

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

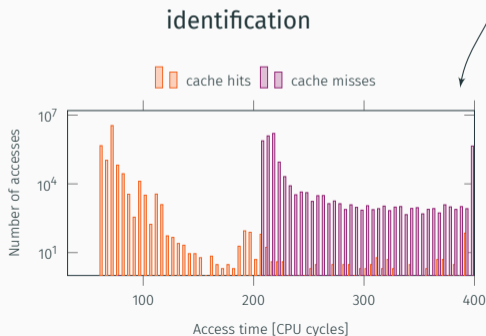
- **hardware** usually modeled as an abstract layer behaving correctly, but possible attacks

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

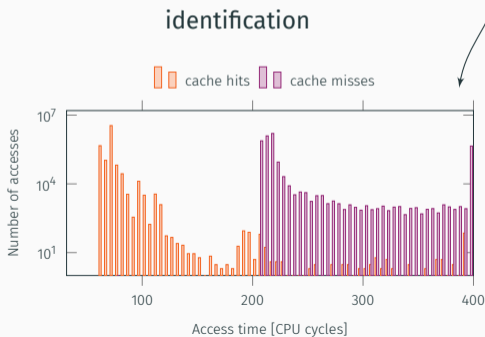
Attacks on micro-architecture

- **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



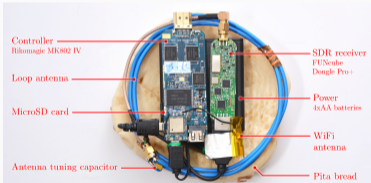
Attacks on micro-architecture

- **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



Attacker model

Hardware-based attacks a.k.a physical attacks



Physical access to hardware
→ embedded devices

VS

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

From small optimizations to side-channel attacks...



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

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

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: $b^e \bmod n$

$X \leftarrow 1$

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

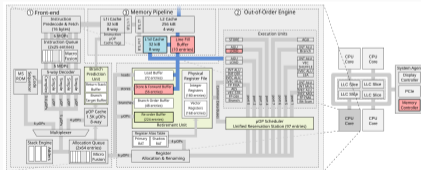
end

return X

Hardware



&



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

applications



OS



hardware



Outline

applications



OS

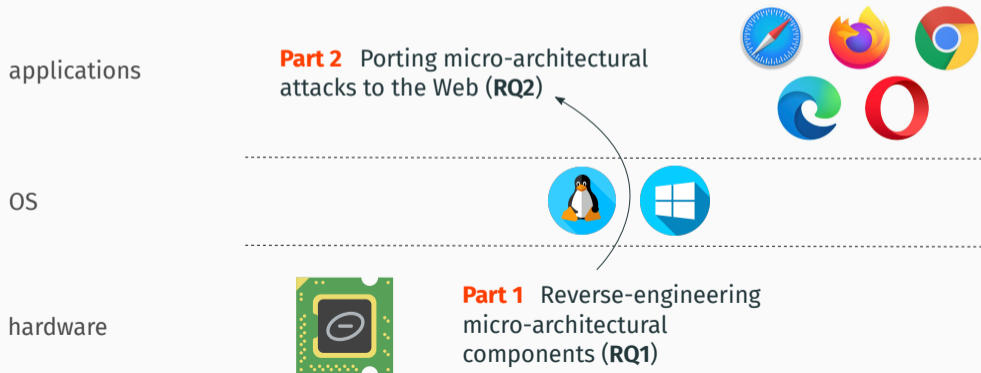


hardware

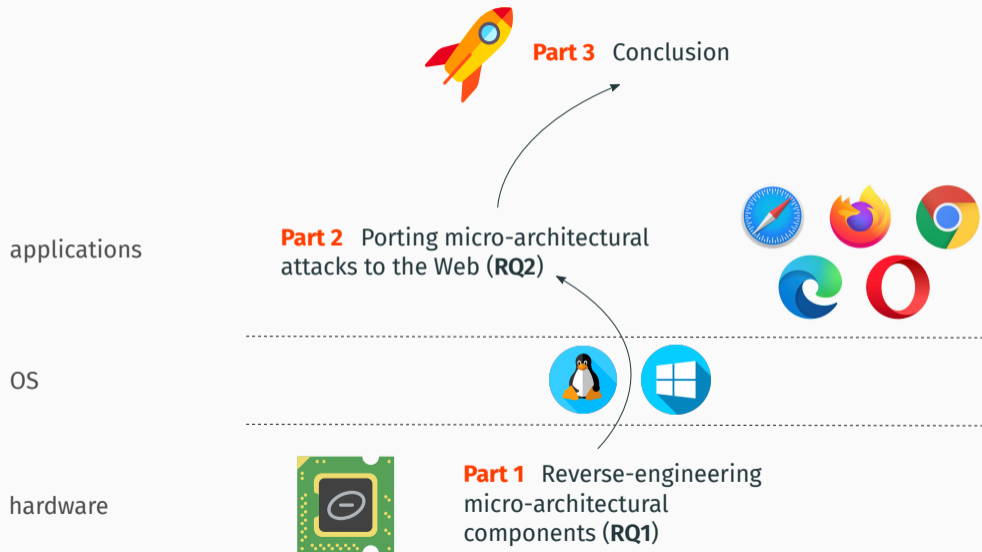


Part 1 Reverse-engineering
micro-architectural
components (**RQ1**)

Outline



Outline

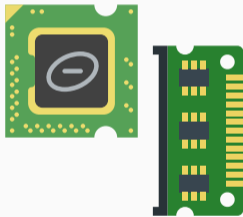


Reverse-engineering micro-architectural components

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?

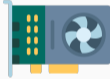
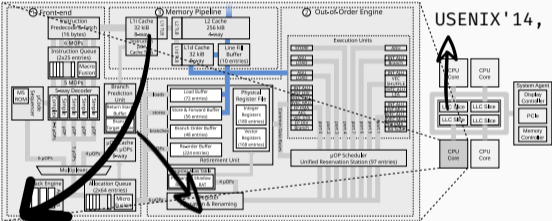
Translation look-aside buffer
USENIX Sec'18

CPU Ports
S&P'19

LLC attacks
USENIX '14, S&P'15



DRAM
USENIX Sec'16



GPU
S&P'18

L1d, L1i, L2 cache
BSDCon'05, CT-RSA'06,
ASIACCS'20

Branch Prediction
CT-RSA'07

Ring Interconnect
USENIX Sec'21, DIMVA'21

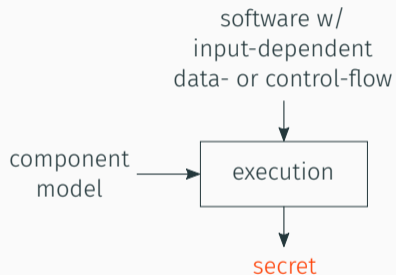
State of the art in 2015:

only the cache and the branch predictor were explored

Motivation

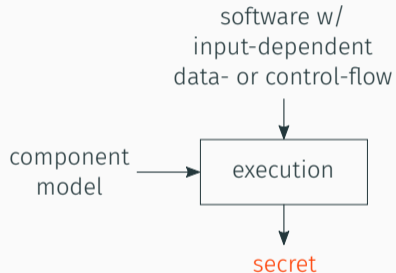
- performance optimizations are mostly **undocumented**
 - side channels come from these optimizations
- understanding them is crucial to **characterize the attack surface**: build new or improve known side-channel primitives

Side-channel analysis

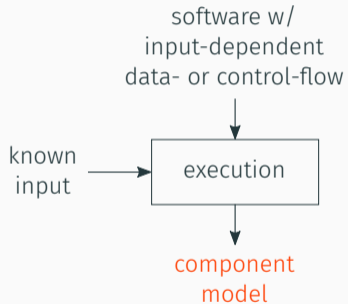


General approach

Side-channel analysis

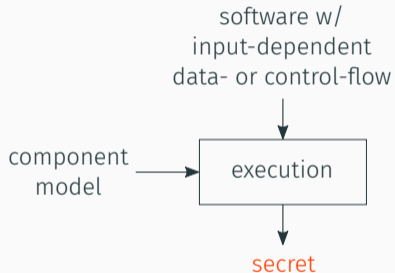


Reverse engineering

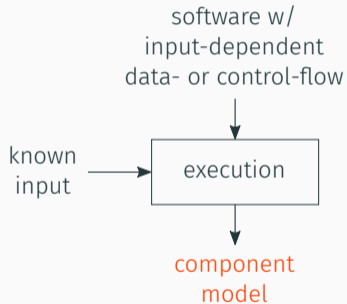


General approach

Side-channel analysis

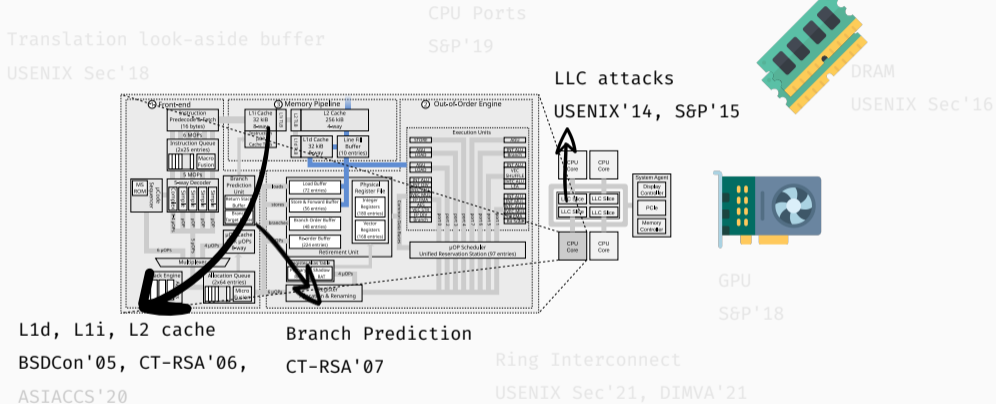


Reverse engineering



Reverse-engineering is the **opposite operation** of side-channel analysis

RQ1. Which hardware component leaks information?

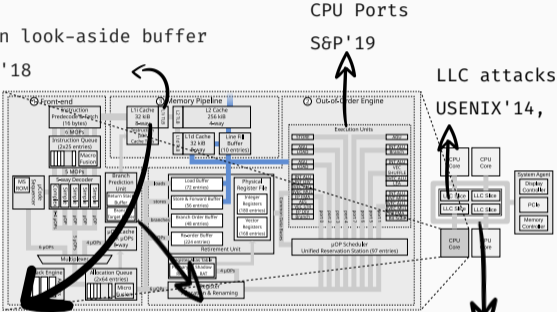


State of the art in 2015:

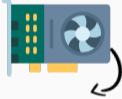
only the cache and the branch predictor were explored

RQ1. Which hardware component leaks information?

Translation look-aside buffer
USENIX Sec '18



USENIX Sec '16



GPU
S&P '18

L1d, L1i, L2 cache
BSDCon'05, CT-RSA'06,
ASIACCS'20

Branch Prediction
CT-RSA'07

Ring Interconnect
USENIX Sec'21, DIMVA'21

State of the art today: each component shared by two processes is a potential micro-architectural side-channel vector

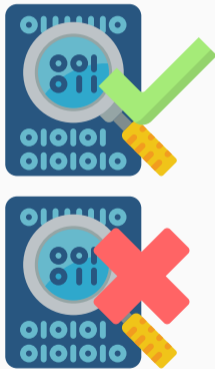
Porting micro-architectural attacks to the Web

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

For example: CVE-2016-0702, CVE-2016-2178, CVE-2016-7440, CVE-2016-7439, CVE-2016-7438, CVE-2018-0495,
CVE-2018-0737, CVE-2018-10846, CVE-2019-9495, CVE-2019-13627, CVE-2019-13628, CVE-2019-13629,
CVE-2020-16150



RQ2b. How to deliver the attack?

4 COLIN PERCIVAL

```
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.

State of the art in 2015

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

applications



OS



hardware



How to obtain such **low-level control** from a **high-level abstraction layer**?

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

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

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?

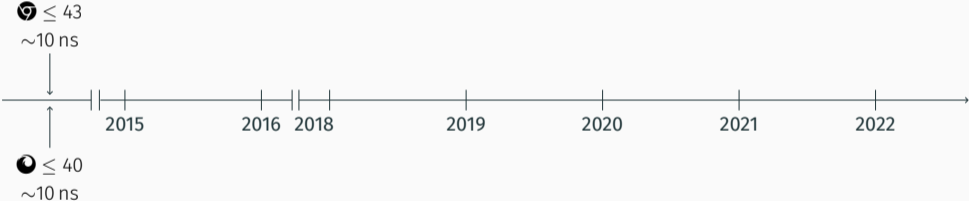
- measure small timing differences: need a **high-resolution timer**
- native: `rdtsc`, timestamp in CPU cycles
- JavaScript: `performance.now()` has the highest resolution

performance.now()

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

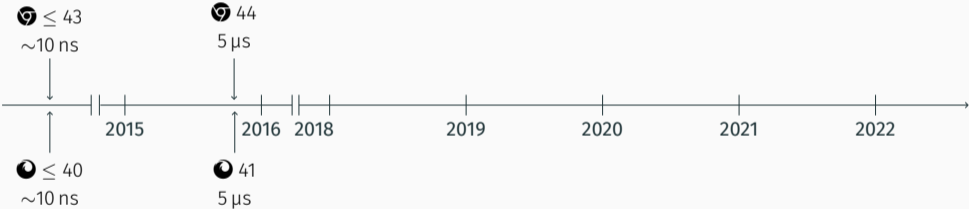
— Mozilla Developer Network

Evolution of timers until today: resolution and countermeasures

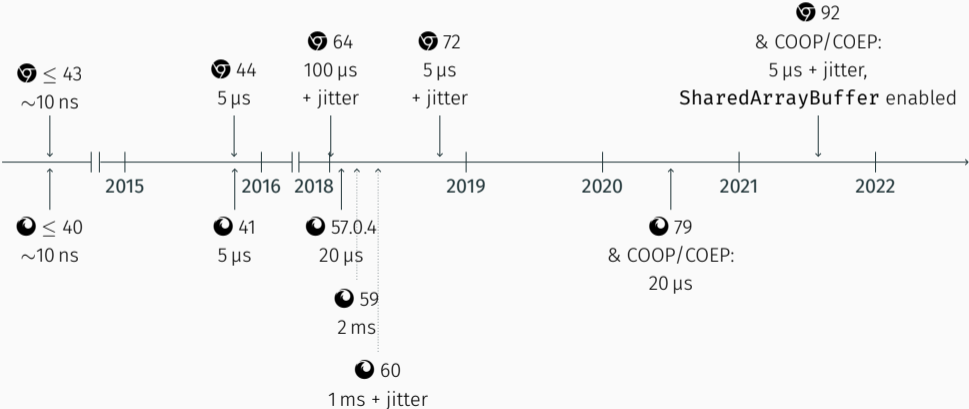


T. Rokicki, C. Maurice, and P. Laperdrix. "Sok: In search of lost time: A review of javascript timers in browsers". In: *EuroS&P*. 2021

Evolution of timers until today: resolution and countermeasures

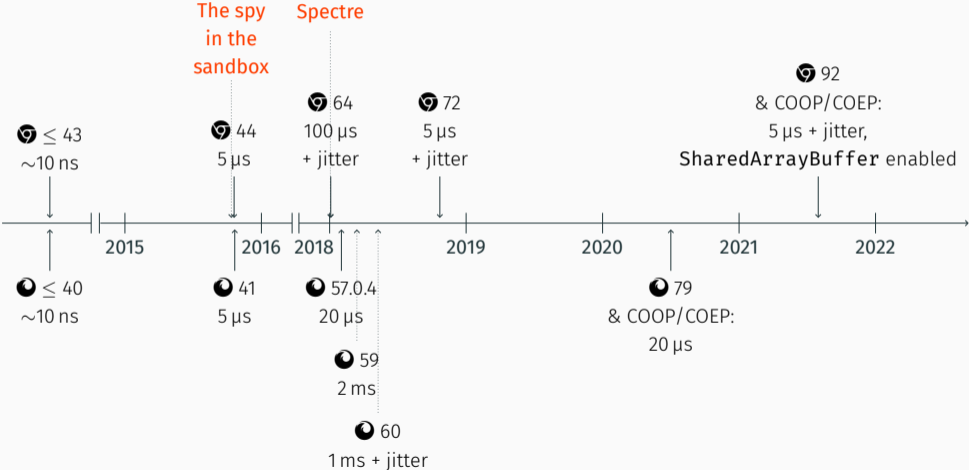


Evolution of timers until today: resolution and countermeasures



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



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

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

- `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 μ s**

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

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

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

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

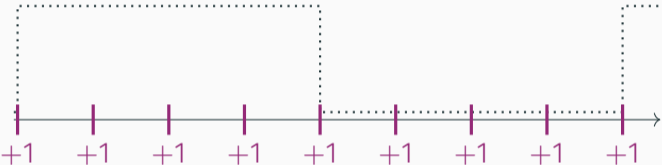
Recovering resolution: Clock interpolation

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



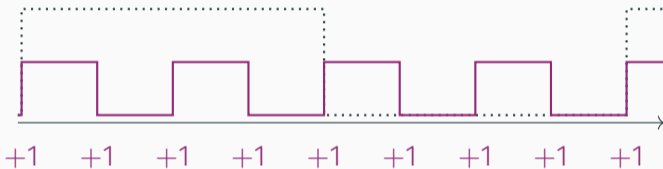
Recovering resolution: Clock interpolation

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



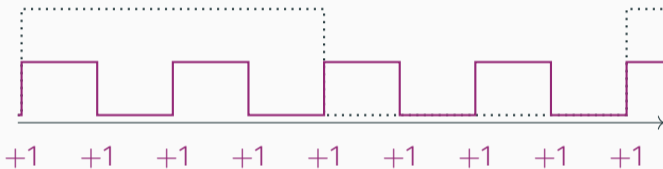
Recovering resolution: Clock interpolation

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



Recovering resolution: Clock interpolation

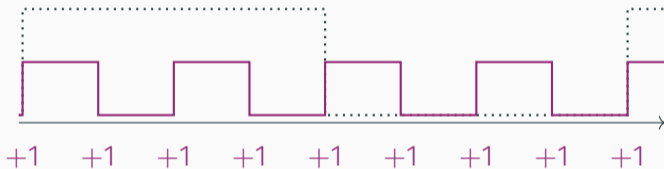
- **measure** how often we can **increment** a variable between two timer ticks



- to measure with high resolution

Recovering resolution: Clock interpolation

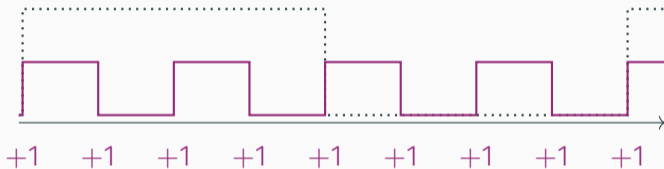
- **measure** how often we can **increment** a variable between two timer ticks



- to measure with high resolution
 - start measurement at **clock edge**

Recovering resolution: Clock interpolation

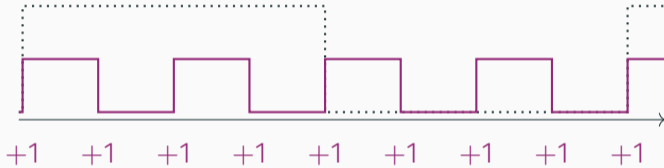
- **measure** how often we can **increment** a variable between two timer ticks



- 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 μ s

- 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**


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



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!



Jitter?



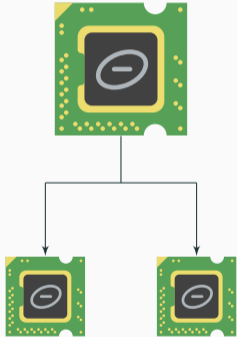
T. Rokicki, C. Maurice, and P. Laperdrix. “Sok: In search of lost time: A review of javascript timers in browsers”. In: *EuroS&P*. 2021



- 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

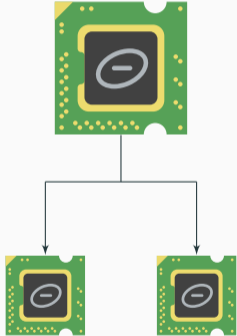
Background: Hyper-threading



Simultaneous computation technology of Intel.

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

Background: Hyper-threading

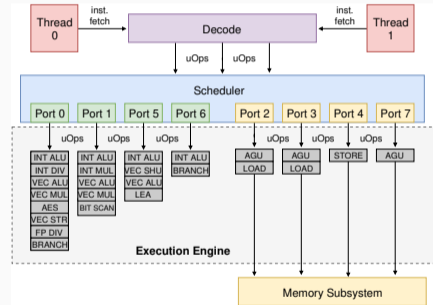


Simultaneous computation technology of Intel.

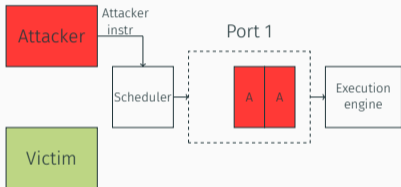
- physical cores are shared between logical cores
 - abstraction at the OS level
- hardware resources are shared between logical cores

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



No contention

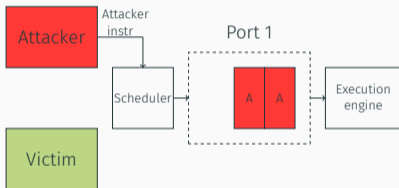


All attacker instructions are
executed in a row

→ fast execution time

Port contention

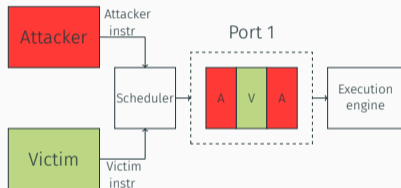
No contention



All attacker instructions are executed in a row

→ fast execution time

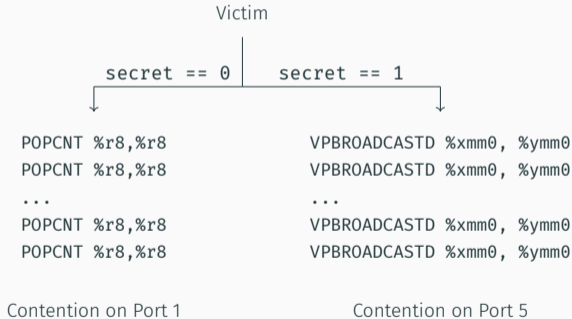
Contention



Victim instructions delay the attacker instructions

→ slow execution time

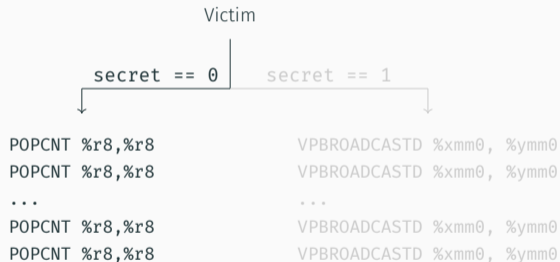
Port contention side-channel attack



← Monitors port usage →



Port contention side-channel attack

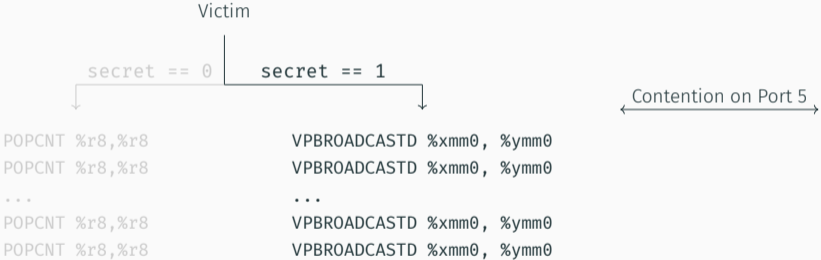


Contention on Port 1




Secret is 0!

Port contention side-channel attack



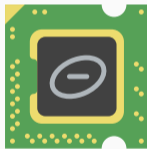
Secret is !

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

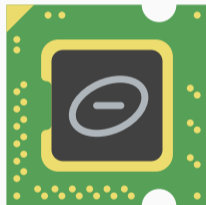
We just solved this problem :)

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

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

#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

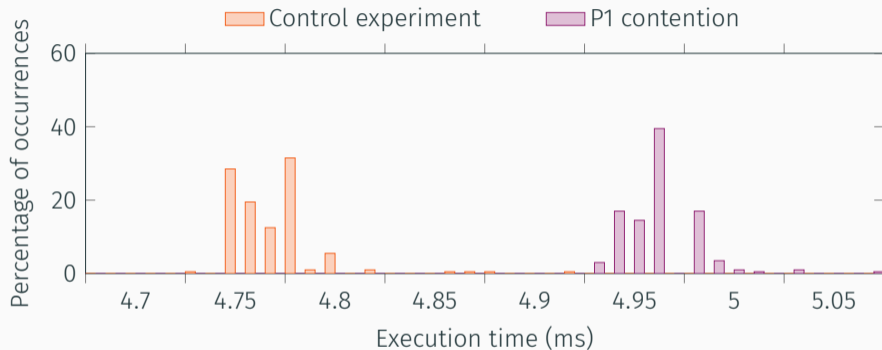


- sandboxed
- JIT compilation



- 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

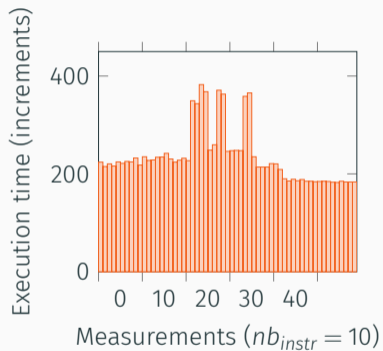


Figure 1: Secret key: 1101001.

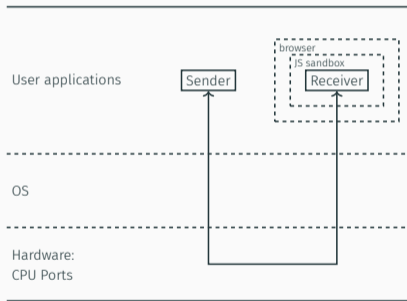
- spatial resolution: 1024 native instructions
- similar to other web-based cache attacks
- timers are the main bottleneck

Port contention covert channel: native-to-web

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

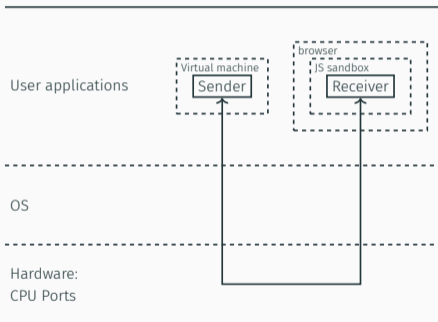
Evaluation:

- 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



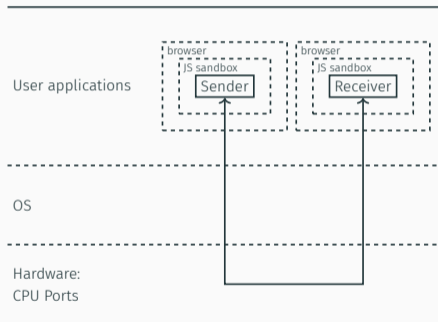
More port contention covert channels

VM-to-host



80 bit/s bandwidth

Cross-browser



200 bit/s bandwidth (physical layer), across different browsers!

RQ2b. How to deliver the attack?

4 COLIN PERCIVAL

```
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.

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 today: many Web-based micro-architectural attacks



Spectre Attacks: Exploiting Speculative Execution

Paul Kocher¹, Jann Horn², Anders Fogh³, Daniel Genkin⁴, Daniel Gruss⁵, Werner Haas⁶, Mike Hamburg⁷, Moritz Lipp⁵, Stefan Mangard⁵, Thomas Prescher⁸, Michael Schwarz⁵, Yuval Yarom⁸

¹ Independent (www.paukocher.com), ² Google Project Zero, ³ G DATA Advanced Analytics, ⁴ University of Pennsylvania and University of Maryland, ⁵ Graz University of Technology, ⁶ Cyberbus Technology, ⁷ Rambus, Cryptography Research Division, ⁸ University of Adelaide and Data61

Conclusion

- 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@cncrs.fr`

 @BloodyTangerine

Micro-architectural attacks: from CPU to browser

Clémentine Maurice, CNRS, CRIStAL

28 June 2023—Journées nationales du GDR Sécurité Informatique