Using the Intel STM for Protected Execution

Eugene D. Myers
Trust Mechanisms
Trusted Systems Research Group
National Security Agency
Fort Meade, Maryland
edm@tycho.ncsc.mil

ABSTRACT

We describe our work to demonstrate an enhanced SMI transfer monitor (STM) to provide protected execution services on the x86 platform. An STM is a hypervisor that executes in x86 system management mode (SMM) and functions as a peer to the hypervisor or operating system. The STM constrains the SMI handler, by hosting the handler in a virtual machine (VM). Otherwise, the SMI handler holds unconstrained access to the platform, which could undermine the assurance provided by D-RTM or TXT. Our STM enhancements create a protected execution capability by extending the STM to support additional VMs (PE/VM). These enhancements utilize the existing capabilities of the x86 processor and, thus, require no additional hardware. We modified an existing hypervisor integrity measurement engine to function in a PE/VM. The related discussion explains how the module can be loaded from a guest virtual machine and how page tables are used to restrict the access that the measurement engine is allowed to memory.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Security and Protection

General Terms

Design, Security

Keywords

Protected execution, SMI transfer monitor, system management mode, security, integrity measurement

1. INTRODUCTION

Protected execution (PE) provides software with the assurance that it can neither be observed nor be tampered with by the platform. This protection includes preventing the operating system and I/O devices from accessing the protected execution memory. Protected execution is not the same as secure execution. Secure execution is concerned with ensuring that a program does what it is supposed to do[16]. Hardware mechanisms are customary to provide this protection either as a physically separate device, which implements the protected code module, e.g. hardware security module (HSM) or integrity monitoring [18]. An example of a physically separate device is a cryptographic coprocessor that internally contains cryptographic software. The cryptographic coprocessor presents the appearance of an I/O device to the rest of the platform. There are commercial devices currently on the market that can accept general purpose programs for execution and provide those programs with the same protections as their native applications[13]. Alternatively, software may configure general purpose hardware mechanisms such that no other entity within the system can violate the configured protections. Hardware introduces significant financial costs to achieve the desired platform security, and software introduces significant flexibility costs to preserve platform security.

Two different categories of applications benefit from the assurance provided by protected execution. The first category is defined by those applications that require protection of long term secrets, i.e. cryptographic key. The separation established by an operating system for a process or provided by a hypervisor for a virtual machine is insufficient to meet this requirement because the operating system or hypervisor have direct access to the sensitive data. The second category is defined by those applications that are trusted to monitor or evaluate platform integrity[14]. Without protected execution, the results returned by the monitor program is suspect as there will always be the possibility that the program may have been tampered as it shares the same memory space with the operating system or hypervisor and, therefore, is subject to tampering by anything that executes within the operating system's memory.

Coprocessors provide a suboptimal implementation for protected execution, especially for measurement software. A coprocessor means that an additional device is added to the platform. Thus, for space and cost reasons, a coprocessor may not be practical for all platforms such as a laptop or other mobile device. Also, measurement software that executes on a coprocessor is easily spoofed because the platform software may manipulate the coprocessor's view of memory by configuration of the DMA protections. Further, coprocessor hosted measurement software does not have any visibility of the processor registers such as the CR3 or the page table pointer, which must be reliably accessed to ensure that the monitor provides an accurate report of the

platform execution. A PE solution that utilizes a mode of the Intel x86 processor[1] called system management mode (SMM) is described in the remainder of this paper. The approach requires no additional hardware to provide protected execution, preserves platform flexibility, protects long-term secrets, and provides reliable access to critical platform register state. We describe our implementation in support of a novel hypervisor integrity monitor based on research published previously[14].

2. BACKGROUND

2.1 Xhim

An example of a measurement engine that requires PE is XHIM[11]. This engine is based on the Linux Kernel Integrity Measurer (LKIM)[14]. XHIM is focused on measuring the relationships between the variables and their critical state within Xen[3]. In addition, The Xen text section is measured by computing a hash of its current state. Finally, security relevant state information of the processor and chipset is obtained. All of this information is exported to an external appraiser. Xhim was originally implemented to run alongside the Xen kernel.

XHIM has three basic requirements that must be met. It needs access to processor state information, it must be self-contained, and must has assurance that it will not be modified during its execution. Hardware state information is necessary because some protections that are established can be affected by modifying privileged processor registers such as model specific registers (MSR) and the control registers (CRx). Self-contained means that it is not dependent on external libraries. The examples used in this paper are based on the XHIM PE implementation.

2.2 System Management Mode

System management mode (SMM) is an operating mode of the Intel x86 processors that provides a protected execution environment for platform functions. When the processor enters SMM, it has access to a separate address space called system management ram (SMRAM), which can be configured to be inaccessible from the other operating modes and effectively provides a protected execution environment. Only platform processors in SMM can access SMRAM. Also, SMRAM is protected from access by I/O devices. However, SMM is the most privileged processor mode as software executing in this mode can access all physical memory and devices. In addition, SMM may see all platform registers as well as modify privileged registers such as CR3.

Two separate, non-overlapping, memory segments define SMRAM: the ASEG and the TSEG. The ASEG is the original SMRAM. It and the legacy graphics region share the same address space. The ASEG is addressed only when the processor is in SMM. Otherwise the graphics region is addressed. The TSEG is a configurable memory region that enjoys SMRAM protections.

SMM can be entered only via a system management interrupt (SMI). This interrupt causes the processor to set its mode to SMM, to save the current state into SMRAM, and to start execution at a specified location within SMRAM. The module executing in response to an SMI is called an SMI handler. While in SMM, all interrupts are inhibited allowing the SMI handler to execute without interference. SMIs are usually generated in response to a hardware event, though in certain platform specific instances an SMI can be triggered from software. Exiting SMM is done by issuing a return out of SMM (`rsm`) instruction. When executed, the processor restores the saved entry state, changes its mode to the previous mode, and continues executing at the interrupted location. During the process of entering and leaving SMM, the operating system is usually unaware of it happening.

SMM is an example of protected execution since the code and data within SMRAM are protected from observation and interference from all platform processors that are not in SMM and from all I/O devices. However, it is the responsibility of the code executing in SMRAM to protect itself by configuring the hardware to prevent such accesses. This suggests that SMM is an ideal platform mode for hypervisor integrity monitoring because the monitor can execute without interference[17]. Otherwise, the monitor would have to share the same address space as the hypervisor, limiting the reliability of the monitor due to potential for compromise of the hypervisor. However, one must also consider the threat to the platform by the SMI handler as it is located in SMRAM. Some efforts have attempted to co-exist[20] with the SMI handler and others have attempted to eliminate the SMI handler[1]. Co-existence means that the monitor shares the same address space with untrusted code in the SMI handler. Elimination of the SMI handler is not feasible because the SMI handler provides critical platform functions, e.g. thermal event handling. A solution that allows for SMM to be used for protected execution and allows for the SMI handler to co-exist in this environment will be discussed in the next section.

2.3 SMI Transfer Monitor

A SMI transfer monitor[2] (STM) is a simple hypervisor that executes on a root-VM (virtual machine) during SMM while the dual-monitor treatment (DMT) is active. DMT was introduced as part of the Intel x86 virtualization technology (VTX)[6]. DMT provides full VTX support during SMM. The legacy SMM mode, which was discussed in the previous section, is the default treatment in Intel terminology. The STM is designed to be BIOS agnostic and should be usable with any platform that has a BIOS conforming with the Intel STM architecture specification[8]. The STM is located in the monitor segment (MSEG), which is contained in the upper addresses of the TSEG discussed in the previous section.

When the dual-monitor treatment is active, there are two hypervisors functioning as peers on an x86 platform. One is a platform hypervisor, such as Xen, or a platform operating system, such as Linux. The second is the STM, which controls the guest-VM that hosts the SMI handler. Both the platform hypervisor and the STM execute in a root-VM, which is the controlling VM. All guest-VMs are initiated from the root-VM. The active hypervisor depends

[1]In this paper, logical processor and processor are the same.

[2]Also called SMM transfer monitor. The Intel documentation uses these terms interchangeably.

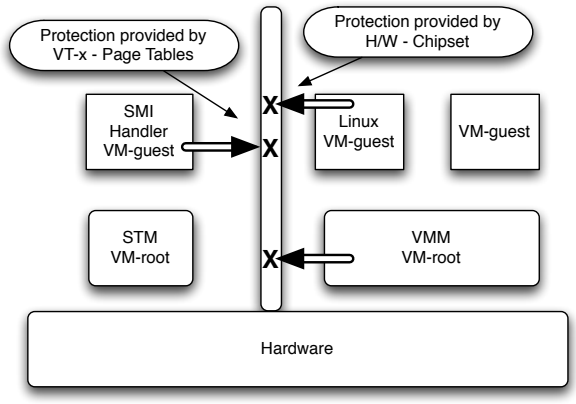

Figure 1: SMI Transfer Monitor (STM) Protections: When the processor is not in SMM, the chipset prevents the processor from accessing SMRAM where the STM and SMI handler are. When the processor is in SMM, a VM encapsulates the SMI handler.

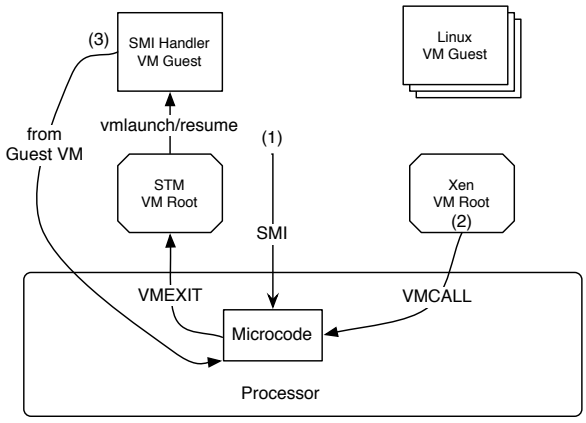

Figure 2: SMM Dual Monitor Treatment VM-exit types: (1) SMI converted into VM-exit; (2) Root-VM `vmcall` instruction; (3) VM-exit from SMM guest-VM because of ether a `rsm` instruction, `vmcall` instruction or configurable condition

on the processor mode. When the processor is in non-SMM, the platform hypervisor is active. When the processor is in SMM, the STM is in control.

An STM reduces the size of the trusted computing base (TCB) by isolating and effectively removing the SMI handler from the measured launch environment (MLE). This is the execution environment that exists at the completion of the dynamic root of trust measurement[12] (D-RTM). The Intel terminology for D-RTM is Trusted Execution Technology (TXT)[7]. The MLE provides a protected known execution environment to begin the startup of a trusted kernel. The runtime assurance of the MLE, which was launched by TXT could be compromised without an STM[19]; because, software executing in SMM has unrestricted access to the platform. SMM software can be compromised by an attack upon SMM[10]. Thus far, it has been shown that an STM mitigates several known SMM attacks[21, 23, 22, 5]

Figure 1 shows how the protections are applied depending upon the mode of the processor. Since the SMI handler is now encapsulated by a virtual machine, its accesses to the rest of the platform are constrained by the guest-VM's page tables. In addition, the guest-VM's access to I/O ports and MSRs are also constrained by settings in the guest-VM's VMCS. A VMCS or virtual machine control structure is an x86 processor data structure that defines a virtual machine. The STM is responsible for properly setting up these tables in accordance with the security policy. However, the STM's access to the platform has no constraints. The STM has to be evaluated to ensure that there are no improper accesses. Fortunately, the STM is designed to be thin, performs limited functions, is relatively easy to understand and fits the notion of a small trusted computing base (TCB).

The STM discussed in this section was developed by Intel and conforms with the Intel STM Architecture Specification[8]. The Intel STM recreates the default treatment SMM environment in a VM expected by the SMI handler. A single page table is shared by all SMI handler guest-VMs, giving all SMI handlers the same view of memory.

2.3.1 STM Interactions

When DMT is active, all transitions into SMM result in a VM-exit. Normally, a VM-exit is a transition from a guest-VM to a root-VM, and a VM-exit can be the result of a variety of events. One of these events is when a guest-VM uses a `vmcall instructio`n, which is used to request a service from the root-VM. However, a root-VM normally cannot use a `vmcall` instruction and its use by a root-VM normally results in a fault. When DMT is active, the behavior of a `vmcall` instruction changes to when a root-VM executes a **vmcall**; the result is a VM-exit into SMM. Included in the change in transition behavior are SMIs, which are converted into VM-exits as well. Finally, a third way a VM-exit occurs is as a result of either a condition or a vmcall instruction executed in an STM guest-VM. The conditions that cause a VM-exit are defined in the VM's configuration. Figure 2 illustrates the different VM-exits into a SMM during DMT. When a STM receives a VM-exit caused by an SMI, the STM starts the SMI handler guest-VM to process the SMI.

Normally, all processors respond to an SMI by entering SMM. When the processor in in DMT, the processors still go into SMM, except now SMM entry is via VM-exit. When the STM receives a VM-exit that is caused by an SMI, it switches to the VMCS of the guest-VM that runs the SMI handler. The STM ensures that the relevant SMRAM memory areas are initialized to replicate the environment that is expected by the SMI handler during the default treatment. When the SMI handler has completed its tasks, it normally uses a `rsm` instruction to switch the processor out of SMM back to its original state. Under DMT, the result of this instruction is a VM-exit, which is received by the STM. Upon receiving the VM-exit, the STM restores the SMM-transfer VMCS and issues a `vmresume` instruction, which restarts the peer VM along with the interrupted state.

The STM interface with the operating system is the `vmcall` instruction. In this case, if the root-VM executes a `vmcall` then a VM-exit into SMM results for that processor. However, in contrast to SMI handling, only one processor enters SMM. All other processors are unaffected by this action. The STM checks the VM-exit status to determine that this VM-exit is the result of a `vmcall` instruction instead of an SMI. In this case, the STM vectors to its application program interface (API) handling section to deal with the request from the peer root-VM[8]. Once it has finished processing the request, it executes a `vmresume` instruction to restart the peer root-VM.

2.3.2 STM Startup

Starting an STM requires cooperation between the BIOS and the MLE. The STM image is included as a part of the BIOS image. During platform startup, the BIOS ensures that the STM's identity and integrity are acceptable before copying the STM image into the MSEG. If the STM has passed the BIOS checks, the BIOS sets a flag in a MSR[3], which indicates that a STM has been loaded into the MSEG. Before starting TXT or D-RTM, the MLE indicates that it supports a STM. If so, the STM image is measured during TXT, and the STM's hash is extended into PCR 17 of the TPM[7]. As a part of this process, the STM's dynamic memory region is also zeroed, which effectively initializes the STM. The MLE or the launch policy evaluates the STM measurement and, if satisfactory, continues the STM launch process. Since the STM is measured as a part of TXT, any tampering of the STM image should show up in the hash.

To launch the STM, the MLE executes a `vmcall` instruction, which causes a VM-exit in SMM. This initial SMM VM-exit activates DMT. Prior to this `vmcall` instruction, the MLE has to setup a control VMCS that will be used during the initial SMM VM-exit. The VM-exit controls and host state necessary to initialize the STM is based upon the contents of the MSEG header and fixed values from the firmware.

The STM has to create two VMCSs during its initialization. The first VMCS is used to manage the STM's peer, which is the MLE/hypervisor; this VMCS is called the SMM-transfer VMCS. The guest portion of the SMM-transfer VMCS is populated with information from the guest portion of the control VMCS previously created by the MLE. The host portion of the SMM-transfer VMCS is filled with the necessary information, such as the STM entry point, which starts the STM root-VM when any subsequent VM-exit caused by a `vmcall` instruction or an SMI.

The second VMCS defines the SMI handler's guest-VM and is called the BIOS VMCS. This VMCS is created based on the information contained within the SMM descriptor data structure. This holds the SMM VM guest entry state for each processor and is created by the BIOS. In addition, it holds the SMI handler location. The STM creates page tables for the SMI handler guest-VM based on this information. However, these tables are generated such that the SMI handler has access only to its pages within SMRAM. No access to the STM's address space is permitted. SMI handler access to memory external to SMRAM is not defined at this time. These accesses are later created based on a combination of the BIOS resource list and the MLE's restrictions.

Once the STM resource list has been configured, the MLE issues a `vmcall` instruction to the STM to start operations. This call must be done for each processor where each processor initializes its STM data structures including establishing the two VMCSs described previously. When each processor has completed the STM initialization, the processor rendezvous. Once all processors have completed initialization, the rendezvous allows all processors to simultaneously execute a `vmresume` to return to the MLE. During this return, each processor enables SMIs[4].

2.3.3 STM Security Policy

The STM controls the SMI handler's actions on the basis of platform resources and by domain. The resource access is controlled based on the contents of the STM resource list. The STM resource list is created from the BIOS resource list and the resources requested by the MLE for its protection. A resource is anything that can be addressed within the platform, to include memory, PCI addresses, I/O addresses, and MSRs. The BIOS resource list is provided by the BIOS, and incorporated into the STM resource list during initialization. This list static and does not change at runtime. The STM resource list is built on a single processor and the effect of the resource list is global.

The MLE can request that the STM prevent access to specific resources. It also can request the full contents of the BIOS resource list as well. In case of conflict, the BIOS's access request takes priority. The reasoning behind this is that if the BIOS cannot access a resource, then it cannot function properly. However, the MLE does have the option to not bring up the STM or it can inform the administrator that the BIOS is requesting access to resources that seem inappropriate. A conflict in a platform that had no previous resource conflicts is a good indicator that something is wrong and should be investigated for unauthorized configurations.

When the SMI handler attempts to access a resource, the STM resource list is consulted. Unless access to a resource is explicitly denied, access is granted. In the event that the STM blocks access to a resource, the SMI handler can provide an exception handler to deal with the situation. Otherwise, the STM will record the error and reset the system.

Domain access checking uses a VMCS pointer database, where each VMCS represents a VM or a domain. When an SMI causes a VM-exit, this database is checked to see if the interrupted VMCS is in the database. If so, the STM will scrub the register state that the SMI handler would normally receive. This action is intended to prevent the SMI handler from seeing sensitive information held within a domain. It is the responsibility of the MLE to maintain the VMCS database and the MLE can configure how restrictive the STM is to be when a particular domain is interrupted.

2.3.4 Discussion

[3]This MSR bit can be set only during SMM.

[4]If there is an STM, SMIs are inhibited when the MLE gains control from TXT. If necessary, the MLE can enable SMIs.

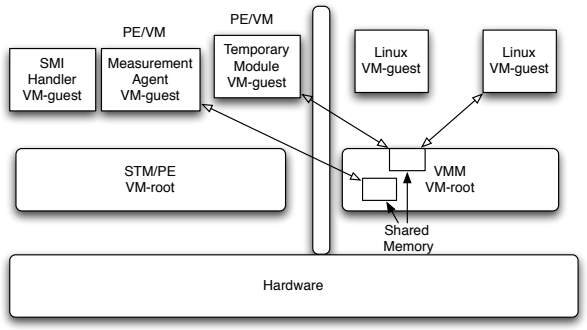

Figure 3: STM/PE Overview

The STM is a capability that the MLE can use to protect itself from the SMI handler. Like any of the other protections provided by the platform such as the DMA (VT-d) protections, the MLE still has to decide how each protection is to be configured. The STM provides a single command for the MLE to turn off BIOS access to any resource that it has not claimed. The MLE can also restrict access to a subset of the unclaimed resources as well. The STM provides the list of resources that the BIOS requires access. This information provides evidence for the system owner to query the BIOS developer about accessing platform resources that MLE deems sensitive. Properly used, the STM can improve the overall security of the system by ensuring that the SMI handler can only access platform resources that it needs.

3. STM/PE

This section describes an implementation of protected execution on a STM (STM/PE). Protected execution is implemented by modifying the STM to support additional guest-VMs (PE/VM) as shown in Figure 3. PE/VMs enjoy the same protection from external observation that the SMI handler VM has. Also, all of the STM guest-VMs normally do not have permissions to access any of the other guest-VMs. In addition, PE/VMs have greater restrictions placed on their accesses than the SMI handler VM. A PE/VM is not allowed any I/O access nor any access to MSRs. Aside from a shared memory region, a PE/VM only has access to SMRAM memory that is assigned to it. The shared memory is intended to be a communications buffer between the PE/VM and an agent on the host system. In the figure, both PE/VMs each have a shared memory area. The temporary PE/VM is sharing its buffer with a host process while the permanent PE/VM, which is the measurement PE/VM, is sharing its buffer with either the VMM or an external agent. Temporary and permanent PE/VMs are discussed in the next section.

3.1 PE/VMs

3.1.1 PE/VM Types

STM/PE currently supports two different types of VMs: temporary and permanent. A temporary PE/VM only runs for a configured maximum amount of time and is then dismantled. It is intended for a user application that has a need for protected execution and the compute time is bounded. Since SMRAM space is limited and the allocated processor is bound to the PE/VM for the duration of the PE, there should be only one temporary PE/VM executing at a time. If a temporary PE/VM is executing and another application requests a PE/VM, the requesting application will receive a busy return. In this case, the requester should wait and try later to request a temporary PE/VM.

The permanent PE/VM is intended to contain a measurement agent that is at the lowest layer of a hierarchy of runtime measurement agents. Since this measurement agent is in a protected space, it can effectively serve as the anchor for the measurement agents above. The loading of the measurement agent needs to be done just after D-RTM and before the STM is stared to ensure that no malicious software gets loaded in its place. The ability to load a permanent PE/VM is disabled once the STM is started by the MLE. If there is no permanent PE/ME, the MLE configure the STM to disable the ability to establish a permanent PE/VM. A permanent PE/VM is never dismantled unless it terminates due to some error condition and only if the STM is configured to dismantle the permanent PE/VM. Otherwise, the PE/VM resources will be retained for subsequent executions. After the initial run, a permanent PE/VM can be executed again by a `vmcall` command.

3.1.2 PE/VM Guest Physical Address Layout

A PE/VM's guest physical address space is divided between SMRAM (MSEG) and the host machine physical address space (non-SMRAM). The SMRAM space is used for the PE module. The non-SMRAM addressable regions are used for a shared buffer and, in the case of a permanent PE/VM read only regions that are used for measurement. The shared regions are located in non-SMRAM areas and are identity mapped from the PE/VM guest physical address space to the host physical address space. The STM security policy prevents any pages located within SMRAM from being identity mapped into a guest-VM physical address space. A data structure defines the memory associated with the PE/VM and also the PE?VM configuration as well. All pointers are physical addresses because the STM understands only physical addresses. The variables described below are in `ModuleInfo`, which is described in greater detail in the appendix.

The executable module and its associated data are located within the MSEG area and mapped into the PE/VM guest physical address space. The guest physical start address of the address space is `AddressSpaceStart` and its length is `AddressSpaceSize`. These locations are mapped into SMRAM. The module is contained within this region. The module's start address is defined by `ModuleStartAddress` and its size is `ModuleSize`. The module entry point is `ModuleEntryPoint`. This allows the executable module to have protected data memory both above and below the executable. The areas outside of the module are read-writable while the module itself is execute-only. The current implementation limits this region to 1 megabyte in size and that limitation is mainly due to the limited size of the MSEG. Figure 4 shows how the addresses are related.

Two pointers describe the PE/VM host memory. The first pointer is the RegionList, which is a list of read-only regions in host memory accessable by a permanent PE/VM. The intent is to provide read-only access to hypervisor or kernel memory for a measurement agent executing in a PE/VM.

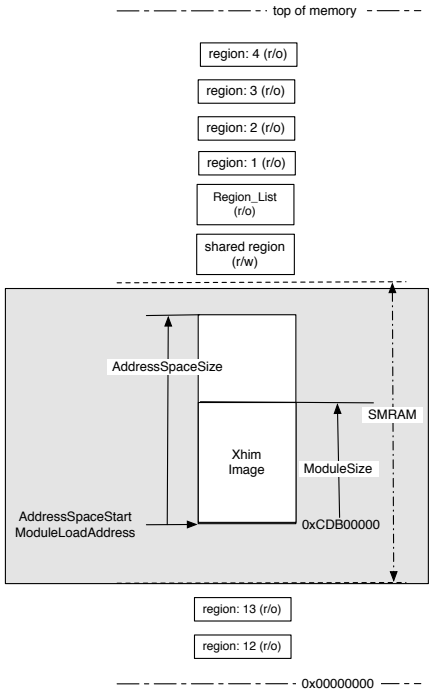

Figure 4: Memory layout for a permanent PE/VM. Layout is defined by data structures in Figure 5. The area shaded in gray is SMRAM. The regions outside of SMRAM are host memory.

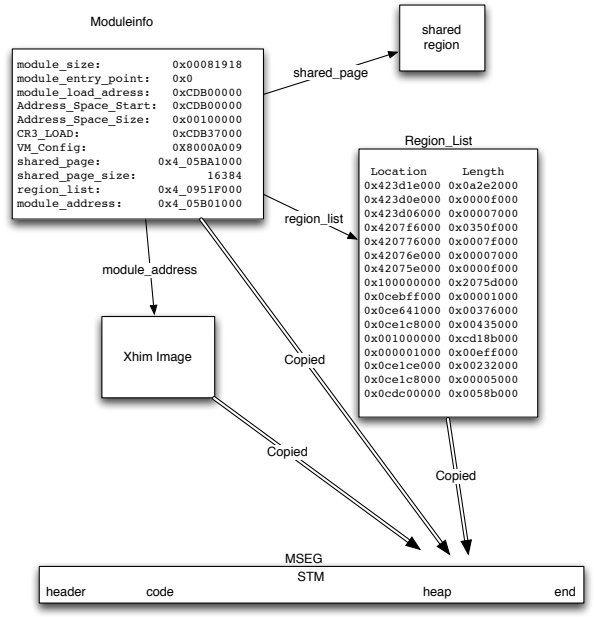

Figure 5: Moving Xhim and configuration information into MSEG. These data structures create the memory layout illustrated in FIgure 4.

The second pointer is called SharedPage and it points to a multi-page read-write region that is a communications buffer between a PE/VM and an outside agent. Both a permanent and a temporary PE/VM have a shared region.

3.1.3 Example: Memory Setup for XHIM

This example starts after Xen has done the `vmcall` as explained in section 3.4. Figure 5 illustrates the `ModuleInfo` data structure, the region list, and the XHIM image being copied to the STM heap, which is located in SMRAM. At this point, the STM has been entered by a VM-exit and the STM has copied the `ModuleInfo` data into an internal data structure. The STM validates all of the addresses to ensure that they do not violate policy. If all addresses are valid, the XHIM image is copied to a block allocated in the STM heap. When the STM builds the page tables for the PE/VM, the XHIM image, shared region, and the regions in the region list will be mapped into the PE/VM as shown in Figure 4.

3.1.4 SMRAM Available for PE/VMs

The amount of memory available for PE/VMs is, in part, dependent upon the size of the MSEG that is configured on the platform. On the development system, the MSEG size was configured to be 3,145,278 bytes (or 0x300000). The STM and its associated static data take around 495000 bytes. This leaves 2,650,112 bytes for heap storage, which is used to allocate VMCSs, the BIOS resource list, and page tables. On an eight-processor system, around 2,300,000 bytes of heap space are left for PE/VM allocation. The maximum PE/VM memory size is arbitrarily set at one megabyte to get around issues of heap fragmentation and to leave space for page tables. The module space is allocated as a single contiguous area of up to one megabyte. A future version could allocate the module space in page size chucks, as this approach was chosen simplify the implementation.

A future version could make more than 250 megabytes of SMRAM available for PE/VMs. By definition, SMRAM is a memory area accessible to the processor only while it is in SMM, and protected from DMA. The DMA protected range (DPR)[7] can be used as additional SMRAM. The DPR is a region of memory that is adjacent to the TSEG. By adjusting the system management range register (SMRR) to include all or part of the DPR, the available SMRAM can be increased. The DPR is configured by the BIOS and this region is where the TXT heap and the SINIT region may be located depending upon the platform configuration, which reduces the maximum size available for SMRAM use.

3.1.5 Performance

In the literature, it is noted that code in SMRAM executes slower than code located in system RAM[1]. This is because most implementations that run in SMM turn off SMM caching due to the risk of cache poisoning attacks[22]. However, Intel has modified their processors to mitigate these attacks. The Intel STM Specification provides guidelines for configuring the processor and invalidating the cache during SMM transitions. STM/PE implements the Intel guidelines.

The performance testing of STM/PE showed a significant difference when using cache while in SMM as opposed to turning the cache off. XHIM was tested in how long it took to start up and then measure Xen. The start time was taken immediately before the `vmcall` instruction and the finish

time was taken immediately after the `vmcall` instruction returned, with the difference providing the run time. With the cache disabled, XHIM took 1811ms for the initial run where the PE/VM along with its page tables had to be setup. Subsequent runs took 1681ms. With cache enabled, the initial run took 24ms and the subsequent runs took 23ms. Round trip with cache enabled, where only a `rsm` instruction was loaded and executed, was .63ms for the initial run and .54ms for subsequent runs. According to the XHIM developer, the execution time for XHIM in SMM with cache enabled is comparable to running it in system memory.

3.2 STM/PE Implementation

To support protected execution, the Intel STM was modified to support additional guest-VM types. The STM has two VMs per processor: one for the SMI handler and the second for the peer (MLE/Xen). STM/PE adds three guest-VM types that can be assigned to any processor as opposed to being dedicated to a processor for the SMI handler and peer VMs. Since the SMI handler VMs share the same page table, this means that the page table management had to modified to support additional VM types. Also, VM-exit handling and the security policy were modified to account for additional VM types.

3.2.1 Page Table Modifications

The STM page table management was extended to handle memory mapping and to handle multiple page tables. The original page table implementation for the SMI handler VMs was designed to handle identity address mapping. This is because the SMI handler only dealt with machine physical addresses in the default treatment and remained in the same physical location in both the default treatment and the dual-monitor treatment.

However, a PE module is promised a contiguous guest physical address space where the addresses start at a configured location and these pages are mapped into SMRAM. Which means that the STM/PE pager deals with address mapping like a normal page table handler. Also, it is desired that the PE module developer be able to configure the VM to use an addressing mode that meets the module's needs. In this implementation, it is possible to configure an address space that does not require the developer to implement page tables. However, the module is free to create and manage its own page tables and to change the processor addressing modes.

STM/PE supports four sets of page tables with each set being designated for a VM type. Page table 0 is for the SMI handler VM, page table 1 is for the permanent PE/VM, page table 2 is for the temporary PE/VM, and page table 3 is currently intended for the SMI handler measurement VM. The PE/VM types are discussed in section 3.1.1. The SMI handler measurement VM is proposed to measure the SMI handler VM and will have visibility into the SMI handler address space. There is no design limit for the number of page tables aside from the amount of memory available in SMRAM.

3.2.2 Security Policy Changes

The STM resource list was modified to support additional VM types by having an STM resource list for each type. This allowed for common code to do the security checking. The appropriate resource list is checked upon an initial access and the appropriate hardware data structures, such as page tables, are updated to allow the hardware to do the enforcing. The access policy was modified for PE/VMs. If the VM is an SMI handler VM, then the default access policy is followed, which is if a resource is not explicitly denied, then access is allowed. In this case, explicitly denied means the resources that the MLE has designated that it wants to be protected. For PE/VMs, the policy is to deny access if the requested resource is not on the STM resource list. In summary, the difference in the access policies is in how the undefined accesses are handled.

3.2.3 VM Creation and Termination

The capability to create and terminate VMs was added to the STM. An unmodifed STM's VMs are always permanent. A PE/VM can be created and terminated at any time. They are also different because a PE/VM's page tables have to support a mixture of identity mapped and non-identity mapped pages, and a variety of addressing modes are allowed. In contrast, an SMI handler's page address translations are normally identity mapped.

VM creation starts by configuring the memory described in `ModuleInfo` as described in Section 3.1.2. Once all of the resources have been allocated, the PE/VM VMCS is then initialized. The VMCS guest state and the processor addressing modes are based on the information provided by `ModuleInfo`. The VMCS configuration is checked both by the STM and the hardware, any problems are reflected back to the requestor. Prior to the launch of the PE/VM the `rax` and `rbx` registers are set to point to the shared buffer and the region list, respectively. For a permanent PE/VM, the VMCS guest-state information is saved and it is reloaded when the permanent PE/VM is restarted. The`rax` and the `rbx` registers are also set to their initial values as well. However, the module has to initialize PE/VM memory.

When the PE/VM has completed its execution, and if the PE/VM is permanent, its VMCS, VM memory, and related data structures are retained in SMRAM for subsequent execution. If the PE/VM is temporary, all of the resources associated with that VM are released.

3.3 Special Cases

3.3.1 SMI Handling While a PE/VM is Active

When a PE/VM is executing, it is possible that an SMI may occur. Normally, an SMI causes all processors to go into SMM. However, the PE/VM processor will not respond to the SMI as SMIs are inhibited while a processor is in SMM. Therefore, all the SMI handler processors will hang waiting for the PE/VM to complete.

To resolve this problem, the STM has to recognize that a processor is executing a PE/VM when an SMI occurs. If so, the PE/VM has to be suspended and an SMI handler VM started in its place. When the STM starts a PE/VM, it sets a status word to indicate that a PE/VM is executing along with the associated processor id.

When the STM processes an SMI VM-exit, it queries this status word to see if a PE/VM is executing. The status

word is protected by a lock, which ensures that only one processor does this check. If a PE/VM is executing, a non-maskable interrupt (NMI) is directed to that PE/VM processor. When the PE/VM processor receives the NMI, it does a VM-exit because it has been configured to exit upon receipt of a NMI[5]. As a precaution, the STM checks to ensure that the NMI received by the processor was not because of an unrelated problem.

The interrupted PE/VM state is saved and the STM takes the same path as if it were returning to the MLE. However, the MLE is never entered because the pending SMI causes a VM-exit back into the STM[6]. The STM processes the SMI VM-exit as normal and starts the SMI handler VM. When the SMI hander VM has completed, the STM gain control. The STM checks to see if a PE/VM was interrupted on this processor. If so, the PE/VM is restarted[7].

3.3.2 *Obtaining Other Processor State*

This section provides a solution where measurement engines need to determine the state of all the processors in a multi-processor system. The problem is one of visibility into the other processors' state and has been the subject of some rather convoluted solutions[1]. The solution here uses a single interrupt to obtain information from the other processors and uses the pointers in their respective VMCSs to obtain the necessary state information.

A `vmcall` instruction is used by a permanent PE/VM to request state information of all the processors. To do this, the STM sends an SMI to the other processors. As a part of this SMI, the STM sets a flag indicating that the current processor sent the SMI. When the other processors gain control, they examine their SMM-transfer VMCS to determine where to obtain the state information. This state information is placed into a buffer that is provided by the caller.

However, determining the exact peer VMCS location needed to gather the state information from depends whether the interrupted processor was a root-VM or guest-VM. If the interrupted processor was a root-VM, the processor state can be found in the SMM-transfer VMCS guest-state as the root-VM's state was transferred to that location during the VM-exit. If the processor was a guest-VM, the executive-VMCS pointer has to be followed to the guest-VM VMCS and the processor state is found in that VMCS's host-state fields. The VMCS host-state field holds the processor state for the root-VM that is loaded when there is a VM-exit. Once each processor has completed its data gathering, a flag is set indicating its completion and its peer is resumed. When the initiating processor determines that data gathering operation has completed in all the other processors, the PE/VM is resumed. The data requested is returned in a buffer provided by the caller.

[5]A VM executing in SMM cannot be configured to VM-exit when an SMI has been received.

[6]If there were a way to directly acknowledge the pending SMI, then the STM could run the SMI handler; when that completes, acknowledge the SMI; and then restart the PE/VM.

[7]A future implementation will allow the PE/VM to request that a handler be called in the event that the PE/VM was interrupted by an SMI. This is used to inform the measurement engine that its measurement has become non-atomic.

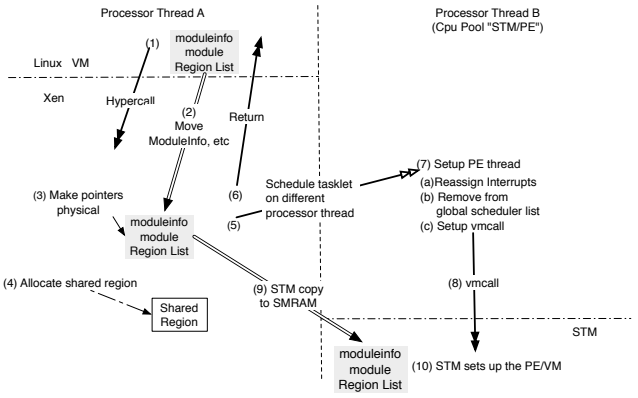

Figure 6: PE/VM Setup flow through the Xen hypervisor. The double arrows are control flows. The double lines are block copies. Numbers 1-7 are discussed in Section 3.4 and 8-10 are discussed in Section 3.1.3

3.4 Modifications to Xen

This section discusses the changes necessary for the Xen hypervisor to support protected execution. Operating system support is necessary because the STM can only be entered via a `vmcall` instruction while the processor is in a root-VM, which can only be done in the hypervisor. Also, STM/PE functioning removes a processor for an extended length of time. A processer is a resource that is managed by the hypervisor and it expects to be able to assign work to a processor. As a result, the hypervisor must be able to deal with the loss of a processor. In addition, every processor is assigned by the Xen hypervisor the task of fielding interrupts. If a PE/VM is executing on a processor at the same time an interrupt managed by the processor occurs, then the fielding of that interrupt will be delayed until the PE/VM has completed its task. This is because interrupts are inhibited while a processor is in SMM. When interrupt processing is delayed, various problems related to system I/O might occur that are not recoverable by the operating system. During STM/PE testing, Xen would either panic, lockup, or attempt to unsuccessfully handle what it perceived as a hardware problem because an interrupt was not processed in a timely fashion.

The process for starting STM/PE is for Xen to be entered via a hypercall. A hypercall was added to Xen to support protected execution. This process is illustrated in Figure 6. The Xen STM driver first moves the `ModuleInfo` data structure, the executable image (module), and the region list to a location within Xen. It then converts the guest virtual addresses within the `ModuleInfo` and the region list into physical addresses for the STM's benefit. Then the STM driver sets up an available processor to be used for STM/PE and then assigns that to the STM/PE CPU pool. It then schedules a tasklet on the processor to be used for STM/PE. A tasklet is a Xen task that provides system services such as clock management and is attached to a specific processor. The hypercall then returns control to the guest-VM, where the software polls for the status of the PE/VM execution.

When the tasklet starts, it first removes its processor from the online map and has the processor's interrupts reallocated to other processors. This is done to fool Xen into thinking that the processor is not there. Then the tasklet executes a `vmcall` instruction, which then causes a VM-exit in the STM. The STM continues as descibed in Section 3.1.3. Once the PE/VM has completed and returned, the tasklet regains control. The tasklet then updates a status word depending upon the return status and the type of request sent to the STM. After this, it places the processor back onto the CPU online map and the processor waits in an idle loop.

As can be inferred from the previous paragraphs, the hypercall returns to the calling VM with status information indicating that the STM/PE was properly setup. Since the PE/VM executes on a separate VM, the initiating guest-VM must poll the Xen STM driver to determine when results are ready or any other PE/VM status. Once there are results available, the initiator then has to request that Xen copy the contents of the shared buffer to a location local to the calling VM.

4. PE/VM APPLICATIONS

SMM is used as a protected execution environment for several defensive mechanisms that monitor hypervisor integrity [1, 20, 17]. However, the downside of using SMM for these mechanisms is that they all have to share the same address space with the SMI handlers. This means that the SMI handlers have to be trusted, which increases the size of the trusted software. The functions of the SMI handler and the hypervisor integrity monitors are mutually exclusive and both functions should reside in separate address spaces. STM/PE provides a viable base for these defensive mechanisms to function from. The STM can maintain separation between a hypervisor integrity monitor and the SMI handlers. In addition, as shown in a previous section, STM/PE can also provide state for the current processor and other processors as well, which removes the need for convoluted methods for obtaining processor state[1]. Finally, STM/PE limits the memory access allowed for an integrity monitor. An example of this access limiting is shown in the discussion about XHIM in the following section.

4.1 XHIM

In Section 2.1, a list of requirements was specified for XHIM's proper functioning that STM/PE meets. To obtain other processor state, STM/PE obtains this information in a straightforward manner as discussed in Section 3.3.2. STM/PE meets the self-containment requirement by providing a protected execution environment for XHIM that neither a host process nor an SMI handler can access. In this environment, XHIM can function without interference from other code executing and from I/O as well. Finally, the requirement to ensure that XHIM does not change during execution is met by ensuring that the XHIM text pages are read-only. This is done by setting the page tables that define the module's executable area to read-only.

4.2 Virtual Trusted Platform Module

STM/PE provides a low cost alternative to improve the assurance for a virtual trusted platform module (vTPM). The normal implementation for a vTPM is for it to be located somewhere in physical memory. Preferably, a vTPM should be located within a trusted VM. However, this still leaves sensitive data such as a private key vulnerable to hardware tampering attacks and also to vulnerabilities in the hypervisor. Using a co-processor mitigates these problems, but at a high cost, which limits this type of assurance to security-sensitive environments that can afford the cost[4]. The amount of SMRAM available to the STM would dictate how a vTPM would be structured. At a minimum, functions that have to deal with sensitive data would be implemented on a STM/PE VM, with the remaining vTPM functions located on a trusted VM. On a platform that assigns the STM a larger memory space, a vTPM could be fully implemented on a PE/VM. In ether case, STM/PE increases the assurance of a vTPM without adding expensive hardware to the platform.

5. COMPARISON TO OTHER WORK

5.1 Flicker

An example of protected execution is Flicker[15], which uses the D-RTM[7] as a means to protect the application. When D-RTM has completed, the platform is placed into a known state. In this state, only a single processor is active and all other processors are idle. The memory is protected from I/O and only the software in the MLE is executing. Though this is a viable method to provide protected execution, it practicality is limited because there is significant overhead to enter the protected environment and the platform is dedicated to executing the module needing to be protected.

5.2 SICE

SICE is an example of protected execution that is implemented on the AMD x86 processor[2]. Like STM/PE it provides an isolated VM environment for a workload. SICE utilizes the protections provided by SMM, and some trusted software to manage the separation. In a multiprocessor environment, SICE supports concurrent isolated environments to run alongside an untrusted host.

The differences between SICE and STM/PE are a result of the differences between the SMM implementations for the AMD and the Intel x86 processors. AMD's SMM has a single addressing mode that is akin to Intel's big real mode, which is real mode with 32-bit addressing. Intel's SMM has VTX support, which allows for the STM. The memory area of the TSEG is defined on each processor for AMD and can span different address ranges. Also, the TSEG definition can be modified at any time in AMD. This feature is used by SICE to separate its virtual machines by setting each processor to a different TSEG value. The Intel TSEG configuration is in the chip set, applies to all processors, and can not change after it is set. The AMD TSEG memory has a maximum size of 4GB while Intel currently limits the TSEG to 8MB. Another significant architectural difference is that the Intel processor can execute a `vmcall` instruction to enter the STM on a single processor. SICE can be entered only by an SMI, which causes all processors to enter SMM, which includes any PE VMs that are executing.

Because of the differences in the processor SMM implementations, SICE has a more complicated software organization than that of STM/PE. The SICE software consists of

two trusted parts: an SMI handler and a security manager. When a protected execution is requested, an SMI is triggered causing the processors to enter SMM giving SICE control. To run a PE module[8], a virtual machine has to be started up in a non-SMM environment. Since the SMI handler cannot directly change privileged processor registers, a `rsm` instruction would result in a return to the caller. To allow the security manager to gain control, the SMI handler modifies the page tables to point to the security manager instead of the original return point. Also, before the `rsm` instruction is executed, the SMI handler has to change the TSEG configuration for that processor so that the both the security manager and the PE module are no longer in SMRAM. This is done because a VM cannot run in SMM. The other processors will not have visibility into this environment, as their processor specific TSEG configurations will make them consider this area as SMRAM[9]. Once the security manager gains control after the `rsm` instruction, it sets up and runs the PE/VM. In contrast, the same action done by the STM is a matter of switching VMCSs and the VM executes within SMRAM.

Both SICE and STM/PE provide similar execution environments in that they both provide a configurable VM for the PE module and a shared memory area, outside of SMRAM, that is used for communications between the protected environment and the host environment. The major difference here is that STM/PE is limited on the amount of memory available for protected execution. Also, They differ on how long a protected execution is allowed to exist. With SICE, all protected executions are permanent and exist until the PE module decides to terminate. On the other hand, STM/PE categorizes PE modules into two categories: permanent, which exists as long as the STM exists, and temporary, which exists only for the duration of the computation.

SICE has to co-exist with the SMI handler, which means the SICE TCB includes the SMI handler along with the SICE components. To get around this issue, either the BIOS vendor incorporates SICE as a part of the SMI handler or that SICE relocates the SMI handler and uses memory protection to isolate the handler. In either case, the SMI handler shares the same memory space, as SICE or the SMI handler would not have the protections given to it by SMM since it would be outside of SMRAM. STM/PE does not have this issue as the STM isolates the SMI handler and, thus, can protect itself from the SMI handler.

6. SUMMARY

This paper has presented a protected execution implementation that utilizes the capabilities of the x86 hardware. A modified STM provides the capability to provide additional VMs that support protected execution. This capability includes allowing a permanent module to be established during the MLE and also allows for a temporary module needing PE during normal system operations. To facilitate these capabilities, an interface was developed for the Xen hypervisor. The objective of this effort is to make this implementation available when Intel releases the STM specification and reference code.

Previous attempts to utilize SMM as a protected location for system measurement always encountered the difficult issue of dealing with the SMI handler. Past solutions to the problem have been to not have the SMI handler in SMM, or have the measurement engine co-exist with the SMI handler. Neither of these options is viable from a security nor a maintenance perspective. An unmodified STM contains the actions of the SMI handler by placing it into a virtual machine and using the page tables and the I/O permissions to limit access.

Adding PE to an STM allows modules to enjoy a protected environment that is separate from the SMI handler. In addition, measurement engines will be allowed to access only the resources that they need to measure, and not full system access. STM/PE improves the security of the platform by constraining the SMI handlers, and providing a protected location for modules to execute along with constraining the actions of these modules.

Future work will include measurement support. A measurement engine will need some secret to sign the measurement report. This secret is either injected by the STM or the measurement engine obtains it from the TPM. Also, the STM needs to be able to attest to a remote party about the SMI handler as well. The current thinking is to have an additional PE/VM that does this function.

One issue that does not get much attention is that the operating system or hypervisor, has to be that SMM is being used for protected execution. Currently, the SMI handler has timing constraints to prevent problems such as lost interrupts from affecting the system[9]. Measurement engines typically take a longer time than then limitations placed on SMI handlers. Therefore, the operating system has to be able to cope with a processor disappearing. In the Xen testing, the only methods are to decouple the processor from the operating system or to dedicate a processor to protected execution.

STM is a vital part of TXT in that it prevents a malicious SMI handler from subverting TXT. By extending the STM to support protected execution, an additional benefit is realized because platform measurement has a protected location to function and functions such as vTPM can have their security critical functions protected from the system. In summary, STM/PE adds additional security functionality to the platform using only the capabilities of the x86 processor and with minimal modifications to the STM.

7. ACKNOWLEDGMENTS

The Author would like to thank Julian Grizzard for his efforts in getting XHIM to function inside a PE/VM.

8. REFERENCES

[1] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: Enabling stealthy

[8]In SICE terminology, this is the "isolated environment."

[9]It is unknown what effect this configuration (not all TSEGs having the same memory range in a MP system) has on the DMA protections that SMRAM is supposed to have. The AMD documentation only refers to SMRAM being protected from processor access. The SICE authors noted: "SICE cannot rely on hardware DMA exclusion," and they had no idea on how AMD hardware handled this case.

in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[2] A. M. Azab, P. NIng, and X. Zhang. Sice: a hardware-level strongly isolated computing environment of x86 multi-core platforms. In *Proceedings of the 18th ACM conference on computer and communications security*, pages 375–388. ACM, 2011.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM symposium on Operating systems principles*, pages 164–177. ACM, 2003.

[4] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vtpm: Virtualizing the trusted platform module. In *15th USENIX Security Symposium*. USENIX Association, 2008.

[5] J. Butterworth, C. Kallenberg, and X. Kovah. Bios chronomancy: Fixing the core root of trust for measurement. In *Blackhat Briefings USA*, 2013.

[6] I. Corporation. Intel 64 and ia-32 architectures software developer's manual.

[7] I. Corporation. Intel trusted execution technology (intel txt) - measured launch developer's guide.

[8] I. Corporation. Intel trusted execution technology (intel txt) - smi transfer monitor (stm) - software architecture specificationintel trusted execution technology (intel txt) - smi transfer monitor (stm) - software architecture specification. To be published soon, October 2011.

[9] B. Delgado and K. L. Karavanic. Performance implications of system management mode. In *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2013.

[10] L. Duflot. Getting int the smram: Smm reloaded. In *CanSecWest*, 2009.

[11] J. B. Grizzard, J. A. Pendergrass, P. A. Loscocco, and G. S. Coker. Analysis of x86 hardware support for runtime measurement of kernels. To be published as an APL tech report, November 2010.

[12] T. C. Group. *TCG D-RTM Architecture*. Trusted Computing Group, 2013.

[13] IBM Corporation. *IBM 4758 Models 2 and 23 PCI Cryptographic Coprocessor*, 2004.

[14] P. A. Loscocco, P. A. Wilson, and A. J. Pendergrass. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, pages 21–29. ACM, 2007.

[15] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2008.

[16] M. Payer, T. Hartmann, and T. Gross. Safe loading - a foundation for secure execution of untrusted programs. In *IEEE Symposium on Security and Privacy*, 2012.

[17] J. Ruthkowska and R. Wojtczuk. Preventing and detecting xen hyperisor subversions. In *Blackhat Briefings USA*, 2008.

[18] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the twenteith ACM symposium on Operating systems principles*, pages 1–16. ACM, 2005.

[19] A. Vasudevan, J. M. McCune, N. Qu, L. van Doorn, and A. Perrig. Requirements for an integrity-protected hypervisor on the x86 hardware virtualized architecture. In A. Acquisti, S. W. Smith, and A. Sadeghi, editors, *TRUST 2010. LNCS*, volume 6101, pages 141–165. Springer, 2010.

[20] J. Wang, A. Stavrou, and A. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Proceedings of the 13th International Symposium on Recent Advamces in Intrusion Detection (RAID 2010)*, 2010.

[21] R. Wojtczuk and J. Ruthkowska. Attacking intel txt via sinit code execution hijacking.

[22] R. Wojtczuk and J. Ruthkowska. Attacking smm memory via intel cpu cache poisoning.

[23] R. Wojtczuk and J. Ruthkowska. Attacking intel trusted execution technology. In *Blackhat DC*, 2009.

APPENDIX

A. APPENDIX - STM/PE VMCALL INTERFACE

The STM/PE `vmcall` instruction interface is an extension of the STM `vmcall` instruction interface between the MLE and STM described in the Intel STM specification[8]. It adds four calls to the interface, two to add a PE/VM, one to run a permanent PE/VM, and the fourth one to disable he loading of PE/VMs.

The STM/PE `vmcall` instruction interface adds an additional data structure to the STM `vmcall` instruction interface. This data structure tells the STM how to configure a PE/VM, defines the memory layout for the portion of the guest-VM that is located within SMRAM, and states the hardware physical memory locations for the shared region and the read-only regions used by the PE/VM. A comprehensive set of status returns are provided to inform the caller of any errors encountered during PE/VM setup and execution. Following is an explanation of the `ModuleInfo` data structure used in the interface.

```
struct ModuleInfo{
    UINT64            ModuleAddress;
    UINT64            ModuleLoadAddress;
    UINT32            ModuleSize;
    UINT32            ModuleEntryPoint;
    UINT64            AddressSpaceStart;
    UINT32            AddressSpaceSize;
    UINT32            VMConfig;
    UINT64            CR3Load;
    UINT64            SharedPage;
    Struct region *   Segment;
    UINT32            SharedPageSize;
    UINT32            reserved;
}
```

`ModuleAddress` - Machine physical address of the location in system memory where the module to be loaded into the

PE/VM is located. This location is in either application memory or kernel memory. It is not in SMRAM.

`ModuleLoadAddress` - Module guest physical load address in PE/VM, must be located within the range AddressSpaceStart : (AddressSpaceStart + AddressSpaceSize)

`ModuleSize` - Module size in bytes

`ModulEntryPoint` - Entry point, relative to the start of the module and located within the module's address range.

`AddressSpaceStart` - Start of the guest physical address space on the PE/VM. The machine physical pages mapped into this guest address space are located within the MSEG. However, they can be mapped anywhere into the guest physical address space. This is the only area within the PE/VM that is not identity mapped. Since the physical pages are located within the MSEG, they are protected in the same manner

`AddressSpaceSize` - Size of the guest physical address space block to be allocated for the executable module and data areas. Realistically, this should be no larger than 1MB as this is limited by how much MSEG space is available after the STM has allocated the space that it needs.

`VMConfig` - Configuration of the PE/VM. Any combination of values that would result in an illegal VM configuration will result in the PE/VM not being started and an error return to the caller. Allowed values are as follows:

`SET_CS_L` (1 << 13) CS.L - set 64-bit mode for CS (valid only when SET_IA32E = 1)

`SET_CS_D` (1 << 14) CS.D - Default mode, 0: 16-bit segment; 1: 32 bit segment. NOTE: CS.D must be zero when CS.L=1 or VMX will not allow the VM to start

`SET_IA32E` (1 << 15) sets IA32 mode; when 1 then CR0.PG, CR0.PE, & CR0.PAE will be set to 1 as well

`SET_CR0_PG` (1 << 31) sets CR0.PG