Side-Channel Lab I

Michael Schwarz

Security Week Graz 2019

Scope

Scope

Scope

Scope

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

Scope

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

Side channels

- **safe software** infrastructure → no bugs, e.g., Heartbleed

Side channels

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

Side channels

- **safe software** infrastructure → no bugs, e.g., Heartbleed
- does not mean safe execution
- information **leaks** because of the **hardware** it runs on

Side channels

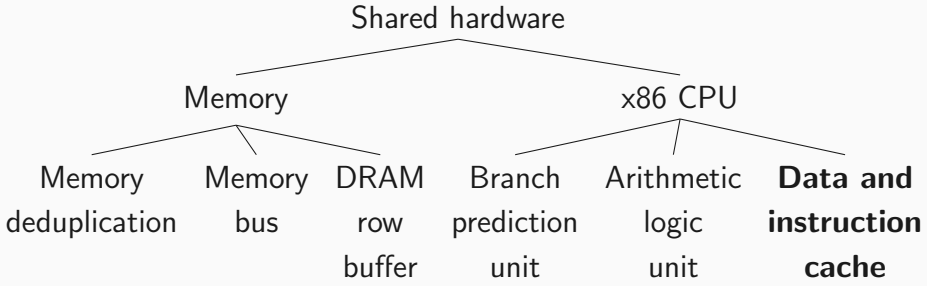
- **safe software** infrastructure → no bugs, e.g., Heartbleed
- does not mean safe execution
- information **leaks** because of the **hardware** it runs on
- no "bug" in the sense of a mistake → lots of performance optimizations

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

Why targeting the cache?

- shared across cores
- fast

Why targeting the cache?

- shared across cores
- fast

→ fast cross-core attacks!

Timing differences

- caches improve performance

Timing differences

- caches improve performance
- SRAM is expensive → small caches

Timing differences

- caches improve performance
- SRAM is expensive → small caches
- different timings for memory accesses

Timing differences

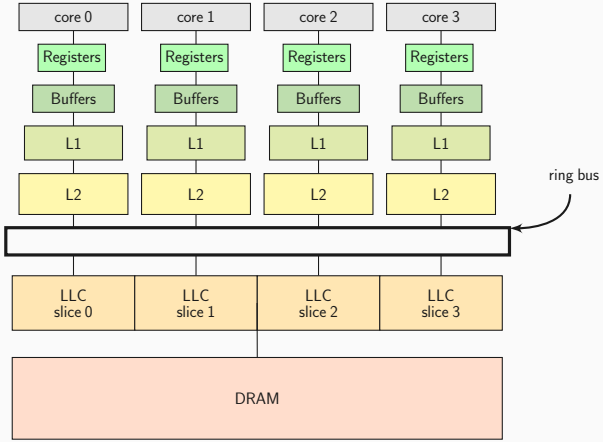
- caches improve performance
- SRAM is expensive → small caches
- different timings for memory accesses
  - data is **cached** → cache hit → **fast**

Timing differences

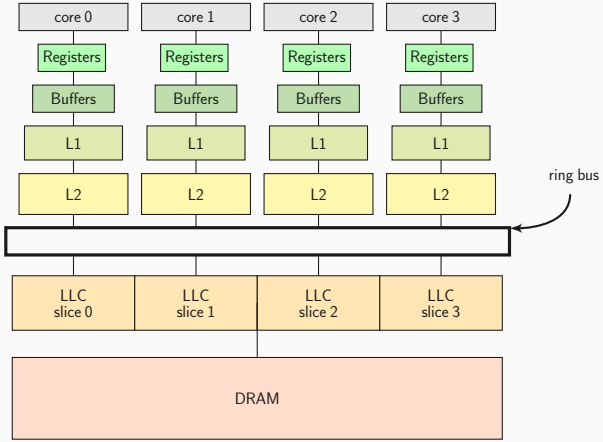
- caches improve performance
- SRAM is expensive → small caches
- different timings for memory accesses
  - data is **cached** → cache hit → **fast**
  - data is **not cached** → cache miss → **slow**

Memory Hierarchy

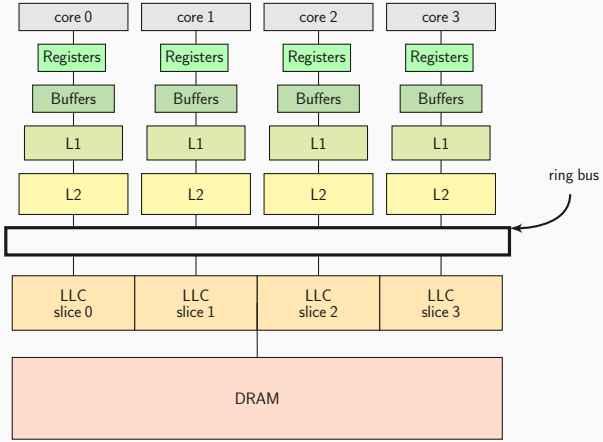
Memory Hierarchy



Memory Hierarchy

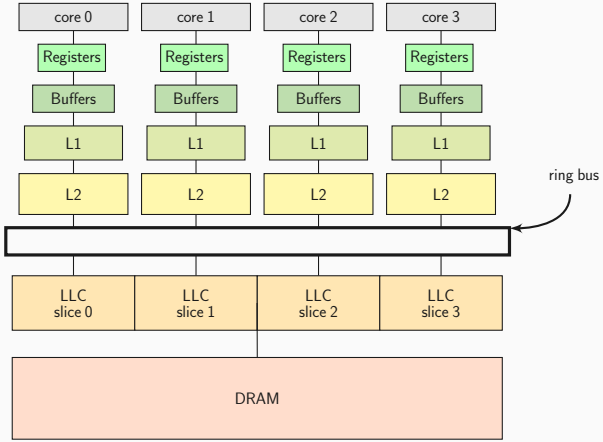


Memory Hierarchy



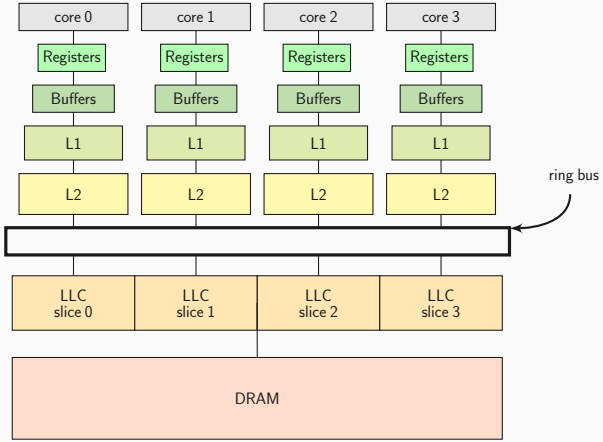
- L1 and L2 are private

Memory Hierarchy



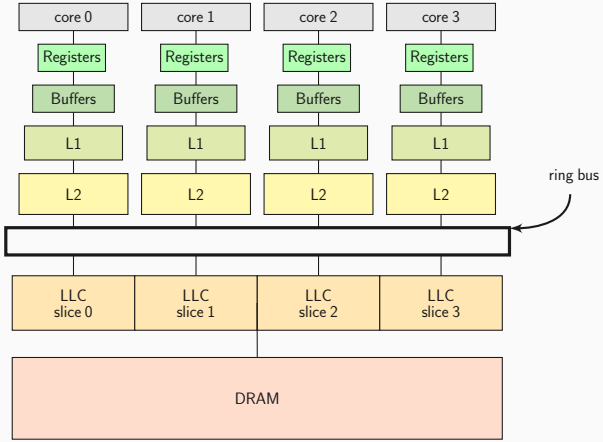
- L1 and L2 are private
- last-level cache:

Memory Hierarchy



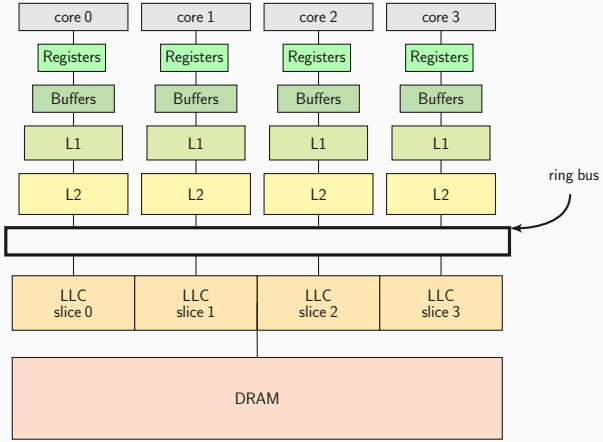
- L1 and L2 are private
- last-level cache:
  - divided in **slices**

Memory Hierarchy



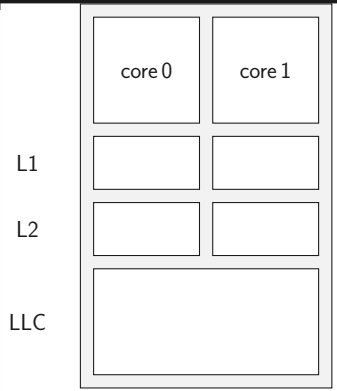
- L1 and L2 are private
- last-level cache:
  - divided in **slices**
  - **shared** across cores

Memory Hierarchy



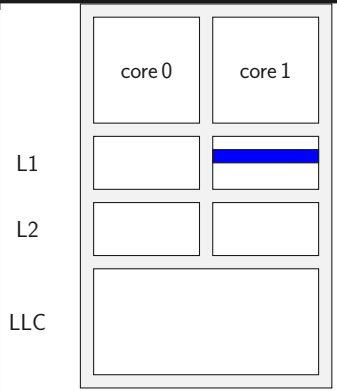
- L1 and L2 are private
- last-level cache:
  - divided in **slices**
  - **shared** across cores
  - **inclusive**

Inclusive property



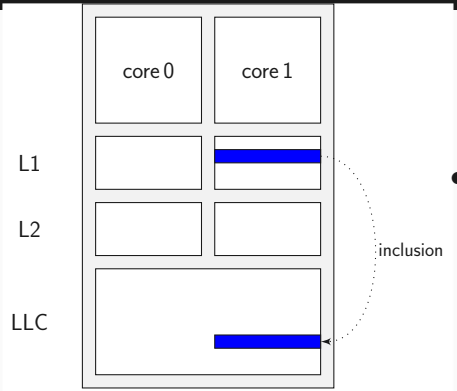
- **inclusive** LLC: superset of L1 and L2

Inclusive property



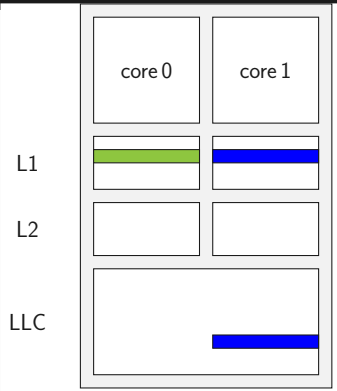
- **inclusive** LLC: superset of L1 and L2

Inclusive property



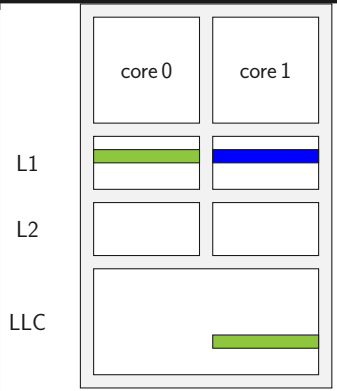
- **inclusive** LLC: superset of L1 and L2

Inclusive property



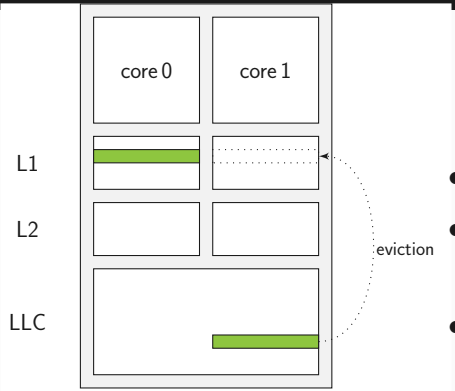
- **inclusive** LLC: superset of L1 and L2

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**

Latency comparison

On current Intel CPUs:

- Registers: 0-1 cycle

Latency comparison

On current Intel CPUs:

- Registers: 0-1 cycle
- L1 cache: 4 cycles

Latency comparison

On current Intel CPUs:

- Registers: 0-1 cycle
- L1 cache: 4 cycles
- L2 cache: 12 cycles

Latency comparison

On current Intel CPUs:

- Registers: 0-1 cycle
- L1 cache: 4 cycles
- L2 cache: 12 cycles
- L3 cache: 26-31 cycles

Latency comparison

On current Intel CPUs:

- Registers: 0-1 cycle
- L1 cache: 4 cycles
- L2 cache: 12 cycles
- L3 cache: 26-31 cycles
- DRAM memory: >120 cycles

Measuring timing leakage

How every timing attack works:

- learn timing of different corner cases

Measuring timing leakage

How every timing attack works:

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

Steps

1. build two cases: cache hits and cache misses

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

Cache hits

Loop:

1. measure time

Cache hits

Loop:

1. measure time
2. access variable (always cache **hit**)

Cache hits

Loop:

1. measure time
2. access variable (always cache **hit**)
3. measure time

Cache hits

Loop:

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

Cache misses

Loop:

1. measure time

Cache misses

Loop:

1. measure time
2. access variable (always cache **miss**)

Cache misses

Loop:

1. measure time
2. access variable (always cache **miss**)
3. measure time

Cache misses

Loop:

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

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)

Time to code

Accurate timings

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

Accurate timings

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

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

Accurate timings

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

Accurate timings

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

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

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

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

Accurate timings

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

Accurate timings

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

Accurate timings

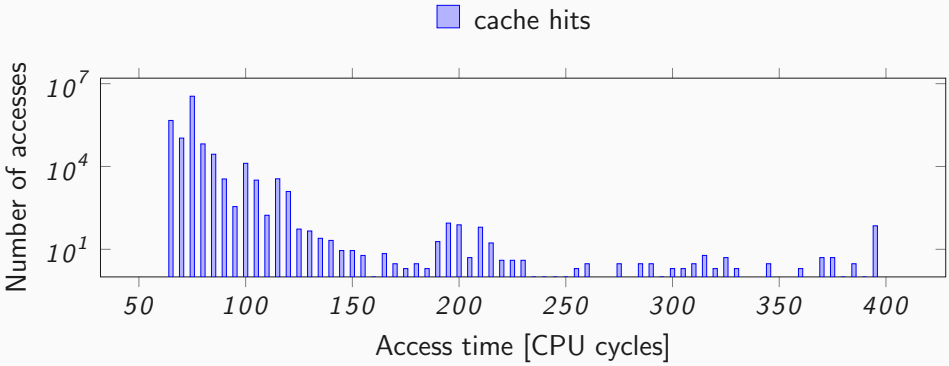
- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

Accurate timings

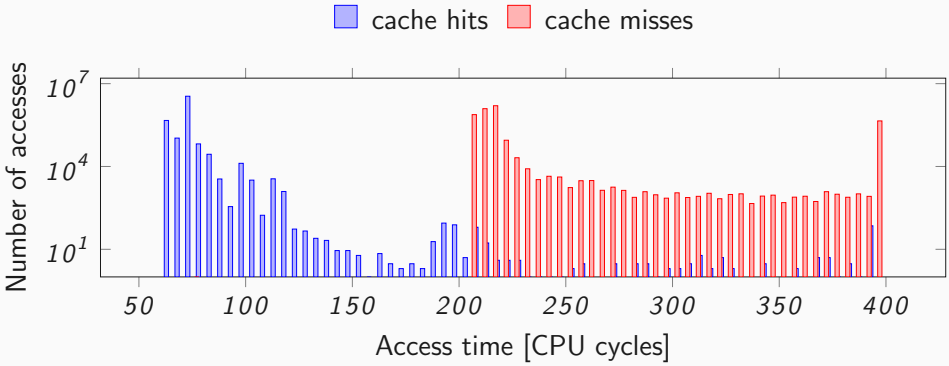
- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

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



Find threshold

- as high as possible

Find threshold

- as high as possible
- most cache hits are below

Find threshold

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

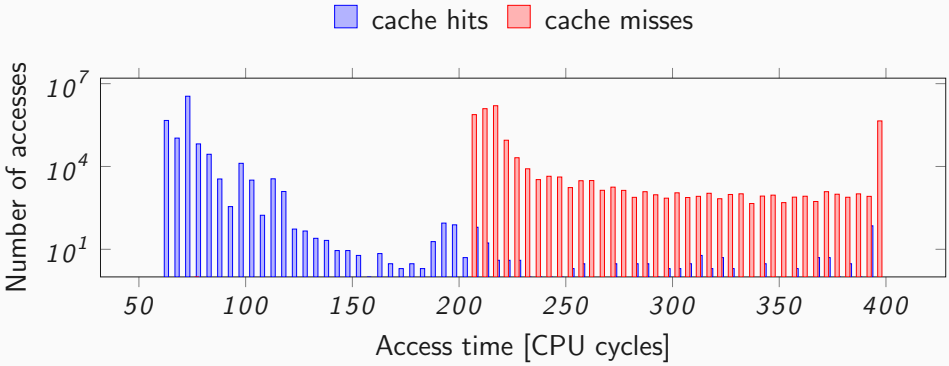
Hit vs. Miss?

- Hit → Data is fetched from buffers, L1, L2, or L3

Hit vs. Miss?

- Hit → Data is fetched from buffers, L1, L2, or L3
- Miss → Data is fetched from DRAM

From Histogram to Attack



Type of attacks

- cache attacks → exploit timing differences of memory accesses

Type of attacks

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

Type of 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

Type of 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

Type of 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

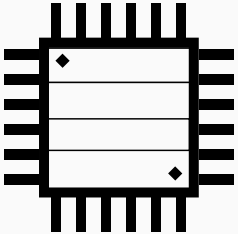
Type of attacks

- 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

Shared Memory

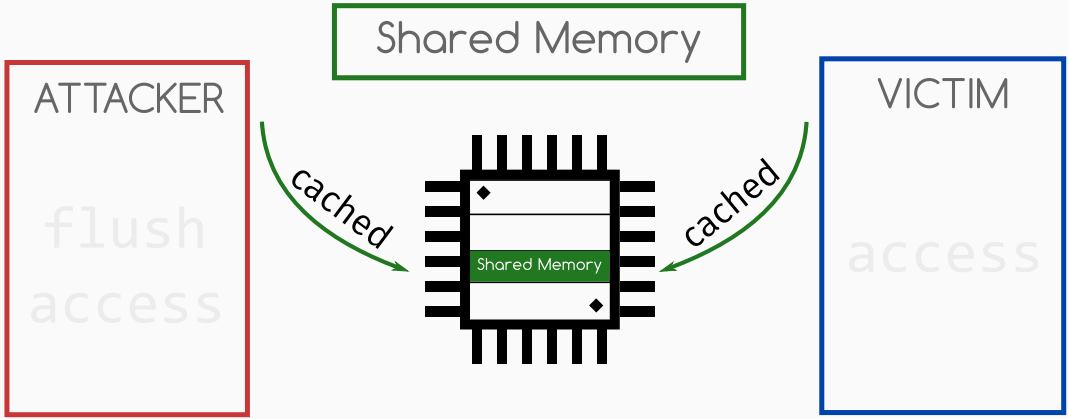
ATTACKER

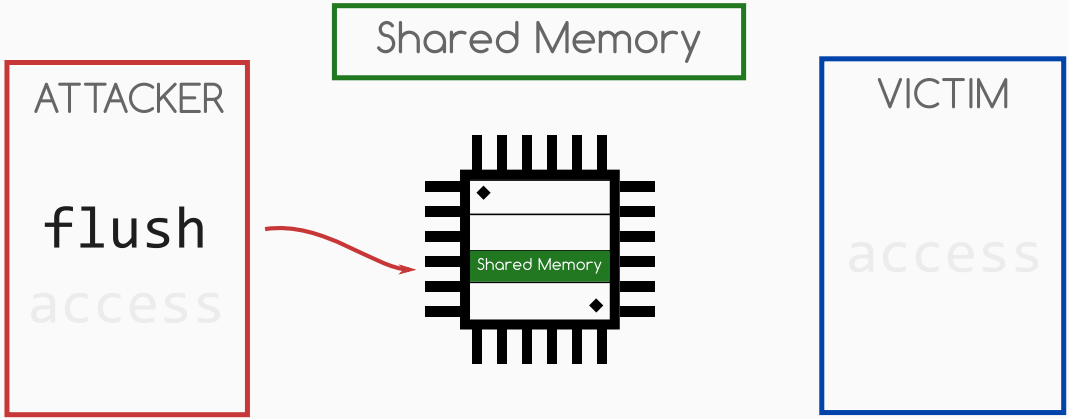
flush
access

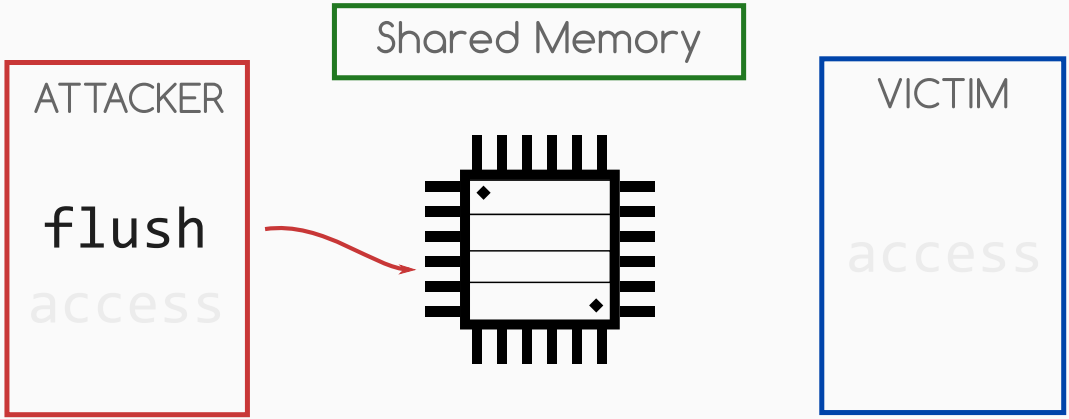


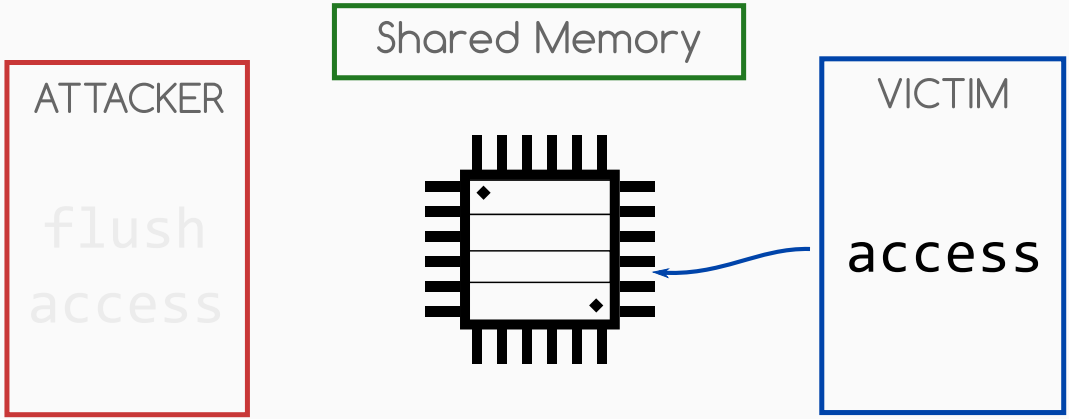
VICTIM

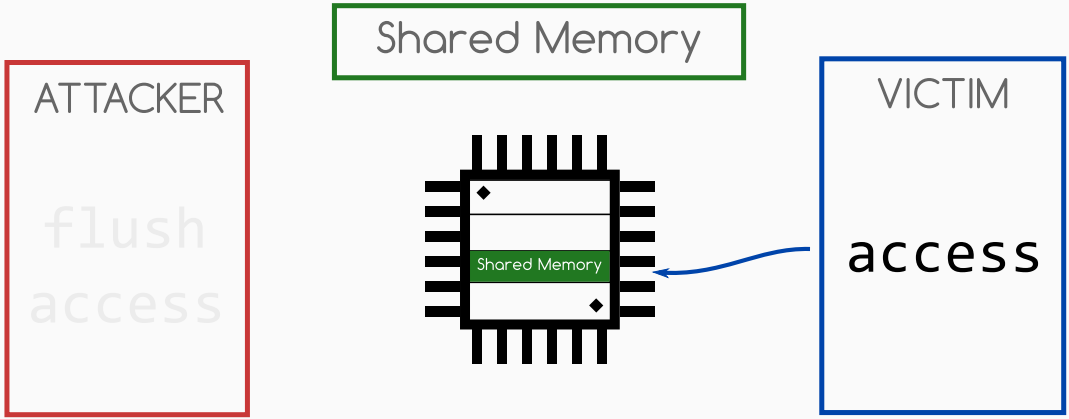
access

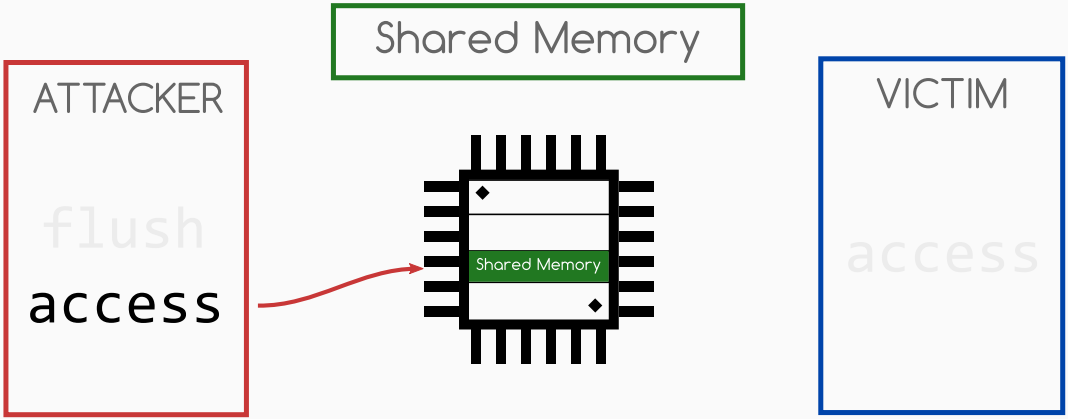

Shared Memory
ATTACKER
flush
access
cached
Shared Memory
cached
VICTIM
access

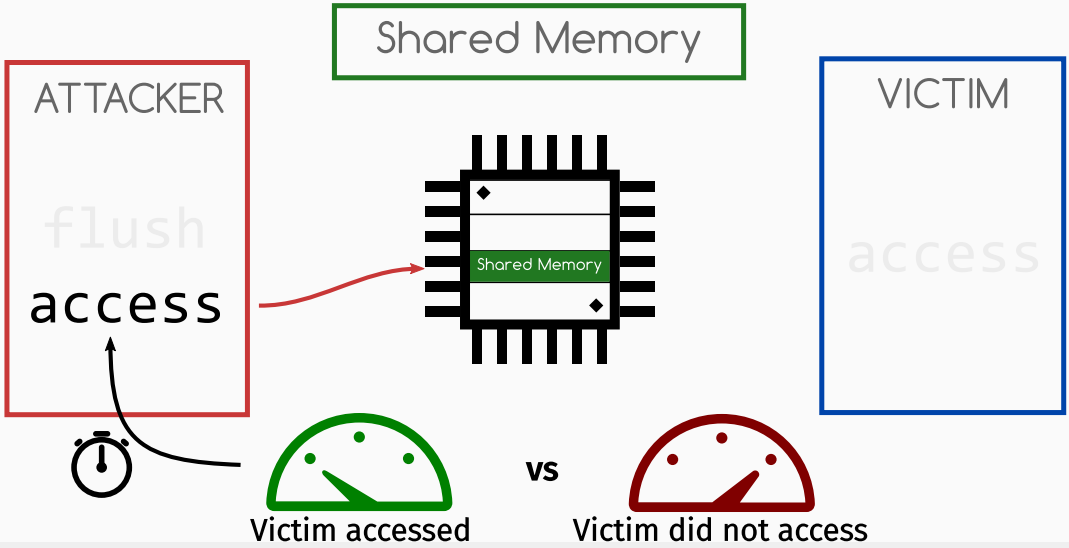

Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access


Shared Memory
ATTACKER
flush
access
VICTIM
access


Shared Memory
ATTACKER
flush
access
VICTIM
access

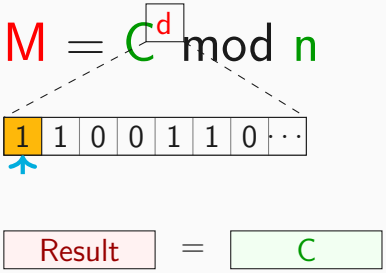

Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access


Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access


Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access
vs
Victim accessed
Victim did not access

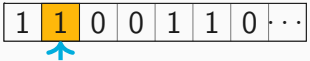
Signatures (RSA)

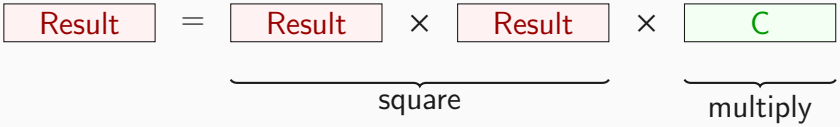
$$M = C^d \bmod n$$


M = C^d mod n
1 1 0 0 1 1 0 ···
Result = C

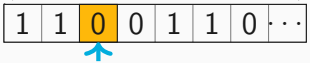
Signatures (RSA)

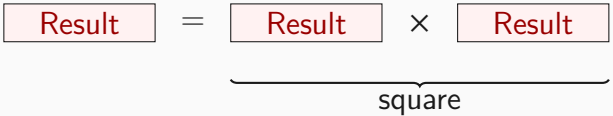
$$M = C^{d} \bmod n$$





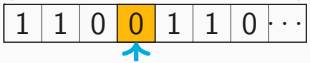
$$M = C^{d} \bmod n$$

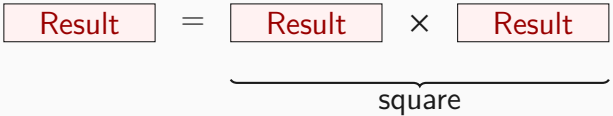




Signatures (RSA)

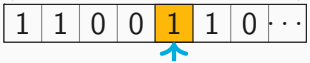
$$M = C^{d} \bmod n$$

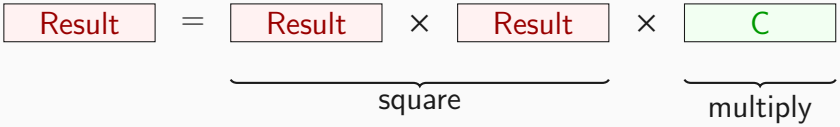




Signatures (RSA)

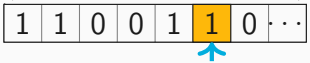
$$M = C^{d} \bmod n$$

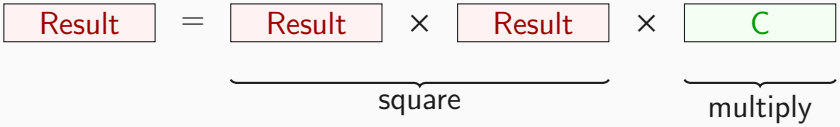


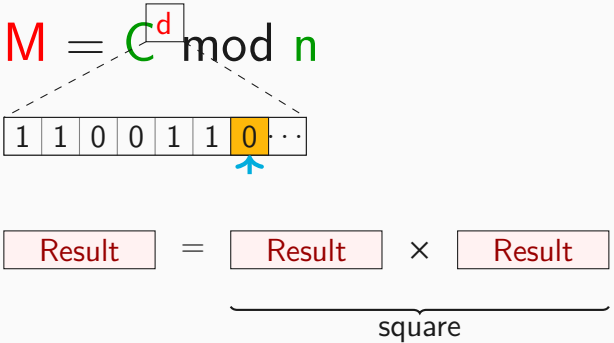


Signatures (RSA)

$$M = C^{d} \bmod n$$






M = C^d mod n
1 1 0 0 1 1 0 ···
Result = Result × Result
square

Time to code

Side-channel attack on user input

- locate **key-dependent** memory accesses

Side-channel attack on user input

- locate **key-dependent** memory accesses
- How to locate key-dependent memory accesses?

Challenges

- It's complicated:

Challenges

- It's complicated:
  - Large binaries and libraries (third-party code)

Challenges

- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)

Challenges

- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries

Challenges

- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries
  - Self-compiled binaries

Challenges

- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries
  - Self-compiled binaries
- Difficult to find **all** exploitable addresses

Cache Template Attacks

Profiling Phase

- Preprocessing step to find exploitable addresses automatically

Exploitation Phase

Cache Template Attacks

Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)

Exploitation Phase

Cache Template Attacks

Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - called “Cache Template”

Exploitation Phase

Cache Template Attacks

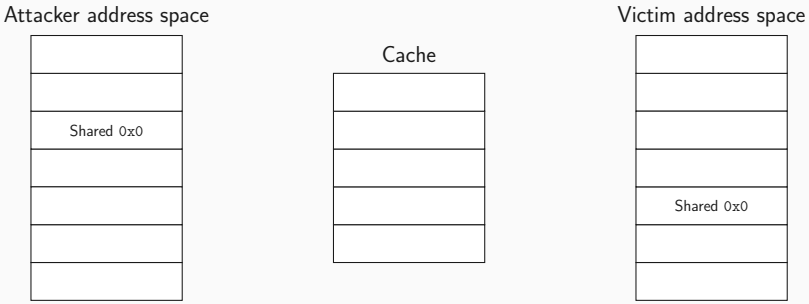
Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - called “Cache Template”

Exploitation Phase

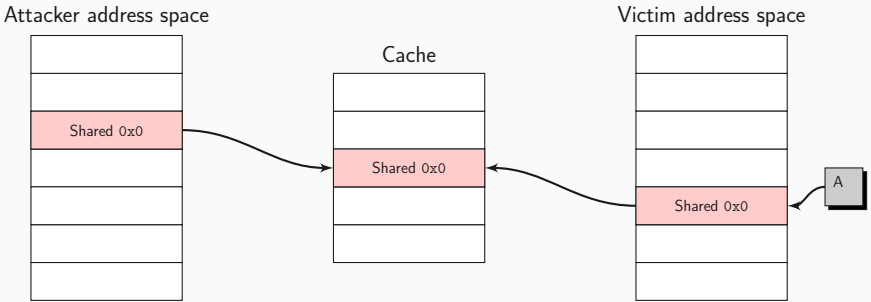
- Monitor exploitable addresses

Profiling Phase



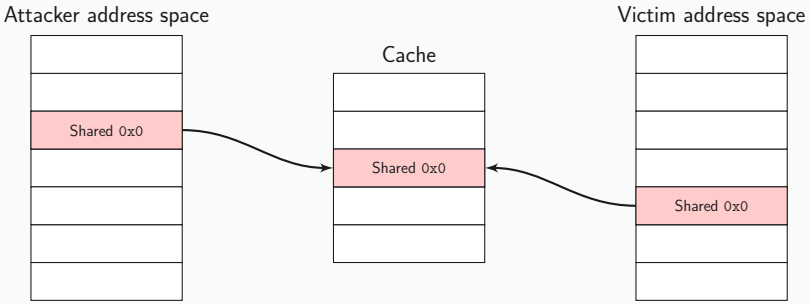
Cache is empty

Profiling Phase



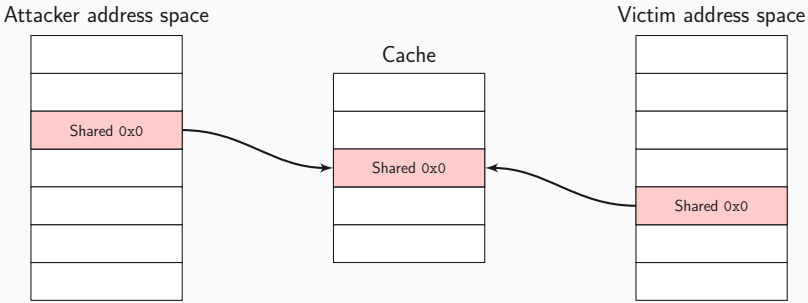
Attacker triggers an event

Profiling Phase



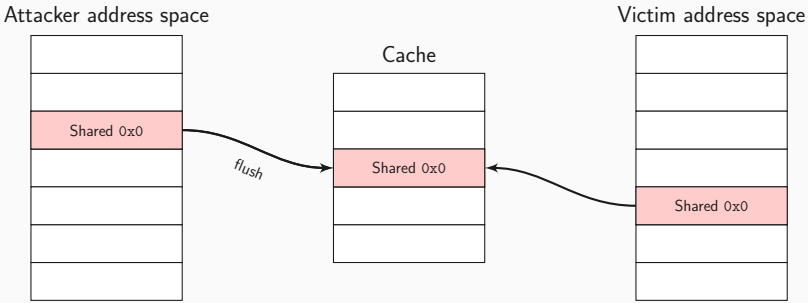
Attacker checks one address for cache hits ("Reload")

Profiling Phase



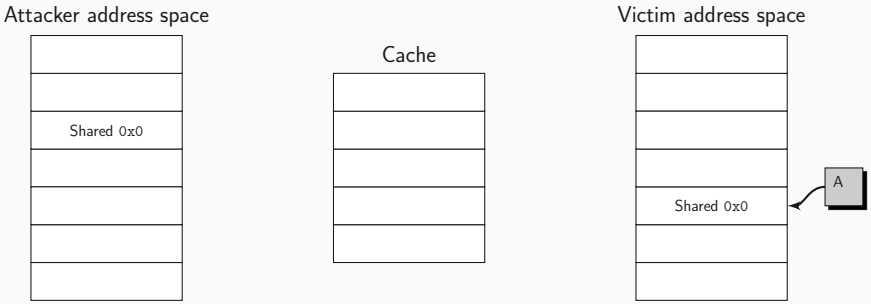
Update number of cache hits per event

Profiling Phase



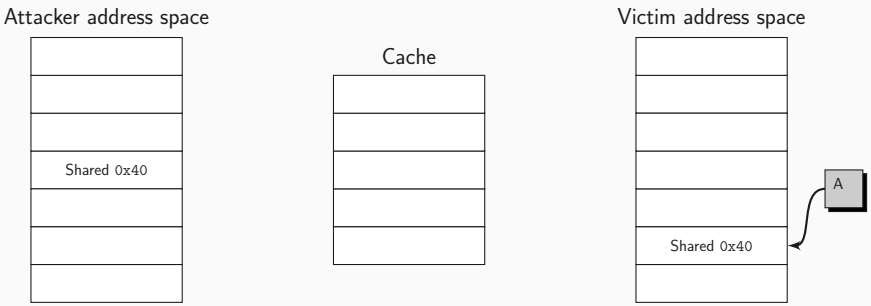
Attacker flushes shared memory

Profiling Phase



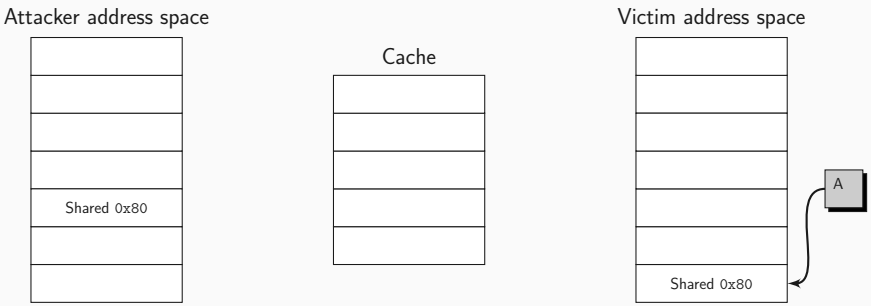
Repeat for higher accuracy

Profiling Phase



Continue with next address

Profiling Phase



Continue with next address

What to profile?


```
$> ps -A | grep gedit
$> cat /proc/<pid>/maps
00400000-00489000 r-xp 00000000 fd:01 396356
/usr/bin/gedit
7f5a96991000-7f5a96a51000 r-xp 00000000 fd:01 399365
/usr/lib/x86_64-linux-gnu/libgdk-3.so.0.2200.30
...
```

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

Profiling a single event


```
$> cd practicals/02_cache_template_attacks/
$> make
$> # start the targeted program (e.g., gedit)
$> sleep 2; ./profiling /usr/lib/x86_64-linux-gnu/
   libgdk-3.so.0.2200.30
```

... and hold down a key in the target program

Profiling a single event


```
$> cd practicals/02_cache_template_attacks/
$> make
$> # start the targeted program (e.g., gedit)
$> sleep 2; ./profiling /usr/lib/x86_64-linux-gnu/
   libgdk-3.so.0.2200.30
```

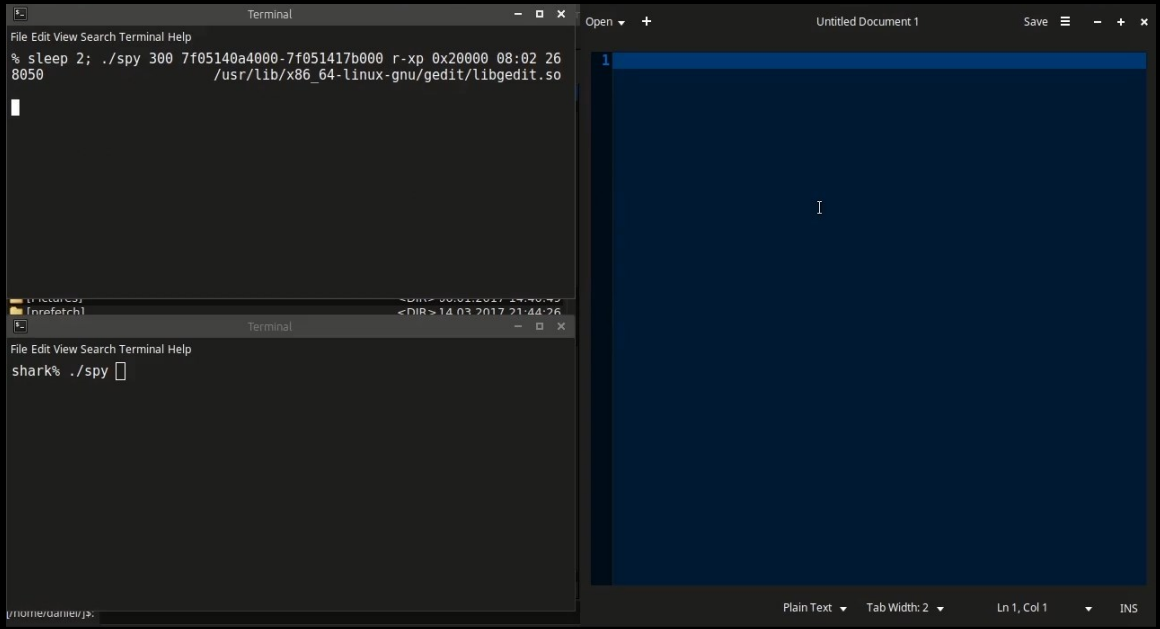
... and hold down a key in the target program

save addresses with peaks!

Exploitation phase

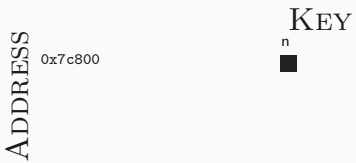

```
$> # ./spy <file> <offset>
$> ./spy /usr/lib/x86_64-linux-gnu/libgdk-3.so.0.2200.30 336896
Monitoring offset 336896
Hit #0
Hit #1
Hit #2
...
```

Time to code

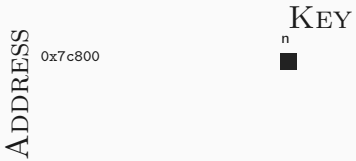


Cache Template Attack Demo

Profiling Phase: 1 Event, 1 Address



Profiling Phase: 1 Event, 1 Address



Example: Cache Hit Ratio for (`0x7c800`, `n`): 200 / 200

Profiling Phase: All Events, 1 Address

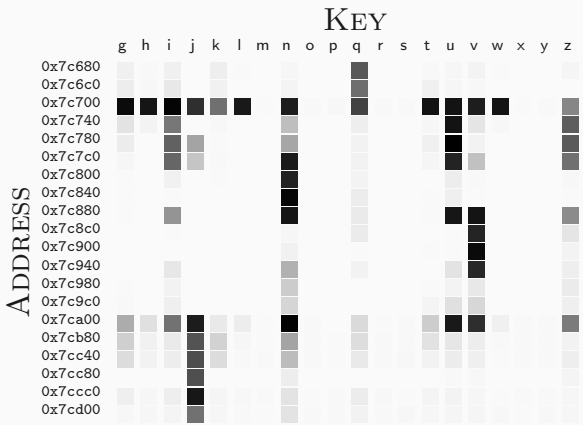
Profiling Phase: All Events, 1 Address

Example: Cache Hit Ratio for $(\texttt{0x7c800}, \texttt{u})$: 13 / 200

Profiling Phase: All Events, 1 Address

Distinguish `n` from other keys by monitoring `0x7c800`

Profiling Phase: All Events, All Addresses



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