(Why) Are Microarchitectural Attacks Really Different than Physical Side-Channel Attacks?

Daniel Gruss

September 10, 2018

Graz University of Technology
Amazon Prime + Probe

Rowhammer.js

Another Flip in the Row

Fantastic Timers

And Where to Find Them

High-Resolution Microarchitectural Attacks in JavaScript

JavaScript Zero

Real JavaScript and Zero Side-Channel Attacks
I HAVE NO IDEA WHAT I'M DOING
Americoin,

Americoin

God shed his blocks on thee!

Americoin, Americoin, God shed his blocks on thee
Stealing Bitcoins?

Daniel Gruss — Graz University of Technology
Stealing Bitcoins?
Stealing Bitcoins?

Daniel Gruss — Graz University of Technology
Stealing Bitcoins?

Daniel Gruss — Graz University of Technology
Application

Untrusted part

Operating System
Application

Untrusted part

Create Enclave

Trusted part

Call Gate

Trusted Fnc.

Operating System

Daniel Gruss — Graz University of Technology
Application

Untrusted part

Create Enclave

Call Truste Fnc.

Call Gate

Trusted part

Truste Fnc.

Operating System
Application

Untrusted part

Create Enclave

Call Trusted Fnc.

Trusted part

Call Gate

Trusted Fnc.

Operating System
Application

Untrusted part

Create Enclave

Call Trusted Fnc.

Trusted part

Call Gate

Trusted Fnc.

Operating System
Create Enclave → Call Trusted Fnc.

Call Gate

Trusted Fnc. → Return

Untrusted part

Trusted part

Application

Operating System
Application

Untrusted part

Create Enclave

Call Trusted Fnc.

Trusted part

Call Gate

Trusted Fnc.

Return

Operating System

Daniel Gruss — Graz University of Technology
Application

Untrusted part

Create Enclave

Call Trusted Fnc.

... 

Trusted part

Call Gate

Trusted Fnc.

Return

Operating System

Daniel Gruss — Graz University of Technology
Application

Untrusted part

Create Enclave

Call Trusted Fnc.

...

Trusted part

Call Gate

Trusted Fnc.

Return

Operating System

Daniel Gruss — Graz University of Technology
Protection from Side-Channel Attacks

Intel SGX does not provide explicit protection from side-channel attacks. It is the enclave developer’s responsibility to address side-channel attack concerns.
Protection from Side-Channel Attacks
Protection from Side-Channel Attacks

Intel SGX does not provide explicit protection from side-channel attacks.
Protection from Side-Channel Attacks

Intel SGX does not provide explicit protection from side-channel attacks. It is the enclave developer’s responsibility to address side-channel attack concerns.
CAN'T BREAK YOUR SIDE-CHANNEL PROTECTIONS

IF YOU DON'T HAVE ANY
- Ledger SGX Enclave for blockchain applications
- BitPay Copay Bitcoin wallet
- Teechain payment channel using SGX
SGX Wallets

- Ledger SGX Enclave for blockchain applications
- BitPay Copay Bitcoin wallet
- Teechain payment channel using SGX

Teechain

[...] We assume the TEE guarantees to hold
SGX Wallets

- Ledger SGX Enclave for blockchain applications
- BitPay Copay Bitcoin wallet
- Teechain payment channel using SGX

**Teechain**

[...] We assume the TEE guarantees to hold and do not consider side-channel attacks [5, 35, 46] on the TEE.
- Ledger SGX Enclave for blockchain applications
- BitPay Copay Bitcoin wallet
- Teechain payment channel using SGX

**Teechain**

[...] We assume the TEE guarantees to hold and do not consider side-channel attacks [5, 35, 46] on the TEE. Such attacks and their mitigations [36, 43] are outside the scope of this work. [...]
Attacking a weak RSA implementation inside SGX

Raw Prime+Probe trace...¹

...processed with a simple moving average... \(^2\)

...allows to clearly see the bits of the exponent\(^3\)

YOU CAN'T DO THAT!

THAT'S AGAINST THE RULES!
WANT TO DISCUSS THREAT MODELS NOW?
Physical Side Channels

Power consumption \cite{KJJ99, MOP08}

Electro-magnetic radiation \cite{RR01, KS09}

Temperature \cite{HS13}

Photonic emission \cite{Sch+12, CSW17}

Acoustic emissions \cite{Bac+10}

Physical access usually relevant, but code execution on device usually not relevant
Physical Side Channels

- Power consumption [KJJ99; MOP08]
Physical Side Channels

- Power consumption [KJJ99; MOP08]
- Electro-magnetic radiation [RR01; KS09]
Physical Side Channels

- Power consumption [KJJ99; MOP08]
- Electro-magnetic radiation [RR01; KS09]
- Temperature [HS13]
Physical Side Channels

- Power consumption [KJJ99; MOP08]
- Electro-magnetic radiation [RR01; KS09]
- Temperature [HS13]
- Photonic emission [Sch+12; CSW17]
Physical Side Channels

- Power consumption [KJJ99; MOP08]
- Electro-magnetic radiation [RR01; KS09]
- Temperature [HS13]
- Photonic emission [Sch+12; CSW17]
- Acoustic emissions [Bac+10]
Physical Side Channels

- Power consumption [KJJ99; MOP08]
- Electro-magnetic radiation [RR01; KS09]
- Temperature [HS13]
- Photonic emission [Sch+12; CSW17]
- Acoustic emissions [Bac+10]

→ Physical access usually relevant, but code execution on device usually not relevant
U can't touch THIS

M C Hammer
Microarchitectural Attacks

1996
Microarchitectural Attacks

1996

2004
Microarchitectural Attacks

1996

2004

2006
Microarchitectural Attacks

1996
2004
2006
2009
Microarchitectural Attacks

1996

2004

2006

2009

2011
Microarchitectural Attacks

1996

2004

2006

2009

2011
Microarchitectural Attacks
Microarchitectural Attacks

1996

2004

2006

2009

2011

2013
Microarchitectural Attacks

1996

2004

2006

2009

2011

2013
Microarchitectural Attacks
Microarchitectural Attacks

1996
2004
2006
2009
2011
2013
2014
Microarchitectural Attacks

1996

2004

2006

2009

2011

2013

2014

Daniel Gruss — Graz University of Technology
Microarchitectural Attacks

1996

2004

2006

2009

2011

2013

2014

2015

Daniel Gruss — Graz University of Technology
Microarchitectural Attacks

2016
Microarchitectural Attacks

2016

2017
Microarchitectural Attacks

2016

2017

2018
Differences and Similarities

- threat model
- temporal component
- observer effect (destructive measurements)
- spatial component
Microarchitectural Attacks - Threat Model

1. Attacker controls code in browser sandbox (e.g., [Ore+15; GMM16])
2. Attacker cannot control any code on the system
Microarchitectural Attacks - Threat Model

- Usually no physical access
Microarchitectural Attacks - Threat Model

- Usually no physical access
- Local code
Microarchitectural Attacks - Threat Model

- Usually no physical access
- Local code
- Co-located code
• Usually no physical access
• Local code
• Co-located code
• Different meanings of “remote”
Microarchitectural Attacks - Threat Model

- Usually no physical access
- Local code
- Co-located code
- Different meanings of “remote”
  1. Attacker controls code in browser sandbox (e.g., [Ore+15; GMM16])
Microarchitectural Attacks - Threat Model

- Usually no physical access
- Local code
- Co-located code
- Different meanings of “remote”
  1. Attacker controls code in browser sandbox (e.g., [Ore+15; GMM16])
  2. Attacker cannot control any code on the system
Truly remote attacks...

Just a few examples:

- Remote timing attacks on crypto ([Ber04; BB05] and many more)
- ThrowHammer [Tat+18] and NetHammer [Lip+17]
- NetSpectre [Sch+18b]
Just a few examples:

- Remote timing attacks on crypto ([Ber04; BB05] and many more)
Truly remote attacks...

Just a few examples:

- Remote timing attacks on crypto ([Ber04; BB05] and many more)
- ThrowHammer [Tat+18] and NetHammer [Lip+17]
Truly remote attacks...

Just a few examples:

- Remote timing attacks on crypto ([Ber04; BB05] and many more)
- ThrowHammer [Tat+18] and NetHammer [Lip+17]
- NetSpectre [Sch+18b]
printf("%d", i);
printf("%d", i);
Cache miss

printf("%d", i);
printf("%d", i);
```c
printf("%d", i);
printf("%d", i);
```
CPU Cache

printf("%d", i);
printf("%d", i);
CPU Cache

```c
printf("%d", i);
printf("%d", i);
```

Cache miss

Request

Response
CPU Cache

```
printf("%d", i);
printf("%d", i);
```

Cache miss

Request

Response

Cache hit
CPU Cache

```c
printf("%d", i);
printf("%d", i);
```

Cache miss

No DRAM access, much faster

Cache hit

Request

Response
CPU Cache

DRAM access, slow

printf("%d", i);
printf("%d", i);

Cache miss

No DRAM access, much faster

Cache hit

Request

Response
Flush+Reload

ATTACKER
flush
access

Shared Memory

VICTIM
access
Flush+Reload

ATTACKER

flush
access

Shared Memory

cached

VICTIM

access

cached
Flush+Reload

ATTACKER

flush

access

Shared Memory

VICTIM

access

Shared Memory
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

Shared Memory

VICTIM

flush
access

access
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

flush

access

Shared Memory

VICTIM

access
Flush+Reload

Shared Memory

ATTACKER

flush

access

VICTIM

access

Victim accessed vs Victim did not access
Physical Side Channels
Temporal Component: Timestamps

Physical Side Channels

- theoretical maximum accuracy of $5.4 \cdot 10^{-44}$ s
Physical Side Channels

- theoretical maximum accuracy of $5.4 \cdot 10^{-44}$s
- feasible today: $850 \cdot 10^{-21}$s
Physical Side Channels

- theoretical maximum accuracy of $5.4 \cdot 10^{-44}s$
- feasible today: $850 \cdot 10^{-21}s$

Microarchitectural Attacks
Physical Side Channels

- theoretical maximum accuracy of $5.4 \cdot 10^{-44}$s
- feasible today: $850 \cdot 10^{-21}$s

Microarchitectural Attacks

- often around nanoseconds
Temporal Component: Timestamps

Physical Side Channels
- theoretical maximum accuracy of $5.4 \cdot 10^{-44}$s
- feasible today: $850 \cdot 10^{-21}$s

Microarchitectural Attacks
- often around nanoseconds
- sometimes much lower
Physical Side Channels
Physical Side Channels

- in the range of multiple GHz
Temporal Component: Sampling Rate

Physical Side Channels

- in the range of multiple GHz

Microarchitectural Attacks
Temporal Component: Sampling Rate

Physical Side Channels

- in the range of multiple GHz

Microarchitectural Attacks

- usually varying frequency (depending on the attack)
Temporal Component: Sampling Rate

Physical Side Channels

- in the range of multiple GHz

Microarchitectural Attacks

- usually varying frequency (depending on the attack)
- between a few ns (< 1 GHz) and multiple seconds (< 1 Hz) (or even worse)
Temporal Component: Sampling Rate

Physical Side Channels

- in the range of multiple GHz

Microarchitectural Attacks

- usually varying frequency (depending on the attack)
- between a few ns (< 1 GHz) and multiple seconds (< 1 Hz) (or even worse)
- strongly dependent on the specific attack
Physical Side Channels

- in the range of multiple GHz

Microarchitectural Attacks

- usually varying frequency (depending on the attack)
- between a few ns (< 1 GHz) and multiple seconds (< 1 Hz) (or even worse)
- strongly dependent on the specific attack
  - device under test = measurement device
Temporal Component: Sampling Rate

Physical Side Channels

- in the range of multiple GHz

Microarchitectural Attacks

- usually varying frequency (depending on the attack)
- between a few ns (< 1 GHz) and multiple seconds (< 1 Hz) (or even worse)
- strongly dependent on the specific attack
  - device under test = measurement device
  - observer effect
device under test = measurement device

- measuring time takes some time
- limits the resolution
- measuring cache hits/misses manipulates the cache state
- virtually all measurements are destructive
Measurement Noise

Flush+Reload has no noise except for:
- Race condition between attacker and victim (observer effect)
- Speculative execution
- Prefetching

Typically > 99.99% precision and recall
Measurement Noise

Flush+Reload has no noise except for:

- Race condition between attacker and victim (observer effect)
- Speculative execution
- Prefetching

Typically >99.99% precision and recall
Flush+Reload has no noise except for:
Flush+Reload has no noise except for:

- Race condition between attacker and victim (observer effect)
Flush+Reload has no noise except for:

- Race condition between attacker and victim (observer effect)
- Speculative execution
Flush+Reload has no noise except for:

- Race condition between attacker and victim (observer effect)
- Speculative execution
- Prefetching
Flush+Reload has no noise except for:

- Race condition between attacker and victim (observer effect)
- Speculative execution
- Prefetching
- ...
Flush+Reload has no noise except for:

- Race condition between attacker and victim (observer effect)
- Speculative execution
- Prefetching
- ...

→ Typically > 99.99% precision and recall
Measuring Processor Operations
Timing Measurements

- Very short timings
- `rdtsc` instruction: “cycle-accurate” timestamps

```c
[...]
rdtsc
function()
rdtsc
[...]
```
What are we measuring?

- Do you measure what you *think* you measure?
- *Out-of-order* execution → what is really executed

```
rdtsc  rdtsc  rdtsc
function()  [...]  rdtsc
[...]  rdtsc  function()
rdtsc  function()  [...]```
• use pseudo-serializing instruction \texttt{rdtscp} (recent CPUs)
• use pseudo-serializing instruction `rdtscp` (recent CPUs)
• and/or use serializing instructions like `cpuid`
• use pseudo-serializing instruction \texttt{rdtscp} (recent CPUs)
• and/or use serializing instructions like \texttt{cpuid}
• and/or use fences like \texttt{mfence}
Accurate Microarchitecture Timing

- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

Intel Publishes Microcode Security Patches, No Benchmarking Or Comparison Allowed!

UPDATE: Intel has resolved their microcode licensing issue which I complained about in this blog post. The new license text is here.
Access time [CPU cycles]

Number of accesses

Cache Hits
Flush+Reload had beautifully nice timings, right?

Well... steps of 2-4 cycles

only 35-70 steps between hits and misses

On some devices only 1-2 steps!
Flush+Reload had beautifully nice timings, right?

Well... steps of 2-4 cycles

only 35-70 steps between hits and misses

On some devices only 1-2 steps!
Flush+Reload had beautifully nice timings, right? Well... steps of 2-4 cycles only 35-70 steps between hits and misses on some devices only 1-2 steps!
Flush+Reload had beautifully nice timings, right?
Flush+Reload had beautifully nice timings, right?
Well... steps of 2-4 cycles
Flush+Reload had beautifully nice timings, right?

Well... steps of 2-4 cycles
  - only 35-70 steps between hits and misses
Flush+Reload had beautifully nice timings, right?

Well... steps of 2-4 cycles
- only 35-70 steps between hits and misses

On some devices only 1-2 steps!
• We can build our own timer [Lip+16; Sch+17]
• We can build our own timer [Lip+16; Sch+17]
• Start a thread that continuously increments a global variable
• We can build our own timer [Lip+16; Sch+17]
• Start a thread that continuously increments a global variable
• The global variable is our timestamp
ARE YOU REALLY expecting to outperform the hardware counter?
Self-built Timer

CPU cycles one increment takes

```
rdtsc 3
```

```
1 timestamp = rdtsc();
```
CPU cycles one increment takes

<table>
<thead>
<tr>
<th>rdtsc</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>4.7</td>
</tr>
</tbody>
</table>

```c
while (1) {
    timestamp++;
}
```
CPU cycles one increment takes

<table>
<thead>
<tr>
<th></th>
<th>rdtsc</th>
<th>C</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>4.7</td>
<td>4.67</td>
</tr>
</tbody>
</table>

1. `mov &timestamp, %rcx`
2. `incl (%rcx)`
3. `jmp 1b`
<table>
<thead>
<tr>
<th>Method</th>
<th>CPU Cycles One Increment Takes</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdtsc</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4.7</td>
</tr>
<tr>
<td>Assembly</td>
<td>4.67</td>
</tr>
<tr>
<td>Optimized</td>
<td>0.87</td>
</tr>
</tbody>
</table>

```
1 mov &timestamp, %rcx
2 1: inc %rax
3 mov %rax, (%rcx)
4 jmp 1b
```
Instructions are

- fetched and decoded in the front-end
Instructions are

- fetched and decoded in the **front-end**
- dispatched to the **backend**
Instructions are

- fetched and decoded in the front-end
- dispatched to the backend
- processed by individual execution units
Temporal Component

trace over time contains information
single spikes contain information
can’t arbitrarily improve clock
microarchitectural attacks somewhat similar to SPA!
single spike can already reveal a secret
Temporal Component

- trace over time contains information
Temporal Component

- trace over time contains information
- single spikes contain information
Temporal Component

- trace over time contains information
- single spikes contain information
- can’t arbitrarily improve clock
Temporal Component

- trace over time contains information
- single spikes contain information
- can’t arbitrarily improve clock
- microarchitectural attacks somewhat similar to SPA
Temporal Component

- trace over time contains information
- single spikes contain information
- can’t arbitrarily improve clock
- microarchitectural attacks somewhat similar to SPA
  → single spike can already reveal a secret
Case Study: Double Fetches

Caused by accessing the shared memory twice

Double-fetch bugs = exploitable double fetches

Can microarchitectural attacks help here?
Case Study: Double Fetches

Caused by accessing the shared memory twice

Double-fetch bugs = exploitable double fetches

Can microarchitectural attacks help here?
Case Study: Double Fetches

Caused by accessing the shared memory twice

Double-fetch bugs = exploitable double fetches

Can microarchitectural attacks help here?
Case Study: Double Fetches

- “time-of-check-to-time-of-use”
Case Study: Double Fetches

- “time-of-check-to-time-of-use”
- Caused by accessing the shared memory twice
Case Study: Double Fetched

- “time-of-check-to-time-of-use”
- Caused by accessing the shared memory twice
- Double-fetch bugs = exploitable double fetches
Case Study: Double Fetches

- “time-of-check-to-time-of-use”
- Caused by accessing the shared memory twice
- Double-fetch bugs = exploitable double fetches
- Can microarchitectural attacks help here?
string
string

`/path/file\0payload\0`

length

Thread 1

```c
strcpy(string, "\path/file\0payload");
open(string, O_CREAT);
```

Thread 2
A Double Fetch

string

```
/path/file\0payload\0
```

length

Thread 1

```c
strcpy(string, "/path/file\0payload");
open(string, O_CREAT);
// <switch to kernel>
```
A Double Fetch

String

```
/ path / file \0pay\load\0
```

Length

Thread 1

```
strcpy (string, "/path/file\0payload");
open (string, O_CREAT);
// <switch to kernel>
int len = strlen (string);
char* local = malloc (len + 1);
```
A Double Fetch

Thread 1

```c
strcpy(string, "/path/file\0payload");
open(string, O_CREAT);
// <switch to kernel>
int len = strlen(string);
char* local = malloc(len + 1);
```

Thread 2

```c
string[10] = 'X';
```
A Double Fetch

Thread 1

```c
strcpy(string, "/path/file\0payload");
open(string, O_CREAT);
// <switch to kernel>
int len = strlen(string);
char* local = malloc(len + 1);
strcpy(local, string);
// <memory corruption>
```

Thread 2

```
schedule
string[10] = 'X';
```
• Idea: memory access can be observed through the cache
• Idea: memory access can be observed through the cache
• Observe cache activity using a cache attack
(Syscall) Fuzzer

Report general bug

DECAF\textsuperscript{4}

(Syscall) Fuzzer

DECAF

Report general bug

DECAF

(Syscall) Fuzzer

Detect double fetches

Report general bug

---


Daniel Gruss — Graz University of Technology
DECAF\textsuperscript{4}

DECAF\(^4\)

(Syscall) Fuzzer → DECAF → Exploit double fetch

Detect double fetches

Double fetch candidates

Report general bug

---

DECAF\textsuperscript{4}

(Syscall) Fuzzer \rightarrow DECAF \rightarrow Exploit double fetch

Detect double fetches

Double fetch candidates

Report general bug

Report double-fetch bug

DECAF

(Syscall) Fuzzer

DECAF

Exploit double fetch

Detect double fetches

Double fetch candidates

Report general bug

Fix double-fetch bug

Report double-fetch bug

Detection via Flush+Reload

Runtime [cycles]

Access time [cycles]
Detection via Flush+Reload

Runtime [cycles]

Access time [cycles]

Data was accessed
Detection via Flush+Reload

runtime [cycles]

260

Access time [cycles]

240

220

200

0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

Runtime [cycles] •10^6

Daniel Gruss — Graz University of Technology
Detection via Flush+Reload
Detection via Flush+Reload

![Graph showing runtime vs. access time]

- Access time [cycles]
- Runtime [cycles]

- First access
- Second access

Daniel Gruss — Graz University of Technology
Only double-fetch bugs are interesting
• Only **double-fetch bugs** are interesting

→ **exploit** while fuzzing
Only double-fetch bugs are interesting

→ exploit while fuzzing

• Flip value as fast as possible?
• Only double-fetch bugs are interesting
  → exploit while fuzzing
• Flip value as fast as possible?
• Better use a trigger
Only double-fetch bugs are interesting

→ exploit while fuzzing

• Flip value as fast as possible?
• Better use a trigger (just like in physical fault attacks!)
Cache-based Trigger

Access time [cycles] vs. Runtime [cycles]

3 3.5 4 4.5 5 5.5 6 6.5 7

200 220 240 260

\( \cdot 10^5 \)

Daniel Gruss — Graz University of Technology
Cache-based Trigger

![Graph showing access time and runtime with labels for first access, modify value, and second access with modified value.](image-url)
Cache-based Trigger

Access delta [cycles] vs. Probability [%]

- **Flush+Reload**: Pink line
- **Flipping**: Dotted blue line

Diagram shows the probability of various cache behavior with respect to access delta in cycles.
Cache-based Trigger

Daniel Gruss | Graz University of Technology
• Problem: modified value $\rightarrow$ exploit
Getting rid of Double Fetch Bugs

- Problem: modified value → exploit
- Idea: Ensure that both accesses are atomic
Getting rid of Double Fetch Bugs

- Problem: modified value → exploit
- Idea: Ensure that both accesses are atomic
  → Another microarchitectural feature: Intel TSX
• Make a sequence of reads and writes atomic
- Make a sequence of reads and writes atomic
- Operations are wrapped in a transaction
Hardware Transactional Memory

- Make a sequence of reads and writes atomic
- Operations are wrapped in a transaction
- Conflicts → transaction is rolled back
• Make a sequence of reads and writes atomic
• Operations are wrapped in a transaction
• Conflicts → transaction is rolled back
• Implemented via the cache
Thread 0

Verb Begin

Verb End

else path of xbegin

Cache

Thread 1
Transactional Memory

Thread 0
- xbegin
- mov
- xend
- else path of xbegin

Thread 1
- mov

Cache
- read set
- data
- read

Thread 0 reads data from the cache, and Thread 1 also reads data from the cache.
Transactional Memory

Thread 0

xbegin
mov
mov

xend
else path of xbegin

read
mov
mov

read set

Cache

data

read
write
mov
mov

Thread 1

Daniel Gruss — Graz University of Technology
Transactional Memory

Thread 0

- xbegin
- mov
- mov
- mov
- xend
- else path of xbegin

Thread 1

- read
- read
- mov
- mov
- read
- read
- mov
- mov
- write
- write
- mov

Cache

- data
- data
- data
- read set

Daniel Gruss — Graz University of Technology
Transactional Memory

Thread 0

xbegin
mov
mov
mov
xend
else path of xbegin

Thread 1

read
mov
mov
read
mov
write
mov
mov
mov

Cache

read
read
write
transactional abort
read set
Transactional Memory

Thread 0

- `xbegin`
- `mov`
- `read`
- `mov`
- `read`
- `mov`
- `xend`
- `else path of xbegin`

First access

Thread 1

- `read`
- `write`
- `mov`
- `write`
- `mov`

Cache

- `data`
- `data`
- `data`
- `read set`

Transactional abort

Daniel Gruss — Graz University of Technology
Transactional Memory

Thread 0

- `xbegin`
- `mov`
- `read`
- `mov`
- `read`
- `mov`
- `read`
- `mov`
- `read`
- `mov`
- `xend`
- else path of `xbegin`
- First access
- transactional abort

Thread 1

- `mov`
- `read`
- `write`
- `mov`
- `write`
- Modification

Cache

- data
- data
- data
- data
- read set

Transaction link:

- Transactional abort
- Read set

www.tugraz.at

Daniel Gruss — Graz University of Technology
Transactional Memory

Thread 0
- xbegin
- mov
- First access
- Second access
- xend
- Else path of xbegin

Thread 1
- mov
- Modification

Cache
- data
- read
- Second access
- transactional abort

Transaction: First access, then Second access. After Second access, transaction aborts.
Transactional Memory

Thread 0
- xbegin
- mov
- First access
- Second access
- xend
- Exploit detected
- Of xbegin

Thread 1
- read
- write
- mov
- Modification

Cache
- data
- read
- write
- transactional abort
- read set

Transaction log:
- First access
- Second access
- Exploit detected

Daniel Gruss — Graz University of Technology
Microarchitectural Defenses

device under test = measurement device

software defenses are possible
e.g., make sure attacker can't compute in parallel to victim

how would that work in the physical world?
Microarchitectural Defenses

Device under test = measurement device

Software defenses are possible, e.g., make sure attacker can’t compute in parallel to victim.

How would that work in the physical world?
Microarchitectural Defenses

Device under test = measurement device

Software defenses are possible e.g., make sure attacker can’t compute in parallel to victim

How would that work in the physical world?
device under test = measurement device
• device under test = measurement device
→ software defenses are possible
• device under test = measurement device
→ software defenses are possible
• e.g., make sure attacker can’t compute in parallel to victim
• device under test = measurement device
→ software defenses are possible
• e.g., make sure attacker can’t compute in parallel to victim
• how would that work in the physical world?
Spatial Component

Physical:
- different offsets on the chip

Microarchitectural:
- different microarchitectural elements

More significant:
- huge virtual address space
- 2^48 different virtual memory locations
- the location is often (part of) the secret
Spatial Component

physical: different offsets on the chip

microarchitectural: different microarchitectural elements

more significant: huge virtual address space

2^48 different virtual memory locations

the location is often (part of) the secret
Spatial Component

physical: different offsets on the chip

microarchitectural: different microarchitectural elements

more significant: huge virtual address space

the location is often (part of) the secret
• physical: different offsets on the chip
Spatial Component

- physical: different offsets on the chip
- microarchitectural:
Spatial Component

- physical: different offsets on the chip
- microarchitectural:
  - different microarchitectural elements
Spatial Component

- physical: different offsets on the chip
- microarchitectural:
  - different microarchitectural elements
  - more significant: huge virtual adress space
Spatial Component

- physical: different offsets on the chip
- microarchitectural:
  - different microarchitectural elements
  - more significant: huge virtual address space
  - $2^{48}$ different virtual memory locations
Spatial Component

- physical: different offsets on the chip
- microarchitectural:
  - different microarchitectural elements
  - more significant: huge virtual address space
  - $2^{48}$ different virtual memory locations
  - the location is often (part of) the secret
Cache Template Attack Demo
Side-Channel Attacks and Fault Attacks?
Attack Categories

Physical

- Side-channel attacks

What about cold boot attacks? [Hal+09]

What about Meltdown/Spectre? [Lip+18; Koc+19]
Physical

- Side-channel attacks
- Fault attacks
Physical

- Side-channel attacks
- Fault attacks
- What about cold boot attacks? [Hal+09]
Attack Categories

Physical

• Side-channel attacks
• Fault attacks
• What about cold boot attacks? [Hal+09]

Microarchitectural
Attack Categories

Physical
- Side-channel attacks
- Fault attacks
- What about cold boot attacks? [Hal+09]

Microarchitectural
- Side-channel attacks
Attack Categories

Physical
- Side-channel attacks
- Fault attacks
- What about cold boot attacks? [Hal+09]

Microarchitectural
- Side-channel attacks
- Fault attacks
Attack Categories

Physical
- Side-channel attacks
- Fault attacks
- What about cold boot attacks? [Hal+09]

Microarchitectural
- Side-channel attacks
- Fault attacks
- What about Meltdown/Spectre? [Lip+18; Koc+19]
Out-of-order state does not become\textit{architecturally visible but} . . .
Out-of-order state does not become **architecturally visible** but . . .
*(volatile char*) 0;
array[84 * 4096] = 0;
• Flush+Reload over all pages of the array
• Flush+Reload over all pages of the array

• “Unreachable” code line was actually executed
Building Meltdown

- Flush+Reload over all pages of the array
- “Unreachable” code line was **actually executed**
- Exception was only thrown **afterwards**
• Out-of-order instructions leave microarchitectural traces
Out-of-order instructions leave microarchitectural traces
- We can see them for example through the cache
• Out-of-order instructions leave microarchitectural traces
  • We can see them for example through the cache
• Give such instructions a name: transient instructions
Out-of-order instructions leave microarchitectural traces

- We can see them for example through the cache
- Give such instructions a name: transient instructions
- We can indirectly observe the execution of transient instructions
• Add another layer of indirection to test

```c
char data = *(char*) 0xffffffff81a000e0;
array[data * 4096] = 0;
```
• Add another layer of indirection to test

```c
char data = *(char*) 0xffffffff81a000e0;
array[data * 4096] = 0;
```

• Then check whether any part of array is cached
- Flush+Reload over all pages of the array

- Index of cache hit reveals data
Building Meltdown

- Flush+Reload over all pages of the array

- Index of cache hit reveals data

- Permission check is in some cases not fast enough
I SHIT YOU NOT

THERE WAS KERNEL MEMORY ALL OVER THE TERMINAL
used with authorization from Sili
cion Graphics, Inc. However, the authors make no claim that Mesa is in any way a compatible replacement for OpenGL or associated with Silicon Graphics, Inc. This version of Mesa provides GLX and DRI capabilities: it is capable of both direct and indirect rendering. For direct rendering, it can use DRI modules from the libg
mschwarz@lab06:~/Documents$
• Basic Meltdown code leads to a crash (segfault)
• Basic Meltdown code leads to a crash (segfault)
• How to prevent the crash?
- Basic Meltdown code leads to a crash (segfault)
- How to prevent the crash?
Intel TSX to suppress exceptions instead of signal handler

```c
if (xbegin() == XBEGIN_STARTED) {
    char secret = *(char*) 0xffffffff81a000e0;
    array[secret * 4096] = 0;
    xend();
}

for (size_t i = 0; i < 256; i++) {
    if (flush_and_reload(array + i * 4096) == CACHE_HIT) {
        printf("%c\n", i);
    }
}
```
Speculative execution to prevent exceptions

```c
int speculate = rand() % 2;
size_t address = (0xffffffff81a000e0 * speculate) +
    (*((size_t*)&zero * (1 - speculate));
if (!speculate) {
    char secret = *(char*) address;
    array[secret * 4096] = 0;
}
for (size_t i = 0; i < 256; i++) {
    if (flush_and_reload(array + i * 4096) == CACHE_HIT) {
        printf("%c\n", i);
    }
}
```
Boot from ROM...
early console in extract_kernel
input_data: 0x00000000001e0a276
input_len: 0x00000000003d48f8
output: 0x0000000000100000
output_len: 0x00000000001bc258
kernel_total_size: 0x000000000dec000
booted via startup_32()
Physical KASLR using RDTSC...
Virtual KASLR using RDTSC...

Decompressing Linux... Parsing ELF... Performing relocations... done.
Booting the kernel.

L1 Terminal Fault

Run `reader <PFN> [cache miss threshold]` to leak hypervisor data from the L1
/
#
index = 0;

char* data = "textKEY";

if (index < 4)
    Prediction

then
LUT[data[index] * 4096]
else
0

Daniel Gruss — Graz University of Technology
index = 0;

char* data = "textKEY";

if (index < 4)
    LUT[data[index] * 4096]
else
    0
index = 0;

char* data = "textKEY";

if (index < 4)

    then

        LUT[data[index] * 4096]

    else

        Prediction

Speculate

0
index = 0;

char* data = "textKEY";

if (index < 4) 

LUT[data[index] * 4096]

else

0

Prediction

Execute

then

Daniel Gruss — Graz University of Technology
index = 1;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction
index = 1;

define data as a character pointer and initialize it with "textKEY"

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction

Spectre v4: Ignore sanitizing write access and use unsanitized old value instead
index = 1;

char* data = "textKEY";

if (index < 4)
    Speculate
    then
    Prediction
    LUT[data[index] * 4096]
else
    0
index = 1;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction

Spectre v1

Daniel Gruss — Graz University of Technology
index = 2;

char* data = "textKEY";

if (index < 4)

Prediction

LUT[data[index] * 4096] 0
index = 2;

char* data = "textKEY";

if (index < 4)
    Prediction
else
    LUT[data[index] * 4096] 0
index = 2;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

0
index = 2;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction
index = 3;

char* data = "textKEY";

if (index < 4)
    Prediction
    LUT[data[index] * 4096]
else
    0
index = 3;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
Prediction

0
index = 3;

char* data = "textKEY";

if (index < 4)
    Speculate
    then
    LUT[data[index] * 4096]
    else
    Prediction
          0

Spectre v1: Ignore sanitizing write access and use unsanitized old value instead
Daniel Gruss — Graz University of Technology
index = 3;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction
index = 4;

char* data = "textKEY";

if (index < 4)
then

LUT[data[index] * 4096]

else

0
index = 4;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0
index = 4;

char* data = "textKEY";

if (index < 4)

Speculate

then

LUT[data[index] * 4096]

else

Prediction

0

Daniel Gruss — Graz University of Technology
index = 4;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]
index = 5;

char* data = "textKEY";

if (index < 4)
then
Prediction

LUT[data[index] * 4096]

else

0
index = 5;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
0

Prediction
index = 5;

char* data = "textKEY";

if (index < 4)
    Speculate
    then
    LUT[data[index] * 4096]

else
    Prediction
    0
index = 5;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

Prediction

Execute

0
index = 6;

char* data = "textKEY";

if (index < 4) {
    LUT[data[index] * 4096];
    Prediction
} else {
    0
}
index = 6;

char* data = "textKEY";

if (index < 4)
    then
    Prediction
    LUT[data[index] * 4096]
    else
    0
index = 6;

char* data = "textKEY";

if (index < 4)

Speculate
then
LUT[data[index] * 4096]

Prediction
else
0
index = 6;

char* data = "textKEY";

if (index < 4)
    LUT[data[index] * 4096]
index = 6;

char* data = "textKEY";

if (index < 4)
    then
        LUT[data[index] * 4096]
    else
        0

Spectre v4: Ignore sanitizing write access and use unsanitized old value instead
Animal* a = bird;

a->move()

fly()

swim()  swim()

LUT[data[a->m] * 4096]

0
Animal* a = bird;

LUT[data[a->m] * 4096]

Prediction

fly() -> swim() -> Speculate

swim()
Animal* a = bird;
a->move()

fly()

swim()

swim()

LUT[data[a->m] * 4096]

0
Animal* a = bird;

a->move();

LUT[data[a->m] * 4096]

fly()

swim()

swim()

0

Prediction
```cpp
Animal* a = bird;
```

```
LUT[data[a->m] * 4096] 0
```

- `a->move()`
  - `fly()`
    - Prediction
  - `swim()`
`Animal* a = bird;`

```
Speculate
fly()
LUT[data[a->m] * 4096]
```
Spectre v2

Animal* a = bird;

a->move()

fly()  

| Prediction | LUT[data[a->m] * 4096] | 0 |

swim()
Animal* a = fish;

a->move()

fly()

fly()  swim()

Prediction

LUT[data[a->m] * 4096]  0
Animal* a = fish;

a->move();

Speculate

fly()

LUT[data[a->m] * 4096]

Prediction

fly()

swim()

0
Animal* a = fish;

a->move()

fly()

fly()

swim()

LUT[data[a->m] * 4096] 0
Animall* a = fish;

LUT[data[a->m] * 4096]

a->move()

Predict

fly()  swim()

Execute

0
Animal* a = fish;

LUT[data[a->m] * 4096]

0
Animal* a = fish;

a->move()

fly()

swim()

LUT[data[a->m] * 4096]

0

Spectre v2: mistrain BTB → mispredict indirect jump/call
Animal* a = fish;

a->move()

fly() → swim()

LUT[data[a->m] * 4096] = 0

Spectre v2: mistrain BTB → mispredict indirect jump/call

Spectre v5: mistrain RSB → mispredict return
• v1.1: Speculatively write to memory locations

---

• v1.1: Speculatively write to memory locations
  → Many more gadgets than previously anticipated

---

• v1.1: Speculatively write to memory locations
  → Many more gadgets than previously anticipated
• v1.2: Ignore writable bit

---

v1.1: Speculatively write to memory locations
   → Many more gadgets than previously anticipated
v1.2: Ignore writable bit
   → not really Spectre but a Meltdown variant

---

Meltdown vs. Spectre

Meltdown attacks

Spectre attacks
Meltdown vs. Spectre

Meltdown attacks
• Meltdown, LazyFP (v3.1), Foreshadow, Foreshadow-NG, ...

Spectre attacks
• v1, v1.1, v2, v4, SpectreRSB (v5)
Meltdown vs. Spectre

Meltdown attacks

- Meltdown, LazyFP (v3.1),
  Foreshadow, Foreshadow-NG, ...
- Out-of-Order Execution

Spectre attacks

- v1, v1.1, v2, v4, SpectreRSB (v5)
- Speculative Execution ⊆ Out-of-Order Execution
Meltdown attacks

- Meltdown, LazyFP (v3.1), Foreshadow, Foreshadow-NG, ...
- Out-of-Order Execution
- no prediction required

Spectre attacks

- v1, v1.1, v2, v4, SpectreRSB (v5)
- **Speculative Execution** ⊂ Out-of-Order Execution
- fundamentally rely on prediction
Meltdown vs. Spectre

Meltdown attacks

- Meltdown, LazyFP (v3.1), Foreshadow, Foreshadow-NG, ...
- Out-of-Order Execution
- no prediction required

→ melt down isolation by ignoring access permissions (e.g., page table bits)

Spectre attacks

- v1, v1.1, v2, v4, SpectreRSB (v5)
- Speculative Execution ⊂ Out-of-Order Execution
- fundamentally rely on prediction
- difficult to mitigate because it does not violate access permissions
Meltdown vs. Spectre

Meltdown attacks

- Meltdown, LazyFP (v3.1), Foreshadow, Foreshadow-NG, ...
- Out-of-Order Execution
- **no prediction required**
- melt down isolation by ignoring access permissions (e.g., page table bits)
- practical mitigation in software (e.g., KAISER)

Spectre attacks

- v1, v1.1, v2, v4, SpectreRSB (v5)
- **Speculative Execution ⊆ Out-of-Order Execution**
- fundamentally rely on prediction
- difficult to mitigate because it **does not violate access permissions**
Conclusion

large-scale attacks due to different threat models could be leveraged to gain a more complete picture.

space for promising mitigations (due to inherent restrictions for the attacker).
large-scale attacks due to different threat model overlap could be leveraged to gain a more complete picture of the attack space for promising mitigations (due to inherent restrictions for the attacker).
large-scale attacks due to different threat model overlap could be leveraged to gain more complete picture space for promising mitigations (due to inherent restrictions for the attacker)
large-scale attacks due to different threat model
• large-scale attacks due to different threat model
• overlap could be leveraged to gain more complete picture
• large-scale attacks due to different threat model
• overlap could be leveraged to gain more complete picture
• space for promising mitigations (due to inherent restrictions for the attacker)
I forgot the “Who am I” slide!!
I forgot the “Who am I” slide!!
I forgot the “Who am I” slide!!1
I forgot the “Who am I” slide!!1

I’m building up a group @ Graz University of Technology
I forgot the “Who am I” slide!!

I’m building up a group @ Graz University of Technology
→ looking for PhD students!

Daniel Gruss — Graz University of Technology
(Why) Are Microarchitectural Attacks Really Different than Physical Side-Channel Attacks?

Daniel Gruss
September 10, 2018
Graz University of Technology


