Cache side-channel attacks

Lab: Monitoring keystroke timing with no privilege

Clémentine Maurice, CNRS, IRISA

July 13, 2018—Summer School Cyber in Occitanie 2018, Montpellier, France

Who am I

- Clémentine Maurice
- Full-time CNRS researcher (Chargée de Recherche)
- IRISA lab, EMSEC group
- clementine.maurice@irisa.fr
- @BloodyTangerine

Scope

- everyday hardware: servers, workstations, laptops, smartphones...

Scope

- everyday hardware: servers, workstations, laptops, smartphones...
- remote side-channel attacks

Side channels

- safe software infrastructure $\rightarrow$ no bugs, e.g., Heartbleed

Side channels

- safe software infrastructure → no bugs, e.g., Heartbleed
- does not mean safe execution

Side channels

- safe software infrastructure → no bugs, e.g., Heartbleed
- does not mean safe execution
- information leaks because of the hardware it runs on
- no “bug” in the sense of a mistake → lots of performance optimizations

Side channels

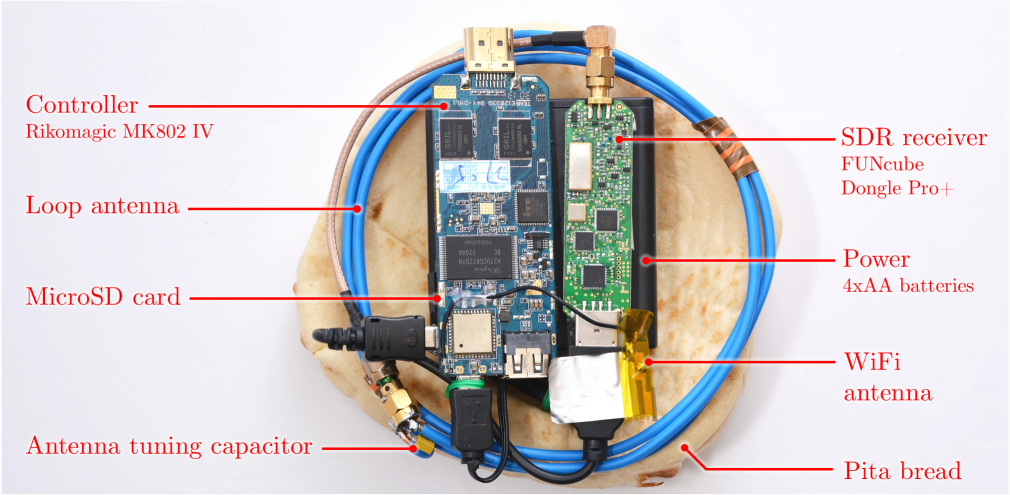
- safe software infrastructure → no bugs, e.g., Heartbleed
- does not mean safe execution
- information leaks because of the hardware it runs on
- no “bug” in the sense of a mistake → lots of performance optimizations

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

Sources of leakage

- via power consumption, electromagnetic leaks

Sources of leakage



Sources of leakage

- via power consumption, electromagnetic leaks
  $\rightarrow$ targeted attacks, physical access

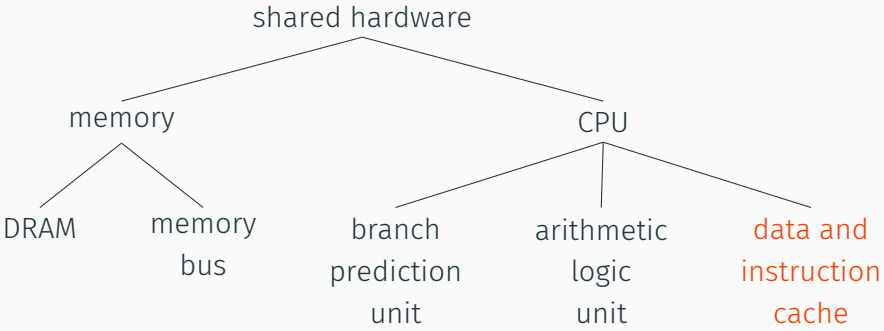
Sources of leakage

- via power consumption, electromagnetic leaks
  $\rightarrow$ 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

Shared hardware



From small optimizations to side channels

- new microarchitectures yearly

From small optimizations to side channels

- new microarchitectures yearly
- performance improvement $\approx 5\%$

From small optimizations to side channels

- new microarchitectures yearly
- performance improvement $\approx 5\%$
- very small optimizations: caches, branch prediction...

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

- 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"

Ken Shirriff, `http://www.righto.com/2013/09/intel-x86-documentation-has-more-pages.html`

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...

Ken Shirriff, `http://www.righto.com/2013/09/intel-x86-documentation-has-more-pages.html`

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 $\rightarrow$ 3510 transistors

Ken Shirriff, `http://www.righto.com/2013/09/intel-x86-documentation-has-more-pages.html`

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 $\rightarrow$ 3510 transistors
- 22-core Intel Xeon Broadwell-E5

Ken Shirriff, `http://www.righto.com/2013/09/intel-x86-documentation-has-more-pages.html`

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

Ken Shirriff, `http://www.righto.com/2013/09/intel-x86-documentation-has-more-pages.html`

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 (sept. 2016): 4670 pages

---

Ken Shirriff, `http://www.righto.com/2013/09/intel-x86-documentation-has-more-pages.html`

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 (sept. 2016): 4670 pages
- (there are actually more manuals than just the SDM)

Ken Shirriff, `http://www.righto.com/2013/09/intel-x86-documentation-has-more-pages.html`

Outline

- Background on cache attacks
- Side-channel attacks on keystroke timings
- Step-by-step attack
- Countermeasures
- Conclusion

Background on cache attacks

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8*.
REX + 88 /r	MOV *r/m8****,*r8****	MR	Valid	N.E.	Move *r8* to *r/m8*.
89 /r	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16*.
89 /r	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32*.
REX.W + 89 /r	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64*.
8A /r	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8*.
REX + 8A /r	MOV *r8****,*r/m8****	RM	Valid	N.E.	Move *r/m8* to *r8*.
8B /r	MOV *r16,r/m16*	RM	Valid	Valid	Move *r/m16* to *r16*.
8B /r	MOV *r32,r/m32*	RM	Valid	Valid	Move *r/m32* to *r32*.
REX.W + 8B /r	MOV *r64,r/m64*	RM	Valid	N.E.	Move *r/m64* to *r64*.
8C /r	MOV *r/m16,Sreg***	MR	Valid	Valid	Move segment register to *r/m16*.
REX.W + 8C /r	MOV *r/m64,Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r/m64*.
8E /r	MOV *Sreg,r/m16***	RM	Valid	Valid	Move *r/m16* to segment register.
REX.W + 8E /r	MOV *Sreg,r/m64***	RM	Valid	Valid	Move *lower 16 bits of r/m64* to segment register.
A0	MOV AL,*moffs8**	FD	Valid	Valid	Move byte at (*seg:offset*) to AL.
REX.W + A0	MOV AL,*moffs8**	FD	Valid	N.E.	Move byte at (*offset*) to AL.
A1	MOV AX,*moffs16**	FD	Valid	Valid	Move word at (*seg:offset*) to AX.
A1	MOV EAX,*moffs32**	FD	Valid	Valid	Move doubleword at (*seg:offset*) to EAX.
REX.W + A1	MOV RAX,*moffs64**	FD	Valid	N.E.	Move quadword at (*offset*) to RAX.

mov

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.

`mov`—What could go wrong?

- lots of exceptions for `mov`

`mov`—What could go wrong?

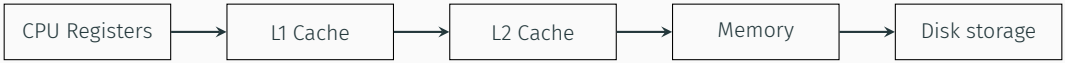
- lots of exceptions for `mov`
- but accessing data loads it to the cache

`mov`—What could go wrong?

- lots of exceptions for `mov`
- but accessing data loads it to the cache

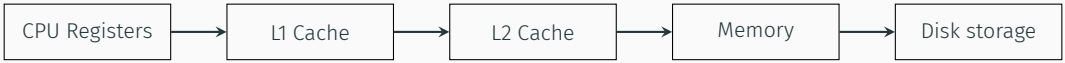
→ side effects on computations!

Memory hierarchy



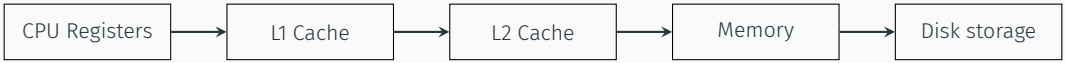
- data can reside in

Memory hierarchy



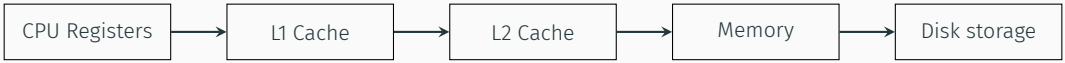
- data can reside in
  - CPU registers

Memory hierarchy



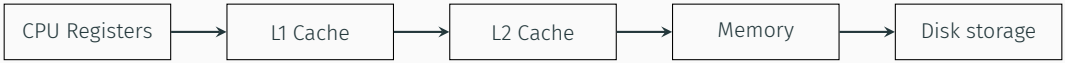
- data can reside in
  - CPU registers
  - different levels of the CPU cache

Memory hierarchy



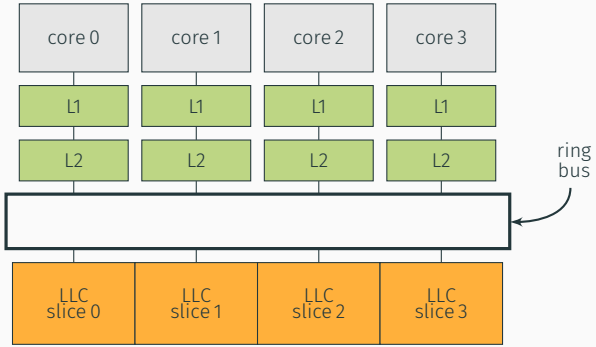
- data can reside in
  - CPU registers
  - different levels of the CPU cache
  - main memory

Memory hierarchy



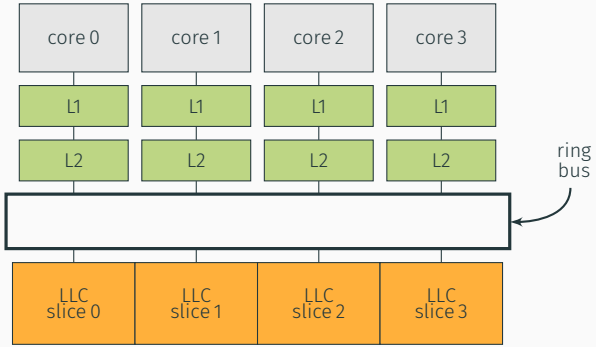
- data can reside in
  - CPU registers
  - different levels of the CPU cache
  - main memory
  - disk storage

Caches on Intel CPUs



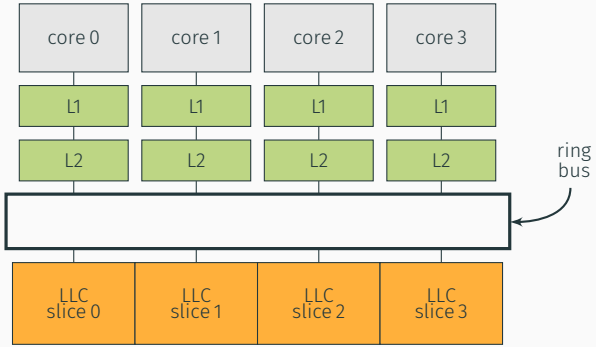
- L1 and L2 are private

Caches on Intel CPUs



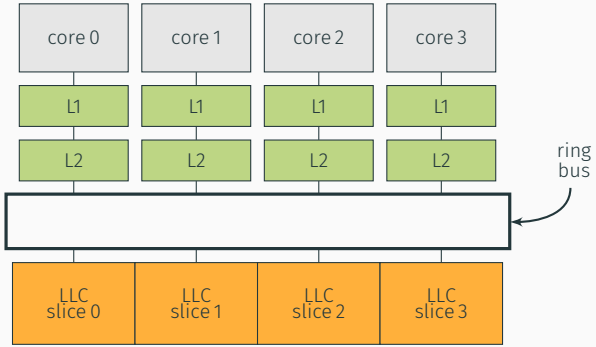
- L1 and L2 are private
- last-level cache

Caches on Intel CPUs



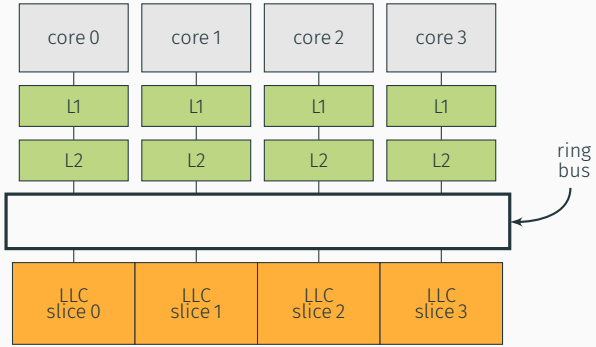
- L1 and L2 are private
- last-level cache
  - divided in slices

Caches on Intel CPUs



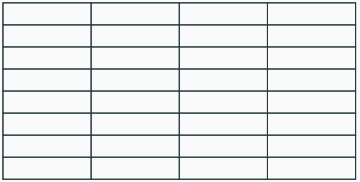
- L1 and L2 are private
- last-level cache
  - divided in slices
  - shared across cores

Caches on Intel CPUs



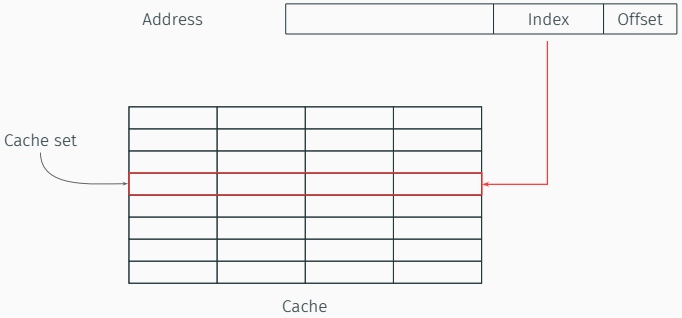
- L1 and L2 are private
- last-level cache
  - divided in slices
  - shared across cores
  - inclusive

Set-associative caches



Cache

Set-associative caches



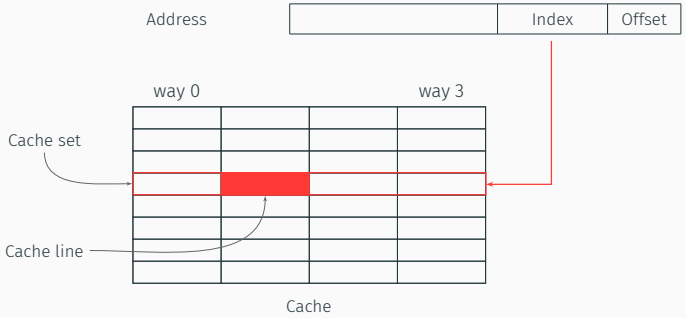
Data loaded in a specific set depending on its address

Set-associative caches

Data loaded in a specific set depending on its address

Several ways per set

Set-associative caches

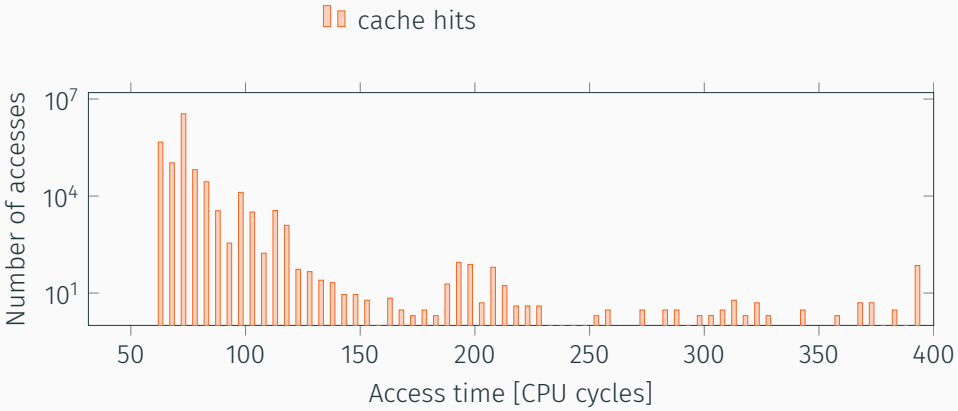


Data loaded in a specific set depending on its address

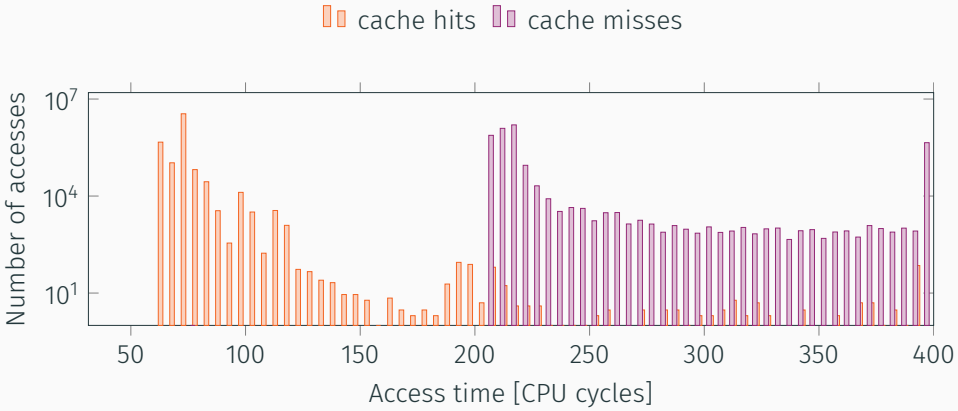
Several ways per set

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

Timing differences



Timing differences



Cache attacks

- cache attacks $\rightarrow$ exploit timing differences of memory accesses

Cache attacks

- cache attacks $\rightarrow$ exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content

Cache attacks

- cache attacks $\rightarrow$ 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 $\rightarrow$ 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

D. Gullasch et al. “Cache Games – Bringing Access-Based Cache Attacks on AES to Practice”. In: *S&P’11*. 2011.

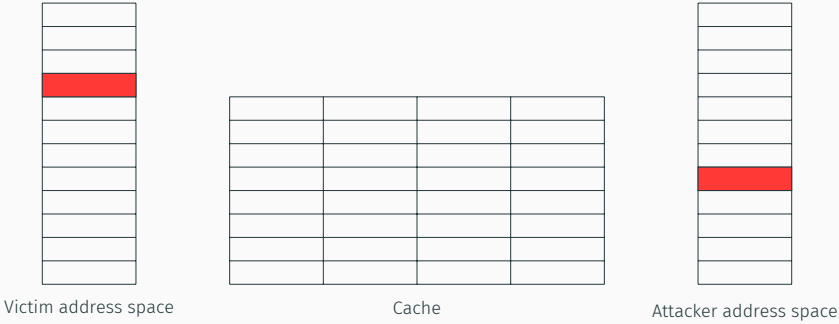
Y. Yarom et al. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *USENIX Security Symposium*. 2014.

D. A. Osvik et al. “Cache Attacks and Countermeasures: the Case of AES”. In: *CT-RSA 2006*. 2006.

C. Percival. “Cache missing for fun and profit”. In: *Proceedings of BSDCan*. 2005.

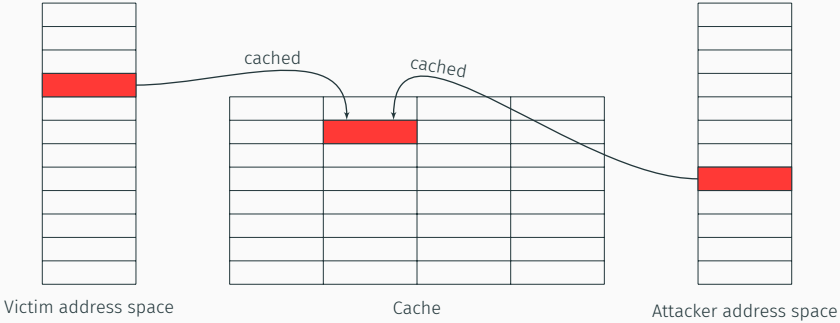
F. Liu et al. “Last-Level Cache Side-Channel Attacks are Practical”. In: *S&P’15*. 2015.

Cache attacks: Flush+Reload



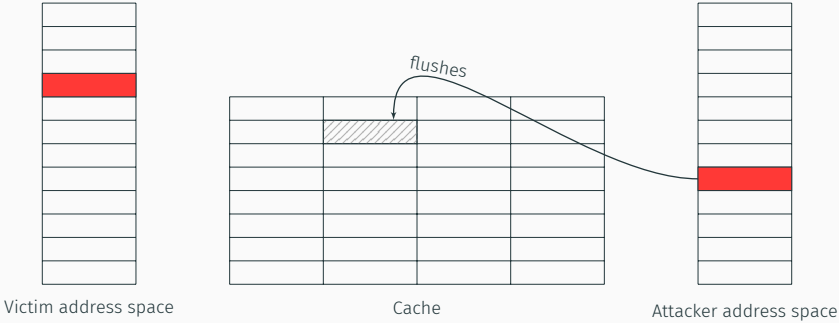
Step 1: Attacker maps shared library (shared memory, in cache)

Cache attacks: Flush+Reload



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

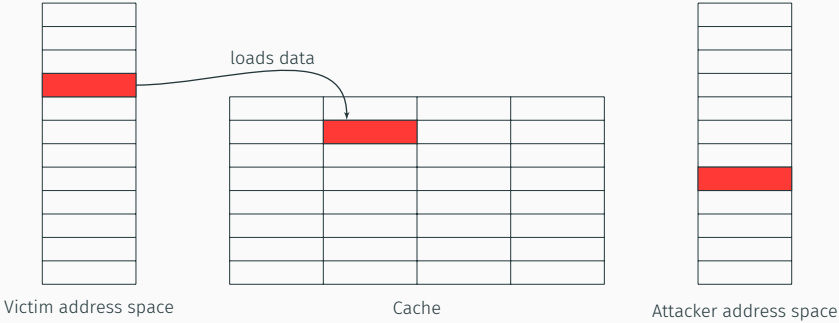
Cache attacks: Flush+Reload



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

Step 2: Attacker flushes the shared cache line

Cache attacks: Flush+Reload

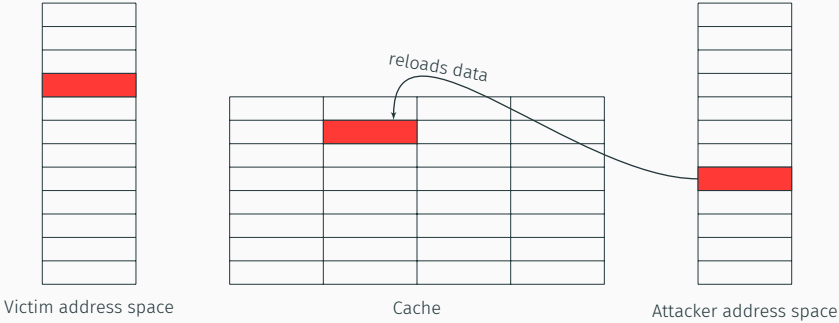


Step 1: Attacker maps shared library (shared memory, in cache)

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)

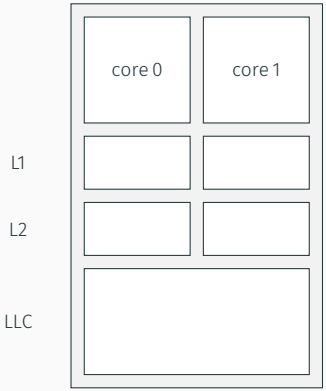
Step 2: Attacker flushes the shared cache line

Step 3: Victim loads the data

Step 4: Attacker reloads the data

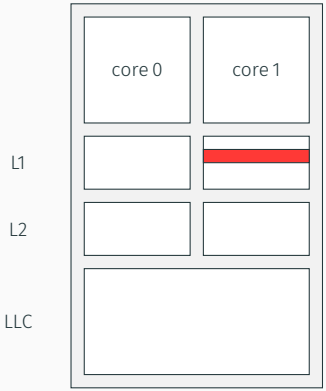
What if there is no shared memory?

Inclusive property



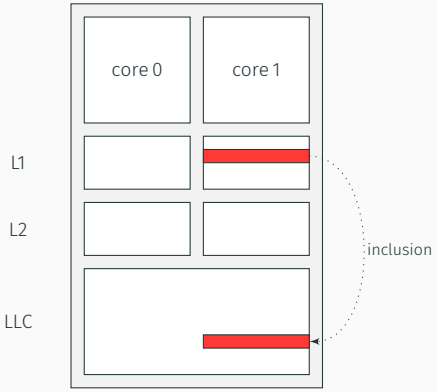
- inclusive LLC: superset of L1 and L2

Inclusive property



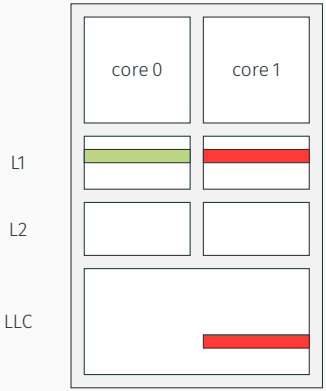
- inclusive LLC: superset of L1 and L2

Inclusive property



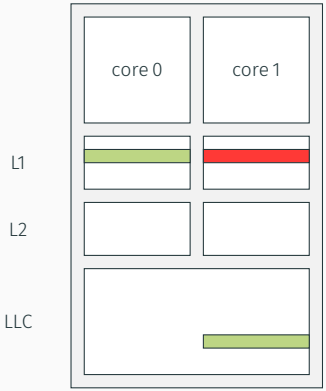
- inclusive LLC: superset of L1 and L2

Inclusive property



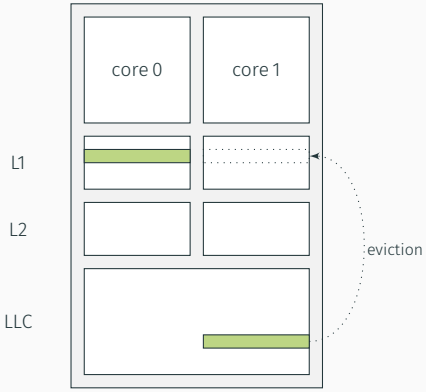
- inclusive LLC: superset of L1 and L2

Inclusive property



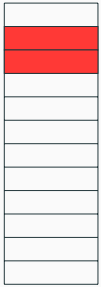
- inclusive LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2

Inclusive property

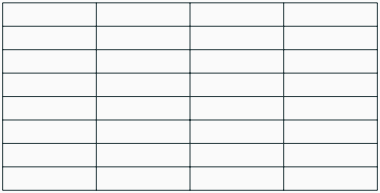


- inclusive LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2
- a core can evict lines in the private L1 of another core

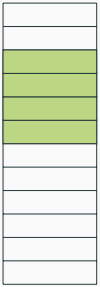
Cache attacks: Prime+Probe



Victim address space

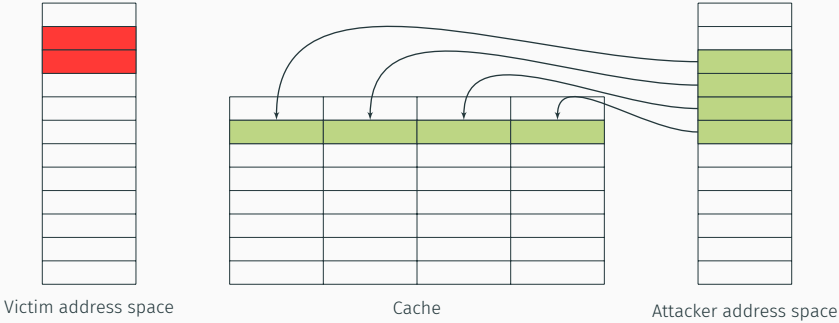


Cache



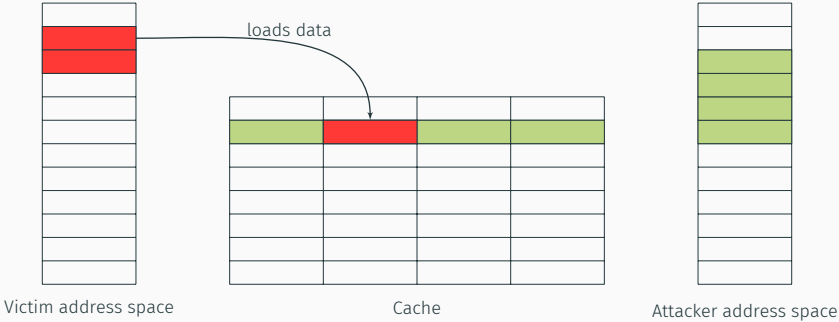
Attacker address space

Cache attacks: Prime+Probe



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

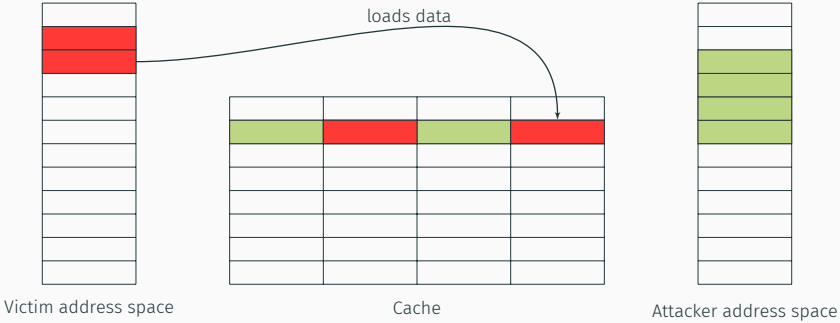
Cache attacks: Prime+Probe



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

Step 2: Victim evicts cache lines while running

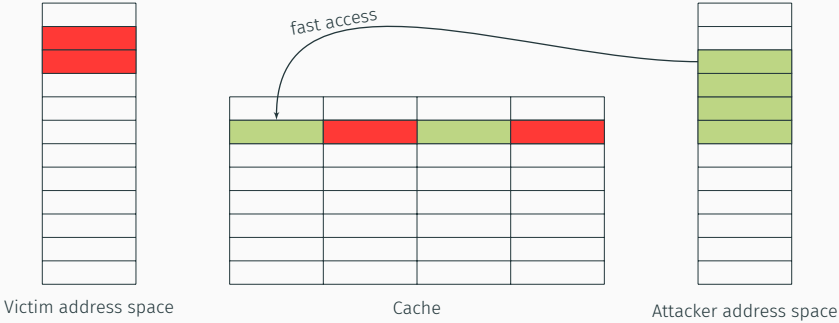
Cache attacks: Prime+Probe



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

Step 2: Victim evicts cache lines while running

Cache attacks: Prime+Probe

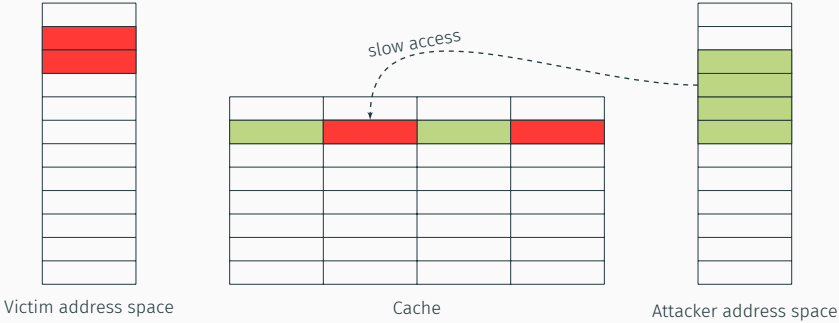


Step 1: Attacker primes, *i.e.*, fills, the cache (no shared memory)

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:

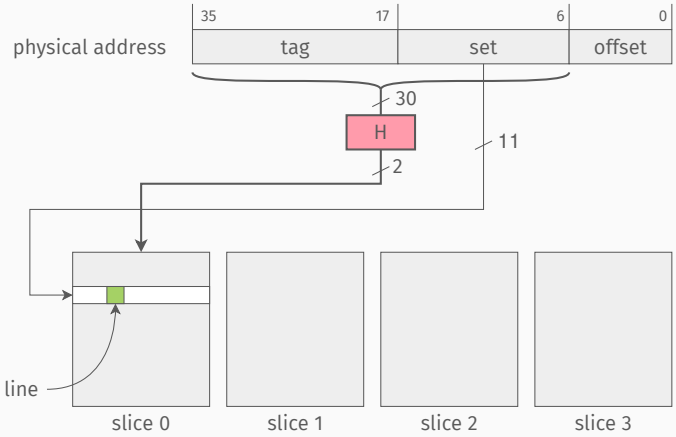
1. which addresses do we access to have congruent cache lines?
2. without any privilege?
3. and in which order do we access them?

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?

Last-level cache addressing



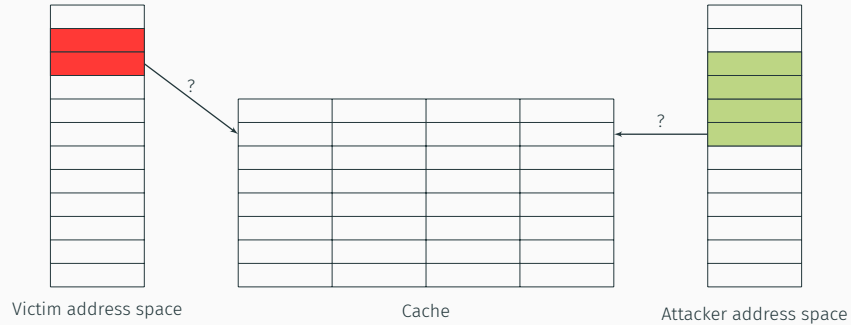
Last-level cache addressing

- 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:

Prime+Probe on recent procesors?

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



→ We reverse-engineered it!

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

Side-channel attacks on keystroke timings

Challenges in exploiting cache leakage

- how to locate key-dependent memory accesses?
- it's complicated
  - large binaries and libraries (third-party code)
  - many libraries (gedit: 60MB)
  - closed-source or unknown binaries
  - self-compiled binaries
- difficult to find all exploitable addresses

Cache Template Attacks

- locating event-dependent memory access → Cache Template Attacks

D. Gruss et al. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015.

Cache Template Attacks

- locating event-dependent memory access $\rightarrow$ Cache Template Attacks
- learning phase

D. Gruss et al. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015.

Cache Template Attacks

- locating event-dependent memory access → Cache Template Attacks
- learning phase
    1. shared library or executable is mapped

D. Gruss et al. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015.

Cache Template Attacks

- locating event-dependent memory access → Cache Template Attacks
- learning phase
  1. shared library or executable is mapped
  2. trigger an event while Flush+Reload one address

D. Gruss et al. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015.

Cache Template Attacks

- locating event-dependent memory access → Cache Template Attacks
- learning phase
  1. shared library or executable is mapped
  2. trigger an event while Flush+Reload one address
     → cache hit: address used by the library/executable

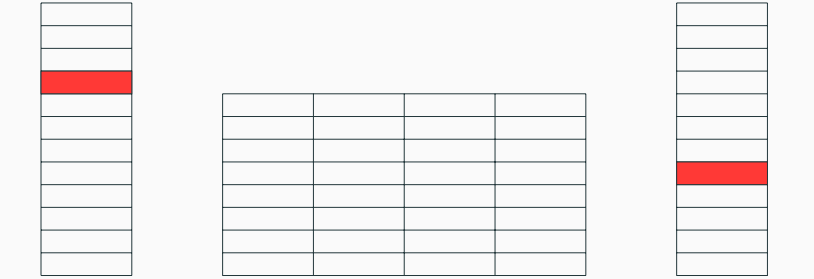
D. Gruss et al. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015.

Cache Template Attacks

- locating event-dependent memory access → Cache Template Attacks
- learning phase
  1. shared library or executable is mapped
  2. trigger an event while Flush+Reload one address
     → cache hit: address used by the library/executable
  3. repeat step 2 for every address

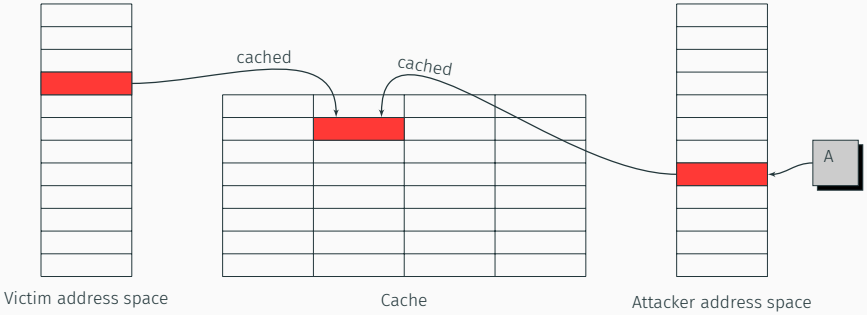
D. Gruss et al. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015.

Profiling Phase (one event)



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

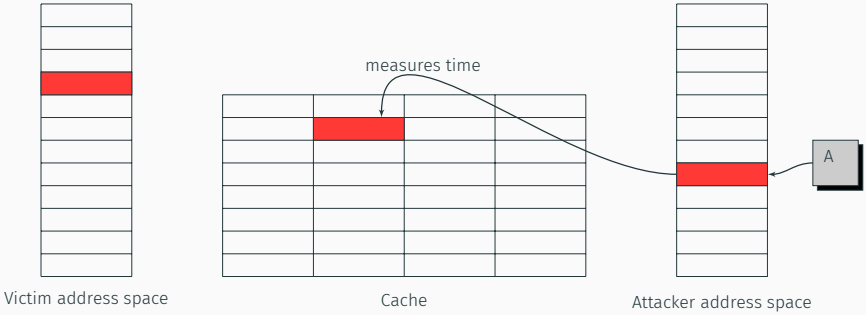
Profiling Phase (one event)



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

Step 2: Attacker triggers an event

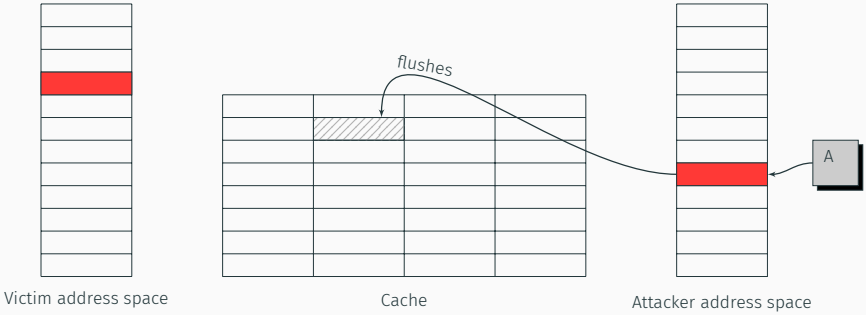
Profiling Phase (one event)



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

Step 2: Attacker triggers an event, checks cache hit

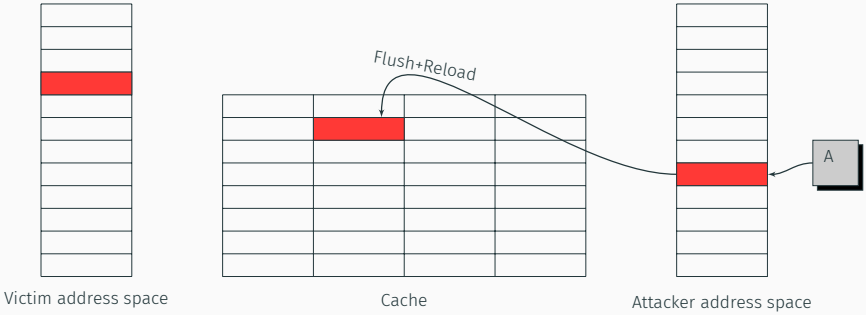
Profiling Phase (one event)



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

Step 2: Attacker triggers an event, checks cache hit, flushes the line

Profiling Phase (one event)

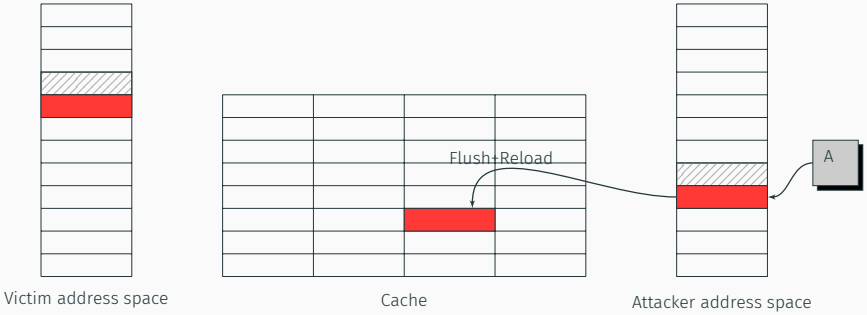


Step 1: Attacker maps shared library (shared memory, in cache), cache is empty

Step 2: Attacker triggers an event, checks cache hit, flushes the line

Step 3: Repeat for same pair (event_i, address_j) and update cache hit count

Profiling Phase (one event)



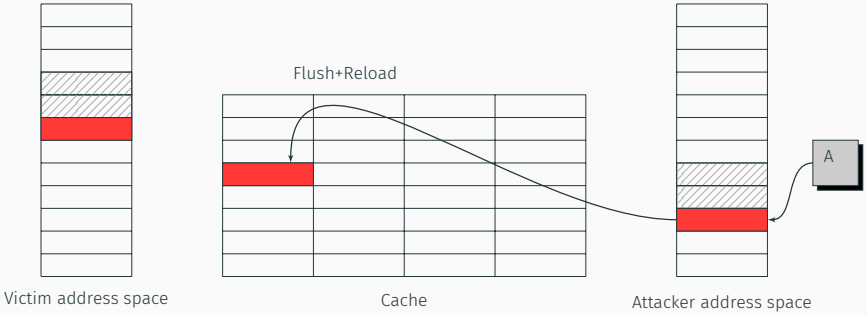
Step 1: Attacker maps shared library (shared memory, in cache), cache is empty

Step 2: Attacker triggers an event, checks cache hit, flushes the line

Step 3: Repeat for same pair (event_i, address_j) and update cache hit count

Step 4: Repeat for next pair (event_i, address_{j+1}), ...

Profiling Phase (one event)



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

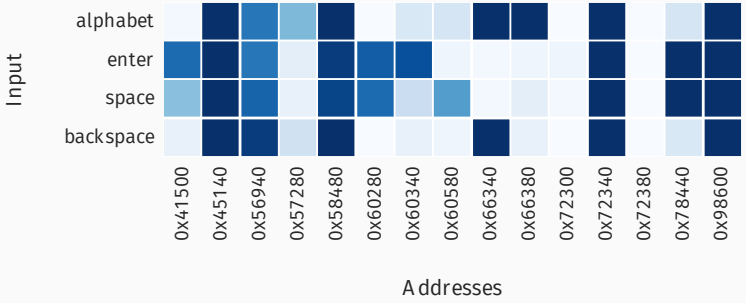
Step 2: Attacker triggers an event, checks cache hit, flushes the line

Step 3: Repeat for same pair (event_i, address_j) and update cache hit count

Step 4: Repeat for next pair (event_i, address_{j+1}), ...

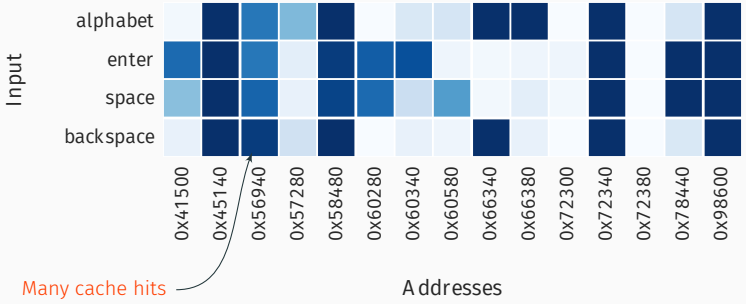
Profiling Phase (several events)

Cache template matrix: how many cache hits for each pair (event, address)?



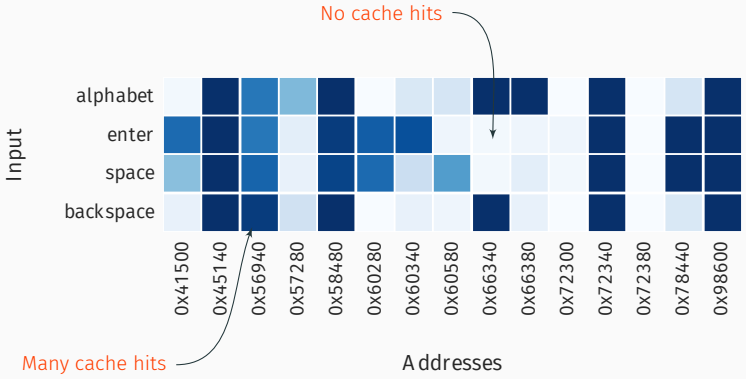
Profiling Phase (several events)

Cache template matrix: how many cache hits for each pair (event, address)?



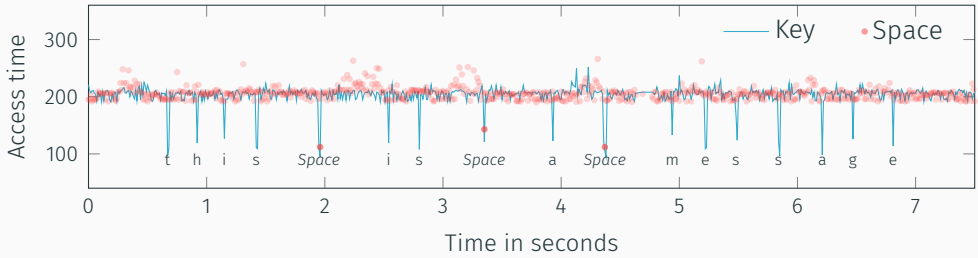
Profiling Phase (several events)

Cache template matrix: how many cache hits for each pair (event, address)?



Exploitation phase: keystrokes

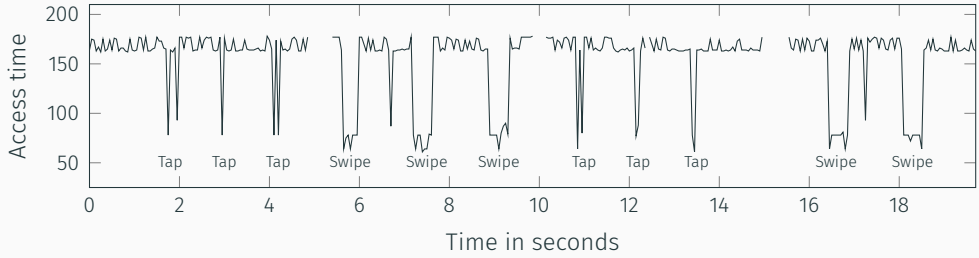
- high-resolution timers → precise inter-keystroke timing
- monitoring two addresses for keys and space
- future work: infer typed words with Hidden Markov Models



M. Lipp et al. “ARMageddon: Cache Attacks on Mobile Devices”. In: *USENIX Security Symposium*. 2016

Exploitation phase: taps and swipes on smartphones

- distinguishing between different types of events by monitoring access time



M. Lipp et al. “ARMageddon: Cache Attacks on Mobile Devices”. In: *USENIX Security Symposium*. 2016.

Side-channel attack on keystrokes: Take-away

- obtaining precise timings is easy

Side-channel attack on keystrokes: Take-away

- obtaining precise timings is easy
- further computations are needed to derive typed words

Side-channel attack on keystrokes: Take-away

- obtaining precise timings is easy
- further computations are needed to derive typed words
- but the attack also allows distinguishing key groups

Side-channel attack on keystrokes: Take-away

- obtaining precise timings is easy
- further computations are needed to derive typed words
- but the attack also allows distinguishing key groups

→ reduces search space for, e.g., password retrieval

Step-by-step attack

Setup

- we need:
  - a machine running on Linux (not virtualized)
  - an Intel CPU

Setup

- we need:
  - a machine running on Linux (not virtualized)
  - an Intel CPU
- I will demonstrate the steps on my machine but everything is ready so that you can try on yours during this session
- find a lab partner if you don't have the right setup

Repository

- clone the repository:

  ```
  git clone https://github.com/clementine-m/cache_template_attacks.git
  ```

- three folders
  1. calibration
  2. profiling
  3. exploitation

Repository

- clone the repository:

  `git clone https://github.com/clementine-m/cache_template_attacks.git`
- three folders
  1. calibration
  2. profiling
  3. exploitation
- note: if you insist on using Windows, you can find some tools in the original git repository `https://github.com/IAIK/cache_template_attacks`, but I don't provide any Windows assistance :)

#1. Calibration

Calibration

How every timing attack works:

- learn timing of different corner cases
- later, we recognize these corner cases by timing only

```
cd calibration
make
./calibration
```

Steps

1. build two cases: cache hits and cache misses
2. time each case many times (get rid of noise)

Steps

1. build two cases: cache hits and cache misses
2. time each case many times (get rid of noise)
3. we have a histogram!

Steps

1. build two cases: cache hits and cache misses
2. time each case many times (get rid of noise)
3. we have a histogram!
4. find a threshold to distinguish the two cases

Step 1.1. Cache hits

Loop:

1. measure time
2. access variable (always cache hit)
3. measure time
4. update histogram with delta

Step 1.2. Cache misses

Loop:

1. measure time
2. access variable (always cache miss)
3. measure time
4. update histogram with delta
5. flush variable (`clflush` instruction)

Step 2: Accurate timings

- very short timings
- `rdtsc` instruction: cycle-accurate timestamps

Step 2: Accurate timings

- very short timings
- `rdtsc` instruction: cycle-accurate timestamps

```
[...]
rdtsc
function()
rdtsc
[...]
```

Step 2: Accurate timings

- do you measure what you think you measure?

Step 2: Accurate timings

- do you measure what you think you measure?
- out-of-order execution

Step 2: Accurate timings

- do you measure what you think you measure?
- out-of-order execution → what is really executed

```
rdtsc
function()
[...]
rdtsc
```

```
rdtsc
[...]
rdtsc
function()
```

```
rdtsc
rdtsc
function()
[...]
```

Step 2: Accurate timings

- use pseudo-serializing instruction `rdtscp` (recent CPUs)

Step 2: Accurate timings

- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`

Step 2: Accurate timings

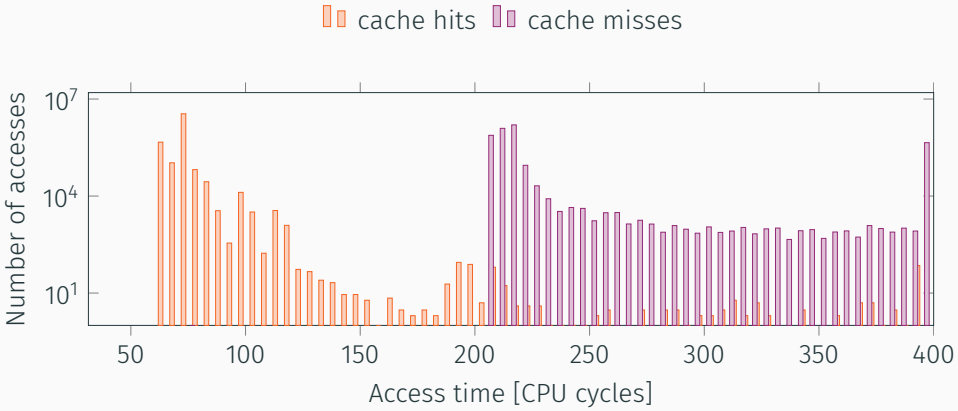
- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

Step 2: Accurate timings

- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

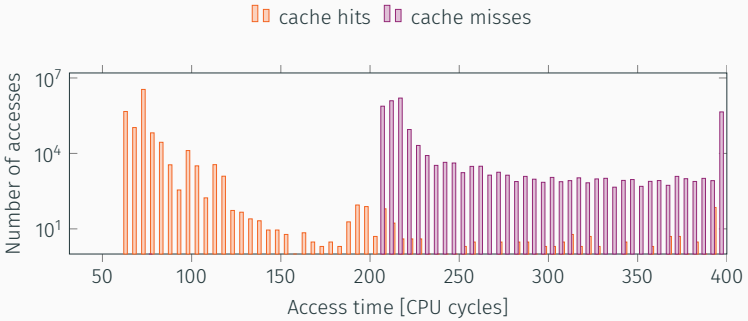
Intel, *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures White Paper*, December 2010.

Step 3: Histogram



Step 4. Find threshold

- as high as possible
- most cache hits are below
- no cache miss below



#2. Profiling

What to profile

Open gedit

(Very) ugly one-liner, from the README of the repository

```
$ cat /proc/`ps -A | grep gedit | grep -oE "^[0-9]+"`/maps |
grep r-x | grep libgedit
```

What to profile

Open gedit

(Very) ugly one-liner, from the README of the repository

```
$ cat /proc/`ps -A | grep gedit | grep -oE "^[0-9]+"`/maps |
grep r-x | grep libgedit
```

If you cannot copy paste ;)

```
$ ps -A | grep gedit                        # copy pid
$ cat /proc/<pid>/maps | grep libgedit      # copy line with r-xp
```

What to profile

Resulting line (memory range, access rights, offset, –, –, file name)

```
7f6e681ea000-7f6e682c3000 r-xp 00000000 fd:01 6423718
```

Profiling

```
$ cd profiling
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

Profiling

```
$ cd profiling
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

```
$ make
$ sleep 3; ./profiling 200 7f6e681ea000-7f6e682c3000 r-xp
00000000 fd:01 6423718
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so
```

... And hold down key in the targeted program

Profiling (a tiny bit faster)

You are probably not seeing a lot of cache hits

Profiling (a tiny bit faster)

You are probably not seeing a lot of cache hits, or any

Profiling (a tiny bit faster)

You are probably not seeing a lot of cache hits, or any

We are searching for hits from offset 0 of the library
$\rightarrow$ nothing handles keystrokes there

Profiling (a tiny bit faster)

You are probably not seeing a lot of cache hits, or any

We are searching for hits from offset 0 of the library
$\rightarrow$ nothing handles keystrokes there

Normally, run the template attack on the whole library but takes a while

Profiling (a tiny bit faster)

You are probably not seeing a lot of cache hits, or any

We are searching for hits from offset 0 of the library
$\rightarrow$ nothing handles keystrokes there

Normally, run the template attack on the whole library but takes a while
Let's start from a different offset, skipping all non executable parts

```
$ sleep 3; ./profiling 200 7f6e681ea000-7f6e682c3000 r-xp
20000 fd:01 6423718 /usr/lib/x86_64-linux-gnu/gedit/libgedit.so
```

Profiling (a tiny bit faster)

You are probably not seeing a lot of cache hits, or any

We are searching for hits from offset 0 of the library
→ nothing handles keystrokes there

Normally, run the template attack on the whole library but takes a while
Let's start from a different offset, skipping all non executable parts

```
$ sleep 3; ./profiling 200 7f6e681ea000-7f6e682c3000 r-xp
20000 fd:01 6423718 /usr/lib/x86_64-linux-gnu/gedit/libgedit.so
```

Save offsets with many cache hits!

Profiling (a tiny bit faster)

You are probably not seeing a lot of cache hits, or any

We are searching for hits from offset 0 of the library
$\rightarrow$ nothing handles keystrokes there

Normally, run the template attack on the whole library but takes a while
Let's start from a different offset, skipping all non executable parts

```
$ sleep 3; ./profiling 200 7f6e681ea000-7f6e682c3000 r-xp
20000 fd:01 6423718 /usr/lib/x86_64-linux-gnu/gedit/libgedit.so
```

Save offsets with many cache hits!

Ideally, start the profiling without triggering any event to eliminate false positives

Output

```
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so, 0x20e40,  15
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so, 0x20e80,  27
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so, 0x20ec0,   7
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so, 0x20f00,  10
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so, 0x20f40,  16
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so, 0x20f80,  13
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so, 0x20fc0,  10
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so, 0x21000,  18
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so, 0x21040,  15
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so, 0x21080,   3
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so, 0x210c0,   1
```

#3. Exploitation

Exploitation

```
$ cd profiling
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

Exploitation

```
$ cd profiling
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

```
$ make
$ ./spy <file> <offset>
```

Exploitation

```
$ cd profiling
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

```
$ make
$ ./spy <file> <offset>
```

Let’s try some offset:

Exploitation

```
$ cd profiling
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

```
$ make
$ ./spy <file> <offset>
```

Let's try some offset: lots of cache hits for `0x20c40`!!!

```
./spy /usr/lib/x86_64-linux-gnu/gedit/libgedit.so 0x20c40
```

Exploitation

```
$ cd profiling
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

```
$ make
$ ./spy <file> <offset>
```

Let's try some offset: lots of cache hits for `0x20c40`!!!

```
./spy /usr/lib/x86_64-linux-gnu/gedit/libgedit.so 0x20c40
```

A cache hit each time the cursor blinks.

Exploitation

```
$ cd profiling
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

```
$ make
$ ./spy <file> <offset>
```

Let's try some offset: lots of cache hits for `0x20c40`!!!

```
./spy /usr/lib/x86_64-linux-gnu/gedit/libgedit.so 0x20c40
```

A cache hit each time the cursor blinks. Not what we want.

Exploitation

```
$ cd profiling
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

```
$ make
$ ./spy <file> <offset>
```

Let's try some offset: lots of cache hits for `0x20c40`!!!

```
./spy /usr/lib/x86_64-linux-gnu/gedit/libgedit.so 0x20c40
```

A cache hit each time the cursor blinks. Not what we want. Let's try another one

Exploitation

```
$ cd profiling
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

```
$ make
$ ./spy <file> <offset>
```

Let's try some offset: lots of cache hits for `0x20c40`!!!

```
./spy /usr/lib/x86_64-linux-gnu/gedit/libgedit.so 0x20c40
```

A cache hit each time the cursor blinks. Not what we want. Let's try another one

```
./spy /usr/lib/x86_64-linux-gnu/gedit/libgedit.so 0x24440
```

Cleaning up the results

We have more than one cache hit per keystroke, in a very short time.

```
8588659923476: Cache Hit (167 cycles) after a pause of 1381237 cycles
8588660655587: Cache Hit (158 cycles) after a pause of 182 cycles
8588662014696: Cache Hit (142 cycles) after a pause of 388 cycles
8592435140102: Cache Hit (139 cycles) after a pause of 1254280 cycles
8592435663328: Cache Hit (152 cycles) after a pause of 120 cycles
8592436855980: Cache Hit (161 cycles) after a pause of 322 cycles
8595876762459: Cache Hit (206 cycles) after a pause of 1133098 cycles
8595877338658: Cache Hit (155 cycles) after a pause of 139 cycles
8595877386776: Cache Hit (155 cycles) after a pause of 9 cycles
8595877512170: Cache Hit (112 cycles) after a pause of 30 cycles
8595877736734: Cache Hit (152 cycles) after a pause of 57 cycles
8595878749423: Cache Hit (145 cycles) after a pause of 273 cycles
8599529228024: Cache Hit (152 cycles) after a pause of 1217393 cycles
8599529824018: Cache Hit (173 cycles) after a pause of 145 cycles
8599530032220: Cache Hit (142 cycles) after a pause of 48 cycles
8599531215638: Cache Hit (145 cycles) after a pause of 334 cycles
```

Cleaning up the results

- have a look at the `flushandreload(void* addr)` function in `spy.c`

Cleaning up the results

- have a look at the `flushandreload(void* addr)` function in `spy.c`
- `if (kpause > 0)` → modify threshold and recompile

Cleaning up the results

- have a look at the `flushandreload(void* addr)` function in `spy.c`
- `if (kpause > 0)` → modify threshold and recompile
- no false positives with `(kpause > 10000)`

Going further

- we can now obtain precise timing for keystrokes
- you can also build a complete matrix for each keystroke to identify key groups

Going further

- we can now obtain precise timing for keystrokes
- you can also build a complete matrix for each keystroke to identify key groups
- you may want to automate event triggering :)

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

Hardware level: Fixing the instruction set?

- `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

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

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

→ attacks still possible (e.g., Prime+Probe)

Hardware level: Stop sharing hardware?

- stop sharing cache

G. Irazoqui et al. “Cross processor cache attacks”. In: *AsiaCCS’16*. 2016

Y. Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS’15*. 2015

Hardware level: Stop sharing hardware?

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

G. Irazoqui et al. “Cross processor cache attacks”. In: *AsiaCCS’16*. 2016

Y. Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS’15*. 2015

Hardware level: Stop sharing hardware?

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

G. Irazoqui et al. “Cross processor cache attacks”. In: *AsiaCCS’16*. 2016

Y. Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS’15*. 2015

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

G. Irazoqui et al. “Cross processor cache attacks”. In: *AsiaCCS’16*. 2016

Y. Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS’15*. 2015

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

G. Irazoqui et al. “Cross processor cache attacks”. In: *AsiaCCS’16*. 2016

Y. Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS’15*. 2015

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

G. Irazoqui et al. “Cross processor cache attacks”. In: *AsiaCCS’16*. 2016

Y. Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS’15*. 2015

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?

G. Irazoqui et al. “Cross processor cache attacks”. In: *AsiaCCS’16*. 2016

Y. Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS’15*. 2015

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

G. Irazoqui et al. “Cross processor cache attacks”. In: *AsiaCCS’16*. 2016

Y. Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS’15*. 2015

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?

G. Irazoqui et al. “Cross processor cache attacks”. In: *AsiaCCS’16*. 2016

Y. Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS’15*. 2015

Hardware level: Changes in microarchitecture

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

Z. Wang et al. “New cache designs for thwarting software cache-based side channel attacks”. In: *ACM SIGARCH Computer Architecture News* 35.2 (June 2007), p. 494

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

A. Fuchs et al. “Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs”. In: *SYSTOR’15*. 2015

Hardware level: Changes in microarchitecture

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

Z. Wang et al. “New cache designs for thwarting software cache-based side channel attacks”. In: *ACM SIGARCH Computer Architecture News* 35.2 (June 2007), p. 494

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

A. Fuchs et al. “Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs”. In: *SYSTOR’15*. 2015

Hardware level: Changes in microarchitecture

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

Z. Wang et al. “New cache designs for thwarting software cache-based side channel attacks”. In: *ACM SIGARCH Computer Architecture News* 35.2 (June 2007), p. 494

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

A. Fuchs et al. “Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs”. In: *SYSTOR’15*. 2015

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

Z. Wang et al. “New cache designs for thwarting software cache-based side channel attacks”. In: *ACM SIGARCH Computer Architecture News* 35.2 (June 2007), p. 494

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

A. Fuchs et al. “Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs”. In: *SYSTOR’15*. 2015

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

Z. Wang et al. "New cache designs for thwarting software cache-based side channel attacks". In: *ACM SIGARCH Computer Architecture News* 35.2 (June 2007), p. 494

J. Kong et al. "Hardware-software integrated approaches to defend against software cache-based side channel attacks". In: *HPCA'09*. 2009.

A. Fuchs et al. "Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs". In: *SYSTOR'15*. 2015

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

---

Z. Wang et al. "New cache designs for thwarting software cache-based side channel attacks". In: *ACM SIGARCH Computer Architecture News* 35.2 (June 2007), p. 494

J. Kong et al. "Hardware-software integrated approaches to defend against software cache-based side channel attacks". In: *HPCA'09*. 2009.

A. Fuchs et al. "Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs". In: *SYSTOR'15*. 2015

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

H. Raj et al. “Resource Management for Isolation Enhanced Cloud Services”. In: *CCSW’09*. 2009

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

M. Schwarz et al. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: *FC’17*. 2017.

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

H. Raj et al. “Resource Management for Isolation Enhanced Cloud Services”. In: *CCSW’09*. 2009

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

M. Schwarz et al. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: *FC’17*. 2017.

System level: Prevention

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

Y. Zhang et al. "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud". In: *CCS'13*. 2013

H. Raj et al. "Resource Management for Isolation Enhanced Cloud Services". In: *CCSW'09*. 2009

B. C. Vattikonda et al. "Eliminating fine grained timers in Xen". In: *CCSW'11*. 2011

M. Schwarz et al. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: *FC'17*. 2017.

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

H. Raj et al. "Resource Management for Isolation Enhanced Cloud Services". In: *CCSW'09*. 2009

B. C. Vattikonda et al. "Eliminating fine grained timers in Xen". In: *CCSW'11*. 2011

M. Schwarz et al. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: *FC'17*. 2017.

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

H. Raj et al. "Resource Management for Isolation Enhanced Cloud Services". In: *CCSW'09*. 2009

B. C. Vattikonda et al. "Eliminating fine grained timers in Xen". In: *CCSW'11*. 2011

M. Schwarz et al. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: *FC'17*. 2017.

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

H. Raj et al. "Resource Management for Isolation Enhanced Cloud Services". In: *CCSW'09*. 2009

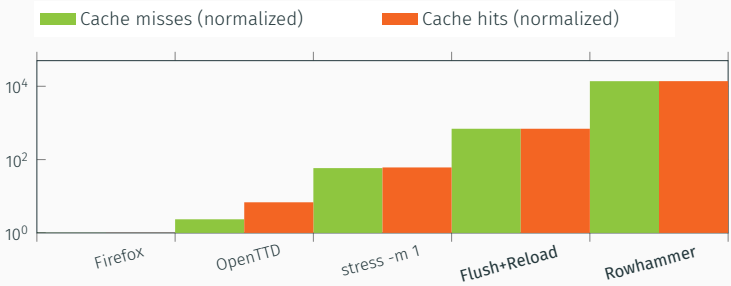
B. C. Vattikonda et al. "Eliminating fine grained timers in Xen". In: *CCSW'11*. 2011

M. Schwarz et al. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: *FC'17*. 2017.

System level: Detect on-going attacks

- using performance counters to monitor cache hits and cache misses

→ risk of false positives



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

D. Gruss et al. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA'16*. 2016

Application level: Detect leakage

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

G. Doychev et al. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *USENIX Security Symposium*. 2013

D. Gruss et al. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015

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

G. Doychev et al. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *USENIX Security Symposium*. 2013

D. Gruss et al. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015

Application level: Write better code

- square-and-multiply-always algorithm
- bit-sliced AES implementation
- hardware implementations (AES-NI, etc.)

Application level: Write better code

- square-and-multiply-always algorithm
- bit-sliced AES implementation
- 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

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

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

`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

---

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

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

O. Acıiçmez et al. “Predicting secret keys via branch prediction”. In: *CT-RSA 2007*. 2007

D. Evtyushkin et al. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. In: *MICRO'16*. 2016

Y. Jang et al. “Breaking kernel address space layout randomization with intel TSX”. In: *CCS'16*. 2016

`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

O. Acıiçmez et al. “Predicting secret keys via branch prediction”. In: *CT-RSA 2007*. 2007

D. Evtyushkin et al. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. In: *MICRO'16*. 2016

Y. Jang et al. “Breaking kernel address space layout randomization with intel TSX”. In: *CCS'16*. 2016

It’s not just caches!

- DRAM
- GPU
- MMU
- TLB

P. Pessl et al. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *USENIX Security Symposium*. 2016.

P. Frigo et al. “Grand Pwning unit: accelerating microarchitectural attacks with the GPU”. In: *S&P 2018*. 2018.

S. van Schaik et al. “Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think”. In: *USENIX Security Symposium*. 2018.

B. Gras et al. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *USENIX Security Symposium*. 2018.

Conclusion

- 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

Conclusion

- more a problem of CPU design than Instruction Set Architecture
- it's also not just the cache
- hard to patch $\rightarrow$ 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 $\rightarrow$ issues linked to performance optimizations
- we would like to keep the optimizations without the attacks

Conclusion

- more a problem of CPU design than Instruction Set Architecture
- it's also not just the cache
- hard to patch $\rightarrow$ 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

Cache side-channel attacks

Lab: Monitoring keystroke timing with no privilege

Clémentine Maurice, CNRS, IRISA

July 13, 2018—Summer School Cyber in Occitanie 2018, Montpellier, France

References i

O. Acıiçmez, J.-P. Seifert, and c. K. Koç. “Predicting secret keys via branch prediction”. In: *CT-RSA 2007*. 2007.

M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. “On subnormal floating point and abnormal timing”. In: *S&P’15*. 2015.

G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *USENIX Security Symposium*. 2013.

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

D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. In: *MICRO’16*. 2016.

P. Frigo, C. Giuffrida, H. Bos, and K. Razavi. “Grand Pwning unit: accelerating microarchitectural attacks with the GPU”. In: *S&P 2018*. 2018.

References ii

A. Fuchs and R. B. Lee. “Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs”. In: *SYSTOR’15*. 2015.

B. Gras, K. Razavi, H. Bos, and C. Giuffrida. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *USENIX Security Symposium*. 2018.

D. Gruss, R. Spreitzer, and S. Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015.

D. Gruss, C. Maurice, K. Wagner, and S. Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA’16*. 2016.

D. Gullasch, E. Bangerter, and S. Krenn. “Cache Games – Bringing Access-Based Cache Attacks on AES to Practice”. In: *S&P’11*. 2011.

N. Herath and A. Fogh. “These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security”. In: *Black Hat 2015 Briefings*. 2015.

G. Irazoqui, T. Eisenbarth, and B. Sunar. “Cross processor cache attacks”. In: *AsiaCCS’16*. 2016.

References iii

Y. Jang, S. Lee, and T. Kim. “Breaking kernel address space layout randomization with intel TSX”. In: *CCS’16*. 2016.

J. Kong, O. Acıiçmez, J.-P. Seifert, and H. Zhou. “Hardware-software integrated approaches to defend against software cache-based side channel attacks”. In: *HPCA’09*. 2009.

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.

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

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.

D. A. Osvik, A. Shamir, and E. Tromer. “Cache Attacks and Countermeasures: the Case of AES”. In: *CT-RSA 2006*. 2006.

C. Percival. “Cache missing for fun and profit”. In: *Proceedings of BSDCan*. 2005.

References iv

P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *USENIX Security Symposium*. 2016.

H. Raj, R. Nathuji, A. Singh, and P. England. “Resource Management for Isolation Enhanced Cloud Services”. In: *CCSW’09*. 2009.

M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: *FC’17*. 2017.

S. van Schaik, K. Razavi, C. Giuffrida, and H. Bos. “Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think”. In: *USENIX Security Symposium*. 2018.

B. C. Vattikonda, S. Das, and H. Shacham. “Eliminating fine grained timers in Xen”. In: *CCSW’11*. 2011.

Z. Wang and R. B. Lee. “New cache designs for thwarting software cache-based side channel attacks”. In: *ACM SIGARCH Computer Architecture News* 35.2 (June 2007), p. 494.

Y. Yarom and K. Falkner. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *USENIX Security Symposium*. 2014.

References v

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

M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. “ARMageddon: Cache Attacks on Mobile Devices”. In: *USENIX Security Symposium*. 2016.