IN CYBER FORUM

EUROPE

26–28 MAR. 2024

LILLE GRAND PALAIS

@FIC_eu

@forumincybereurope

EN/FR

Attaques micro-architecturales : du CPU au navigateur

IN CYBER FORUM EUROPE

Micro-architectural attacks: from CPU to browser

MARCH 26 - 28, 2024 LILLE GRAND PALAIS

@FIC_eu

@forumincybereurope

EN/FR

INCYBER FORUM EUROPE

SPEAKER
INTERVENANTE

Clémentine MAURICE

Chargée de Recherche
CNRS

EN/FR

@FIC_eu

@forumincybereurope

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

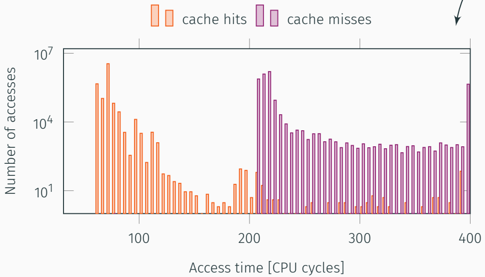
Attacks on micro-architecture

- 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

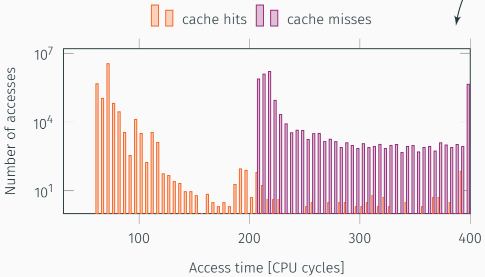
identification



Attacks on micro-architecture

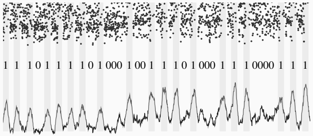
- hardware usually modeled as an abstract layer behaving correctly, but possible attacks
  - faults: bypassing software protections by causing hardware errors
  - side channels: observing side effects of hardware on computations

identification



→

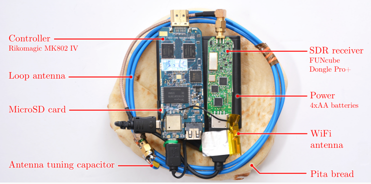
attack



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

Attacker model

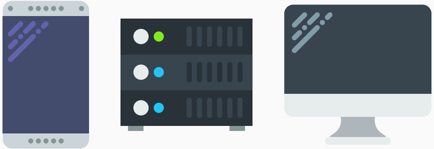
Hardware-based attacks
a.k.a physical attacks



Physical access to hardware
→ embedded devices

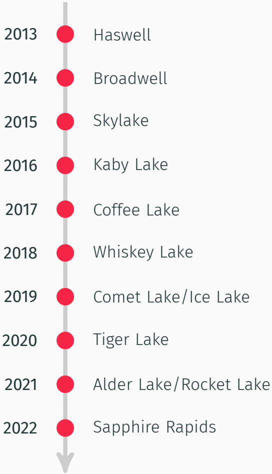
vs

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



Co-located or remote attacker
→ complex systems

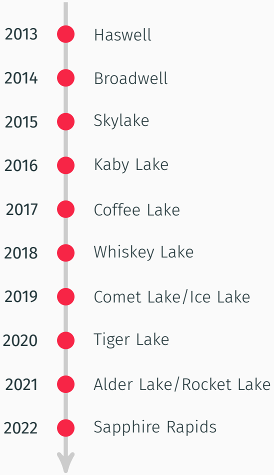
From small optimizations to side-channel attacks...



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

- new micro-architectures yearly

From small optimizations to side-channel attacks...



2013 Haswell

2014 Broadwell

2015 Skylake

2016 Kaby Lake

2017 Coffee Lake

2018 Whiskey Lake

2019 Comet Lake/Ice Lake

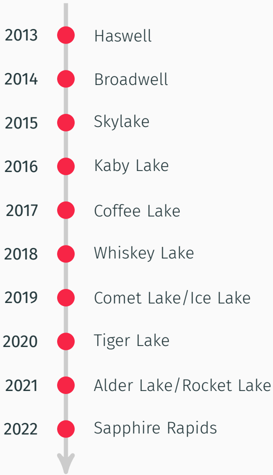
2020 Tiger Lake

2021 Alder Lake/Rocket Lake

2022 Sapphire Rapids

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

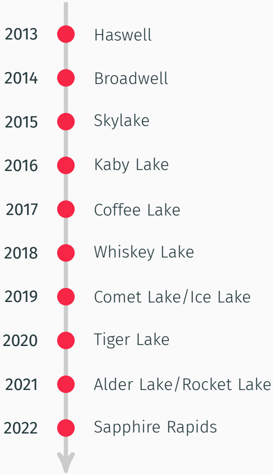
From small optimizations to side-channel attacks...



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

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

From small optimizations to side-channel attacks...



2013 Haswell

2014 Broadwell

2015 Skylake

2016 Kaby Lake

2017 Coffee Lake

2018 Whiskey Lake

2019 Comet Lake/Ice Lake

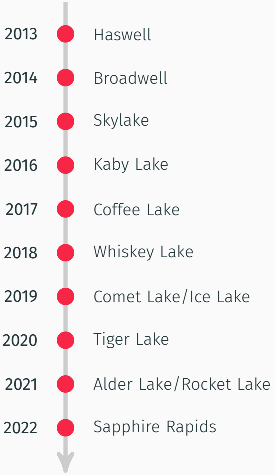
2020 Tiger Lake

2021 Alder Lake/Rocket Lake

2022 Sapphire Rapids

- 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...



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

- 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

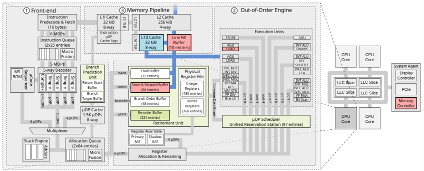
Micro-architectural side-channel attacks: Two faces of the same coin

Implementation

Hardware

```
Algorithm 1: Square-and-multiply exponentiation
Input: base $b$, exponent $e$, modulus $n$
Output: $b^e \mod n$
$X \leftarrow 1$
for $i \leftarrow bitlen(e)$ downto 0 do
    $X \leftarrow \text{multiply}(X, X)$
    if $e_i = 1$ then
        $X \leftarrow \text{multiply}(X, b)$
    end
end
return $X$
```

&



Research questions

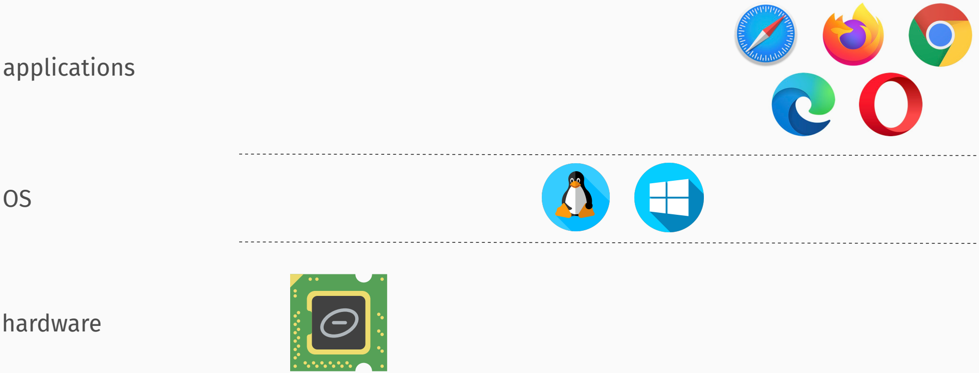
RQ1. Which hardware components are vulnerable...

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

RQ2. Which software implementation is vulnerable...

... and what are the different attack deliveries?

Outline

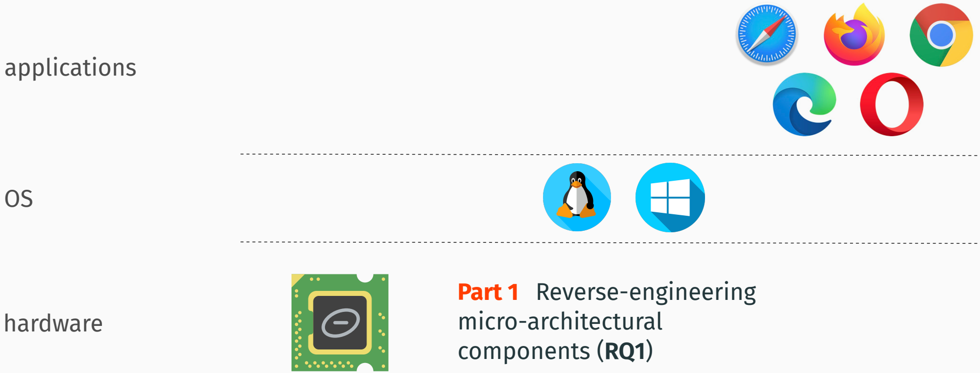


applications

OS

hardware

Outline



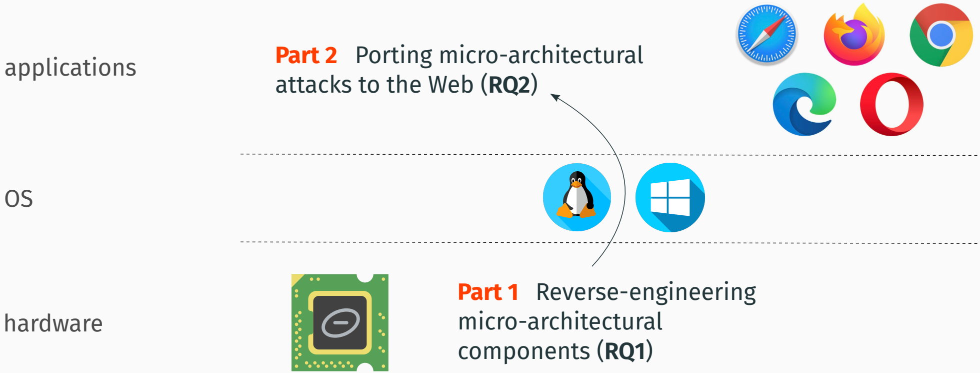
applications

OS

hardware

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

Outline



applications

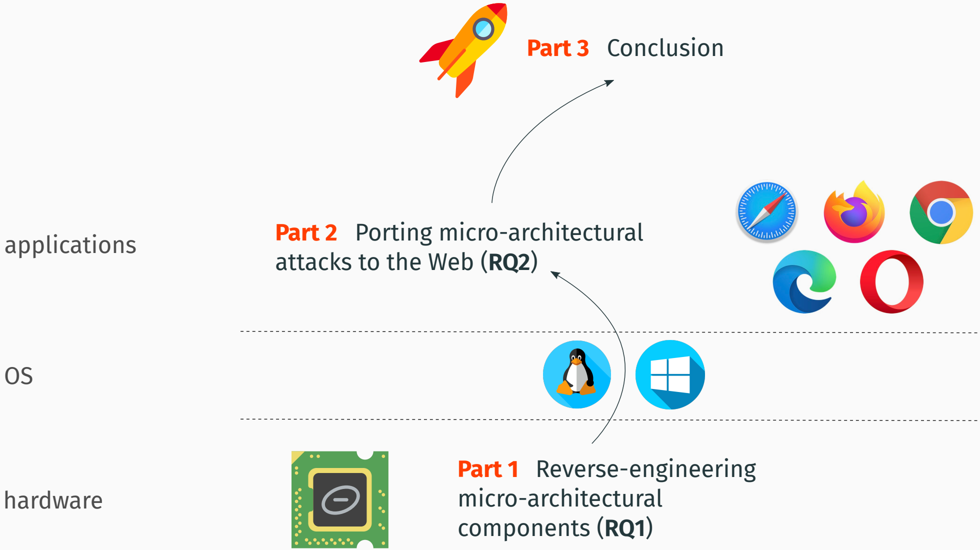
Part 2 Porting micro-architectural attacks to the Web (**RQ2**)

OS

hardware

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

Outline



Part 3 Conclusion

applications

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

OS

hardware

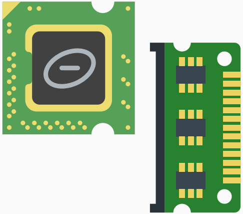
Part 1 Reverse-engineering micro-architectural components (**RQ1**)

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

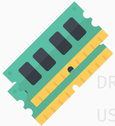
USENIX Sec'18

CPU Ports

S&P'19

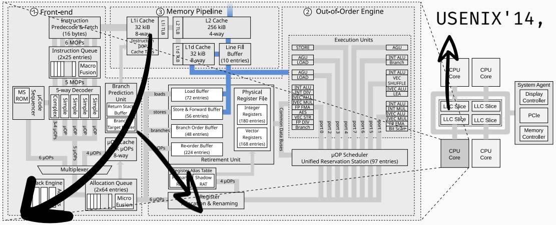
LLC attacks

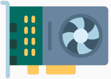
USENIX'14, S&P'15



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

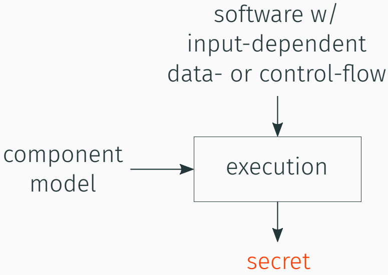
Motivation

- 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

General approach

Side-channel analysis


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

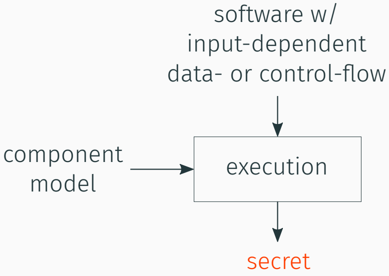
component model

execution

secret

General approach

Side-channel analysis



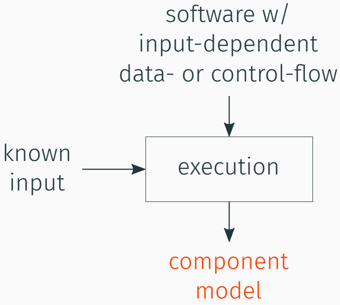
software w/
input-dependent
data- or control-flow

component
model

execution

secret

Reverse engineering



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

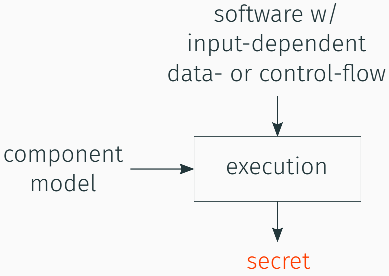
known
input

execution

component
model

General approach

Side-channel analysis



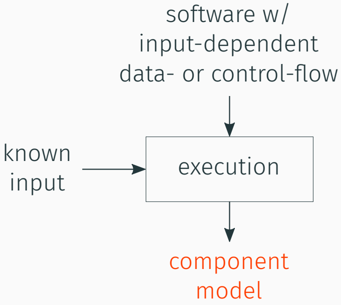
software w/ input-dependent data- or control-flow

component model

execution

secret

Reverse engineering



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

known input

execution

component model

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

RQ1. Which hardware component leaks information?

CPU Ports

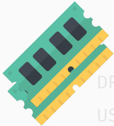
S&P'19

Translation look-aside buffer

USENIX Sec'18

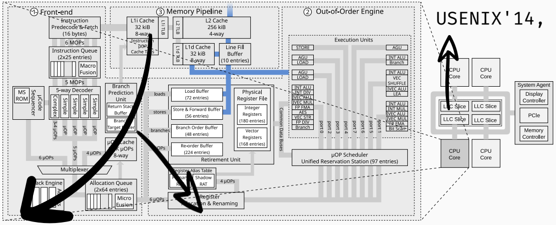
LLC attacks

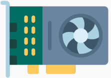
USENIX'14, S&P'15



DRAM

USENIX Sec'16





L1d, L1i, L2 cache

BSDCon'05, CT-RSA'06,

ASIACCS'20

Branch Prediction

CT-RSA'07

GPU

S&P'18

Ring Interconnect

USENIX Sec'21, DIMVA'21

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

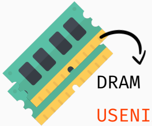
USENIX Sec'18

CPU Ports

S&P'19

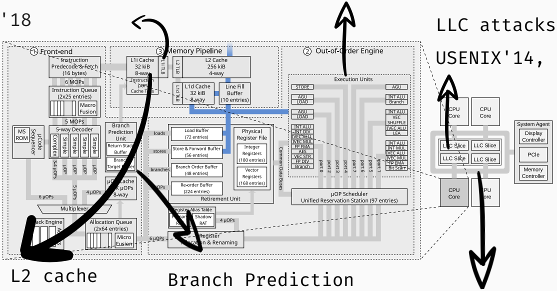
LLC attacks

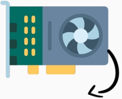
USENIX'14, S&P'15



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

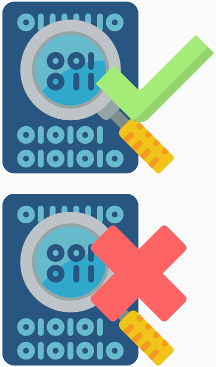
State of the art today: each component shared by two processes is a potential micro-architectural side-channel vector

Porting micro-architectural attacks to the Web

RQ2a. Which software implementation is vulnerable?

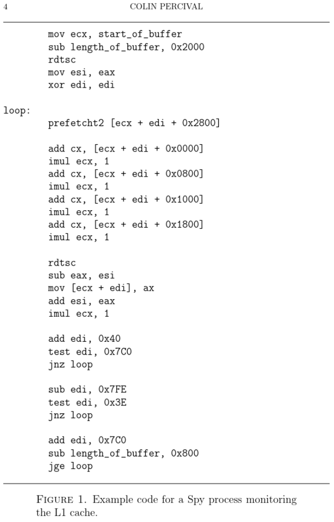
State of the art (more or less)

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.

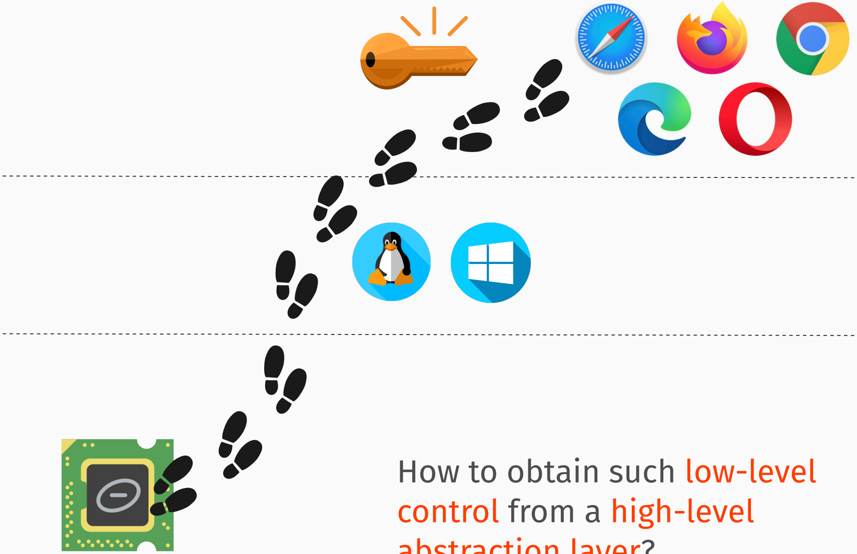
State of the art in 2015

- 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?

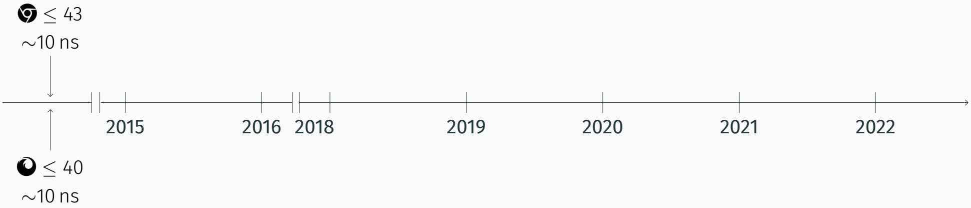
- measure small timing differences: need a high-resolution timer
- native: `rdtsc`, timestamp in CPU cycles
- JavaScript: `performance.now()` has the highest resolution

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

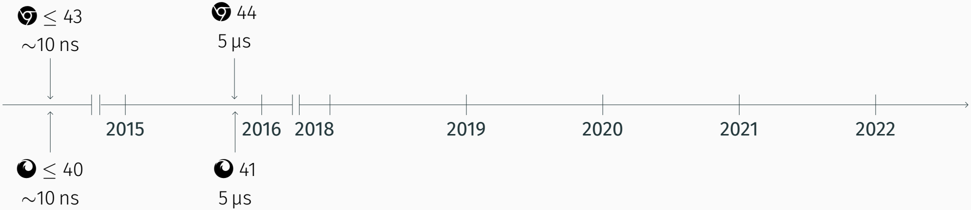
> **`performance.now()`**
>
> [...] represent times as floating-point numbers with up to microsecond precision.
>
> — Mozilla Developer Network

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



≤ 43
∼10 ns

44
5 µs

2015

2016

2018

2019

2020

2021

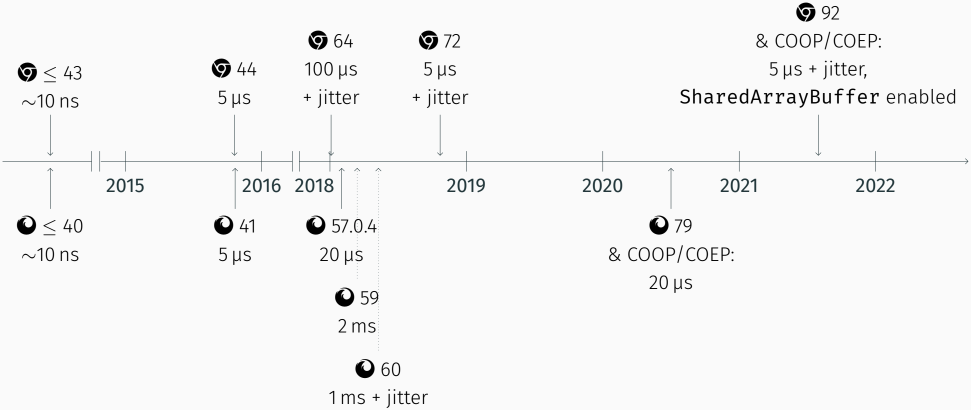
2022

≤ 40
∼10 ns

41
5 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



≤ 43
∼10 ns

44
5 µs

64
100 µs
+ jitter

72
5 µs
+ jitter

92
& COOP/COEP:
5 µs + jitter,
`SharedArrayBuffer` enabled

≤ 40
∼10 ns

41
5 µs

57.0.4
20 µs

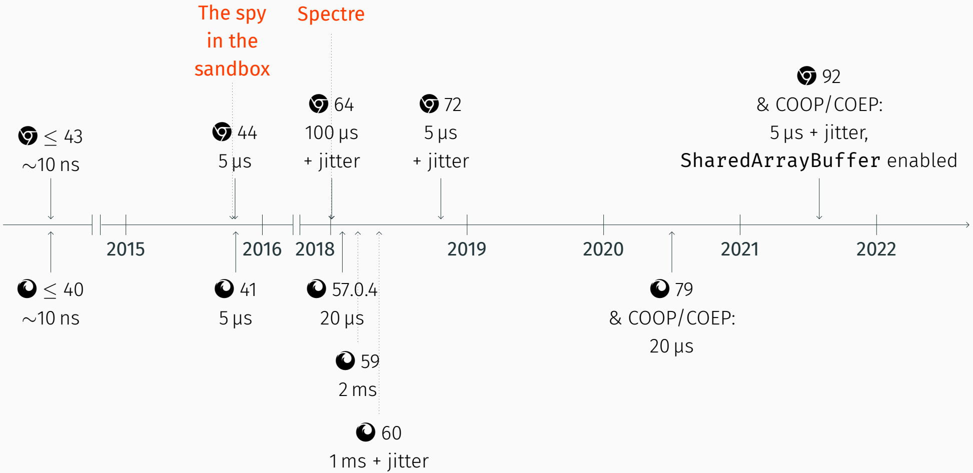
59
2 ms

60
1 ms + jitter

79
& 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

Spectre

≤ 43
~10 ns

44
5 µs

64
100 µs
+ jitter

72
5 µs
+ jitter

92
& COOP/COEP:
5 µs + jitter,
`SharedArrayBuffer` enabled

≤ 40
~10 ns

41
5 µs

57.0.4
20 µs

59
2 ms

60
1 ms + jitter

79
& 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

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
     → clock interpolation, resolution: Firefox/Chrome: 500 ns, Tor: 15 µs

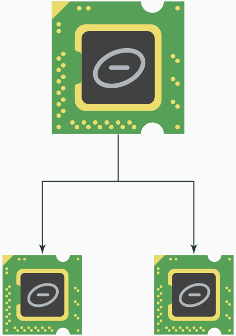
Is clamping an efficient countermeasure?

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
     → clock interpolation, resolution: Firefox/Chrome: 500 ns, Tor: 15 µs
  2. build our own high-resolution timer
     → using `SharedArrayBuffer`, resolution: Firefox: 2 ns, Chrome: 15 ns

Port contention attacks

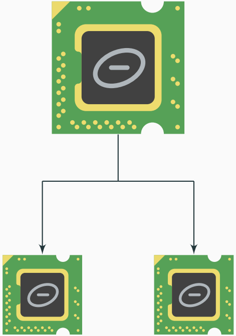
Background: Hyper-threading



Simultaneous computation technology of Intel.

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

Background: Hyper-threading



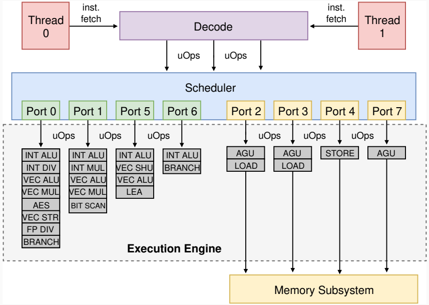
Simultaneous computation technology of Intel.

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

→ hardware resources are shared between logical cores

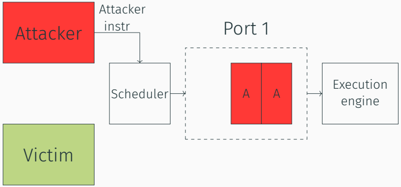
Background: Execution pipeline

- 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



Port contention

No contention

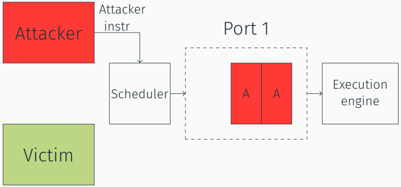


All attacker instructions are executed in a row
→ fast execution time

A. C. Aldaya et al. "Port Contention for Fun and Profit". In: *S&P*. 2019.

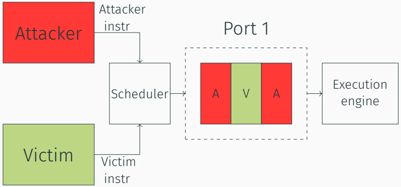
Port contention

No contention



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

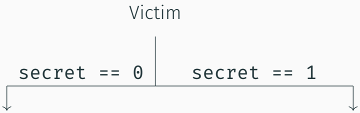
Contention



Victim instructions delay the attacker instructions
→ slow execution time

A. C. Aldaya et al. “Port Contention for Fun and Profit”. In: *S&P*. 2019.

Port contention side-channel attack



Victim

`secret == 0`

`secret == 1`

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

Contention on Port 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`

```
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 1

Secret is 0!

Port contention side-channel attack

Victim

```
secret == 0                           secret == 1

POPCNT %r8,%r8              VPBROADCASTD %xmm0, %ymm0
POPCNT %r8,%r8              VPBROADCASTD %xmm0, %ymm0
...                         ...
POPCNT %r8,%r8              VPBROADCASTD %xmm0, %ymm0
POPCNT %r8,%r8              VPBROADCASTD %xmm0, %ymm0
```

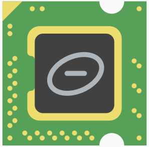
Contention on Port 5

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

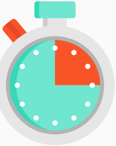
1. No high-resolution timers



2. No control on cores

3. No access to specific instructions

Port contention attacks: Solutions



1. No high-resolution timers

→ we just solved this problem

Port contention attacks: Solutions

1. No high-resolution timers

→ we just solved this problem

2. No control on cores

→ exploit JavaScript multi-threading and work with the scheduler

Port contention attacks: Solutions

1. No high-resolution timers

→ we just solved this problem

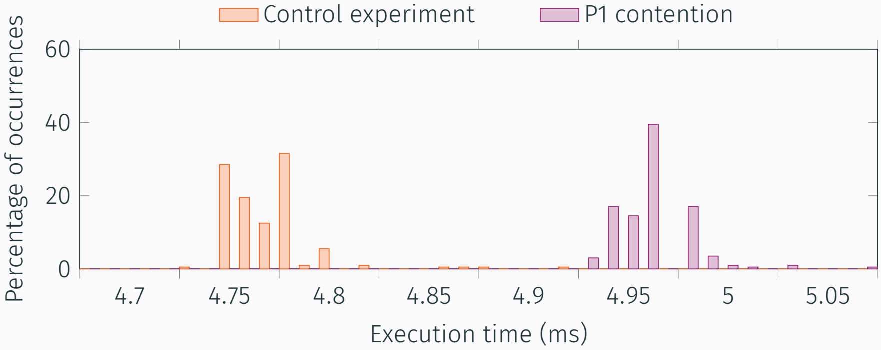
2. No control on cores

→ exploit JavaScript multi-threading and work with the scheduler

3. No access to specific instructions

→ use WebAssembly

Proof-of-concept native-to-web



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

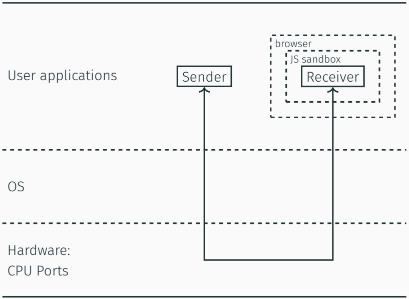
Web : WebAssembly repeatedly calls **`i64.ctz`** and times the execution

Port contention covert channel: native-to-web

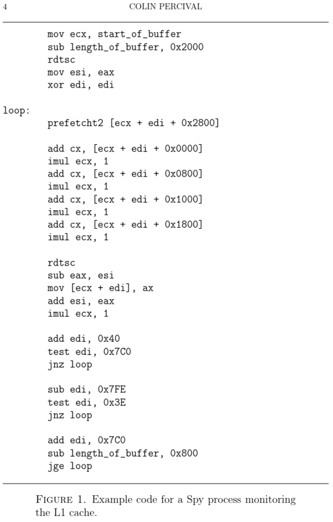
- **Native:** C/x86 sender
- **Web:** WebAssembly receiver

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



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

ASLR on the Line: Practical Cache Attacks on the MMU

Ben Gras* Kaveh Razavi* Erik Bosman Herbert Bos Cristiano Giuffrida
Vrije Universiteit Amsterdam
{beng, kaveh, ejbosman, herbertb, giuffrida}@cs.vu.nl
* Equal contribution joint first authors

Abstract—Address space layout randomization (ASLR) is an important first line of defense against memory corruption attacks and a building block for many modern countermeasures. Existing attacks against ASLR rely on software vulnerabilities and/or on repeated (and detectable) memory probing.

In this paper, we show that neither is a hard requirement and that ASLR is *fundamentally* insecure on modern cache-based architectures, making ASLR and caching conflicting requirements (ASLR⊕Cache, or simply AnC). To support this claim, we describe a new EVICT+TIME cache attack on the virtual address translation performed by the memory management unit (MMU) of modern processors. Our AnC attack relies on the property that the MMU's page-table walks result in caching page-table pages in the shared last-level cache (LLC). As a result, an attacker can derandomize virtual addresses of a victim's *code and data* by locating the cache lines that store the page-table entries used for address translation.

Relying only on basic memory accesses allows AnC to be implemented in JavaScript without any specific instructions or

bruteforcing, if at all possible [16], [60], requires repeat-
edly generating anomalous events (e.g., crashes [5], [17],
[55], exceptions [19], [23], or huge allocations [47]) that are
easy to detect or prevent. For instance, for some attacks [6]
disabling non-fundamental memory management features is
enough [63]. Consequently, even if ASLR does not sto
more advanced attackers, in the eyes of many
as a good first line of defense for prot
as a pivotal building block in m
[15], [36], [42], [52]. In thi
systematically derando
attack on the memory m
that we call ASLR⊕Cach

Previous work has sh
the presence of specific we
features in software. For insta
ASLR if the application is vu
if the system turns on mem

Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript

Daniel Gruss, Clémentine Maurice†, and Stefan Mangard

Graz University of Technology, Austria

Abstract. A fundamental assumption in software security is that a
emory location can only be modified by processes that may write to
mory location. However, a recent study has shown that parasitic
AM can change the content of a memory cell without ac-
cessing other memory locations in a high frequency.
nammer bug occurs in most of today's memory mod-
consequences for the security of all affected systems,
alation attacks.
attacks related to Rowhammer so far rely on the avail-
che flush instruction in order to cause accesses to DRAM
sufficiently high frequency. We overcome this limitation by
mplex cache replacement policies. We show that caches can
into fast cache eviction to trigger the Rowhammer bug with
lar memory accesses. This allows to trigger the Rowhammer
ighly restricted and even scripting environments.
monstrate a fully automated attack that requires nothing but a
te with JavaScript to trigger faults on remote hardware. Thereby
an gain unrestricted access to systems of website visitors. We show
t the attack works on off-the-shelf systems. Existing countermeasures
il to protect against this new Rowhammer attack.

Spectre Attacks: Exploiting Speculative Execution

Paul Kocher[1], Jann Horn[2], Anders Fogh[3], Daniel Genkin[4],
Daniel Gruss[5], Werner Haas[6], Mike Hamburg[7], Moritz Lipp[5],
Stefan Mangard[5], Thomas Prescher[6], Michael Schwarz[5], Yuval Yarom[8]
[1] Independent (www.paulkocher.com), [2] Google Project Zero,
[3] G DATA Advanced Analytics, [4] University of Pennsylvania and University of Maryland,
[5] Graz University of Technology, [6] Cyberus Technology,
[7] Rambus, Cryptography Research Division, [8] University of Adelaide and Data61

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!

Merci !

Contact:

@BloodyTangerine
clementine.maurice@cnrs.fr

IN CYBER FORUM

EUROPE

26–28 MAR. 2024

LILLE GRAND PALAIS

𝕏 @FIC_eu

in @forumincybereurope

EN/FR