Micro-architectural attacks: from CPU to browser

Clémentine Maurice, CNRS, CRIStAL  
@BloodyTangerine  
26 October 2022—RAID 2022 keynote
Execution leaves traces in components
Inspecting these traces allows retrieving secrets!
This requires surgical precision and a great control over CPU components...
This requires surgical precision and a great control over CPU components...
How do we do it from web browsers?
Cache Set Detection (3 min)

• last bit.

for all key bits until the majority of partial keys reached before proceeding to the next bit. This procedure is repeated (bold). The incorrect '0' of the second partial key is deleted since all partial key bits—except for the second one—are '1' recovered (underlined). The sixth key bit is recovered as '1',

Table II gives an example where the topmost 5 bits are already of the wrong partial key matches the recovered key bit again.

other. We apply these actions via majority vote until the key bit is a list of actions necessary to transform one key into the window of a few bits. The output of the edit distance algorithm not over the whole partial keys but only over a lookahead reduce performance overhead, we calculate the edit distance, edit distance to all partial keys that won the majority vote. To correct the current bit of the wrong partial key, we compute the footprint and self-containment of $mbedTLS$ that acts as the victim. As discussed in Section III-B, we building a malware enclave attacking a co-located enclave.

In this section, we evaluate the presented methods by Table III shows various RSA key sizes and the correspond-

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

A. RSA Key Sizes and Exploitation

...the victim. The attacked buffer spans 9 cache sets, out of which 6 show low bit-error rate, as shown in Figure 9. For

$0 1 1 1 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 1 1 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 0 0 0 1 1 1 0 0 0 0 1 ...$ the victim. The attacked buffer spans 9 cache sets, out of which 6 show low bit-error rate, as shown in Figure 9. For
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**

![Graph](image)

- Number of accesses
- Access time [CPU cycles]

### Table II

<table>
<thead>
<tr>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

### Table III

<table>
<thead>
<tr>
<th>Key Size</th>
<th>RSA Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td></td>
</tr>
</tbody>
</table>

### Figure 9

- The 9 cache sets that are used by a key and their error rate
- The peaks in the resampled trace on the bottom clearly indicate '1's.
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

- Cache Set Detection (3 min)
- Prime+Probe (5 s)
- Pre-Processing (110 s)
- Key Recovery (20 s)

### Attack

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

---

**Table II**

<table>
<thead>
<tr>
<th>No. Recovered Key</th>
<th>Access Time [CPU cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table III**

<table>
<thead>
<tr>
<th>RSA Key Sizes</th>
<th>Possible Attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096 b</td>
<td>Prime+Probe</td>
</tr>
<tr>
<td></td>
<td>Bypassing hardware errors</td>
</tr>
<tr>
<td></td>
<td>Bypassing OS security (ASLR)</td>
</tr>
</tbody>
</table>

**Fig. 7**

- Resampled trace on the bottom clearly indicates '1's.

**Fig. 9**

- The attacked buffer spans 9 cache sets, out of which 6 show low bit-error rate, as shown in Figure 9.
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...

- new microarchitectures yearly

2011: Sandy Bridge
2012: Ivy Bridge
2013: Haswell
2014: Broadwell
2015: Skylake
2016: Kaby Lake
2017: Coffee Lake
2018: Whiskey Lake
2019: Comet Lake/Ice Lake
2020: Tiger Lake
From small optimizations...

• new microarchitectures yearly
• performance improvement ≈ 5%
From small optimizations...

- new microarchitectures yearly
- performance improvement ≈ 5%
- very small optimizations: caches, branch prediction...
... To microarchitectural side-channel attacks

- microarchitectural side channels come from these optimizations
... To microarchitectural side-channel attacks

- microarchitectural side channels come from these optimizations
- several processes are sharing microarchitectural components
To microarchitectural side-channel attacks

- microarchitectural side channels come from these optimizations
- several processes are sharing microarchitectural components
- attacker infers information from a (vulnerable) victim process via hardware usage
... To microarchitectural side-channel attacks

- microarchitectural side channels come from these optimizations
- several processes are sharing microarchitectural components
- attacker infers information from a (vulnerable) victim process via hardware usage
- pure-software attacks by unprivileged processes
... To microarchitectural side-channel attacks

- microarchitectural side channels come from these optimizations
- several processes are sharing microarchitectural components
- attacker infers information from a (vulnerable) victim process via hardware usage
- pure-software attacks by unprivileged processes
- sequences of benign-looking actions → hard to detect
Side-channel attacks
Overview of micro-architectural attacks
Outline

Overview of micro-architectural attacks
Porting micro-architectural attacks to the Web
Overview of micro-architectural attacks
Algorithm 1: Square-and-multiply exponentiation

Input: base \( b \), exponent \( e \), modulus \( n \)

Output: \( b^e \mod n \)

\[
X \leftarrow 1 \\
for \ i \leftarrow \text{bitlen}(e) \ downto 0 \ do \\
\quad X \leftarrow \text{multiply}(X, X) \\
\quad \text{if } e_i = 1 \ then \\
\quad \quad X \leftarrow \text{multiply}(X, b) \\
\end{for} \\
end \\
return X \]
1. Which **software implementation** is vulnerable?

2. Which **hardware component** is vulnerable?
1. 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

For example: CVE-2016-0702, CVE-2016-2178, CVE-2016-7440, CVE-2016-7439, CVE-2016-7438, CVE-2018-0495,
CVE-2020-16150
2. 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
Each component shared by two processes is a potential micro-architectural side-channel vector.
Hyper-threading: Same-core attacks

- threads sharing one core **share resources**: L1, L2 cache, branch predictor, TLB...

Translation leak-aside buffer
USENIX Sec'18

PortSmash
S&P'19

L1d, L1i, L2 cache attacks
BSDCon'05, CT-RSA'06

Branch Prediction
CT-RSA'07
Background: Execution pipeline

- 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

<table>
<thead>
<tr>
<th>secret == 0</th>
<th>secret == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>POPCNT %r8,%r8</td>
<td>VPBROADCAST %xmm0, %ymm0</td>
</tr>
<tr>
<td>POPCNT %r8,%r8</td>
<td>VPBROADCAST %xmm0, %ymm0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>POPCNT %r8,%r8</td>
<td>VPBROADCAST %xmm0, %ymm0</td>
</tr>
<tr>
<td>POPCNT %r8,%r8</td>
<td>VPBROADCAST %xmm0, %ymm0</td>
</tr>
</tbody>
</table>

Contention on Port 1

Contention on Port 5

Monitors port usage
Port contention side-channel attack

Victim

secret == 0

secret == 1

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

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

Contention on Port 1

Secret is 0!
Port contention side-channel attack

secret == 0

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

secret == 1

VPBROADCAST %xmm0, %ymm0
VPBROADCAST %xmm0, %ymm0
...
VPBROADCAST %xmm0, %ymm0
VPBROADCAST %xmm0, %ymm0

Contention on Port 5

Secret is 1!
Port contention: applications

- end-to-end attack on a TLS server (OpenSSL 1.1.0h): recovers a P-384 ECDSA private key
  - secret dependent on double-and-add operations of `ec_wNAF_mul` point multiplication
- SMoTherSpectre, a speculative code-reuse attack

---

A. Bhattacharyya et al. “SMoTherSpectre: Exploiting Speculative Execution through Port Contention”. In: CCS. 2019.
Possible side channels using components shared by a core?
Possible side channels using components shared by a core?

Stop sharing a core!
Cross-core attacks!

- cores also share resources: L3 cache, Ring Interconnect, GPU...

Translation leak-aside buffer
USENIX Sec'18

PortSmash
S&P'19

LLC attacks
USENIX'14, S&P'15

Grand Pwning Unit
S&P'18

L1d, L1i, L2
cache attacks
BSDCon'05, CT-RSA'06

Branch Prediction
CT-RSA'07

Lord of the Ring(s)
USENIX Sec'21
Cache timing differences

- Number of accesses
- Access time [CPU cycles]

- cache hits
- cache misses
From theoretical to practical cache attacks

- first theoretical attack in 1996 by Kocher
- first practical attack on RSA in 2005 by Percival, on AES in 2006 by Osvik et al.
- renewed interest for the field in 2014 after Flush+Reload by Yarom and Falkner
- even more interest in 2018 after the disclosure of Spectre and Meltdown

Cache attacks: Flush+Reload

Step 1: Attacker maps shared library (shared memory, in cache)
Cache attacks: Flush+Reload

Step 1: Attacker maps shared library (shared memory, in cache)
Cache attacks: Flush+Reload

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

**Step 2:** Attacker **flushes** the shared cache line
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
Flush+Reload: Applications

- side channel attacks on cryptographic primitives:
  - RSA: 96.7% of secret key bits in a single signature
  - AES: full key recovery in 30000 dec. (a few seconds)
- attacks against pseudorandom number generators
- attacks against RSA key generation
- revival of Bleichenbacher attacks on TLS

Possible side channels using memory deduplication?
Possible side channels using memory deduplication?

Disable memory deduplication!
Cache attacks: Prime+Probe

Victim address space

Cache

Attacker address space
**Cache attacks: Prime+Probe**

**Step 1:** Attacker *primes*, i.e., fills, the cache (no shared memory)
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
Challenges with Prime+Probe

We need to evict cache lines without \texttt{clflush} or shared memory:

1. which addresses do we access to have congruent cache lines?
2. without any privilege?
3. and in which order do we access them?

We need:

1. an \textit{eviction set}: addresses in the same set, in the same slice (issue #1 and #2)
2. an \textit{eviction strategy} (issue #3)
Prime+Probe: Applications

- **cross-VM** side channel attacks on crypto implementations:
  - El Gamal (sliding window): full key recovery in 12 min.
- covert channels between virtual machines in the **cloud**

---

C. Maurice et al. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud”. In: NDSS’17. 2017.
Prime+Probe: Applications

- cross-VM side channel attacks on crypto implementations:
  - El Gamal (sliding window): full key recovery in 12 min.
- covert channels between virtual machines in the cloud
- tracking user behavior in the browser, in JavaScript

C. Maurice et al. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud”. In: NDSS’17. 2017.
Possible side channels using components shared by a CPU?
Possible side channels using components shared by a CPU?

Stop sharing a CPU!?
Cross-CPU attacks!

- CPUs also share resources: DRAM

Translation leak-aside buffer
USENIX Sec'18
PortSmash
S&P'19

L1d, L1i, L2 cache attacks
BSDCon'05, CT-RSA'06

Branch Prediction
CT-RSA'07

LLC attacks
USENIX'14, S&P'15

Grand Pwning Unit
S&P'18

Lord of the Ring(s)
USENIX Sec'21

DRAMA
USENIX Sec'16
Porting micro-architectural attacks to the Web
JavaScript is code executed in a sandbox. It can't do anything nasty since it's in a sandbox, right?
Porting micro-architectural attacks to the Web

- side-channel attacks on the cache, DRAM, MMU, (...), and transient execution attacks like Spectre, ret2spec, RIDL, (...), are coming to web browsers
- very low-level attacks in a high-level language with many abstraction layers in between
- complex but not impossible to perform
- fundamentally hard or impossible to fix in the browser

Side-channel attacks in JavaScript?

- 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
Measuring time
High-resolution timers?

• 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

JavaScript: `performance.now()` has the highest resolution performance.now(). ... represent times as floating-point numbers with up to microsecond precision. — Mozilla Developer Network
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

- **Firefox 41**
  - Resolution: 5 µs
- **Firefox 57.0.4**
  - Resolution: 20 µs
- **Firefox 59**
  - Resolution: 2 ms
- **Firefox 60**
  - Resolution + jitter: 1 ms
- **Firefox 60.4**
  - Resolution + jitter: 20 µs
- **Chrome 44**
  - Resolution: 5 µs
- **Chrome 57.0.4**
  - Resolution + jitter: 100 µs
- **Chrome 72**
  - Resolution + jitter: 5 µs
- **Chrome 79 & COOP/COEP**
  - Resolution: 20 µs

It was better before

• before September 2015: `performance.now()` had a nanosecond resolution

---

It was better before

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

---

It was better before

- before September 2015: `performance.now()` had a nanosecond resolution
- Oren et al. demonstrated cache side-channel attacks in JavaScript
- “fixed” in Firefox 41: `rounding to 5 µs`

We can do better!

- microsecond resolution is not enough

We can do better!

- microsecond resolution is **not enough**
- two approaches

---

We can do better!

- microsecond resolution is not enough
- two approaches
  1. recover a higher resolution from the available timer

We can do better!

- microsecond resolution is **not enough**
- 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
• 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

![Diagram showing clock edges and increments](image-url)
Recovering resolution: Clock interpolation

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

![Clock edge diagram]

- 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

+1 +1 +1 +1 +1 +1 +1 +1 +1

• to measure with high resolution
  • start measurement at clock edge
  • increment a variable until next clock edge
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
• lowering timer resolution is not enough
• adding jitter → makes clock interpolation inefficient (need to redo the measurements to get rid of noise)

• lowering timer resolution is not enough
• adding jitter $\rightarrow$ makes clock interpolation inefficient (need to redo the measurements to get rid of noise)
$\rightarrow$ has no impact on SharedArrayBuffers!

Jitter?

• lowering timer resolution is not enough
• adding jitter → makes clock interpolation inefficient (need to redo the measurements to get rid of noise)
→ 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!

Cache attacks in browsers
Cache attacks: Challenges with JavaScript

1. No high-resolution timers
2. No instruction to flush the cache
3. No knowledge about physical addresses
→ we can distinguish cache hits from cache misses (only $\approx 150$ cycles difference)!

---

Cache attacks in JavaScript: applications

- spying on user behavior: detect mouse and network activity
- covert channel across origins
- covert channel host-to-VM
- website fingerprinting

Other micro-architectural attacks in browsers?
Spectre Attacks: Exploiting Speculative Execution

Other micro-architectural attacks in browsers

T. Rokicki, C. Maurice, and P. Laperdrix.

"Sok: In search of lost time: A review of javascript timers in browsers".

Bonus: you don’t even need JavaScript!

**Attack 5: CSS Prime+Probe**

```html
<div id="pp" class="AAA...AAA">
   <div id="s1">X</div>
   <div id="s2">X</div>
   <div id="s3">X</div>
</div>

Search non existing string

   ==
   Probe the LLC

</div>

#pp:not([class*='jigbaa']) #s1 {
   background-image: url('https://knbdsd.badserver.com');
}

#pp:not([class*='akhevn']) #s2 {
   background-image: url('https://pjemh7.badserver.com');
}

Resolve non existing image

   ==
   TIMER
```

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
- any shared component is a potential side-channel vector
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
- any **shared component** is a potential side-channel vector
- it’s **really** hard not to share a component
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
• any shared component is a potential side-channel vector
• it’s really hard not to share a component
• micro-architectural attacks require a low-level understanding and control over the components, usually achieved with native code
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
• any shared component is a potential side-channel vector
• it’s really hard not to share a component
• micro-architectural attacks require a low-level understanding and control over the components, usually achieved with native code
• but it’s still possible to carry these attacks on from web browsers
Thank you!

Contact

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

Clémentine Maurice, CNRS, CRISTAL
@BloodyTangerine
26 October 2022—RAID 2022 keynote