White Paper

A Tour Beyond BIOS Supporting an SMM Resource Monitor using the EFI Developer Kit II

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

June 2015

Executive Summary

In the current UEFI PI infrastructure, SMM drivers are loaded into SMRAM and execute in a ring-0 privilege environment. However, there are lots of SMRAM based attacks that have occurred in the world [SMM01][SMM02][SMM03][SMM04][SMM05][SMM06]. As such, some platforms might need a way to monitor the SMM driver's actions and block some malicious behavior. Some ideas to provide a least-privilege environment where SMM code behavior can be mediated have been presented before [SMM07][SMM08]. In this paper we explore the creation of a "monitor" management protocol in SMM to resolve this problem and scale as a solution for all possible SMM implementations.

Prerequisite

This paper assumes that audience has EDKII/UEFI firmware development experience [UEFI][UEFI PI Specification] and IA32 SMM knowledge [IA32 Manual]. He or she should be familiar with the UEFI/PI firmware infrastructure (e.g., PEI/DXE) and know the IA32 SMM driver flow. [UEFI Book]

Table of Contents

Overview

Introduction to SMM

System Management Mode (SMM) is a special-purpose operating mode in Intel® architecture based CPUs. SMM was designed to provide for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code. It is intended for use only by system firmware from the manufacturer and not intended to be 3rd party extensible. [IA32 Manual]

Introduction to PI SMM

In order to support SMM in system firmware, [UEFI PI Specification] Volume 4 describes detailed infrastructure on how to support SMM in UEFI PI-based firmware. See below figure 1 for details.

The SMM Initial Program Loader (IPL) will load the SMM Foundation into SMM memory (SMRAM) and the SMM services will start at that time until a system shutdown.

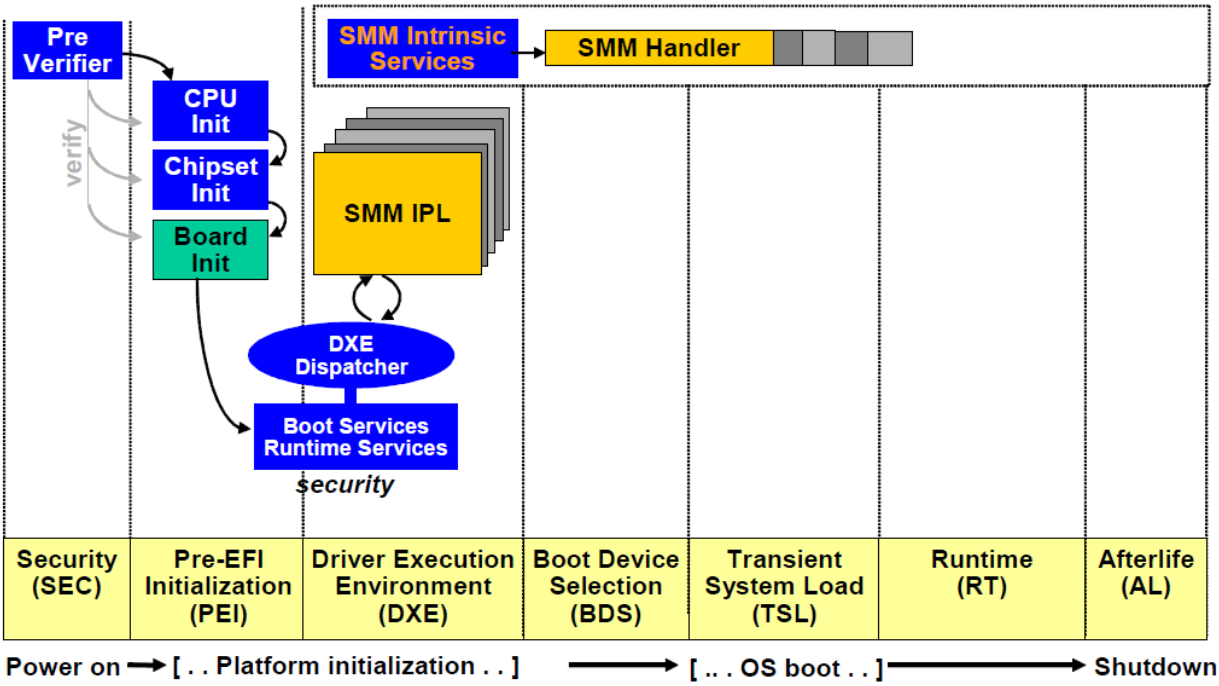

Figure 1 SMM architecture

Introduction to EDKII

EDKII is open source implementation of UEFI PI-based firmware which can boot multiple UEFI-aware operating systems. The EDKII open source project includes the SMM infrastructure which follows the PI specification to provide a capability for loading SMM drivers in the DXE phase of execution.

Summary
This section provided an overview of SMM and EDKII.

PI SMM Execution Environment

In this document we refer to the current PI1.4 SMM specification. A brief introduction is given below. For more details, please refer to [UEFI PI Specification] Volume 4, [IA32 manual] Volume 3 Chapter 34, and [SMM09].

SMM CPU

System Management Mode (SMM) is a generic term used to describe a unique operating mode of the processor which is entered when the CPU detects a special high priority System Management Interrupt (SMI). Upon detection of an SMI, a CPU will switch into SMM, jump to a pre-defined entry vector and save some portion of its state (the "save state") such that execution can be resumed later via the invocation of the RSM instruction.

The SMM CPU driver provides the entry vector, and the SMM CPU driver will transfer control to the SMM Foundation. Once the SMM Foundation returns, the SMM CPU driver will exit SMM mode.

SMM Foundation

The SMM foundation provides the PI SMM infrastructure. We also call it the "PI SMM Core."
https://github.com/tianocore/edk2-MdeModulePkg/tree/master/Core/PiSmmCore

The SMM foundation publishes the System Management System Table (SMST). The SMST is a set of capabilities exported for use by all drivers that are loaded into System Management RAM (SMRAM). The SMST manages the following:
• Dispatch of drivers in SMM
• Allocations of SMRAM
• Installation/discovery of SMM protocols

SMM Driver

The SMM driver is a component that runs inside of the PI SMM infrastructure. The SMM driver is discovered by the SMM Foundation and loaded into SMRAM.

The main responsibility of the SMM driver is to handle different SMI events. For example:
• A generic SMM driver may handle the SetVariable request from a UEFI runtime driver.
• A silicon SMM driver may handle a S3 resume request from silicon.
• A silicon SMM driver may handle hardware error event from hardware component [APEI].
• A platform SMM driver may handle the ENABLE_ACPI software SMI request from the OS.

AN SMM driver may use any services provided by the SMST and any protocol published by another SMM driver. So in most cases, the SMM driver is running in same privilege level as the SMM foundation.

Summary
This section provides a brief introduction to the current PI1.4 SMM infrastructure.

Technique to support SMM Monitor

The purpose of this white paper is to summarize possible ways to support a monitor to SMM driver to see if it does thing right. To that end, there are several choices listed below.

SMM transfer monitor (STM)

[IA32 Manual] Volume 3, 34.15 describes the Dual Monitor treatment of SMIs and SMM.
A dual-monitor treatment is activated through the cooperation of the **executive monitor** (the VMM that operates outside of SMM to provide basic virtualization) and the **SMM-transfer monitor** (**STM**). The STM is the VMM that operates inside SMM, while in VMX operation, to support system-management functions. Control is transferred to the STM through VM exits. VM entries are used to return from SMM. For more information please also refer to [TrustedPlaform] and [SMM08].

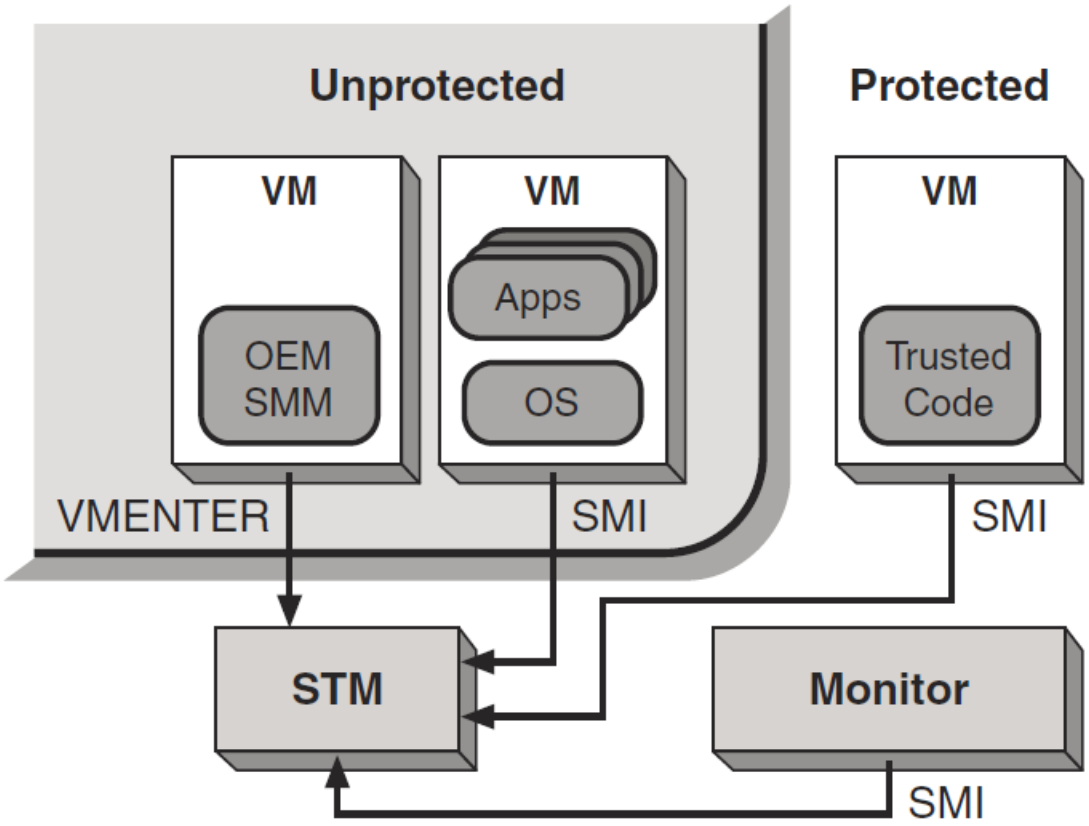

Figure 2 STM architecture

By using the STM, we need to provide a standalone STM in the BIOS, and then depriviledge the SMM CPU, SMM foundation and the platform's SMM drivers.

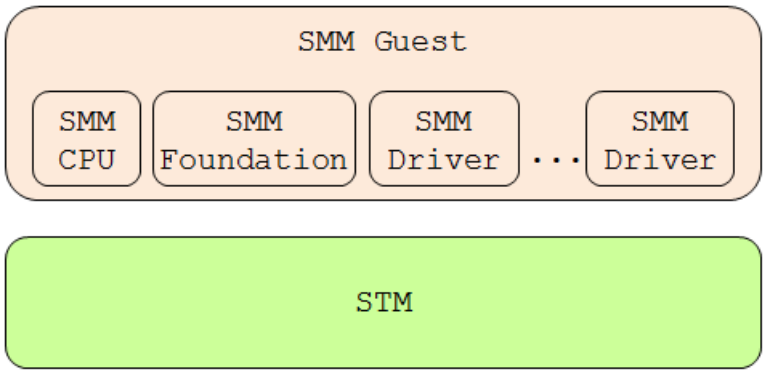

Figure 3 PI-SMM STM Solution

Below provides the various aspects of the STM solution.
Pros: The STM is a full virtualization solution. It provides the highest security, as long as we define a policy.
Cons: The STM relies on CPU Virtualization Technology (VT) and the OS must enable VT before enabling STM.

Ring-3

If a processor does not support STM, the other way to de-privilege SMM driver is to make it run in Ring-3 environment. A possible way is to put SMM CPU driver in Ring0, and then put the SMM foundation and SMM drivers into Ring3.

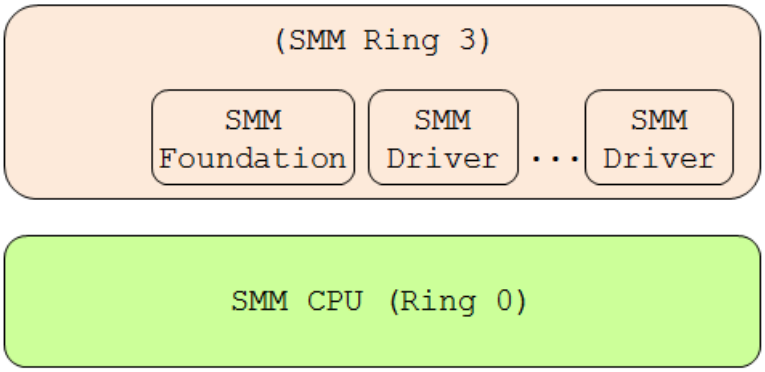

Figure 4 PI-SMM Ring-3 Solution

Pros: Ring3 exits on all processors. It does not rely on any CPU advanced feature.
Cons: Ring3 solution has some know limitations for security. See [IA VMM]. Also, the privilege transitions can be complex to manage.

Page Table

If it is too hard to enable a ring3 solution, the simpler way is to build a page table to prevent SMRAM access. The SMM CPU driver can setup page table with Execution Disable (XD), Write Protection (WP) enabled. Some uses of paging in EDK II has been described in [MEMORY].

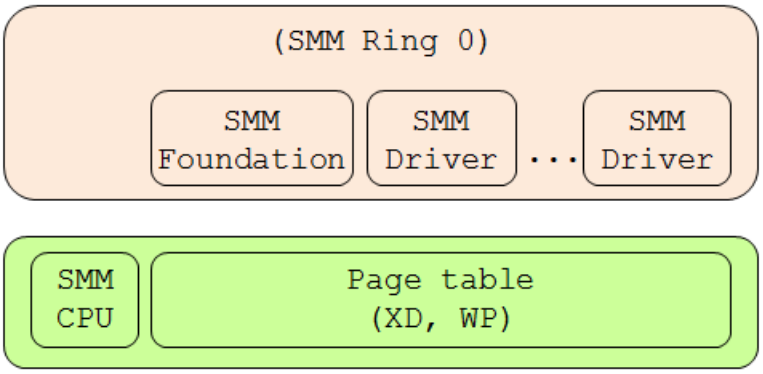

Figure 5 PI-SMM Page Table Solution

Pros: Simpler than Ring3. Already implemented.
Cons: Paging only provides memory protection. The is no IO resource protection. This solution assumes the SMM driver is good actor and the SMM driver will not modify the page table.

Code Access Control MSR

The simplest way to prevent SMM code call-out issue is to enable an MSR - MSR_SMM_FEATURE_CONTROL [IA32 SDM]. When the SMM handler code execution check is enabled, an attempt by the SMM handler to execute outside the ranges specified by SMRR will cause the assertion of an unrecoverable machine check exception (MCE).

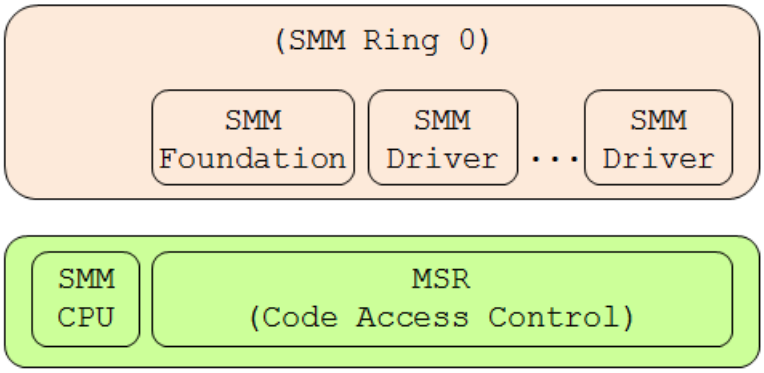

Figure 6 PI-SMM Code Access Control Solution

Pros: Simplest way to prevent SMM call-out.
Cons: This is advanced CPU feature that does not exist for all CPUs. It only handles code access (execute), but it cannot handle data access (read/write) outside SMRAM.

EFI Byte Code (EBC)

EBC is a pure software architecture defined in the UEFI specification. We provide the EBC software-based Virtual Machine Monitor (VMM) as a standalone SMM driver https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/EbcDxe. Using EBC, a normal SMM driver can be compiled with the EBC compiler so that SMM driver will be

executed in EBC Virtual Machine (VM). The software interpreter can be augmented to perform the appropriate RWX checks for a given policy.

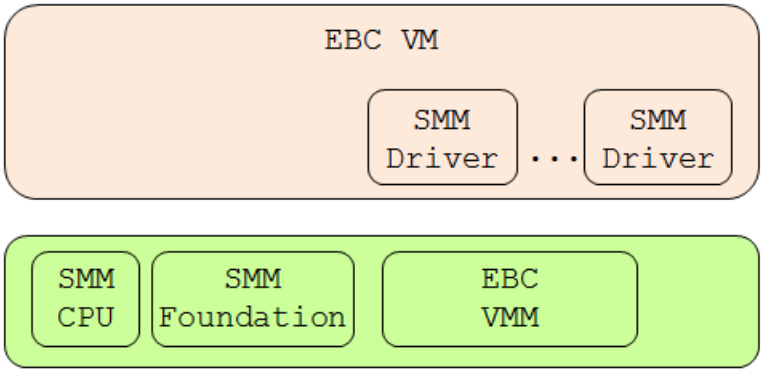

Figure 7 PI-SMM EBC Solution

Pros: EBC does not rely on any hardware feature.
Cons: We still need a way to prevent the SMM VM to call SMST services to access system resources, such as the SmmCpuIo protocol, and SmmPciIo protocol.

Summary
This section provides a survey of existing techniques to support monitoring behavior of SMM drivers inside of SMM.

PI SMM Monitor

Since there are different ways to implement an SMM Monitor for PI SMM, we might want to provide a generic protocol to abstract the implementation.

SMM_MONITOR_INIT_PROTOCOL

This protocol is published inside of PI-SMM. The SMM Monitor loader is the producer of the service. An SMM Driver is the consumer of the service.

```
typedef struct _EFI_SMM_MONITOR_INIT_PROTOCOL {
  EFI_SMM_MONITOR_LOAD_MONITOR          LoadMonitor;
  EFI_SMM_MONITOR_ADD_BIOS_RESOURCE     AddBiosResource;
  EFI_SMM_MONITOR_DELETE_BIOS_RESOURCE  DeleteBiosResource;
  EFI_SMM_MONITOR_GET_BIOS_RESOURCE     GetBiosResource;
  EFI_SMM_MONITOR_GET_MONITOR_STATE     GetMonitorState;
} EFI_SMM_MONITOR_INIT_PROTOCOL;
```

The key function provided by this protocol is to **let an SMM driver declare which resource will be access by SMM driver during SMM runtime**. For example:
• If SMM driver needs to access the SMM communication buffer – 0x67890000~0x67990000, it would invoke `AddBiosResource()` to declare this memory resource, with attribute WRITE_ACCESS/READ_ACCESS. But BIOS should never declare memory resources to be EXECUTION_ACCESS, because executing non-SMRAM code will possibly lead to a security issue.
• If an SMM driver needs to access the ACPI Timer IO port – 0x400, it will call `AddBiosResource()` to declare the IO resource.
• If an SMM driver needs to access the LPC PCI configuration space - Bus(0)/Device(1F)/Function(0), it needs to call `AddBiosResource()` to declare the PCI resource.
• If an SMM driver needs to access a CPU MSR – 0x40E, it needs to call `AddBiosResource()` to declare the CPU MSR resource.

NOTE: There is no need for an SMM driver to declare SMRAM because SMRAM is invisible to OS.

Other APIs include `LoadMonitor()`, which provides a function to load the monitor to a specific SMRAM location, and `GetMonitorState()`, which returns if the monitor is loaded or activated.

SMM_MONITOR_SERVICE_PROTOCOL

This protocol is published as a UEFI protocol. The SMM Monitor UEFI Helper driver is the producer. The OS loader or kernel can be the consumer.

```
typedef struct _EFI_SMM_MONITOR_SERVICE_PROTOCOL {
  EFI_SMM_MONITOR_START                 Start;
  EFI_SMM_MONITOR_STOP                  Stop;
  EFI_SMM_MONITOR_PROTECT_OS_RESOURCE   ProtectOsResource;
```

```
  EFI_SMM_MONITOR_UNPROTECT_OS_RESOURCE     UnprotectOsResource;
  EFI_SMM_MONITOR_GET_OS_RESOURCE           GetOsResource;
  EFI_SMM_MONITOR_RETURN_BIOS_RESOURCE      ReturnBiosResource;
  EFI_SMM_MONITOR_INITIALIZE_PROTECTION     InitializeProtection;
  EFI_SMM_MONITOR_ADD_OS_TRUSTED_REGION     AddOsTrustedRegion;
  EFI_SMM_MONITOR_DELETE_OS_TRUSTED_REGION DeleteOsTrustedRegion;
  EFI_SMM_MONITOR_GET_OS_TRUSTED_REGION     GetOsTrustedRegion;
} EFI_SMM_MONITOR_SERVICE_PROTOCOL;
```

The key function provided by this protocol is to **let the OS loader or kernel declare which OS resources are critical such that the BIOS SMM handler should not touch**. For example:
• If the OS kernel is loaded into memory – 0x34560000~0x56780000, it can call `ProtectOsResource()` to declare this memory resource, with attribute WRITE_ACCESS/READ_ACCESS/EXECUTION_ACCESS, which means the BIOS should never write/read/execute data in this region.

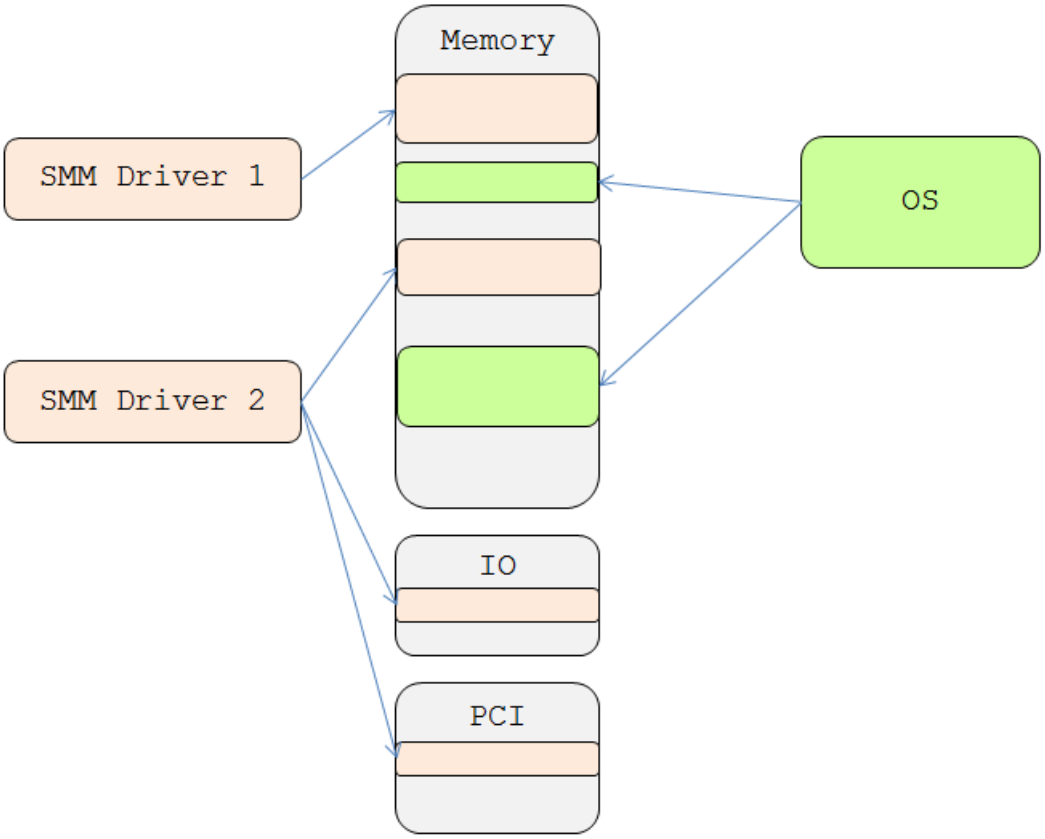

Figure 8 PI-SMM Monitor resource declaration

In figure 8, the RED part designates the resource declared by the BIOS SMM driver. The GREEN part designates a resource declared by OS.

Before the OS loader calls `ProtectOsResource()`, it may call `GetOsResource()` to see if some resource is already claimed by the BIOS to avoid a resource conflict. The SMM Monitor needs reject the `ProtectOsResource()` request if it detects resource conflict. Then the OS loader needs to be aware of that conflict and decide what to do as a next step. The OS loader may choose to refuse booting, or continue to boot without protection, based upon policy.

NOTE: BIOS SMM and OS may declare same resource but with different attribute flag – WRITE_ACCESS/READ_ACCESS. This case is considered as legal, and will not result in resource conflict.

For example:
• If 0x5A5A0000~0x5A5B0000 is data OS passed to the BIOS SMM, the OS can set WRITE_ACCESS and the BIOS SMM can set READ_ACCESS. Then this region is write-only for the OS, while it is read-only for BIOS SMM.
• If 0x5A5C000~0x5A5D0000 is data that the OS retrieves from the BIOS SMM, the OS can set READ_ACCESS and the BIOS SMM can set WRITE_ACCESS. Then this region is read-only for OS, while it is write-only for the BIOS SMM.

The other key function provided by this protocol is to **let the OS kernel declare which OS resource are considered as a "trusted region"**. Only the software SMI generated in the trusted region will be considered as a valid software SMI. For example:
• If the OS kernel is loaded into memory – 0x34560000~0x56780000, it can call `AddOsTrustedRegion()` to declare this memory resource.

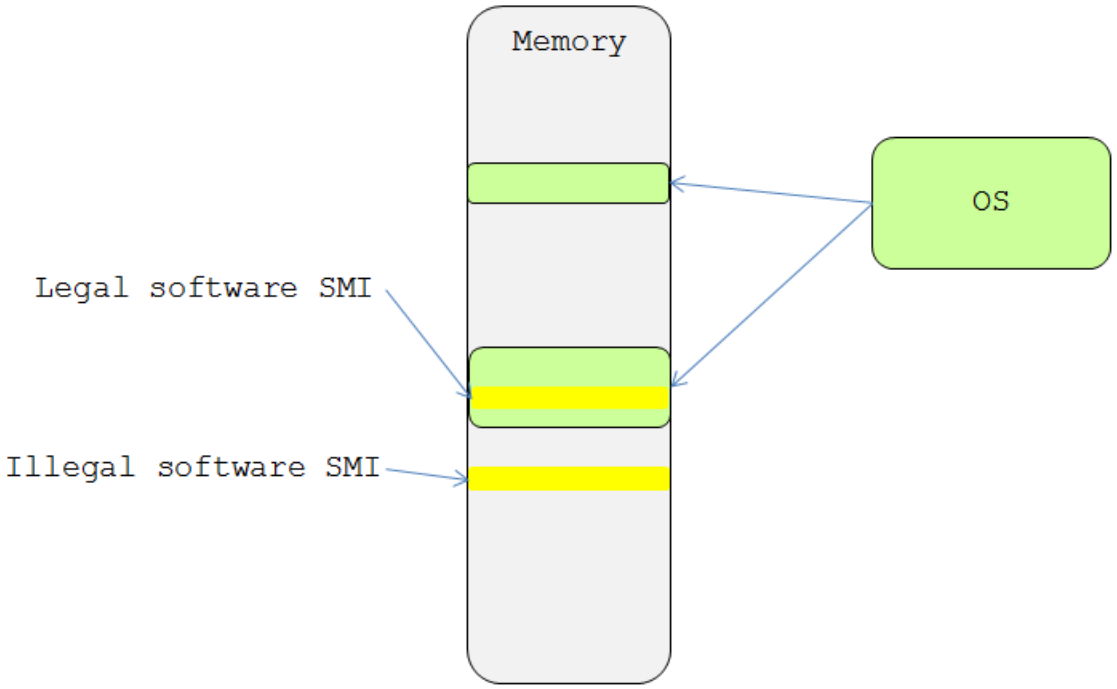

Figure 9 PI-SMM Monitor trusted region declaration

See figure 9. GREEN part means OS trusted region. YELLOW part means CPU instruction pointer (IP) generating this software SMI.

When the SMM monitor detects a software SMI generated, it will also check the CPU IP generating this software SMI. If the IP is inside OS trusted region, it means the SMI was generated as expected, and then the SMM monitor will forward request to corresponding software SMI handler. If not, it means the SMI was generated by unknown region (maybe malicious code), and the SMM monitor will ignore this software SMI request.

Other APIs, `Start()` means to enable SMM monitor. `InitializeProtection()` means to start protect the resource declared in `ProtectResource()`.

BIOS responsibility

The BIOS responsibilities include:
• SMM monitor loader driver produces SMM_MONITOR_INIT_PROTOCOL in the PI-SMM protocol database to let SMM driver consume it.
• All SMM drivers consume SMM_MONITOR_INIT_PROTOCOL to register system resources needed during SMM runtime.
• One SMM platform driver consumes SMM_MONITOR_INIT_PROTOCOL to load the SMM Monitor.
• The SMM monitor UEFI helper driver produces the SMM_MONITOR_SERVICE_PROTOCOL in UEFI protocol database to let the OS consume the service.

NOTE: Although the rules are defined by the SMM drivers and OS loader, an SMM monitor can use some default policy to mark all non-SMRAM memory regions to be Non-Executable. It can provide security by default. This may help with older OS's that are not aware of this capability but for whom such SMM driver protection may be of interest.

OS responsibility

The OS responsibilities include:
• The OS consumes the SMM_MONITOR_SERVICE_PROTOCOL to start the SMM monitor.
• The OS consumes the SMM_MONITOR_SERVICE_PROTOCOL to declare OS critical resources.
• The OS consumes the SMM_MONITOR_SERVICE_PROTOCOL to declare OS trusted regions.
• The OS consumes the SMM_MONITOR_SERVICE_PROTOCOL to initialize protection.

Summary
This section introduces the PI SMM monitor services and typical usage. In the next 2 sections, we will choose 2 typical SMM monitors and introduce how to implement them in the current PI SMM infrastructure.

STM as a monitor

First we will introduce how to use the STM to implement the SMM monitor API.

SMM_MONITOR_INIT_PROTOCOL

The SMM CPU driver could be the producer of the SMM_MONITOR_INIT_PROTOCOL API. During boot, one platform SMM driver will locate the STM binary image in flash and invoke LoadMonitor(), after which the SMM CPU driver will put the whole STM image to a region of SMRAM called the **monitor segment** (MSEG). All code and data for the STM reside in MSEG. Then, the SMM CPU code specifies the location of MSEG by writing to the IA32_SMM_MONITOR_CTL MSR. The format of MSEG header is defined in [IA32 SDM]. It includes:

- MSEG-header revision identifier.
- SMM-transfer monitor features. (IA32 mode or X64 mode)
- GDTR limit.
- GDTR base offset.
- CS selector.
- EIP offset.
- ESP offset.
- CR3 offset.

All of the above information is inside of the STM binary.

During boot, every SMM drivers also needs to call `AddBiosResource()` to report resource information to the SMM CPU driver. The SMM CPU driver saves the information to a special place – BIOS resource list.

SMM_MONITOR_SERVICE_PROTOCOL

According to [IA32 SDM], the STM is activated only if it is enabled and only by the executive monitor. The executive monitor activates the STM by executing a VMCALL in VMX root operation.

A special SMM Monitor UEFI helper driver could be the producer of the SMM_MONITOR_SERVICE_PROTOCOL. All APIs in this protocol will use a VMCALL instruction to communicate with the STM. The STM implementation should know the VMCALL index and jump to corresponding VMCALL handler and return information.

For example, `ProtectOsResource()` will pass a list of resource declared by the OS. The STM will go through the list and maintain its own OS resource list inside of MSEG.

STM memory layout

See below figure 10. The DARK GREEN part is the static STM image, and the LIGHT GREEN part is the MSEG region used by the STM image.

The static STM image includes the following components:

• MSEG header (layout defined in [IA32 SDM], described in the above section).
• GDT
• STM code/data – PE/COFF image
The rest part of MSEG includes the following parts:
• Page Table for STM. It is created by the STM loader – the SMM CPU driver.
• Heap
• Stack – each processor has its own stack.
• SMI VMCS (VMCS for executive monitor) – each processor has its own SMI VMCS.
• SMM VMCS (VMCS for SMM guest) – each processor has its own SMM VMCS.

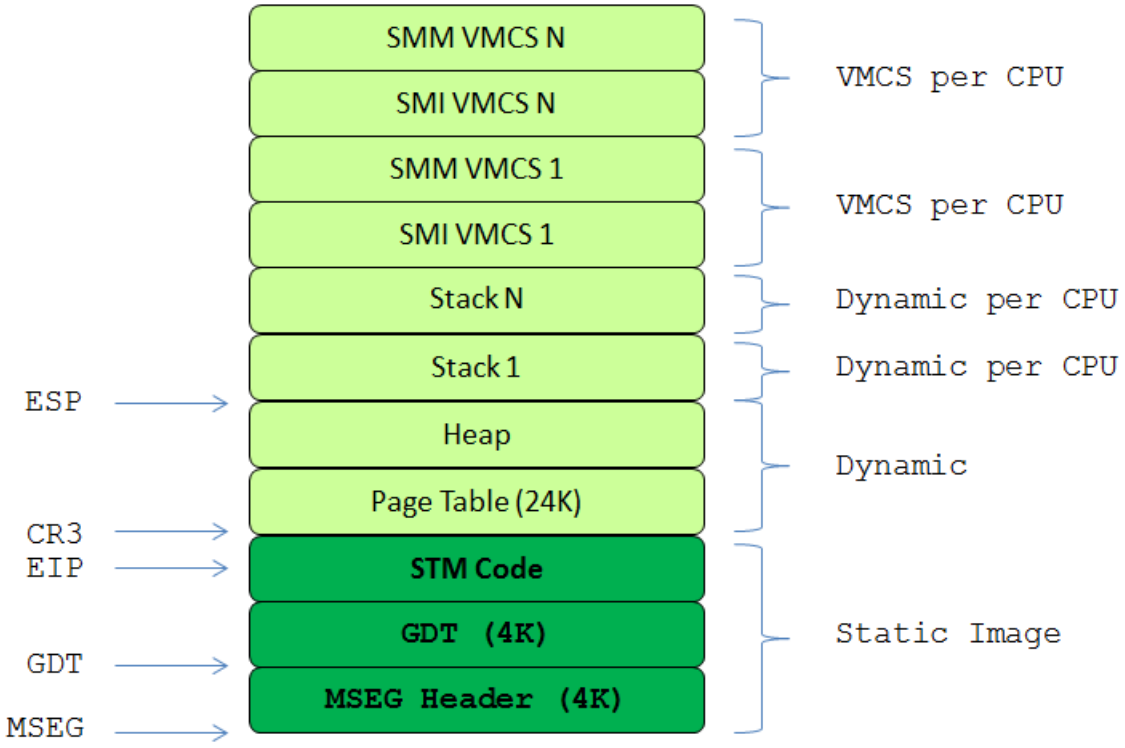

Figure 10 STM memory layout

NOTE: Only the MSEG header is mandatory to follow the IA32 SDM. All of the following portions can be implementation specific.

All of the above memory layout will be constructed in the first VMCALL when the STM is activated. The subsequent VMCALL's or SMI's will follow the STM runtime flow.

STM runtime flow

During runtime, the system enters the STM when an SMM VM occurs. An SMM VM exit is a VM exit that begins outside SMM and that ends in SMM.

See figure 11.

• SMM VM exits result from the arrival of an SMI outside SMM or from the execution of a VMCALL in VMX root operation outside SMM. Execution of the VMCALL in VMX root operation causes an SMM VM exit only if the valid bit is set in the IA32_SMM_MONITOR_CTL MSR.
• The STM will sync SMM state from the SMI VMCS to the SMM save state memory or MSR.
• Then the STM will store the current SMI VMCS and load the SMM VMCS.
• At this point, the STM VM passes control to the SMM guest, the BIOS SMM entry point, to finish the transfer of control for this SMI to the SMM guest.

• Once the BIOS SMM finishes its operation, the BIOS SMM code will execute an RSM. RSM causes a VM Exit back to the STM again.
• Then the STM will store the current SMM VMCS and load the SMI VMCS.
• The STM will then sync the SMM state from the SMM save state memory or MSR to the SMI VMCS.
• Finally, the STM VM passes control to original SMI source, which could be a VMM or a VM Guest.

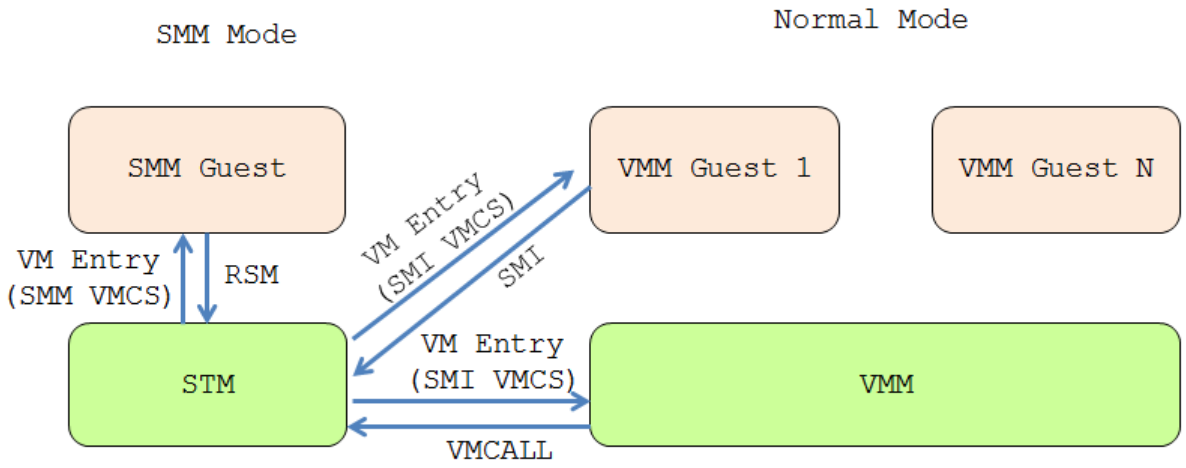

Figure 11 STM runtime

NOTE: Here we only have one SMM Guest, which is BIOS SMM code. It is also legal to have multiple SMM guests. Some SMM guests can be used for special purposes, such as perform sanity checks of the system, record errors, or other system maintenance activity.

STM as monitor

Now the question is: How do we implement the STM to be a monitor?

1) OS resource protection

The STM can get BIOS resources in SMM based upon a predefined location in SMRAM. The STM can also get OS resources based on a VMCALL by OS. Then the STM sets the SMM VMCS based on BIOS resources and OS resources, respectively. For example:

• If the OS requests blocking some pages of memory, the STM will update EPT entry in the VMCS to remove the read access, write access, or execute access bits. If the BIOS SMM handler still accesses these resources, an EPT violation will be triggered. This triggering is a VM exit event, and the system will return back to the STM from the BIOS SMM Guest.
• If the OS requests blocking some IO port, the STM will update the I/O bitmap in the VMCS to remove IO access. If the BIOS SMM handler attends to access these resource, an IO instruction VM exit event will be triggered, and this will cause system to return back to the STM from the BIOS SMM Guest.

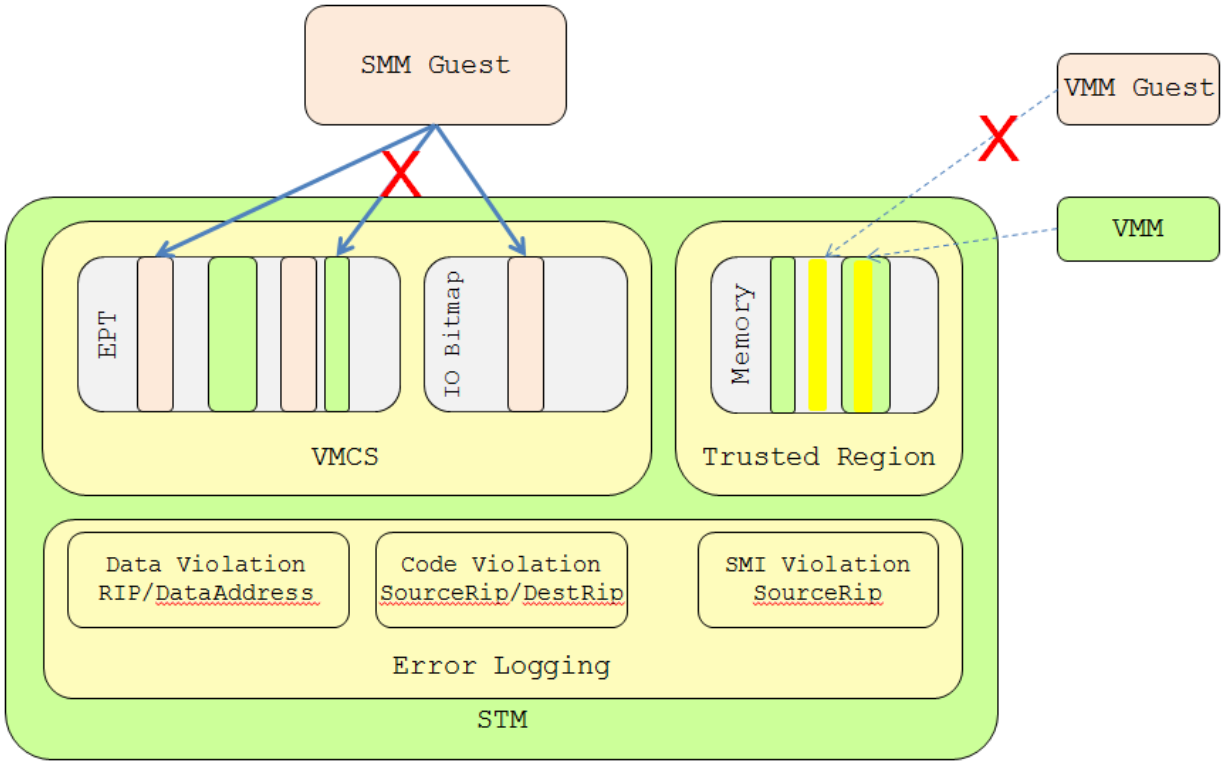

Figure 12 STM as Monitor

2) OS trusted region support

The STM consults the RIP when the SMI VM exit is caused by an IO port access. If the SMI VM exit is caused by a software SMI (e.g. port 0xB2), the STM will check if the source RIP is in an OS trusted region. If it is located within a trusted region, the STM will forward the request to the SMM guest. If not, the STM will ignore this SMI and return back to the SMI guest directly.

3) Error Logging

The STM can also implement error log to record all violations. For example:
• If SMM Guest runs code outside of SMRAM, it will trigger EPT violation. Then STM will record CodeViolation event, with source RIP and destination RIP.
• If SMM Guest accesses OS protected data, it will trigger EPT violation. Then STM will record DataViolation event, with RIP and data address.
• If SMI is not trigged in OS trusted region, STM will record SmiViolation event, with source RIP.

Summary

This section introduces STM and describes how to use the STM as a PI SMM monitor.

Page Tables as a monitor

If the processor does not support an STM, we can set up the page tables to monitor an SMM driver.

SMM_MONITOR_INIT_PROTOCOL

The SMM CPU driver can be the producer of the SMM_MONITOR_INIT_PROTOCOL API. The SMM drivers then call `AddBiosResource()` to report resource information to the SMM CPU driver. The SMM CPU driver saves the resource information into SMRAM.

SMM_MONITOR_SERVICE_PROTOCOL

A special SMM Monitor UEFI helper driver can be the producer of the SMM_MONITOR_SERVICE_PROTOCOL. All APIs in this protocol will use software SMI's to communicate with the SMM CPU driver.
For example, `ProtectOsResource()` will pass a list of resources declared by OS into SMM. The SMM CPU driver will go through the list and maintain its own OS resource list in SMRAM.

Page tables as monitor

How we use page table to make the SMM CPU driver to be a monitor?

1) OS resource protection

The SMM CPU driver may consult BIOS resource list and OS resource list to modify the BIOS SMM page table. For example:
• If the OS requests blocking some pages of memory, the SMM CPU driver will update the SMM page tables to clear present bit, clear R/W bit, or set NX bit. If the BIOS SMM handler still accesses these resources, a page fault exception will be triggered. At this point, the SMM CPU driver can check the exception type to know what happened and then record an error or perform some other mitigation.

The only limitation here is that the page tables can only handle memory protection. They cannot handle IO protection.

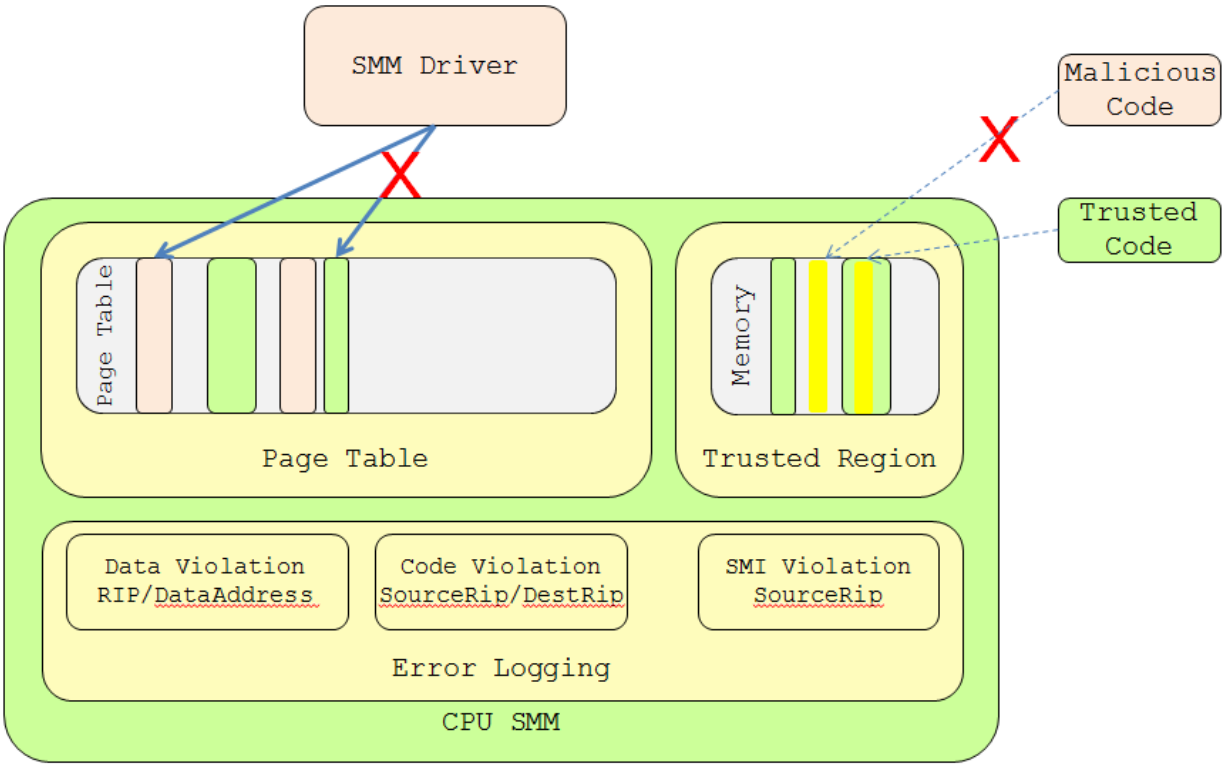

Figure 13 Page Table as Monitor

2) OS trusted region support

The SMM CPU consults the RIP when an SMI is triggered by an IO write. If the SMI is caused by a software SMI (e.g. port 0xB2 access), the SMM CPU will check the SMM save state to determine if the source RIP is within an OS trusted region. If yes, the SMM code will forward the request to the SMM foundation. If no, the STM will ignore this SMI and just issue an RSM to return directly back to the normal operational mode.

3) Error Logging

The SMM CPU driver can also implement an error log to record all violations. For example:

• If SMM driver runs code outside of SMRAM, it will trigger a Page Fault exception. Then the SMM CPU driver will record the CodeViolation event, with the source RIP and destination RIP.

• If the SMM driver accesses OS protected data, it will trigger a Page Fault exception. Then the SMM CPU driver will record a DataViolation event, with the RIP and data address.

• If the SMI is not trigged in an OS trusted region, the SMM CPU driver will record a SmiViolation event, with the source RIP.

Summary

This section introduces how the SMM CPU driver uses page tables as a PI SMM monitor.

Conclusion

The SMM Monitor provides the capability of a reference monitor [MONITOR] for SMM drivers. This monitor can detect improper behavior on the part of SMM drivers. We describe a PI SMM Monitor infrastructure based upon the UEFI PI specification. We also provide a survey of 5 different SMM monitors and introduce how to enable an STM and page tables as embodiments of a PI SMM monitor.

Glossary

DEP – Data Execution Protection.

EBC – EFI Byte Code. See [UEFI Specification].

EPT – Extended Page Table. See [IA32 SDM]

IPL – Initial program loader.

MSEG – Monitor Segment. A special SMRAM for STM.

NX – No Execution. See DEP.

PI – Platform Initialization. Volume 1-5 of the UEFI PI specifications.

SMI – System Management Interrupt. The interrupt to trigger processor into SMM mode.

SMM – System Management Mode. x86 CPU operational mode that is isolated from and transparent to the operating system runtime.

SMRAM – System Management RAM. The memory reserved for SMM mode.

SMRR – System Management Range Register.

STM – SMI Transfer Monitor.

UEFI – Unified Extensible Firmware Interface. Firmware interface between the platform and the operating system.

XD – Execution Disable. See DEP.

VMCS – Virtual Machine Control Structure. See [IA32 SDM]

VT – Virtualization Technology. See [IA32 SDM]

WP – Write Protection.

References

[ACPI] Advanced Configuration and Power Interface, vesion 6.0, www.uefi.org

[APEI] Sakthikumar, Zimmer, "A Tour Beyond BIOS Implementing the ACPI Platform Error Interface with the Unified Extensible Firmware Interface," January 2013, https://firmware.intel.com/sites/default/files/resources/A_Tour_beyond_BIOS_Implementing_APEI_with_UEFI_White_Paper.pdf

[EDK2] UEFI Developer Kit www.tianocore.org

[EDKII specification] A set of specification describe EDKII DEC/INF/DSC/FDF file format, as well as EDKII BUILD. http://tianocore.sourceforge.net/wiki/EDK_II_Specifications

[EMBED] Sun, Jones, Reinauer, Zimmer, "Embedded Firmware Solutions: Development Best Practices for the Internet of Things," Apress, January 2015, ISBN 978-1-4842-0071-1

[IA32 Manual] Intel® 64 and IA-32 Architectures Software Developer Manuals http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[IA VMM] John Scott Robin & Cynthia E. Irvine , "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor", 2000, http://www.usenix.org/events/sec2000/full_papers/robin/robin.pdf

[MEMORY] Yao, Zimmer, Fleming, "A Tour Beyond BIOS Memory Practices in UEFI", June 2015 https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Memory_Practices_with_UEFI.pdf

[MONITOR] Anderson, "Computer Security Technology Planning Study," ESD-TR-73-51, US Air Force Electronic Systems Division, 1973 http://csrc.nist.gov/publications/history/ande72.pdf

[SMM01] Oleksandr Bazhaniuk, et al, "A New Class of Vulnerabilities in SMI Handlers", 2015, https://cansecwest.com/slides/2015/A%20New%20Class%20of%20Vulnin%20SMI%20-%20Andrew%20Furtak.pdf

[SMM02] Rafal Wojtczuk, et al, "Attacks on UEFI Security", 2015, https://bromiumlabs.files.wordpress.com/2015/01/attacksonuefi_slides.pdf

[SMM03] Corey Kallenberg, et al, "How many million BIOSes world you like to infect?", 2015, http://legbacore.com/Research_files/HowManyMillionBIOSWouldYouLikeToInfect_Full2.pdf

[SMM04] Loïc Duflot, et al, "System Management Mode Design and Security Issues", 2010, http://www.ssi.gouv.fr/uploads/IMG/pdf/IT_Defense_2010_final.pdf

[SMM05] Rafal Wojtczuk, et al, "Attacking Intel BIOS", 2009, http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf

[SMM06] "ASUS Eee PC and other series: BIOS SMM Privilege Escalation Vulnerabilities", 2009, http://www.securityfocus.com/archive/1/505590

[SMM07] Dick Wilkins, "UEFI Firmware –Securing SMM", 2015, http://www.uefi.org/sites/default/files/resources/UEFI_Plugfest_May_2015%20Firmware%20-%20Securing%20SMM.pdf

[SMM08] Rafal Wojtczuk, et al, "Attacking Intel® Trusted Execution Technology", 2009, http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf

[SMM09] Jiewen Yao, Zimmer Vincent, "A_Tour_Beyond_BIOS_Launching_Standalone_SMM_Drivers_in_PEI_using_the_EFI_Developer_Kit_II", 2015, http://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Creating_the_Intel_Firmware_Support_Package_Version_1_1_with_the_EFI_Developer_Kit_II.pdf

[TrustedPlatform] David Grawrock, "Dynamics of a Trusted Platform: A Building Block Approach", 2009, http://www.amazon.com/Dynamics-Trusted-Platform-Building-Approach/dp/1934053171

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.5 www.uefi.org

[UEFI Book] Zimmer, et al, "Beyond BIOS: Developing with the Unified Extensible Firmware Interface," 2nd edition, Intel Press, January 2011

[UEFI Overview] Zimmer, Rothman, Hale, "UEFI: From Reset Vector to Operating System," Chapter 3 of *Hardware-Dependent Software*, Springer, February 2009

[UEFI PI Specification] UEFI Platform Initialization (PI) Specifications, volumes 1-5, Version 1.4 www.uefi.org

Authors

Jiewen Yao (jiewen.yao@intel.com) is an EDKII BIOS architect, EDKII TPM2 module maintainer, ACPI/S3 module maintainer, and FSP package owner with Software and Services Group (SSG) at Intel Corporation. Jiewen is a BIOS security researcher and co-invented a VMM in BIOS in patent US007827371 using PI DXE infrastructure.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Senior Principal Engineer with the Software and Services Group (SSG) at Intel Corporation. Vincent chairs the UEFI Security and Networking Subteams in the UEFI Forum.