Side-Channel Security

Chapter 1. Introduction

Daniel Gruss

March 6, 2025

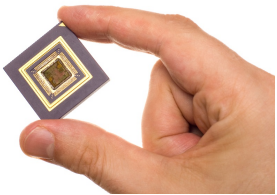
Graz University of Technology

Side-Channel Security

Software

Mobile

Hardware



Team

Daniel Gruss

Rishub Nagpal

Lukas Giner

Roland Czerny

Sudheendra Neela

Rules

- 2 persons per team

→ register via `https://tinyurl.com/mw6exaz7` until Friday March 14, 08:00am

→ include name, email and matriculation number of both team members

Rules

- 2 persons per team

→ register via `https://tinyurl.com/mw6exaz7` until Friday March 14, 08:00am

→ include name, email and matriculation number of both team members

- 2 exercises, each 15 points
- submission via git tag
- points based on exercise interview
- minimum of 4 points per exercise sheet to pass

Grading

- aim for the best or drop out now!
- 26 of 30 points $\rightarrow$ 1

Grading

- aim for the best or drop out now!
- 26 of 30 points $\rightarrow$ 1
- 22 of 30 points $\rightarrow$ 2
- 18 of 30 points $\rightarrow$ 3
- 15 of 30 points $\rightarrow$ 4
- minimum of 4 points per exercise sheet to pass

Second Chance

- Same Task
- Deadline: 1 week after negative exercise interview
- No penalty to reach 50%, then 25% reduction!

Second Chance

- Same Task
- Deadline: 1 week after negative exercise interview
- No penalty to reach 50%, then 25% reduction!
- e.g., 5 points on ex1 $\rightarrow$ missing 10 points on ex2:
  No reduction until 10 points are achieved on ex2, then -25%

Exercises

Ex1: Software Security

- Presentation: Thursday, 06.03.
- Deadline 1: Thursday, 20.03., 08:00am
- Deadline 2: Thursday, 08.05., 08:00am

Exercises

Ex1: Software Security

- Presentation: Thursday, 06.03.
- Deadline 1: Thursday, 20.03., 08:00am
- Deadline 2: Thursday, 08.05., 08:00am

Ex2: Hardware Security

- Presentation: Thursday, 15.05.
- (Preliminary) Deadline: Thursday, 26.06., 08:00am

Exercise 1

up to 15 points from:

- Task 1: Introduction [0.5 P]
- Task 2: Flush+Reload Attack on PIN Entry [3 P]
- Task 3: Covert Channel [5.5 P]
- Task 4: Spectre [3 P]
- Task 5: KASLR is bad, please break it [3 P]

Exercise 1 – Task 1

Both team members:

- Clone your repo, pull from upstream (accessible a few days after team registration, check Discord)
  `https://git.teaching.isec.tugraz.at/scs/scs25/upstream.git`
- Make a histogram (use F+R calibration tool in demo folder)
- Choose a good threshold (the tool will not tell you what is "good")

Exercise 1 – Task 2

- Flush+Reload attack on a PIN entry library
- Library checks each PIN digit and calls 1 of 2 functions
- Recover the key by checking which functions were called

Exercise 1 – Task 3

- Cross-core cache covert channel
- Real/random *binary* data
- Raw capacity, bit error rate → Lukas redrabbyte@Discord
- Speed records: `https://www.isec.tugraz.at/teaching/materials/scs/exercises/ex1/` & Discord

Exercise 1 – Task 4

- Run a Spectre attack on a provided library
- Leak a secret string by exploiting speculative execution
- Be as fast as possible

Exercise 1 – Task 5

- Break KASLR using one of the demonstrated approaches
- Simplest approach: use timing of prefetch instructions
- Bonus Points for using Data Bounce attack (older Intel only)
- Visualize the output of your program
- Use Intel if you can, or ask us about AMD!

Info

- Lecture materials and *exercise hints* at `https://www.isec.tugraz.at/scs/`
- Discord: ISEC, SCS channel

Getting started

Measuring timing leakage

Exploiting timing leakage

CPU caches

Cache attacks

Side channels

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

Side channels

- *safe software* infrastructure $\rightarrow$ no bugs, e.g., Heartbleed
- does not mean safe execution

Side channels

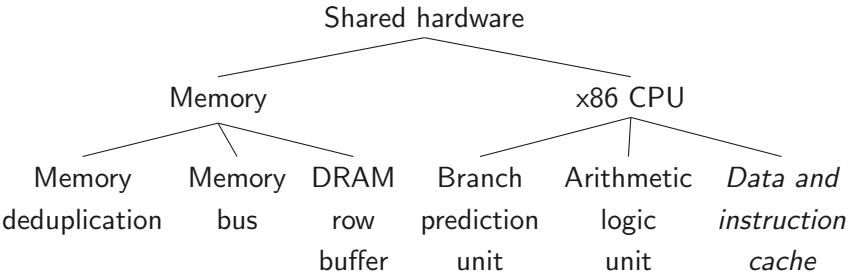
- *safe software* infrastructure → no bugs, e.g., Heartbleed
- does not mean safe execution
- information *leaks* because of the *hardware* it runs on
- no “bug” in the sense of a mistake → lots of performance optimizations

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

Shared hardware



Why targeting the cache?

- shared across cores
- fast

Why targeting the cache?

- shared across cores
- fast

$\rightarrow$ fast cross-core attacks!

Timing differences

- caches improve performance
- SRAM is expensive $\rightarrow$ small caches
- different timings for memory accesses
  - data is *cached* $\rightarrow$ cache hit $\rightarrow$ *fast*
  - data is *not cached* $\rightarrow$ cache miss $\rightarrow$ *slow*

Mesuring timing leakage

How every timing attack works:

- learn timing of different corner cases

Mesuring timing leakage

How every timing attack works:

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

Calibration

```
git clone

https://git.teaching.isec.tugraz.at/scs/scs25/upstream.git

cd library2/demos/calibration/fr
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

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

Step 2. Accurate timings

- do you measure what you *think* you measure?
- *out-of-order* execution $\rightarrow$ 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

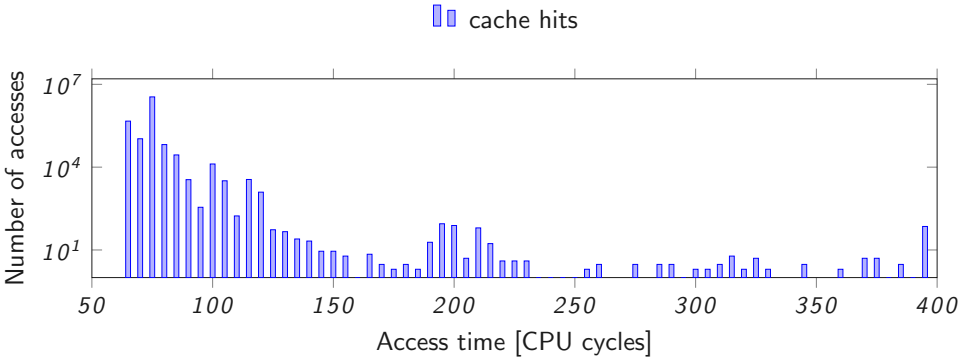
- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

Step 2. Accurate timings

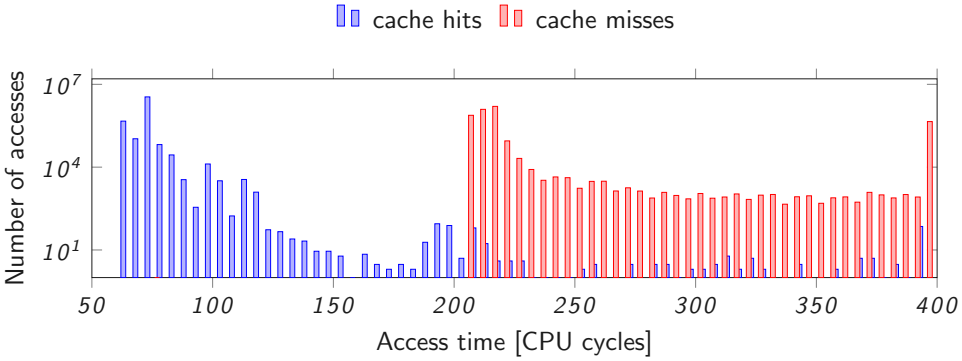
- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

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

Timing differences



Timing differences



Step 4. Find threshold

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

Getting started

Measuring timing leakage

Exploiting timing leakage

CPU caches

Cache attacks

Type of attacks

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

Type of attacks

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

Type of 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

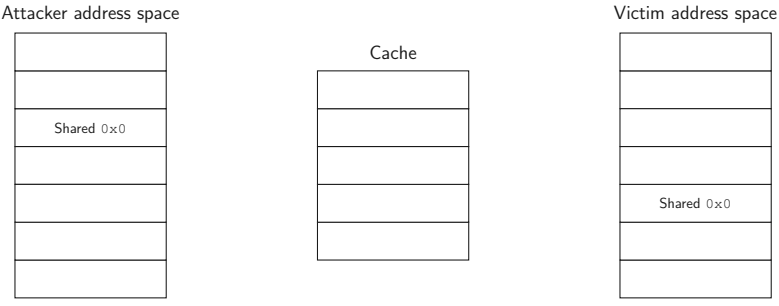
Type of attacks

- 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

Side-channel attack on user input

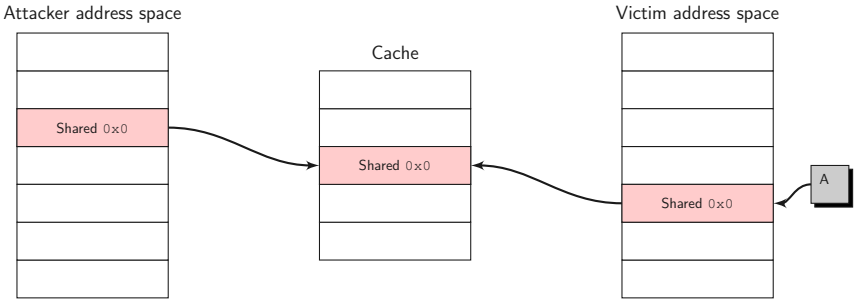
- locate *key-dependent* memory accesses
- with cache template attacks

Profiling Phase: one event



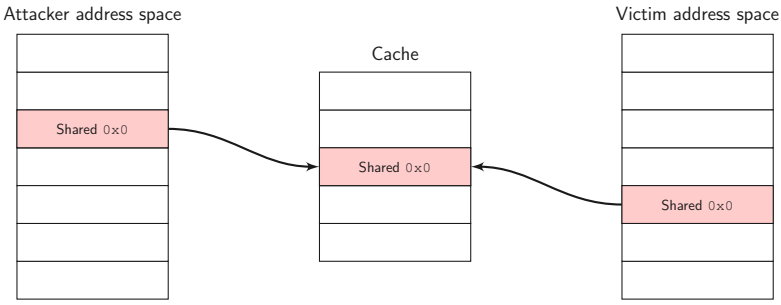
Cache is empty

Profiling Phase: one event



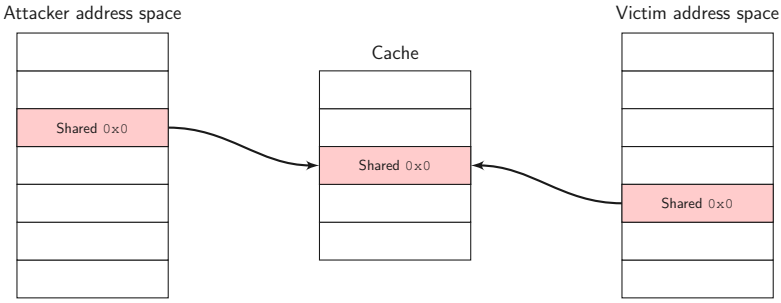
Attacker triggers an event

Profiling Phase: one event



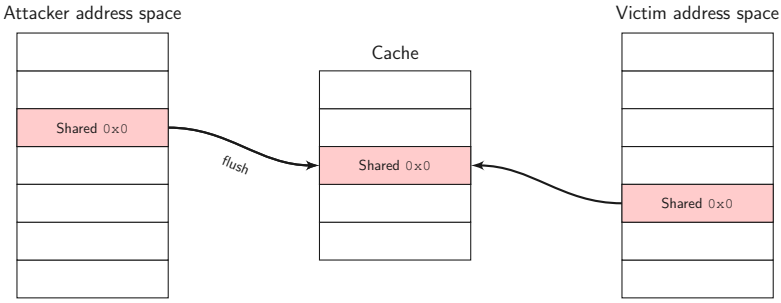
Attacker checks one address for cache hits ("Reload")

Profiling Phase: one event



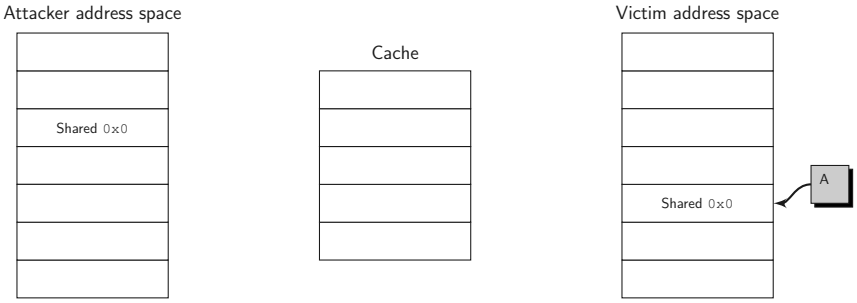
Update number of cache hits per event

Profiling Phase: one event



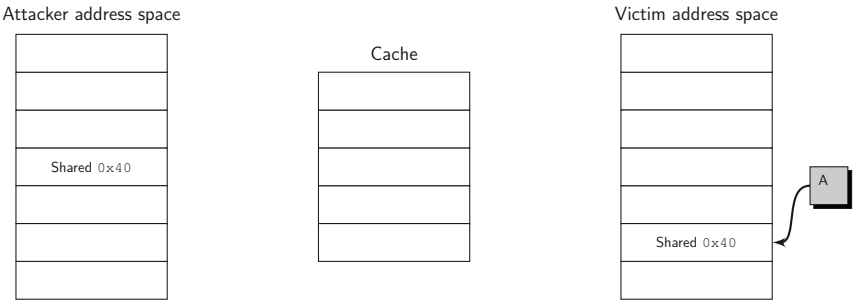
Attacker flushes shared memory

Profiling Phase: one event



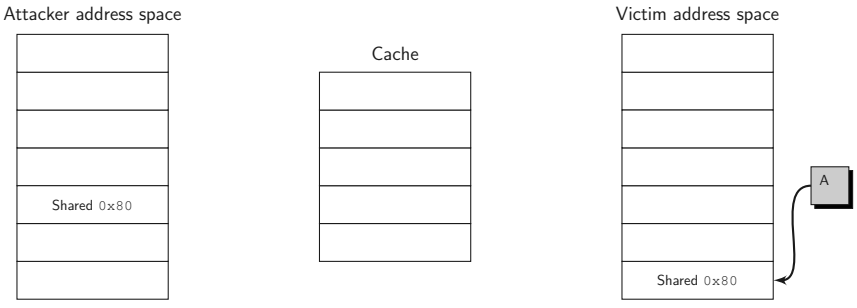
Repeat for higher accuracy

Profiling Phase: one event



Continue with next address

Profiling Phase: one event



Continue with next address

What to profile?

```
# ps -A | grep gedit
# cat /proc/pid/maps
00400000-00489000 r-xp 00000000 08:11 396356
/usr/bin/gedit
7f5a96991000-7f5a96a51000 r-xp 00000000 08:11 399365
/usr/lib/x86_64-linux-gnu/libgdk-3.so.0.1400.14
...
```

memory range, access rights, offset, –, –, file name

Profiling a single event

```
cd cta_examples/profiling/generic_low_frequency_example
# the first parameter is the cache miss threshold
./spy
# start the targeted program
sleep 2; ./spy 200          400000-489000  --    20000
-- -- /usr/bin/gedit
```

... and hold down key in the targeted program

save addresses with peaks!

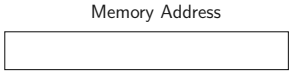
Exploitation phase

```
cd cta_examples/exploitation/generic
./spy threshold file offset
```

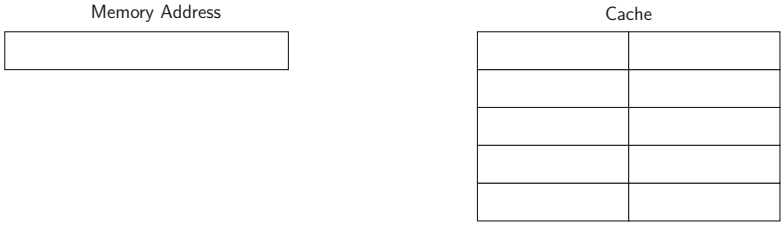
Memory accesses are cached

- Every memory reference goes through the cache
- Transparent to OS and programs

Directly mapped cache



Directly mapped cache



Directly mapped cache

Memory Address

Cache

Tag	Data

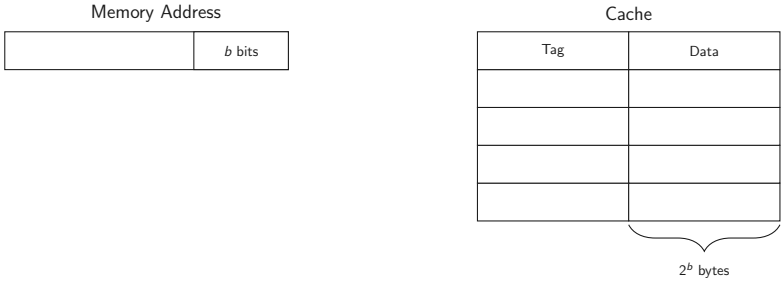
Directly mapped cache

Memory Address

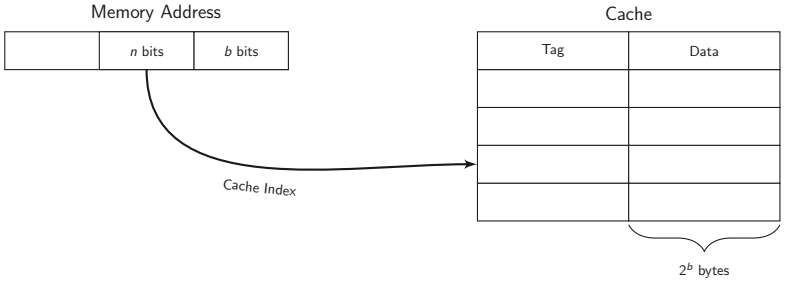
Cache

Tag	Data

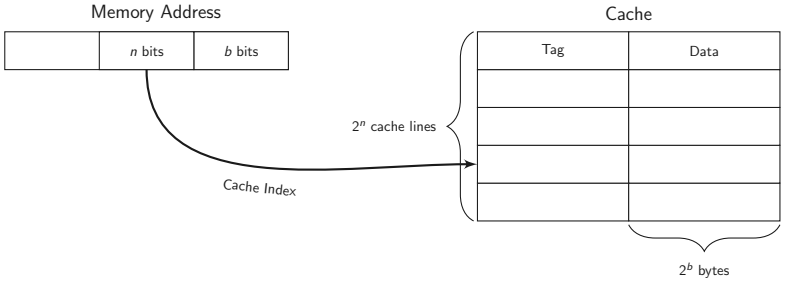
Directly mapped cache



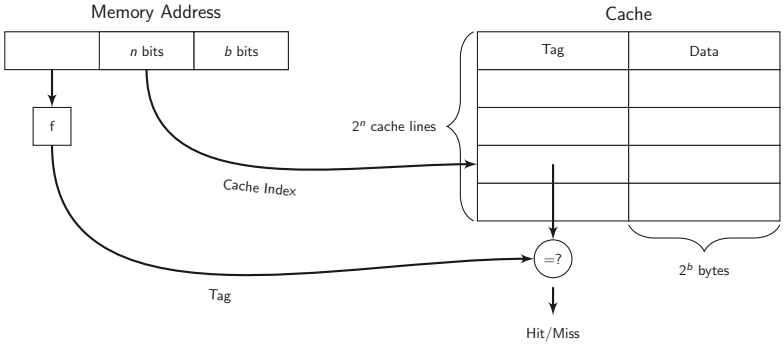
Directly mapped cache



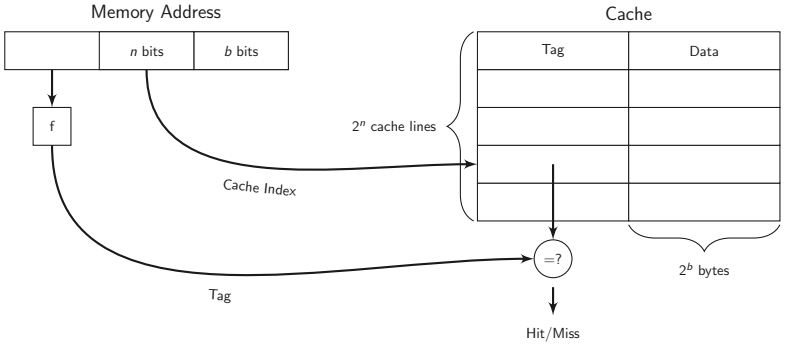
Directly mapped cache



Directly mapped cache

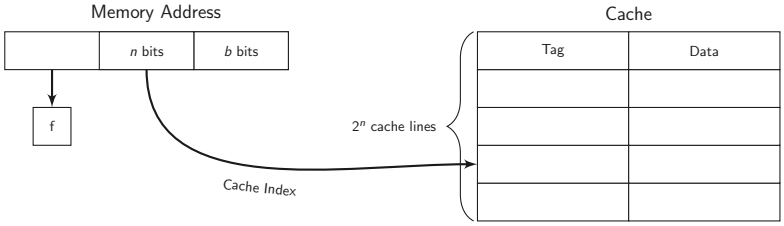


Directly mapped cache

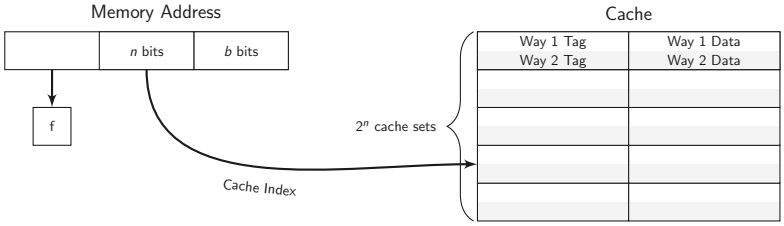


Problem: working on congruent addresses

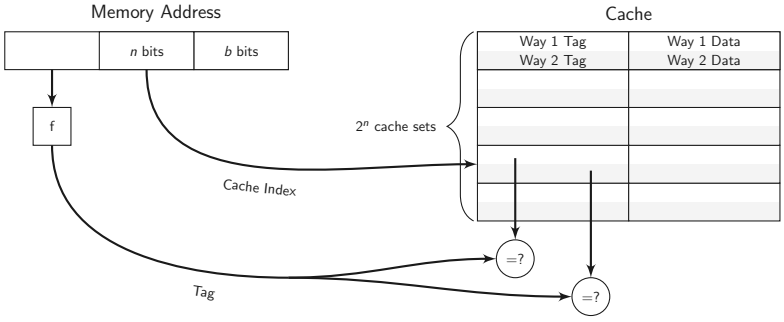
2-way set associativity



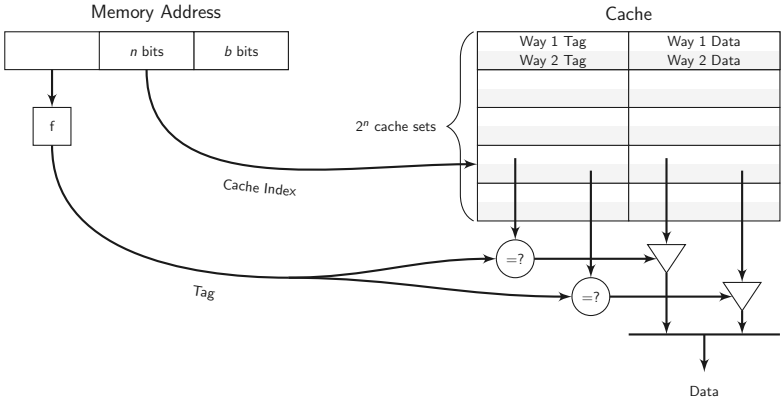
2-way set associativity



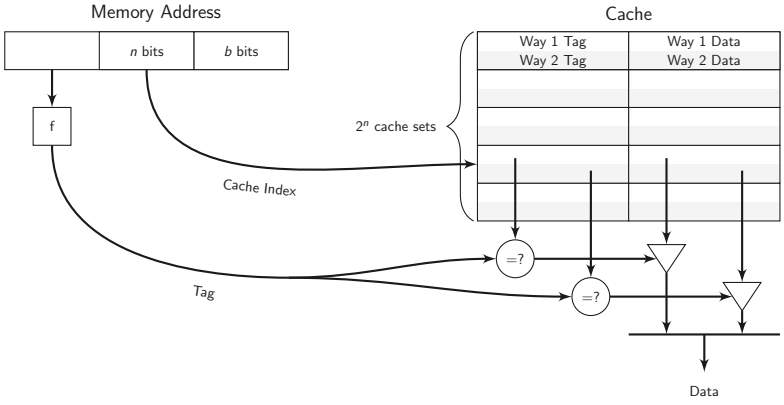
2-way set associativity



2-way set associativity

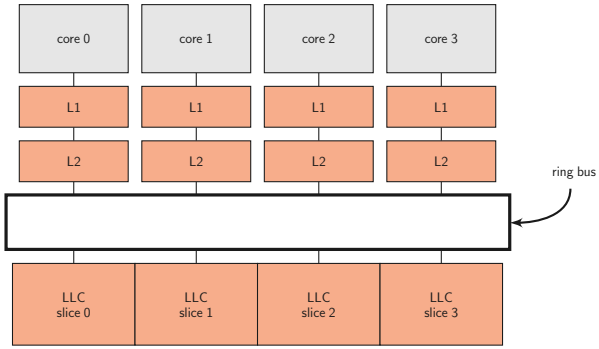


2-way set associativity



$\rightarrow$ replacement policy

Caches today



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

Cache levels: Latency comparison

On current Intel CPUs:

Cache levels: Latency comparison

On current Intel CPUs:

- L1 cache: 4 cycles

Cache levels: Latency comparison

On current Intel CPUs:

- L1 cache: 4 cycles
- L2 cache: 12 cycles

Cache levels: Latency comparison

On current Intel CPUs:

- L1 cache: 4 cycles
- L2 cache: 12 cycles
- L3 cache: 26-31 cycles

Cache levels: Latency comparison

On current Intel CPUs:

- L1 cache: 4 cycles
- L2 cache: 12 cycles
- L3 cache: 26-31 cycles
- DRAM memory: $>$120 cycles

(Unprivileged) cache maintainance

User programs can optimize cache usage:

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

... based on virtual addresses

Getting started

Measuring timing leakage

Exploiting timing leakage

CPU caches

Cache attacks

CPU cache attacks

- cache-based keylogging
- crypto key recovery
  - various implementations (AES, RSA, ECC, ...)
  - up to 97% key bits recovered after 1 encryption
- cross-VM, cross-core, even cross-CPU
- any CPU vendor

Cross-core attacks?

- using the *inclusive* property

Cross-core attacks?

- using the *inclusive* property
- last-level cache is a superset of L1 and L2

Cross-core attacks?

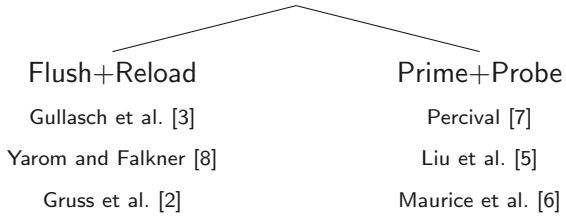
- using the *inclusive* property
- last-level cache is a superset of L1 and L2
- data evicted from last-level cache $\rightarrow$ evicted from L1 and L2

Cross-core attacks?

- using the *inclusive* property
- last-level cache is a superset of L1 and L2
- data evicted from last-level cache $\rightarrow$ evicted from L1 and L2
- a core can evict lines in the private L1 of another core

Access-driven attacks

Attacker monitors *its own activity* to find sets accessed by victim.



Same techniques for covert and side channels

Flush+Reload: Building Blocks

- Shared Library / load binary twice / page deduplication

Flush+Reload: Building Blocks

- Shared Library / load binary twice / page deduplication
- `clflush` throws data out of cache

$\rightarrow$ We can throw other shared code out of the cache

Flush+Reload: Building Blocks

- Shared Library / load binary twice / page deduplication
- `clflush` throws data out of cache

→ We can throw other shared code out of the cache

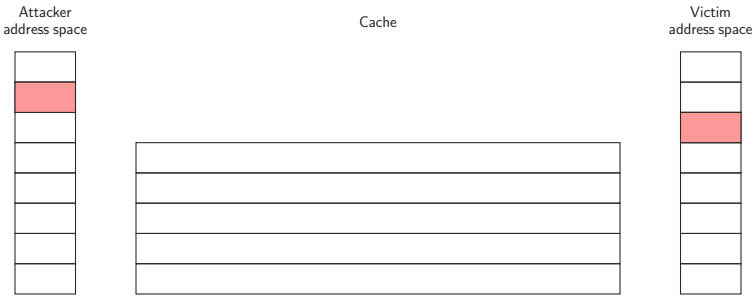
- `rdtsc` / `rdtscp` give accurate timing information

→ We can measure whether shared code is in the cache

Flush+Reload: First steps

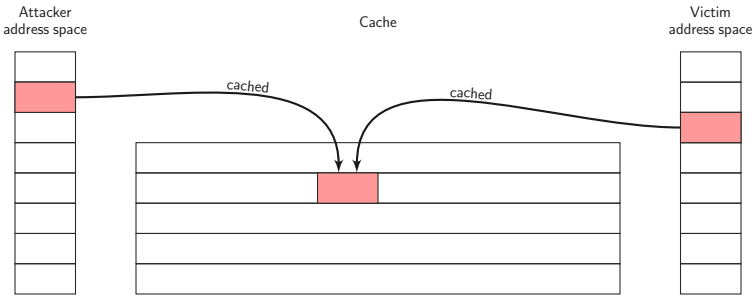
- Measure timing of cached memory
- Measure timing of non-cached memory (flush before measuring)
- Draw a histogram

Flush+Reload



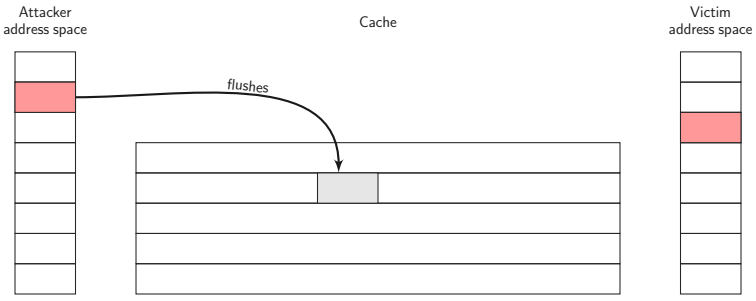
step 0: attacker maps shared library → shared memory, shared in cache

Flush+Reload



step 0: attacker maps shared library → shared memory, shared in cache

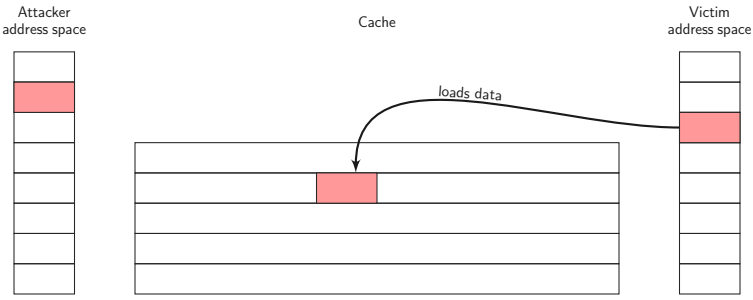
Flush+Reload



step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line

Flush+Reload

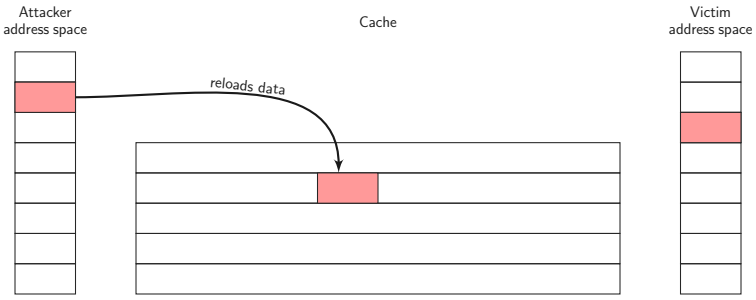


step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line

step 2: victim loads data while performing encryption

Flush+Reload



step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line

step 2: victim loads data while performing encryption

step 3: attacker reloads data → fast access if the victim loaded the line

Flush+Reload

Pros: fine granularity (1 line)

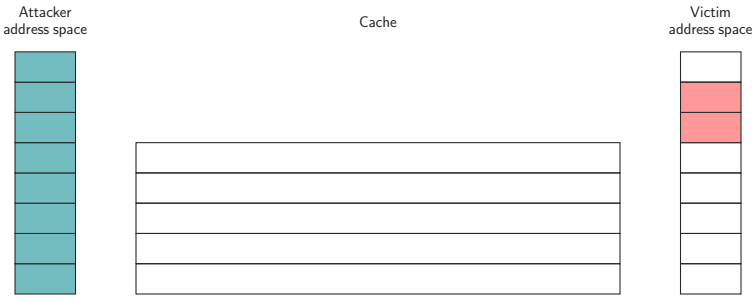
Cons: restrictive

1. needs `clflush` instruction (not available e.g., in JS)
2. needs shared memory

Variants of Flush+Reload

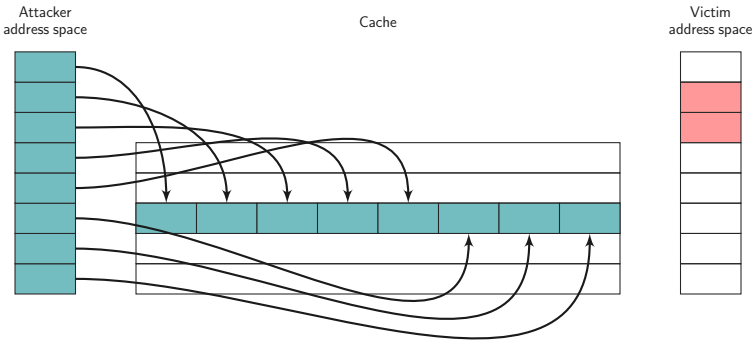
- Flush+Flush [1]
- Evict+Reload [2] on ARM [4]

Prime+Probe



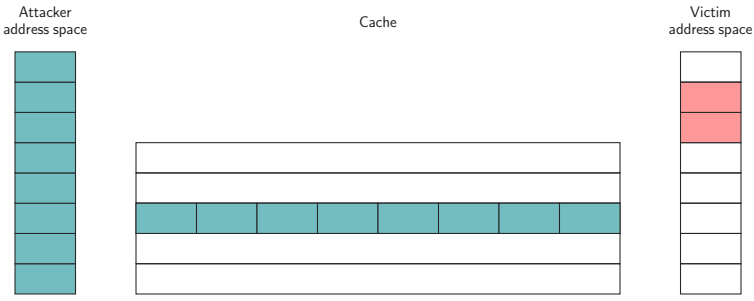
step 0: attacker fills the cache (prime)

Prime+Probe



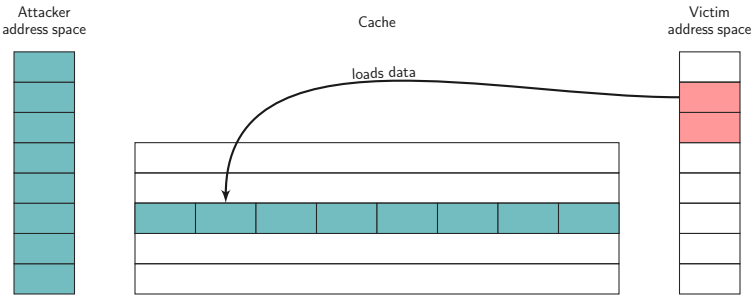
step 0: attacker fills the cache (prime)

Prime+Probe



step 0: attacker fills the cache (prime)

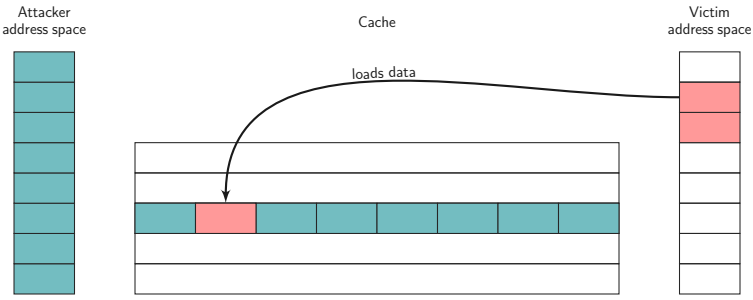
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

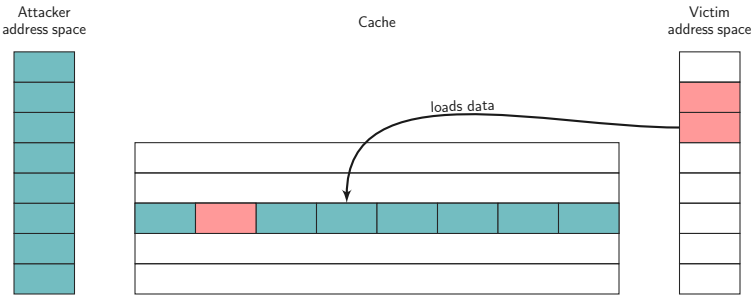
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

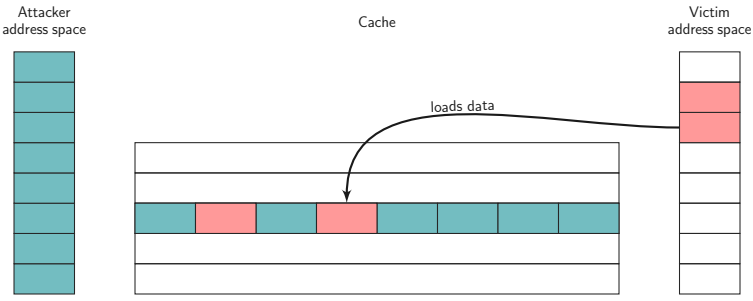
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

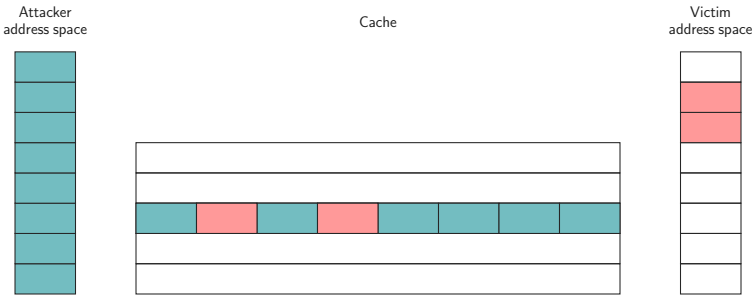
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

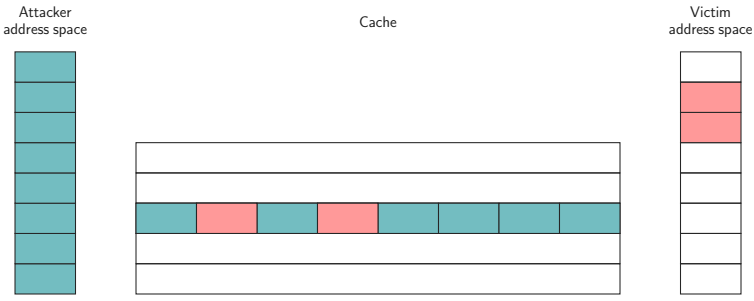
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

Prime+Probe

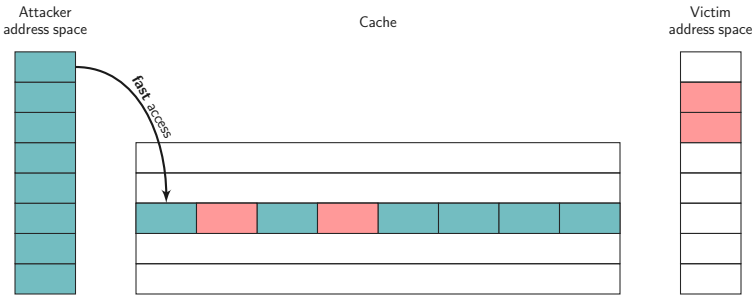


step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

step 2: attacker probes data to determine if the set was accessed

Prime+Probe

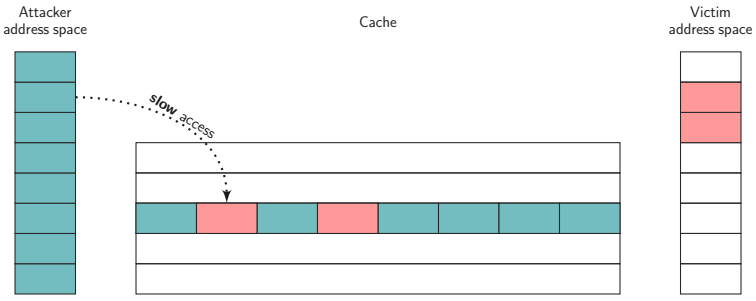


step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

step 2: attacker probes data to determine if the set was accessed

Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

step 2: attacker probes data to determine if the set was accessed

Prime+Probe

Pros: less restrictive

1. no need for `clflush` instruction (not available e.g., in JS)
2. no need for shared memory

Cons: coarser granularity (1 set)

Issues 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?

Side-Channel Security

Chapter 1. Introduction

Daniel Gruss

March 6, 2025

Graz University of Technology

[1] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. (2016). Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA'16*.

[2] Gruss, D., Spreitzer, R., and Mangard, S. (2015). Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium (USENIX Security'15)*.

[3] Gullasch, D., Bangerter, E., and Krenn, S. (2011). Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy (S&P'11)*.

[4] Lipp, M., Gruss, D., Spreitzer, R., and Mangard, S. (2015). ARMageddon: Last-Level Cache Attacks on Mobile Devices. *ArXiv e-prints*.

[5] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. (2015). Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy (S&P'15)*.

[6] Maurice, C., Neumann, C., Heen, O., and Francillon, A. (2015). C5: Cross-Cores Cache Covert Channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'15)*.

[7] Percival, C. (2005). Cache Missing for Fun and Profit. URL: `http://daemonology.net/hyperthreading-considered-harmful/`.

[8] Yarom, Y. and Falkner, K. (2014). FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium (USENIX Security'14)*.