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
• 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
• title says this is a talk about x86 instructions but...
• ... this is *not a talk about software*
• 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
• 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
• 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
• 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
• 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
• 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

Diagram:

- Core 0, Core 1, Core 2, Core 3
- L1, L1, L1, L1
- L2, L2, L2, L2
- LLC slice 0, LLC slice 1, LLC slice 2, LLC slice 3
- Ring bus
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

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>16</th>
<th>17</th>
<th>25</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Index Index Index Index Index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset Offset Offset Offset Offset Offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Cache Diagram](attachment:cache_diagram.png)
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

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!
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

That’s all the assembly you need for today!
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!
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—Move

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Log Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>88 /r</td>
<td>MOV r/m8,r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r8 to r/m8.</td>
</tr>
<tr>
<td>REX + 88 /r</td>
<td>MOV r/m8 *** r8 ***</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r8 to r/m8.</td>
</tr>
<tr>
<td>89 /r</td>
<td>MOV r/m16,r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r16 to r/m16.</td>
</tr>
<tr>
<td>89 /r</td>
<td>MOV r/m32,r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r32 to r/m32.</td>
</tr>
<tr>
<td>REX.W + 89 /r</td>
<td>MOV r/m64,r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r64 to r/m64.</td>
</tr>
<tr>
<td>8A /r</td>
<td>MOV r8,r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m8 to r8.</td>
</tr>
<tr>
<td>REX + 8A /r</td>
<td>MOV r8 *** ,r/m8 ***</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r/m8 to r8.</td>
</tr>
<tr>
<td>8B /r</td>
<td>MOV r16,r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m16 to r16.</td>
</tr>
<tr>
<td>8B /r</td>
<td>MOV r32,r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m32 to r32.</td>
</tr>
<tr>
<td>REX.W + 8B /r</td>
<td>MOV r/m64,r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r/m64 to r64.</td>
</tr>
<tr>
<td>8C /r</td>
<td>MOV r/m16,SReg**</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move segment register to r/m16.</td>
</tr>
<tr>
<td>REX.W + 8C /r</td>
<td>MOV r/m64,SReg**</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move zero extended 16-bit segment register to r/m64.</td>
</tr>
<tr>
<td>8E /r</td>
<td>MOV SReg,r/m16**</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m16 to segment register.</td>
</tr>
<tr>
<td>REX.W + 8E /r</td>
<td>MOV SReg,r/m64**</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move lower 16 bits of r/m64 to segment register.</td>
</tr>
<tr>
<td>A0</td>
<td>MOV AL,moffs8*</td>
<td>FD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move byte at (seg:offset) to AL.</td>
</tr>
<tr>
<td>REX.W + A0</td>
<td>MOV AL,moffs8*</td>
<td>FD</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move byte at (offset) to AL.</td>
</tr>
<tr>
<td>A1</td>
<td>MOV AX,moffs16*</td>
<td>FD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move word at (seg:offset) to AX.</td>
</tr>
<tr>
<td>A1</td>
<td>MOV EAX,moffs32*</td>
<td>FD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move doubleword at (seg:offset) to EAX.</td>
</tr>
<tr>
<td>REX.W + A1</td>
<td>MOV RAX,moffs64*</td>
<td>FD</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move quadword at (offset) to RAX.</td>
</tr>
<tr>
<td>A2</td>
<td>MOV RAX,CR*</td>
<td>RD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move from the control register (CR).</td>
</tr>
</tbody>
</table>
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!
Data can reside in:

- CPU Registers
- L1 Cache
- L2 Cache
- Memory
- Disk storage
Memory Hierarchy

- Data can reside in
  - CPU registers
• 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
Cache attacks

- cache attacks → exploit timing differences of memory accesses
Cache attacks

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

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

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

**Step 1:** Attacker maps shared library (shared memory, in cache)
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
  - lines of the receiver still in cache → fast access
Application #1: Covert channel

- sender and receiver agree on one set
- receiver probes the set continuously
- sender transmits ‘0’ doing nothing
  → lines of the receiver still in cache → fast access
- sender transmits ‘1’ accessing addresses in the set
  → evicts lines of the receiver → slow access
Application #1: Covert channel

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

---

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 $x^{(r)} = (x_0^{(r)}, \ldots, x_{15}^{(r)})$ at each round $r$
  - first round, accessed table indices are

\[ x_i^{(0)} = p_i \oplus k_i \quad \text{for all } i = 0, \ldots, 15 \]
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 \( x^{(r)} = (x_0^{(r)}, \ldots, x_{15}^{(r)}) \) at each round \( r \)
  - first round, accessed table indices are

\[
x_i^{(0)} = p_i \oplus k_i \quad \text{for all } i = 0, \ldots, 15
\]

\( \Rightarrow \) recovering **accessed table indices** \( \Rightarrow \) recovering the key
Application #2: Crypto side-channel attack

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

Application #3: Spying on keystrokes

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

Demo time!

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

Diagram:

- core 0
- core 1
- L1
- L2
- LLC
clflush timing leakage: Part #1

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

![Diagram showing clflush operations on cores and cache levels](image-url)
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` 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
It’s only a few cycles, what could go wrong?!

- new cache attack: **Flush+Flush**

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

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

---

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

---

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

---

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. \texttt{CACHE\_MISSES} → occur after data is flushed
2. \texttt{CACHE\_REFERENCES} → 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. \texttt{CACHE\_MISSES} → occur after data is flushed
2. \texttt{CACHE\_REFERENCES} → occur when reaccessing memory

... without false positives

---

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

• 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 → occur after data is flushed
2. CACHE_REFERENCES → occur when reaccessing memory

... without false positives

- heavy activity on the cache
- also, very short loops of code → 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 → occur after data is flushed
2. CACHE_REFERENCES → occur when reaccessing memory

... without false positives

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

→ 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

<table>
<thead>
<tr>
<th>technique</th>
<th>packet size (KBps)</th>
<th>capacity</th>
<th>receiver stealth</th>
<th>sender stealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush+Flush</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Flush+Flush: Covert channel

<table>
<thead>
<tr>
<th>technique</th>
<th>packet size</th>
<th>capacity (KBps)</th>
<th>receiver stealth</th>
<th>sender stealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush+Flush</td>
<td>28</td>
<td>496</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>28</td>
<td>298</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Flush+Flush: Covert channel

<table>
<thead>
<tr>
<th>technique</th>
<th>packet size</th>
<th>capacity (KBps)</th>
<th>receiver stealth</th>
<th>sender stealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush+Flush</td>
<td>28</td>
<td>496</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>28</td>
<td>298</td>
<td></td>
<td>✗</td>
</tr>
</tbody>
</table>
## Flush+Flush: Covert channel

<table>
<thead>
<tr>
<th>technique</th>
<th>packet size</th>
<th>capacity (KBps)</th>
<th>receiver stealth</th>
<th>sender stealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush+Flush</td>
<td>28</td>
<td>496</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>28</td>
<td>298</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
## Flush+Flush: Covert channel

<table>
<thead>
<tr>
<th>technique</th>
<th>packet size</th>
<th>capacity (KBps)</th>
<th>receiver stealth</th>
<th>sender stealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush+Flush</td>
<td>28</td>
<td>496</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>28</td>
<td>298</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
### Flush+Flush: Covert channel

<table>
<thead>
<tr>
<th>technique</th>
<th>packet size</th>
<th>capacity (KBps)</th>
<th>receiver stealth</th>
<th>sender stealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush+Flush</td>
<td>28</td>
<td>496</td>
<td>✅</td>
<td>✗</td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>28</td>
<td>298</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flush+Flush</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prime+Probe</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Flush+Flush: Covert channel

<table>
<thead>
<tr>
<th>technique</th>
<th>packet size</th>
<th>capacity (KBps)</th>
<th>receiver stealth</th>
<th>sender stealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush+Flush</td>
<td>28</td>
<td>496</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>28</td>
<td>298</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>4</td>
<td>54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flush+Flush</td>
<td>4</td>
<td>52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prime+Probe</td>
<td>4</td>
<td>34</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Flush+Flush: Covert channel

<table>
<thead>
<tr>
<th>technique</th>
<th>packet size</th>
<th>capacity (KBps)</th>
<th>receiver stealth</th>
<th>sender stealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush+Flush</td>
<td>28</td>
<td>496</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>28</td>
<td>298</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>4</td>
<td>54</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Flush+Flush</td>
<td>4</td>
<td>52</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Prime+Probe</td>
<td>4</td>
<td>34</td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>
## Flush+Flush: Covert channel

<table>
<thead>
<tr>
<th>technique</th>
<th>packet size</th>
<th>capacity (KBps)</th>
<th>receiver stealth</th>
<th>sender stealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush+Flush</td>
<td>28</td>
<td>496</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>28</td>
<td>298</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>4</td>
<td>54</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Flush+Flush</td>
<td>4</td>
<td>52</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Prime+Probe</td>
<td>4</td>
<td>34</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
## Flush+Flush: Covert channel

<table>
<thead>
<tr>
<th>technique</th>
<th>packet size</th>
<th>capacity (KBps)</th>
<th>receiver stealth</th>
<th>sender stealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush+Flush</td>
<td>28</td>
<td>496</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>28</td>
<td>298</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Flush+Reload</td>
<td>4</td>
<td>54</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Flush+Flush</td>
<td>4</td>
<td>52</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Prime+Probe</td>
<td>4</td>
<td>34</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Flush+Flush: Side channel on AES T-tables (1)

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

<table>
<thead>
<tr>
<th>technique</th>
<th>number of encryptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush+Reload</td>
<td>250</td>
</tr>
<tr>
<td>Flush+Flush</td>
<td>350</td>
</tr>
<tr>
<td>Prime+Probe</td>
<td>4800</td>
</tr>
</tbody>
</table>

→ 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)

<table>
<thead>
<tr>
<th>technique</th>
<th>time (s)</th>
<th>stealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush+Reload</td>
<td>215</td>
<td>✗</td>
</tr>
<tr>
<td>Prime+Probe</td>
<td>234</td>
<td>✗</td>
</tr>
<tr>
<td>Flush+Flush</td>
<td>163</td>
<td>✓</td>
</tr>
</tbody>
</table>

→ 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)

Physical address:
- Tag: 35
- Set: 17
- Offset: 6

Offset settagphysical address:
- 30

H

Line:
- Slice 0
- Slice 1
- Slice 2
- Slice 3

0 61735

11
• last-level cache → as many slices as cores
• undocumented hash function that maps a physical address to a slice
• designed for performance

For $2^k$ slices:

- physical address: 30 bits
- slice $(o_0, \ldots, o_{k-1})$: $k$ bits
Let’s go back to **clflush**!
clflush timing leakage: Part #2

- **clflush** faster to reach a line on the local slice
• `clflush` faster to reach a line on the local slice
• map physical addresses to slices

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

---

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
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.
NOTE

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

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.
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.

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.

---

A little bit more background before continuing...
Address translation

CR3

PML4
  PML4E 0
  PML4E 1
  ...
  #PML4I
  ...
  PML4E 511

PDPT
  PDPTE 0
  PDPTE 1
  ...
  #PDPTI
  ...
  PDPTE 511

Page Directory
  PDE 0
  PDE 1
  ...
  PDE #PDI
  ...
  PDE 511

Page Table
  PTE 0
  PTE 1
  ...
  PTE #PTI
  ...
  PTE 511

4 KiB Page
  Byte 0
  Byte 1
  ...
  Offset
  ...
  Byte 4095

48-bit virtual address

PML4I (9 b)  PDPTI (9 b)  PDI (9 b)  PTI (9 b)  Offset (12 b)
Address translation caches

Core 0
- ITLB
- DTLB
- PDE cache
- PDPTPE cache
- PML4E cache

Core 1
- ITLB
- DTLB
- PDE cache
- PDPTPE cache
- PML4E cache

Page table structures in system memory (DRAM)
Today’s operating systems:

Shared address space

User memory

Kernel memory

context switch
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!
• tells the CPU “I might need that later”

prefetch: Unusual instructions (1)

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

prefetch: Unusual instructions (1)

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

**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

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

---

Exploiting property #1 + kernel direct-physical map

Physical memory

Virtual address space

Cache
Address-translation oracle

Exploiting property #1 + kernel direct-physical map

Physical memory

Virtual address space

Cache

0

\(2^{47}\)

\(-2^{47}\)

\(-1\)

max. phys.

direct map

cached

cached

physical address in kernel mapping is the correct translation
Address-translation oracle

Exploiting property #1 + kernel direct-physical map

Physical memory

Virtual address space

Cache

flush
Exploiting property #1 + kernel direct-physical map

Physical memory

Virtual address space

Cache

0

\(2^{47}\)

\(-2^{47}\)

\(-1\)

max. phys.

direct map

prefetch
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

![Bar chart showing execution time in cycles for different mapping levels.]

- **PDPT**: 230 cycles
- **PD**: 246 cycles
- **PT**: 222 cycles
- **cached P.**: 181 cycles
- **uncached P.**: 383 cycles
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)

- 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
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 (→ `/proc/pid/pagemap`) → now privileged → bypasses ASLR
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
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
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)`
   → fastest average access time is a driver page

Full attack on Windows 10 in < 12 seconds
Defeating KASLR by locating kernel driver (2)

Page offset in kernel driver region

Avg. exec. time [cycles]
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
  → covert channel

---


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

---


jmp and TSX instructions

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

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

---


Conclusion
Conclusion

- more a problem of CPU design than Instruction Set Architecture
• more a problem of **CPU design** than Instruction Set Architecture
• hard to patch → issues linked to performance **optimizations**
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
• 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