Micro-architectural attacks: from CPU to browser

Clémentine Maurice, CNRS, CRIStAL
28 June 2023—Journées nationales du GDR Sécurité Informatique
Attacks on micro-architecture

- *hardware* usually modeled as an abstract layer behaving correctly
Attacks on micro-architecture

- **hardware** usually modeled as an abstract layer behaving correctly, but possible attacks
Attacks on micro-architecture

- **hardware** usually modeled as an abstract layer behaving correctly, but possible attacks
  - faults: bypassing software protections by causing **hardware errors**
  - side channels: observing **side effects** of hardware on computations
Attacks on micro-architecture

- **hardware** usually modeled as an abstract layer behaving correctly, but possible attacks
  - faults: bypassing software protections by causing **hardware errors**
  - side channels: observing **side effects** of hardware on computations

**Identification**

![Access time vs. Number of accesses]

- Cache hits
- Cache misses

**Table II** shows an example where the topmost 5 bits are already recovered (underlined). The sixth key bit is recovered as '1'.

**Table III** gives an example where the topmost 5 bits are already recovered (underlined). The sixth key bit is recovered as '1'.

**Results**

- **Prime+Probe** (5 s)
- **Pre-Processing** (110 s)
- **Key Recovery** (20 s)

**Figure 7**. On top is one trace's raw measurement over 4 000 000 cycles. The peaks in the resampled trace on the bottom clearly indicate '1's.

**Figure 8**. A high-level overview of the average times for each step of the attack.

**Figure 9**. The 9 cache sets that are used by a 4096 b buffer. The attacked buffer spans 9 cache sets, out of which 6 show low bit-error rate, as shown in Figure 9. For the evaluation, we attack a 4096-bit RSA key as this makes it easy to retrieve secret keys, keystroke identification, and bypassing OS security (ASLR).
Attacks on micro-architecture

- **hardware** usually modeled as an abstract layer behaving correctly, but possible attacks
  - faults: bypassing software protections by causing **hardware errors**
  - side channels: observing **side effects** of hardware on computations

**identification**

**attack**

- retrieving secret keys, keystroke timings
- bypassing OS security (ASLR)
Attacker model

Hardware-based attacks
a.k.a physical attacks

- Physical access to hardware
  → embedded devices

Software-based attacks
a.k.a micro-architectural attacks

- Co-located or remote attacker
  → complex systems
From small optimizations to side-channel attacks...

- new micro-architectures yearly

- Haswell (2013)
- Broadwell (2014)
- Skylake (2015)
- Kaby Lake (2016)
- Coffee Lake (2017)
- Whiskey Lake (2018)
- Comet Lake/Ice Lake (2019)
- Tiger Lake (2020)
- Alder Lake/Rocket Lake (2021)
- Sapphire Rapids (2022)
From small optimizations to side-channel attacks...

- new micro-architectures yearly
- performance improvement $\approx 5\%$

2013 - Haswell
2014 - Broadwell
2015 - Skylake
2016 - Kaby Lake
2017 - Coffee Lake
2018 - Whiskey Lake
2019 - Comet Lake/Ice Lake
2020 - Tiger Lake
2021 - Alder Lake/Rocket Lake
2022 - Sapphire Rapids
From small optimizations to side-channel attacks...

- new micro-architectures yearly
- performance improvement $\approx 5\%$
- very small optimizations: caches, branch prediction...
From small optimizations to side-channel attacks...

- new micro-architectures yearly
- performance improvement $\approx 5\%$
- very small optimizations: caches, branch prediction...
- micro-architectural side channels come from these optimizations
From small optimizations to side-channel attacks...

- new micro-architectures yearly
- performance improvement $\approx 5\%$
- very small optimizations: caches, branch prediction...
- micro-architectural side channels come from these optimizations
- attacker infers information from a (vulnerable) victim process via hardware usage

- 2013: Haswell
- 2014: Broadwell
- 2015: Skylake
- 2016: Kaby Lake
- 2017: Coffee Lake
- 2018: Whiskey Lake
- 2019: Comet Lake/Ice Lake
- 2020: Tiger Lake
- 2021: Alder Lake/Rocket Lake
- 2022: Sapphire Rapids
Micro-architectural side-channel attacks: Two faces of the same coin

Implementation

Algorithm 1: Square-and-multiply exponentiation

Input: base $b$, exponent $e$, modulus $n$
Output: $b^e \mod n$

$X \leftarrow 1$

for $i \leftarrow \text{bitlen}(e)$ downto 0 do

$X \leftarrow \text{multiply}(X, X)$

if $e_i = 1$ then

$X \leftarrow \text{multiply}(X, b)$

end

end

return $X$

Hardware

&
Research questions

RQ1. Which **hardware components** are vulnerable...

    ... and how to use them to leak data?

RQ2. Which **software implementation** is vulnerable...

    ... and what are the different attack deliveries?
Outline

Part 1  Reverse-engineering micro-architectural components (RQ1)
Outline

Part 1 Reverse-engineering micro-architectural components (RQ1)

Part 2 Porting micro-architectural attacks to the Web (RQ2)
Part 1   Reverse-engineering micro-architectural components (RQ1)

Part 2   Porting micro-architectural attacks to the Web (RQ2)

Part 3   Conclusion
Reverse-engineering micro-architectural components
RQ1. Which hardware component leaks information?

State of the art (more or less)

1. spend too much time reading Intel manuals
2. find weird behavior in corner cases
3. exploit it using a known vulnerability
4. publish
5. goto step 1
RQ1. Which hardware component leaks information?

State of the art in 2015:
only the cache and the branch predictor were explored
Motivation

- performance optimizations are mostly **undocumented**
- side channels come from these optimizations
  → understanding them is crucial to **characterize the attack surface**: build new or improve known side-channel primitives
General approach

Side-channel analysis

- Component model
- Execution
- Secret

software w/
input-dependent
data- or control-flow

Reverse-engineering is the opposite operation of side-channel analysis.
General approach

Side-channel analysis

- Component model
- Execution
- Secret
- Software w/ input-dependent data- or control-flow

Reverse engineering

- Known input
- Component model
- Execution
- Software w/ input-dependent data- or control-flow

Reverse engineering is the opposite operation of side-channel analysis.
Reverse-engineering is the opposite operation of side-channel analysis.
RQ1. Which hardware component leaks information?

State of the art in 2015:
only the cache and the branch predictor were explored
RQ1. Which hardware component leaks information?

State of the art today: each component shared by two processes is a potential micro-architectural side-channel vector.
Porting micro-architectural attacks to the Web
RQ2a. Which software implementation is vulnerable?

State of the art (more or less)

1. spend too much time reading OpenSSL code
2. find vulnerability
3. exploit it manually using known side channel → e.g. CPU cache
4. publish
5. goto step 1

RQ2b. How to deliver the attack?

<table>
<thead>
<tr>
<th>mov ecx, start_of_buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub length_of_buffer, 0x2000</td>
</tr>
<tr>
<td>rdtsc</td>
</tr>
<tr>
<td>mov esi, eax</td>
</tr>
<tr>
<td>xor edi, edi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefetch2 [ecx + edi + 0x2800]</td>
</tr>
<tr>
<td>add cx, [ecx + edi + 0x0000]</td>
</tr>
<tr>
<td>imul ecx, 1</td>
</tr>
<tr>
<td>add cx, [ecx + edi + 0x0800]</td>
</tr>
<tr>
<td>imul ecx, 1</td>
</tr>
<tr>
<td>add cx, [ecx + edi + 0x1000]</td>
</tr>
<tr>
<td>imul ecx, 1</td>
</tr>
<tr>
<td>add cx, [ecx + edi + 0x1800]</td>
</tr>
<tr>
<td>imul ecx, 1</td>
</tr>
</tbody>
</table>

| rdtsc                     |
| sub eax, esi              |
| mov [ecx + edi], ax        |
| add esi, eax              |
| imul ecx, 1               |

| add edi, 0x40             |
| test edi, 0x7C0           |
| jnz loop                  |

| sub edi, 0x7EE            |
| test edi, 0x3E            |
| jnz loop                  |

| add edi, 0x7C0            |
| sub length_of_buffer, 0x800 |
| jge loop                  |

State of the art in 2015

- native code, cross process and cross-VM
- lots of (x86) assembly required
How to obtain such low-level control from a high-level abstraction layer?
• side channels are only doing **benign operations**
Side-channel attacks in JavaScript?

- side channels are only doing "benign operations"
- all side-channel attacks: "measuring time"
Side-channel attacks in JavaScript?

- side channels are only doing **benign operations**
  - all side-channel attacks: **measuring time**
  - cache attacks: accessing their own memory
  - port contention attacks: executing specific instructions
Side-channel attacks in JavaScript?

- side channels are only doing **benign operations**
  - all side-channel attacks: **measuring time**
  - cache attacks: accessing their own memory
  - port contention attacks: executing specific instructions
Measuring time
• measure small timing differences: need a **high-resolution timer**
High-resolution timers?

- measure small timing differences: need a high-resolution timer
- native: `rdtsc`, timestamp in CPU cycles
High-resolution timers?

- measure small timing differences: need a high-resolution timer
- native: rdtsc, timestamp in CPU cycles
- JavaScript: performance.now() has the highest resolution
High-resolution timers?

- measure small timing differences: need a **high-resolution timer**
- native: **rdtsc**, timestamp in CPU cycles
- JavaScript: **performance.now()** has the highest resolution

**performance.now()**

[...] represent times as floating-point numbers with up to microsecond precision.

— Mozilla Developer Network
Evolution of timers until today: resolution and countermeasures

T. Rokicki, C. Maurice, and P. Laperdrix. “Sok: In search of lost time: A review of javascript timers in browsers”. In: EuroS&P. 2021
Evolution of timers until today: resolution and countermeasures

- 2015
  - $\leq 43$ ns
  - $\leq 40$ ns

- 2016
  - 44 µs

- 2018
  - 5 µs

- 2019
  - 5 µs

- 2020
  - 1 ms + jitter

- 2021
  - 20 µs

- 2022
  - 2 ms

---

T. Rokicki, C. Maurice, and P. Laperdrix. “Sok: In search of lost time: A review of javascript timers in browsers”. In: EuroS&P. 2021
Evolution of timers until today: resolution and countermeasures

- 2015: ≤ 43 ns
- 2016: 44 ns
- 2018: 64 µs + jitter
- 2019: 72 µs + jitter
- 2020: & COOP/COEP: 5 µs + jitter, SharedArrayBuffer enabled
- 2021: 92 µs + jitter
- 2022: & COOP/COEP: 20 µs

T. Rokicki, C. Maurice, and P. Laperdrix. “Sok: In search of lost time: A review of javascript timers in browsers”. In: EuroS&P. 2021
Evolution of timers until today: resolution and countermeasures

*The spy in the sandbox*

- 2015: \(\leq 43\) ns
- 2016: \(\leq 40\) ns

*Spectre*

- 2016: 5 µs + jitter
- 2018: 100 µs + jitter

& COOP/COEP:

- 2019: 5 µs + jitter
- 2020: 20 µs
- 2021: 20 µs
- 2022: 5 µs + jitter, `SharedArrayBuffer` enabled

Before September 2015

- `performance.now()` had a *nanosecond* resolution

---


Before September 2015

- `performance.now()` had a **nanosecond** resolution
- 2015: Oren et al. demonstrated cache side-channel attacks in JavaScript

---


Before September 2015

- `performance.now()` had a nanosecond resolution
- 2015: Oren et al. demonstrated cache side-channel attacks in JavaScript
- countermeasure in Firefox 41 & Chrome 44: clamping to 5 µs

Is clamping an efficient countermeasure?


- microsecond resolution is not enough for attacks
Is clamping an efficient countermeasure?

- Microsecond resolution is not enough for attacks
- Two approaches

M. Schwarz et al. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: FC. 2017
Is clamping an efficient countermeasure?


- microsecond resolution is **not enough** for attacks
- two approaches
  1. recover a higher resolution from the available timer
Is clamping an efficient countermeasure?

M. Schwarz et al. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: FC. 2017

- microsecond resolution is not enough for attacks
- two approaches
  1. recover a higher resolution from the available timer
  2. build our own high-resolution timer
Recovering resolution: Clock interpolation

- measure how often we can increment a variable between two timer ticks

Firefox/Chrome: 500 ns, Tor: 15 µs
Recovering resolution: Clock interpolation

- **measure** how often we can **increment** a variable between two timer ticks

Firefox/Chrome: 500 ns, Tor: 15 µs
• measure how often we can increment a variable between two timer ticks
- measure how often we can increment a variable between two timer ticks

Firefox/Chrome: 500 ns, Tor: 15 µs
Recovering resolution: Clock interpolation

- measure how often we can increment a variable between two timer ticks

![Diagram showing clock ticks and variable increments]

- to measure with high resolution
Recovering resolution: Clock interpolation

- **measure** how often we can **increment** a variable between two timer ticks
- to measure with high resolution
  - start measurement at **clock edge**
Recovering resolution: Clock interpolation

• measure how often we can increment a variable between two timer ticks

![Clock Interpolation Diagram]

• to measure with high resolution
  • start measurement at clock edge
  • increment a variable until next clock edge

Firefox/Chrome: 500 ns, Tor: 15 µs
Recovering resolution: Clock interpolation

- measure how often we can increment a variable between two timer ticks

- to measure with high resolution
  - start measurement at clock edge
  - increment a variable until next clock edge

- Firefox/Chrome: 500 ns, Tor: 15 µs
Building a timer: Web worker

- feature to share data: `SharedArrayBuffer`
Building a timer: Web worker

- feature to share data: `SharedArrayBuffer`
- web worker can *simultaneously* read/write data
Building a timer: Web worker

- feature to share data: `SharedArrayBuffer`
- web worker can *simultaneously* read/write data
- no message passing overhead
Building a timer: Web worker

- feature to share data: `SharedArrayBuffer`
- web worker can *simultaneously* read/write data
- no message passing overhead
- one dedicated worker for incrementing the shared variable
Building a timer: Web worker

- feature to share data: `SharedArrayBuffer`
- web worker can *simultaneously* read/write data
- no message passing overhead
- one dedicated worker for incrementing the shared variable
- Firefox/Fuzzyfox: 2 ns, Chrome: 15 ns
T. Rokicki, C. Maurice, and P. Laperdrix. “Sok: In search of lost time: A review of javascript timers in browsers”. In: EuroS&P. 2021

- adding jitter → makes clock interpolation inefficient
· adding jitter → makes clock interpolation inefficient
→ has no impact on SharedArrayBuffers!

T. Rokicki, C. Maurice, and P. Laperdrix. “Sok: In search of lost time: A review of javascript timers in browsers”. In: EuroS&P. 2021
Jitter?

T. Rokicki, C. Maurice, and P. Laperdrix. “Sok: In search of lost time: A review of javascript timers in browsers”. In: EuroS&P. 2021

- adding jitter → makes clock interpolation inefficient
  → has no impact on SharedArrayBuffers!
- browsers are adopting better isolation between websites (e.g., Site Isolation) to counter transient execution attacks
- back to higher timer resolution for usability → side-channel attacks are possible again!
Port contention attacks
Background: Hyper-threading

Simultaneous computation technology of Intel.

- physical cores are shared between logical cores
- abstraction at the OS level
Background: Hyper-threading

Simultaneous computation technology of Intel.

- physical cores are shared between logical cores
- abstraction at the OS level

→ hardware resources are shared between logical cores
- instructions are decomposed in uops to optimize Out-of-Order execution
- uops are dispatched to specialized execution units through CPU ports
- deterministic decomposition of instructions into uops
Port contention

No contention

All attacker instructions are executed in a row
→ fast execution time

Port contention

No contention

All attacker instructions are executed in a row → fast execution time

Contention

Victim instructions delay the attacker instructions → slow execution time

Port contention side-channel attack

Victim

secret == 0

POPCNT %r8,%r8
POPCNT %r8,%r8
...
POPCNT %r8,%r8
POPCNT %r8,%r8

Contention on Port 1

secret == 1

VPBROADCASTD %xmm0, %ymm0
VPBROADCASTD %xmm0, %ymm0
...
VPBROADCASTD %xmm0, %ymm0
VPBROADCASTD %xmm0, %ymm0

Contention on Port 5

Monitors port usage
Port contention side-channel attack

Victim

secret == 0

secret == 1

Contention on Port 1

Secret is 0!

POPCNT %r8,%r8
POPCNT %r8,%r8
...
POPCNT %r8,%r8
POPCNT %r8,%r8

VPBROADCASTD %xmm0, %ymm0
VPBROADCASTD %xmm0, %ymm0
...
VPBROADCASTD %xmm0, %ymm0
VPBROADCASTD %xmm0, %ymm0
Port contention side-channel attack

Victim

secret == 0

POPCNT %r8,%r8
POPCNT %r8,%r8
...
POPCNT %r8,%r8
POPCNT %r8,%r8

secret == 1

VPBROADCASTD %xmm0, %ymm0
VPBROADCASTD %xmm0, %ymm0
...
VPBROADCASTD %xmm0, %ymm0
VPBROADCASTD %xmm0, %ymm0

Contention on Port 5

Secret is 1!
Port contention attacks: Challenges with JavaScript

T. Rokicki et al. “Port Contention Goes Portable: Port Contention Side Channels in Web Browsers”. In: ASIACCS. 2022

1. No high-resolution timers
2. No control on cores
3. No access to specific instructions
#1. No high-resolution timers

We just solved this problem :)

---

#2. No control on cores

- JavaScript does not have control on cores
- scheduler tries to balance the workload of physical cores

→ exploit JavaScript multi-threading and work with the scheduler
#3. No access to specific instructions

- sandboxed
- JIT compilation
#3. No access to specific instructions

- sandboxed
- JIT compilation

- sandboxed
- compiled from another language
- smaller, more atomic instructions
Proof-of-concept native-to-web

**Native**: C code runs `TZCNT` x86 instructions (P1 uop) on all physical cores

**Web**: WebAssembly repeatedly calls `i64.ctz` and times the execution
**Port contention side-channel in WebAssembly**

![Diagram](image)

- spatial resolution: 1024 native instructions
- similar to other web-based cache attacks
- timers are the main bottleneck

**Figure 1:** Secret key: 1101001.
Port contention covert channel: native-to-web

- **Native**: C/x86 sender
- **Web**: WebAssembly receiver

**Evaluation:**
- 200 bit/s of effective data (best bandwidth for a web-based covert channel!)
- `stress -m 2`: 170 bit/s
- `stress -m 3`: 25 bit/s
More port contention covert channels

**VM-to-host**

- User applications
- OS
- Hardware: CPU Ports

**Cross-browser**

- User applications
- OS
- Hardware: CPU Ports

80 bit/s bandwidth vs. 200 bit/s bandwidth (physical layer), across different browsers!
RQ2b. How to deliver the attack?

State of the art in 2015

- native code, cross process and cross-VM
- lots of (x86) assembly required

```
RQ2b. How to deliver the attack?

State of the art in 2015
- native code, cross process and cross-VM
- lots of (x86) assembly required

```

```
mov ecx, start_of_buffer
sub length_of_buffer, 0x2000
rdtsc
mov esi, eax
xor edi, edi

loop:
prefetcht2 [ecx + edi + 0x2800]
add cx, [ecx + edi + 0x0000]
imul ecx, 1
add cx, [ecx + edi + 0x0800]
imul ecx, 1
add cx, [ecx + edi + 0x1000]
imul ecx, 1
add cx, [ecx + edi + 0x1800]
imul ecx, 1

rdtsc:
sub eax, esi
mov [ecx + edi], ax
add esi, eax
imul ecx, 1

add edi, 0x40
test edi, 0x7C0
jnz loop

sub edi, 0x7FE
test edi, 0x3E
jnz loop

add edi, 0x7C0
sub length_of_buffer, 0x800
jge loop
```

*Figure 1. Example code for a Spy process monitoring the L1 cache.*
RQ2b. How to deliver the attack?

State of the art today: many Web-based micro-architectural attacks
Conclusion
Conclusions

- first paper by Kocher in 1996: 25 years of research in this area
Conclusions

- first paper by Kocher in 1996: 25 years of research in this area
- domain still in expansion: increasing number of papers published since 2015
Conclusions

• first paper by Kocher in 1996: 25 years of research in this area
• domain still in expansion: increasing number of papers published since 2015
• micro-architectural attacks require a:
  • low-level understanding of the components → reverse-engineering
  • low-level control of the components usually achieved with native code → still possible to deliver these attacks from web browsers
→ work across all abstraction layers
Thank you!

Contact

✉️ clementine.maurice@cnrs.fr
🐦 @BloodyTangerine
Micro-architectural attacks: from CPU to browser

Clémente Maurice, CNRS, CRISiAL
28 June 2023—Journées nationales du GDR Sécurité Informatique