Co-processor-based Behavior Monitoring

Application to the Detection of Attacks Against the System Management Mode

Ronny Chevalier [1,2] Maugan Villatel [1] David Plaquin [1] Guillaume Hiet [2]

SILM, November 20, 2019 (Previously presented at ACSAC'17)

[1] HP Labs, United Kingdom (ronny.chevalier@hp.com, maugan.villatel@hp.com, david.plaquin@hp.com)

[2] CIDRE Team, CentraleSupélec/Inria/CNRS/IRISA, France (ronny.chevalier@centralesupelec.fr, guillaume.hiet@centralesupelec.fr)

Introduction

SMM Behavior Monitoring

- Approach overview
- How to define a correct behavior?
- How to monitor?

Evaluation

Related Work

Conclusion

Computers rely on firmware

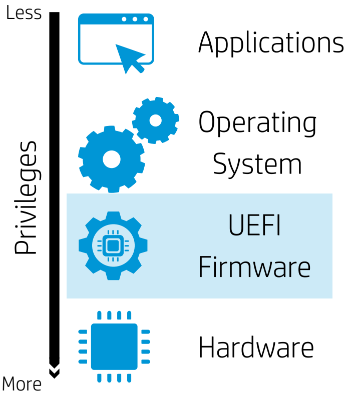
Where can we find firmware?

Mother boards (e.g., BIOS), hard disks, network cards,...

Here, we focus on BIOS/UEFI-compliant firmware

What is it?

- Low-level software
- Tightly linked to hardware
- Early execution
- Highly privileged runtime software
- Stored in a flash



What is the problem?

BIOSs are often written in unsafe languages (i.e., C & assembly)

Memory safety errors (e.g., use after free or buffer overflow)

BIOSs are not exempt from vulnerabilities[1]

Why compromise a BIOS?

- Malware can be hard to detect (stealth)
- Malware can be persistent (survives even if the HDD/SSD is changed) and costly to remove

What do we want?

- Boot time integrity
- Runtime integrity

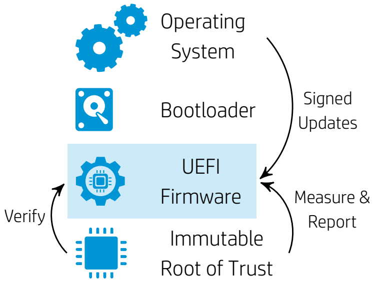
---

[1] Kallenberg et al. 2013; Bazhaniuk et al. 2015.

What are the currently used solutions?

Boot time

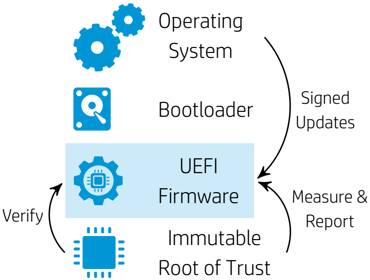
- Signed updates
- Signature verification before executing
- Measurements and reporting to a TPM chip
- Immutable hardware root of trust



What are the currently used solutions?

Boot time

- Signed updates
- Signature verification before executing
- Measurements and reporting to a TPM chip
- Immutable hardware root of trust



Runtime

Isolation of critical services available while the OS is running

→ our focus is on the System Management Mode (SMM)

Introducing the System Management Mode (SMM)

Highly privileged execution mode for x86 processors

Runtime services

BIOS update, power management, UEFI variables handling, etc.

How to enter the SMM?

- Trigger a System Management Interrupt (SMI)
- SMIs code & data are stored in a protected memory region: System Management RAM (SMRAM)

BIOS code is not exempt from vulnerabilities affecting SMM[2]

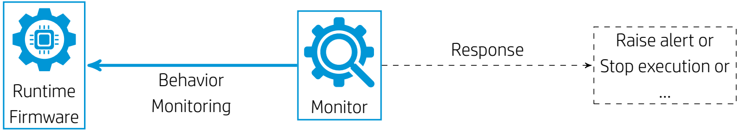
Why is it interesting for an attacker?

- Only mode that can write to the flash containing the BIOS
- Arbitrary code execution in SMM gives full control of the platform

[2] Bazhaniuk et al. 2015; Bulygin, Bazhaniuk, et al. 2017; Pujos 2016.

Our objective

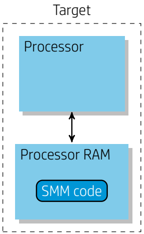
Our goal is to detect attacks that modify the **expected behavior** of the SMM by **monitoring** its behavior **at runtime**.



Such goal raises the following questions:

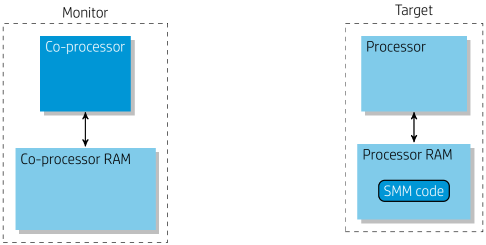
- How to ensure the integrity of the monitor?
- How to define a correct behavior?
- How to monitor?

Approach overview

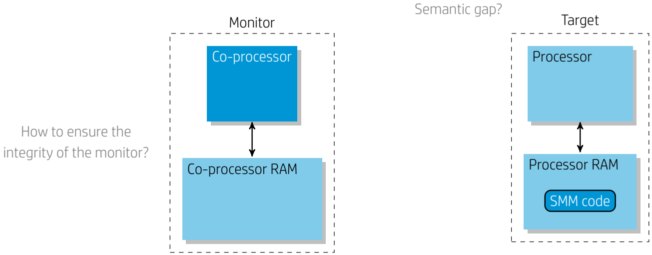


Approach overview

How to ensure the integrity of the monitor?

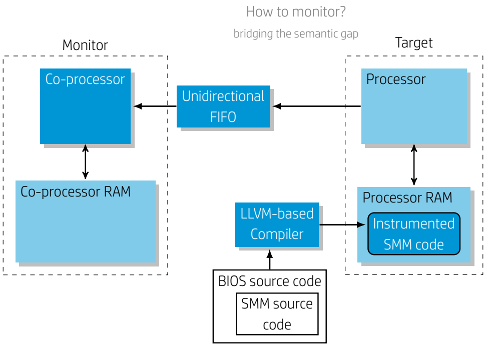


Approach overview

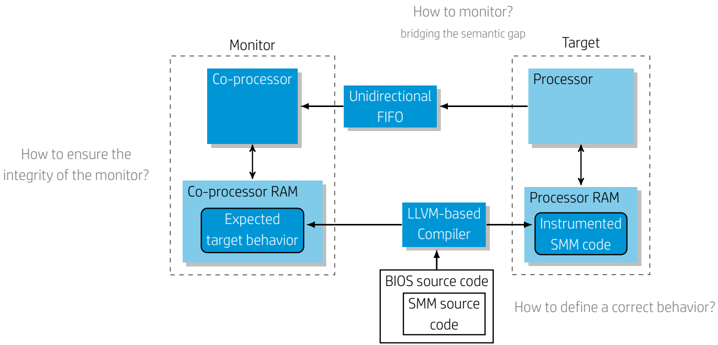


Approach overview

How to ensure the integrity of the monitor?



Approach overview



How to define a correct behavior?

Our use case: SMM code

- Written in unsafe languages (i.e., C & assembly)
  $\rightarrow$ Such languages are often targeted by attacks hijacking the control flow
- Tightly coupled to hardware
  $\rightarrow$ Such software modifies hardware configuration registers

Control Flow Graph (CFG)

Define the control flow that the software is expected to follow

$\rightarrow$ Control Flow Integrity (CFI)

Invariants on CPU registers

Define rules that registers are expected to satisfy

$\rightarrow$ CPU registers integrity

How to define a correct behavior?

Control Flow Integrity (CFI): principle

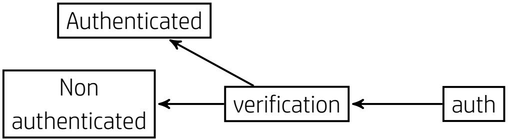
Example

```
void auth(int a, int b) {
    char buffer[512];

    [...vuln...]

    verification(buffer);
}
void verification(char *input) {
    if (strcmp(input, "secret") == 0)
        authenticated();
    else
        non_authenticated();
}
```

Simplified graph



How to define a correct behavior?

Control Flow Integrity (CFI): principle

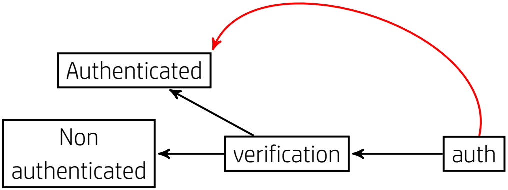
Example

```
void auth(int a, int b) {
    char buffer[512];

    [...vuln...]

    verification(buffer);
}
void verification(char *input) {
    if (strcmp(input, "secret") == 0)
        authenticated();
    else
        non_authenticated();
}
```

Simplified graph



How to define a correct behavior?

Control Flow Integrity (CFI): principle

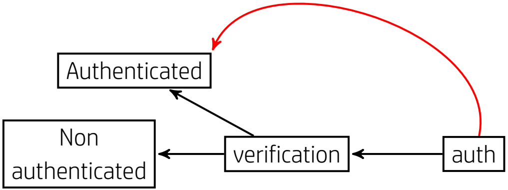
Example

```
void auth(int a, int b) {
    char buffer[512];

    [...vuln...]

    verification(buffer);
}
void verification(char *input) {
    if (strcmp(input, "secret") == 0)
        authenticated();
    else
        non_authenticated();
}
```

Simplified graph



Goal: constrain the execution path to follow a control-flow graph (CFG)

How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

We focus on indirect branches integrity

Type-based verification

Ensures the integrity of indirect calls

```
typedef struct SomeStruct {
  [...]
  char (*foo)(int);
} SomeStruct;
int bar(SomeStruct *s) {
  char c;
  [...]


  c = s->foo(31);
  [...]
}
```

How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

We focus on indirect branches integrity

Type-based verification

Ensures the integrity of indirect calls

```
typedef struct SomeStruct {
  [...]
  char (*foo)(int);
} SomeStruct;
int bar(SomeStruct *s) {
  char c;
  [...]


  c = s->foo(31);
  [...]
}
```

How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

We focus on indirect branches integrity

Type-based verification

Ensures the integrity of indirect calls

```
typedef struct SomeStruct {
  [...]
  char (*foo)(int);
} SomeStruct;
int bar(SomeStruct *s) {
  char c;
  [...]


  c = s->foo(31);
  [...]
}
```

How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

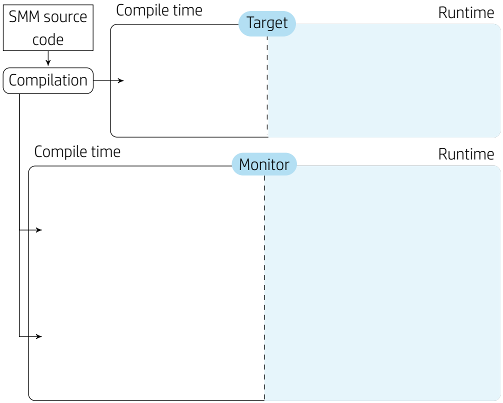
We focus on indirect branches integrity

Type-based verification

Ensures the integrity of indirect calls

```
typedef struct SomeStruct {
  [...]
  char (*foo)(int);
} SomeStruct;
int bar(SomeStruct *s) {
  char c;
  [...]


  c = s->foo(31);
  [...]
}
```



How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

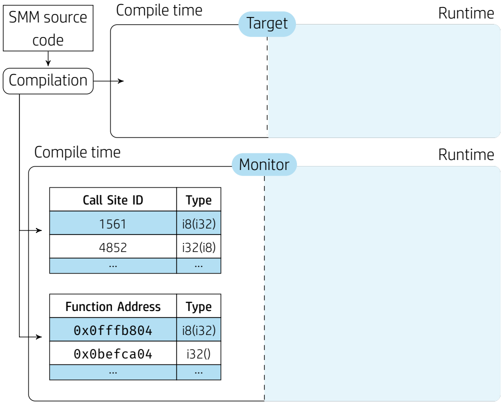
We focus on indirect branches integrity

Type-based verification

Ensures the integrity of indirect calls

```
typedef struct SomeStruct {
  [...]
  char (*foo)(int);
} SomeStruct;
int bar(SomeStruct *s) {
  char c;
  [...]


  c = s->foo(31);  /* Call Site ID = 1561 */
  [...]
}
```



How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

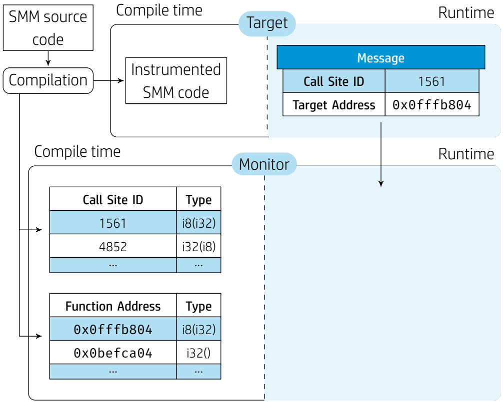
We focus on indirect branches integrity

Type-based verification

Ensures the integrity of indirect calls

```
typedef struct SomeStruct {
  [...]
  char (*foo)(int);
} SomeStruct;
int bar(SomeStruct *s) {
  char c;
  [...]

  [SendMessage(1561, s->foo)]
  c = s->foo(31); /* Call Site ID = 1561 */
  [...]
}
```



How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

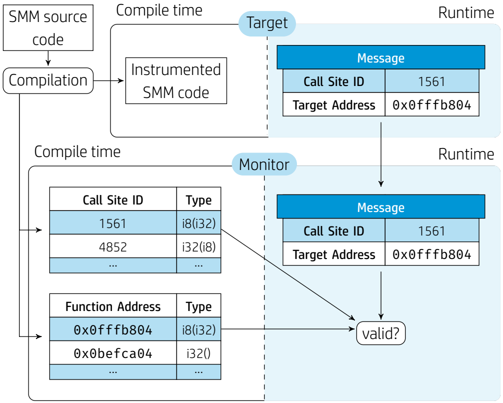
We focus on indirect branches integrity

Type-based verification

Ensures the integrity of indirect calls

```
typedef struct SomeStruct {
  [...]
  char (*foo)(int);
} SomeStruct;
int bar(SomeStruct *s) {
  char c;
  [...]

  [SendMessage(1561, s->foo)]
  c = s->foo(31); /* Call Site ID = 1561 */
  [...]
}
```

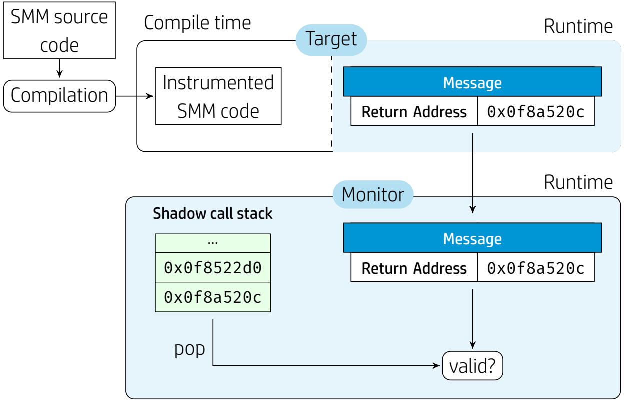


How to define a correct behavior?

Control Flow Integrity (CFI): shadow call stack

Shadow call stack

Ensures integrity of the return address on the stack



How to define a correct behavior?

CPU registers integrity

SMM code is tightly coupled to hardware

- Generic detection methods (e.g., CFI) are not aware of hardware specificities
- Adhoc detection methods are needed

Some interesting registers for an attacker

- **SMBASE**: Defines the SMM entry point
- **CR3**: Physical address of the page directory

$\rightarrow$ Their value is stored in memory and is not supposed to change at runtime

How to protect such registers?

- Send the expected values at boot time
- Send messages at runtime containing these values to detect any discrepancy

Introduction

SMM Behavior Monitoring

- Approach overview
- How to define a correct behavior?
- How to monitor?

Evaluation

Related Work

Conclusion

How to monitor?

Communication channel constraints

Security constraints

- Message integrity
- Chronological order
- Exclusive access

Performance constraints

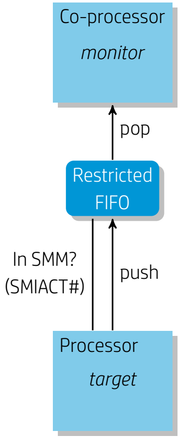
- Acceptable latency of an SMI as defined by Intel BIOS Test Suite: 150 µs
- More than 150 µs per SMI handler leads to degradation of performance or user experience

How to monitor?

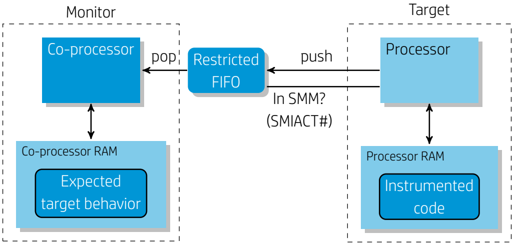
Communication channel design

Additional hardware component

- Chronological order
  → FIFO
- Message integrity
  → Restricted FIFO
- Exclusive access
  → Check if CPU is in SMM (SMIACT# signal)
- Performance
  → Use a low latency interconnect



In summary



We isolate the monitor

- Dedicated co-processor
- Private memory

We bridge the semantic gap

- Communication channel
- Instrumentation of the target code to send messages

We allow the definition of multiple correct behaviors

- Flexible, multiple possibilities
  - CFI
  - CPU registers integrity

Our experimental setup

Our prototype is implemented in a simulated and emulated environment

SMM code implementations used

- EDK2: foundation of many BIOSes (Apple, HP, Intel,...)
  $\rightarrow$ UEFI Variables SMI handlers
- coreboot: perform hardware initialization (used on some Chromebooks)
  $\rightarrow$ Hardware-specific SMI handlers

We want to emulate SMM environment and features

QEMU emulator for security evaluation

We want to simulate accurately the performance impact

gem5 simulator for performance evaluation

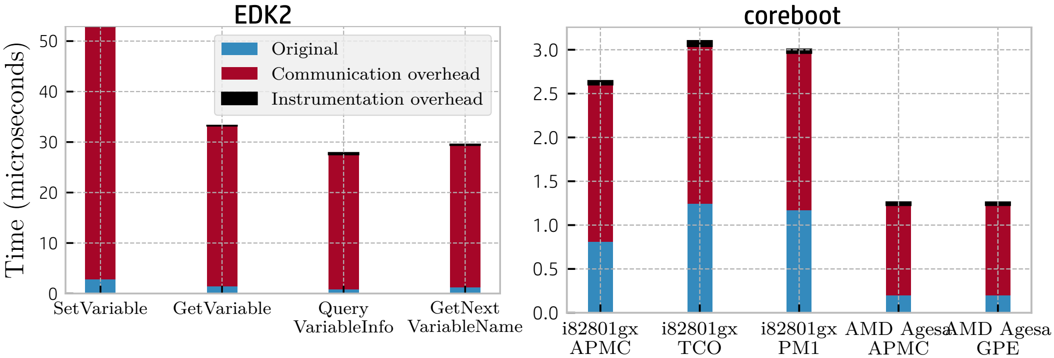
Security evaluation

We simulated attacks & vulnerabilities similar to those found in real-world BIOSes

Vulnerability	Attack Target	Security Advisories	Detected
Buffer overflow	Return address	CVE-2013-3582	Yes
Arbitrary write	Function pointer	CVE-2016-8103	Yes
Arbitrary write	SMBASE	LEN-4710	Yes
Insecure call	Function pointer	LEN-8324	Yes

Performance evaluation

Running time overhead for SMI handlers



- Under the 150 microseconds limit defined by Intel
- Most of the communication overhead is due to the shadow call stack

Introduction

SMM Behavior Monitoring

- Approach overview
- How to define a correct behavior?
- How to monitor?

Evaluation

Related Work

Conclusion

Related work

Snapshot-based approaches

✗ Transient attacks

Copilot [Petroni et al. 2004]

DeepWatch [Bulygin and Samyde 2008]

Event-driven approaches

✓ Detect transient attacks

Ki-Mon [Lee et al. 2013]

✗ Semantic gap

MGuard [Liu et al. 2013]

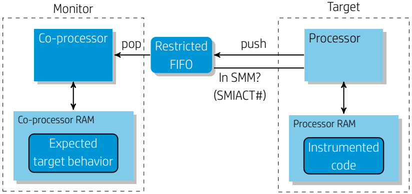
✗ Semantic gap

What did we do? What did we learn?

Our contributions

- Event-based approach to monitor firmware
- Prototype implementing our approach
- Evaluation of our prototype

Our approach



Results

- Detection of state-of-the-art attacks
- Acceptable performance (< 150 µs Intel threshold)

Future work

- Non-control data attacks
- Adaptation to other firmware

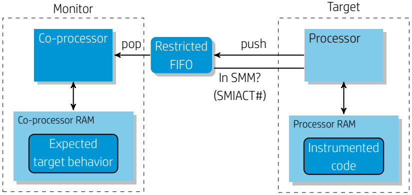
Thanks for your attention!

Questions?

Our contributions

- Event-based approach to monitor firmware
- Prototype implementing our approach
- Evaluation of our prototype

Our approach



Results

- Detection of state-of-the-art attacks
- Acceptable performance (< 150 µs Intel threshold)

Future work

- Non-control data attacks
- Adaptation to other firmware

Backup

Security evaluation

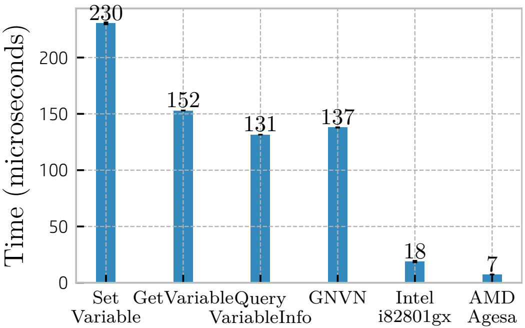
Number and size of equivalence classes for the type-based verification

Our analysis with EDK II gave:

- 158 equivalence classes of size 1,
- 24 of size 2,
- 42 of size 3,
- 2 of size 5,
- 1 of size 9,
- and 1 of size 13.

Performance evaluation

Co-processor time to process messages



Performance evaluation

Number of packets sent due to the instrumentation

SMI Handler	Number of packets sent: Shadow stack (SS)	Indirect call (IC)	SMBASE & CR3 (SC)	Total number of packets
EDK II				
VariableSmm				
SetVariable	384	4	4	392
GetVariable	240	4	4	248
QueryVariableInfo	299	4	4	208
GetNextVariableName	212	4	4	220
coreboot				
Intel i82801gx				
APMC/TCO/PM1	8	2	4	14
AMD Agesa Hudson				
APMC/GPE	4	0	4	8

Table 1: Number of packets sent during one SMI handler (Number of packets per message type: SS=2, IC=2, SC=4)

Threat model & assumptions

The target sends messages to describe its own behavior

Key point

The attacker must alter the control flow (i.e., behavior) in order to forge messages

→ The attacker cannot send messages in lieu of the target **without first being detected**

What are the attacker's capabilities before the attack?

Complete control over the OS (e.g., can trigger as many SMI as necessary)

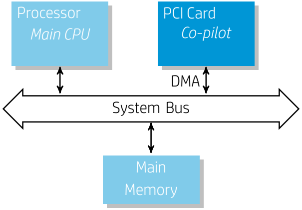
What kind of attack?

Runtime attack by triggering memory corruption issues in an SMI handler (e.g., ROP)

Related work

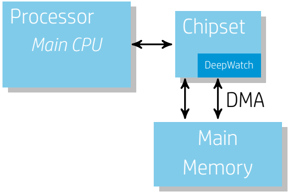
Snapshot-based approach

Copilot [Petroni et al. 2004]



✓ Flexible ✗ Cannot monitor SMM code
✗ Semantic gap ✗ Transient attacks
✗ Additional hardware

DeepWatch [Bulygin and Samyde 2008]

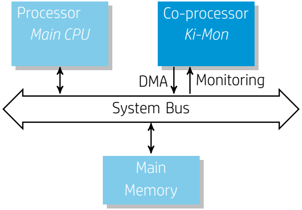


✓ Flexible ✓ Can monitor SMM code
✗ Semantic gap ✗ Transient attacks
✓ No additional hardware

Related work

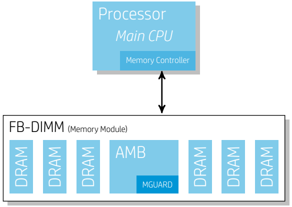
Event-driven approach

Ki-Mon [Lee et al. 2013]



✓ Flexible ✓ Could monitor SMM code
✗ Semantic gap ✓ Detect transient attacks
✗ Additional hardware

MGuard [Liu et al. 2013]



✓ Flexible ✓ Can monitor SMM code
✗ Semantic gap ✓ Detect transient attacks
✗ Requires FB DIMM Memory

Related work

Hardware-based CFI approach

Incorporate CFI policy directly in processors [Davi et al. 2015]

Future CFI technology in Intel processors? [Intel Corporation 2016]

Advantages

✓ Can monitor SMM code

✓ Efficient

✓ Semantic gap

✓ Detect transient attacks

Limitations

✗ Precision loss

✗ Not flexible (i.e., one detection method)

✗ Requires to modify the processor

Communication channel

Mailboxes

High latency

Need to design an intermediate hardware component

Restricted FIFO to store temporarily messages

PCIe

- Designed to maximize I/O throughput
- Not suited to send many small packets (coarse-grained interaction)

CPU Interconnects (QPI, HyperTransport)

- Designed to minimize latency
- Suited to exchange small packets (fine-grained interaction)

SMBASE integrity

Save State Area

The processor stores its context at SMI entry and restores it at SMI exit

SMBASE

Location of the SMRAM in RAM, stored in the save state area

What if an attacker overwrites the SMBASE?

- Need to exit the SMI and retrigger a SMI
- The new SMBASE is used
- Arbitrary code execution in SMM

Solution

- At boot time: Send the expected value to the monitor
- At runtime: Send the current value at each SMI exit

Performance evaluation

Firmware size

Size of firmware code is limited by the amount of flash (e.g., 8MB or 16MB)

EDK2

- +17 408 bytes in firmware code
- +0.6% increase in size for the compressed firmware

coreboot

- Could not compile the whole firmware with our LLVM toolchain (clang not supported by coreboot)
- AMD Agesa Hudson SMI handlers: +568 bytes
- Intel i82801gx SMI handlers: +3448 bytes

Code integrity at runtime

Multiple options

Page tables

Recent BIOSes can enable write protection for SMM code pages[3]

HP Sure Start Gen3[4]

Detects attempts to modify SMM code

Notifies and takes actions per a predefined policy

[3] https://lists.01.org/pipermail/edk2-devel/2016-November/004185.html

[4] http://www8.hp.com/h20195/v2/GetPDF.aspx/4AA6-9339ENW.pdf

References i

Bazhaniuk, Oleksandr et al. (2015). “A new class of vulnerabilities in SMI handlers”. CanSecWest, Vancouver, Canada.

Bulygin, Yuriy, Oleksandr Bazhaniuk, et al. (2017). “BARing the System: New vulnerabilities in Coreboot & UEFI based systems”. REcon Brussels.

Bulygin, Yuriy and David Samyde (2008). “Chipset based approach to detect virtualization malware”. Black Hat USA.

Davi, Lucas et al. (2015). “Hafix: Hardware-assisted flow integrity extension”. In: *Proceedings of the 52nd Annual Design Automation Conference*. ACM, p. 74.

Intel Corporation (June 2016). *Control-flow Enforcement Technology Preview*.

Kallenberg, Corey et al. (2013). “Defeating Signed BIOS Enforcement”. EkoParty, Buenos Aires.

Lee, Hojoon et al. (2013). “KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object”. In: *Proceedings of the 22th USENIX Security Symposium*, pp. 511–526.

References ii

Liu, Ziyi et al. (2013). "CPU Transparent Protection of OS Kernel and Hypervisor Integrity with Programmable DRAM". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: ACM, pp. 392–403.

Petroni Jr., Nick L. et al. (Aug. 2004). "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor". In: *Proceedings of the 13th USENIX Security Symposium*, pp. 179–194.

Pujos, Bruno (May 2016). *SMM unchecked pointer vulnerability*. URL: `http://esec-lab.sogeti.com/posts/2016/05/30/smm-unchecked-pointer-vulnerability.html`.

Images Credits

URLs provided

Image	Name	Author	License
	Application	Christopher	CC BY 3.0 US
	Chip Settings	Luis Rodrigues	CC BY 3.0 US
	Gear	Jonathan Higley	CC0 1.0 Universal
	Harddrive	Creaticca Creative Agency	CC BY 3.0 US
	Microchip	Creative Stall	CC BY 3.0 US
	Research	Gregor Cresnar	CC BY 3.0 US