Microarchitectural attacks

ARCHI 2019 - Lorient

Vianney Lapôtre

Université Bretagne Sud / Lab-STICC

May 23, 2019

Introduction to microarchitectural attacks

- Exploiting the behavior of modern computer systems to bypass security primitives or get access to secret data
- Recent architectures have been mainly designed to reach high performances
  - To do so, several optimizations are necessary
    - caches, speculation, prediction, ...
- ***These optimizations open new perspectives for attackers***

Attacker's weapons

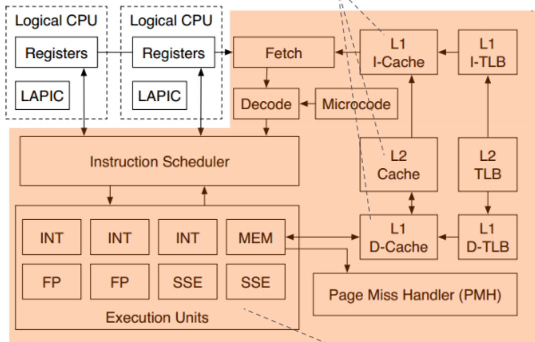
- Attacker
  - infers information from a victim process via hardware usage
  - executes unprivileged software pieces that execute sequences of benign-looking actions
- **These attacks can be hard to detect !**

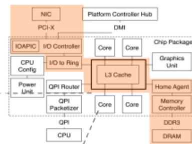
Attacker's weapons

- Execution time
- Shared ressources
- Branch prediction
- Speculation
- Out of order execution
- Memory access pattern and memory access time
- Faults injections (by exploiting microarchitecture features)
- ...

From in-core to cross-core attacks

Cache memory hierarchy

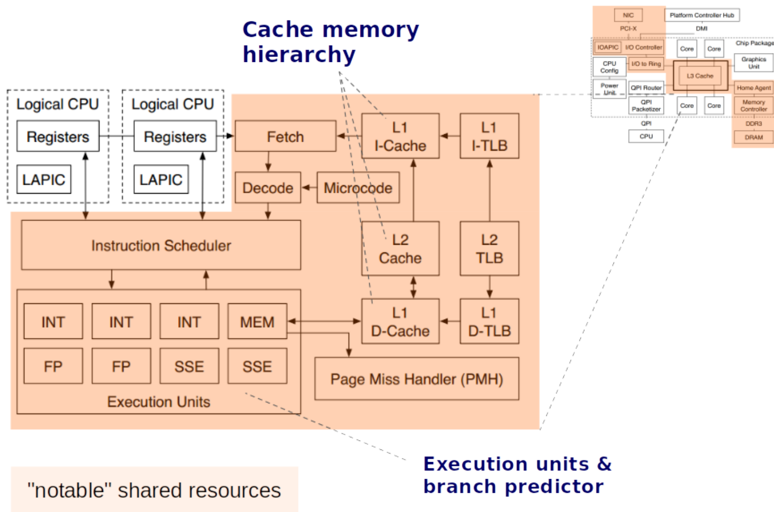




Execution units & branch predictor

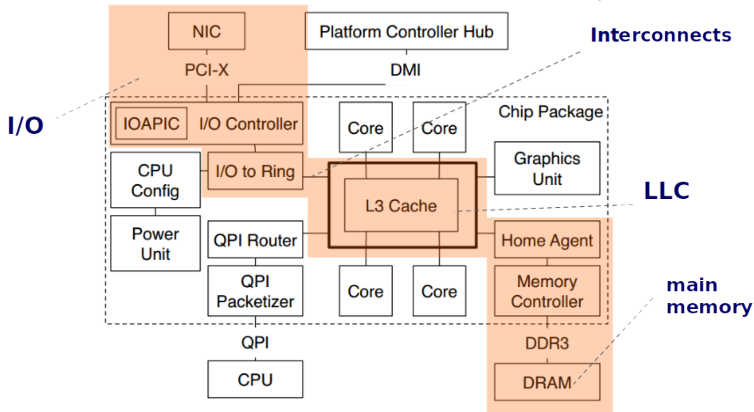
"notable" shared resources

From in-core to cross-core attacks



- We should stop sharing a core !

From in-core to cross-core attacks



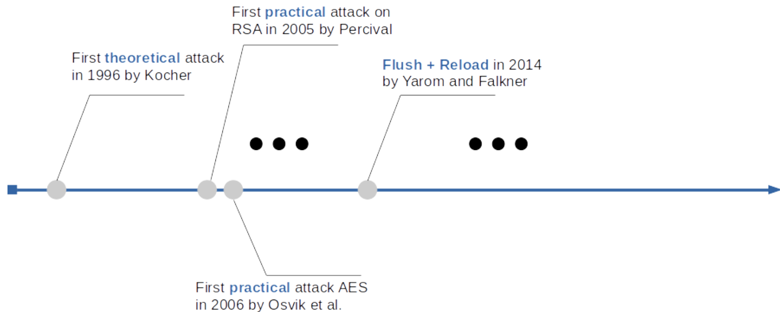
"notable" shared resources

Inclusive caches

- Inclusive property
  - Intel X86_64 architectures
  - Last Level Cache (LLC) is a super set of L1 and L2
    - data evicted from the LLC is also evicted from L1 and L2
- **Thus, a core can evict lines in the private L1 of another core**

Cache-based side channel attacks

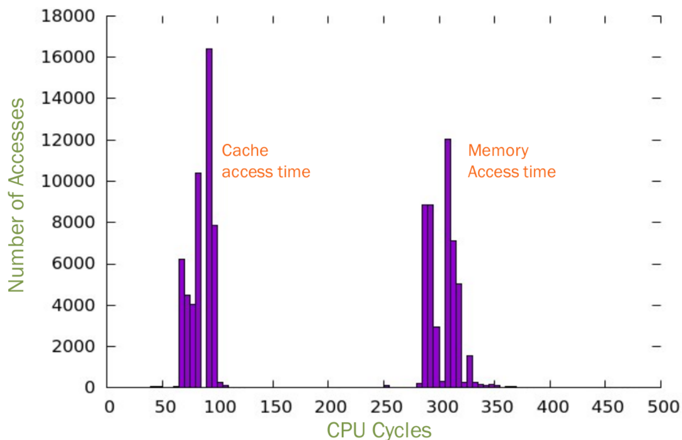
Cache-based SCA important dates



Attacks ideas

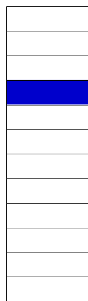
- Caches are small
  - because SRAM is expensive
- Timing variation when accessing a data
  - When data is **cached** => **Cache hit** (i.e. fast access time)
  - When data is **not cached** => **cache miss** (i.e. Slow access time)
- An attacker will exploit this timing variations in order to deduce information regarding a victim process

Timing variation

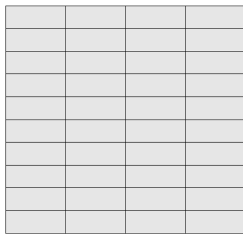


Results measured on Intel i5 for F+R Attack implementation.

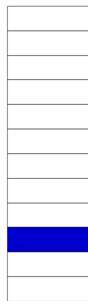
FLUSH + RELOAD



Victim address space

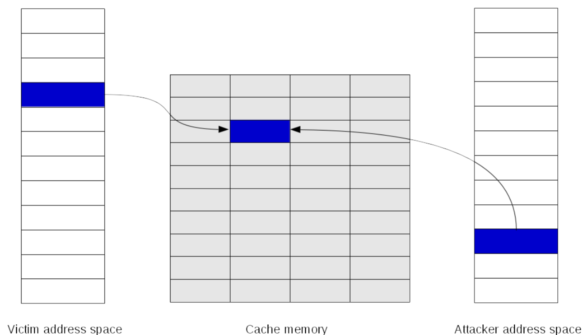


Cache memory



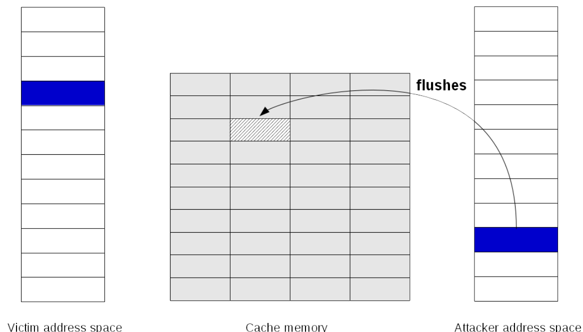
Attacker address space

FLUSH + RELOAD



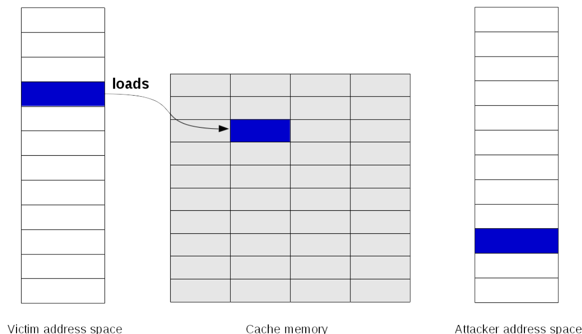
- Step 1: Attacker maps shared library (shared memory, in cache)

FLUSH + RELOAD



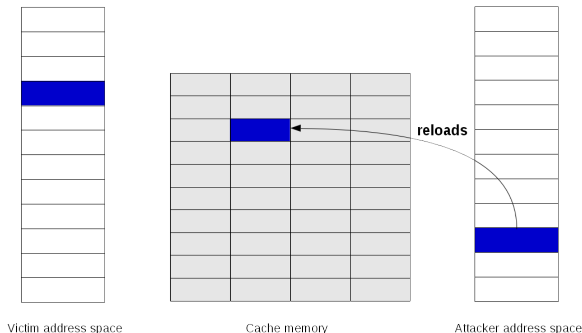
- Step 1: Attacker maps shared library (shared memory, in cache)
- Step 2: Attacker **flushes** the shared cache line

FLUSH + RELOAD



- Step 1: Attacker maps shared library (shared memory, in cache)
- Step 2: Attacker **flushes** the shared cache line
- Step 3: Victim loads the data

FLUSH + RELOAD



- Step 1: Attacker maps shared library (shared memory, in cache)
- Step 2: Attacker **flushes** the shared cache line
- Step 3: Victim loads the data
- Step 4: Attacker **reloads** the data and times this access

FLUSH + RELOAD

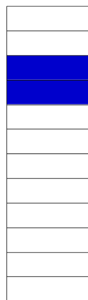
- Pros
  - fine granularity : 1 cache line
- Cons
  - main assumption : shared memory

FLUSH + RELOAD

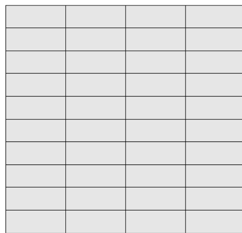
Lab-STICC Université Bretagne Sud ubs

- Pros
  - fine granularity : 1 cache line
- Cons
  - main assumption : shared memory
- Countermeasure
  - don't share sensitive library
  - disable memory deduplication (e.g. virtual machine)

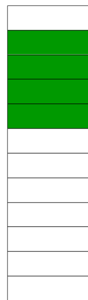
PRIME + PROBE



Victim address space



Cache memory

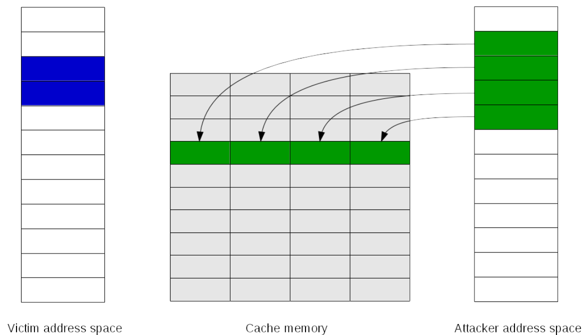


Attacker address space

- In this scenario memory is not shared !
  - collision due to the mapping in the LLC is exploited
  - this mapping has been reverse-engineered [1]

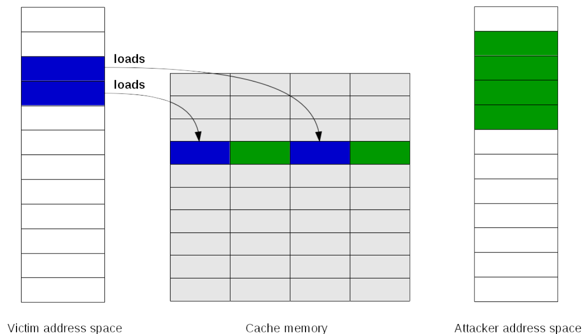
[1] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. “Reverse Engineering Intel Complex Addressing Using Performance Counters”. In: RAID'15. 2015

PRIME + PROBE



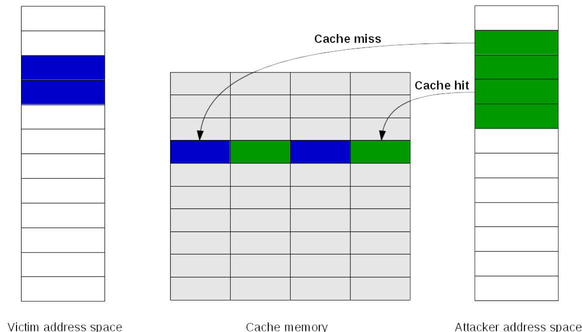
- Step 1: Attacker fills the cache => **PRIME**

PRIME + PROBE



- Step 1: Attacker fills the cache => **PRIME**
- Step 2: Victims runs evicting some cache lines

PRIME + PROBE



- Step 1: Attacker fills the cache => **PRIME**
- Step 2: Victims runs evicting some cache lines
- Step 3: Attacker access his data again => **PROBE**

PRIME + PROBE

- cross-VM side channel attacks on crypto algorithms
  - El Gamal (sliding window): full key recovery in 12 min. [2]
- tracking user behavior in the browser, in JavaScript [3]
- covert channels between virtual machines in the cloud [4]

[2] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: S&P'15. 2015.

[3] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications". In: CCS'15. 2015.

[4] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud". In: NDSS'17. 2017.

PRIME + PROBE

Lab-STICC Université Bretagne Sud

- Countermeasure
  - strong isolation
    - stop sharing CPUs ?
    - crypto in Hardware
  - randomization
- Detection mechanisms
  - hardware performance counters can help

Meltdown

- https://meltdownattack.com/ [5]

[5] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. "Meltdown". 2018

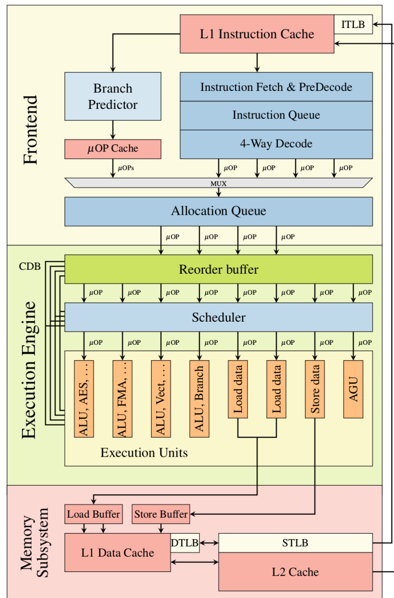
Background: Out-of-order execution

- Optimization technique maximizing the utilization of all execution units of a CPU
  - without OoO : instructions are processed in the sequential order
  - with OoO : the CPU executes instructions as soon as all required resources are available
- In practice, OoO execution is coupled with speculative execution
  - processor's OoO logic processes instructions before the CPU is certain whether the instruction will be needed and committed

Background: Speculative execution & Branch Prediction

- Speculative execution can refers (but not only) to an instruction sequence following a branch
- Branch prediction units are used to obtain an educated guess of which path (TAKEN / NOT TAKEN) will have to be executed
- Independent instructions that lie on the predicted path can be executed in advance
  - If the prediction was correct, their results can be directly used
  - If the prediction was incorrect, their results are cleared

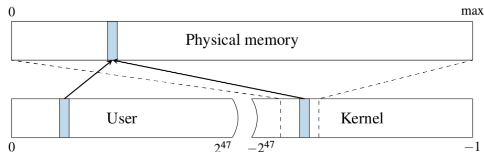
Background : Intel's Skylake microarchitecture



Background : Address spaces

- CPUs supports virtual address spaces in order to isolate processes from each other
  - A virtual address space is divided into a set of pages that can be mapped to physical memory throuh a multi-level page translation table
  - These tables contains the virtual to physical mapping and the protection properties (read/write/execute/user-access)
  - On each context switch, the OS updates a special register with the next process' translation table

Background : Address spaces



- The virtual space is split into a user part and kernel part which can be accessed when CPU is running in privileged mode only
- The entire physical memory is typically mapped in the kernel
  - On Linux and OS X => via a *direct-physical* map

Starting to play. . .

Lab-STICC

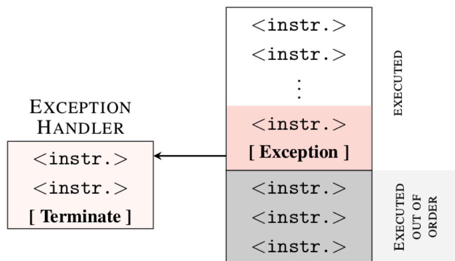
Université Bretagne Sud

```
...; // some code lines
raise_exception();
//call to access() is never reached
access(probe_array[data*4096]);
```

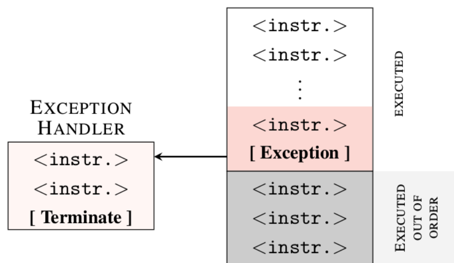
Starting to play…

```
...; // some code lines
raise_exception();
//call to access() is never reached
access(probe_array[data*4096]);
```

- But…



Starting to play. . .



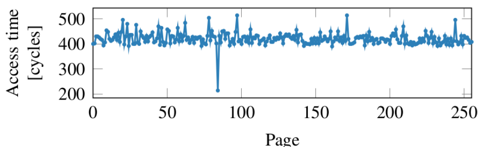
- Of course, there is no visible effect en registers or memory
- But, microarchitectural side effects exist
  - due to OoO execution, **referenced memory is stored in the cache**
  - it opens the door for cache-based SCA

Starting to play... using F+R

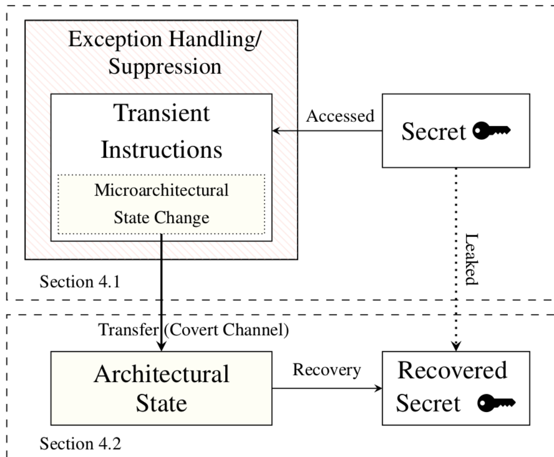
Lab-STICC Université Bretagne Sud ubs

```
raise_exception();
//call to access() is never reached
access(probe_array[data*4096]);
```

- Considering *probe_array* is of type *char* and memory pages size is 4kB
  - F+R approach can be used to determine which page has been retrieved from the memory
    - **so, we deduce the value of 'data'**



Meltdown : Overview



Meltdown : Executing Transient instructions

- Trying to access user-inaccessible pages triggers an exception which generally terminates the application
  - the attacker has to manage this exception while targeting a secret stored at a inaccessible address
- Two approaches
  - exception handling and exception suppression

Meltdown : Executing Transient instructions

- **Exception handling**: catch the exception after executiong the transient instruction sequence
  - by forking the attacking process before accessing the invalid memory location
    - the child process only accesses the invalid memory location while the parent process recovers the secret by observing the microarchitectural side effects
  - by installing a signal handler that is executed when the targeted exception is triggered
    - this allows the execution of the transient instruction sequence and prevent the application from crashing

Meltdown : Executing Transient instructions

- **Exception suppression**: prevent the exception from occurring
  - by exploiting transactional memory (TSX) which groups memory accesses into one seemingly atomic operation
    - when an exception occurs within the transaction, the architectural state is reseted and the program execution continues without disruption
  - by exploiting branch prediction to speculatively execute instructions that would not be executed in the correct execution path
    - this approach requires a training of the branch predictor

Meltdown : Attack description

- **Step 1**: The content of an attacker-chosen memory location, which is inaccessible to the attacker, is loaded into a register
- **Step 2**: A transient instruction accesses a cache line based on the secret content of the register
- **Step 3**: The attacker uses Flush+Reload to determine the accessed cache line and hence the secret stored at the chosen memory location.

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Meltdown : Attack description

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc ; * 4k (page size)
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- **Step 1: Reading the secret**
  - Load the targeted byte value into AL (least significant bit of the RAX register)
  - transient instruction sequence (line 5-7) are already decoded and allocated
    - As soon as the targeted byte is observed on the data bus, these instruction begin their execution
  - MOV instruction of line 4 leads to an exception => race condition between raising this exception and step 2

Meltdown : Attack description

```
1  ; rcx = kernel address
2  ; rbx = probe array
3  retry:
4  mov al, byte [rcx]
5  shl rax, 0xc   ; * 4k (page size)
6  jz retry
7  mov rbx, qword [rbx + rax]
```

- Step 2 : Transmitting the secret
  - If the transient sequence instruction is executed before the MOV is retired, it can be used to transmit the secret
    - ensuring that the probe array is not cached
    - using an indirect memory access dependent of the secret

Meltdown : Attack description

```
1  ; rcx = kernel address
2  ; rbx = probe array
3  retry:
4  mov al, byte [rcx]
5  shl rax, 0xc ; * 4k (page size)
6  jz retry
7  mov rbx, qword [rbx + rax]
```

- Step 3 : Receiving the secret
  - using a microarchitectural SCA => F+R in meltdown
  - When the step 2 is a success, a unique cache line of the probe array is cached
    - F+R is used to determined the position (i.e the secret !) of this cache line

- These 3 steps are repeated in order to dump the entire physical memory
- Since accesing the kernel memory raises an exception it is necessary to use an *Exception handling* or an *Exception suppression* method

Meltdown : Mitigation

- Hardware approach
  - disable out-of-order execution
  - serializing the permission check and the register fetch
  - **More realistic** : introduce a hard split of user space and kernel space
- Software approach
  - Kaiser [6]: kernel modifications to not have the kernel mapped in the user space
    - originally developed to prevent SCA breaking KASLR
    - called Kenel Page-Table isolation (PKTI) in Linux kernel
    - still has some limitations (several privileged memory location have to be mapped in user space)

[6] Gruss, D., Lipp, M., Schwarz , M., Fellner, R., Maurice, C., AND Mangard, S. "KASLR is Dead: Long Live KASLR". In International Symposium on Engineering Secure Software and Systems (2017), Springer, pp. 161–176.

Conclusion

Lab-STICC Université Bretagne Sud ubs

- Attacks on microarchitectures is a *hot* topic
- Futur processors will have to be designed taking into account these leaks
- Software designers implementing security functions have to learn about the underlying microarchitecture in order to provide pieces of code that take into account the hardware behavior
- For strong security requirements, relying on dedicated hardware could enhance the global security of the system