Software-based Microarchitectural Attacks

Daniel Gruss
January 10, 2018

Graz University of Technology
• Daniel Gruss
• Post-Doc @ Graz University of Technology
• Twitter: @lavados
• Email: daniel.gruss@iaik.tugraz.at
• security and privacy rely on secrets (unknown to attackers)

• secrets can leak through side channels
• security and privacy rely on secrets (unknown to attackers)

• secrets can leak through side channels

• software-based → no physical access
Wall is built by Operating System using Hardware Features
Wall disappears because hardware is broken
How to be fast at ...

... cooking:

- move faster?
How to be fast at ... cooking:

- move faster? yes, helps
... cooking:

- move faster? yes, helps
- mise en place = have everything prepared
• Not everything at home?
• Not everything at home? → fetch from grocery store
• Something I run into every time:
• Not everything at home? → fetch from grocery store
• Something I run into every time:
  • “[...] Now add the peeled and cooked potatoes [...]”
There are the potatoes...
• Have to wait an hour until potatoes are ready...
What now?

• Have to wait an hour until potatoes are ready... → **Latency**
• Have to wait an hour until potatoes are ready... → Latency
• Modern processors avoid latency mostly using two mechanisms:
What now?

- Have to wait an hour until potatoes are ready... → **Latency**
- Modern processors avoid latency mostly using two mechanisms:
  - Caching
What now?

- Have to wait an hour until potatoes are ready... \(\rightarrow\) **Latency**
- Modern processors avoid latency mostly using two mechanisms:
  - Caching
    - \(\rightarrow\) do not run to grocery every time, but keep (a stock of) most ingredients ready at home
• Have to wait an hour until potatoes are ready... ⇒ **Latency**

• Modern processors avoid latency mostly using two mechanisms:
  • Caching
    • do not run to grocery every time, but keep (a stock of) most ingredients ready at home
  • Parallelism
• Have to wait an hour until potatoes are ready... → **Latency**

• Modern processors avoid latency mostly using two mechanisms:
  • Caching
    • → do not run to grocery every time, but keep (a stock of) most ingredients ready at home
  • Parallelism
    • → cook potatoes in time before they are needed
• buffer frequently used memory

• every memory reference goes through the cache

• transparent to OS and programs
• buffer frequently used memory

• every memory reference goes through the cache

• transparent to OS and programs
Memory Address
### Directly Mapped Cache

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Directly Mapped Cache

Memory Address

<table>
<thead>
<tr>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
## Directly Mapped Cache

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tag</td>
</tr>
<tr>
<td></td>
<td>Data</td>
</tr>
</tbody>
</table>

Daniel Gruss — Graz University of Technology
Directly Mapped Cache

Memory Address

<table>
<thead>
<tr>
<th></th>
<th>$b$ bits</th>
</tr>
</thead>
</table>

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$2^b$ bytes
Directly Mapped Cache

Memory Address

\[
\begin{array}{c|c}
 n \text{ bits} & b \text{ bits} \\
\end{array}
\]

Cache

\[
\begin{array}{c|c}
 \text{Tag} & \text{Data} \\
\hline
 & \\
 & \\
 & \\
\end{array}
\]

Cache Index

\[2^b \text{ bytes}\]
Directly Mapped Cache

Memory Address

\[ n \text{ bits} \quad b \text{ bits} \]

Cache

\[ \begin{array}{|c|c|}
\hline
\text{Tag} & \text{Data} \\
\hline
\end{array} \]

\[ 2^n \text{ cache lines} \]

Cache Index

\[ 2^b \text{ bytes} \]
Directly Mapped Cache

![Diagram of Directly Mapped Cache](image.png)

- Memory Address
  - Memory Address
  - Tag: \( f \) bits
  - Cache Index: \( n \) bits
  - Tag: \( b \) bits

- Cache
  - \( 2^n \) cache lines
  - Tag: \( 2^n \) entries
  - Data: \( 2^b \) bytes

- Hit/Miss
  - Tag match: Hit
  - Tag mismatch: Miss
Directly Mapped Cache

Problem: working on congruent addresses
2-way Set-Associative Cache

Memory Address

\[ f \]

\[ n \text{ bits} \rightarrow \text{Cache Index} \rightarrow 2^n \text{ cache lines} \]

\[ b \text{ bits} \]

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ 2^n \text{ cache lines} \]
2-way Set-Associative Cache

Memory Address

- f
- n bits
- b bits

Cache Index

Cache

- Way 1 Tag
- Way 2 Tag
- Way 1 Data
- Way 2 Data

$2^n$ cache sets
2-way Set-Associative Cache

Memory Address

\[ f \]

\[ n \text{ bits} \quad b \text{ bits} \]

Cache Index

\[ \text{Cache} \]

\[ 2^n \text{ cache sets} \]

Way 1 Tag \quad Way 1 Data

Way 2 Tag \quad Way 2 Data

Tag

= ?

= ?
2-way Set-Associative Cache

Memory Address

\[ f \]

\[ \text{Cache Index} \]

\[ \text{Tag} \]

\[ n \text{ bits} \]

\[ b \text{ bits} \]

\[ 2^n \text{ cache sets} \]

\[ \text{Way 1 Tag} \]

\[ \text{Way 2 Tag} \]

\[ \text{Way 1 Data} \]

\[ \text{Way 2 Data} \]

\[ \rightarrow \text{replacement policy is important} \]
Data and Instruction Caches

- core 0
  - L1
  - L2

- core 1
  - L1
  - L2

- core 2
  - L1
  - L2

- core 3
  - L1
  - L2

- LLC slice 0
- LLC slice 1
- LLC slice 2
- LLC slice 3

last-level cache:
- shared
- inclusive
last-level cache:
- *shared*
- *inclusive*

→ shared memory is shared in cache, across cores!
Flush+Reload

Attacker address space

Cache

Victim address space
Flush+Reload

Attacker address space

<table>
<thead>
<tr>
<th>cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>cached</td>
</tr>
</tbody>
</table>

Cache

Victim address space
Flush+Reload

Attacker address space

Cache

flushes

Victim address space

Daniel Gruss — Graz University of Technology
Flush+Reload

Attacker address space

reloads data

Cache

Victim address space
Cache Template Attack Demo

```
% sleep 2; /sys 300 710514604000-710514176000 r-xp 0x200000 00:02 26 0000
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so
```
## Key

<table>
<thead>
<tr>
<th>Character</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>0x7c680</td>
</tr>
<tr>
<td>h</td>
<td>0x7c6c0</td>
</tr>
<tr>
<td>i</td>
<td>0x7c700</td>
</tr>
<tr>
<td>j</td>
<td>0x7c740</td>
</tr>
<tr>
<td>k</td>
<td>0x7c780</td>
</tr>
<tr>
<td>l</td>
<td>0x7c7c0</td>
</tr>
<tr>
<td>m</td>
<td>0x7c800</td>
</tr>
<tr>
<td>n</td>
<td>0x7c840</td>
</tr>
<tr>
<td>o</td>
<td>0x7c880</td>
</tr>
<tr>
<td>p</td>
<td>0x7c8c0</td>
</tr>
<tr>
<td>q</td>
<td>0x7c900</td>
</tr>
<tr>
<td>r</td>
<td>0x7c940</td>
</tr>
<tr>
<td>s</td>
<td>0x7c980</td>
</tr>
<tr>
<td>t</td>
<td>0x7cc0</td>
</tr>
<tr>
<td>u</td>
<td>0x7ca0</td>
</tr>
<tr>
<td>v</td>
<td>0x7cb80</td>
</tr>
<tr>
<td>w</td>
<td>0x7cc40</td>
</tr>
<tr>
<td>x</td>
<td>0x7ccc0</td>
</tr>
<tr>
<td>y</td>
<td>0x7ccd0</td>
</tr>
<tr>
<td>z</td>
<td>0x7cd00</td>
</tr>
</tbody>
</table>
Can we break the wall?
- Fetch data now, check later
Parallelism: Running ahead
We will see in a minute how this turns into a problem!
Why address translation: Run multiple processes securely on a single CPU

- Let applications run in their own virtual address space
- Create exchangeable map from “virtual memory” to “physical memory”
- Privileges are checked on memory accesses
- Managed by the operating system kernel
Address translation on x86-64

48-bit virtual address

CR3

PML4
PML4E 0
PML4E 1

···

#PML4I

PML4E 511

PDPT
PDPT 0
PDPT 1

···

#PDPTI

PDPT 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

PML4I (9 b)
PDPTI (9 b)
PDI (9 b)
PTI (9 b)
Offset (12 b)
Problem: translation tables are stored in physical memory
Solution: Address Translation Caches

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

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

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

Shared address space

User memory

Kernel memory

context switch
Address-Space Layout Randomization (ASLR)

- Kernel and drivers at randomized offsets in virtual memory
- Mitigates code reuse attacks e.g. return-oriented-programming
- Attacks based on read primitives or write primitives
Address-Space Layout Randomization (ASLR)

- Kernel and drivers at randomized offsets in virtual memory
- Mitigates code reuse attacks e.g. return-oriented-programming
- Attacks based on read primitives or write primitives
- But: leaking kernel/driver addresses defeats ASLR
Kernel direct-physical map

Virtual address space

Physical memory

max. phys.

0

User

Kernel

0

$2^{47}$

$-2^{47}$

$-1$

direct map

0

0 max. phys.

Daniel Gruss — Graz University of Technology
Available on many operating systems / hypervisors

- OS X, Linux, BSD
- Xen PVM (Amazon EC2)
- A large fraction also on Windows (not directly though)
Where is the kernel?
Where is the kernel?
Where is the kernel?

Not 7 possibilities, but up to $2^{47}$. 
Prefetch: Locate Kernel Driver (defeat KASLR)
Prefetch: Kernel Memory Layout

Virtual address space

Physical memory

max. phys.

direct map

User

Kernel

2^{47}

−2^{47}

0

0

max. phys.

2^{47}

−1

Daniel Gruss — Graz University of Technology
Prefetching Kernel Addresses

![Graph showing the minimum access latency against page offset in the kernel direct map. The x-axis represents the page offset, and the y-axis represents the minimum access latency. The graph displays a sharp drop in latency at a specific offset.](image-url)
Today’s operating systems:

Shared address space

- User memory
- Kernel memory

context switch

Stronger kernel isolation:

User address space

- User memory
- Not mapped

context switch

Kernel address space

- Not mapped
- Kernel memory

Interrupt dispatcher

Daniel Gruss — Graz University of Technology
How to leak a secret indirectly?

Exception Handling/Suppression

Transient Instructions

Microarchitectural State Change

Accessed

Secret

Leaked

Architectural State

Transfer (Covert Channel)

Recovered Secret

Recovery

Section 4.1

Section 4.2

Daniel Gruss — Graz University of Technology
register = *secret_from_kernel; // read secret into register
user_accessible_array[register] = 123; // use register as index to array

... and afterwards ...

index = flushreload(user_accessible_array);

// now we know the following:
index == *secret_from_kernel
• Let processor run ahead (parallelism!)
• Access secret and transmit it via Flush+Reload (caching!)
• Operations are unrolled, but not the cache state
asm volatile("movq (%rsi), %rsi\n"
  "movb (%rbx), %al\n"
  "shl $12, %rax\n"
  "mov (%rdx,%rax,1), %rcx\n"
: : "a"(0x1FF), "c"(vaddr), "b"(iaddr), "d"(array_ptr), "S"(0));
• Right path?
• Right path?
  • Good!
• Right path?
  • Good!
• Wrong path?
• Right path?
  • Good!
• Wrong path?
  • Go back and continue from junction
if (x < array1_size)
    y = array2[array1[x] * 4096];
Similar as with Meltdown:
• Similar as with Meltdown:
  • Side effects are not unrolled
Similar as with Meltdown:
- Side effects are not unrolled
- Anyone can see your footsteps on the road
Leaking a picture like in CSI Cyber

meltdown@meltdown ~/ppm2 % taskset 1 ./imgdump 0x375a00000 14919 > output.flif
Reading from 0xffff880375a00000

I
Leaking a photo
Spying on passwords
Leaking Passwords from your Password Manager

Daniel Gruss — Graz University of Technology
1. microarchitectural attacks can be *widely automated*
1. microarchitectural attacks can be *widely automated*
   - as seen in the videos
1. microarchitectural attacks can be *widely automated*
   - as seen in the videos
2. unknown and *novel side channels* are likely to exist
1. microarchitectural attacks can be *widely automated*
   - as seen in the videos
2. unknown and *novel side channels* are likely to exist
1. microarchitectural attacks can be *widely automated*
   - as seen in the videos
2. unknown and *novel side channels* are likely to exist
   - as we’ve shown with Meltdown and Spectre
1. microarchitectural attacks can be *widely automated*
   - as seen in the videos
2. unknown and *novel side channels* are likely to exist
   - as we’ve shown with Meltdown and Spectre
3. *minimal requirements* enable attacks through websites
1. microarchitectural attacks can be *widely automated*
   - as seen in the videos
2. unknown and *novel side channels* are likely to exist
   - as we’ve shown with Meltdown and Spectre
3. *minimal requirements* enable attacks through websites
1. microarchitectural attacks can be *widely automated*
   - as seen in the videos
2. unknown and *novel side channels* are likely to exist
   - as we’ve shown with Meltdown and Spectre
3. *minimal requirements* enable attacks through websites
   - no permissions or anything required for these attacks
1. microarchitectural attacks can be *widely automated*
   - as seen in the videos
2. unknown and *novel side channels* are likely to exist
   - as we’ve shown with Meltdown and Spectre
3. *minimal requirements* enable attacks through websites
   - no permissions or anything required for these attacks
4. constructing countermeasures is difficult and requires solid understanding of attacks
1. microarchitectural attacks can be *widely automated*
   - as seen in the videos

2. unknown and *novel side channels* are likely to exist
   - as we’ve shown with Meltdown and Spectre

3. *minimal requirements* enable attacks through websites
   - no permissions or anything required for these attacks

4. constructing countermeasures is difficult and requires solid understanding of attacks
   - call it luck or call it a good idea: our KAISER patch protected against Meltdown before Meltdown was discovered
Software-based
Microarchitectural Attacks

Daniel Gruss
January 10, 2018
Graz University of Technology