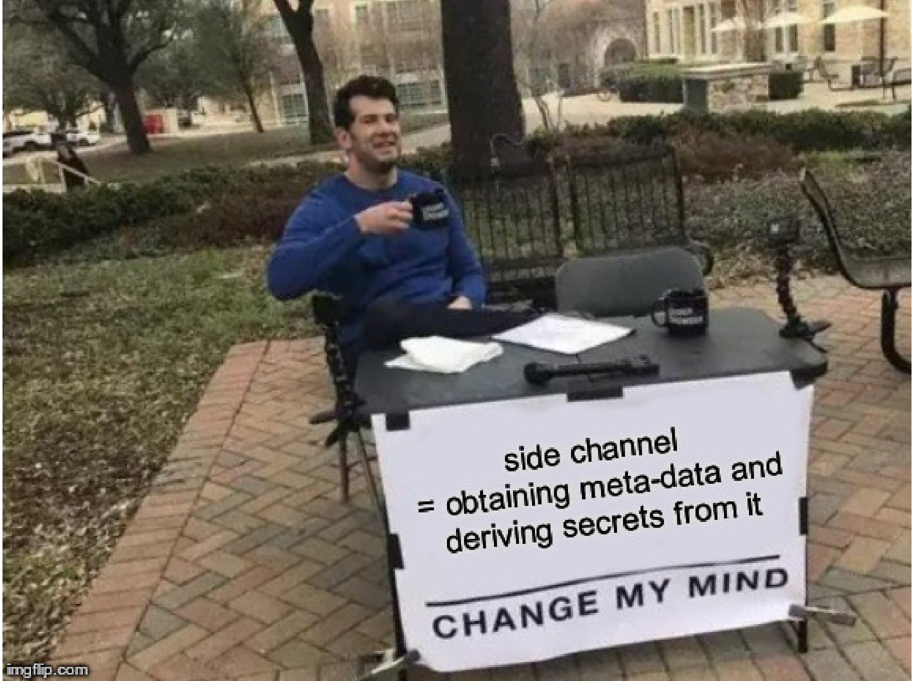
Side-Channel Security

Chapter 2: Caches & Cache Attacks

Daniel Gruss, Roland Czerny

March 6, 2025

Graz University of Technology


side channel
= obtaining meta-data and
deriving secrets from it
CHANGE MY MIND
imgflip.com

TLB and Paging

- Paging: memory translated page-wise from virtual to physical
- TLB (translation lookaside buffer) caches virtual to physical mapping
- TLB has some latency

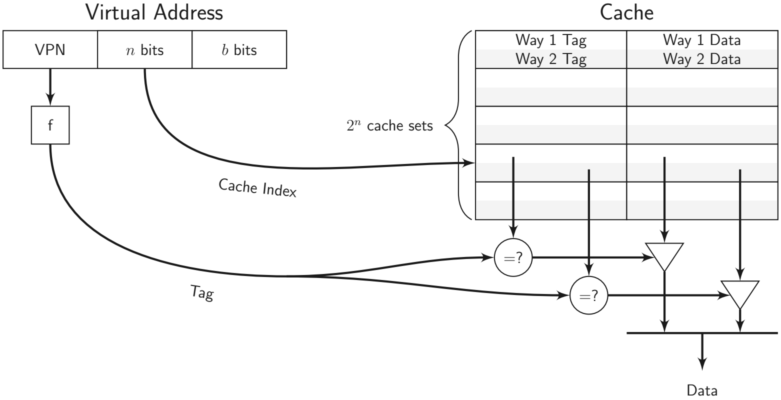
TLB and Paging

- Paging: memory translated page-wise from virtual to physical
- TLB (translation lookaside buffer) caches virtual to physical mapping
- TLB has some latency
- Worst case for Cache: mapping not in TLB, need to load mapping from RAM
- Solution: Use virtual addresses instead of physical addresses

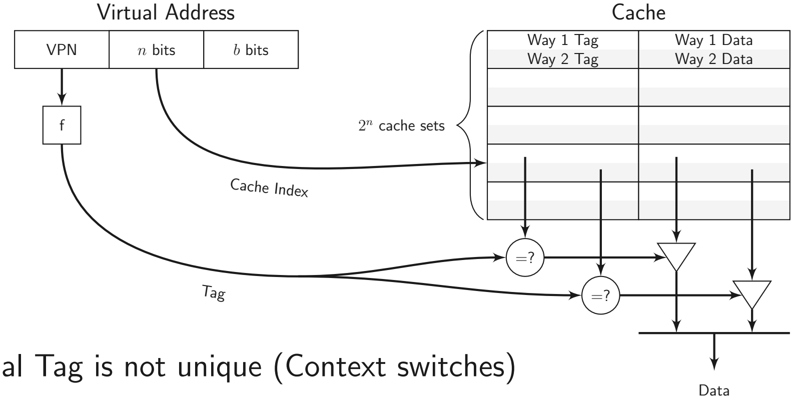
Cache indexing methods

- VIVT: Virtually indexed, virtually tagged
- PIPT: Physically indexed, physically tagged
- PIVT: Physically indexed, virtually tagged
- VIPT: Virtually indexed, physically tagged

VIVT

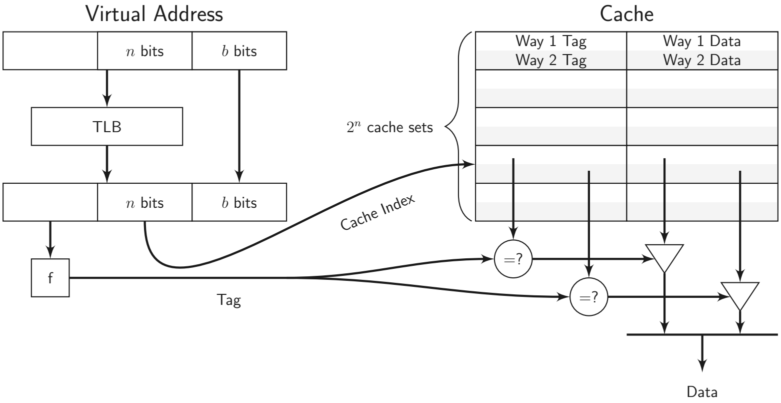

Virtual Address
VPN
n bits
b bits
f
Cache Index
Tag
2^n cache sets
Cache
Way 1 Tag
Way 2 Tag
Way 1 Data
Way 2 Data
=?
=?
Data

VIVT



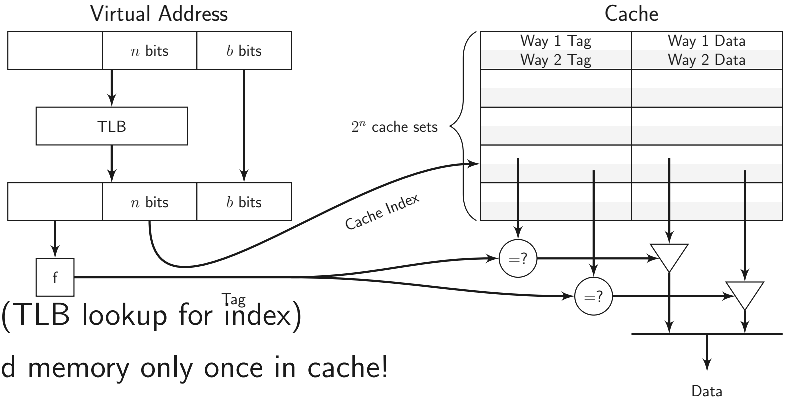
- Fast
- Virtual Tag is not unique (Context switches)
- Shared memory more than once in cache

PIPT



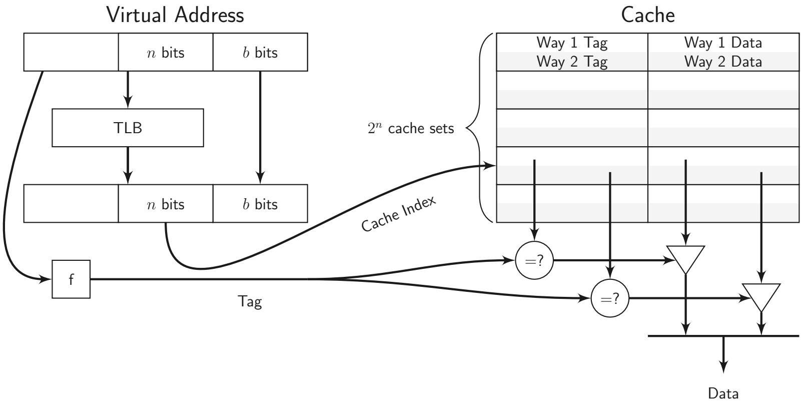
Virtual Address
n bits
b bits
TLB
n bits
b bits
f
Tag
Cache Index
2^n cache sets
Cache
Way 1 Tag
Way 2 Tag
Way 1 Data
Way 2 Data
=?
=?
Data

PIPT

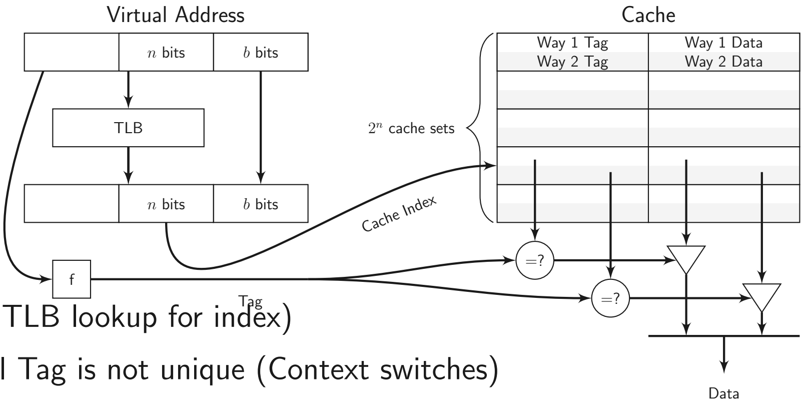


- Slow (TLB lookup for index)
- Shared memory only once in cache!

(PIVT)

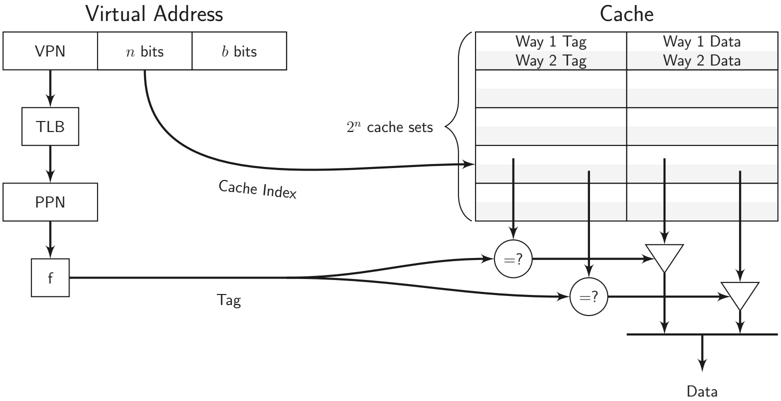

Virtual Address
n bits
b bits
TLB
n bits
b bits
f
Tag
Cache Index
2^n cache sets
Cache
Way 1 Tag
Way 2 Tag
Way 1 Data
Way 2 Data
=?
=?
Data

(PIVT)

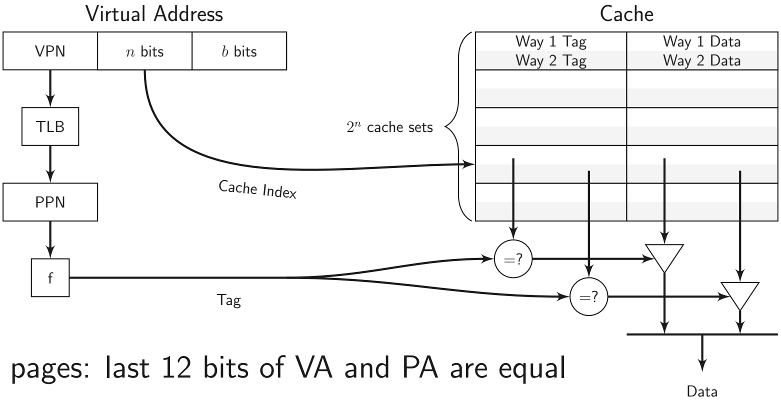


- Slow (TLB lookup for index)
- Virtual Tag is not unique (Context switches)
- Shared memory more than once in cache

VIPT


Virtual Address
VPN
n bits
b bits
TLB
PPN
f
Tag
Cache Index
2^n cache sets
Cache
Way 1 Tag
Way 2 Tag
Way 1 Data
Way 2 Data
=?
=?
Data

VIPT



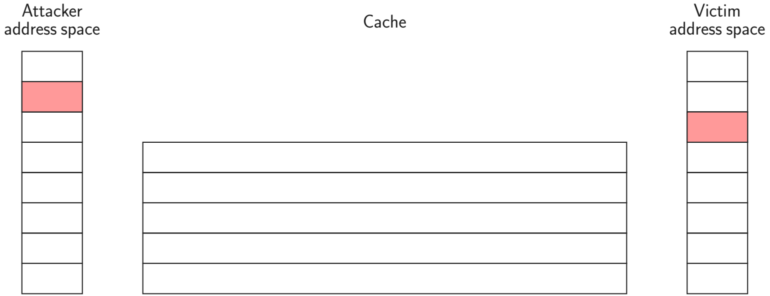
- Fast
- 4 KiB pages: last 12 bits of VA and PA are equal
- Using more bits is unpractical (like VIVT)

→ Cache size $\leq$ # ways $\cdot$ page size

Remarks

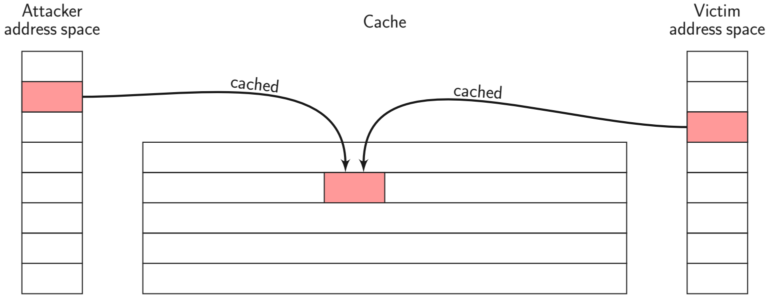
- L1 caches: VIVT or VIPT
- L2/L3 caches: PIPT

Flush+Reload



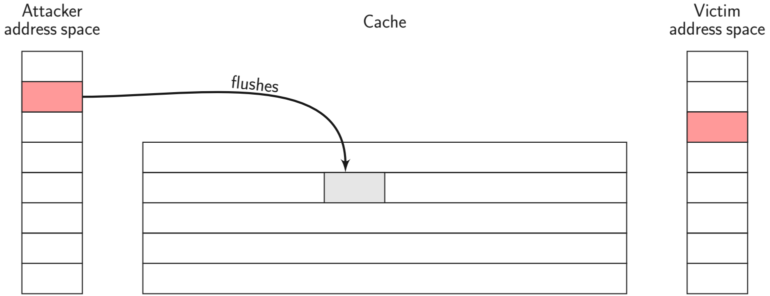
step 0: attacker maps shared library → shared memory, shared in cache

Flush+Reload



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

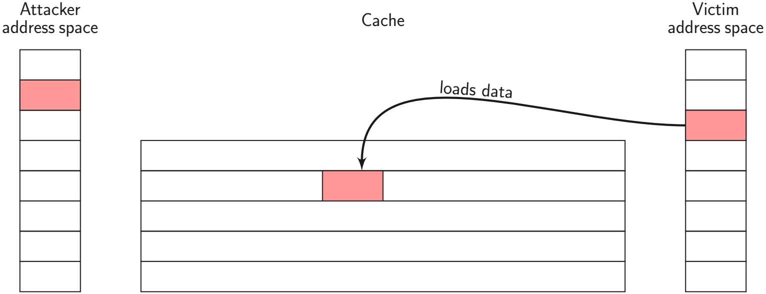
Flush+Reload



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

step 1: attacker flushes the shared line

Flush+Reload

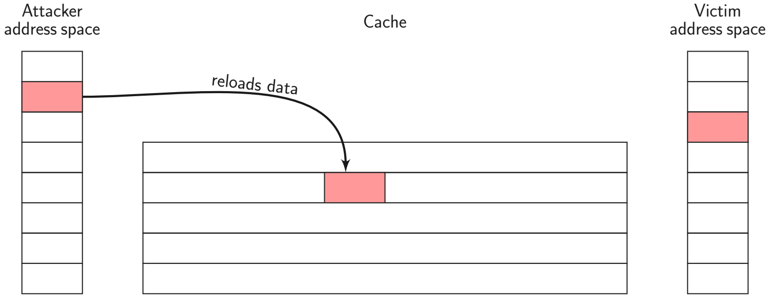


step 0: attacker maps shared library → shared memory, shared in cache

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)

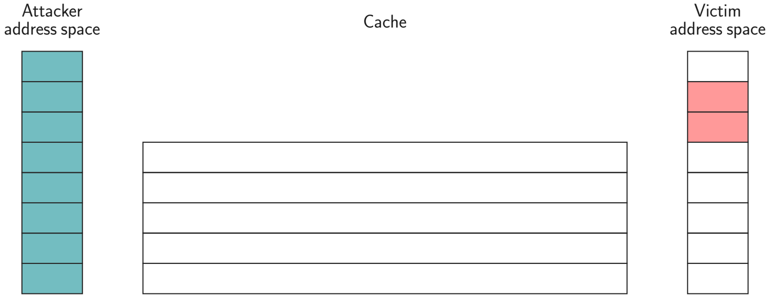
Cons: restrictive

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

Variants of Flush+Reload

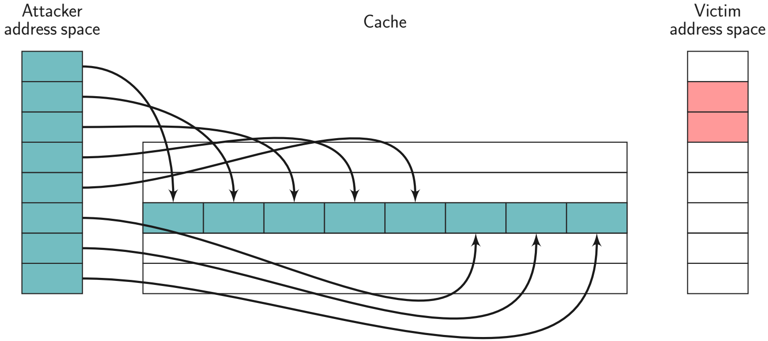
- Flush+Flush [2]
- Evict+Reload [3] on ARM [5]

Prime+Probe



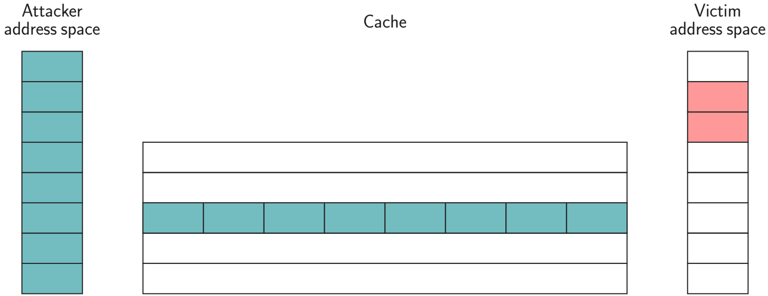
step 0: attacker fills the cache (prime)

Prime+Probe



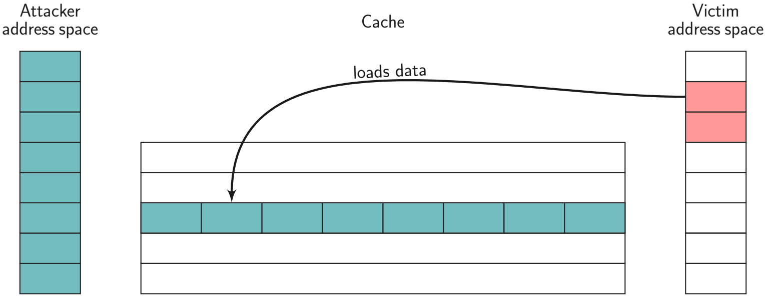
step 0: attacker fills the cache (prime)

Prime+Probe



step 0: attacker fills the cache (prime)

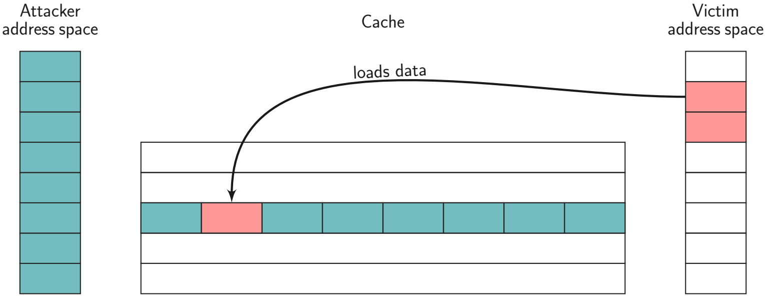
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

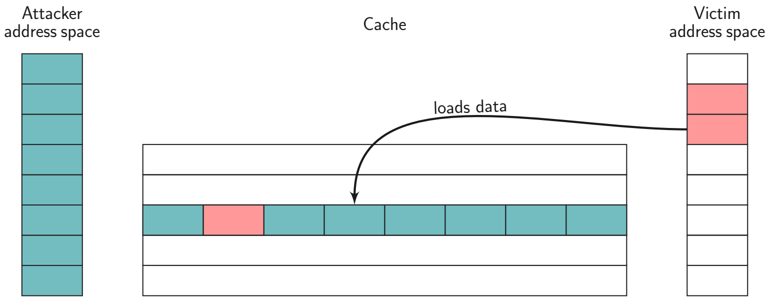
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

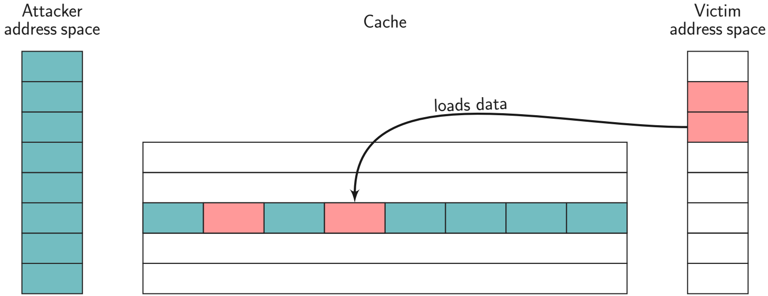
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

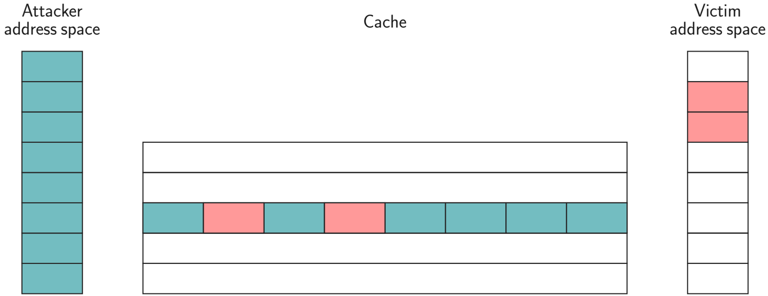
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

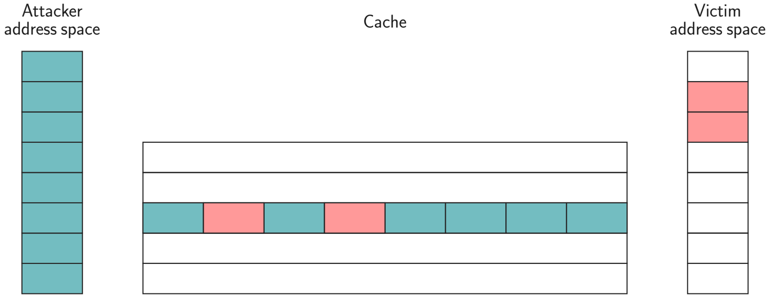
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

Prime+Probe

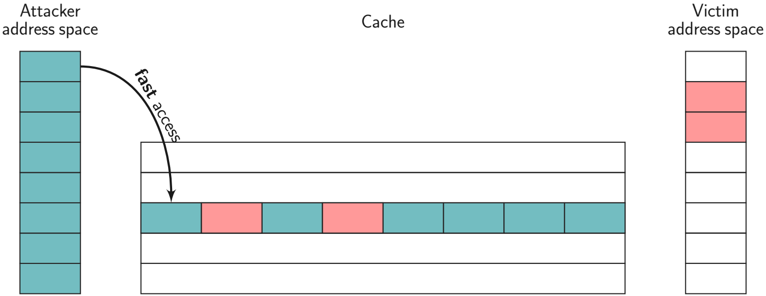


step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

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

Prime+Probe

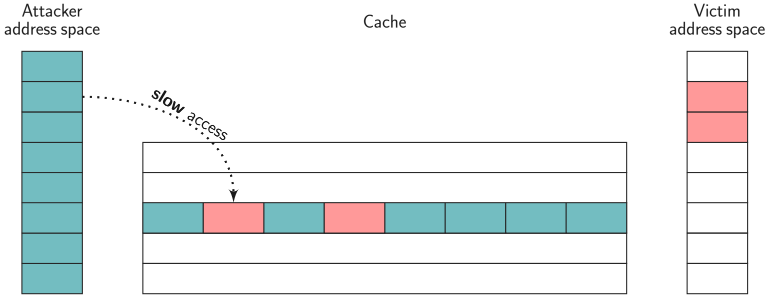


step 0: attacker fills the cache (prime)

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

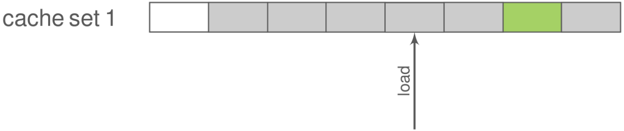
Cons: coarser granularity (1 set)

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?

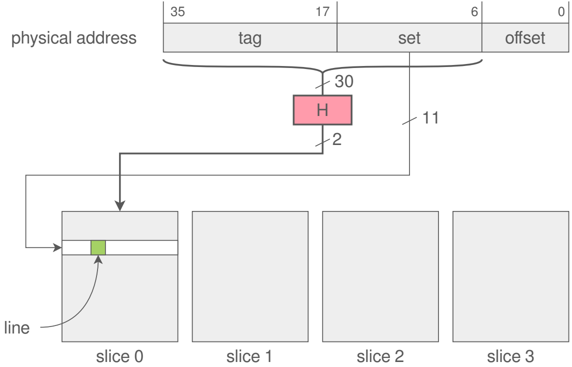
#1.1: Which physical addresses to access?



“LRU eviction”:

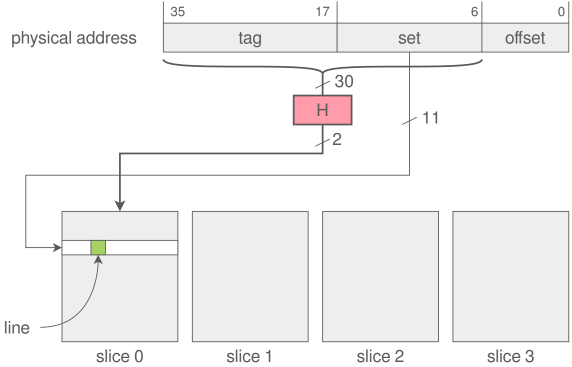
- assume that cache uses LRU replacement
- accessing n addresses from the same cache set to evict an n-way set
- eviction from last level → from whole hierarchy (it’s inclusive!)

#1.2: Which addresses map to the same set?



- function H that maps slices is undocumented
- reverse-engineered by [4, 7, 9]

#1.2: Which addresses map to the same set?



- function H that maps slices is undocumented
- reverse-engineered by [4, 7, 9]
- hash function basically an XOR of address bits

#1.2: Which addresses map to the same set?

3 functions, depending on the number of cores

		Address bit																													
		3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	0	0	0	0
		7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6
2 cores	o_0						⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
4 cores	o_0					⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1				⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕				⊕	
8 cores	o_0		⊕	⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1	⊕		⊕	⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕				⊕	
	o_2	⊕	⊕	⊕	⊕			⊕	⊕			⊕	⊕			⊕	⊕			⊕			⊕			⊕	⊕				⊕		

#2: Obtain information without root privileges

- last-level cache is physically indexed

#2: Obtain information without root privileges

- last-level cache is physically indexed
- root privileges needed for physical addresses

#2: Obtain information without root privileges

- last-level cache is physically indexed
- root privileges needed for physical addresses
- use 2 MB pages → lowest 21 bits are the same as virtual address

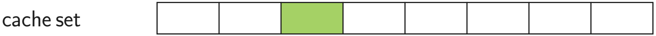
#2: Obtain information without root privileges

- last-level cache is physically indexed
- root privileges needed for physical addresses
- use 2 MB pages → lowest 21 bits are the same as virtual address

→ enough to compute the cache set

#3.1: Replacement policy on older CPUs

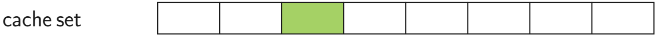
“LRU eviction” memory accesses



cache set

#3.1: Replacement policy on older CPUs

“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first

#3.1: Replacement policy on older CPUs

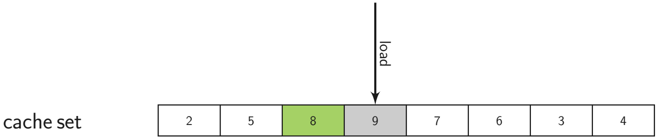
“LRU eviction” memory accesses

cache set	2	5	8	1	7	6	3	4

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

#3.1: Replacement policy on older CPUs

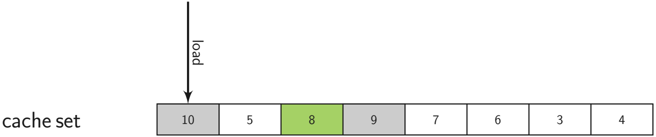
"LRU eviction" memory accesses



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

#3.1: Replacement policy on older CPUs

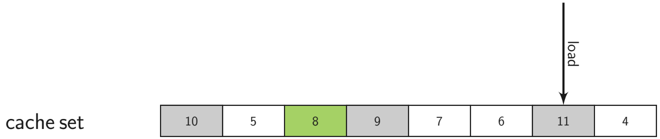
“LRU eviction” memory accesses



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

#3.1: Replacement policy on older CPUs

"LRU eviction" memory accesses



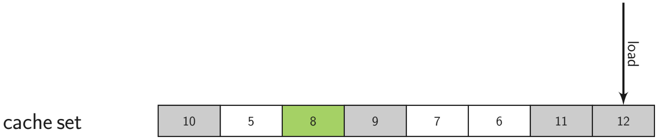
load

cache set	10	5	8	9	7	6	11	4

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

#3.1: Replacement policy on older CPUs

“LRU eviction” memory accesses



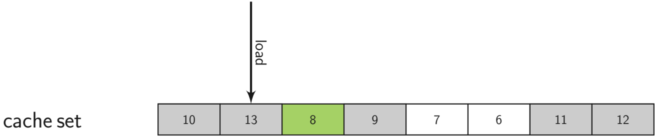
load

cache set	10	5	8	9	7	6	11	12

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

#3.1: Replacement policy on older CPUs

"LRU eviction" memory accesses



load

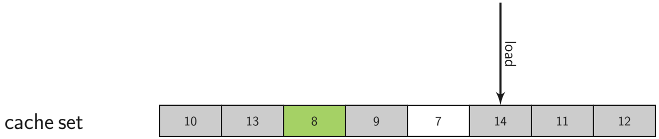
cache set

10 | 13 | 8 | 9 | 7 | 6 | 11 | 12

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

#3.1: Replacement policy on older CPUs

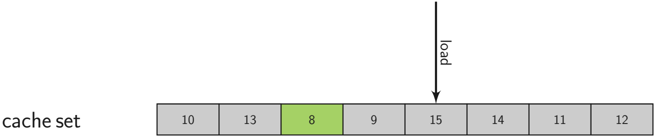
“LRU eviction” memory accesses



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

#3.1: Replacement policy on older CPUs

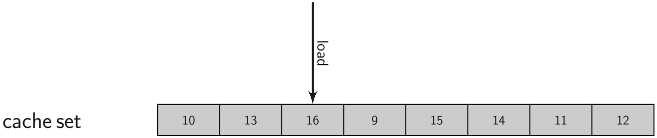
"LRU eviction" memory accesses



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

#3.1: Replacement policy on older CPUs

“LRU eviction” memory accesses



load

cache set

10	13	16	9	15	14	11	12

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

#3.2: Replacement policy on recent CPUs

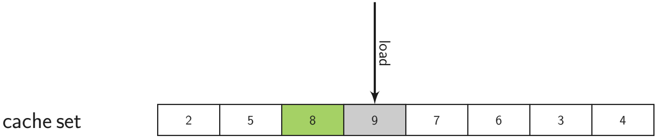
“LRU eviction” memory accesses

cache set	2	5	8	1	7	6	3	4

- no LRU replacement

#3.2: Replacement policy on recent CPUs

“LRU eviction” memory accesses



load

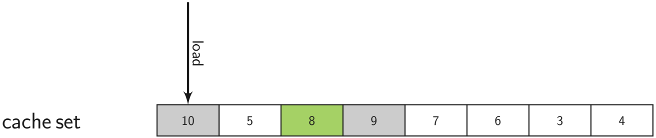
cache set

2	5	8	9	7	6	3	4

- no LRU replacement

#3.2: Replacement policy on recent CPUs

“LRU eviction” memory accesses



load

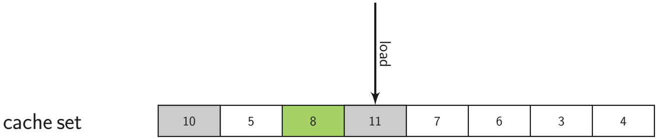
cache set

10	5	8	9	7	6	3	4

- no LRU replacement

#3.2: Replacement policy on recent CPUs

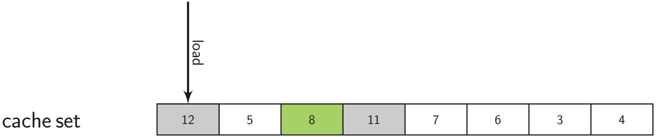
“LRU eviction” memory accesses



- no LRU replacement

#3.2: Replacement policy on recent CPUs

“LRU eviction” memory accesses



load

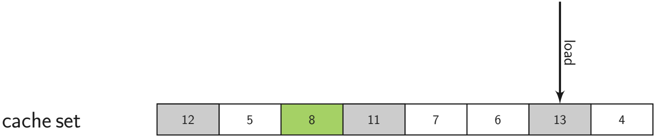
cache set

12	5	8	11	7	6	3	4

- no LRU replacement

#3.2: Replacement policy on recent CPUs

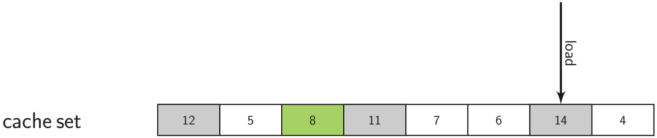
"LRU eviction" memory accesses



- no LRU replacement

#3.2: Replacement policy on recent CPUs

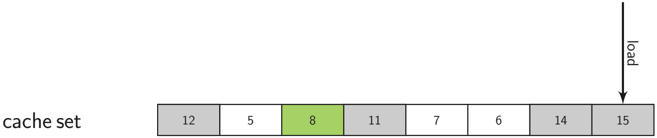
“LRU eviction” memory accesses



- no LRU replacement

#3.2: Replacement policy on recent CPUs

“LRU eviction” memory accesses



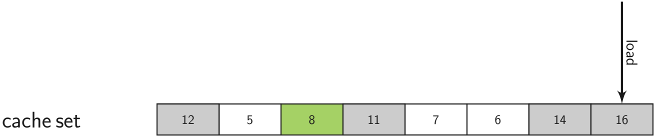
load

cache set	12	5	8	11	7	6	14	15

- no LRU replacement

#3.2: Replacement policy on recent CPUs

“LRU eviction” memory accesses



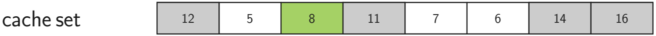
load

cache set	12	5	8	11	7	6	14	16

- no LRU replacement

#3.2: Replacement policy on recent CPUs

“LRU eviction” memory accesses



cache set

12	5	8	11	7	6	14	16

- no LRU replacement
- only 75% success rate on Haswell

#3.2: Replacement policy on recent CPUs

“LRU eviction” memory accesses

cache set	12	5	8	11	7	6	14	16

- no LRU replacement
- only 75% success rate on Haswell
- more accesses → higher success rate, but too slow

#3.3: Cache eviction strategy

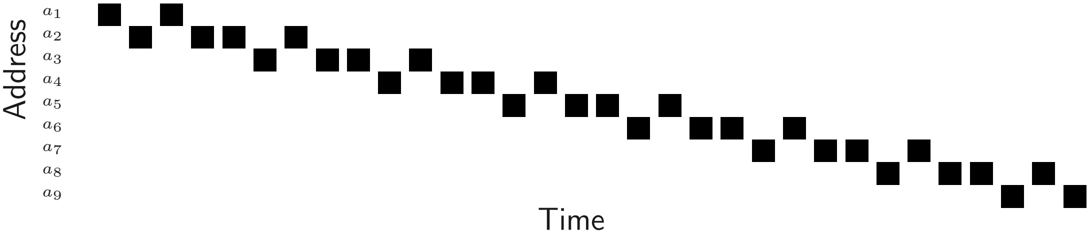


Figure 1: Fast and effective on Haswell. Eviction rate >99.97%.

Cache covert channels

Side channels vs covert channels

- side channel: attacker spies a victim process
- covert channel: communication between two processes
  - that are not supposed to communicate
  - that are collaborating

1-bit cache covert channels

ideas for 1-bit channels:

1-bit cache covert channels

ideas for 1-bit channels:

- Prime+Probe: use one cache set to transmit

  0: sender does not access the set → low access time in receiver

  1: sender does access the set → high access time in receiver

1-bit cache covert channels

ideas for 1-bit channels:

- Prime+Probe: use one cache set to transmit

  0: sender does not access the set → low access time in receiver

  1: sender does access the set → high access time in receiver

- Flush+Reload/Flush+Flush/Evict+Reload: use one address to transmit

  0: sender does not access the address → high access time in receiver

  1: sender does access the address → low access time in receiver

1-bit covert channels

- 1 bit data, 0 bit control?

1-bit covert channels

- 1 bit data, 0 bit control?
- idea: divide time into slices (e.g., $50\mu s$ frames)
- synchronize sender and receiver with a shared clock

Problems of 1-bit covert channels

- errors?

Problems of 1-bit covert channels

- errors? → error-correcting codes
- retransmission may be more efficient (less overhead)
- desynchronization
- optimal transmission duration may vary

Multi-bit covert channels

- combine multiple 1-bit channels

Multi-bit covert channels

- combine multiple 1-bit channels
- avoid interferences

→ higher performance

Multi-bit covert channels

- combine multiple 1-bit channels
- avoid interferences

→ higher performance

- use 1-bit for sending = true/false

Packets / frames

Organize data in packets / frames:

- some data bits
- check sum
- sequence number

→ keep sender and receiver synchronous

→ check whether retransmission is necessary

Efficient retransmission

How can the sender know when to retransmit?

- idea: acknowledge packets (requires a backward channel)

Efficient retransmission

How can the sender know when to retransmit?

- idea: acknowledge packets (requires a backward channel)
- use some bits as a backward channel
- use the same bits as a backward channel (sender sending bit/receiver sending bit)

Efficient retransmission

How can the sender know when to retransmit?

- idea: acknowledge packets (requires a backward channel)
- use some bits as a backward channel
- use the same bits as a backward channel (sender sending bit/receiver sending bit)
- why wait for retransmission?

Efficient retransmission

How can the sender know when to retransmit?

- idea: acknowledge packets (requires a backward channel)
- use some bits as a backward channel
- use the same bits as a backward channel (sender sending bit/receiver sending bit)
- why wait for retransmission?

→ sender should retransmit until receiver acknowledged

Raw capacity C

- number of bits per second
- measure over ≥ 1 minute

s bits transmitted in 1 minute:

$$C = \frac{s}{60}$$

Bit error rate p

- count bits that are wrong w
- count total bits sent b_s
- count total bits received b_r

Error rate:

$$p = \frac{w + |b_r - b_s|}{\max\left(b_s, b_r\right)},$$

or if $b_r = b_s$:

$$p = \frac{w}{b_r},$$

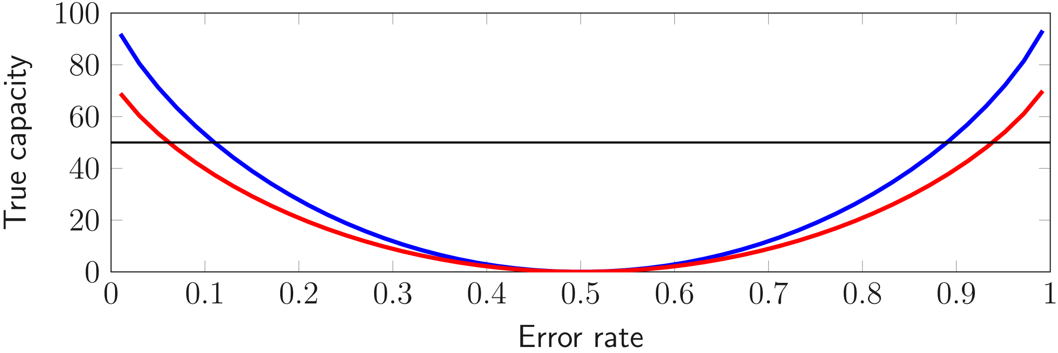
Capacity

True capacity T:

$$T = C \cdot (1 + ((1 - p) \cdot \log_2 (1 - p) + p \cdot \log_2 (p)))$$

C is the *raw capacity* and p is the *bit error rate*.

Capacity



True capacity
100
80
60
40
20
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Error rate

State of the art

method	raw capacity	err. rate	true capacity	env.
F+F [2]	3968Kbps	0.840%	3690Kbps	native
F+R [2]	2384Kbps	0.005%	2382Kbps	native
E+R [5]	1141Kbps	1.100%	1041Kbps	native
P+P [8]	601Kbps	0.000%	601Kbps	native
P+P [6]	600Kbps	1.000%	552Kbps	virt
P+P [8]	362Kbps	0.000%	362Kbps	native

Cache template attacks

Cache Template Attacks

Cache Template Attacks

- State of the art: cache attacks are powerful
- Problem: manual identification of attack targets

Cache Template Attacks

- State of the art: cache attacks are powerful
- Problem: manual identification of attack targets
- Solution: Cache Template Attacks
- Automatically find any secret-dependent cache access
- Can be used for attacks and to improve software

Cache Template Attacks

- State of the art: cache attacks are powerful
- Problem: manual identification of attack targets
- Solution: Cache Template Attacks
- Automatically find any secret-dependent cache access
- Can be used for attacks and to improve software
- Examples:
  - Cache-based keylogger
  - Automatic attacks on crypto algorithms

Challenges

- How to locate key-dependent memory accesses?

Challenges

- How to locate key-dependent memory accesses?
- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries
  - Self-compiled binaries

Challenges

- How to locate key-dependent memory accesses?
- 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
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - called “Cache Template”

Cache Template Attacks

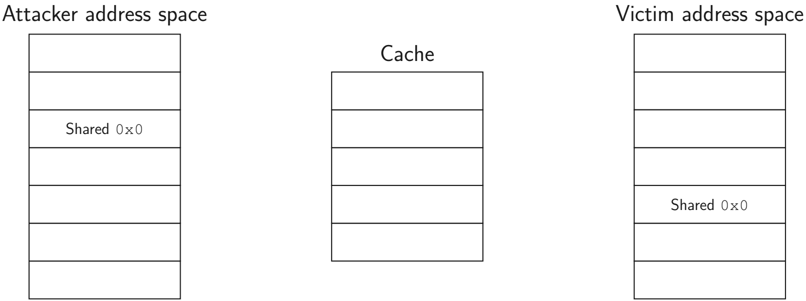
Profiling Phase

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

Exploitation Phase

- Monitor exploitable addresses

Profiling Phase



Attacker address space

Shared `0x0`

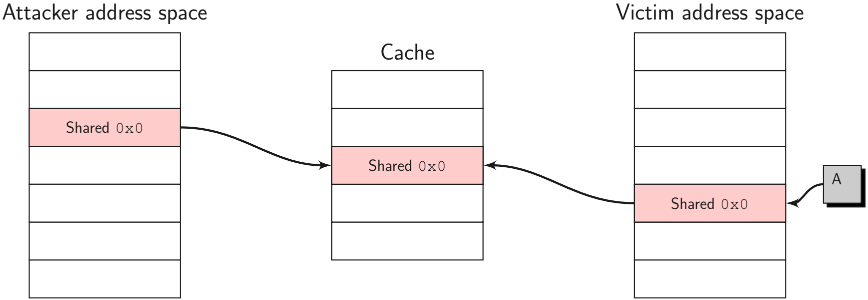
Cache

Victim address space

Shared `0x0`

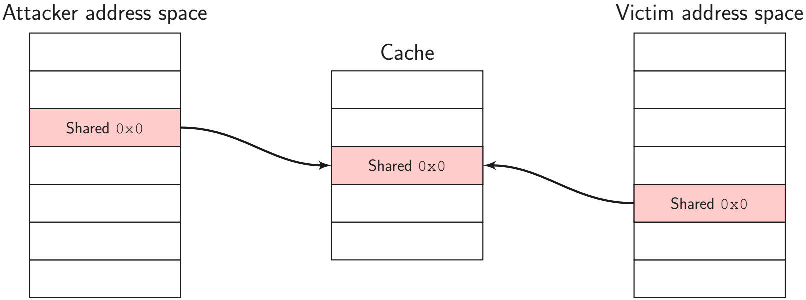
Cache is empty

Profiling Phase



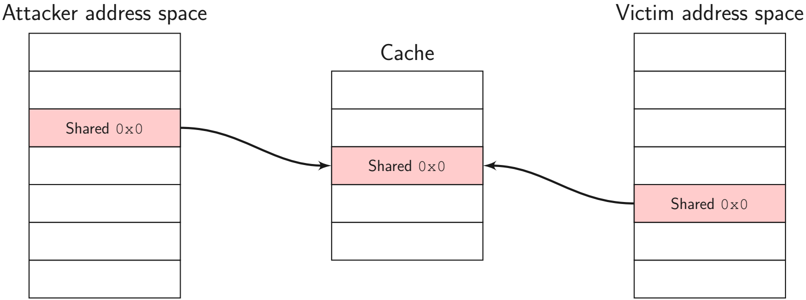
Attacker triggers an event

Profiling Phase



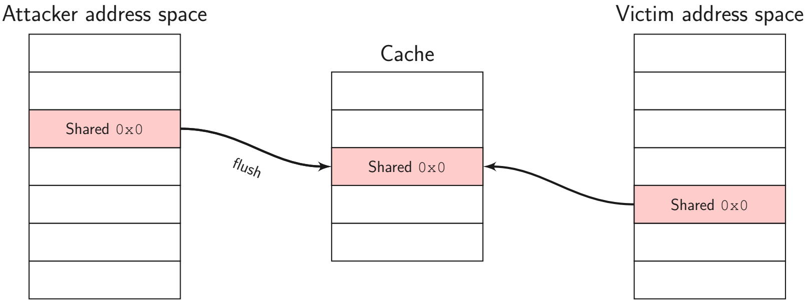
Attacker checks one address for cache hits (“Reload”)

Profiling Phase



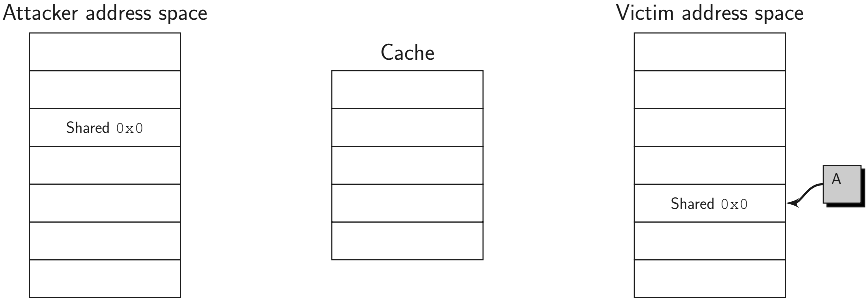
Update cache hit ratio (per event and address)

Profiling Phase



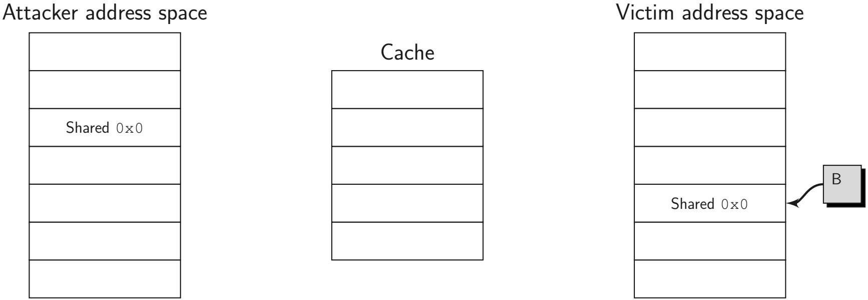
Attacker flushes shared memory

Profiling Phase



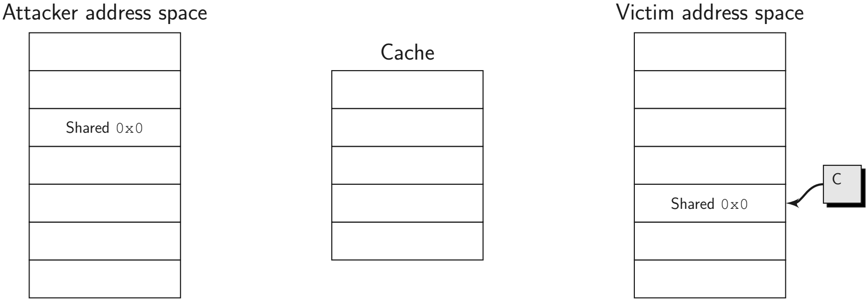
Repeat for higher accuracy

Profiling Phase



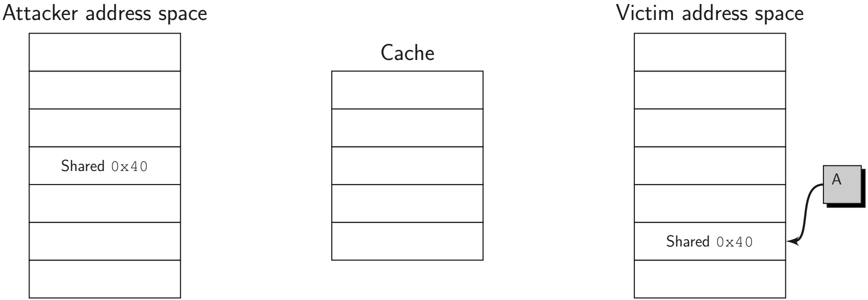
Repeat for all events

Profiling Phase



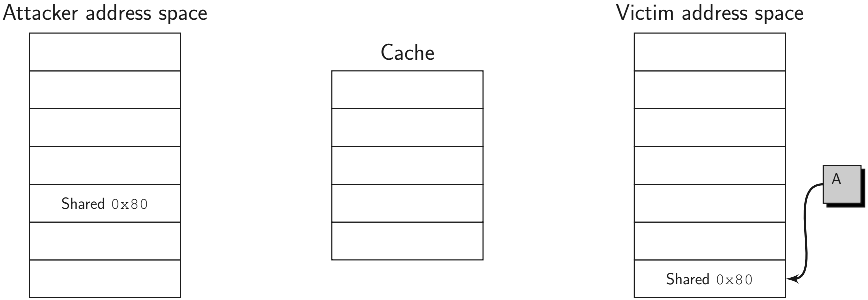
Repeat for all events

Profiling Phase

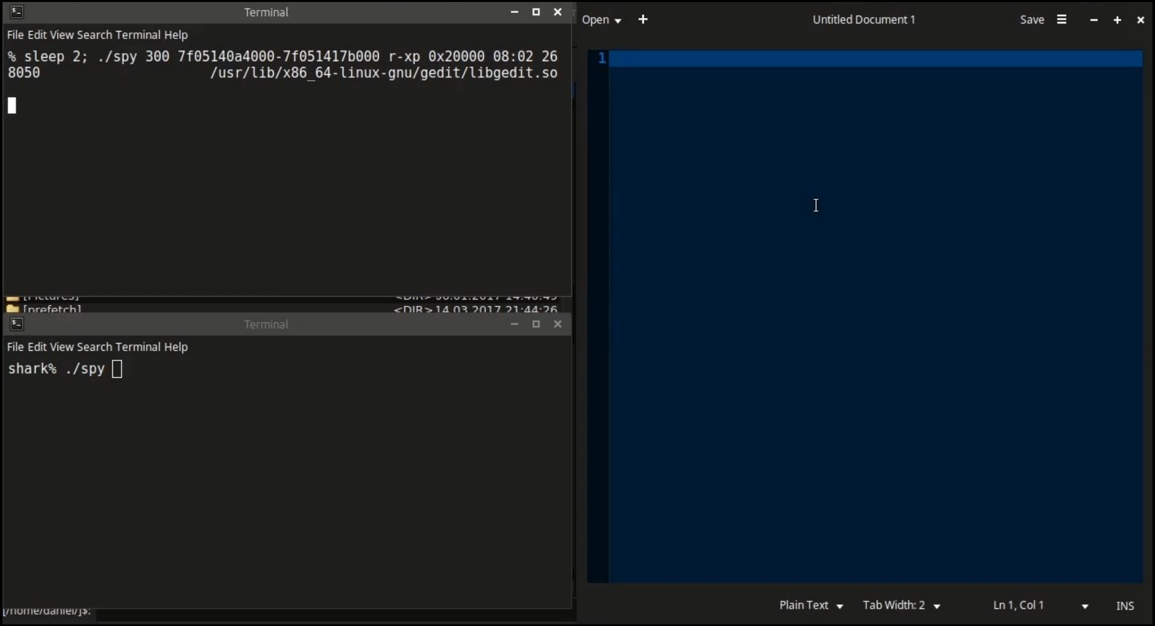


Continue with next address

Profiling Phase



Continue with next address

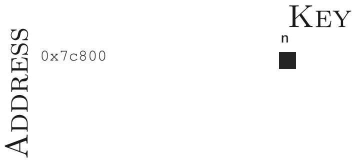

Terminal
File Edit View Search Terminal Help
[prefetch] <DIR> 14.03.2017 21:44:26
Terminal
File Edit View Search Terminal Help
/home/daniel/]$:
Open
Untitled Document 1
Save
Plain Text
Tab Width: 2
Ln 1, Col 1
INS

```
% sleep 2; ./spy 300 7f05140a4000-7f051417b000 r-xp 0x20000 08:02 26
8050                    /usr/lib/x86_64-linux-gnu/gedit/libgedit.so
```

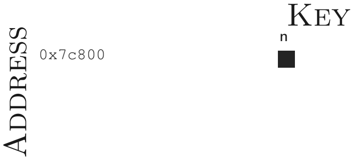
```
shark% ./spy
```

Cache Template Attack Demo

Profiling Phase: 1 Event, 1 Address

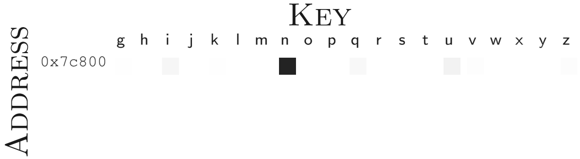


Profiling Phase: 1 Event, 1 Address

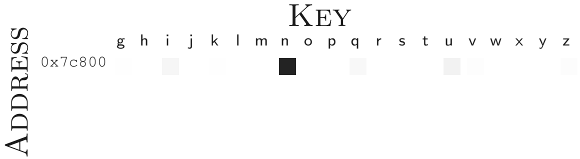


Example: Cache Hit Ratio for $(\texttt{0x7c800}, \texttt{n})$: 200 / 200

Profiling Phase: All Events, 1 Address

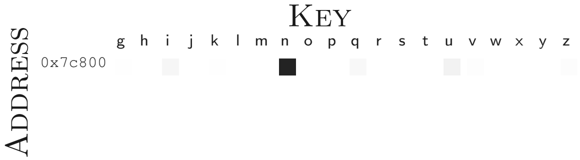


Profiling Phase: All Events, 1 Address



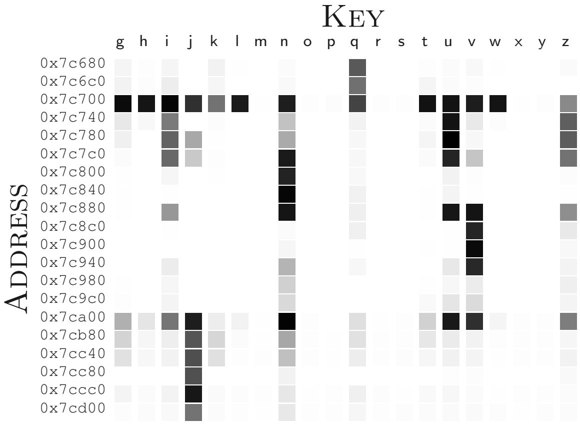
Example: Cache Hit Ratio for $(\texttt{0x7c800}, \texttt{u})$: 13 / 200

Profiling Phase: All Events, 1 Address



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

Profiling Phase: All Events, All Addresses



Attack 3: Locate AES T-Tables

AES uses T-Tables (precomputed from S-Boxes)

- 4 T-Tables
- $$T_0\left[k_{\{0,4,8,12\}} \oplus p_{\{0,4,8,12\}}\right]$$
  $$T_1\left[k_{\{1,5,9,13\}} \oplus p_{\{1,5,9,13\}}\right]$$
  ...
- If we know which entry of T is accessed, we know the result of $k_i \oplus p_i$.
- Known-plaintext attack (p_i is known) $\rightarrow$ k_i can be determined

Attack 3: Locate AES T-Tables

AES T-Table implementation from OpenSSL 1.0.2

Attack 3: Locate AES T-Tables

AES T-Table implementation from OpenSSL 1.0.2

- Most addresses in two groups:
  - Cache hit ratio 100% (always cache hits)
  - Cache hit ratio 0% (no cache hits)

Attack 3: Locate AES T-Tables

AES T-Table implementation from OpenSSL 1.0.2

- Most addresses in two groups:
    - Cache hit ratio 100% (always cache hits)
    - Cache hit ratio 0% (no cache hits)
- One 4096 byte memory block:
    - Cache hit ratio of 92%
    - Cache hits depend on key value and plaintext value
    - The T-Tables

Attack 4: AES T-Table Template Attack

AES T-Table implementation from OpenSSL 1.0.2

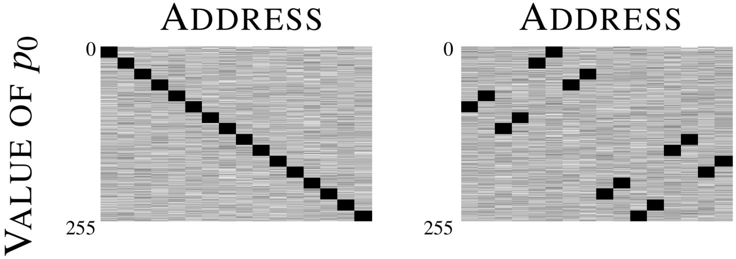
- Known-plaintext attack
- Events: encryption with only one fixed key byte

Attack 4: AES T-Table Template Attack

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event

Attack 4: AES T-Table Template



$k_0 = \texttt{0x00}$

$k_0 = \texttt{0x55}$

(transposed)

Attack 4: AES T-Table Template Attack

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Exploitation phase:
    - Eliminate key candidates

Attack 4: AES T-Table Template Attack

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Exploitation phase:
    - Eliminate key candidates
    - Reduction of key space in first-round attack:
        - 64 bits after 16–160 encryptions

Attack 4: AES T-Table Template Attack

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Exploitation phase:
    - Eliminate key candidates
    - Reduction of key space in first-round attack:
        - 64 bits after 16–160 encryptions
    - State of the art: full key recovery after 30000 encryptions

AES T-Table Template Attack: Exercise Hints

- Exploitation phase only
  - get table addresses from `AESlib_get_table_address([0-3])`

AES T-Table Template Attack: Exercise Hints

- Exploitation phase only
    - get table addresses from `AESlib_get_table_address([0-3])`
- Eliminate key candidates
    - Fix one plaintext byte at a time

AES T-Table Template Attack: Exercise Hints

- Exploitation phase only
  - get table addresses from `AESlib_get_table_address([0-3])`
- Eliminate key candidates
  - Fix one plaintext byte at a time
  - One cacheline will be accessed with 100% probability
    - Infer the corresponding key byte upper nibble
    - $p_i \oplus ? = T_{entry}$

AES T-Table Template Attack: Exercise Hints

- Exploitation phase only
  - get table addresses from `AESlib_get_table_address([0-3])`
- Eliminate key candidates
  - Fix one plaintext byte at a time
  - One cacheline will be accessed with 100% probability
    - Infer the corresponding key byte upper nibble
    - $p_i \oplus ? = T_{entry}$
  - Repeat for other plaintext bytes

AES T-Table Template Attack: Exercise Hints

- Exploitation phase only
    - get table addresses from `AESlib_get_table_address([0-3])`
- Eliminate key candidates
    - Fix one plaintext byte at a time
    - One cacheline will be accessed with 100% probability
        - Infer the corresponding key byte upper nibble
        - $p_i \oplus ? = T_{entry}$
    - Repeat for other plaintext bytes
- Recover 64 bits of key
- 3 bonus points for full key recovery

Conclusion

- Technique to find any cache side-channel leakage
  - Attacks
  - Detect vulnerabilities

Conclusion

- Technique to find any cache side-channel leakage
  - Attacks
  - Detect vulnerabilities
- Works on virtually all Intel CPUs
- Works even with unknown binaries

Conclusion

- Technique to find any cache side-channel leakage
    - Attacks
    - Detect vulnerabilities
- Works on virtually all Intel CPUs
- Works even with unknown binaries
- Marks a change of perspective:

Conclusion

- Technique to find any cache side-channel leakage
    - Attacks
    - Detect vulnerabilities
- Works on virtually all Intel CPUs
- Works even with unknown binaries
- Marks a change of perspective:
    - Large scale analysis of binaries
    - Large scale automated attacks

Conclusion

- Technique to find any cache side-channel leakage
    - Attacks
    - Detect vulnerabilities
- Works on virtually all Intel CPUs
- Works even with unknown binaries
- Marks a change of perspective:
    - Large scale analysis of binaries
    - Large scale automated attacks
- Apply templating to other side-channel leakage

Flush+Reload Variants

Evict+Reload

- Variant of Flush+Reload with cache eviction instead of `clflush`
- Works on ARMv7
- Applicable to millions of devices
- Cache Template Attacks using Evict+Reload
- ARMageddon: Last-Level Cache Attacks on Mobile Devices [5]

Flush+Flush: Motivation

- cache attacks → many cache misses
- detect via performance counters

Flush+Flush: Motivation

- cache attacks → many cache misses
- detect via performance counters

→ good idea, but is it good enough?

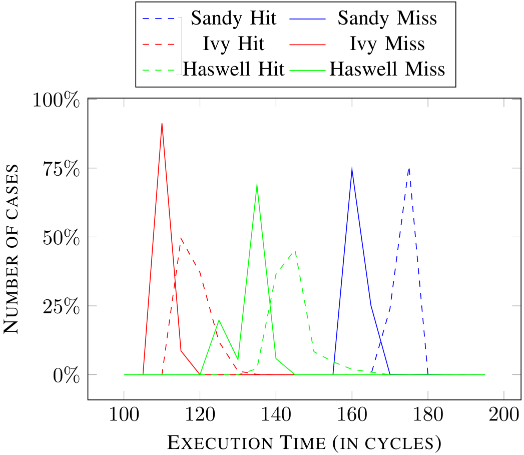
Flush+Flush: Motivation

- cache attacks → many cache misses
- detect via performance counters

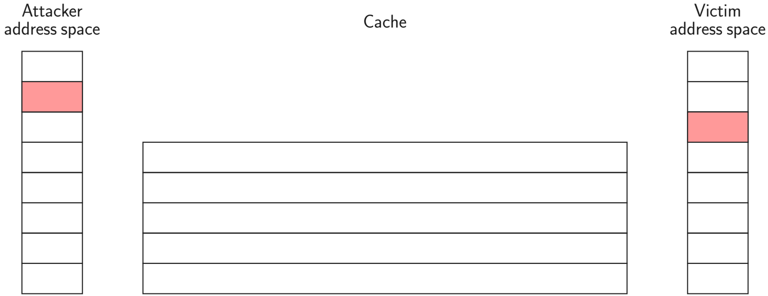
→ good idea, but is it good enough?

  - causing a cache flush $\neq$ causing a cache miss

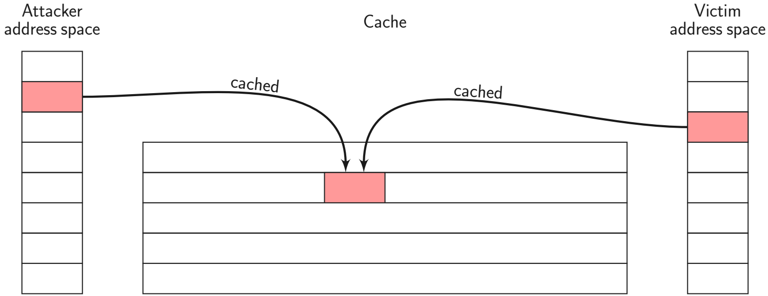
`clflush` execution time



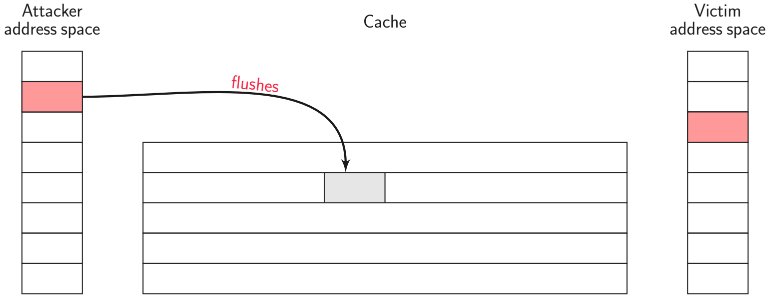
Flush+Flush



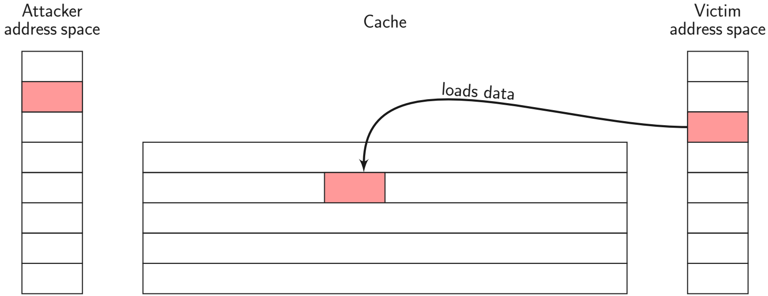
Flush+Flush


Attacker
address space
Cache
Victim
address space
cached
cached

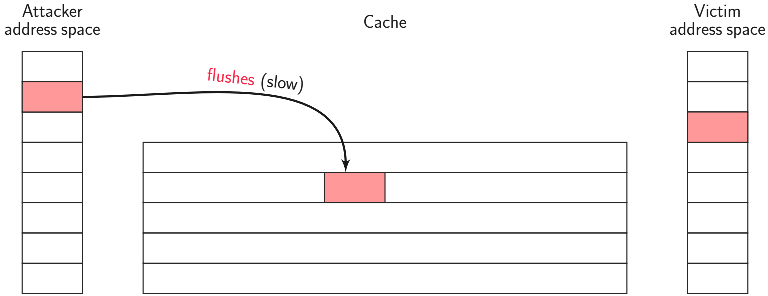
Flush+Flush


Attacker
address space
Cache
Victim
address space
flushes

Flush+Flush


Attacker
address space
Cache
Victim
address space
loads data

Flush+Flush


Attacker
address space
Cache
Victim
address space
flushes (slow)

Flush+Flush: Conclusion

- attacker causes no direct cache misses

  → fast

  → stealthy

Flush+Flush: Conclusion

- attacker causes no direct cache misses
  - → fast
  - → stealthy
- same side channel targets as Flush+Reload

Flush+Flush: Conclusion

- attacker causes no direct cache misses
  - → fast
  - → stealthy
- same side channel targets as Flush+Reload
- 496 KB/s covert channel

Cache Attacks on mobile devices?

- powerful cache attacks on Intel x86 in the last 10 years
- nothing like Flush+Reload or Prime+Probe on mobile devices

Cache Attacks on mobile devices?

- powerful cache attacks on Intel x86 in the last 10 years
- nothing like Flush+Reload or Prime+Probe on mobile devices

→ why?

Prefetch Side-Channel Attacks

Overview

- prefetch instructions don’t check privileges
- prefetch instructions leak timing information

Overview

- prefetch instructions don't check privileges
- prefetch instructions leak timing information

exploit this to:

- locate a driver in kernel = defeat KASLR
- translate virtual to physical addresses

Intel being overspecific

NOTE

Intel being overspecific

NOTE

Using the PREFETCH instruction is recommended only if data does not fit in cache.

Intel being overspecific

NOTE

Using the PREFETCH instruction is recommended only if data does not fit in cache. Use of software prefetch should be limited to memory addresses that are managed or owned within the application context.

Intel being overspecific

NOTE

Using the PREFETCH instruction is recommended only if data does not fit in cache. Use of software prefetch should be limited to memory addresses that are managed or owned within the application context. Prefetching to addresses that are not mapped to physical pages can experience non-deterministic performance penalty.

Intel being overspecific

NOTE

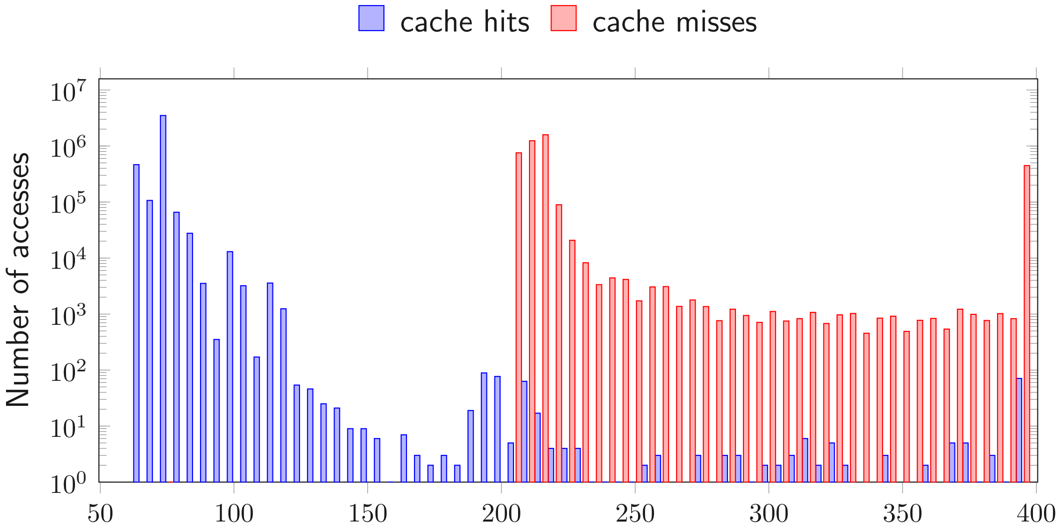
Using the PREFETCH instruction is recommended only if data does not fit in cache. Use of software prefetch should be limited to memory addresses that are managed or owned within the application context. Prefetching to addresses that are not mapped to physical pages can experience non-deterministic performance penalty. For example specifying a NULL pointer (0L) as address for a prefetch can cause long delays.

CPU Caches

Memory (DRAM) is slow compared to the CPU

- buffer frequently used memory
- every memory reference goes through the cache
- based on physical addresses

Memory Access Latency



Unprivileged cache maintainance

Optimize cache usage:

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

... based on virtual addresses

Software prefetching

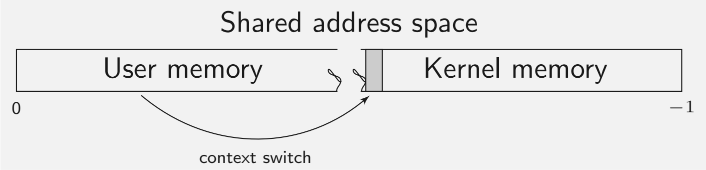
`prefetch` instructions are somewhat unusual

- hints – can be ignored by the CPU
- do not check privileges or cause exceptions

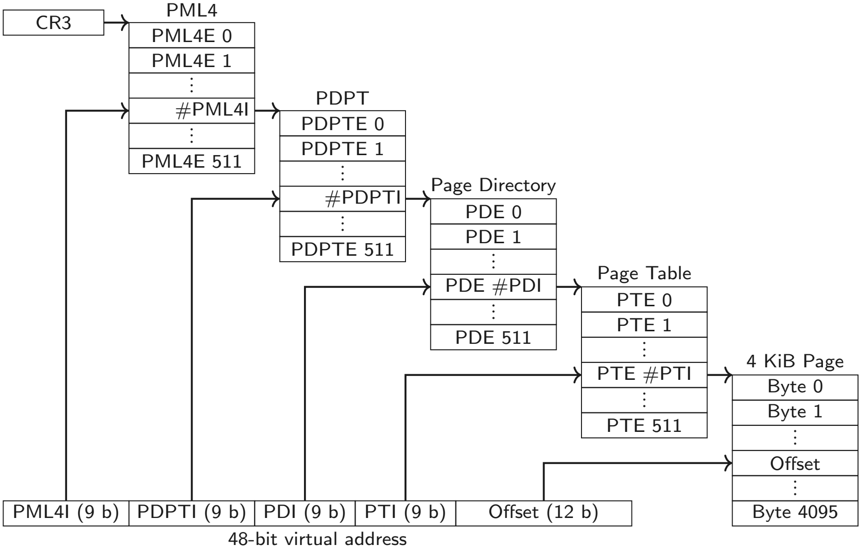
but they do need to translate virtual to physical

Kernel must be mapped in every address space

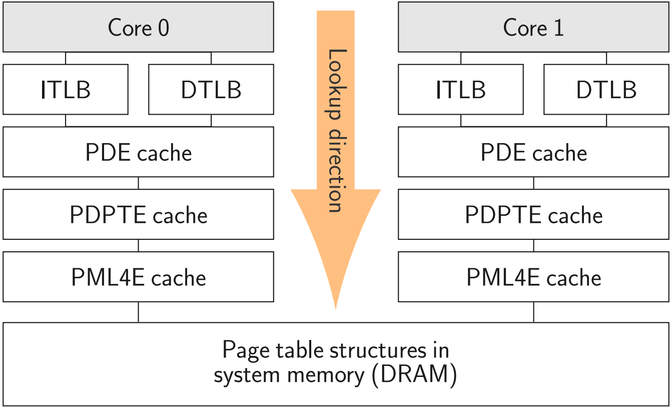
Today's operating systems:



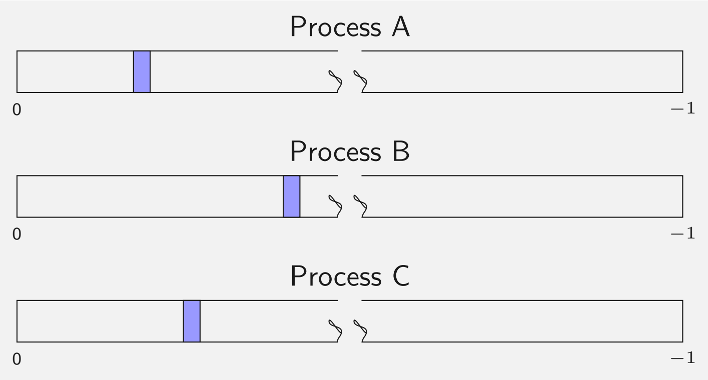
Address translation on x86-64



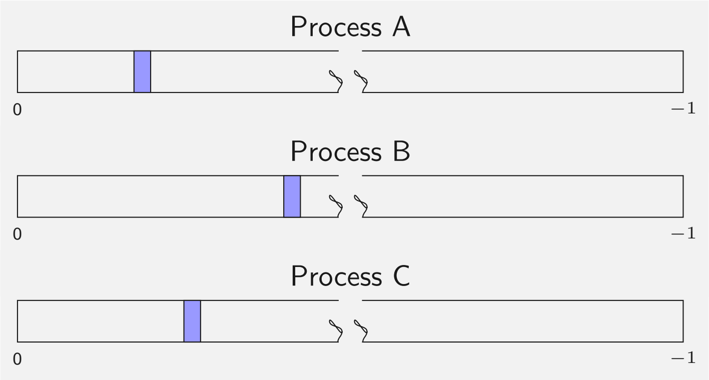
Address Translation Caches



Address Space Layout Randomization (ASLR)

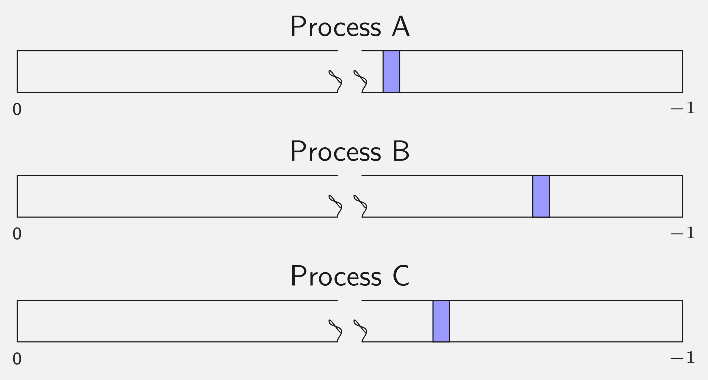


Address Space Layout Randomization (ASLR)

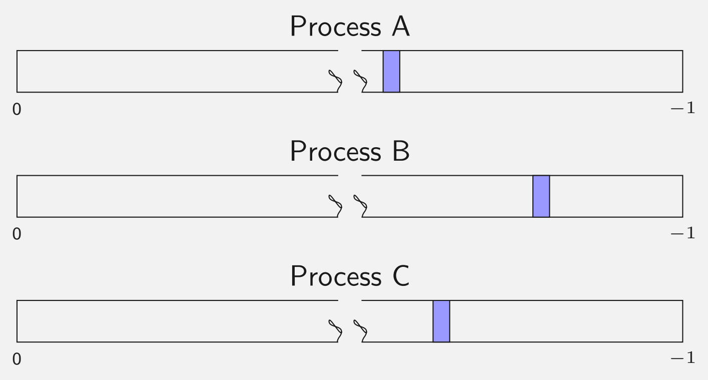


Same library – different offset!

Kernel Address Space Layout Randomization (KASLR)

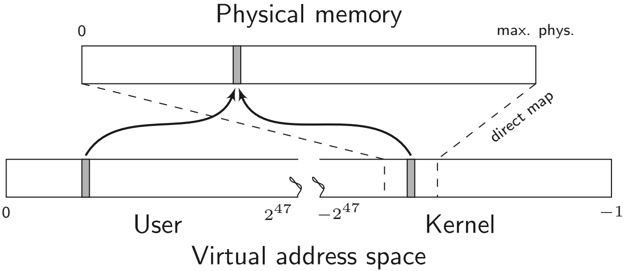


Kernel Address Space Layout Randomization (KASLR)

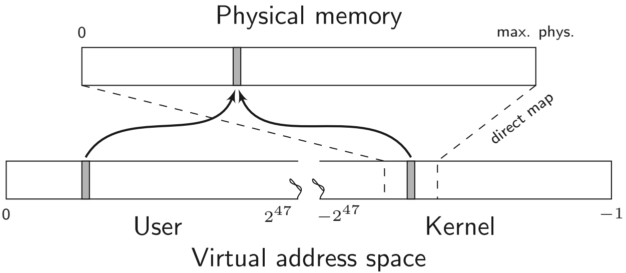


Same driver – different offset!

Kernel direct-physical map

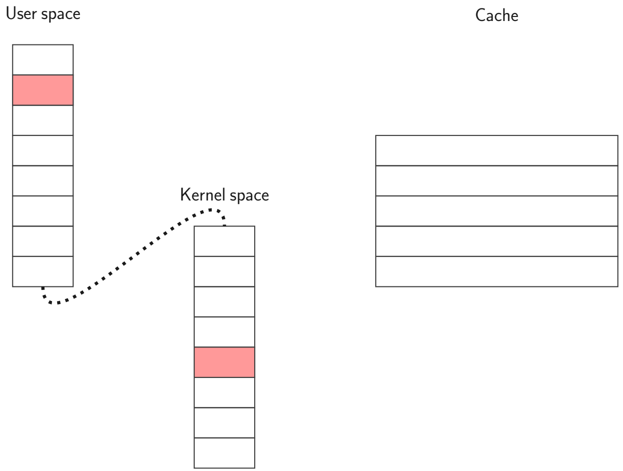


Kernel direct-physical map

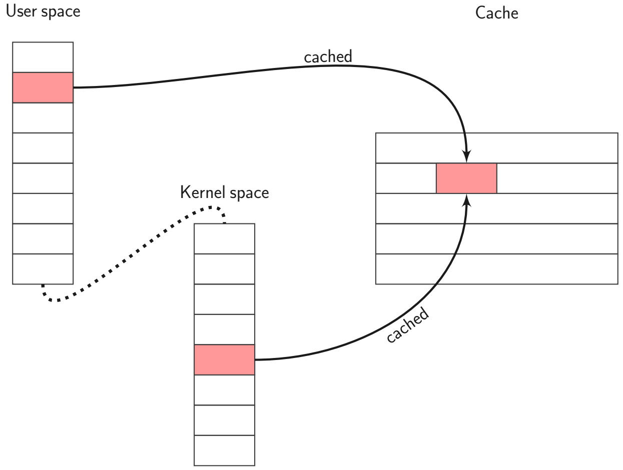


OS X, Linux, BSD, Xen PVM (Amazon EC2)

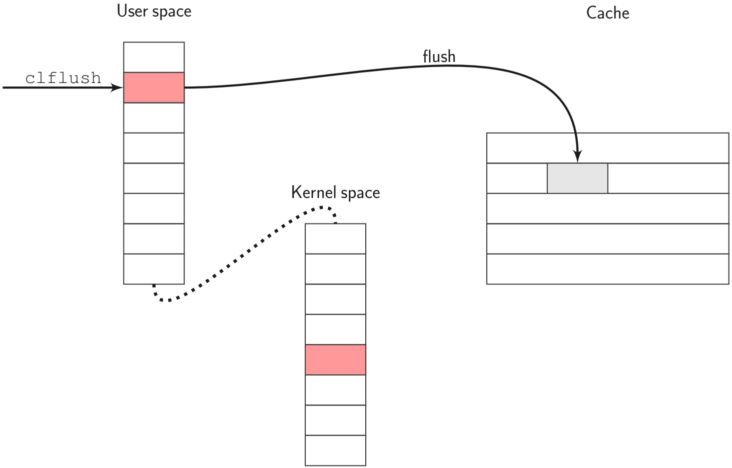
Address-Translation Oracle



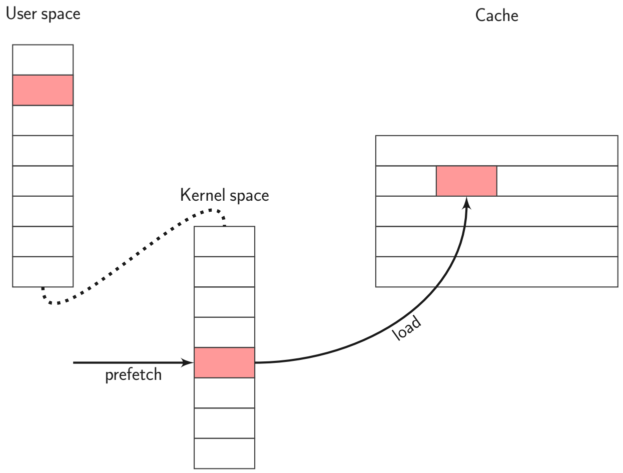
Address-Translation Oracle



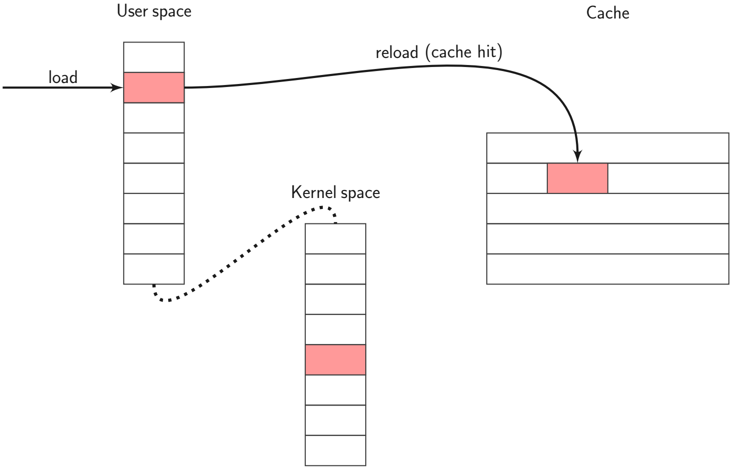
Address-Translation Oracle



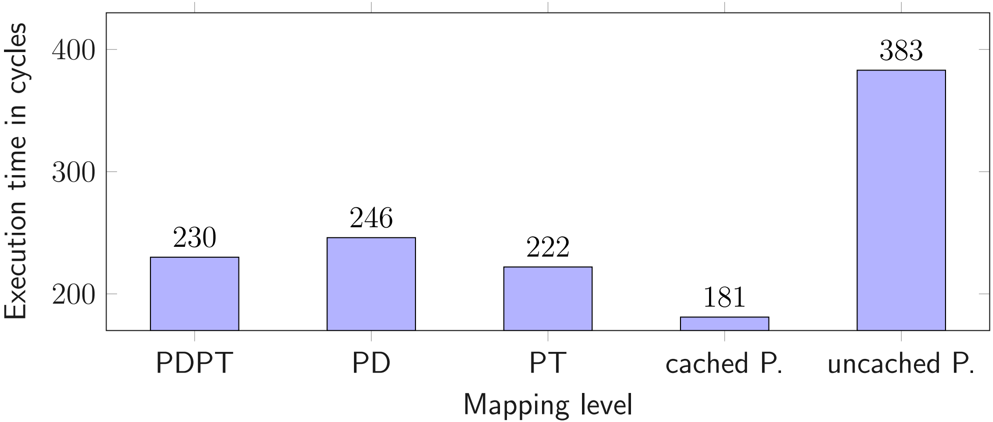
Address-Translation Oracle



Address-Translation Oracle

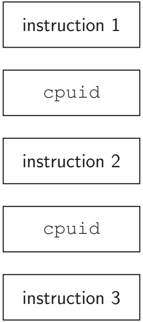


Translation-Level Oracle



Timing the prefetch instruction

The CPU may reorder instructions



instruction 1

`cpuid`

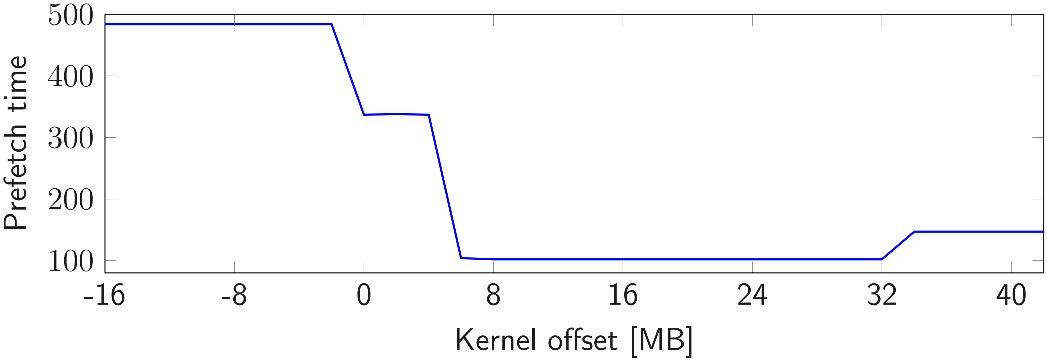
instruction 2

`cpuid`

instruction 3

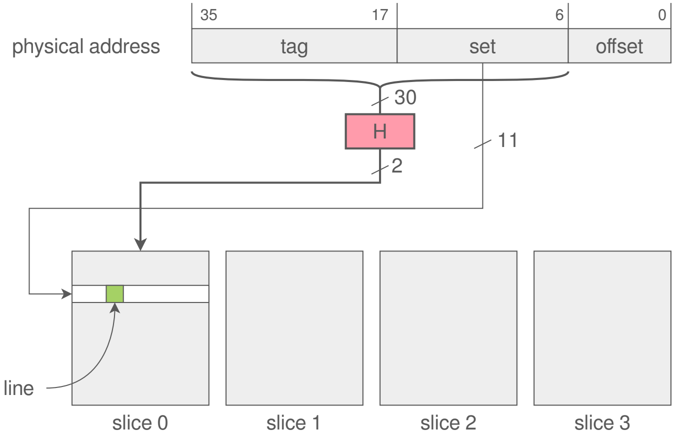
but not over `cpuid`!

Breaking KASLR with Prefetch



Last-level Cache Addressing

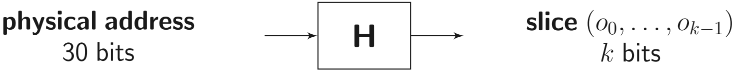
Last-level cache addressing



Last-level cache addressing

- 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:



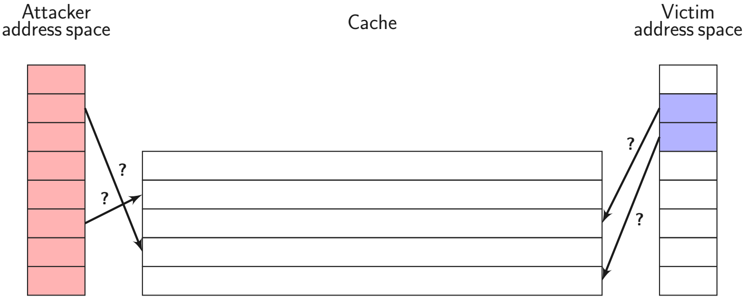
physical address
30 bits

H

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

Prime+Probe on recent procesors?

Complex addressing → impossible to target a set



Reverse engineering method

1. find some way to map one address to one slice

Reverse engineering method

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

Reverse engineering method

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

How to map addresses to slices?

How to map addresses to slices?

- with performance counters

How to map addresses to slices?

- with performance counters
- with a timing attack

How to map addresses to slices?

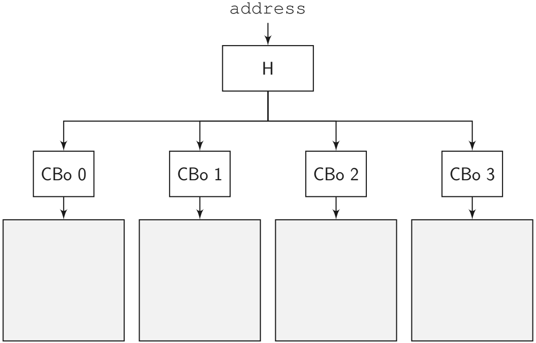
- with performance counters
- with a timing attack
  - using `clflush`

How to map addresses to slices?

- with performance counters
- with a timing attack
    - using `clflush`
    - using memory access

Method #1: Performance counters

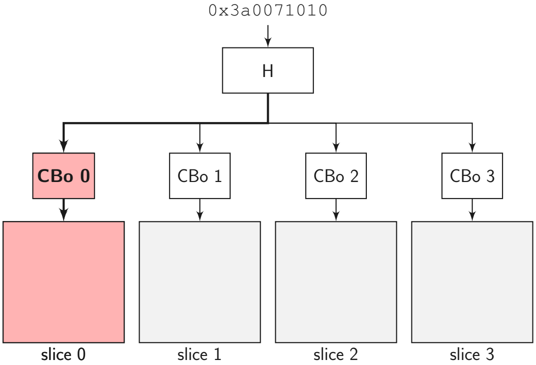
- event UNC_CBO_CACHE_LOOKUP counts accesses to a slice



	slice 0	slice 1	slice 2	slice 3
UNC_CBO_CACHE_LOOKUP	0	0	0	0

Method #1: Performance counters

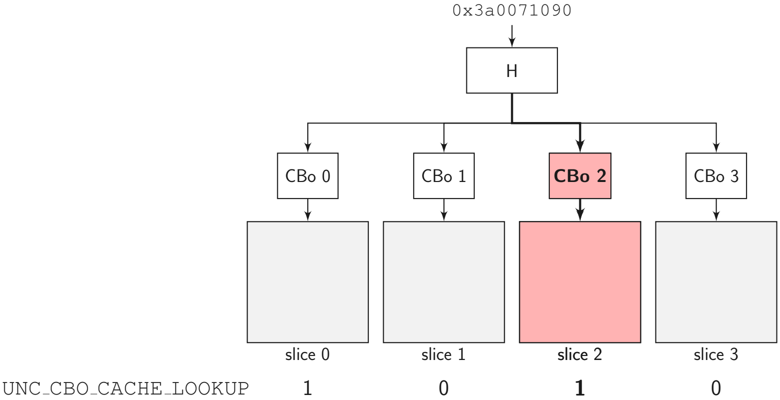
- event UNC_CBO_CACHE_LOOKUP counts accesses to a slice



UNC_CBO_CACHE_LOOKUP	**1**	0	0	0

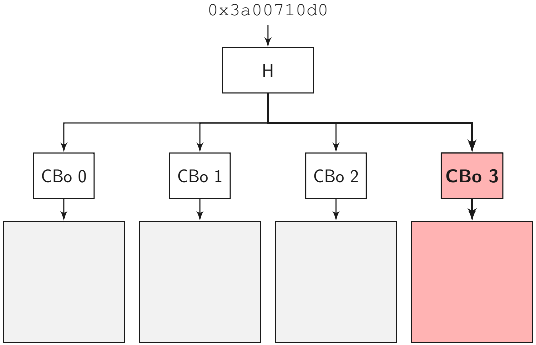
Method #1: Performance counters

- event UNC_CBO_CACHE_LOOKUP counts accesses to a slice



Method #1: Performance counters

- event UNC_CBO_CACHE_LOOKUP counts accesses to a slice



	slice 0	slice 1	slice 2	slice 3
UNC_CBO_CACHE_LOOKUP	1	0	1	**1**

Mapping a physical addr. to a slice [7]

1. translate virtual to physical address with `/proc/pid/pagemap`

Mapping a physical addr. to a slice [7]

1. translate virtual to physical address with `/proc/pid/pagemap`
2. set up monitoring session

Mapping a physical addr. to a slice [7]

1. translate virtual to physical address with `/proc/pid/pagemap`
2. set up monitoring session
3. repeat access to a single address

Mapping a physical addr. to a slice [7]

1. translate virtual to physical address with `/proc/pid/pagemap`
2. set up monitoring session
3. repeat access to a single address

   → hint: `clflush` is already counted as an access

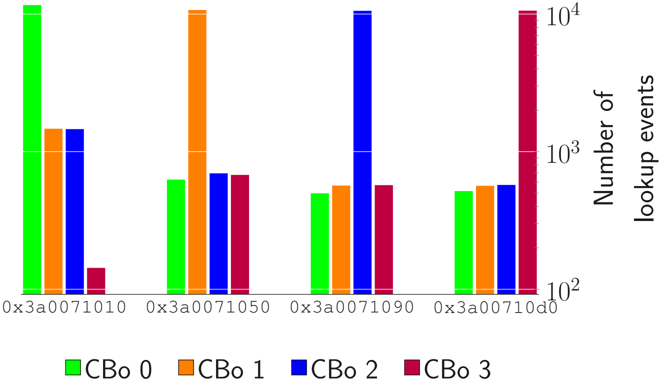
Mapping a physical addr. to a slice [7]

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

Mapping a physical addr. to a slice [7]

1. translate virtual to physical address with `/proc/pid/pagemap`
2. set up monitoring session
3. repeat access to a single address
   → hint: `clflush` is already counted as an access
4. read `UNC_CBO_CACHE_LOOKUP` event for each CBo
5. slice is the one that has the maximum lookup events

Mapping physical addresses to slices



Method #2: Flush+Flush [2]

- side channel similar to Flush+Reload, using only `clflush` execution time differences

Method #2: Flush+Flush [2]

- side channel similar to Flush+Reload, using only `clflush` execution time differences
- time difference between cached and not cached

Method #2: Flush+Flush [2]

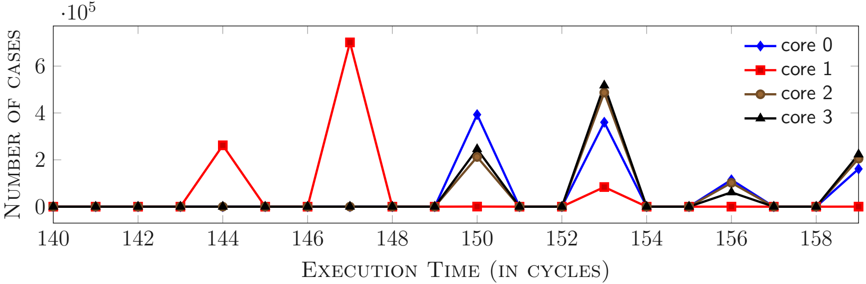
- side channel similar to Flush+Reload, using only `clflush` execution time differences
- time difference between cached and not cached
- time difference between slices

Last-level cache and Ring interconnect

Intel Optimization Manual:

“The LLC hit latency [...] depends on the core location relative to the LLC block, and how far the request needs to travel on the ring.”

`clflush` time difference between slices



`clflush` histogram for an address in slice 1 on different cores

Inferring the function

Two cases:

1. linear function: 2^n number of cores
2. non-linear function: the rest

Methods

- brute-force
- smarter method
- let a solver do it! [1]

Perspective: the first two methods assume that we know what the functions look like (XORs of address bits)

Brute force

For each bit of output $o_0, \ldots, o_{k-1}$

1. try one function

→ e.g., $b_6 \oplus b_7 \oplus \ldots \oplus b_{32}$

2. test if the function corresponds to the mapping you already have

Brute force

For each bit of output $o_0, \ldots, o_{k-1}$

1. try one function

$\rightarrow$ e.g., $b_6 \oplus b_7 \oplus \ldots \oplus b_{32}$

2. test if the function corresponds to the mapping you already have
3. if not → try another function until it works!

Brute force

For each bit of output $o_0, \ldots, o_{k-1}$

1. try one function

$\rightarrow$ e.g., $b_6 \oplus b_7 \oplus \ldots \oplus b_{32}$

2. test if the function corresponds to the mapping you already have
3. if not → try another function until it works!

Fail fast, but still tolerate some errors

Finding linear function (1/2) [7]

- XOR is commutative: $A \oplus B \Leftrightarrow B \oplus A$

Finding linear function (1/2) [7]

- XOR is commutative: $A \oplus B \Leftrightarrow B \oplus A$
- XOR is associative: $A \oplus (B \oplus C) \Leftrightarrow (A \oplus B) \oplus C$

Finding linear function (1/2) [7]

- XOR is commutative: $A \oplus B \Leftrightarrow B \oplus A$
- XOR is associative: $A \oplus (B \oplus C) \Leftrightarrow (A \oplus B) \oplus C$
- you can treat all the input bits independently

Finding linear function (2/2) [7]

For each bit of output $o_0, \ldots, o_{k-1}$

- find two addresses that differ by a single bit b_i
- compare output
  - if different: b_i is part of the function
  - if equal: b_i is not part of the function

Side-Channel Security

Chapter 2: Caches & Cache Attacks

Daniel Gruss, Roland Czerny

March 6, 2025

Graz University of Technology

References

[1] Gerlach, L., Schwarz, S., Faroß, N., and Schwarz, M. (2024). Efficient and Generic Microarchitectural Hash-Function Recovery. In *S&P*.

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

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

[4] Inci, M. S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., and Sunar, B. (2015). Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. *Cryptology ePrint Archive, Report 2015/898*.

[5] Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. (2016). ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*.

[6] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. (2015). Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.

[7] Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., and Francillon, A. (2015). Reverse Engineering Intel Complex Addressing Using Performance Counters. In *RAID*.

[8] Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Alberto Boano, C., Mangard, S., and Römer, K. (2017). Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*.

[9] Yarom, Y., Ge, Q., Liu, F., Lee, R. B., and Heiser, G. (2015). Mapping the Intel Last-Level Cache. *Cryptology ePrint Archive, Report 2015/905*.