15] using SeM-Prepare, and then ran it on SeM-Simulator. SPEC CPU2006 was chosen because it targets the causes for overheads in SeM.

Performance is measured relative to the corresponding non-secure application. Overhead caused by memory encryption and authentication, including memory accesses for fetching missing GCM seeds, their BMT hash authentication, and

cache contamination, comes inherently from the chosen memory encryption and authentication technique (and its implementation), as in any secure architecture; we therefore rely on previous works' simulations [16,17] for these.

Fig. 4 shows the performance penalty with and without memory encryption relative to no security at all (running the benchmarks unchanged). The mean penalty is under 1.9%, of which 1.8% is for memory encryption, so SeM adds merely 0.1%. (I/O traffic and allocated memory are also in Fig. 4.) Thus, no other solution (past or future) can do much better.

All programs successfully changed modes, invoked system calls, used files, and allocated secure memory. Finally, we verified that the SeM-ready benchmark results matched the original ones. Besides evaluating SeM's performance, this also serves as a strong indication for the applicability of SeM to existing binaries, without requiring programming effort.

The overall area overhead for SeM is <0.1% of a PC-CPU. Power overhead is mainly for the cryptographic primitives used for the memory encryption, and is negligible. Additional memory footprint is ~5% for CM counters and BMT, but the actual percentage depends on the seed- and cache block size.

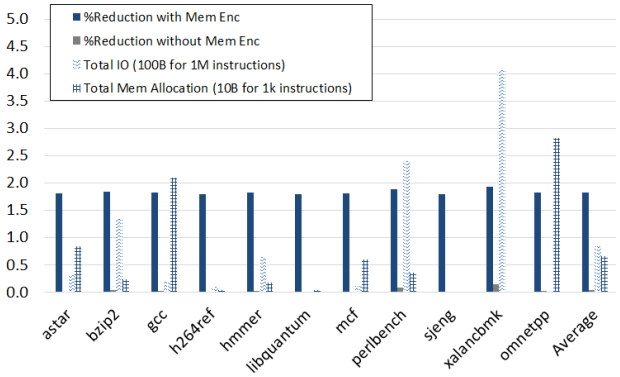

Figure 4: SeM %perf reduction, memory allocations and I/O per instruction, for SPEC CPU 2006

7. RELATED WORK

Proposed "Software based" solutions for running workloads securely (e.g., [2, 6, 14, 24, 25]) assume that both the hardware and at least some of the platform's software (e.g., hypervisor, VMM, OS) can be trusted. They typically comprise additional or modified software. "Hardware based" ones (e.g., [1,3,5,12,13,18,20,21,22,28], only assume that certain platform hardware can be trusted, and typically comprise both additional hardware and some software.

Some **software based** solutions use a trusted hypervisor [14]. [24] provides separation among applications (on a trusted OS). [2,25] are trusted VMM based solutions. Overshadow [2] provides a comprehensive solution for protecting a secure process running on an untrusted OS, and [6] suggests recompiling the OS while inserting hardware abstraction layer code. Software based solutions (including hypervisor/ VMM based ones) provide many insights and tools, but they are inherently susceptible to attacks on or by an untrusted service provider; thus we differ critically in the assumed threat model. There is also a major opportunity to reduce the inevitable performance overhead by hardware.

Hardware based solutions, among them SeM, require hardware modifications, but are generally more capable. Most provide secure compartments. E.g., Bastion [20] is a hardware attested software solution, protecting a VM from software and hardware attacks. Finer protection granularity (as in SeM) obviates the need to keep an entire OS for each secure program, requiring less code in the trusted code base (TCB), thereby making it more reliable (fewer bugs). [1] was the first significant hardware solution protecting a secret program, but its memory encryption techniques limited performance, and some vulnerabilities where subsequently pointed out. [3,18] improved performance and solved the vulnerabilities of [1], but performance still degrades. Some works rely on the verified state of the machine by secure boot [28] or trusted BIOS code [35]. However, OS and driver updates complicate state verification, as each update leads to a new state. [5] protects only a single process at any given time. [13] isolates at page granularity, requiring the programmer to specify the protected areas. We differ from these in the threat model, usage assumption and/or performance.

Intel's SGX [22] addresses the same threat model as SeM's. It allows an unprotected process to instantiate a small secure memory region (an *enclave*). Code and data within the enclave are protected from software and hardware attacks. SGX2 [30] adds dynamic memory allocation, enclave runtime permission management, and lazy loading of code into an enclave. Operations inside an enclave are limited, so overheads caused by entering/exiting an enclave limit its performance. SGX's SDK directly targets applications developed for it, and these cannot run elsewhere. Software extensions to SGX enhance its applicability up to running some unmodified binaries, but performance is still limited. These were discussed in Sec. 1.

8. CONCLUSIONS

The Secure Machine (SeM) is an extended CPU architecture that uses a novel hardware based security management unit (SMU) and a software tool, enables running a program securely even on a platform with unchanged and untrusted OS, Hypervisor, VMM, and hardware other than the CPU chip. Existing binaries are automatically instrumented to run on SeM as part of the submission to the secure cloud, requiring no programming efforts. SeM-Prepare does this by analyzing the binaries, statically linking external libraries, and adding wrapper functions for memory allocation and required system calls, and finally encrypting and signing. This essentially allows running any application (new or existing) on SeM.

SeM reduces performance by at most 2% relative to no security at all, and 95% of the reduction stems from memory encryption and authentication, which are not unique to SeM.

The basic SeM architecture is extendable to permit parallel workloads (multi-thread, -core and -computer) by adding support for dynamic process allocation and process migration, some additional functionality to the SMU, and supporting SMU-SMU communication using content encryption and SMU identity authentication. This will be reported elsewhere.

Jointly considering security, performance, and backward compatibility, we believe that SeM constitutes a major step towards widely usable secure computing on untrusted platforms. Topics for further study include direct hardware support for secure I/O: storage, network and RDMA.

References

[1] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, 2000.

[2] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. a Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in ASPLOS'08, 2008.

[3] G. E. Suh, C. W. ODonnell, and S. Devadas, "Aegis: A single-chip secure processor," IEEE Des. Test Comput., vol. 24, no. 6, 2007.

[4] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," *Proc. 11th USENIX Conf. Oper. Syst. Des. Implement.*, vol. 33, no. 3, pp. 267–283, 2014.

[5] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for Protecting Critical Secrets in Microprocessors," *Proc. 32nd Annu. Int. Symp. Comput. Archit.*, vol. 0, no. C, 2005.

[6] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual Ghost: Protecting Applications from Hostile Operating Systems John," in ASPLOS 2014.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *PLDI,* 2005.

[8] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, " System V Application Binary Interface, AMD64 Architecture Processor Supplement", https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf, 2013

[9] O. Shwartz and Y. Birk, "SDSM: Fast and scalable security support for directory-based distributed shared memory," *Proc. 2016 IEEE Int. Symp. Hardw. Oriented Secur. Trust. HOST 2016*, 2016.

[10] Shuttleworth, M. (2006). Ubuntu: Linux for human beings, http://www.ubuntu.com/

[11] R.L. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun ACM, vol. 21*, pp. 120-126, 1978.

[12] A. M. Azab, P. Ning, and X. Zhang, "SICE: A Hardware-Level Strongly Isolated Computing Environment for x86 Multi-core Platforms," in *CCS'11*, 2011, pp. 375–388.

[13] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution," in MICRO'14, 2014.

[14] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "InkTag: Secure Applications on an Untrusted Operating System," in ASPLOS'13, pp. 253–264, 2013.

[15] J.L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News, vol. 34*, pp. 1-17, 2006.

[16] B. Rogers, S. Chhabra, Y. Solihin, and M. Prvulovic, "Using address independent seed encryption and bonsai merkle trees to make secure processors OS-and performance-friendly"*, in MICRO*'07, 2007.

[17] D. A. McGrew and J. Viega, "The security and performance of the Galois/counter mode (GCM) of operation," *Secur. Perform. Galois/Counter Mode Oper. (Full Version)*, pp. 343–55, 2004.

[18] P. Williams and R. Boivie, "CPU support for secure executables," in *Trust and Trustworthy Computing*, Springer, 2011, pp. 172-187.

[19] S. Checkoway, "Iago Attacks : Why the System Call API is a Bad Untrusted RPC Interface," ASPLOS'13, 2013.

[20] D. Champagne and R.B. Lee, "Scalable architectural support for trusted software," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 2010.

[21] T. Alves and D. Felton, "TrustZone: Integrated hardware and software security," ARM White Paper, vol. 3, 2004.

[22] I. Anati, S. Gueron, S. Johnson and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proc. of the 2nd Int. Workshop on Hardw. and Archit. Support for Secur. and Priv.*, HASP, 2013.

[23] V.D. Gligor, "A Note on the Denial-of-Service Problem." in *IEEE Symposium on Security and Privacy*, pp. 139-149, 1983.

[24] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trust visor: Efficient TCB reduction and attestation," *Proc. - IEEE Symp. Secur. Priv.*, pp. 143–158, 2010.

[25] K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda, "PrivExec: Private execution as an operating system service," in *Proc. - IEEE Symp. Secur. Priv.*, 2013.

[26] S. Shinde, "PANOPLY : Low-TCB Linux Applications with SGX Enclaves," NDSS, 2017.

[27] C.-C. Tsai, D. E. Porter, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. a. Kalodner, V. Kulkarni, and D. Oliveira, "Cooperation and security isolation of library OSes for multi-process applications," The 9th European Conference on Computer Systems - EuroSys '14, 2014.

[28] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, "SecureME : A Hardware-Software Approach to Full System Security," *Proc. Int. Conf. Supercomput.*, pp. 108–119, 2011.

[29] D. Genkin, A. Shamir, and E. Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis," in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2014, vol. 8616 LNCS.

[30] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave," Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, p. 10:1--10:9, 2016.

[31] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming," ACM Transactions on Information and System Security, vol. 15, no. 1, pp. 1–34, 2012.

[32] O. Shwartz and Y. Birk, "Sound Covert: A Fast and Silent Communication Channel through the Audio Buffer." Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on. IEEE, 2017.

[33] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, M. L. Stillwell, D. Goltzsche, D. Eyers, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in OSDI, 2016, pp. 689–704.

[34] Intel® 64 and IA-32 Architectures Software Developer's Manual, https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

APPENDIX: SMU Instructions

Setup and results:

- *SMU_GenKeys ()* // Generate a pair of public-private keys (PbK, PrK). Optimization: prepare beforehand and store in a FIFO. Return Value: PbK, signed by the SMU.
- *SMU_StoreKeys (PbK, enc[SymK && HashK], phash, FirstLEP)* // RSA decrypt enc[Skey && Mkey,by PbK] using PrK and store the keys, phash, and FirstLEP in a table entry.
- *SMU_SetPID (phash)* // If no table entry with *phash* exists, report an error; else, set the current PID in the found entry. Destroy any remnants of an existing SMU entry with the same PID, and purge such blocks in the cache.
- *SMU_GetResults (PID, rand)* // Returns PID's error status padded by rand and signed & encrypted w. Skey & Mkey.

Context switch:

- *SMU_EvictContext ()* // Stores the content of the SMU Sealed Storage in the secure process memory.
- *SMU_RestoreContext (PID)* // Loads the content of the SMU sealed storage from the secure process memory.

New load/store instructions: (Only run if *Auth = True*)

- *SMU_StoreNA (address, data)* // Stores data into a memory block regardless of the block's Auth status, and resets its Auth bit.
- *SMU_LoadNA (address)* // Loads data from a memory block whose Auth bit is False.
- *SMU_InitA(addr, size)* // Fills a memory block with '0's regardless of its Auth status, and sets its Auth bit. Used along with write-no-allocate
- *SMU_syscall (argnum)* // calls a system call, and leaves *argnum* register-arguments in place on mode change