# Side-Channel Security

## Chapter 2: Cache Template Attacks

**Daniel Gruss**

March 9, 2021

Graz University of Technology

# 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

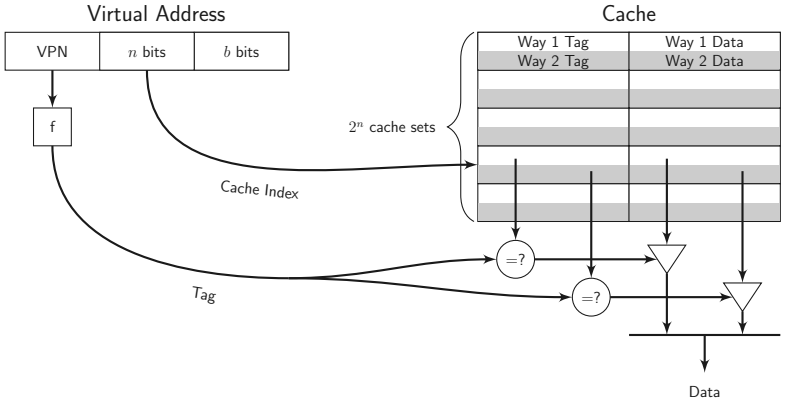
# TLB and Paging

- 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

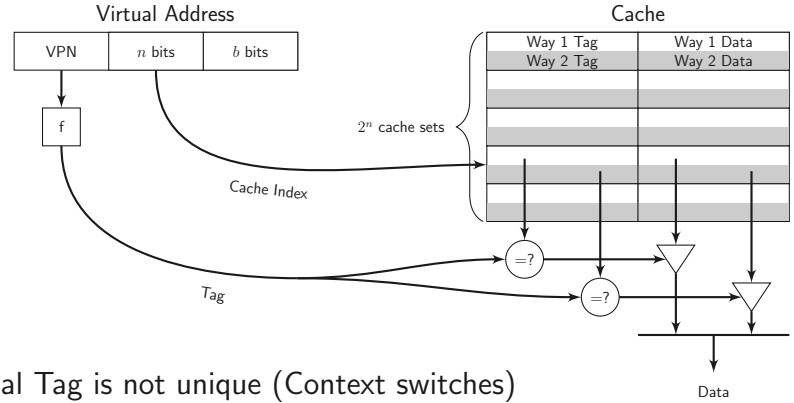
# Cache indexing methods

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

# VIVT

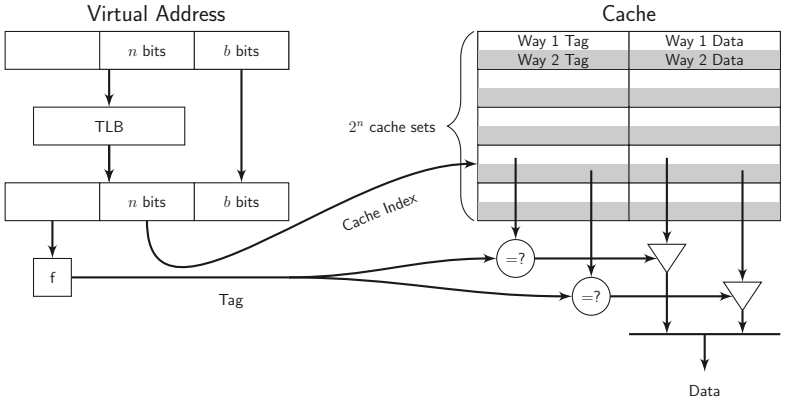

Virtual Address
VPN
n bits
b bits
f
Cache
Way 1 Tag
Way 2 Tag
Way 1 Data
Way 2 Data
2^n cache sets
Cache Index
Tag
=?
=?
Data

# VIVT

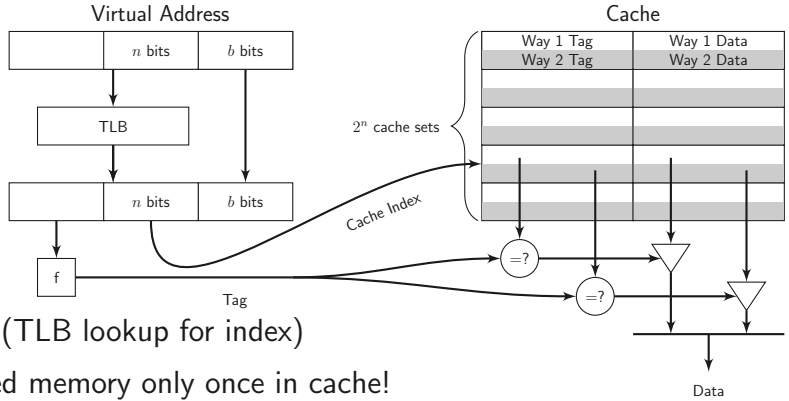



- Fast
- Virtual Tag is not unique (Context switches)
- Shared memory more than once in cache

# PIPT

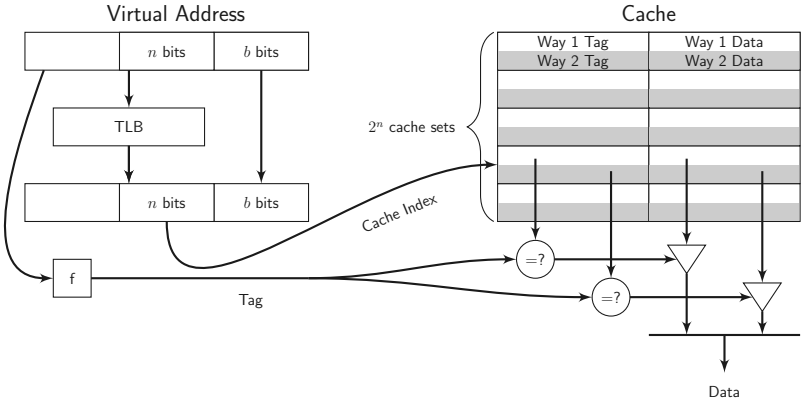

Virtual Address
n bits
b bits
TLB
n bits
b bits
f
Tag
Cache Index
Cache
Way 1 Tag
Way 1 Data
Way 2 Tag
Way 2 Data
2^n cache sets
=?
=?
Data

# PIPT

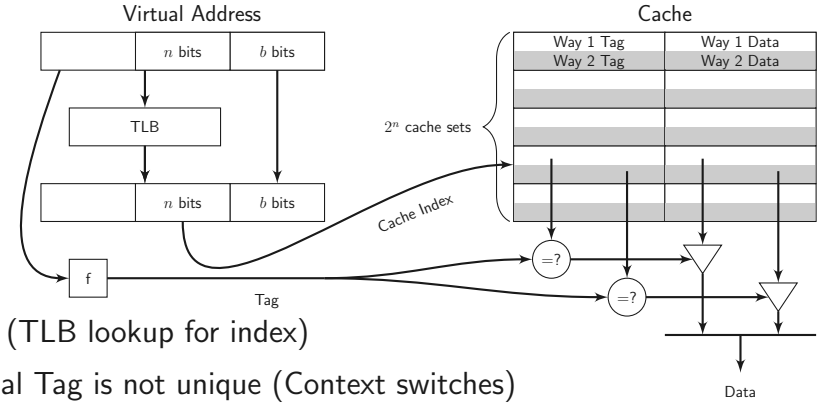



- Slow (TLB lookup for index)
- Shared memory only once in cache!

# (PIVT)

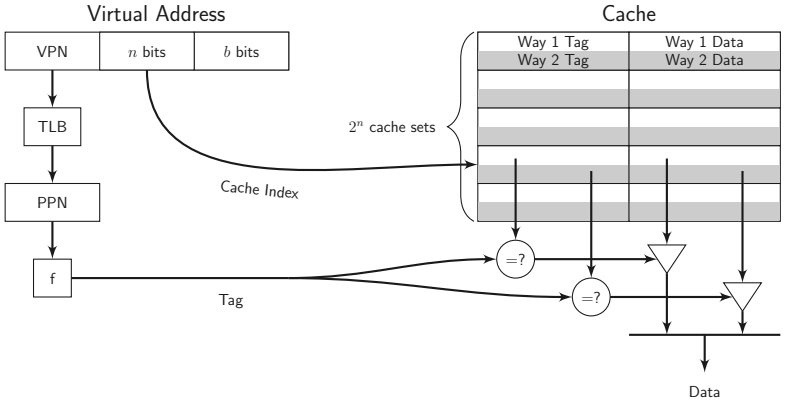

Virtual Address
n bits
b bits
TLB
n bits
b bits
f
Tag
Cache Index
2^n cache sets
Cache
Way 1 Tag
Way 2 Tag
Way 1 Data
Way 2 Data
=?
=?
Data

# (PIVT)

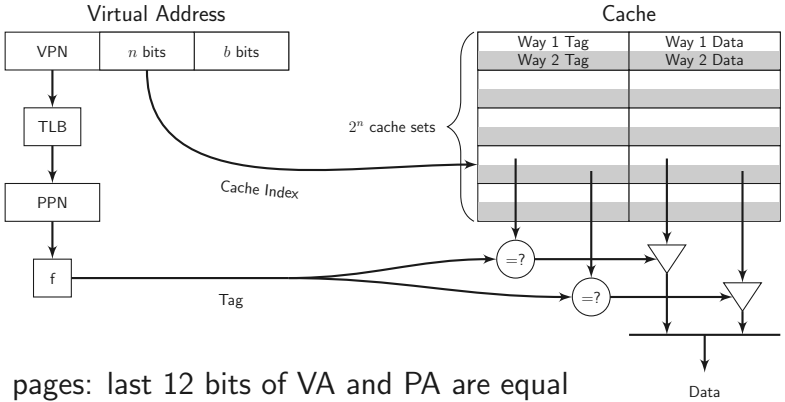



- Slow (TLB lookup for index)
- Virtual Tag is not unique (Context switches)
- Shared memory more than once in cache

# VIPT


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

# VIPT



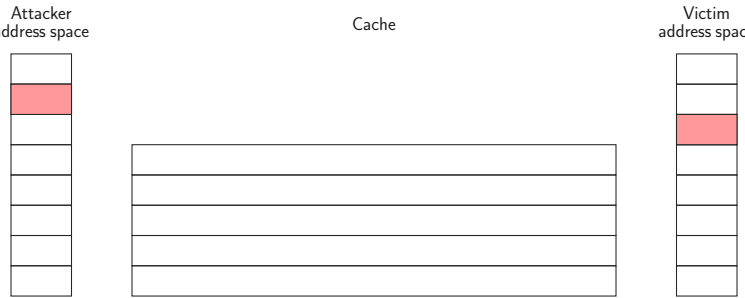

- Fast
- 4 KiB pages: last 12 bits of VA and PA are equal
- Using more bits is unpractical (like VIVT)

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

# Remarks

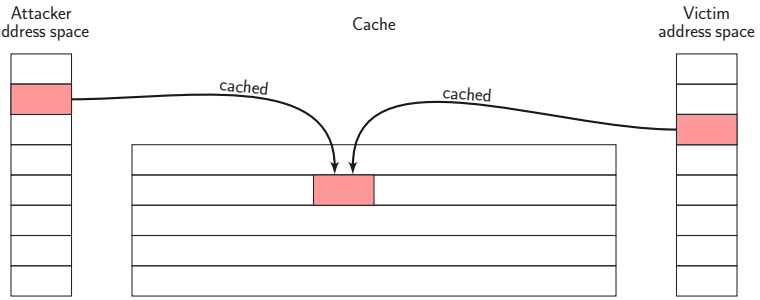
- L1 caches: VIVT or VIPT
- L2/L3 caches: PIPT

# Flush+Reload



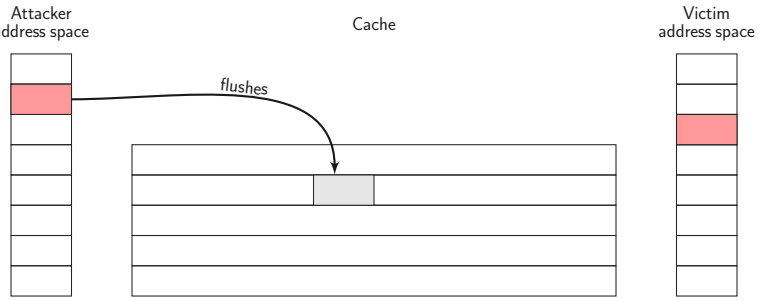

**step 0**: attacker maps shared library → shared memory, shared in cache

# Flush+Reload




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

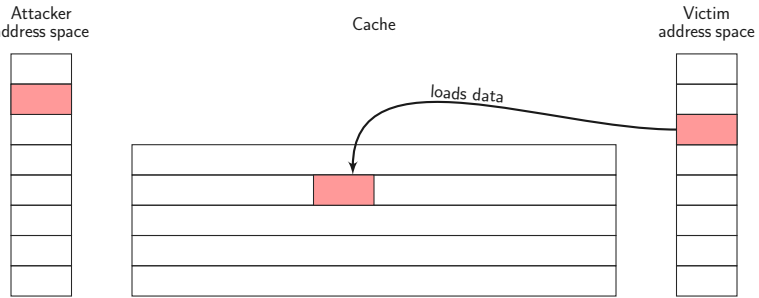
# Flush+Reload




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

**step 1**: attacker flushes the shared line

# Flush+Reload

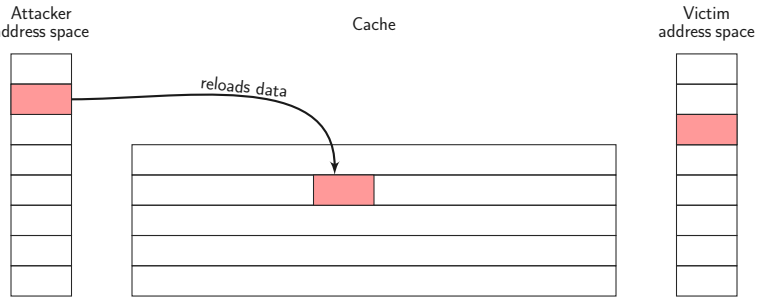



**step 0**: attacker maps shared library → shared memory, shared in cache

**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)

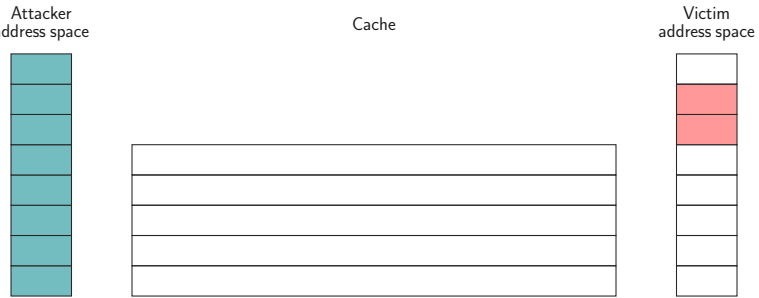
Cons: restrictive

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

# Variants of Flush+Reload

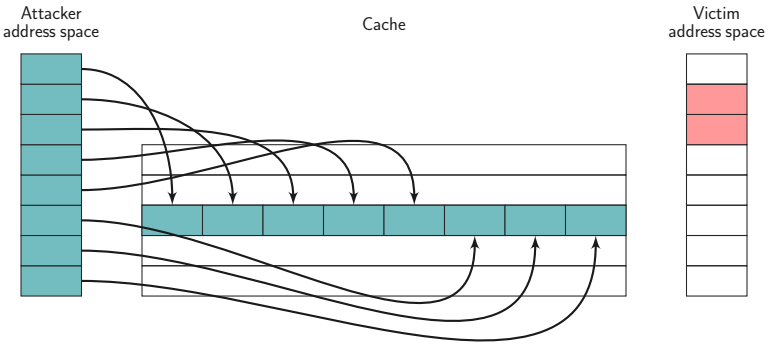
- Flush+Flush [1]
- Evict+Reload [2] on ARM [4]

# Prime+Probe



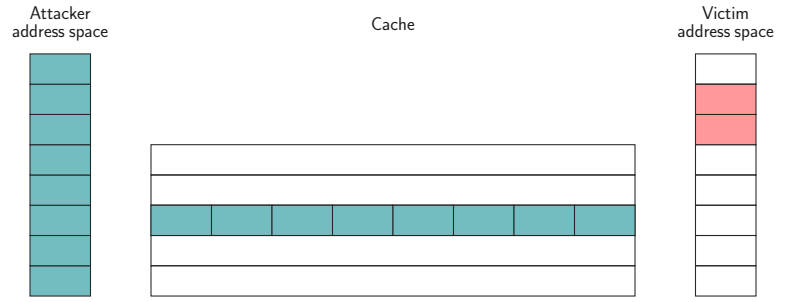

**step 0**: attacker fills the cache (prime)

# Prime+Probe



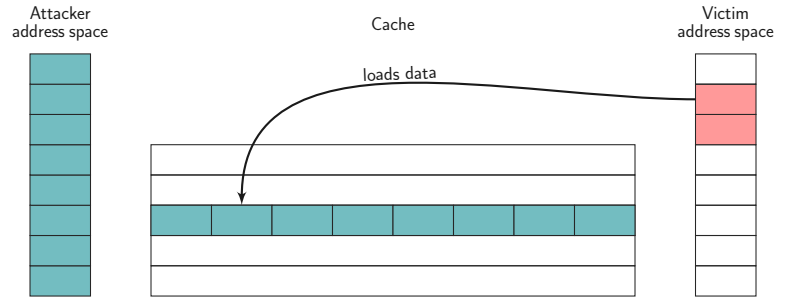

**step 0**: attacker fills the cache (prime)

# Prime+Probe




**step 0**: attacker fills the cache (prime)

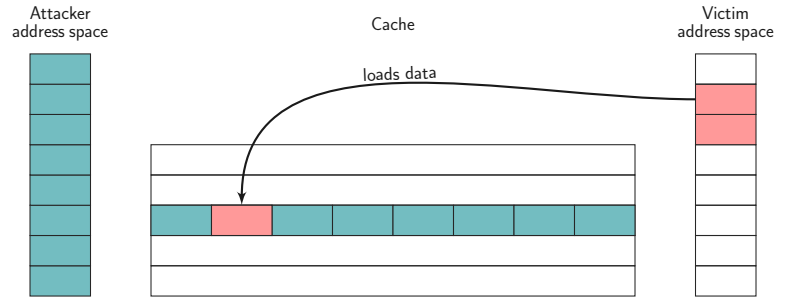
# Prime+Probe




**step 0**: attacker fills the cache (prime)

**step 1**: victim evicts cache lines while performing encryption

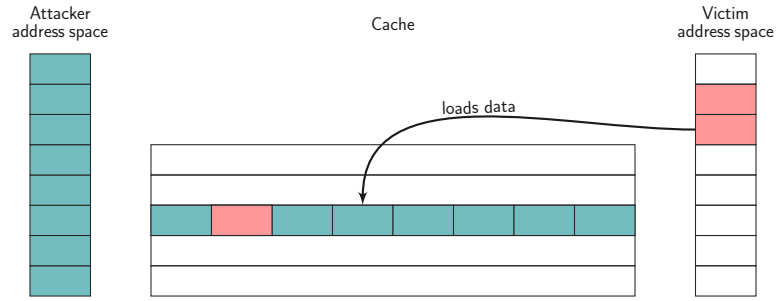
# Prime+Probe




**step 0**: attacker fills the cache (prime)

**step 1**: victim evicts cache lines while performing encryption

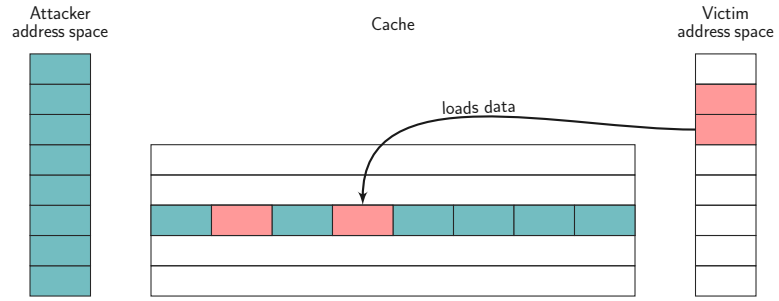
# Prime+Probe




**step 0**: attacker fills the cache (prime)

**step 1**: victim evicts cache lines while performing encryption

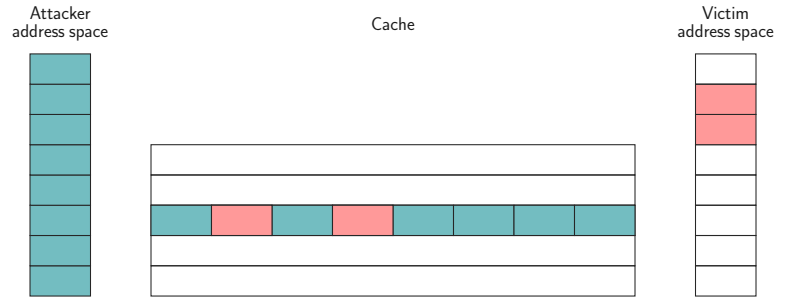
# Prime+Probe




**step 0**: attacker fills the cache (prime)

**step 1**: victim evicts cache lines while performing encryption

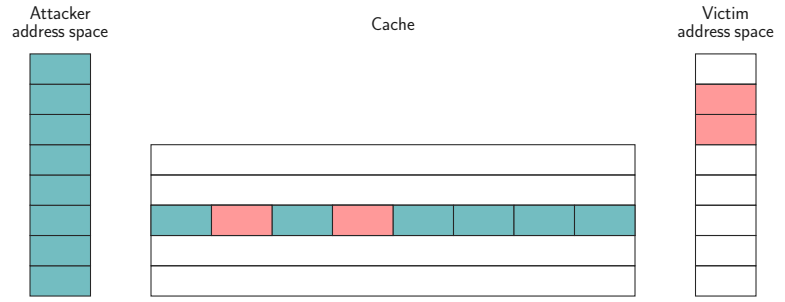
# Prime+Probe




**step 0**: attacker fills the cache (prime)

**step 1**: victim evicts cache lines while performing encryption

# Prime+Probe

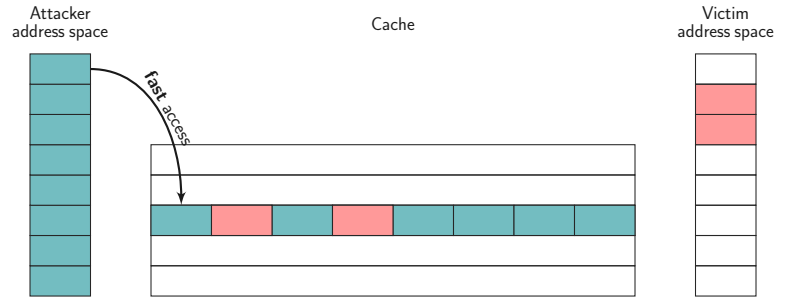



**step 0**: attacker fills the cache (prime)

**step 1**: victim evicts cache lines while performing encryption

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

# Prime+Probe

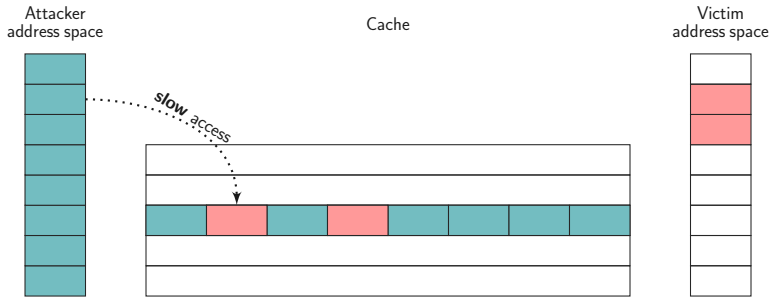



**step 0**: attacker fills the cache (prime)

**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

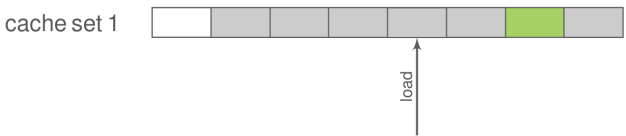
Cons: coarser granularity (1 set)

# 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?

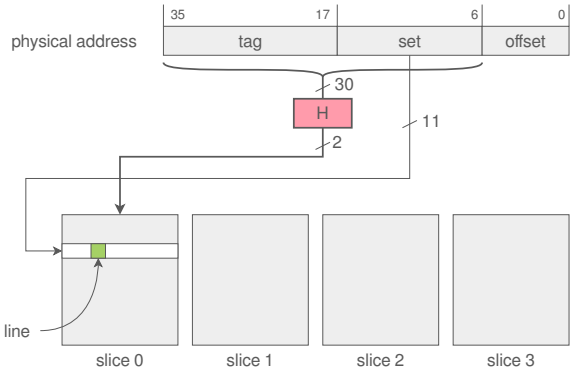
# #1.1: Which physical addresses to access?




"LRU eviction":

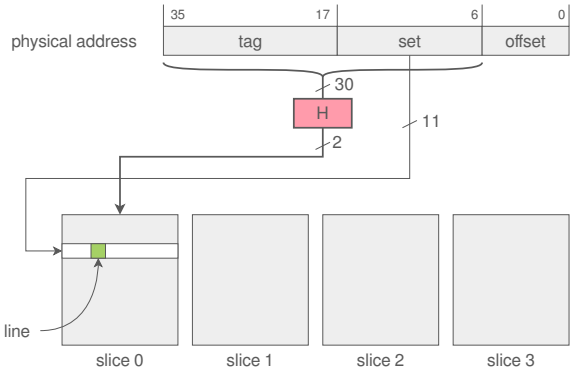
- assume that cache uses LRU replacement
- accessing $n$ addresses from the same cache set to evict an $n$-way set
- eviction from last level $\rightarrow$ from whole hierarchy (it's inclusive!)

# #1.2: Which addresses map to the same set?




- function H that maps slices is undocumented
- reverse-engineered by [3, 6, 8]

# #1.2: Which addresses map to the same set?




- function H that maps slices is undocumented
- reverse-engineered by [3, 6, 8]
- 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 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 |
| 2 cores | $o_0$ | | | | | | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | | | | ⊕ |
| 4 cores | $o_0$ | | | | | ⊕ | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | | | | ⊕ |
| | $o_1$ | | | | ⊕ | ⊕ | | ⊕ | | ⊕ | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | | ⊕ | | | | ⊕ | |
| 8 cores | $o_0$ | | ⊕ | ⊕ | | ⊕ | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | | | | ⊕ |
| | $o_1$ | ⊕ | | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | | ⊕ | | | | ⊕ | |
| | $o_2$ | ⊕ | ⊕ | ⊕ | ⊕ | | | ⊕ | ⊕ | | | ⊕ | ⊕ | | | ⊕ | ⊕ | | | ⊕ | | | ⊕ | | | ⊕ | ⊕ | | | | ⊕ | | |

# #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 $\rightarrow$ 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 $\rightarrow$ lowest 21 bits are the same as virtual address

$\rightarrow$ enough to compute the cache set

# #3.1: Replacement policy on older CPUs

"LRU eviction" memory accesses

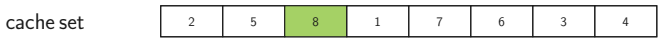
# #3.1: Replacement policy on older CPUs

"LRU eviction" memory accesses


- LRU replacement policy: oldest entry first

# #3.1: Replacement policy on older CPUs

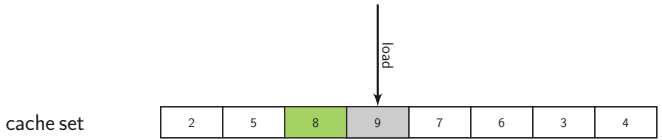
"LRU eviction" memory accesses




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

# #3.1: Replacement policy on older CPUs

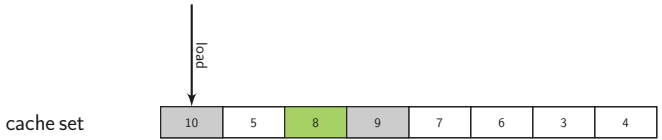
"LRU eviction" memory accesses




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

# #3.1: Replacement policy on older CPUs

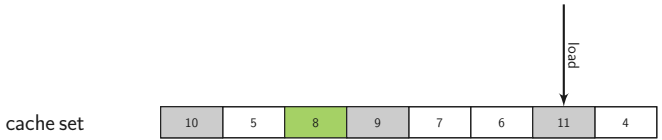
"LRU eviction" memory accesses




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

# #3.1: Replacement policy on older CPUs

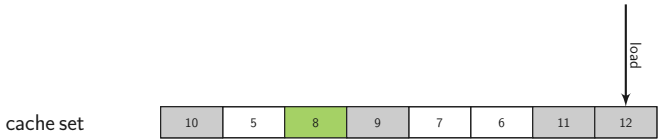
"LRU eviction" memory accesses




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

# #3.1: Replacement policy on older CPUs

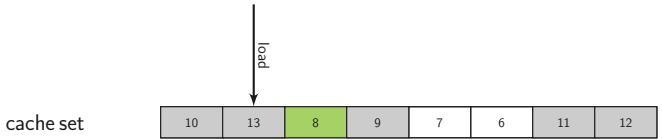
"LRU eviction" memory accesses




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

# #3.1: Replacement policy on older CPUs

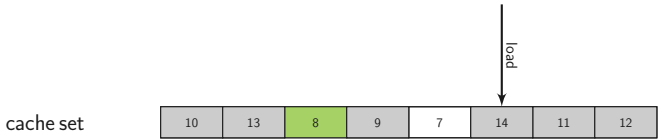
"LRU eviction" memory accesses




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

# #3.1: Replacement policy on older CPUs

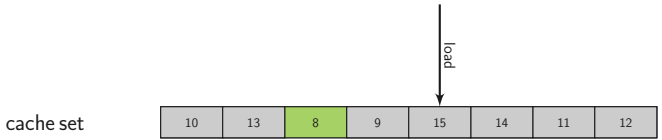
"LRU eviction" memory accesses




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

# #3.1: Replacement policy on older CPUs

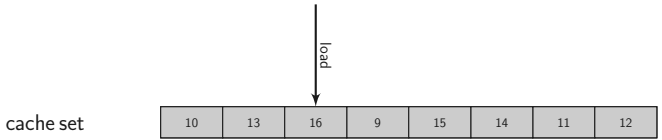
"LRU eviction" memory accesses




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

# #3.1: Replacement policy on older CPUs

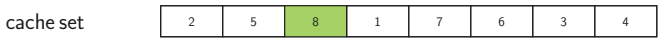
"LRU eviction" memory accesses




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

# #3.2: Replacement policy on recent CPUs

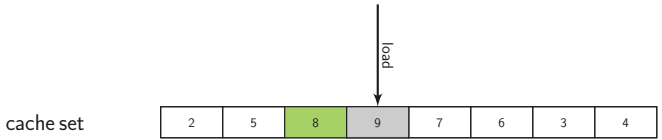
"LRU eviction" memory accesses




- no LRU replacement

# #3.2: Replacement policy on recent CPUs

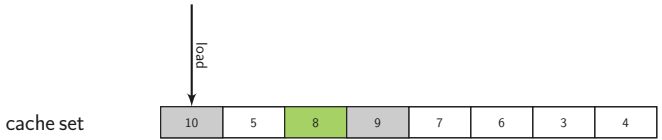
"LRU eviction" memory accesses




- no LRU replacement

# #3.2: Replacement policy on recent CPUs

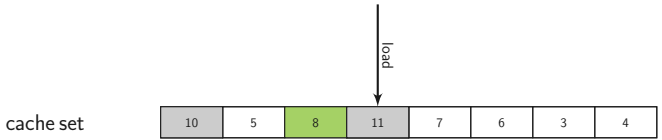
"LRU eviction" memory accesses




- no LRU replacement

# #3.2: Replacement policy on recent CPUs

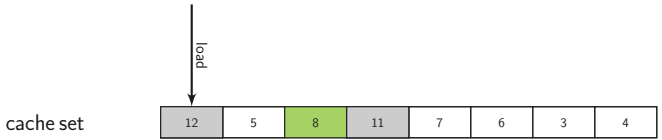
"LRU eviction" memory accesses




- no LRU replacement

# #3.2: Replacement policy on recent CPUs

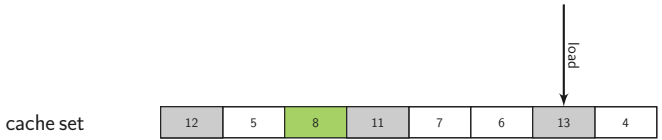
"LRU eviction" memory accesses




- no LRU replacement

# #3.2: Replacement policy on recent CPUs

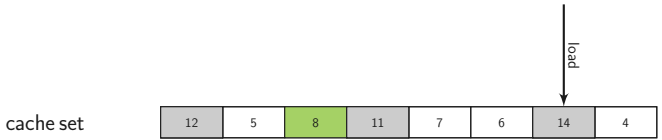
"LRU eviction" memory accesses




- no LRU replacement

# #3.2: Replacement policy on recent CPUs

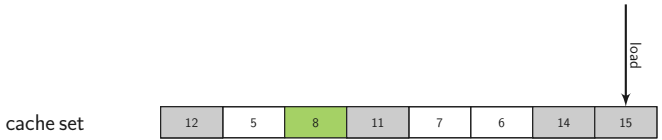
"LRU eviction" memory accesses




- no LRU replacement

# #3.2: Replacement policy on recent CPUs

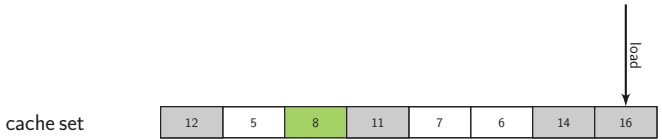
"LRU eviction" memory accesses




- no LRU replacement

# #3.2: Replacement policy on recent CPUs

"LRU eviction" memory accesses




- no LRU replacement

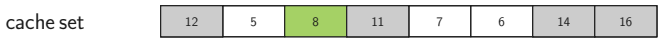
# #3.2: Replacement policy on recent CPUs

"LRU eviction" memory accesses


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

# #3.2: Replacement policy on recent CPUs

"LRU eviction" memory accesses




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

# #3.3: Cache eviction strategy

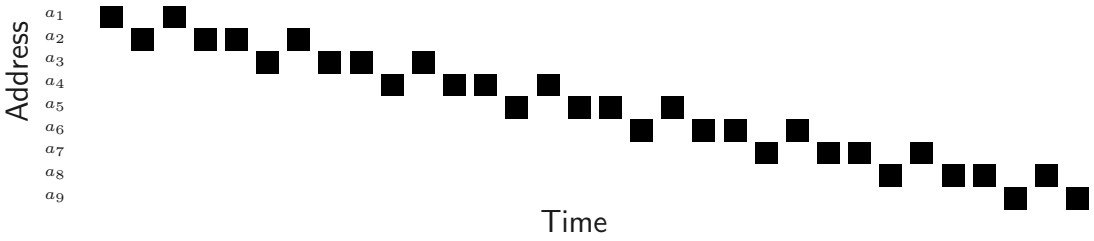



**Figure 1:** Fast and effective on Haswell. Eviction rate >99.97%.

# 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 $\rightarrow$ low access time in receiver
  - 1: sender does access the set $\rightarrow$ 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? $\rightarrow$ 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

$\rightarrow$ higher performance

# Multi-bit covert channels

- combine multiple 1-bit channels
- avoid interferences

$\rightarrow$ higher performance

- use 1-bit for sending = true/false

# Packets / frames

Organize data in packets / frames:

- some data bits
- check sum
- sequence number

$\rightarrow$ keep sender and receiver synchronous

$\rightarrow$ 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?

$\rightarrow$ sender should retransmit until receiver acknowledged

# Raw capacity $C$

- number of bits per second
- measure over $\geq 1$ 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:

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

or if $b_r = b_s$:

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

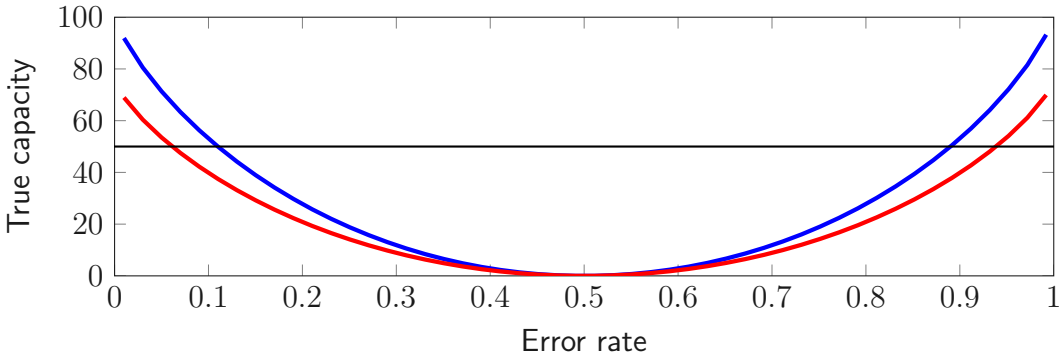
# Capacity

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 [1] | 3968Kbps | 0.840% | 3690Kbps | native |
| F+R [1] | 2384Kbps | 0.005% | 2382Kbps | native |
| E+R [4] | 1141Kbps | 1.100% | 1041Kbps | native |
| P+P [7] | 601Kbps | 0.000% | 601Kbps | native |
| P+P [5] | 600Kbps | 1.000% | 552Kbps | virt |
| P+P [7] | 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

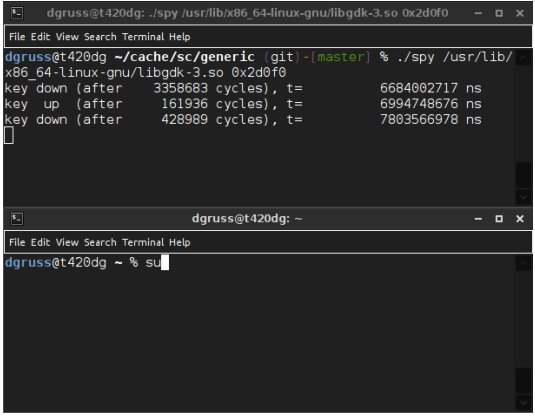
# Cache Template Attacks

- 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

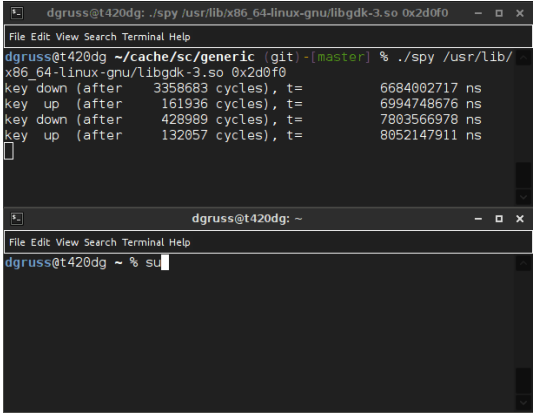
# Cache Template Attacks

- 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

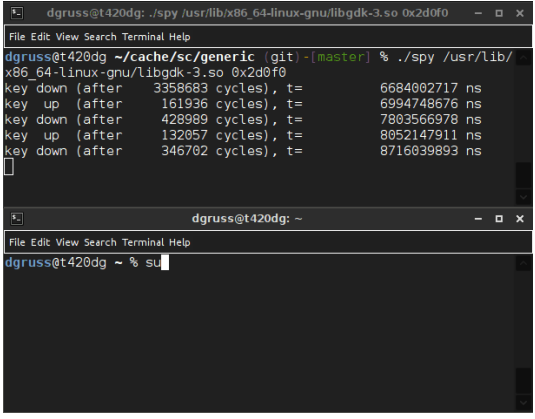
# Can We Build a Cache-Based Keylogger?



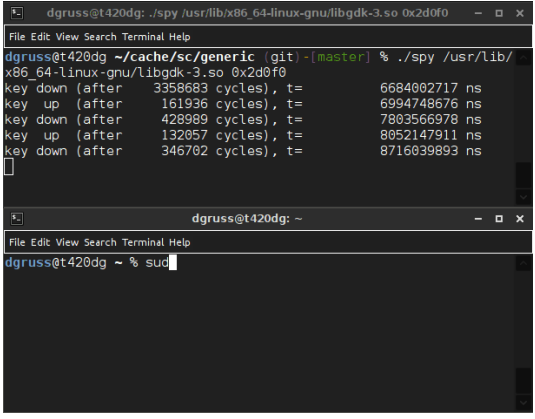
# Can We Build a Cache-Based Keylogger?



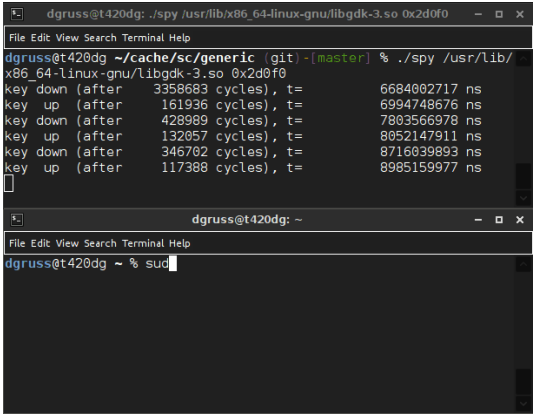
# Can We Build a Cache-Based Keylogger?



# Can We Build a Cache-Based Keylogger?



# Can We Build a Cache-Based Keylogger?



# 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

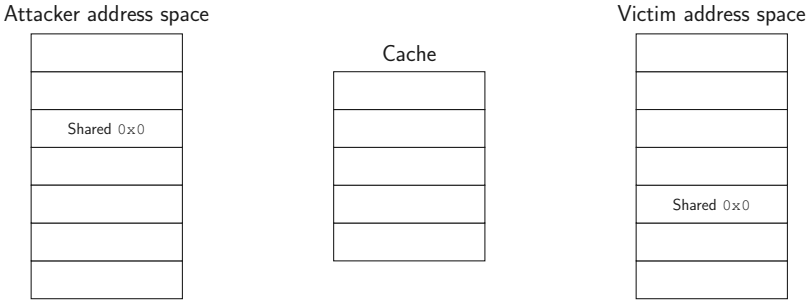
Profiling Phase

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

Exploitation Phase

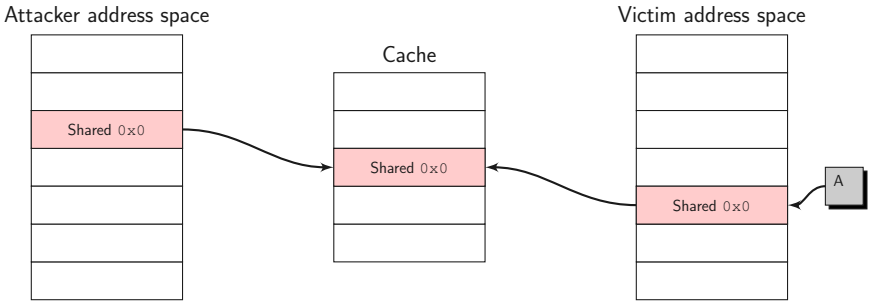
- Monitor exploitable addresses

# Profiling Phase



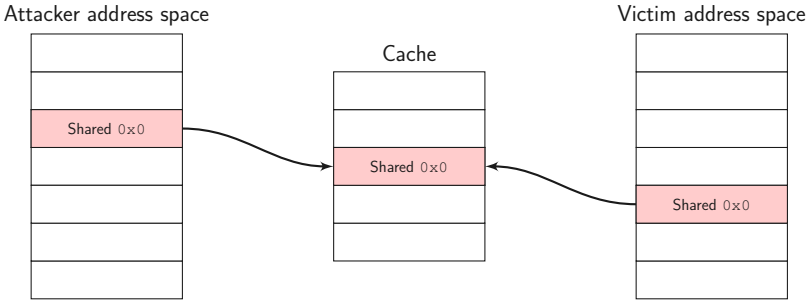

Cache is empty

# Profiling Phase



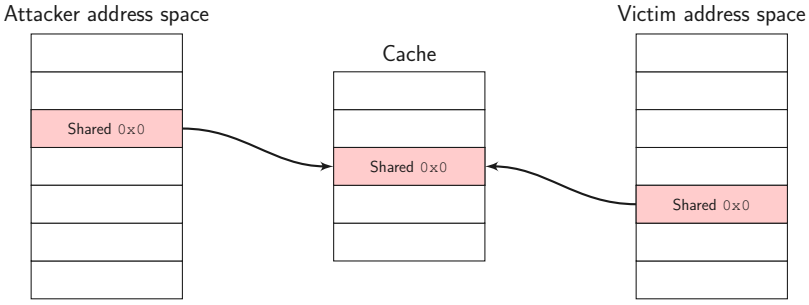

Attacker triggers an event

# Profiling Phase



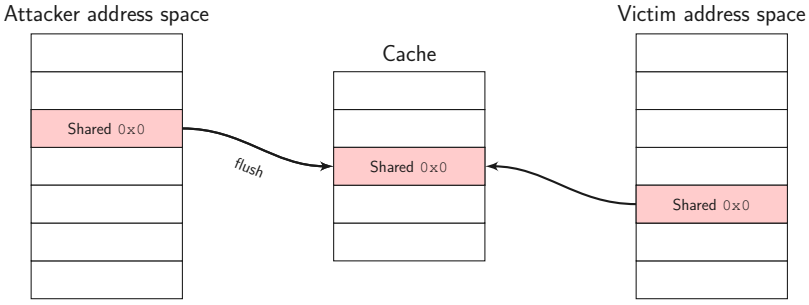

Attacker checks one address for cache hits ("Reload")

# Profiling Phase



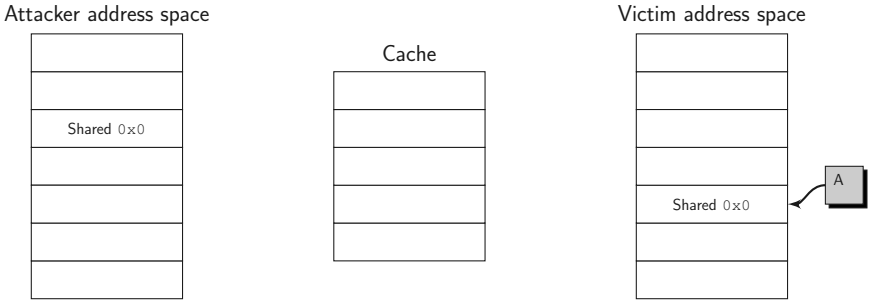

Update cache hit ratio (per event and address)

# Profiling Phase



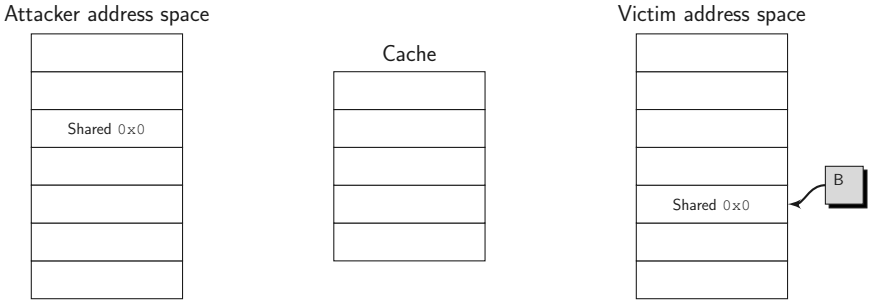

Attacker flushes shared memory

# Profiling Phase



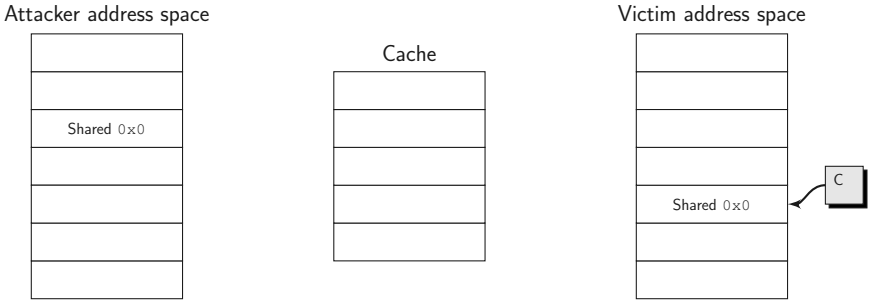

Repeat for higher accuracy

# Profiling Phase



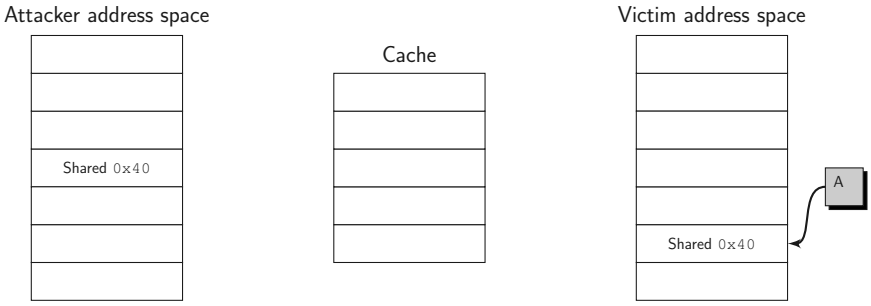

Repeat for all events

# Profiling Phase



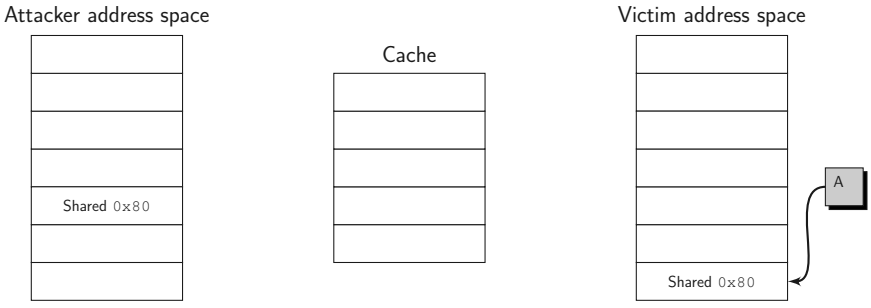

Repeat for all events

# Profiling Phase

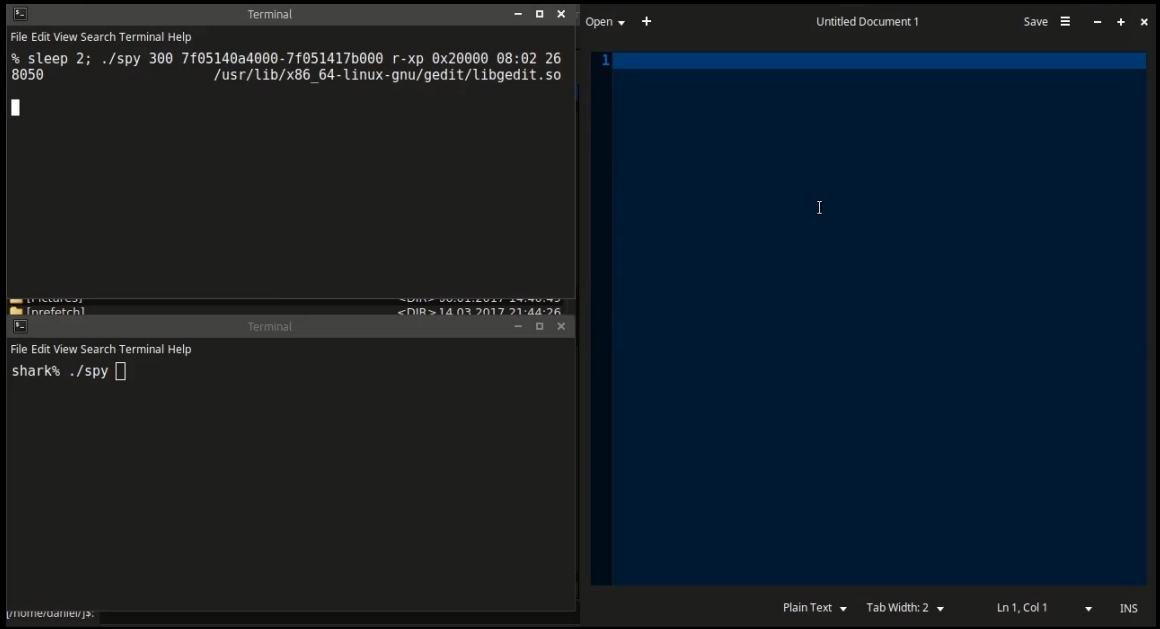



Continue with next address

# Profiling Phase

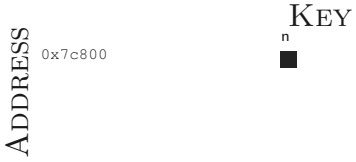



Continue with next address

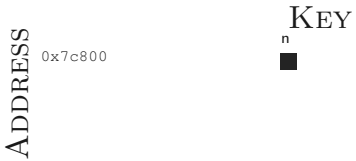



Cache Template Attack Demo

# 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

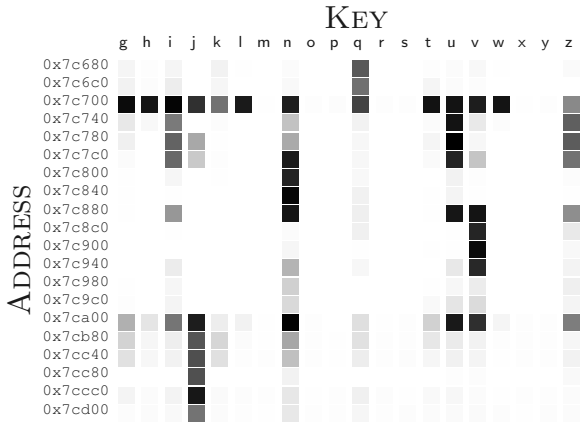
# Profiling Phase: All Events, 1 Address


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 Attack

AES T-Table implementation from OpenSSL 1.0.2

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

# Attack 4: AES T-Table Template Attack

AES T-Table implementation from OpenSSL 1.0.2

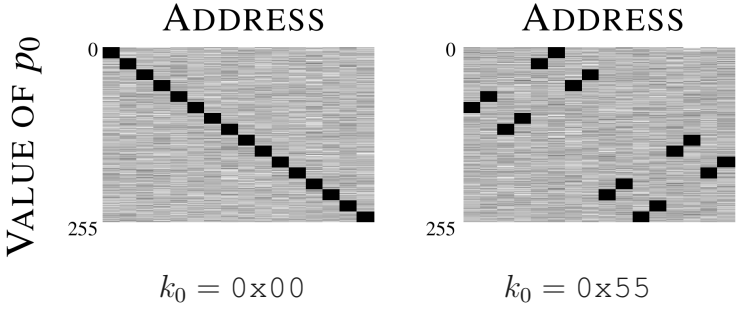
- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event
- Exploitation phase:
  - Eliminate key candidates
  - Reduction of key space in first-round attack:
    - 64 bits after 16–160 encryptions

# Attack 4: AES T-Table Template Attack

AES T-Table implementation from OpenSSL 1.0.2

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

# Attack 4: AES T-Table Template




(transposed)

# Conclusion

- Novel technique to find any cache side-channel leakage
  - Attacks
  - Detect vulnerabilities

# Conclusion

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

# Conclusion

- Novel 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

- Novel 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

# 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 [4]

# Flush+Flush: Motivation

- cache attacks $\rightarrow$ many cache misses
- detect via performance counters

# Flush+Flush: Motivation

- cache attacks $\rightarrow$ many cache misses
- detect via performance counters

$\rightarrow$ good idea, but is it good enough?

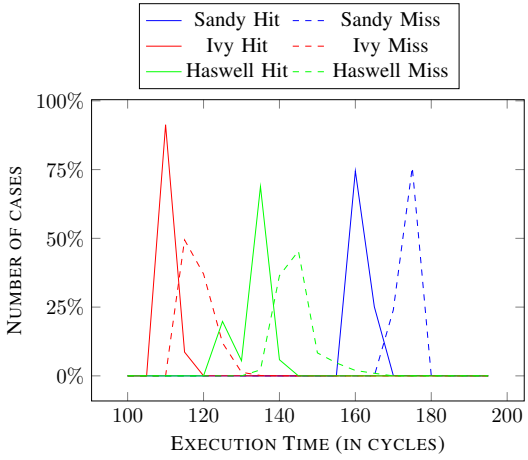
# Flush+Flush: Motivation

- cache attacks $\rightarrow$ many cache misses
- detect via performance counters

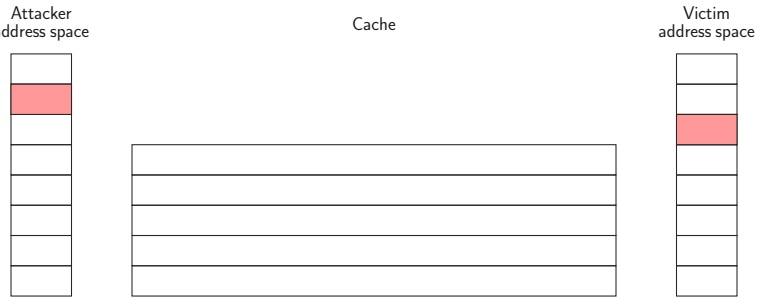
$\rightarrow$ good idea, but is it good enough?

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

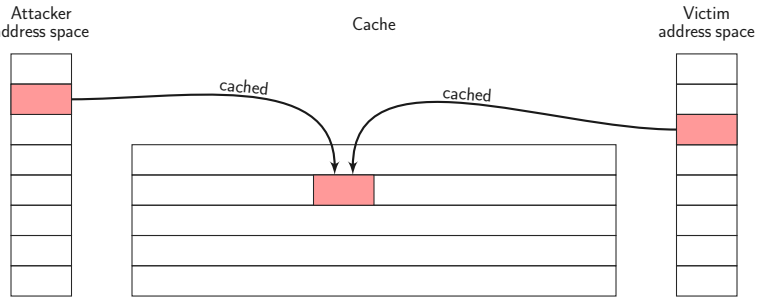
# `clflush` execution time



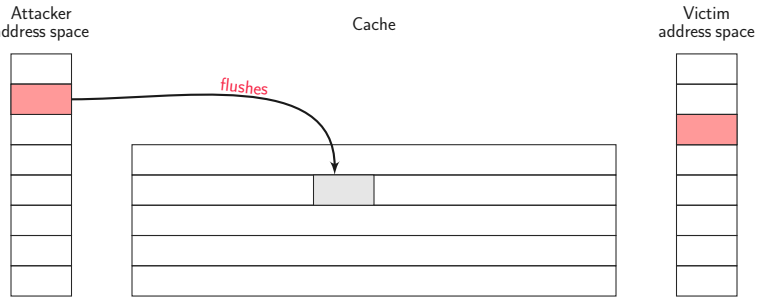
# Flush+Flush



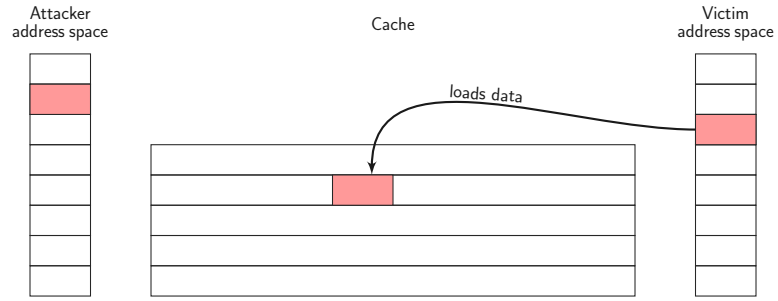
# Flush+Flush


Attacker
address space
Cache
Victim
address space
cached
cached

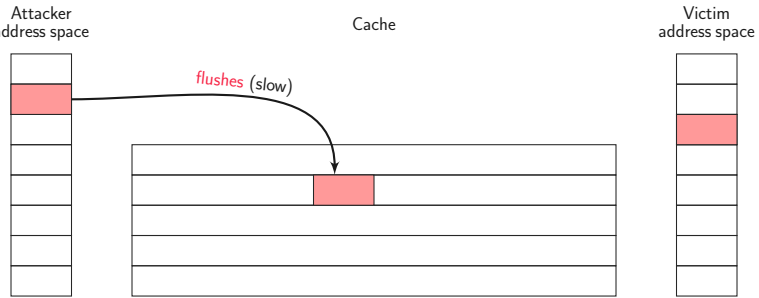
# Flush+Flush


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
  - $\rightarrow$ fast
  - $\rightarrow$ stealthy

# Flush+Flush: Conclusion

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

# Flush+Flush: Conclusion

- attacker causes no direct cache misses
  - $\rightarrow$ fast
  - $\rightarrow$ 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

$\rightarrow$ why?

# ARMageddon in a nutshell

1. no flush instruction

# ARMageddon in a nutshell

1. no flush instruction $\rightarrow$ Evict+Reload

# ARMageddon in a nutshell

1. no flush instruction $\rightarrow$ Evict+Reload
2. pseudo-random replacement

# ARMageddon in a nutshell

1. no flush instruction $\rightarrow$ Evict+Reload
2. pseudo-random replacement $\rightarrow$ eviction strategies from Rowhammer.js

# ARMageddon in a nutshell

1. no flush instruction $\rightarrow$ Evict+Reload
2. pseudo-random replacement $\rightarrow$ eviction strategies from Rowhammer.js
3. cycle counters require root

# ARMageddon in a nutshell

1. no flush instruction $\rightarrow$ Evict+Reload
2. pseudo-random replacement $\rightarrow$ eviction strategies from Rowhammer.js
3. cycle counters require root $\rightarrow$ new timing methods

# ARMageddon in a nutshell

1. no flush instruction $\rightarrow$ Evict+Reload
2. pseudo-random replacement $\rightarrow$ eviction strategies from Rowhammer.js
3. cycle counters require root $\rightarrow$ new timing methods
4. last-level caches not inclusive

# ARMageddon in a nutshell

1. no flush instruction $\rightarrow$ Evict+Reload
2. pseudo-random replacement $\rightarrow$ eviction strategies from Rowhammer.js
3. cycle counters require root $\rightarrow$ new timing methods
4. last-level caches not inclusive $\rightarrow$ let L1 spill to LLC

# ARMageddon in a nutshell

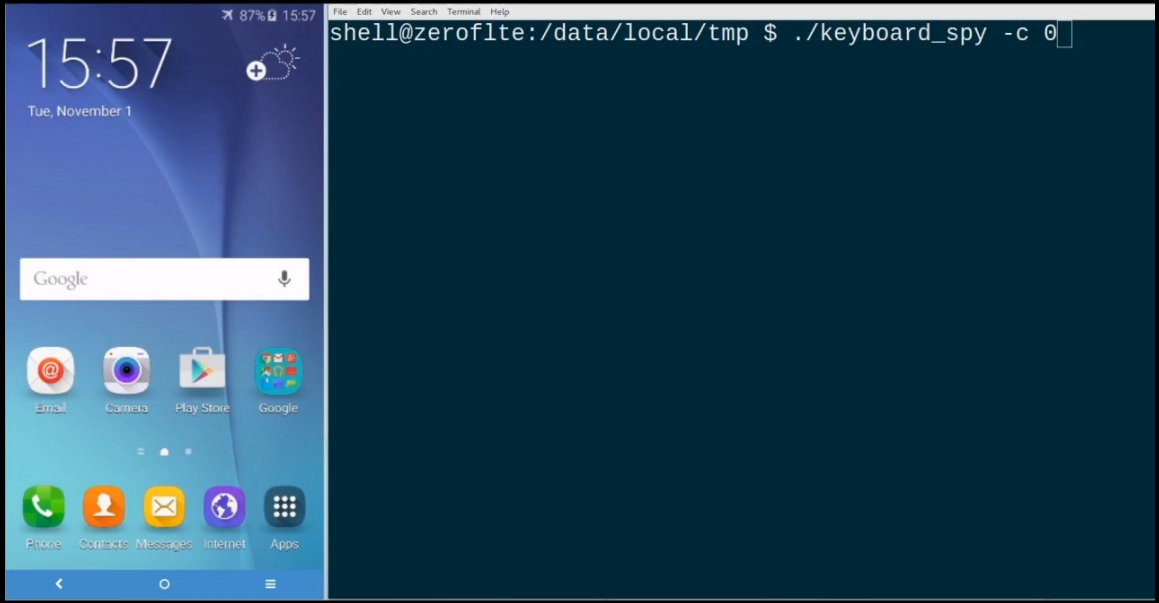
1. no flush instruction $\rightarrow$ Evict+Reload
2. pseudo-random replacement $\rightarrow$ eviction strategies from Rowhammer.js
3. cycle counters require root $\rightarrow$ new timing methods
4. last-level caches not inclusive $\rightarrow$ let L1 spill to LLC
5. multiple CPUs

# ARMageddon in a nutshell

1. no flush instruction $\rightarrow$ Evict+Reload
2. pseudo-random replacement $\rightarrow$ eviction strategies from Rowhammer.js
3. cycle counters require root $\rightarrow$ new timing methods
4. last-level caches not inclusive $\rightarrow$ let L1 spill to LLC
5. multiple CPUs $\rightarrow$ remote fetches + flushes

# ARMageddon in a nutshell

1. no flush instruction $\rightarrow$ Evict+Reload
2. pseudo-random replacement $\rightarrow$ eviction strategies from Rowhammer.js
3. cycle counters require root $\rightarrow$ new timing methods
4. last-level caches not inclusive $\rightarrow$ let L1 spill to LLC
5. multiple CPUs $\rightarrow$ remote fetches + flushes




ARMageddon Demo

# 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**

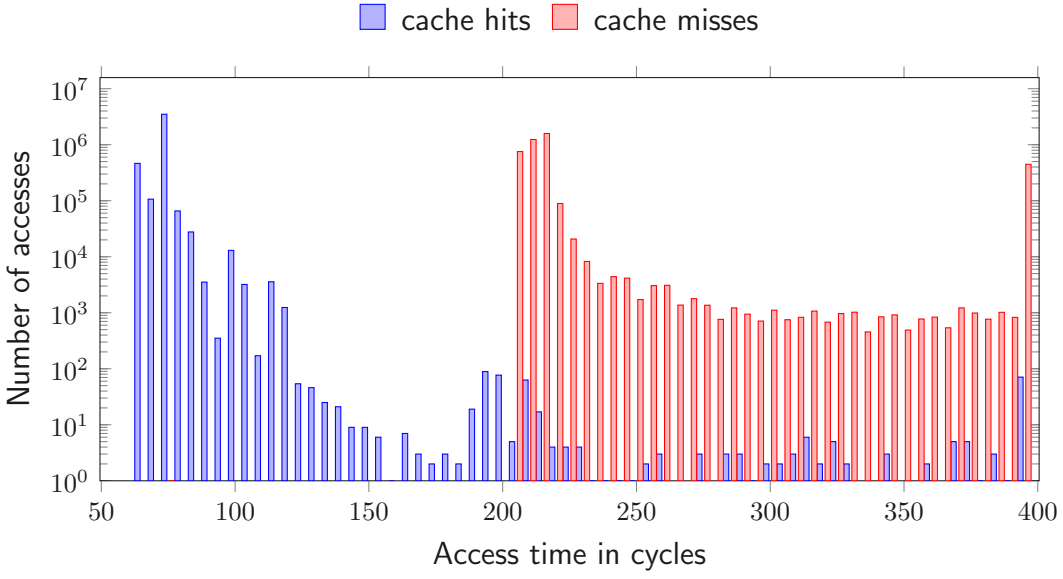
Using the PREFETCH instruction is recommended only if data does not fit in cache. Use of software prefetch should be limited to memory addresses that are managed or owned within the application context. Prefetching to addresses that are not mapped to physical pages can experience non-deterministic performance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

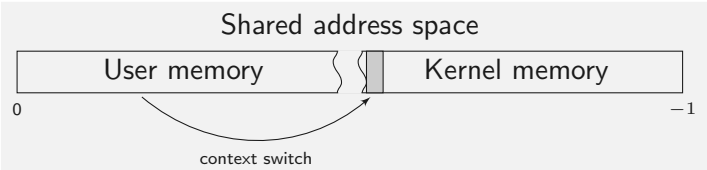
`prefetch` instructions are somewhat unusual

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

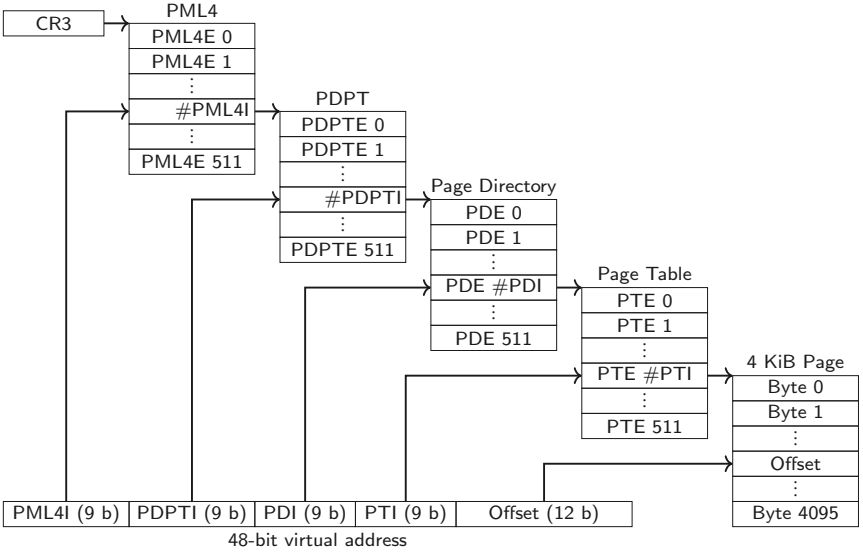
but they do need to translate virtual to physical

# Kernel must be mapped in every address space

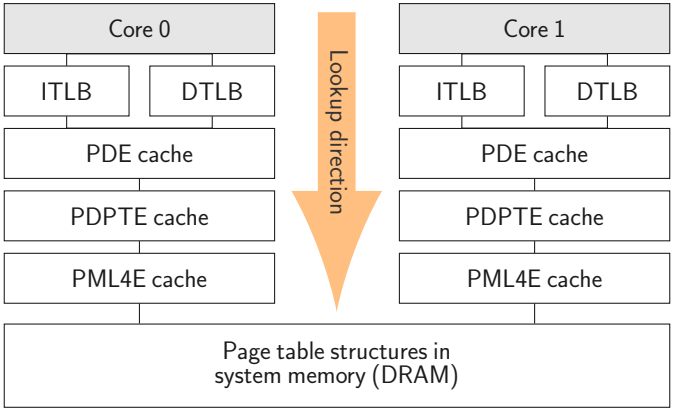
**Today's operating systems:**



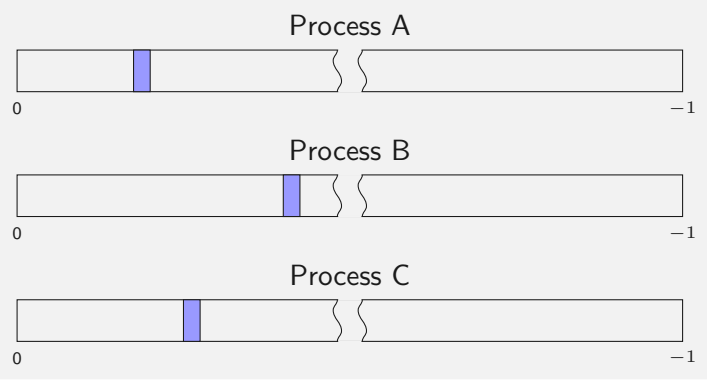
# Address translation on x86-64



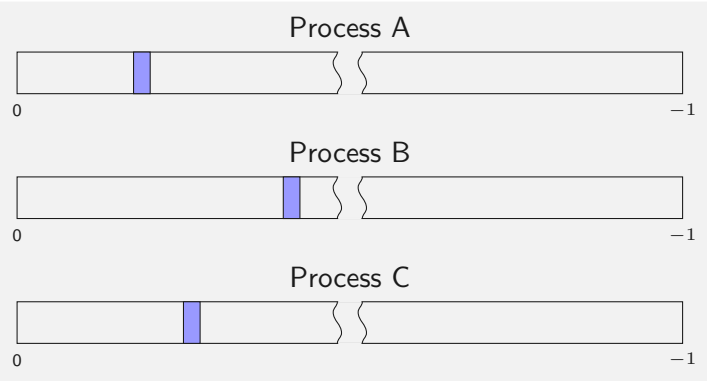
# Address Translation Caches



# Address Space Layout Randomization (ASLR)

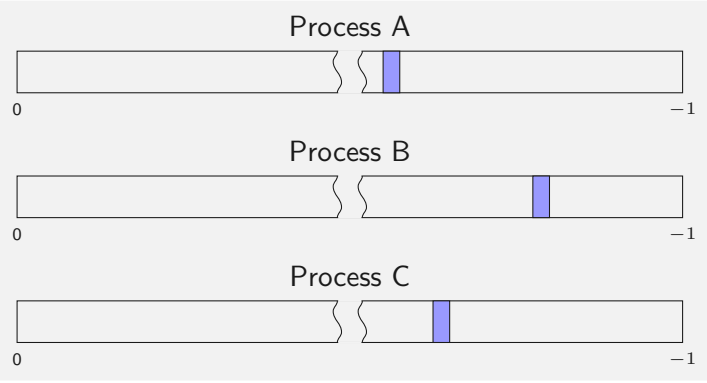


# Address Space Layout Randomization (ASLR)

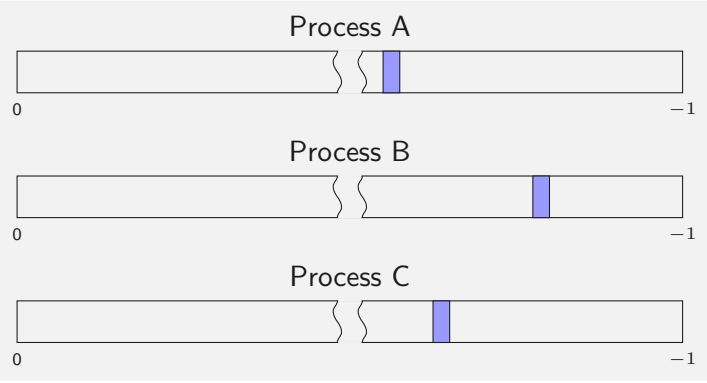



Same library – different offset!

# Kernel Address Space Layout Randomization (KASLR)

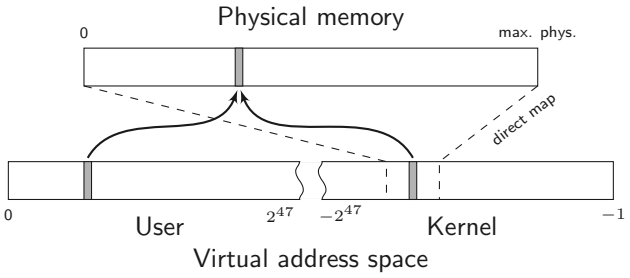


# Kernel Address Space Layout Randomization (KASLR)

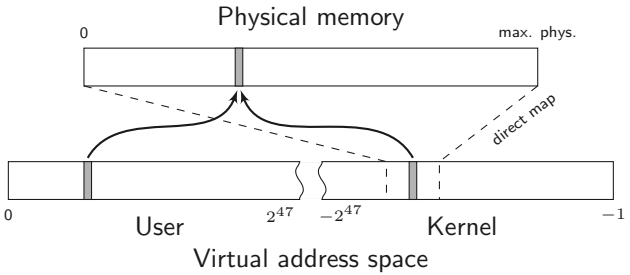



Same driver – different offset!

# Kernel direct-physical map

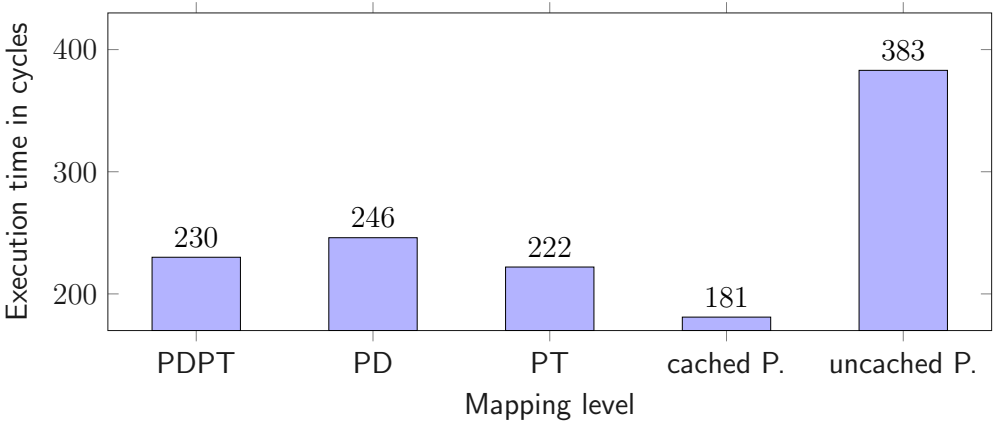


# Kernel direct-physical map

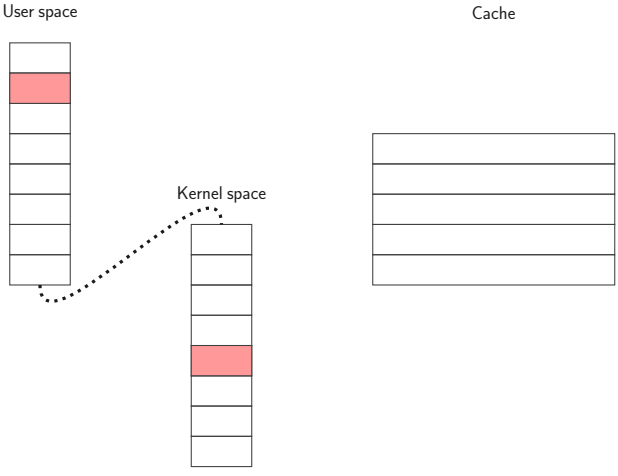



OS X, Linux, BSD, Xen PVM (Amazon EC2)

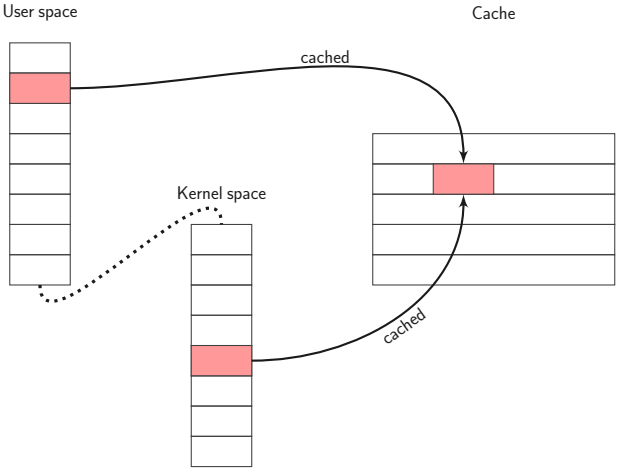
# Translation-Level Oracle



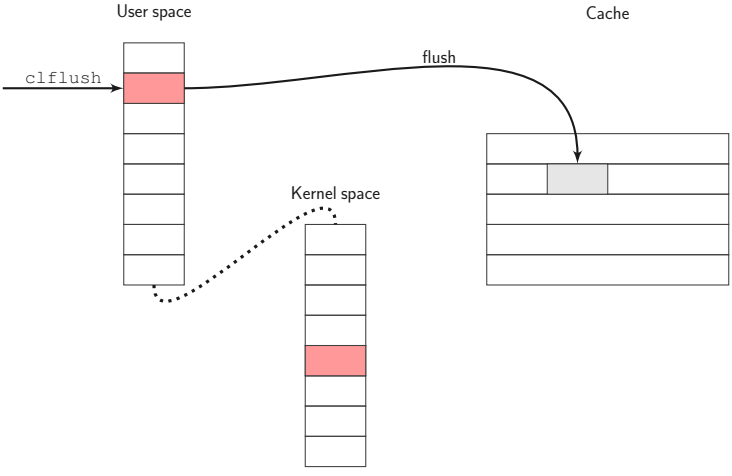
# Address-Translation Oracle



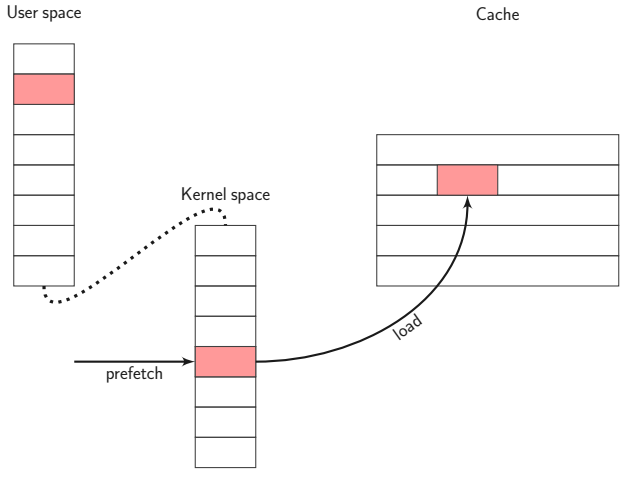
# Address-Translation Oracle



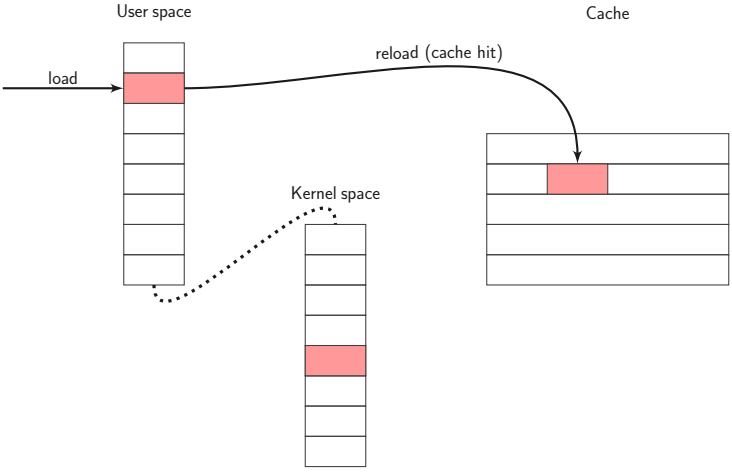
# Address-Translation Oracle



# Address-Translation Oracle

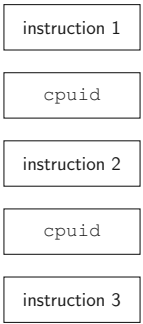


# Address-Translation Oracle



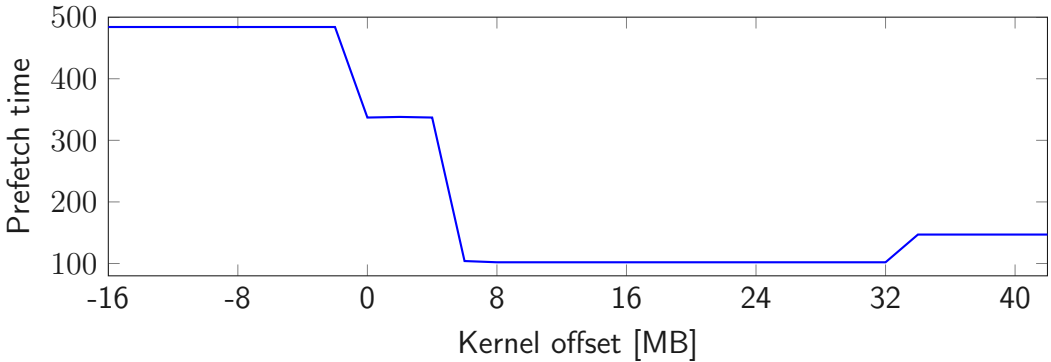
# Timing the prefetch instruction

The CPU may reorder instructions




but not over `cpuid`!

# Breaking KASLR with Prefetch



# Side-Channel Security

## Chapter 2: Cache Template Attacks

**Daniel Gruss**

March 9, 2021

Graz University of Technology

# References


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

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

[3] 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*.

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

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

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

[7] 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*.

[8] 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*.