Introduction to cache side-channel attacks

Clémentine Maurice, CNRS, IRISA
October 9, 2018—Séminaire du DIT, ENS Rennes
Who am I

- Clémentine Maurice
- Chargée de Recherche CNRS
- IRISA lab, EMSEC group
- clementine.maurice@irisa.fr
- @BloodyTangerine
Everyday hardware: servers, workstations, laptops, smartphones...
Side channels

- **safe software** infrastructure → no bugs, e.g., buffer overflows
Side channels

- safe software infrastructure → no bugs, e.g., buffer overflows
- does not mean safe execution
Side channels

- **safe software** infrastructure → no bugs, e.g., buffer overflows
- does not mean safe execution
- **secrets leak** because of the **hardware** it runs on
- no “bug” in the sense of a mistake → lots of performance optimizations
Side channels

• **safe software** infrastructure → no bugs, e.g., buffer overflows
• does not mean safe execution
• **secrets leak** because of the **hardware** it runs on
• no “bug” in the sense of a mistake → lots of performance optimizations

→ crypto and sensitive info., e.g., keystrokes and mouse movements
Sources of leakage

• via power consumption, electromagnetic leaks
Sources of leakage

• via power consumption, electromagnetic leaks
  • targeted attacks, physical access
  • shared hardware and microarchitecture
    • “remote” attacks, no physical access to the device
Sources of leakage

- via power consumption, electromagnetic leaks
  → targeted attacks, physical access
Sources of leakage

• via power consumption, electromagnetic leaks
  → targeted attacks, physical access

• via shared hardware and microarchitecture
Sources of leakage

- via power consumption, electromagnetic leaks
  → targeted attacks, physical access

- via shared hardware and microarchitecture
  → “remote” attacks, no physical access to the device
Shared hardware

- Memory
  - DRAM
  - Memory bus

- CPU
  - Branch prediction unit
  - Arithmetic logic unit
  - Data and instruction cache

- Data and instruction cache
From small optimizations to side channels

- 2008: Nehalem
- 2012: Sandy Bridge
- 2013: Ivy Bridge
- 2014: Haswell
- 2015: Broadwell
- 2016: Skylake
- 2017: Kaby Lake
- 2018: Coffee Lake

- new microarchitectures yearly
- very small optimizations: caches, branch prediction...
- ... leading to side channels
- no documentation on this intellectual property
From small optimizations to side channels

- New microarchitectures yearly
- Performance improvement ≈ 5%

Year | Model
--- | ---
2008 | Nehalem
2012 | Sandy Bridge
2013 | Ivy Bridge
2014 | Haswell
2015 | Broadwell
2016 | Skylake
2017 | Kaby Lake
2018 | Coffee Lake
From small optimizations to side channels

- New microarchitectures yearly
- Performance improvement ≈ 5%
- Very small optimizations: caches, branch prediction...

Years:
- 2008: Nehalem
- 2012: Sandy Bridge
- 2013: Ivy Bridge
- 2014: Haswell
- 2015: Broadwell
- 2016: Skylake
- 2017: Kaby Lake
- 2018: Coffee Lake

No documentation on this intellectual property.
From small optimizations to side channels

- New microarchitectures yearly
- Performance improvement $\approx 5\%$
- Very small optimizations: caches, branch prediction...
- ... leading to side channels
From small optimizations to side channels

- 2008: Nehalem
- 2012: Sandy Bridge
- 2013: Ivy Bridge
- 2014: Haswell
- 2015: Broadwell
- 2016: Skylake
- 2017: Kaby Lake
- 2018: Coffee Lake

- new microarchitectures yearly
- performance improvement $\approx 5\%$
- very small optimizations: caches, branch prediction...
- ... leading to side channels
- no documentation on this intellectual property
Today’s CPU complexity

- “Intel x86 documentation has more pages than the 6502 has transistors”

Today’s CPU complexity

• “Intel x86 documentation has more pages than the 6502 has transistors”
• 6502: 8-bit microprocessor, used in the Apple II, Commodore 64, Atari 800...

Today’s CPU complexity

- “Intel x86 documentation has more pages than the 6502 has transistors”
- 6502: 8-bit microprocessor, used in the Apple II, Commodore 64, Atari 800...
  - year: 1975 → 3510 transistors

---

Today’s CPU complexity

- “Intel x86 documentation has more pages than the 6502 has transistors”
- 6502: 8-bit microprocessor, used in the Apple II, Commodore 64, Atari 800...
  - year: 1975 → 3510 transistors
- 22-core Intel Xeon Broadwell-E5

“Intel x86 documentation has more pages than the 6502 has transistors”

- 6502: 8-bit microprocessor, used in the Apple II, Commodore 64, Atari 800...
  - year: 1975 → 3510 transistors
- 22-core Intel Xeon Broadwell-E5
  - year: 2016 → 7.2 billion transistors

---

Today’s CPU complexity

- “Intel x86 documentation has more pages than the 6502 has transistors”
- 6502: 8-bit microprocessor, used in the Apple II, Commodore 64, Atari 800...
  - year: 1975 → 3510 transistors
- 22-core Intel Xeon Broadwell-E5
  - year: 2016 → 7.2 billion transistors
- Intel Software Developer’s Manuals (may. 2018): 4844 pages

Today’s CPU complexity

• “Intel x86 documentation has more pages than the 6502 has transistors”
• 6502: 8-bit microprocessor, used in the Apple II, Commodore 64, Atari 800…
  • year: 1975 → 3510 transistors
• 22-core Intel Xeon Broadwell-E5
  • year: 2016 → 7.2 billion transistors
• Intel Software Developer’s Manuals (may. 2018): 4844 pages
• (there are actually more manuals than just the SDM)

Outline

• Background on cache attacks
• Reverse-engineering
• Practical attacks
• Countermeasures and open challenges
• Conclusion
Background on cache attacks
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Legacy Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>88 lr</td>
<td>MOV r/m8,r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r8 to r/m8.</td>
</tr>
<tr>
<td>REX + 88 lr</td>
<td>MOV r/m8**,<strong>r8</strong>*</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r8 to r/m8.</td>
</tr>
<tr>
<td>89 lr</td>
<td>MOV r/m16,r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r16 to r/m16.</td>
</tr>
<tr>
<td>89 lr</td>
<td>MOV r/m32,r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r32 to r/m32.</td>
</tr>
<tr>
<td>REX.W + 89 lr</td>
<td>MOV r/m64,r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r64 to r/m64.</td>
</tr>
<tr>
<td>8A lr</td>
<td>MOV r8,r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m8 to r8.</td>
</tr>
<tr>
<td>REX + 8A lr</td>
<td>MOV r8***,<strong>r/m8</strong>*</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r/m8 to r8.</td>
</tr>
<tr>
<td>8B lr</td>
<td>MOV r16,r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m16 to r16.</td>
</tr>
<tr>
<td>8B lr</td>
<td>MOV r32,r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m32 to r32.</td>
</tr>
<tr>
<td>REX.W + 8B lr</td>
<td>MOV r/m64,r64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r/m64 to r64.</td>
</tr>
<tr>
<td>8C lr</td>
<td>MOV r/m16,Sreg**</td>
<td>MR</td>
<td>Valid</td>
<td>Move segment register to r/m16.</td>
<td></td>
</tr>
<tr>
<td>REX.W + 8C lr</td>
<td>MOV r/m64,Sreg**</td>
<td>MR</td>
<td>Valid</td>
<td>Move zero extended 16-bit segment register to r/m64.</td>
<td></td>
</tr>
<tr>
<td>8E lr</td>
<td>MOV Sreg,r/m16**</td>
<td>RM</td>
<td>Valid</td>
<td>Move r/m16 to segment register.</td>
<td></td>
</tr>
<tr>
<td>REX.W + 8E lr</td>
<td>MOV Sreg,r/m64**</td>
<td>RM</td>
<td>Valid</td>
<td>Move lower 16 bits of r/m64 to segment register.</td>
<td></td>
</tr>
<tr>
<td>A0</td>
<td>MOV AL,moffs8*</td>
<td>FD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move byte at (seg:offset) to AL.</td>
</tr>
<tr>
<td>REX.W + A0</td>
<td>MOV AL,moffs8*</td>
<td>FD</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move byte at (offset) to AL.</td>
</tr>
<tr>
<td>A1</td>
<td>MOV AX,moffs16*</td>
<td>FD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move word at (seg:offset) to AX.</td>
</tr>
<tr>
<td>A1</td>
<td>MOV EAX,moffs32*</td>
<td>FD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move doubleword at (seg:offset) to EAX.</td>
</tr>
<tr>
<td>REX.W + A1</td>
<td>MOV RAX,moffs64*</td>
<td>FD</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move quadword at (offset) to RAX.</td>
</tr>
</tbody>
</table>
## 64-Bit Mode Exceptions

### #GP(0)
- If the memory address is in a non-canonical form.
- If an attempt is made to load SS register with NULL segment selector when CPL = 3.
- If an attempt is made to load SS register with NULL segment selector when CPL < 3 and CPL ≠ RPL.

### #GP(selector)
- If segment selector index is outside descriptor table limits.
- If the memory access to the descriptor table is non-canonical.
- If the SS register is being loaded and the segment selector’s RPL and the segment descriptor’s DPL are not equal to the CPL.
- If the SS register is being loaded and the segment pointed to is a nonwritable data segment.
- If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
- If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.

### #SS(0)
- If the stack address is in a non-canonical form.

### #SS(selector)
- If the SS register is being loaded and the segment pointed to is marked not present.

### #PF(fault-code)
- If a page fault occurs.

### #AC(0)
- If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### #UD
- If attempt is made to load the CS register.
- If the LOCK prefix is used.
What could go wrong?

- lots of exceptions for mov
• lots of exceptions for `mov`
• but accessing data loads it to the cache
mov—What could go wrong?

- lots of exceptions for `mov`
- but accessing data loads it to the cache
  → side effects on computations!
Memory hierarchy

- data can reside in
Memory hierarchy

- data can reside in
  - CPU registers
Memory hierarchy

- data can reside in
  - CPU registers
  - different levels of the CPU cache
Memory hierarchy

- data can reside in
  - CPU registers
  - different levels of the CPU cache
  - main memory
Memory hierarchy

- data can reside in
  - CPU registers
  - different levels of the CPU cache
  - main memory
  - disk storage
Caches on Intel CPUs

- L1 and L2 are private
Caches on Intel CPUs

- L1 and L2 are private
- Last-level cache

*Diagram showing core 0, 1, 2, and 3 with L1 and L2 caches shared across cores, and ring bus connecting LLC slices 0, 1, 2, and 3.*
Caches on Intel CPUs

- L1 and L2 are private
- last-level cache
  - divided in slices
Caches on Intel CPUs

- L1 and L2 are private
- Last-level cache
  - Divided in slices
  - Shared across cores
Caches on Intel CPUs

- L1 and L2 are private
- last-level cache
  - divided in slices
  - shared across cores
  - inclusive
Set-associative caches

<table>
<thead>
<tr>
<th>Address</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cache
Set-associative caches

Data loaded in a specific set depending on its address
Set-associative caches

Data loaded in a specific **set** depending on its address

Several **ways** per set
Set-associative caches

Data loaded in a specific set depending on its address

Several ways per set

Cache line loaded in a specific way depending on the replacement policy
Timing differences

Access time [CPU cycles]

Number of accesses

- cache hits
Timing differences

Access time [CPU cycles]

Number of accesses

.cache hits
cache misses
Cache attacks

- cache attacks → exploit timing differences of memory accesses
Cache attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
Cache attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
- covert channel: two processes communicating with each other
  - not allowed to do so, e.g., across VMs
Cache attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
- covert channel: two processes communicating with each other
  - not allowed to do so, e.g., across VMs
- side-channel attack: one malicious process spies on benign processes
  - e.g., steals crypto keys, spies on keystrokes
Cache attacks techniques

• two (main) techniques
  1. Flush+Reload (Gullasch et al., Osvik et al., Yarom et al.)
  2. Prime+Probe (Percival, Osvik et al., Liu et al.)

• exploitable on x86 and ARM

**Cache attacks: Flush+Reload**

Step 1: Attacker maps shared library (shared memory, in cache)
Cache attacks: Flush+Reload

Step 1: Attacker maps shared library (shared memory, in cache)
Cache attacks: Flush+Reload

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

Step 2: Attacker *flushes* the shared cache line
Cache attacks: 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
Cache attacks: 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
What if there is no shared memory?
Inclusive property

- Inclusive LLC: superset of L1 and L2
• inclusive LLC: superset of L1 and L2
• inclusive LLC: superset of L1 and L2
Inclusive property

- inclusive LLC: superset of L1 and L2

- data evicted from the LLC is also evicted from L1 and L2
- a core can evict lines in the private L1 of another core
Inclusive property

- **inclusive LLC**: superset of L1 and L2
- Data evicted from the LLC is also evicted from L1 and L2
Inclusive property

- inclusive LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2
- a core can **evict lines** in the private L1 of another core
Cache attacks: Prime+Probe

Victim address space

Cache

Attacker address space
Cache attacks: Prime+Probe

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

Step 2: Victim evicts cache lines while running
Step 1: Attacker primes, i.e., fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running
Cache attacks: Prime+Probe

**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
Cache attacks: Prime+Probe

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
Challenges with Prime+Probe

We need to evict caches lines without `clflush` or shared memory:

1. which addresses do we access to have congruent cache lines?
2. without any privilege?
3. and in which order do we access them?
We need to evict cache lines without `clflush` or shared memory:

1. which addresses do we access to have congruent cache lines?
2. without any privilege?
3. and in which order do we access them?
Last-level cache addressing

physical address

<table>
<thead>
<tr>
<th>35</th>
<th>17</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>set</td>
<td>offset</td>
<td></td>
</tr>
</tbody>
</table>

line

slice 0

slice 1

slice 2

slice 3
Last-level cache addressing

- last-level cache $\rightarrow$ as many slices as cores
- **undocumented** hash function that maps a physical address to a slice
- designed for performance

For $2^k$ slices:

- physical address: 30 bits
- slice $(o_0, \ldots, o_{k-1})$: $k$ bits
Prime+Probe on recent processors?

Undocumented function → impossible to target the same set in the same slice

Victim address space

Cache

Attacker address space
Reverse-engineering last-level cache
Reverse engineering method

1. find some way to map one address to one slice
Reverse engineering method

1. find some way to map one address to one slice
2. repeat for every address with a 64B stride
Reverse engineering method

1. find some way to map **one address** to **one slice**
2. **repeat** for every address with a 64B stride
3. infer a **function** out of it
How to map addresses to slices?

- with performance counters (Maurice et al., 2015)
- with a timing attack
  - using `clflush` (using Gruss et al., 2016)
  - using memory accesses (Yarom et al., 2015)
How to map addresses to slices?

• with performance counters (Maurice et al., 2015)
• with a timing attack
  • using clflush (using Gruss et al., 2016)
  • using memory accesses (Yarom et al., 2015)

C. Maurice et al. “Reverse Engineering Intel Complex Addressing Using Performance Counters”. In: RAID’15. 2015
Mapping with performance counters

- event **UNC_CBO_CACHE_LOOKUP** counts accesses to a slice

```
address

H

CBo 0  CBo 1  CBo 2  CBo 3

slice 0  slice 1  slice 2  slice 3

UNC_CBO_CACHE_LOOKUP  0  0  0  0
```
Mapping with performance counters

- event UNC_CBO_CACHE_LOOKUP counts accesses to a slice

![Diagram showing mapping with performance counters]

UNC_CBO_CACHE_LOOKUP

<table>
<thead>
<tr>
<th>Event</th>
<th>Slice 0</th>
<th>Slice 1</th>
<th>Slice 2</th>
<th>Slice 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
• event `UNC_CBO_CACHE_LOOKUP` counts accesses to a slice
Mapping with performance counters

- event `UNC_CBO_CACHE_LOOKUP` counts accesses to a slice

![Diagram showing the mapping of slices and access counts to CBo]
1. translate virtual to physical address with `/proc/pid/pagemap`
Mapping a physical address to a slice

1. translate virtual to physical address with /proc/pid/pagemap
2. set up monitoring session
1. translate virtual to physical address with `/proc/pid/pagemap`
2. set up monitoring session
3. repeat access to a single address
1. translate virtual to physical address with `/proc/pid/pagemap`
2. set up monitoring session
3. repeat access to a single address
   → `clflush` is already counted as an access
Mapping a physical address to a slice

1. translate virtual to physical address with /proc/pid/pagemap
2. set up monitoring session
3. repeat access to a single address
   → clflush is already counted as an access
4. read UNC_CBO_CACHE_LOOKUP event for each CBo
Mapping a physical address to a slice

1. translate virtual to physical address with \texttt{/proc/pid/pagemap}
2. set up monitoring session
3. repeat access to a single address
   \quad \rightarrow \texttt{clflush} is already counted as an access
4. read \texttt{UNC\_CBO\_CACHE\_LOOKUP} event for each CBo
5. slice is the one that has the maximum lookup events
Mapping physical addresses to slices

Number of lookup events

- CBo 0
- CBo 1
- CBo 2
- CBo 3
Inferring the function

Two cases:

1. $2^n$ number of cores: linear function $\rightarrow$ XORs of address bits
   - solve the linear equation
   - or brute force (not that long)

2. the remainder: non-linear function
Last-level cache linear functions

3 functions, depending on the number of cores

<table>
<thead>
<tr>
<th></th>
<th>Address bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 3 3 3 3 3 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 0 0 0 0</td>
</tr>
<tr>
<td>2 cores</td>
<td>3 3 3 3 3 3 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 0 0 0 0</td>
</tr>
<tr>
<td>4 cores</td>
<td>3 3 3 3 3 3 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 0 0 0 0</td>
</tr>
<tr>
<td>8 cores</td>
<td>3 3 3 3 3 3 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 0 0 0 0</td>
</tr>
</tbody>
</table>

Function valid for Sandy Bridge, Ivy Bridge, Haswell, Broadwell
Practical applications
Covert channels in the cloud

- covert channel: two processes communicating with each other
  - not allowed to do so, e.g., across VMs
Covert channels in the cloud

- covert channel: two processes communicating with each other
  - not allowed to do so, e.g., across VMs
- literature: stops working with noise on the machine
Covert channels in the cloud

- covert channel: two processes communicating with each other
  - not allowed to do so, e.g., across VMs
- literature: stops working with noise on the machine
- solution? “Just use error-correcting codes”
Why can’t we just use error correcting codes?

(a) Transmission without errors
Why can’t we just use error correcting codes?

(a) Transmission without errors

(b) Noise: substitution error
Why can’t we just use error correcting codes?

(a) Transmission without errors

(b) Noise: substitution error

(c) Sender descheduled: insertions
Why can’t we just use error correcting codes?

(a) Transmission without errors

Sender: 1 0 0 1 1 0
Receiver: 1 0 0 1 1 0

(b) Noise: substitution error

Sender: 1 0 0 1 1 0
Receiver: 1 1 0 1 1 0

(c) Sender descheduled: insertions

Sender: 1 0 1 0 0 1 1 0
Receiver: 1 0 0 0 0 0 1 1 0

(d) Receiver descheduled: deletions

Sender: 1 0 0 1 1 0
Receiver: 1 0 0 0 0 0 0
Our robust covert channel

• **physical** layer:
  • transmits words as a sequence of ‘0’s and ‘1’s
  • deals with synchronization errors

• **data-link** layer:
  • divides data to transmit into packets
  • corrects the remaining errors

---

C. Maurice et al. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud”. In: *NDSS’17*. 2017
• sender and receiver agree on one set
Physical layer: Sending ‘0’s and ‘1’s

- sender and receiver agree on one set
- receiver probes the set continuously
• sender and receiver agree on one set
• receiver probes the set continuously
• sender **transmits '0'** doing nothing
  → lines of the receiver still in cache → **fast access**
Physical layer: Sending ‘0’s and ‘1’s

- sender and receiver agree on one set
- receiver probes the set continuously
- sender **transmits ’0’** doing nothing
  - → lines of the receiver still in cache → **fast access**
- sender **transmits ’1’** accessing addresses in the set
  - → evicts lines of the receiver → **slow access**
Eviction set generation

- need a set of addresses in the same cache set and same slice
Eviction set generation

- need a set of addresses in the **same cache set** and **same slice**
- problem: slice number depends on all bits of the physical address
Eviction set generation

- need a set of addresses in the **same cache set and same slice**
- problem: slice number depends on all bits of the physical address

- we can build a set of addresses in the **same cache set and same slice**
Eviction set generation

- need a set of addresses in the **same cache set and same slice**
- problem: slice number depends on all bits of the physical address

![Physical Address Diagram]

- we can build a set of addresses in the **same cache set and same slice**
- without knowing **which slice**
Eviction set generation

- need a set of addresses in the **same cache set and same slice**
- problem: slice number depends on all bits of the physical address

![Diagram of physical address]

- we can build a set of addresses in the **same cache set and same slice**
- without knowing **which slice**
  → we use a **jamming agreement**
Handling synchronization errors

- Deletion errors: request-to-send scheme that also serves as ack

- 3-bit sequence number

- Request: encoded sequence number (7 bits)

- '0'-insertion errors: error detection code

- Berger codes

  - Appending the number of '0's in the word to itself

  - Property: a word cannot consist solely of '0's

Data

Physical layer word

Data

12 bits
Handling synchronization errors

- deletion errors: request-to-send scheme that also serves as ack
  - 3-bit sequence number
  - request: encoded sequence number (7 bits)
Handling synchronization errors

- deletion errors: request-to-send scheme that also serves as ack
  - 3-bit sequence number
  - request: encoded sequence number (7 bits)
- '0'-insertion errors: error detection code → Berger codes
  - appending the number of '0's in the word to itself
  → property: a word cannot consist solely of '0's
Synchronization (before)
Synchronization (after)
Synchronization (after)
Data-link layer: Error correction

- Reed-Solomon codes to correct the remaining errors
Data-link layer: Error correction

- Reed-Solomon codes to correct the remaining errors
- RS word size = physical layer word size = 12 bits
- packet size = \(2^{12} - 1 = 4095\) RS words
- 10% error-correcting code: 409 parity and 3686 data RS words
Error correction (after)
<table>
<thead>
<tr>
<th>Environment</th>
<th>Bit rate</th>
<th>Error rate</th>
<th>Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>75.10 KBps</td>
<td>0.00%</td>
<td>–</td>
</tr>
<tr>
<td>Environment</td>
<td>Bit rate</td>
<td>Error rate</td>
<td>Noise</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Native</td>
<td>75.10 KBps</td>
<td>0.00%</td>
<td>–</td>
</tr>
<tr>
<td>Native</td>
<td>36.03 KBps</td>
<td>0.00%</td>
<td>stress -m 1</td>
</tr>
</tbody>
</table>

The table above shows the bit rate, error rate, and noise for different environments. The noise level is indicated by `–` for Native environments and `stress -m 1` for the stress test on the sender and receiver VMs. The bit rate and error rate are measured in Kilobits per second (KBps) and as a percentage, respectively.
## Evaluation

<table>
<thead>
<tr>
<th>Environment</th>
<th>Bit rate</th>
<th>Error rate</th>
<th>Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>75.10 KBps</td>
<td>0.00%</td>
<td>–</td>
</tr>
<tr>
<td>Native</td>
<td>36.03 KBps</td>
<td>0.00%</td>
<td>stress -m 1</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>45.25 KBps</td>
<td>0.00%</td>
<td>–</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment</th>
<th>Bit rate</th>
<th>Error rate</th>
<th>Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>75.10 KBps</td>
<td>0.00%</td>
<td>–</td>
</tr>
<tr>
<td>Native</td>
<td>36.03 KBps</td>
<td>0.00%</td>
<td>stress -m 1</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>45.25 KBps</td>
<td>0.00%</td>
<td>–</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>45.09 KBps</td>
<td>0.00%</td>
<td>web server serving files on sender VM</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>42.96 KBps</td>
<td>0.00%</td>
<td>stress -m 2 on sender VM</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>42.26 KBps</td>
<td>0.00%</td>
<td>stress -m 1 on receiver VM</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>37.42 KBps</td>
<td>0.00%</td>
<td>web server on all 3 VMs, stress -m 4 on 3rd VM, stress -m 1 on sender and receiver VMs</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>34.27 KBps</td>
<td>0.00%</td>
<td>stress -m 8 on third VM</td>
</tr>
</tbody>
</table>
Building an SSH connection

- **Hypervisor**
- **Last Level Cache (LLC)**
- **Covert Channel**
- **Prime+Probe**
- **TCP**
- **File System**
- **TCP Client** (e.g. ssh)
- **TCP Server** (e.g. sshd)
- **Socket**
- **File**
## SSH evaluation

### Between two instances on Amazon EC2

<table>
<thead>
<tr>
<th>Noise</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>No noise</td>
<td>✓</td>
</tr>
<tr>
<td><strong>stress</strong> -m 8 on third VM</td>
<td>✓</td>
</tr>
<tr>
<td>Web server on third VM</td>
<td>✓</td>
</tr>
<tr>
<td>Web server on SSH server VM</td>
<td>✓</td>
</tr>
<tr>
<td>Web server on all VMs</td>
<td>✓</td>
</tr>
<tr>
<td><strong>stress</strong> -m 1 on server side</td>
<td>unstable</td>
</tr>
</tbody>
</table>
SSH evaluation

Between two instances on Amazon EC2

<table>
<thead>
<tr>
<th>Noise</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>No noise</td>
<td>✓</td>
</tr>
<tr>
<td><code>stress -m 8</code> on third VM</td>
<td>✓</td>
</tr>
<tr>
<td>Web server on third VM</td>
<td>✓</td>
</tr>
<tr>
<td>Web server on SSH server VM</td>
<td>✓</td>
</tr>
<tr>
<td>Web server on all VMs</td>
<td>✓</td>
</tr>
<tr>
<td><code>stress -m 1</code> on server side</td>
<td>unstable</td>
</tr>
</tbody>
</table>

Telnet also works with occasional corrupted bytes with `stress -m 1`
Countermeasures
Countermeasures

- different levels: hardware, system, application
- different goals
  - remove interferences
  - add noise to interferences
  - make it impossible to measure interferences
Hardware level: Fixing the instruction set?

- clflush
  - unprivileged line eviction

- rdtsc
  - unprivileged fine-grained timing
    - make it privileged
    - require changes to the architecture
    - there are other sources of timing attacks still possible (e.g., Prime+Probe)
• clflush
  • unprivileged line eviction → make it privileged
Hardware level: Fixing the instruction set?

- **clflush**
  - unprivileged line eviction → make it privileged
  - leaks timing information
Hardware level: Fixing the instruction set?

- **clflush**
  - unprivileged line eviction → make it **privileged**
  - leaks timing information → make it **constant-time**

- require changes to the architecture
- there are other sources of timing attacks still possible (e.g., Prime+Probe)
Hardware level: Fixing the instruction set?

- **clflush**
  - unprivileged line eviction $\rightarrow$ make it privileged
  - leaks timing information $\rightarrow$ make it constant-time
- **rdtsc**
  - unprivileged fine-grained timing
Hardware level: Fixing the instruction set?

- **clflush**
  - unprivileged line eviction → make it privileged
  - leaks timing information → make it constant-time

- **rdtsc**
  - unprivileged fine-grained timing → make it privileged
Hardware level: Fixing the instruction set?

- clflush
  - unprivileged line eviction → make it privileged
  - leaks timing information → make it constant-time
- rdtsc
  - unprivileged fine-grained timing → make it privileged

→ require changes to the architecture
Hardware level: Fixing the instruction set?

- **clflush**
  - unprivileged line eviction → make it **privileged**
  - leaks timing information → make it **constant-time**

- **rdtsc**
  - unprivileged fine-grained timing → make it **privileged**

→ require changes to the architecture
→ there are other sources of timing
→ attacks still possible (e.g., Prime+Probe)
Hardware level: Stop sharing hardware?

- stop sharing cache

---

Hardware level: Stop sharing hardware?

- **stop sharing cache** → attacks are getting better
  - first attacks on L1 → same core

---


Hardware level: Stop sharing hardware?

- **stop sharing cache** → attacks are getting better
  - first attacks on L1 → same core → stop sharing core

---


Hardware level: Stop sharing hardware?

- **stop sharing cache** → attacks are getting better
  - first attacks on L1 → same core → stop sharing core
  - current attacks on LLC → across cores

---


Hardware level: Stop sharing hardware?

- **stop sharing cache** → attacks are getting better
  - first attacks on L1 → same core → stop sharing core
  - current attacks on LLC → across cores → stop sharing CPU

---

Hardware level: Stop sharing hardware?

- **stop sharing cache** → attacks are getting better
  - first attacks on L1 → same core → stop sharing core
  - current attacks on LLC → across cores → stop sharing CPU
  - 2016: first attack across processors

---


Hardware level: Stop sharing hardware?

- **stop sharing cache** → attacks are getting better
  - first attacks on L1 → same core → stop sharing core
  - current attacks on LLC → across cores → stop sharing CPU
  - 2016: first attack across processors → what next?

---

Hardware level: Stop sharing hardware?

- **stop sharing cache** → attacks are getting better
  - first attacks on L1 → same core → stop sharing core
  - current attacks on LLC → across cores → stop sharing CPU
  - 2016: first attack across processors → what next?

- **not an option** for cost reasons in the cloud

---


Hardware level: Stop sharing hardware?

- **stop sharing cache** → attacks are getting better
  - first attacks on L1 → same core → stop sharing core
  - current attacks on LLC → across cores → stop sharing CPU
  - 2016: first attack across processors → **what next?**

- **not an option** for cost reasons in the cloud

- what about JavaScript attacks?

---


Hardware level: Changes in microarchitecture

- secure cache designs: Random-Permutation Cache, Partition-Locked Cache

---


J. Kong et al. “Hardware-software integrated approaches to defend against software cache-based side channel attacks”. In: HPCA’09. 2009.

Hardware level: Changes in microarchitecture

- secure cache designs: Random-Permutation Cache, Partition-Locked Cache
  → expensive, not always high performance

---


J. Kong et al. “Hardware-software integrated approaches to defend against software cache-based side channel attacks”. In: HPCA’09. 2009.

Hardware level: Changes in microarchitecture

- secure cache designs: Random-Permutation Cache, Partition-Locked Cache → expensive, not always high performance
- disruptive prefetching: random hardware prefetches

---


J. Kong et al. “Hardware-software integrated approaches to defend against software cache-based side channel attacks”. In: HPCA’09. 2009.

Hardware level: Changes in microarchitecture

- secure cache designs: Random-Permutation Cache, Partition-Locked Cache
  → expensive, not always high performance
- disruptive prefetching: random hardware prefetches
  → adding noise makes attacks harder, not impossible

---


J. Kong et al. “Hardware-software integrated approaches to defend against software cache-based side channel attacks”. In: HPCA’09. 2009.

Hardware level: Changes in microarchitecture

- secure cache designs: Random-Permutation Cache, Partition-Locked Cache
  → expensive, not always high performance
- disruptive prefetching: random hardware prefetches
  → adding noise makes attacks harder, not impossible
  → trade-off security/performance/cost

---


J. Kong et al. “Hardware-software integrated approaches to defend against software cache-based side channel attacks”. In: HPCA’09. 2009.

Hardware level: Changes in microarchitecture

- secure cache designs: Random-Permutation Cache, Partition-Locked Cache
  → expensive, not always high performance
- disruptive prefetching: random hardware prefetches
  → adding noise makes attacks harder, not impossible
  → trade-off security/performance/cost
  → performance and cost win, no implementation by manufacturers

J. Kong et al. “Hardware-software integrated approaches to defend against software cache-based side channel attacks”. In: HPCA’09. 2009.
System level: Prevention

• L1 cache cleansing

Y. Zhang et al. “Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud”. In: CCS’13. 2013
B. C. Vattikonda et al. “Eliminating fine grained timers in Xen”. In: CCSW’11. 2011
System level: Prevention

- L1 cache cleansing
  → if applied to LLC → same as no cache, *disastrous performance*

---

Y. Zhang et al. “Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud”. In: CCS’13. 2013
B. C. Vattikonda et al. “Eliminating fine grained timers in Xen”. In: CCSW’11. 2011
System level: Prevention

- L1 cache cleansing
  → if applied to LLC → same as no cache, disastrous performance
- page coloring → reduces cache sharing

---

Y. Zhang et al. “Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud”. In: CCS’13. 2013


B. C. Vattikonda et al. “Eliminating fine grained timers in Xen”. In: CCSW’11. 2011

System level: Prevention

- L1 cache cleansing
  - if applied to LLC → same as no cache, disastrous performance
- page coloring → reduces cache sharing
  - limited number of colors + bad performance
  - doesn’t prevent Flush+Reload

---

Y. Zhang et al. “ Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud”. In: CCS’13. 2013
B. C. Vattikonda et al. “Eliminating fine grained timers in Xen”. In: CCSW’11. 2011
System level: Prevention

- L1 cache cleansing
  - if applied to LLC → same as no cache, disastrous performance

- page coloring → reduces cache sharing
  - limited number of colors + bad performance
  - doesn’t prevent Flush+Reload

- noise in timers or no timer

Y. Zhang et al. “Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud”. In: CCS’13. 2013
B. C. Vattikonda et al. “Eliminating fine grained timers in Xen”. In: CCSW’11. 2011
System level: Prevention

- **L1 cache cleansing**
  → if applied to LLC → same as no cache, *disastrous performance*

- **Page coloring** → reduces cache sharing
  → limited number of colors + *bad performance*
  → doesn’t prevent Flush+Reload

- **Noise in timers or no timer**
  → adding noise makes attacks harder, not impossible
  → removing timers is *not realistic*

---

Y. Zhang et al. “Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud”. In: CCS’13. 2013


B. C. Vattikonda et al. “Eliminating fine grained timers in Xen”. In: CCSW’11. 2011

System level: Detect on-going attacks

- using **performance counters** to monitor cache hits and cache misses → risk of false positives

![Bar chart showing cache misses and hits for Firefox, OpenTTD, stress -m 1, Flush+Reload, and Rowhammer](chart.png)

_N. Herath et al. “These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security”. In: Black Hat 2015 Briefings. 2015_

Application level: Detect leakage

- CacheAudit: static analysis of source code
- Cache Template Attacks: dynamic approach

Application level: Detect leakage

- CacheAudit: static analysis of source code
- Cache Template Attacks: dynamic approach

→ limited to side-channels → covert channels still possible
→ most effective for critical code

Application level: Write better code

- no branch or data access depending on a secret
- hardware implementations (AES-NI, etc.)
Application level: Write better code

- no branch or data access depending on a secret
- hardware implementations (AES-NI, etc.)

→ protecting crypto is possible and necessary!

→ a few CVEs that have been treated: CVE-2005-0109, CVE-2013-4242,
   CVE-2014-0076, CVE-2016-0702, CVE-2016-2178, CVE-2016-7440, CVE-2016-7439,
   CVE-2016-7438, CVE-2018-0737, ...
Bigger perspective and conclusions
rdseed and floating point operations

- **rdseed**
  - request a random seed to the *hardware random number generator*
  - fixed number of precomputed random bits, takes time to regenerate them
  $\rightarrow$ covert channel

---

D. Evtyushkin et al. “Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations”. In: *CCS’16*. 2016

**rdseed and floating point operations**

- **rdseed**
  - request a random seed to the **hardware random number generator**
  - fixed number of precomputed random bits, takes time to regenerate them
  → covert channel

- **fadd, fmul**
  - **floating-point unit**
  - floating point operations running time depends on the operands
  → bypassing Firefox’s same origin policy via SVG filter timing attack

---


M. Andrysco et al. “On subnormal floating point and abnormal timing”. In: S&P’15. 2015
jmp and TSX instructions

- jmp
  - branch prediction and branch target prediction → **branch prediction unit**
  - covert channels, side-channel attacks on crypto, bypassing kernel ASLR

---


**jmp and TSX instructions**

- **jmp**
  - branch prediction and branch target prediction → **branch prediction unit**
  - covert channels, side-channel attacks on crypto, bypassing kernel ASLR

- **TSX instructions**
  - extension for **transactional memory** support in hardware
  - bypassing kernel ASLR

---


It’s not just caches!

DRAM, GPU, MMU, TLB...

Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks

Bon Grze, Vrije Universiteit Amsterdam
Kaveh Razavi Vrije Universiteit Amsterdam
Herbert Bos Vrije Universiteit Amsterdam
Cristiane Giuffrida Vrije Universiteit Amsterdam

Abstract

In cloud computing environments, multiple tenants are often co-hosted on the same physical server. Thus, preventing information leakage between tenants is a major challenge. While the isolation between tenants is achieved with virtualization, shared hardware, such as the CPU cache, can leak sensitive information. For security reasons, device memory between tenants is usually isolated. Furthermore, processing virtualization does not allow a virtual CPU to access the memory of another virtual CPU. If this setting cache attacks do not work and only show a limited benefit, we propose a new technique called Translation Leak-aside Buffer (TLB) attacks. These attacks have been shown to leak secret information in a reliable and fine-grained way, as well as compromising fundamental security defenses such as ASLR. The most prominent class of side-channel attack-leak information via the shared CPU data or instruction caches. Hence, the community has developed a variety of powerful new defenses to protect shared caches against these attacks. In this paper, we revisit this assumption that all shared cache accesses can leak secret information. We show that if microarchitectural attacks can leak secret information, then virtualization that is based on stealing sensitive information from the virtual machine becomes impossible. By using our novel techniques, we can force the virtual machine to share secret information in a way that makes it impossible for the attacker. As a result, we demonstrate a new attack on virtualization libraries that allows us to leak secret information even when the virtual machine is isolated from the physical machine.
• more a problem of **CPU design** than Instruction Set Architecture
Conclusion

• more a problem of CPU design than Instruction Set Architecture
• it’s also not just the cache
more a problem of **CPU design** than Instruction Set Architecture

it’s also not just the cache

hard to patch → issues linked to performance **optimizations**
Conclusion

• more a problem of CPU design than Instruction Set Architecture
• it’s also not just the cache
• hard to patch → issues linked to performance optimizations
• we would like to keep the optimizations without the attacks
• more a problem of CPU design than Instruction Set Architecture
• it’s also not just the cache
• hard to patch → issues linked to performance optimizations
• we would like to keep the optimizations without the attacks
• very interesting and active field of research!
Questions?

Contact

✉️ clementine.maurice@irisa.fr
🐦 @BloodyTangerine
Introduction to cache side-channel attacks

Clémentine Maurice, CNRS, IRISA
October 9, 2018—Séminaire du DIT, ENS Rennes


