Applying the Principle of Least Privilege to System Management Interrupt Handlers with the Intel SMI Transfer Monitor

Brian Delgado
Intel Corp
USA
brian.delgado@intel.com

Tejaswini Vibhute*
Intel Corp
USA
tejaswini.vibhute@intel.com

Karen L. Karavanic
Portland State University
USA
karavan@pdx.edu

ABSTRACT

Recent years have seen a growing concern over System Management Mode (SMM) and its broad access to platform resources. The SMI Transfer Monitor (STM) is Intel's most powerful executing CPU context. The STM is a firmware-based hypervisor that applies the principle of least privilege to powerful System Management Interrupt (SMI) handlers that control runtime firmware. These handlers have traditionally had full access to memory as well as the register state of applications and kernel code even when their functionality did not require it. The STM has been been enabled for UEFI and, most recently, coreboot firmware, adding protection against runtime SMM-based attacks as well as establishing a firmware-based Trusted Execution Environment (TEE) capability. We provide a detailed overview of the STM architecture, evaluate its protections, and quantify its performance. Our results show the STM can protect against published SMM vulnerabilities with tolerable performance overheads.

KEYWORDS

firmware, virtualization, system management mode, smm, stm, smi

ACM Reference Format:
Brian Delgado, Tejaswini Vibhute*, and Karen L. Karavanic. 2020. Applying the Principle of Least Privilege to System Management Interrupt Handlers with the Intel SMI Transfer Monitor. In *Hardware and Architectural Support for Security and Privacy (HASP '20), October 17, 2020, Virtual, Greece.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3458903.3458907

1 INTRODUCTION

System Management Mode (SMM) is a highly-privileged CPU mode present on Intel® and AMD® x86 processors. The firmware run in SMM performs many important operations at runtime such as low-level hardware control, handling memory errors, and controlling performance. This code is loaded by computer manufacturers. End users are not able to inspect its operation or control its accesses on most production platforms. SMM code can access memory and registers belonging to the operating system, Measured Launch Environment (MLE)[1] or applications. SMM's privileged and isolated environment makes it a useful mechanism for a variety of critical system tasks.

Unfortunately, the same features that make SMM an effective mode for critical system tasks also make it an attractive attack surface to deploy malicious code due to its broad access privileges over system memory and registers. Compromised SMM code has the ability to locate user secrets in memory or register contents. SMM compromises have occurred using techniques including buffer overflows [28], SMM call outs [15], cache attacks [34], and improper system configurations [18]. As SMM can be used in some implementations to update the BIOS SPI chip [14], attackers could also achieve persistence by injecting malicious code into the BIOS SPI chip which would be loaded upon successive reboots. To reduce SMM's privileges, UEFI memory protections were added that limit SMM's ability to access hostside memory [35]. However there are no fine-grained controls over these accesses and other resources such as registers are still potentially vulnerable.

Addressing the potential misuse of SMM is important to protect the system functionality that uses it. In addition, recent research has proposed and explored novel uses of SMM for protecting server platforms. Methodologies have been developed for SMM-based runtime integrity measurement [2, 8, 33] and for Trusted Execution Environments or TEEs [25]. Proposers of such innovative approaches must answer the question: Can we trust SMM?

To more thoroughly address the potential vulnerability of SMM, Intel created the SMI Transfer Monitor (STM) [12]. The STM[2] is an SMM-based hypervisor that constitutes the highest privilege layer on the platform. It virtualizes System Management Interrupt (SMI) handlers and restricts their access to specified resources based on a protection policy. The STM can defend against SMI handler accesses to particular MSRs, memory ranges, IO Ports, or PCI/PCIe devices. This creates a barrier between SMM and the host software.

The STM software architecture is designed to allow a hypervisor to protect itself from SMI handler accesses over its resources. The Intel TXT launch process allows detection of changed hypervisor files prior to launch, however, SMI handlers could potentially be compromised after launch. The STM provides a way to mitigate runtime attacks from compromised SMI handlers extending the protections provided to the hypervisor.

The BIOS and MLE must opt-in to the STM before it can be activated. Without this, the STM would not be operational. The Opt-In requirement resolves a tension between a MLE that does not fully

*Performed some of this work while at Portland State University.

HASP '20, October 17, 2020, Virtual, Greece

ACM ISBN 978-1-4503-8898-6/20/10…$15.00
https://doi.org/10.1145/3458903.3458907

[1] The MLE is host software (e.g. hypervisor or operating system) that goes through a measured launch, for example with Intel Trusted Execution Technology (TXT).

[2] The STM is also referred to as a dual-monitor, peer monitor, or SMM Transfer Monitor in various literature.

trust the runtime SMI code and SMI handlers that require certain resources in order to operate. The STM allows the SMI handlers to access all of the resources that the BIOS statically requested at boot time. The STM provides this list to the MLE so that it can inspect the set of resources prior to its launch. The MLE may tear down if it does not approve of the list. The MLE provides a set of resources to the STM that it wants protected from the SMI handler. The STM applies these restrictions on the SMI handler to enforce the protection policy. The MLE can issue subsequent modifications of the protection policy at runtime, however, the BIOS cannot.

The STM can be viewed from two perspectives: the STM CPU functionality as described by the Intel Software Developer Manuals [13] and the STM Software Architecture Specification [12] that describes one implementation of an STM software stack. This software stack specifies an interface between the MLE and the STM to provide commands, along with specific data structures.

In this paper we examine the configuration, functionality, and performance of STM from the perspectives of the CPU feature and software implementation. We provide the first published treatment of the design and architecture of the STM, summarize its components, and demonstrate its ability to de-privilege SMI handlers. We also highlight the STM's firmware-based TEE capability.

The key contributions of this paper are:

(1) STM security policies that defend against known vulnerabilities.
(2) A methodology for evaluating STM protections.
(3) A preliminary analysis of STM performance overhead.

The rest of the paper is organized as follows: Section 2 provides background on SMM and virtualization. Section 3 provides an overview of the STM architecture. We discuss our STM setup in Section 4. We demonstrate examples of the STM's protections in Section 5. We provide performance details on the STM including its impact on SMI latency and application performance in Section 6. We discuss related work in Section 7. We provide our conclusions in Section 8.

2 BACKGROUND

In this section, we provide background on SMM and virtualization. These may be skipped by readers already familiar with these topics.

2.1 System Management Mode Overview

SMM is a highly-privileged operating CPU mode present on Intel and AMD processors. It was originally added to the 386SL processor [13]. On Intel platforms, this mode is used to perform platform functions related to flash updates, power management, handling memory errors, system hardware control, device emulation, managing UEFI variables, among many other uses [21, 27]. SMM code resides in a hardware-protected memory region called SMRAM. A range register called the SMM Range Register (SMRR) protects SMM memory from being read or modified by code executing outside of SMM. SMRAM is also protected from DMA access via DMA protection over the TSEG memory region that, by convention, matches the SMRR. CPU threads enter SMM by asserting an SMI. An SMI is the highest privilege interrupt and has higher priority than any other interrupt even Non Maskable Interrupts (NMIs) [13].

When an SMI is generated and there is no active STM, the CPU threads will transition to SMM entry code at the next instruction boundary. They may enter SMM in 16 bit mode, transition to 32 bit mode, and then to 64 bit mode. The interrupted register state of each CPU thread is stored in the SMRAM Save State Map [13]. SMM Dispatcher code locates the appropriate SMI handler and transfers one CPU thread to the SMI handler while other CPU threads wait. Figure 1 provides an overview of this flow.

When in SMM, the SMM page tables are operational and have traditionally been configured to allow SMI handlers to access host-side memory. These SMM page tables are not affected by how the host-side hypervisor has configured its page tables, for those are only operational when the CPU threads are not in SMM.

Once the work of the SMI has been completed, all of the CPU threads issue an RSM (resume) instruction. Upon exiting SMM, the saved processor state is restored from the SMRAM Save State Map and the CPU threads start executing from where they were interrupted.

Recent research has explored novel security-related uses of SMM. Examples include EPA-RIMM [8], SPECTRE [37], HyperSentry [2], and HyperCheck [33] which use SMM to perform runtime integrity measurement for rootkit detection. EPA-RIMM extended SMM to utilize all available cores of a multicore architecture for runtime integrity measurement.

2.2 Virtualization

Virtualization creates an abstract hardware environment for each virtual machine by software called the hypervisor. The hypervisor has direct access to the hardware and the ability to allocate resources to its virtual machines. Intel's Virtualization Technology (VTx) provides CPU-level support that simplifies the enabling of operating systems to support virtualization and also reduces performance degradations. As the STM is a VTx hypervisor, we provide background on the necessary relevant VTx details.

To distinguish the hypervisor context from that of the virtual machine, VTx establishes a Root (hypervisor) and Non-Root mode (virtual machine). Only the Virtual Machine Extensions (VMX) Root mode is able to perform privileged VTx instructions. The hypervisor can optionally allow virtual machines to directly access hardware. Certain instructions executed in the virtual machine are trapped and handled by the hypervisor. During this handling, the processor transitions from VMX Non-Root mode to the VMX Root mode (a "VMEXIT" event). After the hypervisor assists the operation, it can

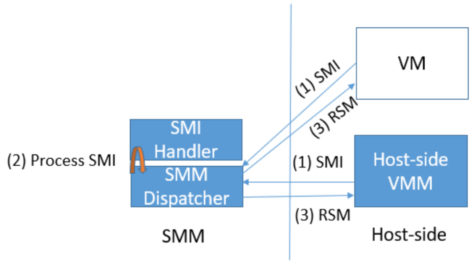


Figure 1: SMI Processing without STM

resume the guest by transitioning back to VMX Non-Root mode via the VMRESUME instruction. This is a "VMENTRY" event.

The hypervisor maintains most CPU state in a Virtual Machine Control Structure (VMCS). This VMCS has a host and a guest portion for the hypervisor and guest CPU state, respectively. The hypervisor will allocate one VMCS per virtual CPU in the VM. These VMCSs allow the hypervisor to manage the VM's privileges and execution. VTx provides bitmap-based controls that govern whether accesses to particular system resources such as IO Ports, MSRs, memory ranges, and PCI devices will trap to the hypervisor. The hypervisor can perform these operations on behalf of the VM and inject the results into the virtualized guest's state, or it can intercept the operation. A virtual machine can issue a request to its hypervisor using a VMCALL. These instructions cause a VMEXIT from the VM so that the hypervisor can perform an operation on behalf of its guest. When the hypervisor needs to turn off virtualization, it issues a VMXOFF instruction.

3 STM ARCHITECTURE

In this section, we describe the architecture of the STM reference implementation. We provide details on its provisioning, launch, runtime operation, and teardown.

3.1 STM Overview

When the STM and MLE are operational, there are two hypervisors present on the system. The STM virtualizes the SMI handler and the MLE virtualizes guest operating systems. From a privilege perspective, the STM is the higher privilege level as the MLE issues VMCALLs to the STM, just as a VM would issue to its hypervisor. This is shown in Figure 2. The STM enforces restrictions over the SMI handler's accesses to MLE resources to apply the principle of least privilege. The STM is only operational when it has been launched by the MLE and one of the following occurs:

(1) An SMI is generated.
(2) An SMI handler causes an exception during SMI processing.
(3) The MLE or SMI handler issues a VMCALL.

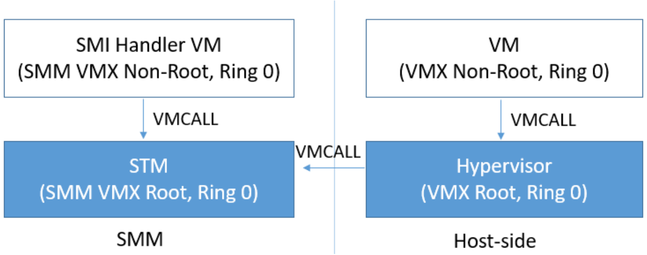


Figure 2: VMCALLs with STM

3.2 STM Threat Model

The STM's threat model assumes that attackers can exploit vulnerabilities in SMI handlers at runtime to compromise the MLE. The MLE is assumed to be trusted to launch the STM, configure it, and tear it down. However, care must be taken by the STM to avoid compromises of its integrity by either the SMI handlers or the MLE. The STM protects its own memory region from the SMI handler via page tables and relies upon SMRR and TSEG protections to guard its memory from host-side code. Technologies such as Intel TXT and Intel Boot Guard [22] can be used to gain assurance that the SMI handlers are initially unmodified. These technologies should also be used to verify the integrity of the MLE before launching it. Thus, ensuring that the boot time static integrity of these various components is maintained. The STM endeavors to provide assurance at runtime that the set of accesses performed by the SMI handler are in accordance with the prescribed policy. By applying the principle of least privilege to SMI handlers, the risk of compromised SMI handlers can be mitigated.

The STM applies a traditional security design approach that adds a more trusted layer below code that is not fully trusted. The STM hypervisor is smaller than traditional hypervisors. For example, the STM codebase is roughly 10% the size of Xen 4.12 in lines of code based on our CLOC [6] analysis. Compared to the set of SMI handlers on a typical platform, the STM accesses fewer platform resources and does not perform a varied set of operations. The STM is cryptographically measured prior to launch by the TXT ACM module to determine if the STM module has unexpectedly changed. Unlike OEM SMI handlers, the STM is open-source and available for inspection.

The STM architecture does not allow the host-side hypervisor to make changes that impact the security of the SMI handlers or STM. A compromised host-side hypervisor would only be able to disable the STM and remove the enhanced protections over SMI handler accesses to host-side resources provided by the STM. If the attacker had compromised the hypervisor and disabled the STM, the hypervisor would already be compromised and disabling STM protections would not result in additional practical benefit. In this scenario, the hypervisor would perform like a legacy hypervisor that had not performed an Opt-In to the system and would continue functioning without STM protections.

3.3 STM Provisioning

The STM requires BIOS provisioning before it can be launched by the MLE. The BIOS must reserve a memory region in SMRAM called MSEG ("Monitor Segment"). It then must copy the STM binary into MSEG from a firmware volume and opt-in to the STM by setting bit 0 of IA32_SMM_MONITOR_CTL MSR (0x9B) and also recording the location of MSEG by programming the address into bits 31:12 of this MSR.

The BIOS also creates a TXT_PROCESSOR_SMM_CPU_DESCRIPTOR data structure in memory that provides configuration information that the STM refers to while configuring the SMI handler. A key element of this data structure is the BIOS Required Resource list which enumerates the set of resources needed by the SMI handlers at runtime.

STM Opt-In for the MLE has a static portion in which the MLE header is updated during compile time to indicate support for an STM. For a TXT launch, the SINIT ACM module will consult this bit to determine if the MLE supports an STM.

3.4 STM Launch

The STM launch is performed by the MLE via the following steps:

(1) For TXT scenarios: The MLE must indicate support for the STM in the MLE header. This allows pre-launch software, e.g. trusted boot, to trigger configuration and measurement of the STM when it issues the GETSEC[SENTER] instruction. This instruction masks (disables) SMIs until the STM re-enables them. SMI masking allows the STM to be launched without SMIs occurring during the STM launch process. The measurement of the STM is stored in TPM PCR 17 and provides an ability to inspect the STM's hash before proceeding. The "SINIT" module[3] also creates initial page tables for the STM to access the first 4GB of memory. This allows the STM to begin in 64 bit paged and protected mode.
(2) The MLE allocates a VMCS that is used only for launching the STM and issue a VMCLEAR for this VMCS to prepare it for use. It then issues a VMCALL on a single CPU thread ("InitializeProtectionVMCALL").
(3) The CPU checks the Opt-In bit of the IA32_SMM_MONITOR_CTL MSR and locates the STM from the MSEG location specified in the MSR. If the Opt-In bit is not set, the launch will not succeed.
(4) The CPU consults the STM header which is located at the base of the STM binary in MSEG and utilizes this to configure the STM entry environment. The header specifies the code offset where the first STM instructions will be found.

Once the first CPU thread enters the STM, it creates an initial environment to prepare the STM for entry by all CPU threads. This includes establishing internal data structures and creating additional page tables, as needed, to address additional memory. The STM also needs to establish a VMCS for transitions to and from the MLE.[4] The STM must also establish a VMCS for the SMI handler guest.[5]

Following the STM's initial initialization, it is able to service VMCALLs from the MLE and the SMI handler. Each VMCALL brings only one CPU thread into the STM and allows other CPU threads to continue processing. In scenarios where it is desirable to send all CPU threads into the STM (for example, when establishing a new protection policy), it is the responsibility of the calling environment to rendezvous the CPU threads.

3.5 Protection Requests

3.5.1 Resource Protection. When the first thread returns from the STM to the MLE, the MLE may optionally request a list of all resources that the BIOS SMI handler requested access to via the GetBiosResource VMCALL to the STM. The MLE can then inspect the list and issue its protection request(s) via the ProtectResource VMCALL. For the STM to protect a resource, the BIOS must not have requested access to the resource in the Required Resource list. Applying protection requests before SMIs are unmasked allows the STM to enforce the protections before SMI handlers are able to be active.

The STM uses VTx permission bitmaps as the cornerstone of its ability to set policies over the set of accesses allowed to the SMI handler guest. MSR and IO Port protections are accomplished by configuring the MSR and IO bitmaps, respectively. Memory and MMIO protections are enabled by configuring the EPT for the SMI handler guest. The MLE provides the address of host physical memory pages for the STM to protect. PCI device protections are accomplished by adjusting the IO Bitmap.

The STM informs the MLE which resources it was able to protect. After the MLE has completed its protection requests, it performs a CPU rendezvous and issues the StartStm VMCALL on all CPU threads which directs the STM to launch its SMI handler guest, turn SMIs back on and resume operation.

3.5.2 SMRAM Save State Map Protections. The MLE can request the STM to protect the MLE's and virtual machine's registers in the SMRAM Save State Map from being viewed or modified by the SMI handler via the Protected Domains feature. To accomplish this, the MLE executes a ManageVmcsDatabase VMCALL to request the STM to store a VMCS pointer along with one of the following policy options [12].

(1) "Unprotected": Present all register state to the SMI handler and do not revert any modifications
(2) "Integrity Protected OUT/IN": Preserves integrity of register state. It only allows register modifications by the SMI handler to IO ports listed in the IO Port trap list and EAX.
(3) "Fully Protected OUT/IN": Preserves confidentiality and integrity of register state. This option only presents minimal registers to SMI handler and rolls back changes to other registers.
(4) "Fully Protected": Preserves confidentiality and integrity. The policy precludes IO IN/OUT operations. No register modifications by the SMI handler are allowed.

3.6 SMI Processing with an STM

When an STM is present, the STM receives all the CPU threads following an SMI as shown in Figure 3. The STM receives control in 64 bit mode with paging enabled. (Figure 1 provides the flow for the non-STM scenario described in Section 2.1.)

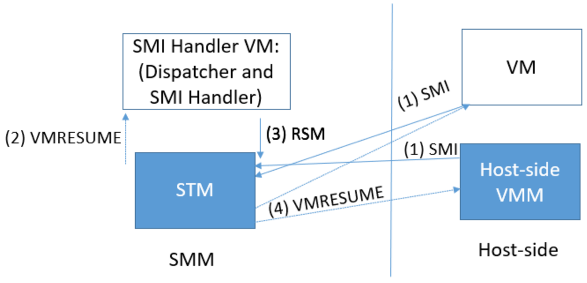


Figure 3: SMI Processing with STM

The STM uses the VMCS associated with each CPU thread to retrieve and set state for the interrupted host-side context across all the CPUs instead of the traditional SMRAM Save State Map. The STM populates the SMRAM Save State Map for each CPU thread

[3] An authenticated code module ("ACM") used by Intel TXT
[4] This is termed the "SMI VMCS" in the STM reference implementation as it represents the inbound path for an SMI.
[5] This is termed the "SMM" VMCS in the STM reference implementation.

based on their corresponding host-side VMCS. This populated SMRAM Save State Map can be accessed by the SMI handler during its execution. If the VMCS is for one of the Protected Domains (Section 3.5.2), the registers may be scrubbed based on the policy before the SMI handler VM gets control and also roll back changes to the Save State Map. At this point, the STM loads the SMI handler VMCS and is ready to invoke the SMI handler.

The STM sends all the CPU threads in the SMI handler VM. The STM then assigns only one CPU thread to perform the actual SMI handler task in the SMI handler VM while the other CPU threads are sent to execute a spinloop in the SMI handler VM until the actual work CPU thread completes its execution. Once the work thread finishes its execution, it and the other CPU threads execute the RSM instruction to VMEXIT from the SMI handler VM into the STM. The STM returns control to the interrupted context via a VMRESUME into the MLE or its guests. The STM may also receive VMEXITs from its SMI handler guest due to the SMI handler accessing IO Ports, MSRs or memory beyond the usual set of VMEXIT conditions such as executing exiting CPU instructions among other causes.

3.7 STM Teardown

The MLE can tear itself and its guests down by executing the VMXOFF instruction. The STM will not execute VMXOFF as this must be done by the MLE. The IA32_SMM_MONITOR_CTL MSR controls whether SMIs will be unmasked when the MLE executes VMXOFF.

4 EXPERIMENT INFRASTRUCTURE

We deployed the STM on a Minnowboard Turbot dual core with Silvermont Atom processors system with 2 GB RAM, with MinnowMax 1.01 UEFI firmware, the Xen 4.11 hypervisor, and Ubuntu 14.04 with kernel 4.11 for Xen's Domain 0. The Minnowboard Turbot is an open developer platform that allows modification of the firmware to support the STM. In this section we describe modifications to enable the STM on this platform. We used publicly-available Xen code to launch the STM and define security policies.[6]

4.1 STM enabling and Opt-In

Minnowboard Max's firmware code is based on an open-source UEFI EDK-II implementation.[7] But this implementation does not support STM in its entirety. Hence, we first built the STM obtained from its repository[8] in isolation and then compiled EDK-II along with the newly built STM binary. This process enables adding STM support to the BIOS when combined with adding the BIOS support modules from the repository as well.

4.2 STM Launch

The Minnowboard Turbot does not support TXT and hence the boot process of our setup does not go through the SINIT-based measurement which would occur with a true TXT-based MLE.

We perform the following steps to launch the STM:

1. *BIOS STM Opt-In*: The Valid bit of the `SMM_MON_CTL` MSR should be set and Intel VT-x should be enabled.
2. *Allocate a temporary VMCS per logical CPU*: The Xen patches create a temporary VMCS during the initial STM launch VMCALL. This VMCS is used only for the very first VMCALL on each CPU.
3. *Initiate InitializeProtection VMCALL on only one logical processor*: On this VMCALL, the STM initializes the BIOS resource list and sets up the environment for running the SMI handler as its guest. This VMCALL is executed on logical CPU 0, the BSP.
4. *Execute ProtectResources VMCALL*: In the ProtectResourceVMCALL, we provide a set of protection policies for the STM to enforce. We describe these policies in Section 5.
5. *Execute StartStm VMCALL on all the logical CPUs*: On this Xen VMCALL, the STM launches the SMM guest. The VMCALL for this step must be executed on all the logical CPUs.
6. *Manage VMCS Database VMCALL*: This VMCALL allows fine-grained protection policies over SMI handler access to host-side CPU register state. This can preclude SMI handlers from viewing or modifying CPU registers from the interrupted context.

Each of the above steps are performed only if their previous step was successfully executed. If any of the step fails then changes are rolled back and the STM is not enabled [32].

When destroying its VMs, the MLE may remove the corresponding VMCSs from the VMCS Database. To do so, the MLE should invoke the ManageVmcsDatabase VMCALL with the remove option and the pointer to the VMCS to be removed. The patches implemented this in Xen when destroying the VM. To teardown the STM, the Stop STM VMCALL is invoked in vmx_cpu_down before VMXOFF is issued.

5 STM PROTECTION EVALUATION

In order to evaluate the ability of the SMI Transfer Monitor to protect SMI handlers, we added and removed policies to/from the STM, and verified the results. To enable these tests we implemented an evaluation tool, `Sandboxed`, to trigger SMI handler accesses. In this section we list the vulnerabilities we address, then describe `Sandboxed`, then detail our study results.

5.1 Known Vulnerabilities Tested

Security researchers have demonstrated multiple methods of attacking a running operating system from SMM. One method, Thinkpwn, demonstrates that a vulnerable SMI handler can be co-opted to execute attacker-provided code from a pointer in a CPU register [5]. Another example, LightEater, demonstrated SMM's ability to extract secrets from the Tails OS [15]. Schiffman, et al. demonstrated an SMM-based keylogger that intercepted USB events before they were delivered to the OS kernel [26]. SMMDecoy provides an example of an SMM-based USB or PS/2 keyboard injection attack [19].

Some attacks have leveraged improper platform configurations, for example, improperly allowing SMM's memory to be compromised by not setting lock bits that protect it [9]. Rutkowska, J. demonstrated an attack that leveraged CPU cache poisoning to write into SMRAM memory [34]. Intel later added the SMRR feature to address this issue [29]. SMM has also been reported to be potentially vulnerable to Spectre side channel attacks [31]. UEFI

[6] https://github.com/PPerfLab/xen/tree/stm-optin
[7] https://github.com/tianocore/edk2
[8] https://github.com/jyao1/STM

added a SMM Communications Buffer to provide a more secure method of exchanging data between SMM and non-SMM code. This can avoid attacks that trick SMI handlers into modifying their own data improperly or corrupting host memory [36].

STM protection policies are designed to defend against classes of attacks in which SMI handlers can be coerced to improperly access host-software resources or maliciously target them. We focus our evaluation on four previously published vulnerabilities: SMM-installed hypervisor rootkits, ThinkPwn, LightEater, and keylogger/keyboard injection attacks.

5.2 STM Evaluation Tool: Sandboxed

We created a tool called `Sandboxed` to test the effectiveness of the STM policies. `Sandboxed` has four components: `CloneDriver` (SMI handler), `SmiGen` (Linux kernel module), `Harness` (Ring 3 application), and `HackMe` (Linux kernel module). The `CloneDriver` takes commands from `SmiGen` that direct its accesses to memory, MMIO, MSR and IO ports. The `HackMe` kernel module provides a target for the CloneDriver to compromise. It allocates a kernel page and continuously reads and writes to it. It can also read an MSR value and poll an IO port like a PS/2 keylogger. We also use `HackMe` to check for signs of attack. Figure 4 shows an example of the flow.

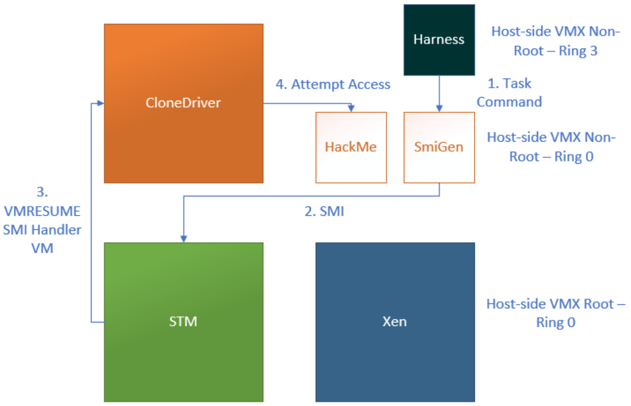

Harness
Host-side VMX Non-Root – Ring 3
CloneDriver
4. Attempt Access
1. Task Command
HackMe
SmiGen
Host-side VMX Non-Root – Ring 0
3. VMRESUME SMI Handler VM
2. SMI
STM
Xen
Host-side VMX Root – Ring 0

Figure 4: Sandboxed Access Flow

The `CloneDriver` and the `SmiGen` communicate via commands specified in registers. The `Harness` invokes the `SmiGen` with different commands to test the applied policies with commands constructed as per *Task* specification. `SmiGen` is the kernel module that generates the SMI for `CloneDriver` with specific commands. This module takes a *Task* data structure as input. The fields of this structure and the values that each field can take are detailed in **Table 1**.

5.3 Protection Against Attacks on Memory

We perform two evaluations for the STM's memory protection capabilities.

5.3.1 Hypervisor Rootkit Injection and ThinkPwn-style SMM call outs.

The first experiment ensures that the Xen hypervisor address space is not writable by a malicious SMI handler. Such a handler could install and launch rootkits if it were able to modify the memory [20]. While the attacker, in this example, has SMM access and could run a rootkit from SMM, placing a rootkit in the hypervisor from SMM allows the rootkit to operate for longer periods of time without preempting the entire system.

Table 1: Task Structure

Field	Description
Resource Type	MSR, IO Port, Memory Region
Resource	The name of the resource
Operation	Read or write
New Value	Value to set to the specified resource

We specified a policy to protect Xen's kernel address space from being written and executed by the SMI handler. Thus, we set the `RWXAttribute` field of the `STM_RSC_MEM_DESC` structure to preclude Write and Execute access.

Task fields set as: [Resource Type: "Memory", Resource: "Virtual address of Xen Memory Page", Operation: "Write", New Value: "abcdefgh"].

With this STM policy, the `CloneDriver` was unable to modify the specified memory page and would thus be unable to install the rootkit. The same policy helps defend against SMM Callout attacks such as ThinkPwn, in which SMM code executes non-SMM code with SMM privileges [23]. We simulated the ThinkPwn attack with a non-STM BIOS and with Page Table Isolation disabled. On such a BIOS, the `CloneDriver` was able to call into an arbitrary function in the Xen kernel space. However, with the above Xen kernel memory protection policy with STM, such an attack failed with STM flagging the attempt as an EPT violation.

5.3.2 Attempting to steal memory secrets from SMM, similar to Lighteater.

The second experiment attempts to steal a secret from the HackMe kernel module's memory similar to the LightEater attack [15]. We make a protection request for host memory that contains the memory page allocated by `HackMe` and invoke the `Harness` with the specified *Task*. Upon receiving the task via SMI, the `CloneDriver` converts the given virtual address to the physical address of the page and attempts to read it.

Task fields set as: [Resource Type: "Memory", Resource: "Virtual address of allocated kernel page", Operation: "Read"].

We tested this policy by attempting a read access to the allocated kernel page. Since, we have specified a kernel page protection policy as part of ProtectResource VMCALL, the STM initiated a protection exception indicating the `CloneDriver` tried to violate a protection policy.

5.4 Control-based attacks

MSRs can be an important source of information for a malicious SMI handler. Such a handler could read or write to certain MSRs that may be used by the host software for effective functioning. For example, attackers could attempt to hook MSR `IA32_SYSENTER_EIP` to alter the control flow [11].

We specified a policy to protect MSR `IA32_SYSENTER_EIP`. We set the `ReadMask` and `WriteMask` fields of the `STM_RSC_MSR_DESC` structure to 0xFFFFFFFF for MSR IA32_SYSENTER_EIP, indicating the SMI handler is not allowed to read nor write this MSR.

Task fields set as: [Resource Type: "MSR", Resource: "IA32_SYSENTER_EIP", Operation: "Write", New Value: "0xf"].

To demonstrate MSR protections on an MSR, we tried to modify the value of `IA32_SYSENTER_EIP` from the `CloneDriver` on the STM-enabled system. We invoke the `Harness` with the specified *Task* fields. When the `CloneDriver` tries to write to the MSRs, the write was trapped by the STM and the STM generated a resource protection violation.

5.5 PS/2 keyloggers

SMMDecoy provides an example of an SMM-based USB or PS/2 keyboard injection attack that generates fake user authentication credentials [19]. If these credentials are later used, the inference is that the credentials were gathered by a keylogger on the system. For the PS/2 flow, the flow relies upon writing to IO port 0x60 and 0x64 to inject data into the keyboard buffer.

To defend against this type of attack, we set a resource policy to intercept writes to 0x60 and 0x64.

Task fields set as: [Resource Type: "IO Port", Resource: "0x64", Operation: "Write", New Value: "0xf"].

We evaluated the policy on a write to port 0x64. Upon attempting this write, the STM intercepted the operation and triggered an exception.

5.6 Discussion

We designed and implemented an evaluation infrastructure, then used it to show that the STM successfully defended against four types of SMI handler-based attacks: hypervisor rootkit injection, memory stealing, MSR-based control attacks, and keyboard attacks. We note that our experiments implemented and tested each policy in isolation. However, there may be scenarios in which certain SMI handlers may need access to a given set of resources while other SMI handlers should be precluded these accesses. With the STM, it is also possible to create additional VMs such that independent protection policies can be applied. More sophisticated scenarios, for example, only selectively allowing particular accesses based on an analysis on state, would require additional enabling in the STM. Direct testing of these scenarios is out of the scope of this paper.

6 PERFORMANCE

The time spent in SMM needs to be carefully managed as it preempts all other processing. Intel's BIOS Bits utility provides a guideline that warns if SMIs take over 150μs. An empirical bound has been established by Delgado et al., who showed adverse effects on system correctness and performance impacts with time spent in SMM exceeding 1.5ms [7]. As SMI processing time preempts other usages, carefully managing SMM time becomes important.

To measure the performance impact of the STM, we evaluate the performance overhead of two SMIs for both STM-disabled and STM-enabled configurations. Our experiments use the configuration described in Section 4.

We generated SMIs from the SmiGen kernel module and varied the value written to port 0xB2 to trigger three SMIs for benchmark purposes. The set of SMIs on a given system can vary significantly, therefore we focus on three examples as follows: 1. A "Null" SMI highlights the impact of the context switches between the host-side, the STM, and an SMI handler, 2. The ACPI Enable SMI shows the impact of the STM on an actual SMI, and 3. A SHA hashing example to provide an example of a workload inside an STM TEE.

We measure the SMI latency as observed by the host software. We started by recording the time required to enter and exit SMM on an STM-enabled system.

We first evaluate the "Null SMI" that does not perform meaningful work in SMM, however exercises the full code path through the STM and the SMI handler before returning to the OS. On the Minnowboard Turbot dual core the SMI latency was 168.4μs with the STM disabled and 214.2μs with the STM enabled. Thus, the STM adds roughly 27.2% to the total SMI processing cost. We then analyze the performance of the "ACPI Enable" SMI. This SMI handler performs multiple IO Port reads and writes. Based on our measurements, we observed that the STM added 26.2% to the SMI latency.

To further analyze the STM latency, we leveraged the STM's performance gathering capability. We also added tracepoints in the code path of the `Sandboxed` tool to evaluate the end-to-end SMI performance. Figure 5 shows the SMI flow under an STM. We initiated a 1K memory hash SMI from the `Sandboxed` tool's `SmiGen` module. Upon receiving this request, the `CloneDriver` performs a SHA256 measurement of the specified memory location. Without the performance gathering capability, this task took 243μs. While with the capability enabled, the task was completed in 253μs. Thus, the performance collection added only 10μs to the latency.

The SMI latency results for the Null SMI and ACPI Enable show that the STM introduces about 27% additional SMI latency. The use of the STM requires extra transitions through the STM itself as well as running the SMI handler in a virtual machine. The SMI time required modestly exceeds the SMI latency guideline on our test system. The total user impact is a product of time spent in SMM and the SMI frequency. Although there is no published data on SMI frequencies, we can assume that with a system with one SMI per second, the impact would be quite minimal.

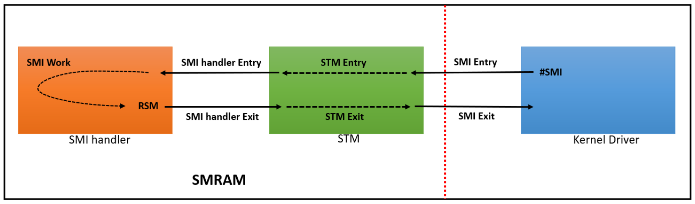


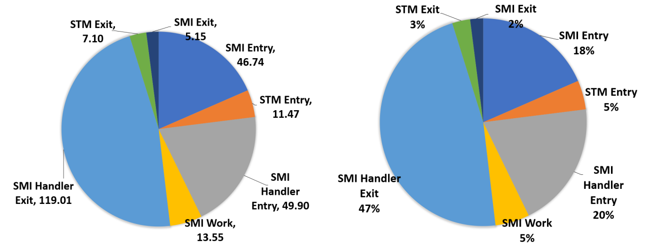
Figure 5: SMI flow with STM - Tracing

We leveraged the following SMM performance model as described in [8] to analyze the STM latency.

$$T_m = T_{entry} + T_{work} + T_{exit} \quad (1)$$

where T_m is the total SMI latency, T_{entry} is the time to enter the SMI handler from the kernel driver, T_{exit} is the time to exit from the SMI handler and return to the kernel driver and T_{work} is the time spent in the SMI handler working. From Figure 5,

$$T_{entry} = SMIEntry + STMEntry + SMIhandlerEntry \quad (2)$$



(a) SMI Latency: Absolute Values

(b) SMI Latency: Percentages

Figure 6: SMI Round Trip Time Analysis

$$T_{exit} = SMIhandlerExit + STMExit + SMIExit \tag{3}$$

Figure 6 illustrates the amount of time spent in each phase of transition. STM processing adds only 18.57μs which is only 8% of the total SMI latency. The maximum time is spent in transition between the SMM VM and the STM and MLE VM (kernel driver) and the STM.

7 RELATED WORK

Multiple approaches have been leveraged to improve SMI handler security. HP SureStart [4] provides a method of analyzing the control flow of SMI handlers using a co-processor to determine if they call a function with an unexpected call type signature. The approach also performs checks over SMBASE and CR3 to determine if they have unexpectedly changed at runtime. BIOS Guard [22] restricts which code modules can update the SPI chip which stores the firmware. This can help protect against injections of malicious code into the SPI chip but does not address runtime attacks on SMI handlers.

UEFI Secure Boot provides detection of modified UEFI DXE modules. These modules execute prior to the operating system boot loader. Intel Boot Guard extends these detection capabilities earlier in the reset process by anchoring to hardware, microcode, and the Boot Guard ACM [22]. Intel TXT provides the ability to measure and verify the launch of an operating system or hypervisor. It leverages Safer Mode Extensions (SMX) and an ACM to verify the kernel and initrd [10]. With these capabilities, the platform owner can determine that the measured files have not been tampered with at measurement time.

Trusted Execution Environments (TEEs) provide a mechanism to run code with the ability to maintain confidentiality and integrity [1]. Intel SGX provides a TEE capability for applications that runs the trusted portion of an application in an enclave. The untrusted portion of an application can call trusted functions in the enclave. The trusted portion can maintain its confidentiality and integrity against malicious privileged software. AMD SEV provides confidentiality of a virtual machine's memory. SEV uses a virtual machine-specific key to encrypt the memory contents [24]. Arm CPUs can feature TrustZone which divides execution contexts into Secure and Non-secure worlds to protect the confidentiality and integrity of code and data. Execution of the Secure world can be accomplished via a Secure Monitor Call (SMC). A proposal has been made to create an SMM-like capability using TrustZone [17]. While TrustZone provides the ability to isolate the Secure world from the normal world, it does not provide isolation between Secure world code. Arm Secure Partitions adds a virtualization-based "Secure Partition Manager" to isolate trusted applications in the Secure world from each other [3]. Keystone provides a TEE for the Risc V architecture [16]. This approach creates a security monitor that does not have any resource management tasks, simplifying its design. Keystone leverages RISC-V's physical memory protection (PMP) to safeguard specific memory ranges and establish its security monitor. STM PE extends the STM to be a TEE by loading a hypervisor integrity measurement agent into a VM hosted by the STM. It also supports loading a temporary code module into another virtual machine of the STM [25]. As these VMs are isolated from a potentially malicious host-software code and from each other, they benefit from confidentiality and integrity over their code and data. The EPA-RIMM tool for runtime integrity measurement leverages the STM to host a hypervisor integrity measurement agent that operates even if host software is compromised [8].

8 CONCLUSIONS AND FUTURE WORK

We have provided a detailed description of Intel's STM architecture. We provided examples of security policies that defend against specific SMM-based attacks by precluding accesses to platform resources. We find the STM provides fine-grained protections against SMI handler accesses while introducing latency costs slightly over Intel's BIOS Bits [30] guidelines yet within the empirical limit determined by Delgado et al.

The STM also provides a firmware-based TEE capability for Intel platforms. This capability provides a way to run isolated workloads away from a potentially malicious hypervisor and other executing code. As all time spent in SMM comes at the cost of host processing, these applications needs to be carefully managed. STM protections and TEE capabilities provide the means for an effective runtime defense mechanism that can be extended to run applications in an isolated context. For future work, we plan to evaluate the ability to apply more fine-grained permissions over SMI handlers by creating multiple VMs, each with custom STM policies.

ACKNOWLEDGMENTS

The material in this paper is based in part upon work supported by the National Science Foundation under Grant No. 1528185.

REFERENCES

[1] Ahmed M. Azab and et al. Ning, Peng. 2014. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. ACM, New York, NY, USA, 90–102. https://doi.org/10.1145/2660267.2660350

[2] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. 2010. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/1866307.1866313

[3] Sandrine Bailleux. 2018. Secure Partitions. OSFC.

[4] Ronny Chevalier, Maugan Villatel, David Plaquin, and Guillaume Hiet. 2017. Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode. In *Proceedings of the 33rd Annual Computer Security Applications Conference* (Orlando, FL, USA) *(ACSAC 2017)*. ACM, New York, NY, USA, 399–411. https://doi.org/10.1145/3134600.3134622

[5] Cr4sh. 2016. Thinkpwn. https://github.com/Cr4sh/ThinkPwn

[6] Al Danial. 2020. CLOC. https://github.com/AlDanial/cloc

[7] Brian Delgado and Karen L. Karavanic. 2013. Performance implications of System Management Mode. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 163–173. https://doi.org/10.1109/IISWC.2013.6704682

[8] Brian Delgado, Tejaswini Vibhute, John Fastabend, and Karen Karavanic. 2019. EPA-RIMM: An Efficient, Performance-Aware Runtime Integrity Measurement Mechanism for Modern Server Platforms. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 422–434.

[9] Loıc Duflot, Daniel Etiemble, and Olivier Grumelard. 2006. Using CPU system management mode to circumvent operating system security functions. *CanSecWest/core06* (2006).

[10] Jake Edge. 2010. Enabling Intel TXT in Fedora. https://lwn.net/Articles/382077/

[11] Infosec Institute. 2014. Hooking System Calls Through MSRs. (2014). https://resources.infosecinstitute.com/hooking-system-calls-msrs

[12] Intel. 2015. SMI Transfer Monitor (STM) User Guide.

[13] Intel. 2020. Intel® 64 and IA-32 Architectures Software Developer's Manual (Vol. 3).

[14] Corey Kallenberg, John Butterworth, Xeno Kovah, and C Cornwell. 2013. Defeating signed bios enforcement. *EkoParty, Buenos Aires* (2013).

[15] Corey Kallenberg and Xeno Kovah. 2015. How Many Million BIOSes Would you Like to Infect? CanSecWest, Vancouver, Canada.

[16] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[17] Tony Lo. 2016. Tailoring TrustZone as SMM Equivalent. UEFI PlugFest. http://www.uefi.org/sites/default/files/resources/UEFI_Plugfest_March_2016_AMI.pdf

[18] John Loucaides and Yuriy Bulygin. 2014. Platform Security Assessment with CHIPSEC. https://cansecwest.com/slides/2014/Platform.

[19] Ijlal Loutfi. 2019. SMMDecoy: Detecting GPU Keyloggers using Security by Deception Techniques. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*. INSTICC, SciTePress, 580–587.

[20] Shangcong Luan. 2016. Exploit Two Xen Hypervisor Vulnerabilities. (2016). BlackHat 2016.

[21] Keith Mannthey. 2009. System Management Interrupt Free Hardware. http://linuxplumbersconf.org/2009/slides/Keith-Mannthey-SMIplumers-2009.pdf

[22] Alex Matrosov. 2017. Who Watch BIOS Watchers? https://medium.com/@matrosov/bypass-intel-boot-guard-cc05edfca3a9

[23] Alex Matrosov, Eugene Rodionov, and Sergey Bratus. 2019. *Rootkits and bootkits: reversing modern malware and next generation threats*. No Starch Press.

[24] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. 2018. A Comparison Study of Intel SGX and AMD Memory Encryption Technology. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '18)*. ACM, New York, NY, USA.

[25] Eugene D. Myers. 2018. Using the Intel STM for Protected Execution. http://www.platformsecuritysummit.com/2018/speaker/myers/STMPE2Intelv84a.pdf

[26] Joshua Schiffman and David Kaplan. 2014. The smm rootkit revisited: Fun with usb. In *2014 Ninth International Conference on Availability, Reliability and Security*. IEEE, 279–286.

[27] TianoCore. 2019. UEFI Variable Runtime Cache. https://github.com/tianocore/tianocore.github.io/wiki/UEFI-Variable-Runtime-Cache

[28] TianoCore. 2020. 8. SMRAM Overwrite in SmmVariableHandler. https://edk2-docs.gitbook.io/security-advisory/smram_overwrite_in_smmvariablehandler

[29] TianoCore. 2020. EDK II Secure Code Review Guide. https://edk2-docs.gitbook.io/edk-ii-secure-code-review-guide/code_review_guidelines_for_boot_firmware/hardware_input

[30] Josh Triplett and Bert Tripplet. 2011. BITS: BIOS Implementation Test Suite. http://www.linuxplumbersconf.org/2011/ocw/system/presentations/867/original/bits.pdf.

[31] Liam Tung. 2018. Ex-Intel security expert: This new Spectre attack can even reveal firmware secrets. https://www.zdnet.com/article/ex-intel-security-expert-this-new-spectre-attack-can-even-reveal-firmware-secrets/

[32] Tejaswini Vibhute. 2018. *EPA-RIMM-V: Efficient Rootkit Detection for Virtualized Environment*. Master's thesis. Portland State University.

[33] Jiang Wang, Angelos Stavrou, , and Anup Ghosh. 2010. HyperCheck: A Hardware-assisted Integrity Monitor. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection* (Ottawa, Ontario, Canada) *(RAID'10)*. Springer-Verlag, Berlin, Heidelberg, 158–177. http://dl.acm.org/citation.cfm?id=1894166.1894178

[34] Rafal Wojtczuk and Joanna Rutkowska. 2009. Attacking SMM memory via Intel CPU cache poisoning. *Invisible Things Lab* (2009), 16–18.

[35] Jiewen Yao. 2016. [edk2] [PATCH V2 0/6] Enable SMM page level protection.. https://lists.01.org/pipermail/edk2-devel/2016-November/004185.html

[36] Jiewen Yao, Vincent Zimmer, and Star Zeng. 2016. *A Tour Beyond BIOS Secure SMM Communication in the EFI Developer Kit II*. Technical Report. Intel.

[37] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. 2013. SPECTRE: A Dependable Introspection Framework via System Management Mode. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN '13)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/DSN.2013.6575343