What could possibly go wrong with <insert x86 instruction here>?

Clémentine Maurice, Moritz Lipp

December 2016—33rd Chaos Communication Congress

Who we are

- **Clémentine Maurice**
- PhD in computer science, Postdoc @ Graz University Of Technology
- @BloodyTangerine
- clementine.maurice@iaik.tugraz.at

Who we are

- Moritz Lipp
- PhD student @ Graz University Of Technology
- @mlqxyz
- moritz.lipp@iaik.tugraz.at

Introduction

- title says this is a talk about x86 instructions but...

Introduction

- title says this is a talk about x86 instructions but…
- … this is not a talk about software

Introduction

- title says this is a talk about x86 instructions but...
- ... this is not a talk about software
- assuming “safe” software

Introduction

- title says this is a talk about x86 instructions but...
- ... this is not a talk about software
- assuming “safe” software
- does not mean safe execution

Introduction

- title says this is a talk about x86 instructions but…
- … this is not a talk about software
- assuming “safe” software
- does not mean safe execution
- information leaks because of the underlying hardware

Introduction

- title says this is a talk about x86 instructions but...
- ... this is not a talk about software
- assuming “safe” software
- does not mean safe execution
- information leaks because of the underlying hardware

→ cache attacks without memory accesses and bypassing kernel ASLR

Introduction

- title says this is a talk about x86 instructions but…
- … this is not a talk about software
- assuming “safe” software
- does not mean safe execution
- information leaks because of the underlying hardware

→ cache attacks without memory accesses and bypassing kernel ASLR

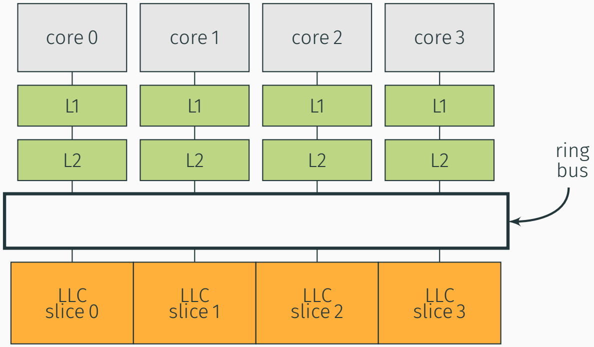
→ cache attacks can also be mounted on ARM, not solely on x86

Outline

- Background
- `mov` — The beginning of cache attacks
- `clflush` — Cache attacks without memory accesses
- `prefetch` — Lost in translation
- Bonus track — Even more instructions, even more attacks

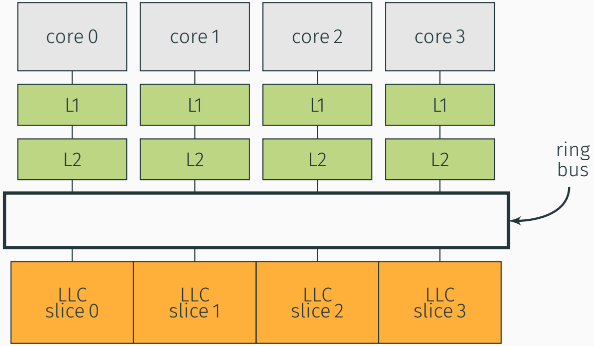
Introduction

Caches on Intel CPUs



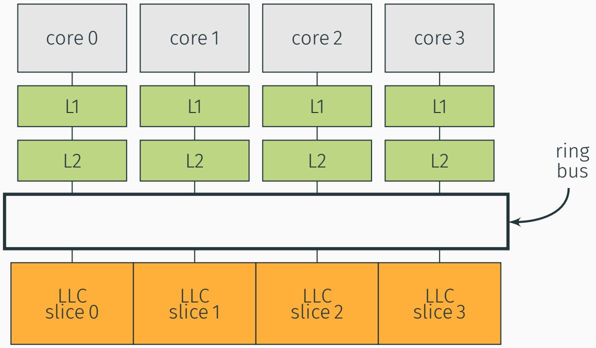
- L1 and L2 are private

Caches on Intel CPUs



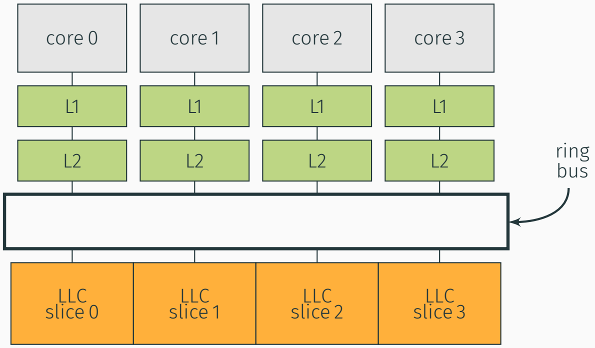
- L1 and L2 are private
- last-level cache

Caches on Intel CPUs



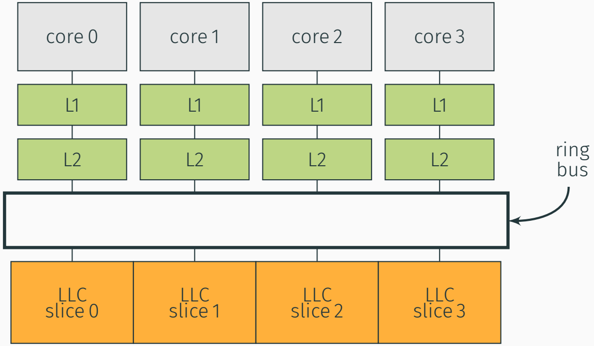
- L1 and L2 are private
- last-level cache
    - divided in slices

Caches on Intel CPUs



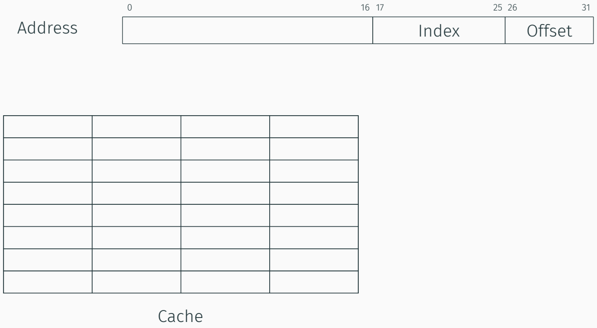
- L1 and L2 are private
- last-level cache
    - divided in slices
    - shared across cores

Caches on Intel CPUs

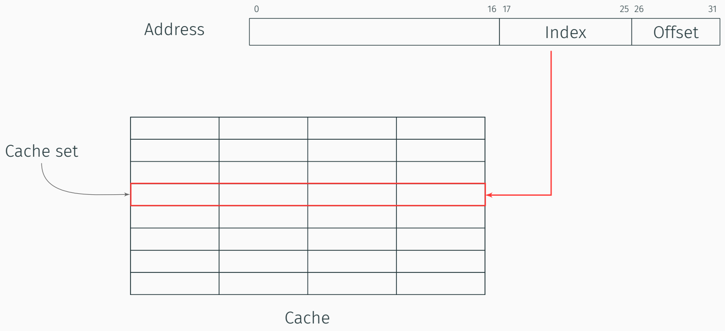


- L1 and L2 are private
- last-level cache
    - divided in slices
    - shared across cores
    - inclusive

Set-associative caches

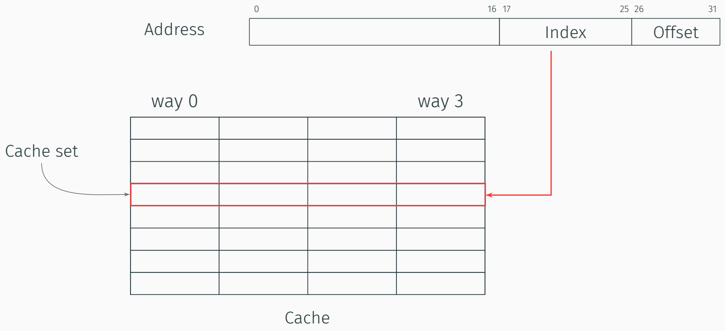


Set-associative caches



Data loaded in a specific set depending on its address

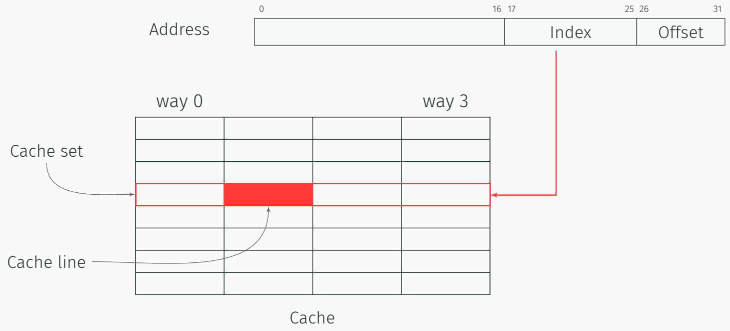
Set-associative caches



Data loaded in a specific set depending on its address

Several ways per set

Set-associative caches



Data loaded in a specific set depending on its address

Several ways per set

Cache line loaded in a specific way depending on the replacement policy

Today’s menu

Three instructions

Today’s menu

Three instructions

1. `mov`: accesses data in the main memory

Today’s menu

Three instructions

1. `mov`: accesses data in the main memory
2. `clflush`: removes cache line from the cache

Today’s menu

Three instructions

1. `mov`: accesses data in the main memory
2. `clflush`: removes cache line from the cache
3. `prefetch`: prefetches cache line for future use

Today's menu

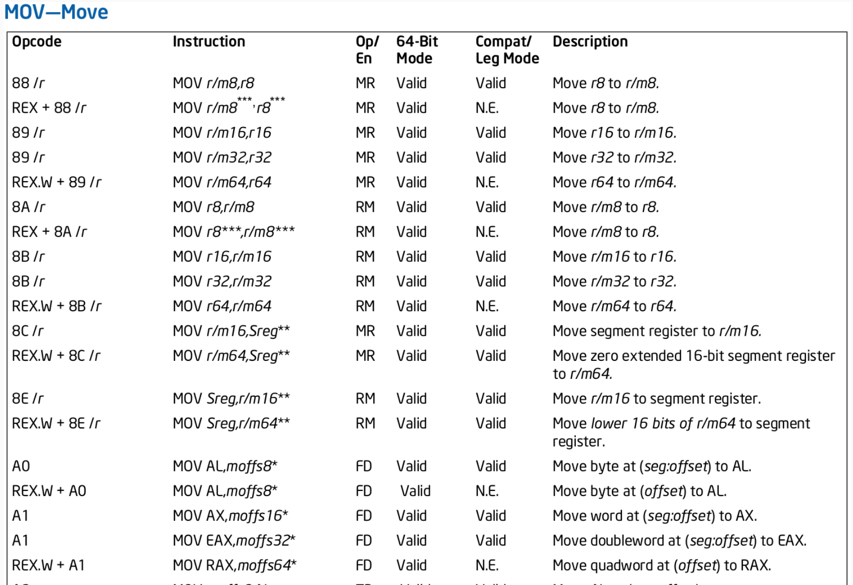
Three instructions

1. `mov`: accesses data in the main memory
2. `clflush`: removes cache line from the cache
3. `prefetch`: prefetches cache line for future use

That's all the assembly you need for today!

mov

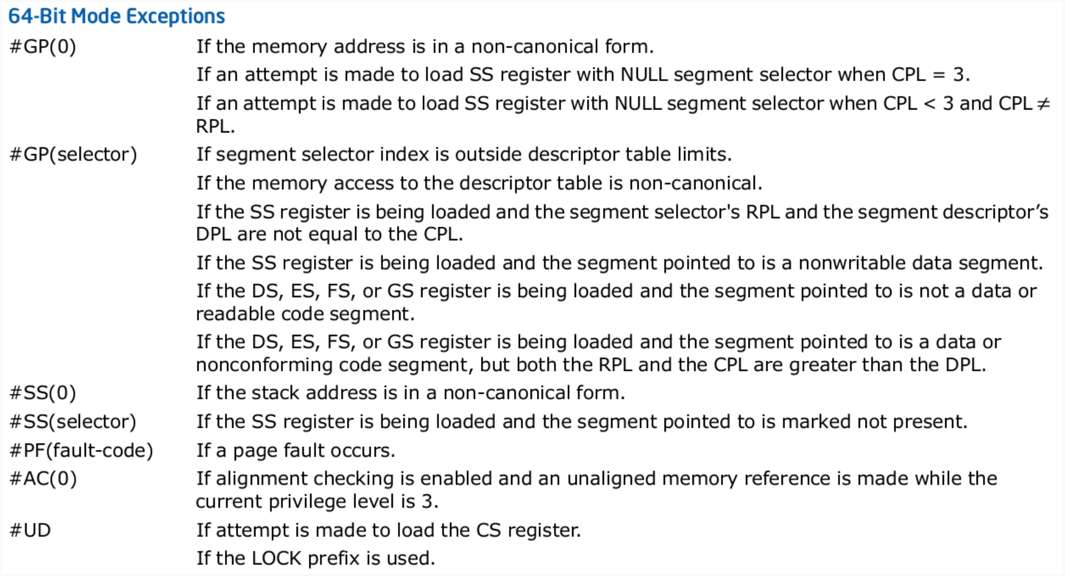
mov



MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 */r*	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8.*
REX + 88 */r*	MOV *r/m8***,r8****	MR	Valid	N.E.	Move *r8* to *r/m8.*
89 */r*	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16.*
89 */r*	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32.*
REX.W + 89 */r*	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64.*
8A */r*	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8.*
REX + 8A */r*	MOV *r8***,r/m8****	RM	Valid	N.E.	Move *r/m8* to *r8.*
8B */r*	MOV *r16,r/m16*	RM	Valid	Valid	Move *r/m16* to *r16.*
8B */r*	MOV *r32,r/m32*	RM	Valid	Valid	Move *r/m32* to *r32.*
REX.W + 8B */r*	MOV *r64,r/m64*	RM	Valid	N.E.	Move *r/m64* to *r64.*
8C */r*	MOV *r/m16,Sreg***	MR	Valid	Valid	Move segment register to *r/m16.*
REX.W + 8C */r*	MOV *r/m64,Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r/m64.*
8E */r*	MOV *Sreg,r/m16***	RM	Valid	Valid	Move *r/m16* to segment register.
REX.W + 8E */r*	MOV *Sreg,r/m64***	RM	Valid	Valid	Move *lower 16 bits of r/m64* to segment register.
A0	MOV AL,*moffs8**	FD	Valid	Valid	Move byte at (*seg:offset*) to AL.
REX.W + A0	MOV AL,*moffs8**	FD	Valid	N.E.	Move byte at (*offset*) to AL.
A1	MOV AX,*moffs16**	FD	Valid	Valid	Move word at (*seg:offset*) to AX.
A1	MOV EAX,*moffs32**	FD	Valid	Valid	Move doubleword at (*seg:offset*) to EAX.
REX.W + A1	MOV RAX,*moffs64**	FD	Valid	N.E.	Move quadword at (*offset*) to RAX.

mov



64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If an attempt is made to load SS register with NULL segment selector when CPL = 3.
	If an attempt is made to load SS register with NULL segment selector when CPL < 3 and CPL ≠ RPL.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the memory access to the descriptor table is non-canonical.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor’s DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a nonwritable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the stack address is in a non-canonical form.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

`mov`—What could go wrong?

- lots of exceptions for `mov`

`mov`—What could go wrong?

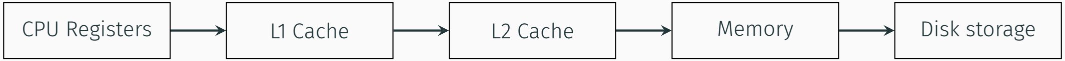
- lots of exceptions for `mov`
- but accessing data loads it to the cache

`mov`—What could go wrong?

- lots of exceptions for `mov`
- but accessing data loads it to the cache

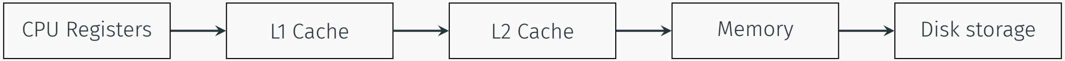
→ side effects on computations!

Memory Hierarchy



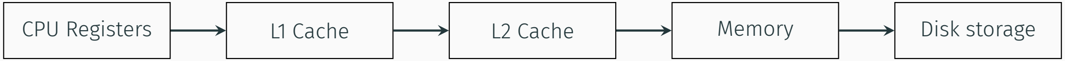
- Data can reside in

Memory Hierarchy



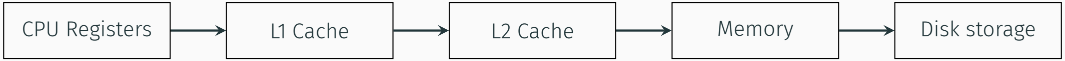
- Data can reside in
    - CPU registers

Memory Hierarchy



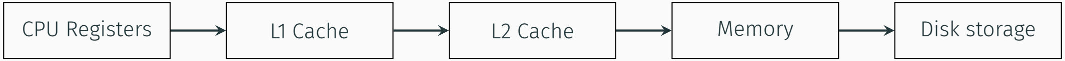
- Data can reside in
  - CPU registers
  - Different levels of the CPU cache

Memory Hierarchy



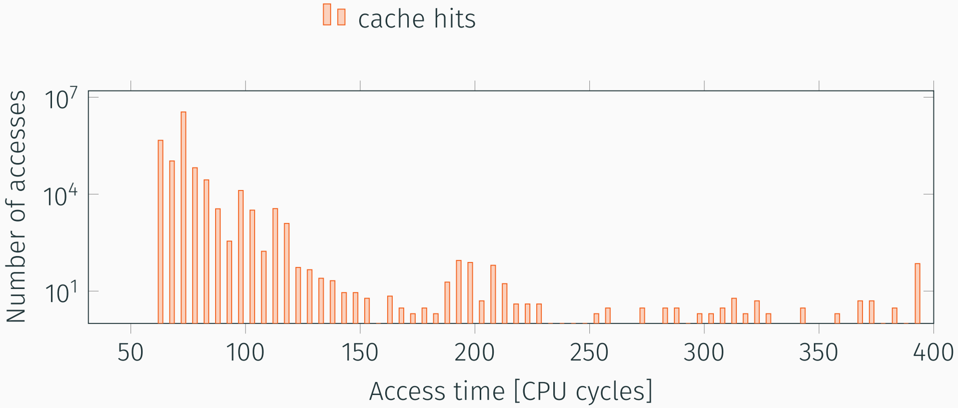
- Data can reside in
    - CPU registers
    - Different levels of the CPU cache
    - Main memory

Memory Hierarchy

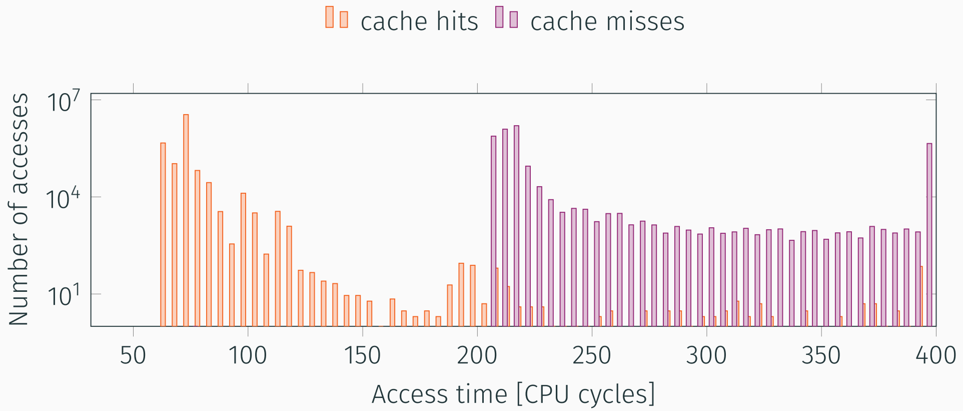


- Data can reside in
  - CPU registers
  - Different levels of the CPU cache
  - Main memory
  - Disk storage

Timing differences



Timing differences



Cache attacks

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

Cache attacks

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

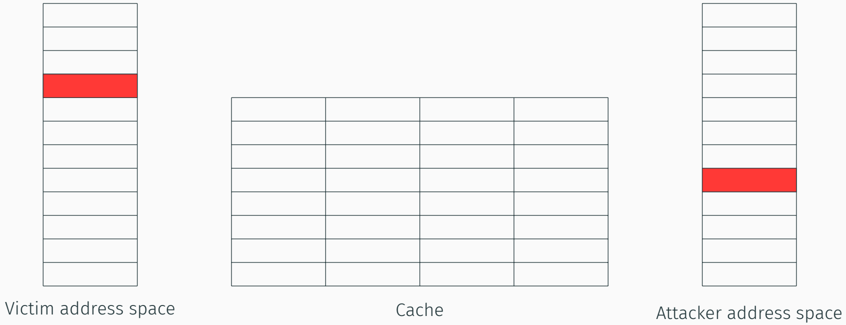
Cache attacks

- cache attacks $\rightarrow$ exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
- covert channel: two processes communicating with each other
    - not allowed to do so, e.g., across VMs

Cache attacks

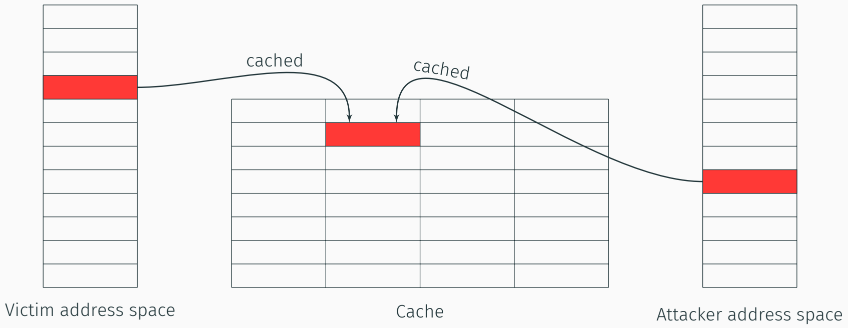
- cache attacks $\rightarrow$ exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
- covert channel: two processes communicating with each other
    - not allowed to do so, e.g., across VMs
- side-channel attack: one malicious process spies on benign processes
    - e.g., steals crypto keys, spies on keystrokes

Cache attacks: *Flush+Reload*



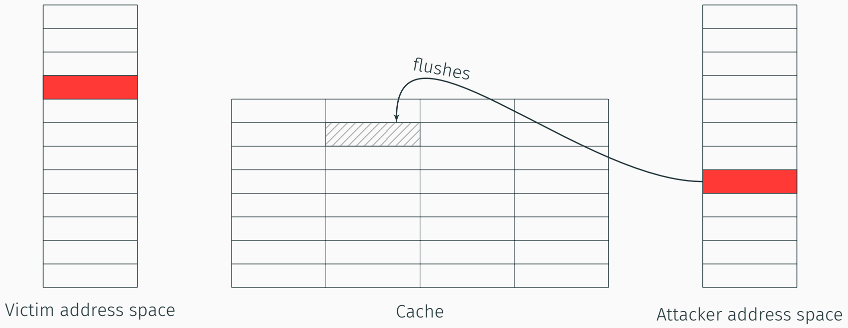
Step 1: Attacker maps shared library (shared memory, in cache)

Cache attacks: *Flush+Reload*



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

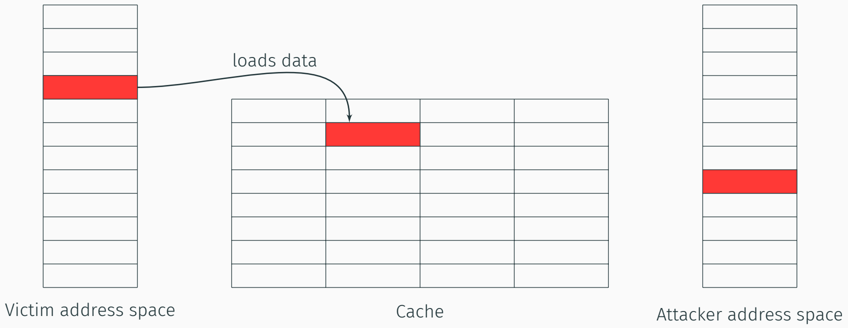
Cache attacks: *Flush+Reload*



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

Step 2: Attacker flushes the shared cache line

Cache attacks: *Flush+Reload*

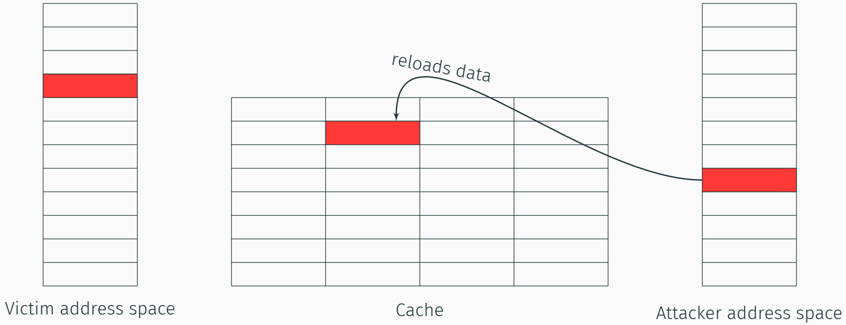


Step 1: Attacker maps shared library (shared memory, in cache)

Step 2: Attacker flushes the shared cache line

Step 3: Victim loads the data

Cache attacks: *Flush+Reload*



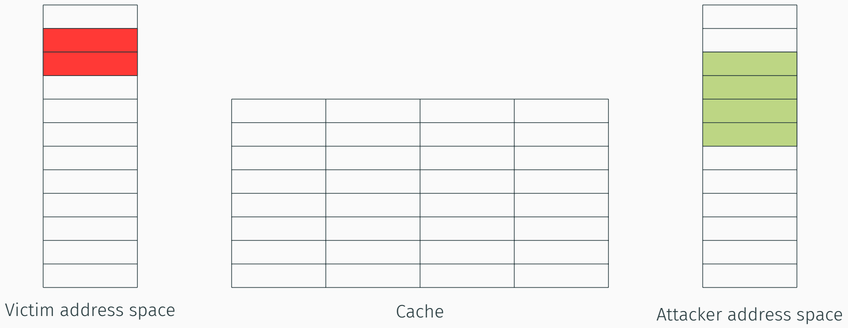
Step 1: Attacker maps shared library (shared memory, in cache)

Step 2: Attacker flushes the shared cache line

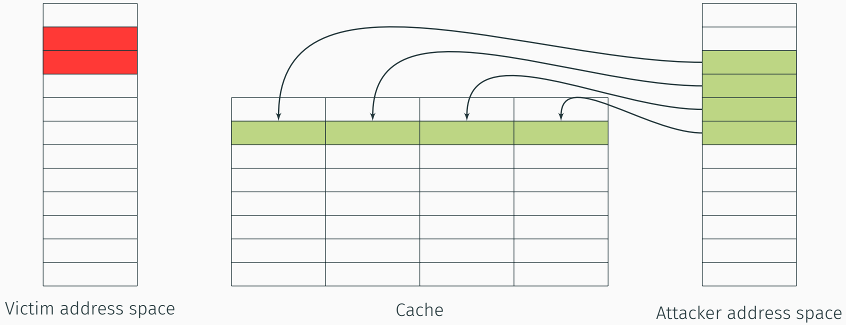
Step 3: Victim loads the data

Step 4: Attacker reloads the data

Cache attacks: *Prime+Probe*

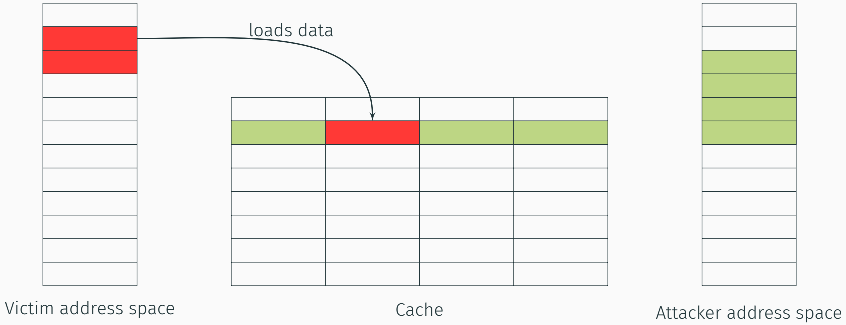


Cache attacks: *Prime+Probe*



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

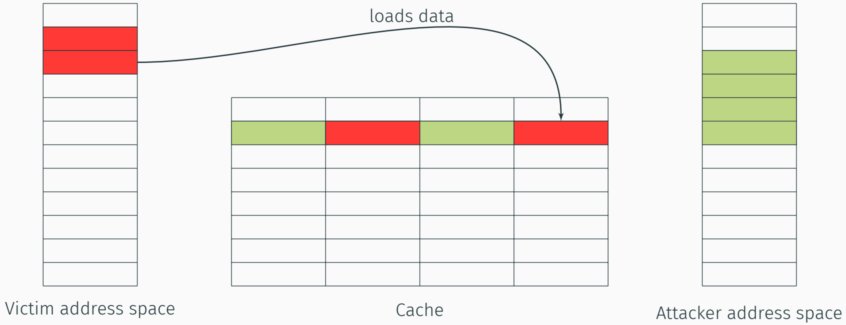
Cache attacks: *Prime+Probe*



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

Step 2: Victim evicts cache lines while running

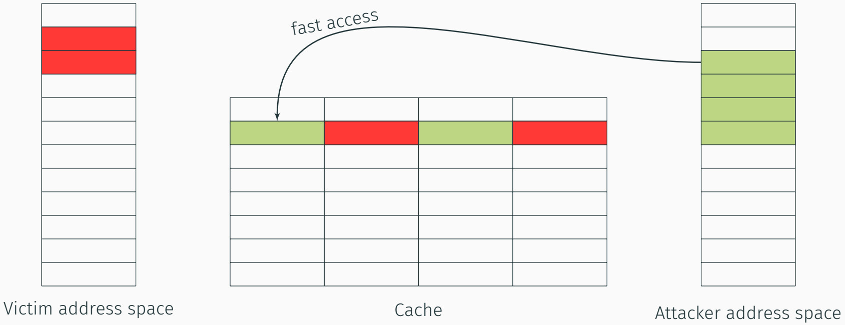
Cache attacks: *Prime+Probe*



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

Step 2: Victim evicts cache lines while running

Cache attacks: *Prime+Probe*

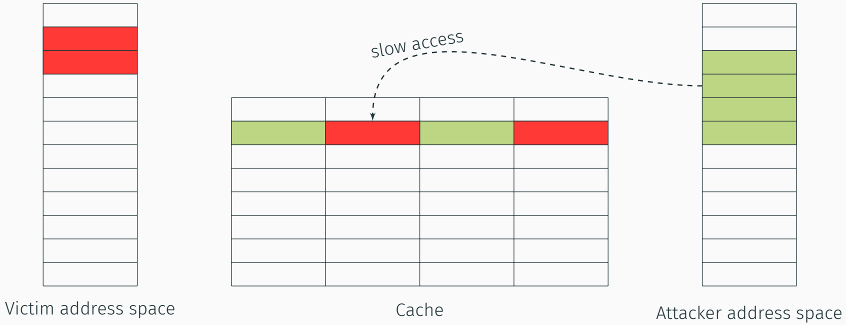


Step 1: Attacker primes, *i.e.*, fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running

Step 3: Attacker probes data to determine if set has been accessed

Cache attacks: *Prime+Probe*



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

Step 2: Victim evicts cache lines while running

Step 3: Attacker probes data to determine if set has been accessed

Application #1

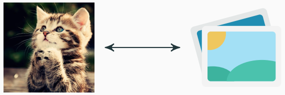
Covert Channel

Application #1: Covert channel

- Malicious privacy gallery app

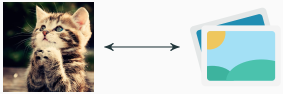
Application #1: Covert channel

- Malicious privacy gallery app
    - No permissions except accessing your images



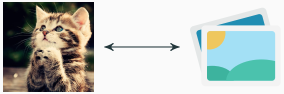
Application #1: Covert channel

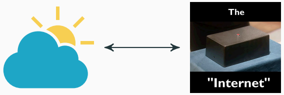
- Malicious privacy gallery app
    - No permissions except accessing your images
- Malicious weather widget



Application #1: Covert channel

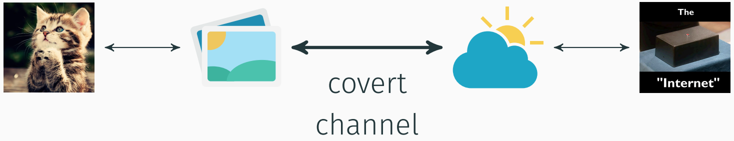
- Malicious privacy gallery app
    - No permissions except accessing your images
- Malicious weather widget
    - No permissions except accessing the Internet





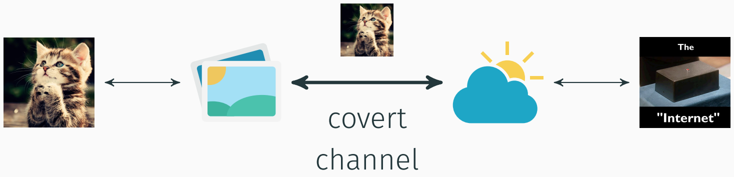
Application #1: Covert channel

- Malicious privacy gallery app
  - No permissions except accessing your images
- Malicious weather widget
  - No permissions except accessing the Internet



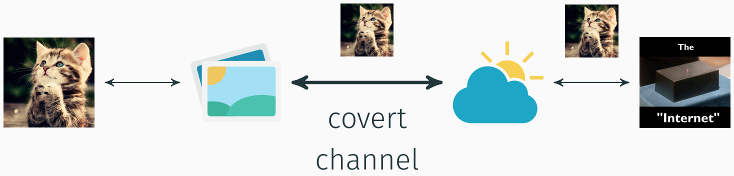
Application #1: Covert channel

- Malicious privacy gallery app
    - No permissions except accessing your images
- Malicious weather widget
    - No permissions except accessing the Internet



Application #1: Covert channel

- Malicious privacy gallery app
    - No permissions except accessing your images
- Malicious weather widget
    - No permissions except accessing the Internet



Application #1: Covert channel

- sender and receiver agree on one set

Application #1: Covert channel

- sender and receiver agree on one set
- receiver probes the set continuously

Application #1: Covert channel

- sender and receiver agree on one set
- receiver probes the set continuously
- sender transmits ’0’ doing nothing

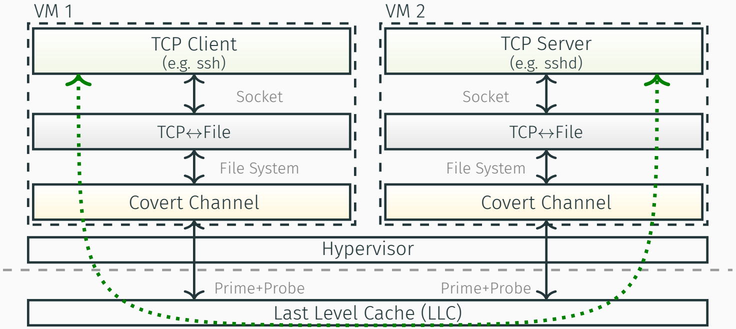
  $\rightarrow$ lines of the receiver still in cache $\rightarrow$ fast access

Application #1: Covert channel

- sender and receiver agree on one set
- receiver probes the set continuously
- sender transmits ’0’ doing nothing
  - $\rightarrow$ lines of the receiver still in cache $\rightarrow$ fast access
- sender transmits ’1’ accessing addresses in the set
  - $\rightarrow$ evicts lines of the receiver $\rightarrow$ slow access

Application #1: Covert channel

- Prime+Probe: low requirements, works e.g., between VMs in Amazon EC2
  - error-free covert channel (40–75KBps) $\rightarrow$ SSH connection over the cache



C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud". In: *NDSS'17*. to appear. 2017.

Application #2

Crypto side-channel attack

Application #2: Crypto side-channel attack

- AES T-Tables: fast software implementation

Application #2: Crypto side-channel attack

- AES T-Tables: fast software implementation
- uses precomputed look-up tables

Application #2: Crypto side-channel attack

- AES T-Tables: fast software implementation
- uses precomputed look-up tables
- one-round known-plaintext attack by Osvik et al. (2006)
    - p plaintext and k secret key
    - intermediate state $x^{(r)} = (x_0^{(r)}, \dots, x_{15}^{(r)})$ at each round r
    - first round, accessed table indices are

$$x_i^{(0)} = p_i \oplus k_i \qquad \text{for all } i = 0, \dots, 15$$

Application #2: Crypto side-channel attack

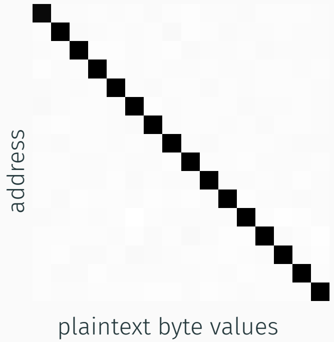
- AES T-Tables: fast software implementation
- uses precomputed look-up tables
- one-round known-plaintext attack by Osvik et al. (2006)
    - p plaintext and k secret key
    - intermediate state $x^{(r)} = (x_0^{(r)}, \dots, x_{15}^{(r)})$ at each round r
    - first round, accessed table indices are

$$x_i^{(0)} = p_i \oplus k_i \qquad \text{for all } i = 0, \dots, 15$$

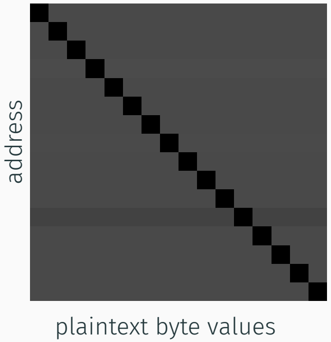
$\rightarrow$ recovering accessed table indices $\Rightarrow$ recovering the key

Application #2: Crypto side-channel attack

- monitoring which T-Table entry is accessed ($k_0 = \texttt{0x00}$)



Flush+Reload



Prime+Probe

Application #2: Crypto side-channel attack

- it’s an old attack...

Application #2: Crypto side-channel attack

- it’s an old attack…
- everything should be fixed by now…

Application #2: Crypto side-channel attack

- it's an old attack...
- everything should be fixed by now...
- Bouncy Castle on Android $\rightarrow$ default implementation uses T-Tables
- many implementations you find online use pre-computed values

Application #3

Spying on keystrokes

Application #3: Spying on keystrokes

- Flush+Reload: fine-grained attack → spy on keystrokes

M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. “ARMageddon: Cache Attacks on Mobile Devices”. In: *USENIX Security Symposium*. 2016

Application #3: Spying on keystrokes

- Flush+Reload: fine-grained attack → spy on keystrokes

Demo time!

M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. “ARMageddon: Cache Attacks on Mobile Devices”. In: *USENIX Security Symposium*. 2016

clflush

`clflush`—What could go wrong?

- `clflush`: invalidates from every level the cache line containing the address

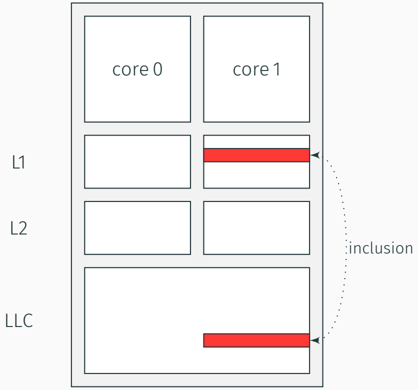
`clflush`—What could go wrong?

- `clflush`: invalidates from every level the cache line containing the address
- in itself enables Flush+Reload attacks

`clflush`—What could go wrong?

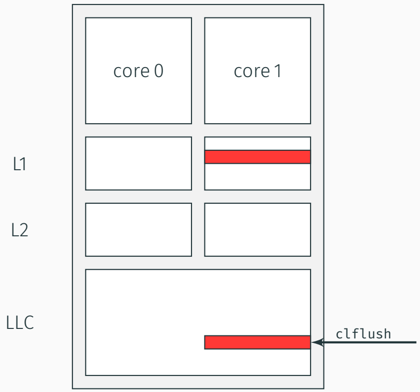
- `clflush`: invalidates from every level the cache line containing the address
- in itself enables Flush+Reload attacks
- but there's more!

`clflush` timing leakage: Part #1



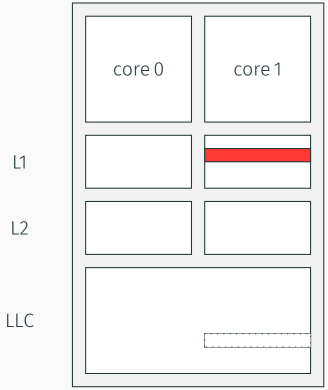
- `clflush` on cached data

`clflush` timing leakage: Part #1



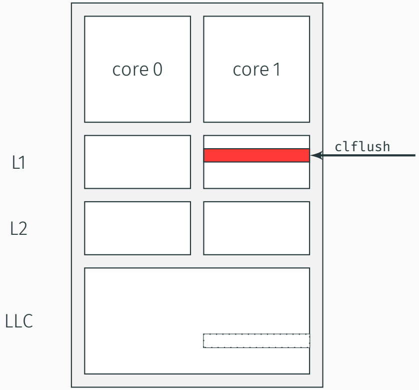
- `clflush` on cached data
    - goes to LLC, flushes line

`clflush` timing leakage: Part #1



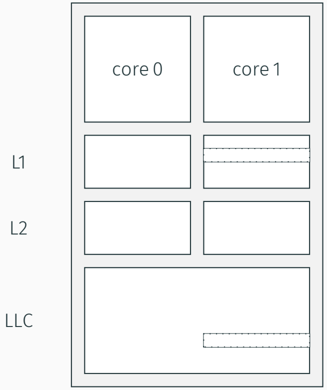
- `clflush` on cached data
  - goes to LLC, flushes line

`clflush` timing leakage: Part #1



- `clflush` on cached data
  - goes to LLC, flushes line
  - flushes line in L1-L2

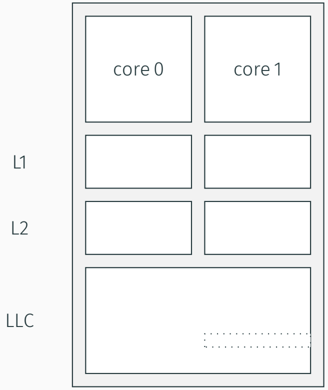
`clflush` timing leakage: Part #1



- `clflush` on cached data
  - goes to LLC, flushes line
  - flushes line in L1-L2

  → slow

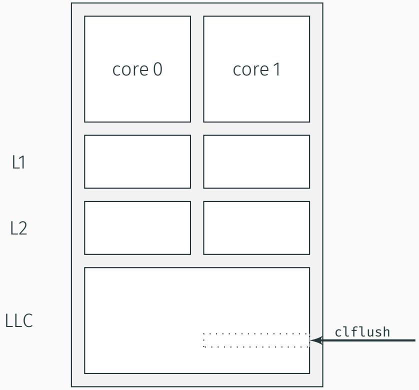
`clflush` timing leakage: Part #1



- `clflush` on cached data
  - goes to LLC, flushes line
  - flushes line in L1-L2

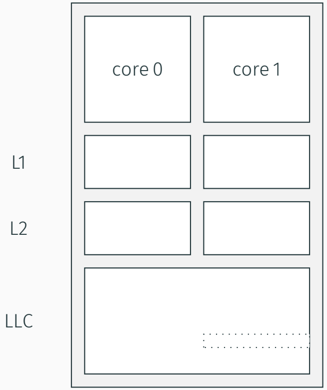
  → slow
- `clflush` on non-cached data

`clflush` timing leakage: Part #1



- `clflush` on cached data
  - goes to LLC, flushes line
  - flushes line in L1-L2
  - → slow
- `clflush` on non-cached data
  - goes to LLC, does nothing

`clflush` timing leakage: Part #1

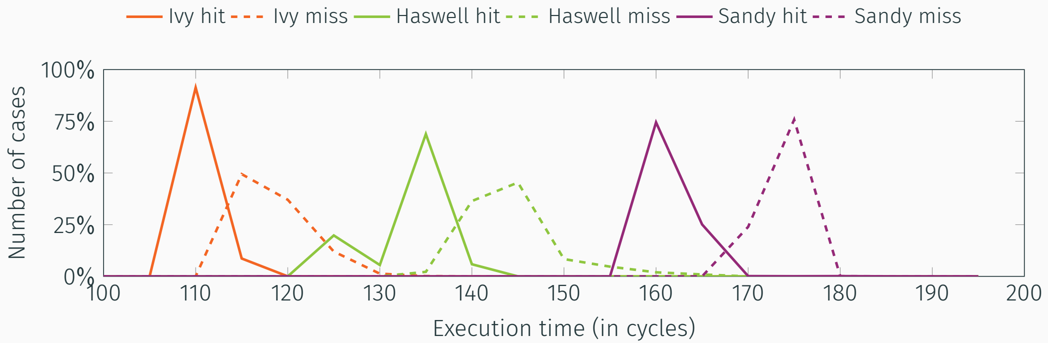


- `clflush` on cached data
  - goes to LLC, flushes line
  - flushes line in L1-L2

  → slow
- `clflush` on non-cached data
  - goes to LLC, does nothing

  → fast

`clflush` timing leakage: Part #1



It’s only a few cycles, what could go wrong?!

- new cache attack: Flush+Flush

D. Gruss, C. Maurice, K. Wagner, and S. Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA’16*. 2016.

It’s only a few cycles, what could go wrong?!

- new cache attack: Flush+Flush
- covert channels and side-channel attacks

D. Gruss, C. Maurice, K. Wagner, and S. Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA’16*. 2016.

It’s only a few cycles, what could go wrong?!

- new cache attack: Flush+Flush
- covert channels and side-channel attacks
- stealthier than previous cache attacks

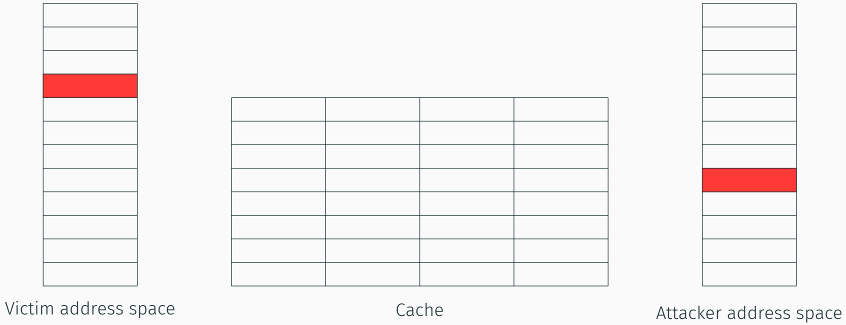
D. Gruss, C. Maurice, K. Wagner, and S. Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA’16*. 2016.

It's only a few cycles, what could go wrong?!

- new cache attack: Flush+Flush
- covert channels and side-channel attacks
- stealthier than previous cache attacks
- faster than previous cache attacks

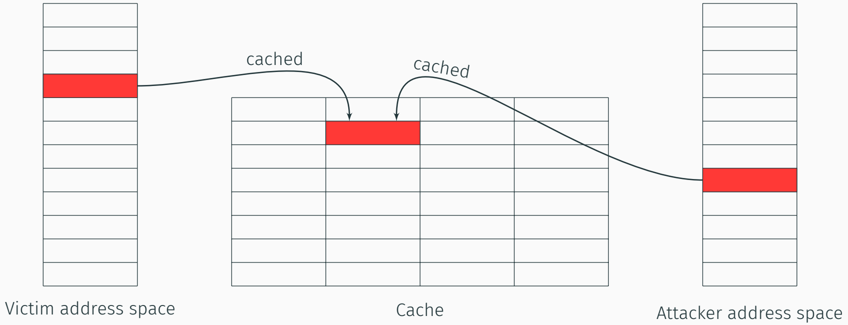
D. Gruss, C. Maurice, K. Wagner, and S. Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *DIMVA'16*. 2016.

New cache attack: Flush+Flush



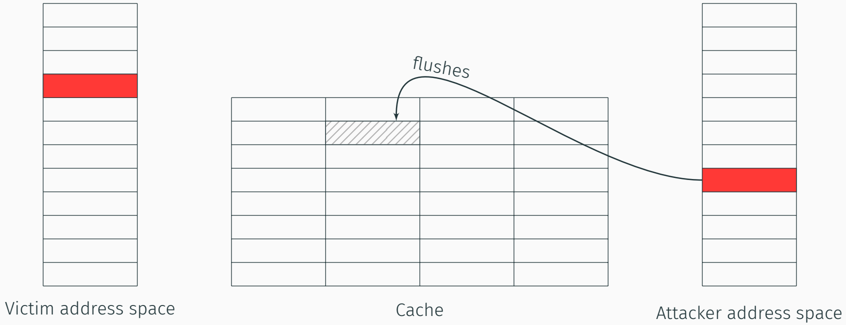
Step 1: Attacker maps shared library (shared memory, in cache)

New cache attack: Flush+Flush



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

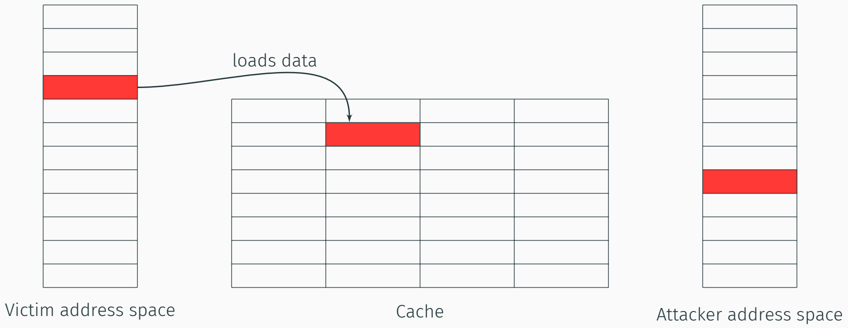
New cache attack: Flush+Flush



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

Step 2: Attacker flushes the shared cache line

New cache attack: Flush+Flush

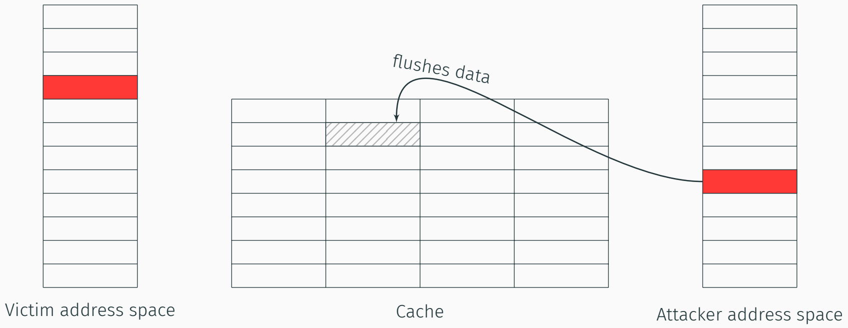


Step 1: Attacker maps shared library (shared memory, in cache)

Step 2: Attacker flushes the shared cache line

Step 3: Victim loads the data

New cache attack: Flush+Flush



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

Step 2: Attacker flushes the shared cache line

Step 3: Victim loads the data

Step 4: Attacker flushes the data

Evaluating stealthiness

Detecting cache attacks and Rowhammer with performance counters

1. `CACHE_MISSES` $\rightarrow$ occur after data is flushed
2. `CACHE_REFERENCES` $\rightarrow$ occur when reaccessing memory

N. Herath and A. Fogh. “These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security”. In: *Black Hat 2015 Briefings*. 2015

Evaluating stealthiness

Detecting cache attacks and Rowhammer with performance counters

1. `CACHE_MISSES` → occur after data is flushed
2. `CACHE_REFERENCES` → occur when reaccessing memory

… without false positives

N. Herath and A. Fogh. "These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security". In: *Black Hat 2015 Briefings*. 2015

Evaluating stealthiness

Detecting cache attacks and Rowhammer with performance counters

1. `CACHE_MISSES` $\rightarrow$ occur after data is flushed
2. `CACHE_REFERENCES` $\rightarrow$ occur when reaccessing memory

... without false positives

- heavy activity on the cache

N. Herath and A. Fogh. “These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security”. In: *Black Hat 2015 Briefings*. 2015

Evaluating stealthiness

Detecting cache attacks and Rowhammer with performance counters

1. `CACHE_MISSES` $\rightarrow$ occur after data is flushed
2. `CACHE_REFERENCES` $\rightarrow$ occur when reaccessing memory

… without false positives

- heavy activity on the cache
- also, very short loops of code $\rightarrow$ low pressure on the iTLB

N. Herath and A. Fogh. “These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security”. In: *Black Hat 2015 Briefings*. 2015

Evaluating stealthiness

Detecting cache attacks and Rowhammer with performance counters

1. `CACHE_MISSES` $\rightarrow$ occur after data is flushed
2. `CACHE_REFERENCES` $\rightarrow$ occur when reaccessing memory

... without false positives

- heavy activity on the cache
- also, very short loops of code $\rightarrow$ low pressure on the iTLB

$\rightarrow$ normalize the events by `ITLB_RA+ITLB_RM`

N. Herath and A. Fogh. “These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security”. In: *Black Hat 2015 Briefings*. 2015

Flush+Flush: Covert channel

technique	packet size	capacity (KBps)	receiver stealth	sender stealth
Flush+Flush	28			
Flush+Reload	28			

Flush+Flush: Covert channel

technique	packet size	capacity (KBps)	receiver stealth	sender stealth
Flush+Flush	28	496		
Flush+Reload	28	298		

Flush+Flush: Covert channel

technique	packet size	capacity (KBps)	receiver stealth	sender stealth
Flush+Flush	28	496	✓	
Flush+Reload	28	298	✗	

Flush+Flush: Covert channel

technique	packet size	capacity (KBps)	receiver stealth	sender stealth
Flush+Flush	28	496	✓	✗
Flush+Reload	28	298	✗	✗

Flush+Flush: Covert channel

technique	packet size	capacity (KBps)	receiver stealth	sender stealth
Flush+Flush	28	496	✓	✗
Flush+Reload	28	298	✗	✗

Flush+Flush: Covert channel

technique	packet size	capacity (KBps)	receiver stealth	sender stealth
Flush+Flush	28	496	✓	✗
Flush+Reload	28	298	✗	✗
Flush+Reload	4			
Flush+Flush	4			
Prime+Probe	4			

Flush+Flush: Covert channel

technique	packet size	capacity (KBps)	receiver stealth	sender stealth
Flush+Flush	28	496	✓	✗
Flush+Reload	28	298	✗	✗
Flush+Reload	4	54		
Flush+Flush	4	52		
Prime+Probe	4	34		

Flush+Flush: Covert channel

technique	packet size	capacity (KBps)	receiver stealth	sender stealth
Flush+Flush	28	496	✓	✗
Flush+Reload	28	298	✗	✗
Flush+Reload	4	54	✗	
Flush+Flush	4	52	✓	
Prime+Probe	4	34	✗	

Flush+Flush: Covert channel

technique	packet size	capacity (KBps)	receiver stealth	sender stealth
Flush+Flush	28	496	✓	✗
Flush+Reload	28	298	✗	✗
Flush+Reload	4	54	✗	✓
Flush+Flush	4	52	✓	✓
Prime+Probe	4	34	✗	✗

Flush+Flush: Covert channel

technique	packet size	capacity (KBps)	receiver stealth	sender stealth
Flush+Flush	28	496	✓	✗
Flush+Reload	28	298	✗	✗
Flush+Reload	4	54	✗	✓
Flush+Flush	4	52	✓	✓
Prime+Probe	4	34	✗	✗

Flush+Flush: Side channel on AES T-tables (1)

Number of encryptions to determine the upper 4 bits of a key byte

technique	number of encryptions
Flush+Reload	250
Flush+Flush	350
Prime+Probe	4 800

$\rightarrow$ same performance for Flush+Flush and Flush+Reload

Flush+Flush: Side channel on AES T-tables (2)

Stealthiness comparison on 256 million encryptions (synchronous attack)

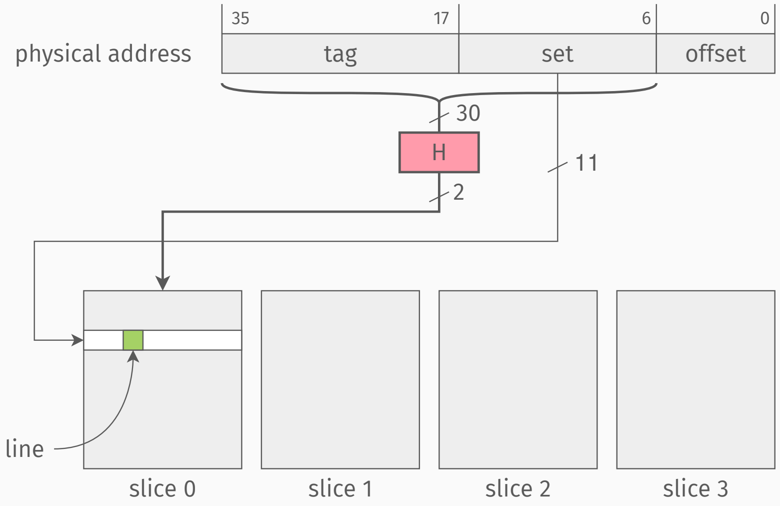
technique	time (s)	stealth
Flush+Reload	215	✗
Prime+Probe	234	✗
Flush+Flush	163	✓

→ Flush+Flush is the only stealth spy process

→ others need to be slowed down too much to be practical

A little bit more background before continuing…

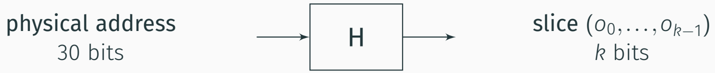
Last-level cache (1)



Last-level cache (2)

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

For 2^k slices:



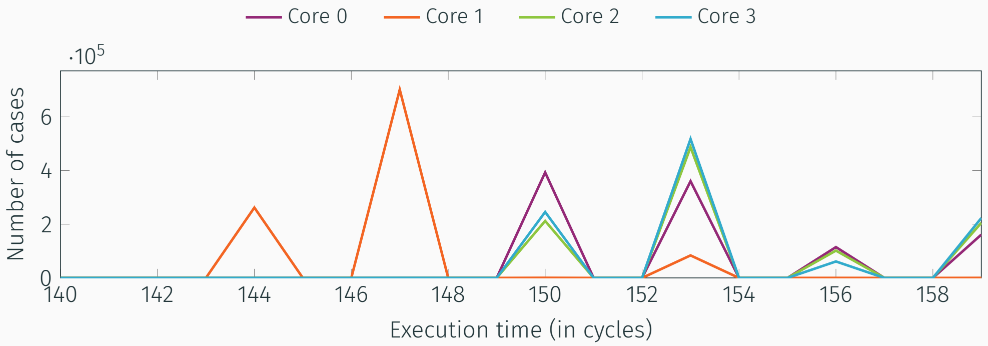
Let’s go back to `clflush`!

`clflush` timing leakage: Part #2

- `clflush` faster to reach a line on the local slice

clflush timing leakage: Part #2

- `clflush` faster to reach a line on the local slice



`clflush` timing leakage: Part #2

- map physical addresses to slices

`clflush` timing leakage: Part #2

- map physical addresses to slices
- one way to reverse-engineer the addressing function

`clflush` timing leakage: Part #2

- map physical addresses to slices
- one way to reverse-engineer the addressing function
- other way: using performance counters[1]

[1] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. “Reverse Engineering Intel Complex Addressing Using Performance Counters”. In: *RAID'15*. 2015

prefetch

`prefetch` instructions

`prefetch` fetches the line of data from memory containing the specified byte

6 `prefetch` instructions:

- `prefetcht0`: suggests CPU to load data into L1
- `prefetcht1`: suggests CPU to load data into L2
- `prefetcht2`: suggests CPU to load data into L3
- `prefetchnta`: suggests CPU to load data for non-temporal access
- `prefetchw`: suggests CPU to load data with intention to write
- `prefetchwt1`: suggests CPU to load vector data with intention to write

`prefetch` according to Intel

NOTE

Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* 2014

`prefetch` according to Intel

NOTE

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

Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* 2014

`prefetch` according to Intel

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 Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* 2014

prefetch according to Intel

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 Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* 2014

prefetch according to Intel

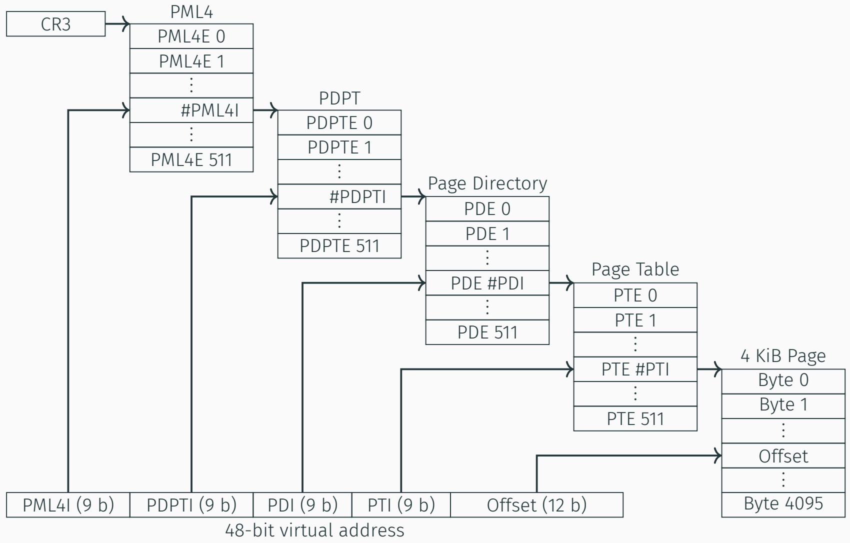
NOTE

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.

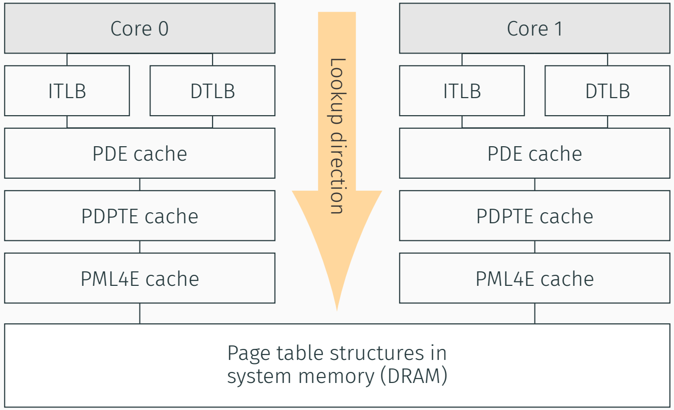
Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* 2014

A little bit more background before continuing...

Address translation

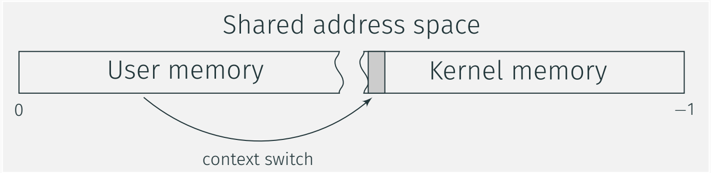


Address translation caches

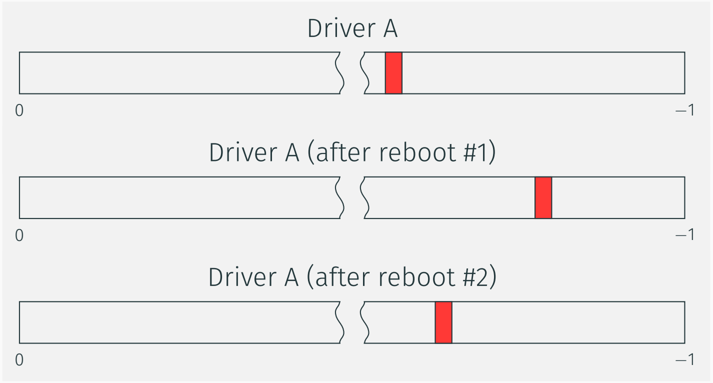


Kernel is mapped in every process

Today’s operating systems:

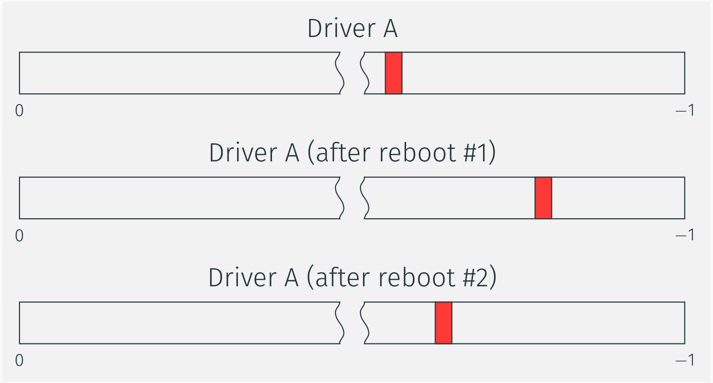


Kernel Address Space Layout Randomization (KASLR)



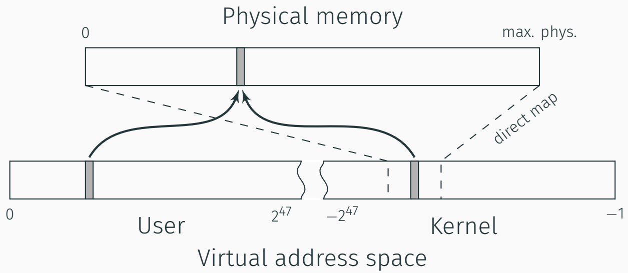
- same driver, different offset at each boot

Kernel Address Space Layout Randomization (KASLR)



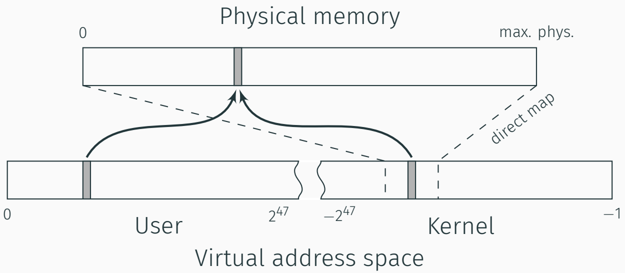
- same driver, different offset at each boot
- leaking kernel/driver addresses defeats KASLR

Kernel direct-physical map



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

Kernel direct-physical map



- OS X, Linux, BSD, Xen PVM (Amazon EC2)
- not Windows

Let’s go back to `prefetch`!

`prefetch`: Unusual instructions (1)

- tells the CPU “I might need that later”

D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. . In: *CCS’16*. 2016

`prefetch`: Unusual instructions (1)

- tells the CPU "I might need that later"
- hint—may be ignored by the CPU

D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR". . In: *CCS'16*. 2016

`prefetch`: Unusual instructions (1)

- tells the CPU "I might need that later"
- hint—may be ignored by the CPU
- generates no faults

D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR". . In: *CCS'16*. 2016

`prefetch`: Unusual instructions (1)

- tells the CPU "I might need that later"
- hint—may be ignored by the CPU
- generates no faults

Property #1: do not check privileges

D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR". . In: *CCS'16*. 2016

`prefetch`: Unusual instructions (2)

- operand is a virtual address

D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. . In: *CCS’16*. 2016

`prefetch`: Unusual instructions (2)

- operand is a virtual address
- but it needs to translate the virtual address to a physical address

D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. . In: *CCS’16*. 2016

`prefetch`: Unusual instructions (2)

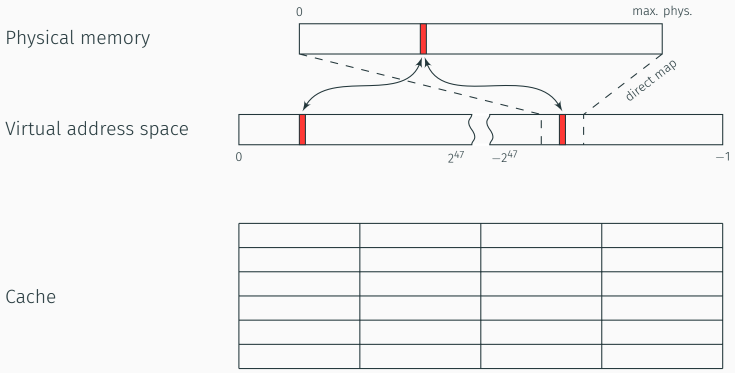
- operand is a virtual address
- but it needs to translate the virtual address to a physical address

Property #2: execution time varies

D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. . In: *CCS'16*. 2016

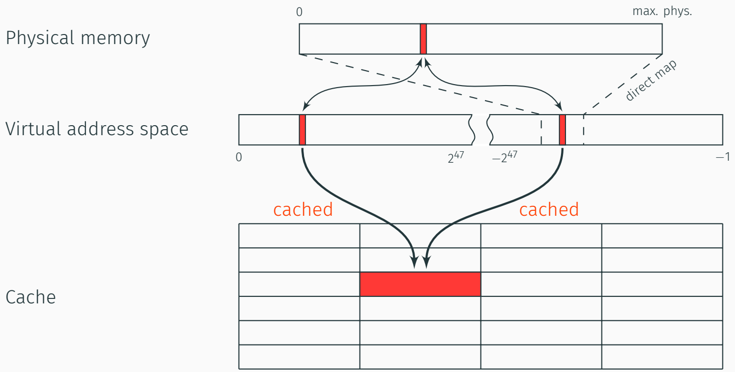
Address-translation oracle

Exploiting property #1 + kernel direct-physical map



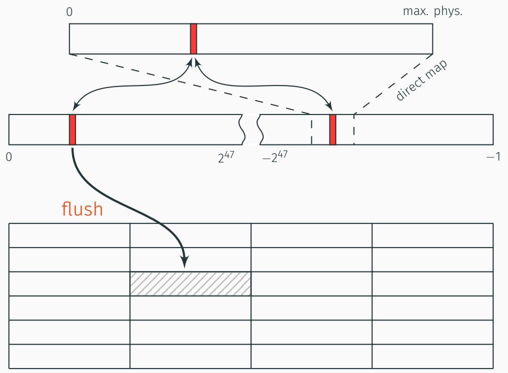
Address-translation oracle

Exploiting property #1 + kernel direct-physical map



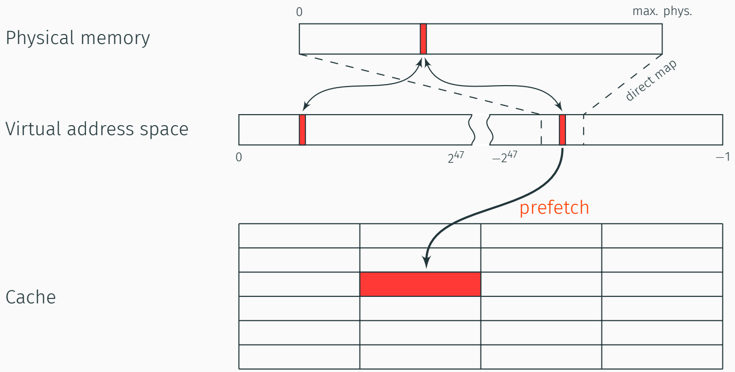
Address-translation oracle

Exploiting property #1 + kernel direct-physical map



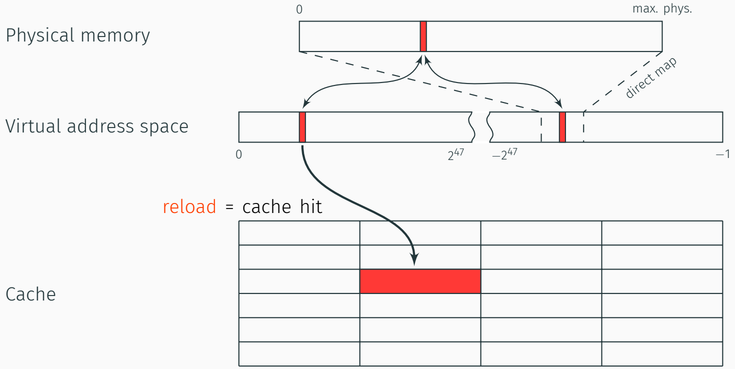
Address-translation oracle

Exploiting property #1 + kernel direct-physical map



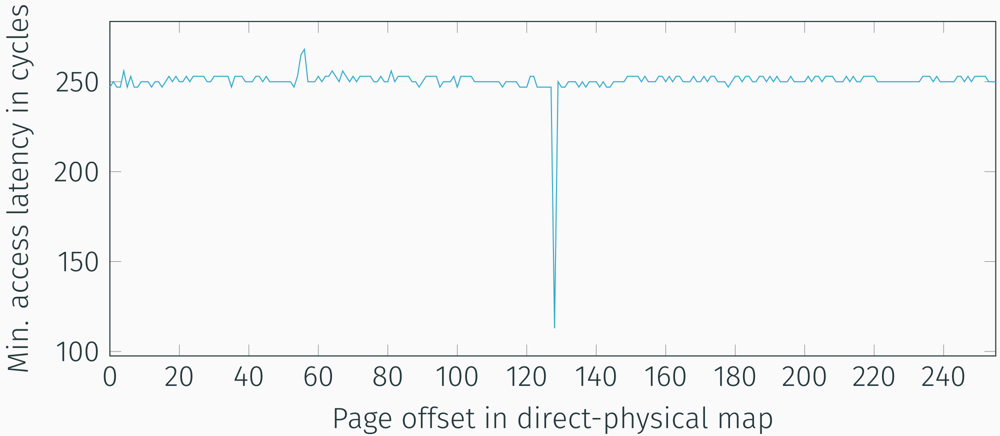
Address-translation oracle

Exploiting property #1 + kernel direct-physical map



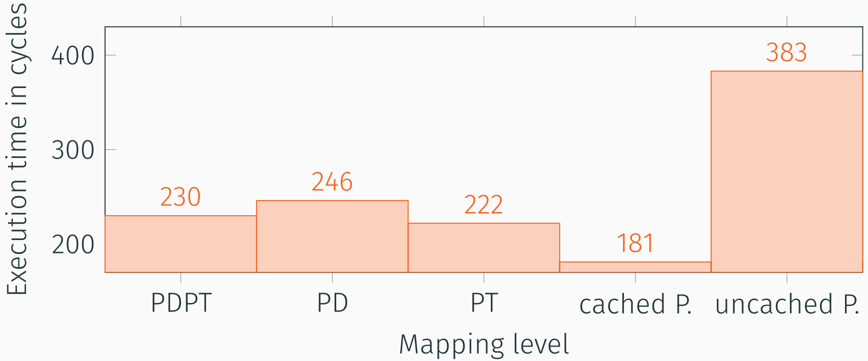
- cache hit → physical address in kernel mapping is the correct translation

Address-translation oracle



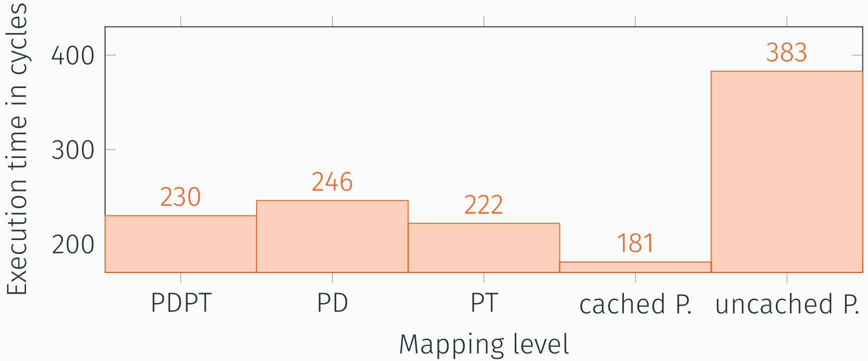
Translation-level oracle

Exploiting property #2



Translation-level oracle

Exploiting property #2



- timing depends on where the translation stops

Prefetch side-channel attacks

Using the two oracles

Prefetch side-channel attacks

Using the two oracles

- variants of cache attacks (e.g., Flush+Prefetch)

Prefetch side-channel attacks

Using the two oracles

- variants of cache attacks (e.g., Flush+Prefetch)
- Rowhammer attacks on privileged addresses

Prefetch side-channel attacks

Using the two oracles

- variants of cache attacks (e.g., Flush+Prefetch)
- Rowhammer attacks on privileged addresses
- recovering translation levels of a process ($\rightarrow$ `/proc/pid/pagemap`)

  $\rightarrow$ now privileged $\rightarrow$ bypasses ASLR

Prefetch side-channel attacks

Using the two oracles

- variants of cache attacks (e.g., Flush+Prefetch)
- Rowhammer attacks on privileged addresses
- recovering translation levels of a process ($\rightarrow$ `/proc/pid/pagemap`)

  $\rightarrow$ now privileged $\rightarrow$ bypasses ASLR
- translating virtual addresses to physical addresses ($\rightarrow$ `/proc/pid/pagemap`)

  $\rightarrow$ now privileged $\rightarrow$ re-enables ret2dir exploits

Prefetch side-channel attacks

Using the two oracles

- variants of cache attacks (e.g., Flush+Prefetch)
- Rowhammer attacks on privileged addresses
- recovering translation levels of a process (→ `/proc/pid/pagemap`)
  - → now privileged → bypasses ASLR
- translating virtual addresses to physical addresses (→ `/proc/pid/pagemap`)
  - → now privileged → re-enables ret2dir exploits
- locating kernel drivers
  - → bypasses KASLR

Defeating KASLR by locating kernel driver (1)

For all mapped pages, found with the translation-level oracle

Defeating KASLR by locating kernel driver (1)

For all mapped pages, found with the translation-level oracle

1. evict translation caches: `Sleep()` / access large memory buffer

Defeating KASLR by locating kernel driver (1)

For all mapped pages, found with the translation-level oracle

1. evict translation caches: `Sleep()` / access large memory buffer
2. perform syscall to driver

Defeating KASLR by locating kernel driver (1)

For all mapped pages, found with the translation-level oracle

1. evict translation caches: `Sleep()` / access large memory buffer
2. perform syscall to driver
3. time `prefetch(page address)`

Defeating KASLR by locating kernel driver (1)

For all mapped pages, found with the translation-level oracle

1. evict translation caches: `Sleep()` / access large memory buffer
2. perform syscall to driver
3. time `prefetch(page address)`

→ fastest average access time is a driver page

Defeating KASLR by locating kernel driver (1)

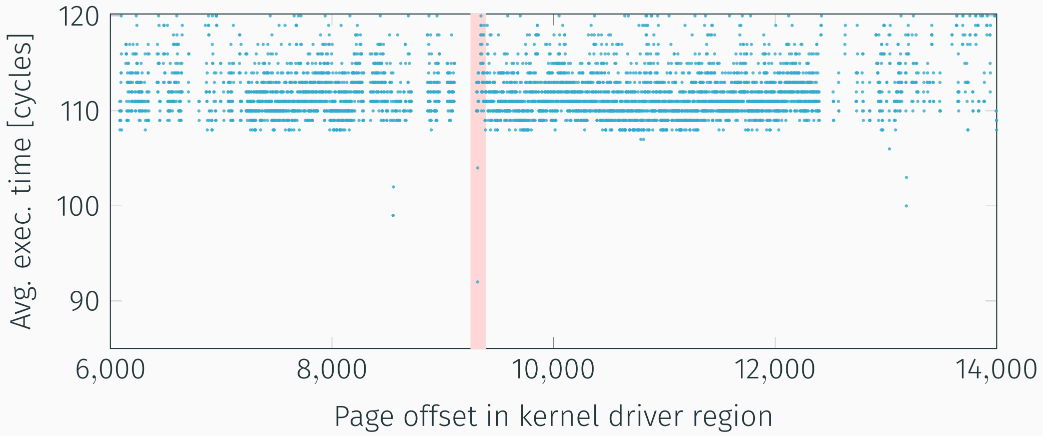
For all mapped pages, found with the translation-level oracle

1. evict translation caches: `Sleep()` / access large memory buffer
2. perform syscall to driver
3. time `prefetch(page address)`

$\rightarrow$ fastest average access time is a driver page

Full attack on Windows 10 in $<$ 12 seconds

Defeating KASLR by locating kernel driver (2)



That’s not all folks!

`rdseed` and floating point operations

- `rdseed`
  - request a random seed to the hardware random number generator
  - fixed number of precomputed random bits, takes time to regenerate them

  $\rightarrow$ covert channel

D. Evtyushkin and D. Ponomarev. “Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations”. In: *CCS'16*. 2016

M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. “On subnormal floating point and abnormal timing”. In: *S&P'15*. 2015

`rdseed` and floating point operations

- `rdseed`
  - request a random seed to the hardware random number generator
  - fixed number of precomputed random bits, takes time to regenerate them

  → covert channel
- `fadd`, `fmul`
  - floating point operations
  - running time depends on the operands

  → bypassing Firefox's same origin policy via SVG filter timing attack

D. Evtyushkin and D. Ponomarev. "Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations". In: *CCS'16*. 2016

M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. "On subnormal floating point and abnormal timing". In: *S&P'15*. 2015

`jmp` and TSX instructions

- `jmp`
  - branch prediction and branch target prediction
  - → covert channels, side-channel attacks on crypto, bypassing kernel ASLR

O. Acıiçmez, J.-P. Seifert, and c. K. Koç. “Predicting secret keys via branch prediction”. In: *CT-RSA 2007*. 2007

D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. . In: *MICRO’16*. 2016

Y. Jang, S. Lee, and T. Kim. “Breaking kernel address space layout randomization with intel TSX”. . In: *CCS’16*. 2016

`jmp` and TSX instructions

- `jmp`
  - branch prediction and branch target prediction

  → covert channels, side-channel attacks on crypto, bypassing kernel ASLR
- TSX instructions
  - extension for hardware transactional memory support

  → bypassing kernel ASLR

O. Acıiçmez, J.-P. Seifert, and c. K. Koç. “Predicting secret keys via branch prediction”. In: *CT-RSA 2007*. 2007

D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. . In: *MICRO'16*. 2016

Y. Jang, S. Lee, and T. Kim. “Breaking kernel address space layout randomization with intel TSX”. . In: *CCS'16*. 2016

Conclusion

Conclusion

- more a problem of CPU design than Instruction Set Architecture

Conclusion

- more a problem of CPU design than Instruction Set Architecture
- hard to patch $\rightarrow$ issues linked to performance optimizations

Conclusion

- more a problem of CPU design than Instruction Set Architecture
- hard to patch $\rightarrow$ issues linked to performance optimizations
- quick fixes like removing instructions won’t work

Conclusion

- more a problem of CPU design than Instruction Set Architecture
- hard to patch → issues linked to performance optimizations
- quick fixes like removing instructions won't work

→ we keep finding new instructions that leak information

What could possibly go wrong with <insert x86 instruction here>?

Clémentine Maurice, Moritz Lipp

December 2016—33rd Chaos Communication Congress