FIC 2020

International Cybersecurity Forum

28th, 29th & 30th January 2020

LILLE GRAND PALAIS

#FIC2020

@FIC_eu . Forum FIC

Quelle confiance peut-on placer dans les plateformes matérielles qui exécutent nos applications?

Guillaume Hiet
Clémentine Maurice

WWW.FORUM-FIC.COM

Who are we?

Guillaume Hiet

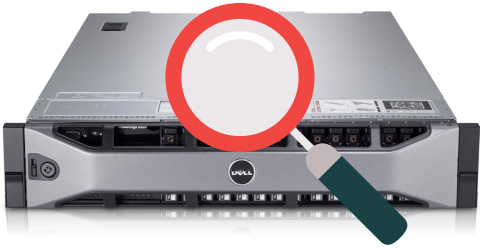
Assistant professor, CentraleSupélec
@GuillaumeHiet

Clémentine Maurice

CNRS researcher
@BloodyTangerine

Introduction

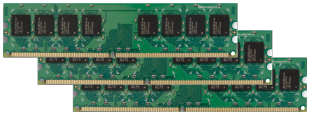
server



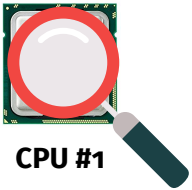
server

CPU #1

CPU #2

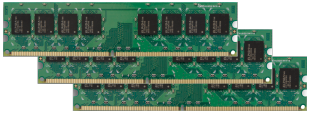


DRAM

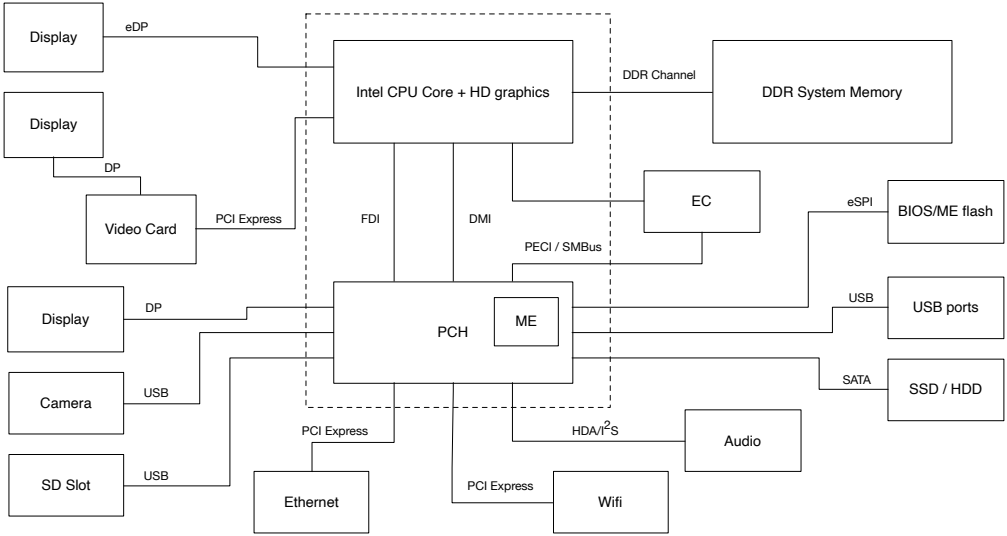


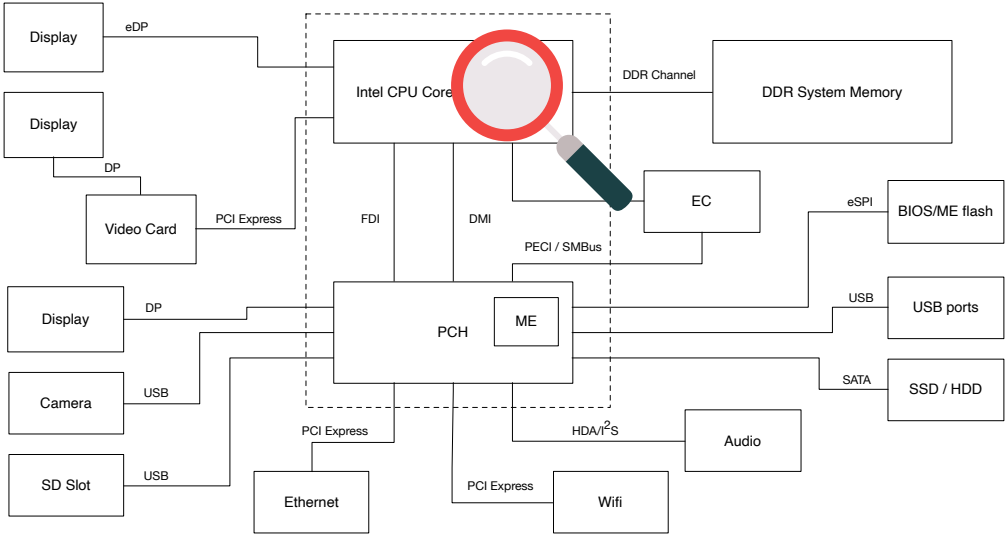
CPU #1

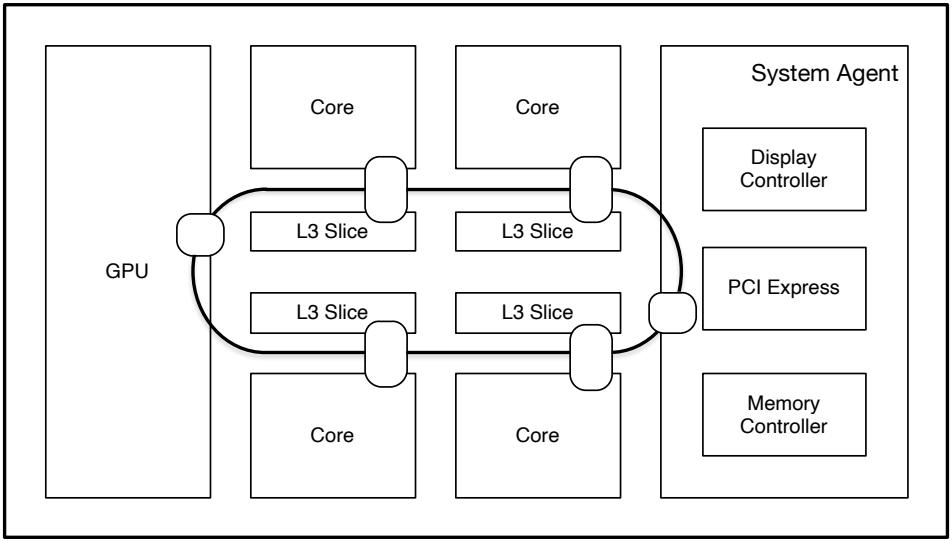
CPU #2

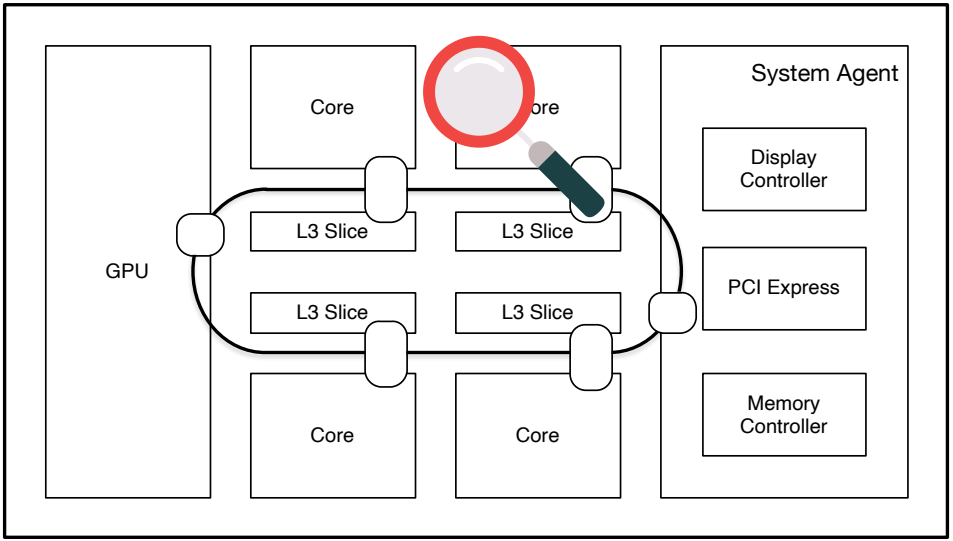


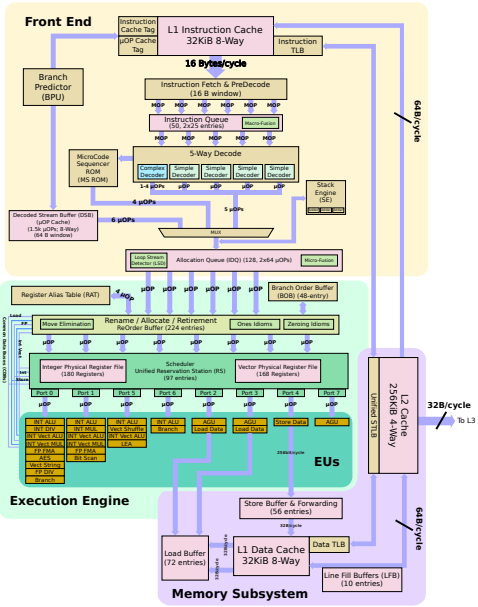
DRAM

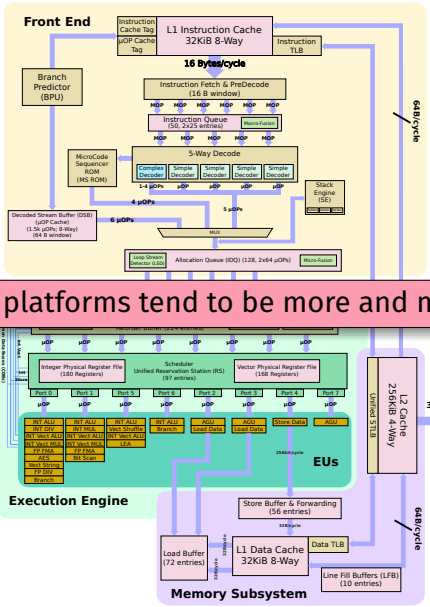

Display
eDP
Display
DP
Video Card
PCI Express
Intel CPU Core + HD graphics
DDR Channel
DDR System Memory
FDI
DMI
EC
eSPI
BIOS/ME flash
PECI / SMBus
Display
DP
PCH
ME
USB
USB ports
Camera
USB
SATA
SSD / HDD
PCI Express
HDA/I2S
Audio
SD Slot
USB
Ethernet
PCI Express
Wifi


Display
eDP
Display
DP
Video Card
PCI Express
Intel CPU Core
DDR Channel
DDR System Memory
FDI
DMI
EC
eSPI
BIOS/ME flash
PECI / SMBus
Display
DP
PCH
ME
USB
USB ports
Camera
USB
SATA
SSD / HDD
PCI Express
HDA/I2S
Audio
SD Slot
USB
Ethernet
PCI Express
Wifi


GPU
Core
Core
Core
Core
L3 Slice
L3 Slice
L3 Slice
L3 Slice
System Agent
Display Controller
PCI Express
Memory Controller


GPU
Core
Core
L3 Slice
L3 Slice
L3 Slice
L3 Slice
Core
Core
System Agent
Display Controller
PCI Express
Memory Controller

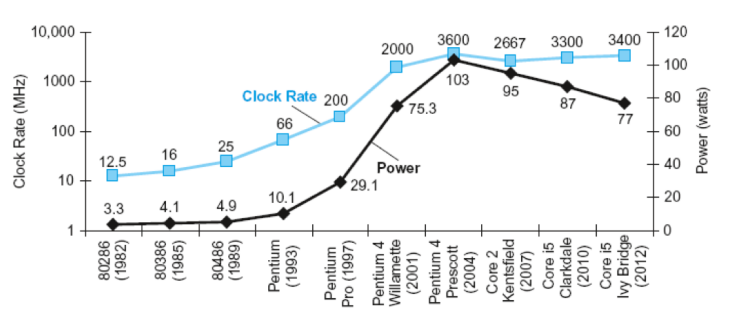

Front End
Instruction Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
Stack Engine (SE)
Decoded Stream Buffer (DSB) (uOP Cache) (1.5k µOPs, 8-Way) (64 B window)
6 µOPs
4 µOPs
5 µOPs
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
64B/cycle
Register Alias Table (RAT)
Branch Order Buffer (BOB) (48-entry)
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
INT Vect Shuffle
INT Vect ALU2
LEA
Branch
AGU
Load Data
Store Data
FP MUL
FP FMA
FP DIV
EUs
Execution Engine
L2 Cache 256KiB 4-Way
Unified STLB
32B/cycle
To L3
Store Buffer & Forwarding (56 entries)
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem


Front End
Instruction Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
uOP Cache Tag
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
1-4 µOPs
Stack Engine (SE)
Decoded Stream Buffer (DSB) (uOP Cache) (1.5k µOPs, 8-Way) (64 B window)
4 µOPs
5 µOPs
6 µOPs
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
64B/cycle
Hardware platforms tend to be more and more complex
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
INT Vect Shuffle
INT Vect ALU2
INT Vect MUL2
INT Vect MUL4
FP FMA
FP FMA2
LEA
Bit Scan
AGU
Load Data
Store Data
Shift
Vect Shuffle
FP DIV
Branch
EUs
Execution Engine
L2 Cache 256KiB 4-Way
Unified STLB
32B/cycle
To L3
Store Buffer & Forwarding (56 entries)
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem

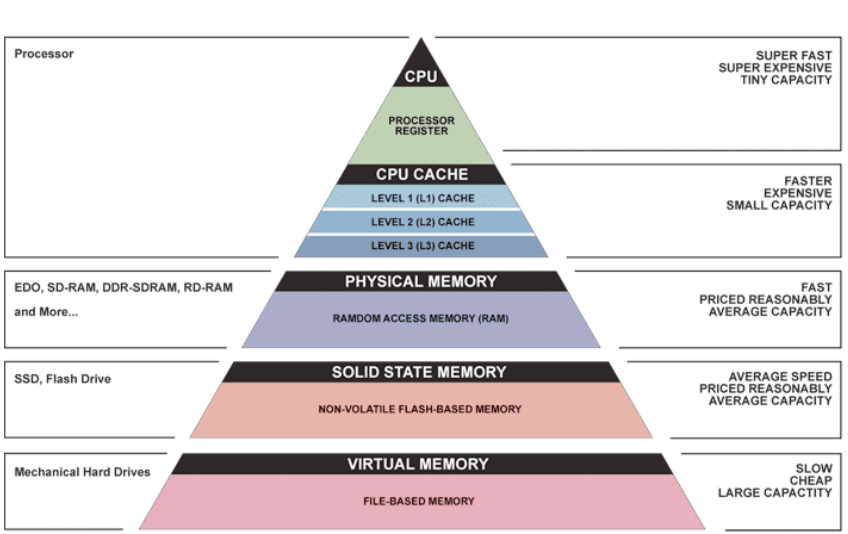
Performance summary

$$CPU\ time = \frac{Instructions}{Program} \times \frac{Clock\ cycles}{Instruction} \times \frac{Seconds}{Clock\ cycle}$$

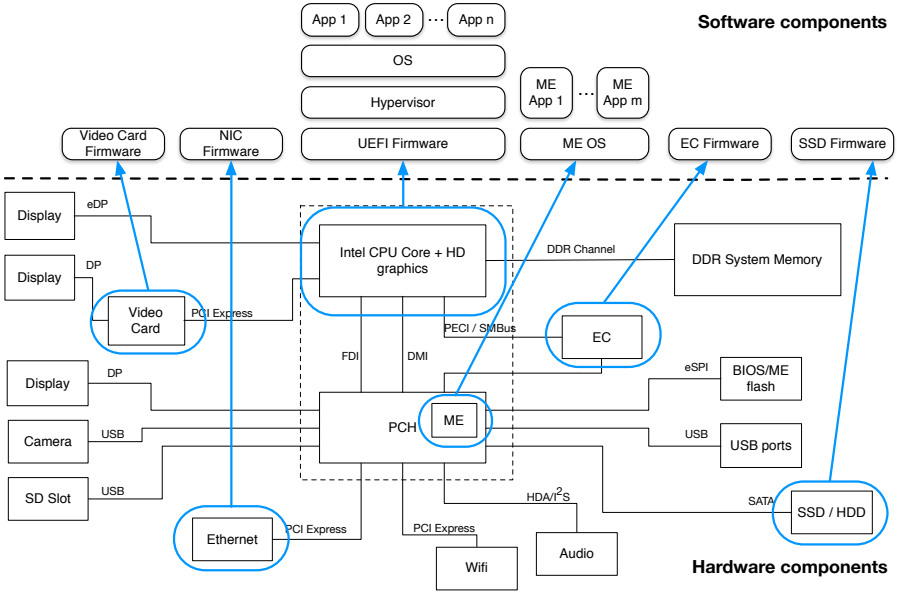
How to increase the performances?

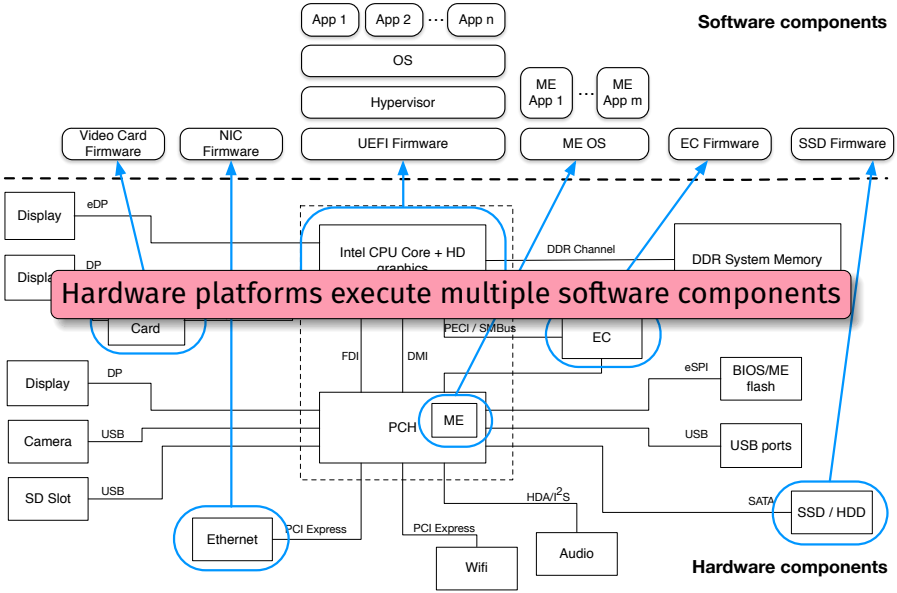


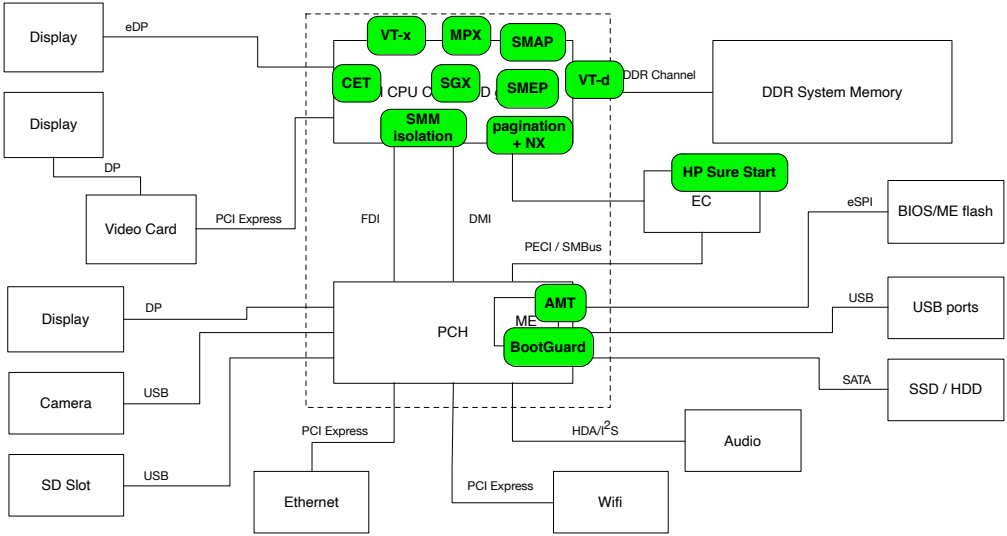
Why do we need caches?

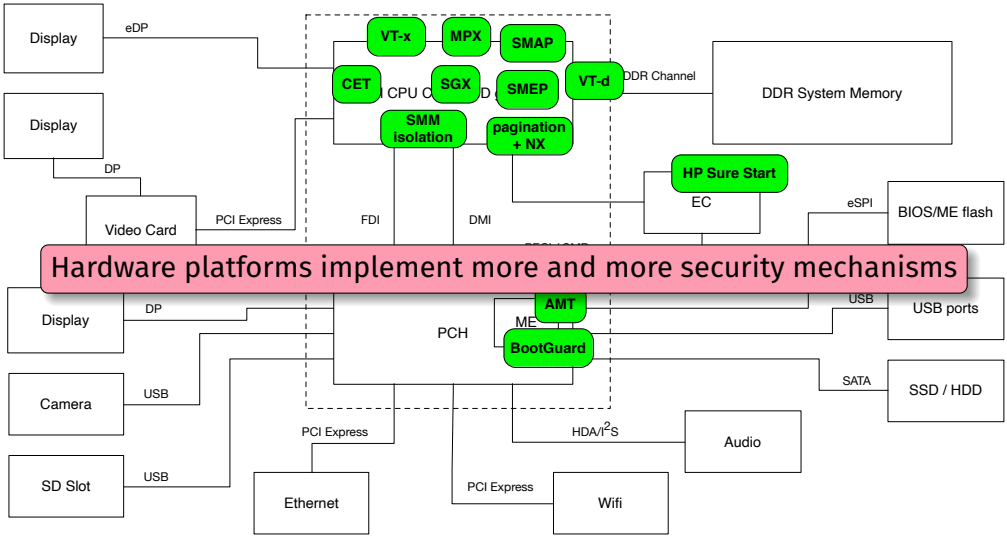


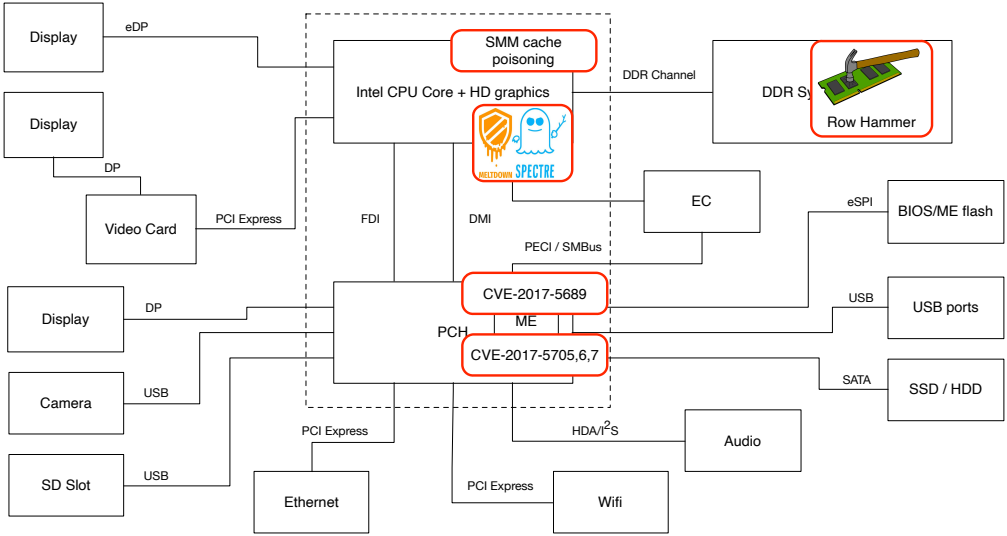
▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

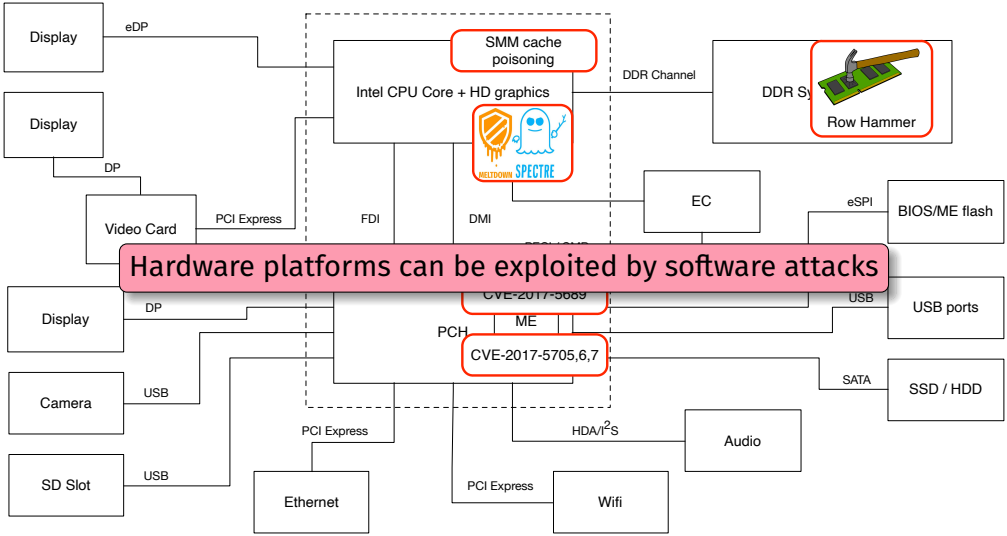

Software components
App 1
App 2
App n
OS
Hypervisor
ME App 1
ME App m
Video Card Firmware
NIC Firmware
UEFI Firmware
ME OS
EC Firmware
SSD Firmware
Display
eDP
Display
DP
Video Card
PCI Express
Intel CPU Core + HD graphics
DDR Channel
DDR System Memory
PECI / SMBus
EC
FDI
DMI
Display
DP
eSPI
BIOS/ME flash
PCH
ME
Camera
USB
USB
USB ports
SD Slot
USB
HDA/I2S
SATA
SSD / HDD
Ethernet
PCI Express
PCI Express
Wifi
Audio
Hardware components


App 1
App 2
App n
Software components
OS
Hypervisor
ME App 1
ME App m
Video Card Firmware
NIC Firmware
UEFI Firmware
ME OS
EC Firmware
SSD Firmware
Display
eDP
DP
Intel CPU Core + HD graphics
DDR Channel
DDR System Memory
Hardware platforms execute multiple software components
Card
PECI / SMBus
EC
FDI
DMI
eSPI
BIOS/ME flash
Display
DP
Camera
USB
PCH
ME
USB
USB ports
SD Slot
USB
HDA/I2S
SATA
SSD / HDD
Ethernet
PCI Express
PCI Express
Wifi
Audio
Hardware components


Display
eDP
Display
DP
Video Card
PCI Express
Display
DP
Camera
USB
SD Slot
USB
VT-x
MPX
SMAP
CET
CPU C
SGX
SMEP
VT-d
DDR Channel
SMM isolation
pagination + NX
DDR System Memory
HP Sure Start
EC
eSPI
BIOS/ME flash
FDI
DMI
PECI / SMBus
PCH
AMT
ME
BootGuard
USB
USB ports
SATA
SSD / HDD
PCI Express
HDA/I2S
Audio
Ethernet
PCI Express
Wifi


Display
eDP
Display
DP
Video Card
PCI Express
VT-x
MPX
SMAP
CET
CPU C
SGX
SMEP
VT-d
DDR Channel
DDR System Memory
SMM isolation
pagination + NX
HP Sure Start
EC
eSPI
BIOS/ME flash
FDI
DMI
Hardware platforms implement more and more security mechanisms
Display
DP
AMT
ME
PCH
BootGuard
USB
USB ports
Camera
USB
SATA
SSD / HDD
PCI Express
HDA/I2S
Audio
SD Slot
USB
Ethernet
PCI Express
Wifi


Display
eDP
Display
DP
Video Card
PCI Express
SMM cache poisoning
Intel CPU Core + HD graphics
DDR Channel
DDR Sy
Row Hammer
MELTDOWN
SPECTRE
FDI
DMI
EC
eSPI
BIOS/ME flash
PECI / SMBus
Display
DP
CVE-2017-5689
ME
PCH
CVE-2017-5705,6,7
USB
USB ports
Camera
USB
SATA
SSD / HDD
PCI Express
HDA/I2S
Audio
SD Slot
USB
Ethernet
PCI Express
Wifi


Display
eDP
Display
DP
Video Card
PCI Express
SMM cache poisoning
Intel CPU Core + HD graphics
DDR Channel
DDR Sy
Row Hammer
MELTDOWN
SPECTRE
EC
eSPI
BIOS/ME flash
FDI
DMI
Hardware platforms can be exploited by software attacks
Display
DP
CVE-2017-5689
USB
USB ports
ME
PCH
CVE-2017-5705,6,7
SATA
SSD / HDD
Camera
USB
PCI Express
HDA/I2S
Audio
SD Slot
USB
Ethernet
PCI Express
Wifi

Outline

1. Introduction

2. Side-channel attacks on SGX

3. Detection of Attacks Against the System Management Mode

4. SILM Thematic Semester

5. Conclusion

Side-channel attacks on SGX

Side channels

- channels that are outside of the functional specification, i.e., that are not supposed to carry useful information

Side channels

- channels that are outside of the functional specification, i.e., that are not supposed to carry useful information
- however they can leak secret information, exploiting implementations

Side channels

- channels that are outside of the functional specification, i.e., that are not supposed to carry useful information
- however they can leak secret information, exploiting implementations
- can be performed by software, e.g., measuring memory accesses

Side channels

- channels that are outside of the functional specification, i.e., that are not supposed to carry useful information
- however they can leak secret information, exploiting implementations
- can be performed by software, e.g., measuring memory accesses

→ attacker monitors which cache lines are accessed, not the content

Sandboxes vs Trusted Execution Environments (TEEs)

- sandboxes assume trusted system and untrusted application

→ protects system from harm

→ e.g., JavaScript in the browser

Sandboxes vs Trusted Execution Environments (TEEs)

- sandboxes assume trusted system and untrusted application

→ protects system from harm

→ e.g., JavaScript in the browser

- TEEs assume an untrusted system and a trusted application

→ isolates the application

→ e.g., cloud, DRM, app that manipulates secrets...

Threat model

- malicious OS
- only the CPU is trusted

$\rightarrow$ TEE memory is encrypted $\rightarrow$ inaccessible to the OS

Threat model

- malicious OS
- only the CPU is trusted

$\rightarrow$ TEE memory is encrypted $\rightarrow$ inaccessible to the OS

Implementations

- Intel: SGX
- ARM and AMD: TrustZone

Threat model

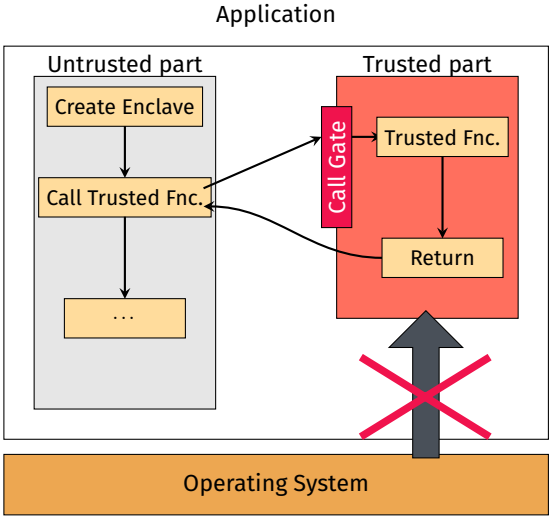
- malicious OS
- only the CPU is trusted

$\rightarrow$ TEE memory is encrypted $\rightarrow$ inaccessible to the OS

Implementations

- Intel: SGX
- ARM and AMD: TrustZone

What is SGX?


Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
...
Call Gate
Trusted Fnc.
Return
Operating System

Let's go back to the threat model

Previous work showed that code inside an enclave
is not protected from side channels from outside

Let's go back to the threat model

Previous work showed that code inside an enclave
is not protected from side channels from outside

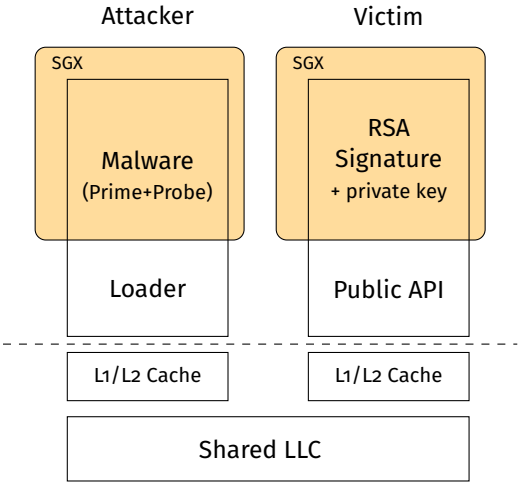
What if the enclave contains malicious code?

SGX limitations

Classical exploits cannot be mounted within SGX:

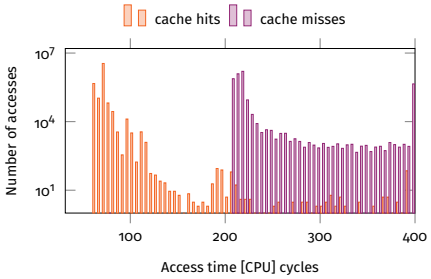
- no syscalls
- no shared memory/libraries
- no interprocess communication
- blocked instructions

Side-channel attacks on SGX (Schwarz et al. 2017)



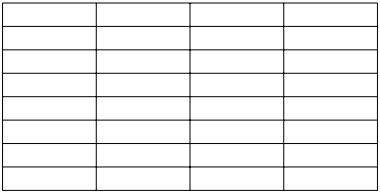
Prime+Probe (1/2)

- exploits timing differences between:
  - cached data (fast)
  - uncached data (slow)
- targets a cache set
- works across CPU cores (shared LLC)

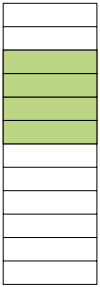


Prime+Probe (2/2)

Victim address space

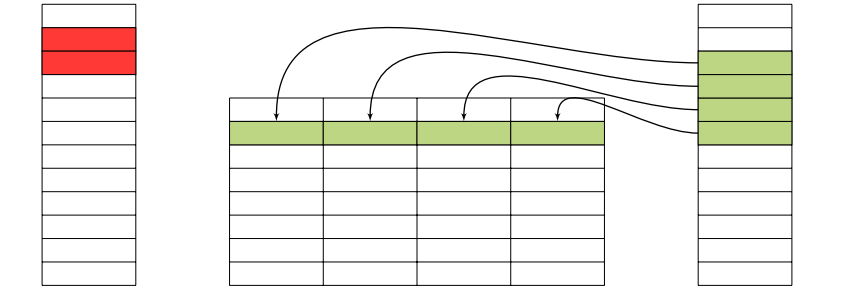


Cache



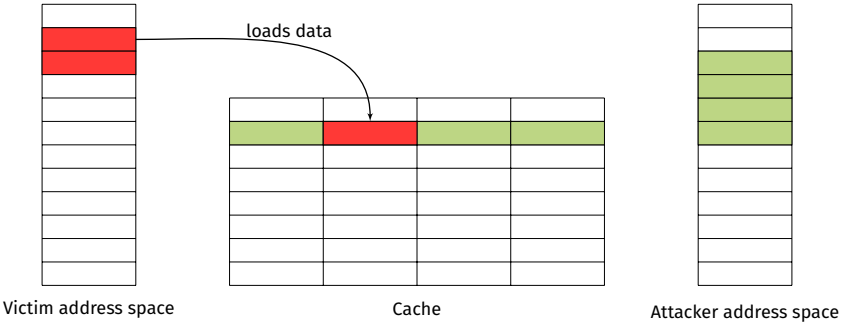
Attacker address space

Prime+Probe (2/2)



Step 1: Attacker primes, i.e., fills, the cache (no shared memory)

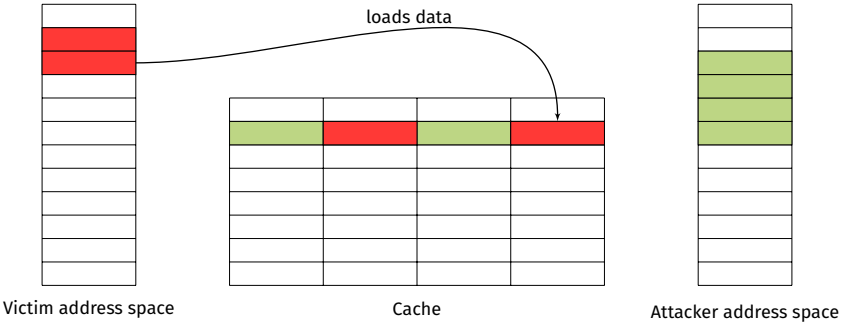
Prime+Probe (2/2)



Step 1: Attacker primes, i.e., fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running

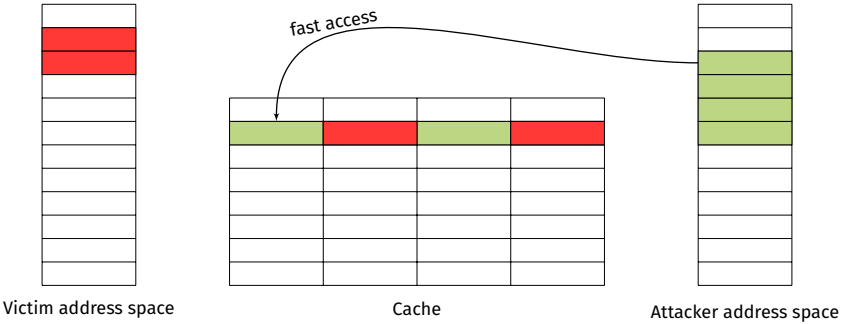
Prime+Probe (2/2)



Step 1: Attacker primes, i.e., fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running

Prime+Probe (2/2)

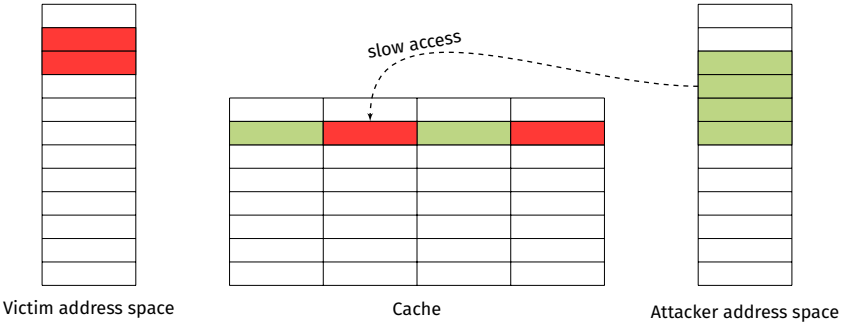


Step 1: Attacker primes, i.e., fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running

Step 3: Attacker probes data to determine if set has been accessed

Prime+Probe (2/2)



Step 1: Attacker primes, i.e., fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running

Step 3: Attacker probes data to determine if set has been accessed

Attack on RSA square-and-multiply exponentiation (1/2)

mbedTLS version 2.3.0 (fixed since)

Algorithm 1: Square-and-multiply exponentiation

Input: base b, exponent e, modulus n

Output: $b^e \mod n$

$X \leftarrow 1$

for $i \leftarrow bitlen(e)$ **downto** 0 **do**

  $X \leftarrow \text{multiply}(X, X)$

  if $e_i = 1$ **then**

    $X \leftarrow \text{multiply}(X, b)$

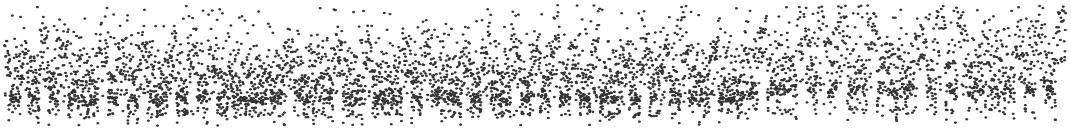
  end

end

return X

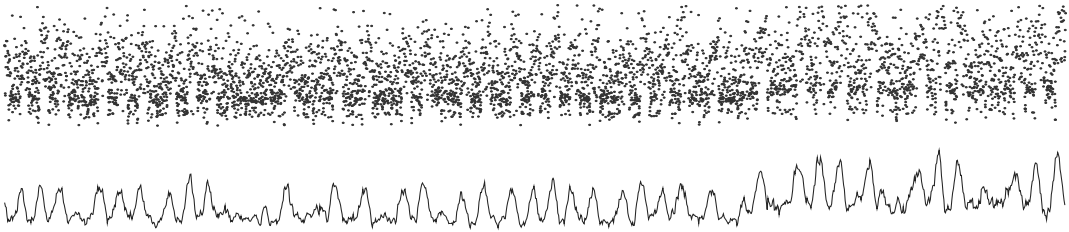
Attack on RSA square-and-multiply exponentiation (2/2)

- raw Prime+Probe trace on the buffer holding the multiplier *b*



Attack on RSA square-and-multiply exponentiation (2/2)

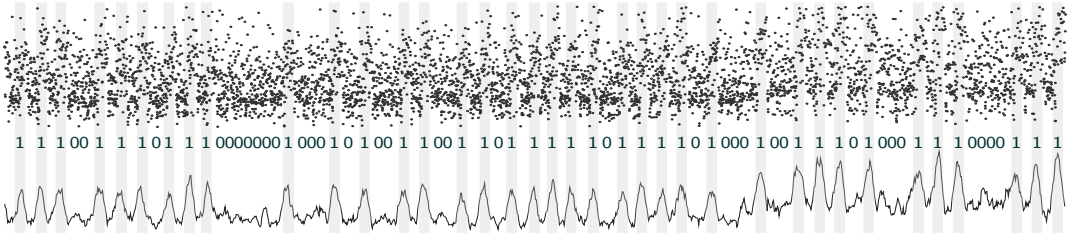
- raw Prime+Probe trace on the buffer holding the multiplier *b*
- processed with a simple moving average



Attack on RSA square-and-multiply exponentiation (2/2)

- raw Prime+Probe trace on the buffer holding the multiplier *b*
- processed with a simple moving average
- allows to clearly recover the bits of the secret exponent

→ 96% of a 4096-bit RSA key from a single trace, full key using 11 traces



Timers

We need to differentiate between events a few nanoseconds apart:

- `rdtsc` instruction does that very well
- unprivileged…

Timers

We need to differentiate between events a few nanoseconds apart:

- `rdtsc` instruction does that very well
- unprivileged…
- … but not permitted inside SGX enclaves

Timers

We need to differentiate between events a few nanoseconds apart:

- `rdtsc` instruction does that very well
- unprivileged…
- … but not permitted inside SGX enclaves

Build your own timer!

- start a thread that continuously increments a global variable
- the global variable is our timestamp

How about detection?

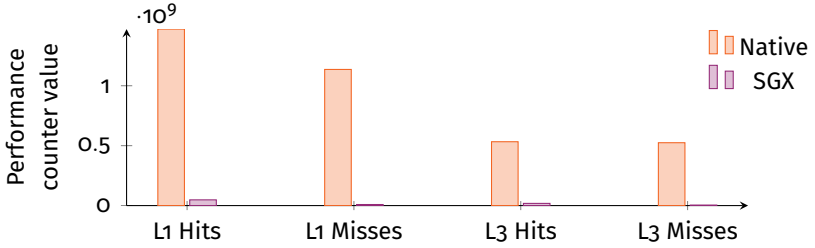
- OS cannot inspect the enclave: traditional methods fail

How about detection?

- OS cannot inspect the enclave: traditional methods fail
- performance counters can detect cache attacks → high rate of cache misses

How about detection?

- OS cannot inspect the enclave: traditional methods fail
- performance counters can detect cache attacks → high rate of cache misses
- however they are disabled inside the enclave



Lessons learned

- hardware optimizations (e.g., cache) can create side channels to exploit weak software implementations

Lessons learned

- hardware optimizations (e.g., cache) can create side channels to exploit weak software implementations
- SGX does not protect from side-channel attacks
  - → enclaves are not a magic protection
  - → Intel: ”this is not in the threat model”

Lessons learned

- hardware optimizations (e.g., cache) can create side channels to exploit weak software implementations
- SGX does not protect from side-channel attacks
  - → enclaves are not a magic protection
  - → Intel: ”this is not in the threat model”
- SGX can actually protect malware
  - → security features can be abused too
  - → rethink enclave protection?

Detection of Attacks Against the System Management Mode

Computers rely on firmware

Where can we find firmware?

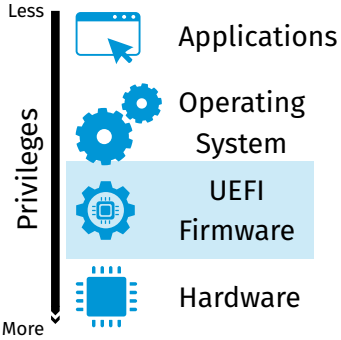
- Hard disks, network cards, etc.
- Motherboards BIOS/UEFI

What is it?

- Low-level software
- Tightly linked to hardware
- Stored into a dedicated flash memory

Boot time vs Runtime

- Early execution and configuration
- Highly privileged runtime software



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 (Kallenberg et al. 2013; Bazhaniuk et al. 2015)

Why compromise a BIOS?

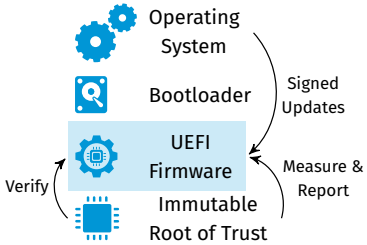
- Malware can be hard to detect (stealth)
- Malware can be persistent (survives even if the HDD/SSD is changed)

What do we want?

- Boot time integrity
- Runtime integrity (some platforms are rarely rebooted)

Boot time

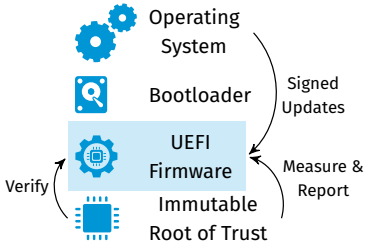
- Signed updates
- Signature verification before executing
- Measurements and reporting to a TPM chip
- Immutable hardware root of trust



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 runtime services available while the OS is running:

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

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

SMM is the highest privileged execution mode for x86 processors.

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)

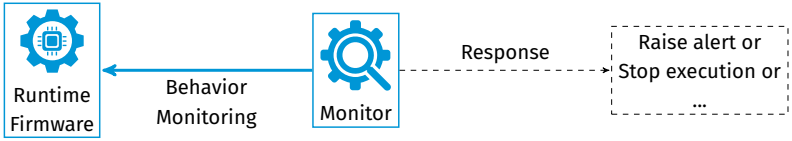
BIOS code is not exempt from vulnerabilities affecting SMM
(Bazhaniuk et al. 2015; Bulygin et al. 2017; Pujos 2016)

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

Our objective

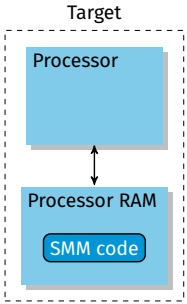
Our goal is to detect attacks that modify the **expected behavior** of the SMM by **monitoring** its behavior **at runtime**.



Such goal raises the following questions:

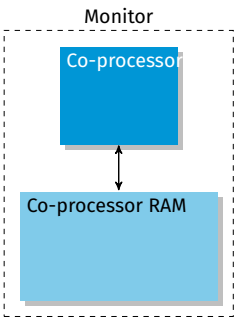
- How to ensure the integrity of the monitor?
- How to define a correct behavior?
- How to monitor?

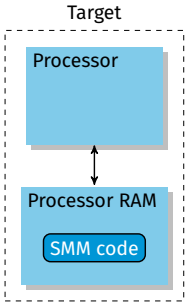
Approach overview (Chevalier et al. 2017)



Approach overview (Chevalier et al. 2017)

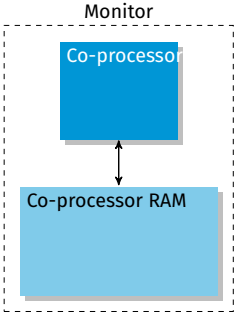
How to ensure the integrity of the monitor?



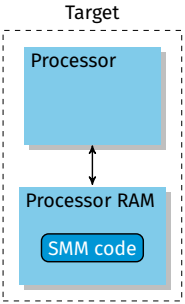


Approach overview (Chevalier et al. 2017)

How to ensure the integrity of the monitor?



Semantic gap?

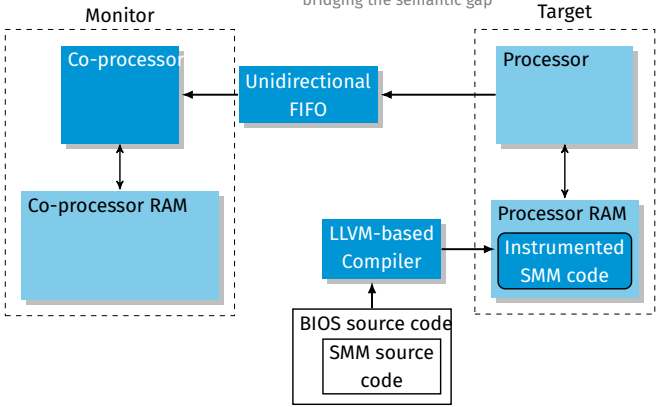


Approach overview (Chevalier et al. 2017)

How to monitor?

bridging the semantic gap

How to ensure the integrity of the monitor?

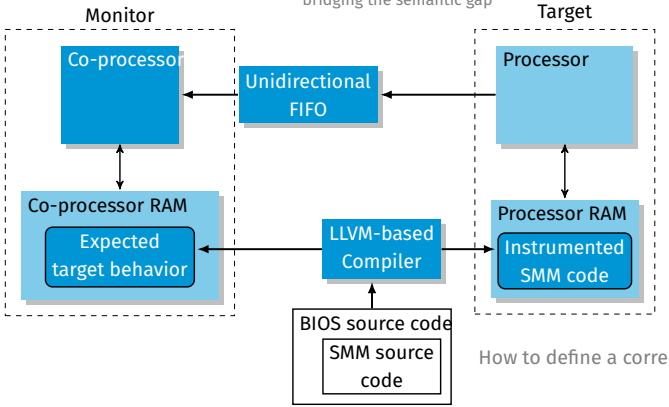


Approach overview (Chevalier et al. 2017)

How to monitor?

bridging the semantic gap

How to ensure the integrity of the monitor?



How to define a correct behavior?

How to define a correct behavior?

Our use case: SMM code

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

Control Flow Graph (CFG)

Define the control flow that the software is expected to follow

→ Control Flow Integrity (CFI)

Invariants on CPU registers

Define rules that registers are expected to satisfy

→ CPU registers integrity

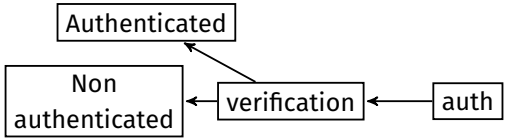
Example

```
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



Control Flow Integrity (CFI): principle

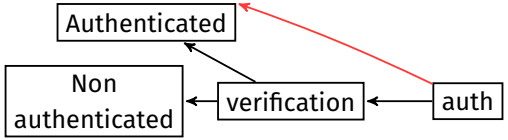
Example

```
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



Control Flow Integrity (CFI): principle

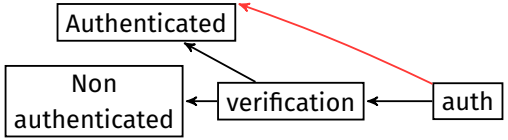
Example

```
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)

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

→ Their value is saved in memory and is not supposed to change at runtime

How to protect such registers?

- Send the expected values at boot time
- Send current values at runtime to detect any discrepancy

Security constraints

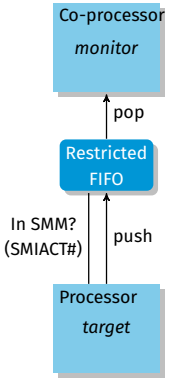
- Message integrity
- Chronological order
- Exclusive access

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

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



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,...)
  → UEFI Variables SMI handlers
- coreboot: perform hardware initialization (used on some Chromebooks)
  → 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

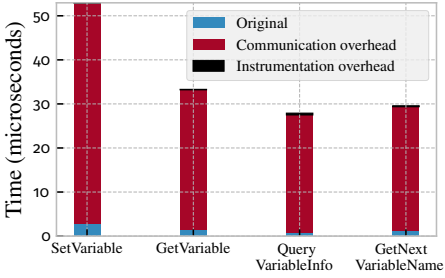
We simulated attacks that exploited vulnerabilities similar to those found in real-world BIOSes

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

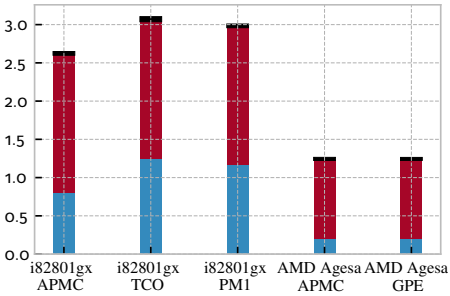
Running time overhead for SMI handlers

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

EDK2



coreboot



Lessons learned

- SMM code can be vulnerable to memory errors
- Attackers exploiting such vulnerabilities can gain full control of the platform

Lessons learned

- SMM code can be vulnerable to memory errors
- Attackers exploiting such vulnerabilities can gain full control of the platform
- We can detect such attacks using a behavior and event-based intrusion detection system implemented on an isolated co-processor
- Acceptable performance ($<$ 150 µs Intel threshold)

Lessons learned

- SMM code can be vulnerable to memory errors
- Attackers exploiting such vulnerabilities can gain full control of the platform
- We can detect such attacks using a behavior and event-based intrusion detection system implemented on an isolated co-processor
- Acceptable performance ($<$ 150 µs Intel threshold)
- What can we do next?
  - React and restore the platform in a sane state
  - Implement other detection approaches to detect other classes of attacks (e.g non-control data attacks)

SILM

Objectives

- Promote the scientific, teaching and industrial transfer activities on a specific subject
- Identify scientific and technological challenges in that field
- Propose a strategic action plan

Organization and funding

- Funded by the DGA
- Managed by Inria, on behalf of all the partners of the PEC research centre (Pôle d'excellence cyber)
- Led by one or several researchers from PEC partners

`https://semestres-cyber.inria.fr/en/`

Actions

- Organization of different events: a summer school, dedicated workshops and a regular seminar
- Animation of a working group and publication of a white-paper
- Invitation of researchers

Three complementary research areas

1. **Analysing the behavior and state of hardware components**
   - Fuzzing
   - Reverse-engineering
   - Trace mechanisms
   - Automated specification analysis
2. **Assessing the security of these hardware components**
   - Side-channel attacks
   - Fault injection
   - Exploiting unspecified behaviors
3. **Detecting or preventing software attacks**
   - Using dedicated hardware components
   - Software countermeasures against hardware vulnerabilities

Organization team

G. Hiet
(CS/Inria, CIDRE)
Project coordinator, Workshops, Working Group & White Book

J.-L. Lanet
(Inria LHS/CIDRE)
Workshops

C. Maurice
(CNRS, EMSEC)
Summer school

F. Tronel
(CS/Inria, CIDRE)
Workshops, Summer school

R. Lashermes
(Inria)
Seminar

- 47 participants from 7 countries: students, young researchers and engineers
- Lectures, labs and CTF, over 4 days and a half
- Organizing committee: **Clémentine Maurice**, Frédéric Tronel
- Slides and videos of the presentations available on the web site

`https://silm-school.inria.fr/`

- November 20-21 2019 in Rennes, France during the European Cyber Week
- First session in common with C&ESAR conference
- 11 invited speakers + 1 paper accepted by C&ESAR conference
- Organizing committee: **Guillaume Hiet**, **Frédéric Tronel**, Jean-Louis Lanet
- Slides and videos of the presentations available on the web site

`https://silm-workshop.inria.fr`

- June 19, 2020 in Genova, Italy
- Workshop collocated with the 5th IEEE European Symposium on Security and Privacy
- Invited speakers + CFP
- Organizing committee: **Guillaume Hiet**, **Frédéric Tronel**, Jean-Louis Lanet
- **Submit articles!**

`https://silm-workshop-2020.inria.fr`

SILM Seminar

- One Friday/month, 2 presentations, at Inria in Rennes, France
- Organizing committee: **Ronan Lashermes**
- Slides and videos of the presentations available on the web site

`https://semestres-cyber.inria.fr/en/silm-seminar/`

Working group

- Objectives: review the state-of-the-art, identify scientific challenges, technical obstacles, industrial transfer perspectives
- Managed by **Guillaume Hiet**
- Deliverables and outcomes: white paper + list of project proposals that should be funded in priority

Conclusion

Vulnerabilities on CPU micro-architectures and firmware are a serious issue

- Be careful of the considered threat model
- Often combined vulnerabilities in hardware (weak isolation) and software (use of vulnerable algorithms)

What can we do?

- Remove hardware optimization (HT, speculative execution, cache, etc.)?
  - Implies a huge impact on performances!
- Patch vulnerable code
  - How can we patch all the vulnerabilities? What about third parties?
- Propose generic solutions combining hardware and software
  - We need hardware support (same story as memory errors)
  - Such mechanism has to be configured by software (OS, application or compiler support)

Bibliography

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

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

Chevalier, Ronny et al. (Dec. 2017). “Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode”. In: *ACSAC 2017 - 33rd Annual Computer Security Applications Conference*. Vol. 2017. Proceedings of the 33rd Annual Computer Security Applications Conference. Orlando, United States: ACM, pp. 399–411. DOI: 10.1145/3134600.3134622. URL: https://hal.inria.fr/hal-01634566.

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

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

Schwarz, Michael et al. (2017). “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: *DIMVA*.

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);
  [...]
}
```

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);
  [...]
}
```

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);
  [...]
}
```

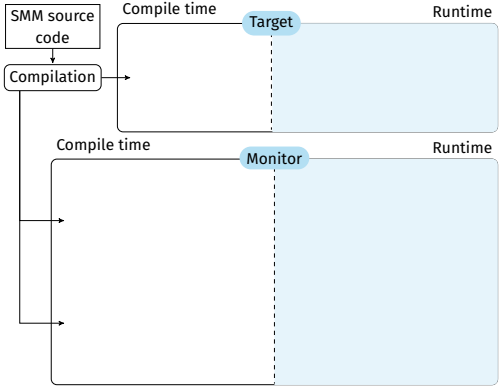
We focus on indirect branches integrity

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);
  [...]
}
```



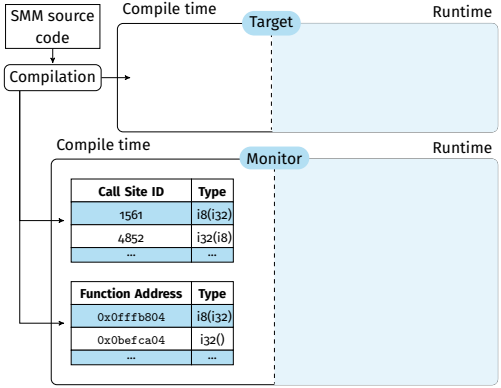
We focus on indirect branches integrity

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 */
  [...]
}
```



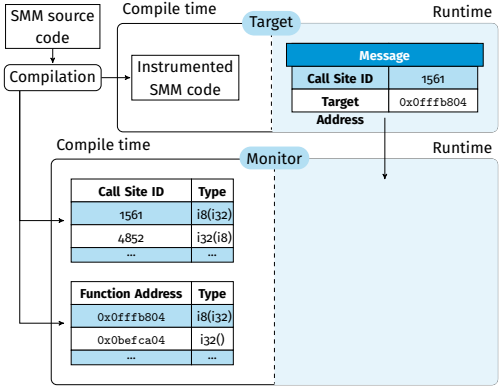
We focus on indirect branches integrity

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 */
  [...]
}
```



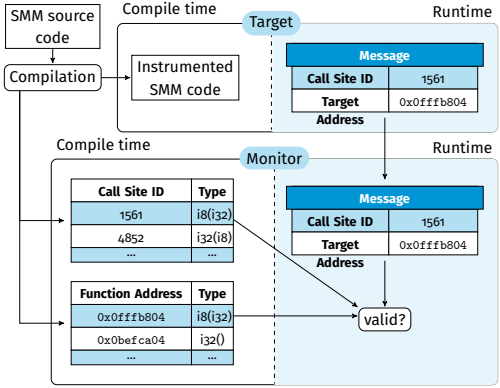
We focus on indirect branches integrity

Type-based verification

Ensures the integrity of indirect calls

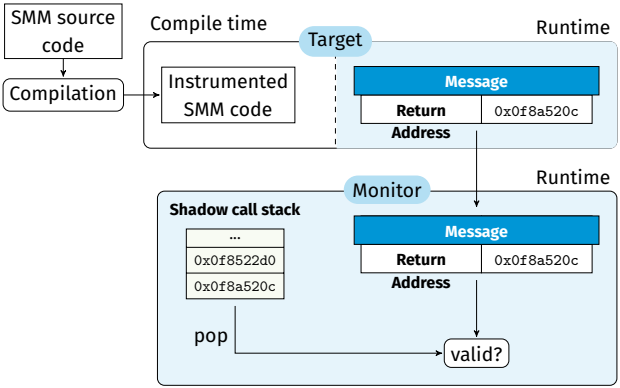
```
typedef struct SomeStruct {
  [...]
  char (*foo)(int);
} SomeStruct;
int bar(SomeStruct *s) {
  char c;
  [...]

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



Shadow call stack

Ensures integrity of the return address on the stack



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.

Credits

- Slide 3, Google
- Slide 8, WikiChip `https://en.wikichip.org/`
- Slide 9, Instructor materials for "Computer Organization and Design, RISC-V Edition", Patterson & Hennessy, 2018
- Slide 10, Rian J. Lang
- Slides 16, 20 icons made by Smashicons from `www.flaticon.com`
- Slides 17, 19, 15, 26 icons made by Freepik from `www.flaticon.com`
- Slide 15, icon made by monkik from `www.flaticon.com`

Images Credits

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