Micro-architectural attacks: from CPU to browser

Clémentine Maurice, CNRS
@BloodyTangerine
July 7 2022—Summer School “Cyber in Nancy”, Nancy, France
Joint work with

- Thomas Rokicki (IRISA, France)
- Yossi Oren, Marina Botvinnik (Ben Gurion University, Israel)
- Daniel Gruss, Michael Schwarz, Moritz Lipp, Raphael Spreitzer, Peter Pessl, Stefan Mangard (TU Graz, Austria)
- and many other co-authors!
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?
• Chapter 1  Introduction to micro-architectural attacks
• Chapter 2  Side-channel techniques
• Chapter 3  Side-channel attacks from web browsers
Chapter 1: Introduction to micro-architectural attacks
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
Attacker model for software-based attacks

- no physical access to the device
Attacker model for software-based attacks

- no physical access to the device
- can execute unprivileged code on the same machine as victim
Attacker model for software-based attacks

- no physical access to the device
- can execute unprivileged code on the same machine as victim
- what are the scenarios in which this happens?
Attacker model for software-based attacks

- no physical access to the device
- can execute unprivileged code on the same machine as victim
- what are the scenarios in which this happens?
  - you install some program on your machine/smartphone
  - you have a virtual machine on some physical machine (cloud)
  - some JavaScript runs on a web page
Micro-architectural attacks: scope

Everyday hardware: servers, workstations, laptops, smartphones...
Algorithm 1: Square-and-multiply exponentiation

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

$X \leftarrow 1$

for $i \leftarrow \text{bitlen}(e) \text{ downto } 0$ do
    $X \leftarrow \text{multiply}(X, X)$
    if $e_i = 1$ then
        $X \leftarrow \text{multiply}(X, b)$
    end
end

return $X$
Research questions

1. Which **software implementation** is vulnerable?

2. Which **hardware component** is vulnerable?
Type of attacks

Active attacks: destroying the vault
Passive attacks: listening to the vault internal mechanisms

Active attacks: laser, varying temperature, clock glitching...
Passive attacks: timing, power consumption, electromagnetic radiation...
Type of attacks

**active attacks**: destroying the vault

**passive attacks**: listening to the vault internal mechanisms
Type of attacks

**active attacks:** destroying the vault

**passive attacks:** listening to the vault internal mechanisms

**active attacks:** laser, varying temperature, clock glitching...

**passive attacks:** timing, power consumption, electromagnetic radiation...
1. Fault attacks

2. Side-channel attacks
Type of attacks

1. Fault attacks

2. Side-channel attacks

3. Transient execution attacks
Fault attacks

- pushing hardware **outside of its functional requirements** (power, heat, clock...) to trigger a fault in the system
- most fault attacks are hardware-based ones, but it is possible to **trigger hardware faults in software** too (Rowhammer)
- most of the gaming consoles that have been hacked have been by fault injection

https://media.ccc.de/search/?q=console+hacking
Side-channel attacks

- exploit the implementation of a system
- based on channels that are outside of the functional specification, i.e., that are not supposed to carry useful information
- however these channels can leak secret information
Side channels in "real life" (1/2)
Side channels in "real life" (2/2)

http://content.time.com/time/subscriber/article/0,33009,970860,00.html
Side channels in computers

- **safe software** infrastructure → no bugs, e.g., buffer overflows

- Information leaks because of implementation and hardware

- No "bug" in the sense of a mistake → lots of performance optimizations

- Crypto and sensitive info., e.g., keystrokes and mouse movements
Side channels in computers

- **safe software** infrastructure $\rightarrow$ no bugs, e.g., buffer overflows
- does not mean safe execution

- information leaks because of implementation and hardware
- no "bug" in the sense of a mistake $\rightarrow$ lots of performance optimizations

- crypto and sensitive info., e.g., keystrokes and mouse movements
Side channels in computers

- **safe software** infrastructure $\rightarrow$ no bugs, e.g., buffer overflows
- does not mean safe execution
- information **leaks** because of **implementation** and **hardware**
- no “bug” in the sense of a mistake $\rightarrow$ lots of performance optimizations
Side channels in computers

- **safe software** infrastructure → no bugs, e.g., buffer overflows
- does not mean safe execution
- information leaks because of implementation and hardware
- no “bug” in the sense of a mistake → lots of performance optimizations

→ crypto and sensitive info., e.g., keystrokes and mouse movements
To perform a side-channel attack on some software you need both:

- shared and vulnerable hardware
  - no side channel if every memory access takes the same time
  - or if you cannot share the hardware component
- a vulnerable implementation
  - vulnerable implementation \( \neq \) vulnerable algorithm
Hardware vs. implementations

To perform a side-channel attack on some software you need both:

- shared and vulnerable hardware
  - no side channel if *every* memory access takes the same time
  - or if you cannot share the hardware component
- a vulnerable implementation
  - vulnerable implementation $\neq$ vulnerable algorithm
  - we can attack specific implementations of AES and RSA
  - does not mean that AES and RSA are broken
  $\rightarrow$ not all implementations are created equal

https://access.redhat.com/blogs/766093/posts/1976303
An example of side channel (1)

post '/login' do
  if not valid_user(params[:user])
    "Username incorrect"
  else
    if verify_password(params[:user], params[:password])
      "Access granted"
    else
      "Password incorrect"
    end
  end
end
post '/login' do
  if not valid_user(params[:user])
    "Username or password incorrect"
  else
    if verify_password(params[:user], params[:password])
      "Access granted"
    else
      "Username or password incorrect"
    end
  end
end

An example of side channel (3)

post '/login' do
  if not valid_user(params[:user])
    "Username or password incorrect"
    busy_wait()
  else
    if verify_password(params[:user], params[:password])
      "Access granted"
    else
      "Username or password incorrect"
    end
  end
end
Is constant timing enough?
Each component shared by two processes is a potential micro-architectural side-channel vector.
Side channels: Caches, DRAM, GPU, TLB, CPU ports, Ring interconnect...

Translation leak-aside buffer
USENIX Sec'18

PortSmash
S&P'19

DRAMA
USENIX Sec'16

Lord of the Ring(s)
USENIX Sec'21

Grand Pwning Unit
S&P'18
Hardware: From small optimizations to side channels

- New micro-architectures yearly
- Performance improvement ≈ 5%
- Very small optimizations: caches, branch prediction...
- Leading to side channels
- No documentation on this intellectual property
Hardware: From small optimizations to side channels

- new micro-architectures yearly
- performance improvement \( \approx 5\% \)
Hardware: From small optimizations to side channels

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

Years:
- 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
Hardware: From small optimizations to side channels

- New micro-architectures yearly
- Performance improvement $\approx 5\%$
- Very small optimizations: caches, branch prediction...
- ... leading to side channels

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
Hardware: From small optimizations to side channels

- New micro-architectures yearly
- Performance improvement $\approx 5\%$
- Very small optimizations: caches, branch prediction...
- ... leading to side channels
- No documentation on this intellectual property
What can we do with side-channel attacks?
RSA encryption

Generating an RSA encryption system requires the following steps:

- randomly selecting two prime numbers \( p \) and \( q \) and calculating \( n = pq \)
- choosing a public exponent \( e \). GnuPG uses \( e = 65537 \)
- calculating a private exponent \( d = e^{-1} \mod (p - 1)(q - 1) \)

The private key is the triple \((p, q, d)\).

The decryption function is \( D(c) = c^d \mod n \).
Generating an RSA encryption system requires the following steps:

- randomly selecting two prime numbers $p$ and $q$ and calculating $n = pq$
- choosing a public exponent $e$. GnuPG uses $e = 65537$
- calculating a private exponent $d = e^{-1} \mod (p - 1)(q - 1)$

The private key is the triple $(p, q, d)$.

The decryption function is $D(c) = c^d \mod n$

But multiplying $c$ by itself $d$ times is too slow!
Generating an RSA encryption system requires the following steps:

- randomly selecting two prime numbers \( p \) and \( q \) and calculating \( n = pq \)
- choosing a public exponent \( e \). GnuPG uses \( e = 65537 \)
- calculating a private exponent \( d = e^{-1}(\text{mod}(p - 1)(q - 1)) \)

The private key is the triple \((p, q, d)\).

The decryption function is \( D(c) = c^d \mod n \)

But multiplying \( c \) by itself \( d \) times is too slow! → we have fast exponentiation implementations!
Algorithm 1: GnuPG 1.4.13 Square-and-multiply exponentiation

Input: base $c$, exponent $d$, modulus $n$

Output: $c^d \mod n$

$X \leftarrow 1$

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

$X \leftarrow \text{square}(X)$

$X \leftarrow X \mod n$

if $d_i = 1$ then

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

$X \leftarrow X \mod n$

end

end

return $X$
Attacking GnuPG 1.4.13 RSA exponentiation

- monitor the square and multiply functions with Flush+Reload to recover the bits of the secret exponent

Attacking GnuPG 1.4.13 RSA exponentiation

- monitor the square and multiply functions with Flush+Reload to recover the bits of the secret exponent

mbedTLS 2.3.0 RSA square-and-multiply exponentiation

mbedTLS version 2.3.0 (2017), "fixes" the issue with a single operation multiply

Algorithm 2: mbedTLS 2.3.0 Square-and-multiply exponentiation

Input: base $c$, exponent $d$, modulus $n$
Output: $c^d \mod n$

$X \leftarrow 1$

for $i \leftarrow \text{bitlen}(d)$ downto 0 do
  $X \leftarrow \text{multiply}(X,X)$
  $X \leftarrow X \mod n$
  if $d_i = 1$ then
    $X \leftarrow \text{multiply}(X,c)$
    $X \leftarrow X \mod n$
  end
end
return $X$
Attacking mbedTLS 2.3.0 RSA exponentiation

- raw Prime+Probe trace on the buffer holding the multiplier $c$
Attacking mbedTLS 2.3.0 RSA exponentiation

- raw Prime+Probe trace on the buffer holding the multiplier $c$
- processed with a simple moving average
Attacking mbedTLS 2.3.0 RSA exponentiation

- raw Prime+Probe trace on the buffer holding the multiplier $c$
- processed with a simple moving average
- allows to clearly recover the **bits of the secret exponent**
Let’s get back to our example

```
post '/login' do
  if not valid_user(params[:user])
    "Username or password incorrect"
    busy_wait()
  else
    if verify_password(params[:user], params[:password])
      "Access granted"
    else
      "Username or password incorrect"
    end
  end
end
end
```
Transient execution attacks

- novel class of attacks ≠ side-channel attacks
  → transient execution attacks leak the actual target data
- disclosed in 2018 with Spectre and Meltdown

https://transient.fail/
Transient execution attacks

• novel class of attacks $\neq$ side-channel attacks

→ transient execution attacks leak the actual target data

• disclosed in 2018 with Spectre and Meltdown

• SO MANY VARIANTS

https://transient.fail/
Transient execution attacks

- CPU avoids waiting for input data or availability of execution units
  → out-of-order execution and speculation
- sequential semantics is preserved
Transient execution attacks

- CPU avoids waiting for input data or availability of execution units
  → out-of-order execution and speculation
- sequential semantics is preserved

- some instructions are never committed, *i.e.*, finally executed
  - instructions that cause an exception + following instructions
  - instructions in branches that are mispredicted
- these instructions are called transient instructions
Transient execution attacks

- CPU avoids waiting for input data or availability of execution units
  → out-of-order execution and speculation
- sequential semantics is preserved
- some instructions are never committed, *i.e.*, finally executed
  - instructions that cause an exception + following instructions
  - instructions in branches that are mispredicted
- these instructions are called transient instructions
- architectural state → everything is fine
Transient execution attacks

- attacker uses a **covert channel** to encode the secret
- issue: instructions not committed leave traces in **microarchitecture**
- microarchitectural state is not supposed to be visible...
- ... but we know how to **recover the state of caches**
Transient execution attacks

- attacker uses a covert channel to encode the secret
- issue: instructions not committed leave traces in microarchitecture
- microarchitectural state is not supposed to be visible...
- ... but we know how to recover the state of caches
- microarchitectural state $\rightarrow$ everything is not fine
Transient execution attacks

- attacker uses a **covert channel** to encode the secret
- issue: instructions not committed **leave traces in microarchitecture**
- microarchitectural state is not supposed to be visible...
- ... but we know how to recover the state of caches
- microarchitectural state → everything is **not fine**

- leaking kernel memory, recovering passwords...
Transient execution attacks

• attacker uses a **covert channel** to encode the secret
• issue: instructions not committed **leave traces in microarchitecture**
• microarchitectural state is not supposed to be visible...
• ... but we know how to **recover the state of caches**
• microarchitectural state → everything is **not fine**

• leaking kernel memory, recovering passwords...
• difficult to fix: lazy error handling was a bug, but speculative execution is a feature!
Chapter 2: Side-channel techniques
Cache side-channel attacks
Data can reside in

- CPU Registers
- L1 Cache
- L2 Cache
- L3 Cache
- Memory
- Disk storage
Data can reside in

- CPU registers
Data can reside in

- CPU registers
- different levels of the CPU cache
Memory hierarchy

Data can reside in

- CPU registers
- different levels of the CPU cache
- main memory
Memory hierarchy

Data can reside in

- CPU registers
- different levels of the CPU cache
- main memory
- disk storage
Set-associative caches

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cache
Set-associative caches

Data loaded in a specific set depending on its address
Set-associative caches

Data loaded in a specific set depending on its address
Several ways per set
Set-associative caches

Data loaded in a specific set depending on its address
Several ways per set
Cache line loaded in a specific way depending on the replacement policy
Cache attacks

- cache attacks → exploit timing differences of memory accesses
Cache attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
Cache attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, **not the content**
- covert channel: two processes *communicating* with each other
  - *not allowed* to do so, e.g., across VMs
Cache attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
- covert channel: two processes communicating with each other
  - not allowed to do so, e.g., across VMs
- side-channel attack: one malicious process spies on benign processes
  - e.g., steals crypto keys, spies on keystrokes
Timing attacks

How every timing attack works:

- learn timing of different *corner cases*
- later, we recognize these corner cases by timing only
How every timing attack works:

• learn timing of different **corner cases**
• later, we recognize these corner cases by timing only
• here, corner cases: **hits and misses**
First step: building the histogram

1. build two cases: cache hits and cache misses
2. time each case many times (get rid of noise)
First step: building the histogram

1. build two cases: cache hits and cache misses
2. time each case many times (get rid of noise)
3. we have a histogram!
First step: building the histogram

1. build two cases: cache hits and cache misses
2. time each case many times (get rid of noise)
3. we have a **histogram**!
4. find a **threshold** to distinguish the two cases
Loop:

1. measure time
2. access variable (always cache hit)
3. measure time
4. update histogram with delta
Building the histogram: cache misses

Loop:

1. **flush** variable (**clflush** instruction)
2. measure time
3. access variable (always cache **miss**)
4. measure time
5. update histogram with delta
Timing differences

![Graph showing cache hits and misses over access time in CPU cycles. The x-axis represents access time in CPU cycles ranging from 50 to 400. The y-axis represents the number of accesses on a logarithmic scale, ranging from $10^1$ to $10^7$. The graph displays two sets of bars: orange bars for cache hits and purple bars for cache misses.]
Finding the threshold

- as high as possible → most cache hits are below
- no cache miss below

![Graph showing cache hits and cache misses over access time](image-url)
How to measure time accurately? (1/3)

- very short timings
- \texttt{rdtsc} instruction: cycle-accurate timestamps
How to measure time accurately? (1/3)

- very short timings
- \texttt{rdtsc} instruction: cycle-accurate timestamps

\[
\cdots
\texttt{rdtsc}
\texttt{function()}
\texttt{rdtsc}
\cdots
\]
• do you measure what you think you measure?
• do you measure what you think you measure?
• out-of-order execution
• do you measure what you think you measure?
• out-of-order execution → what is really executed

\[
\begin{align*}
\text{rdtsc} & \quad \text{rdtsc} & \quad \text{rdtsc} \\
\text{function()} & \quad \ldots & \quad \text{rdtsc} \\
\ldots & \quad \text{rdtsc} & \quad \text{function()} \\
\text{rdtsc} & \quad \text{function()} & \quad \ldots
\end{align*}
\]
How to measure time accurately? (3/3)

- use pseudo-serializing instruction `rdtscp` (recent CPUs)
• use pseudo-serializing instruction `rdtscp` (recent CPUs)
• and/or use serializing instructions like `cpuid`
How to measure time accurately? (3/3)

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

Cache attacks techniques

- two (main) techniques
  1. **Flush+Reload** (Gullasch et al., Osvik et al., Yarom et al.)
  2. **Prime+Probe** (Percival, Osvik et al., Liu et al.)

- exploitable on **x86** and **ARM**

- used for both covert channels and side-channel attacks

- many variants: Flush+Flush, Evict+Reload, Prime+Scope, Prime+Abort...

---

Spatial and temporal resolution

- **spatial** resolution: what can I monitor? A page? A set? A line?
  → a spatial resolution of a 4KB page means that you cannot distinguish two memory accesses within a 4KB page
- **temporal** resolution: how often can I perform a monitoring operation?
  → a temporal resolution of 1ms means that you cannot monitor more than one event every 1ms: if an event happens every $1\mu s$, you can only capture 0.1% of events
Spatial and temporal resolution

• **spatial** resolution: what can I monitor? A page? A set? A line?
  → a spatial resolution of a 4KB page means that you cannot distinguish two memory accesses within a 4KB page

• **temporal** resolution: how often can I perform a monitoring operation?
  → a temporal resolution of 1ms means that you cannot monitor more than one event every 1ms: if an event happens every $1\mu s$, you can only capture 0.1% of events

Both influence the type of attacks that you can perform: an attacker that can only monitor a 4KB page every minute obtains less information than an attacker that can monitor a cache line every 100ns.
Cache attack: Flush+Reload

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

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

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

**Step 2:** Attacker flushes the shared cache line
Cache attack: 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
Cache attack: 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

- cross-VM (memory-deduplication enabled) 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

Flush+Reload: Pros and cons

Pros

- high spatial resolution: 1 line
- high temporal resolution

Cons

- restrictive
  1. needs `clflush` instruction (not available e.g., on ARM-v7)
  2. needs shared memory
Flush+Reload: Shared memory? (1/2)

Shared library → shared in physical memory
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

mmap() creates a new mapping in the virtual address space of the calling process. [...] The contents of a file mapping are initialized using length bytes starting at offset offset in the file (or other object) referred to by the file descriptor fd.
Flush+Reload: Shared memory? (2/2)

Page deduplication

Virtual Address Space

Process A

Process B

Physical Address Space
Flush+Reload: Shared memory? (2/2)

Page deduplication

Processes started independently
Flush+Reload: Shared memory? (2/2)

Page deduplication
Flush+Reload: Shared memory? (2/2)

Page deduplication
Flush+Reload: Shared memory? (2/2)

Page deduplication
Flush+Reload: Shared memory? (2/2)

Page deduplication
Page deduplication

Flush+Reload: Shared memory? (2/2)
Flush+Reload: Shared memory? (2/2)

Page deduplication

Virtual Address Space

Process A

Deduplication Thread

Process B

Physical Address Space
Page deduplication
Flush+Reload: Shared memory? (2/2)

Page deduplication

Virtual Address Space

Deduplication Thread

Physical Address Space

Process A

Process B

≠
Flush+Reload: Shared memory? (2/2)

Page deduplication

Virtual Address Space

Deduplication Thread

\[\neq\]

Physical Address Space
Flush+Reload: Shared memory? (2/2)

Page deduplication
Flush+Reload: Shared memory? (2/2)

Page deduplication

Virtual Address Space

Deduplication Thread

Physical Address Space

Process A

Process B
Flush+Reload: Shared memory? (2/2)

Page deduplication

Virtual Address Space

Process A

Deduplication Thread

Process B

Physical Address Space
Page deduplication
Flush+Reload: Shared memory? (2/2)

Page deduplication

Virtual Address Space

Process A

Deduplication Thread

Physical Address Space

Process B
Flush+Reload: Shared memory? (2/2)

Page deduplication

Process A

Virtual Address Space

Deduplication Thread

Process B

Physical Address Space

Done!
Flush+Reload: Shared memory? (2/2)

Page deduplication

Virtual Address Space

Process A

Deduplication Thread

Process B

Physical Address Space
What if there is no shared memory?
What if there is no shared memory?

There is no memory deduplication, and no accessible shared library from browsers
Inclusive property

- **inclusive LLC**: superset of L1 and L2

![Diagram showing core 0 and core 1 with L1, L2, and LLC levels]
Inclusive property

- **inclusive LLC**: superset of L1 and L2
Inclusive property

- **inclusive LLC**: superset of L1 and L2
• inclusive LLC: superset of L1 and L2
Inclusive property

- **inclusive LLC**: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2
Inclusive property

- **inclusive LLC**: superset of L1 and L2
- Data evicted from the LLC is also evicted from L1 and L2
- A core can **evict lines** in the private L1 of another core
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)
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
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
Prime+Probe: Applications

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

Clémentine Maurice et al. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud”. In: NDSS. 2017.
Prime+Probe: Pros and cons

Pros

less restrictive
1. no need for `clflush`
2. no need for shared memory

Cons

- lower spatial resolution: 1 set
- lower temporal resolution: probe $n$ addresses to evict 1 line
- prone to noise
Prime+Probe in practice

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?

---

Prime+Probe in practice

We need to evict cache lines without `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 **eviction set**: addresses in the same set and same slice (issues #1 and #2)
2. an **eviction strategy**: the order in which we access the eviction set (issue #3)

---

Port contentention side-channel 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

secret == 1

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

Contention on Port 1

Contention on Port 5

Monitors port usage
Port contention side-channel attack

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

Contestion 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 \texttt{ec\_wNAF\_mul} point multiplication
- SMoTherSpectre, a speculative code-reuse attack

---

Atri Bhattacharyya et al. “SMoTherSpectre: Exploiting Speculative Execution through Port Contention”. In: CCS. 2019.
Port contention: Pros and cons

Pros

• very high spatial resolution: 1 instruction!
• high temporal resolution
• more resistant to noise if processes do not share a physical core
• no offline phase of creating an eviction set

Cons

• restrictive: requires SMT enabled + co-location on the same physical core
• mapping from instructions to port can change from one generation to another
Chapter 3: Side-channel attacks from web browsers
Side-channel attacks in JavaScript?

- JavaScript is code executed in a sandbox
- Can't do anything nasty since it is in a sandbox, right?
- Except 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?

- JavaScript is code executed in a sandbox
- can’t do anything nasty since it is in a sandbox, right?
Side-channel attacks in JavaScript?

- JavaScript is code executed in a sandbox
- can’t do anything nasty since it is in a sandbox, right?
- except side channels are only doing **benign operations**
Side-channel attacks in JavaScript?

- JavaScript is code executed in a sandbox
- can’t do anything nasty since it is in a sandbox, right?
- except side channels are only doing benign operations
  - all side-channel attacks: measuring time
Side-channel attacks in JavaScript?

- JavaScript is code executed in a sandbox
- can’t do anything nasty since it is in a sandbox, right?
- except 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
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: \texttt{rdtsc}, timestamp in CPU cycles

---

---
• 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** (2015): resolution 5 µs
- **Firefox 57.0.4** (2016): resolution 20 µs
- **Firefox 59** (2017): resolution 2 ms
- **Firefox 60** (2018): resolution + jitter 1 ms
- **Firefox 79** & COOP/COEP (2019): resolution 20 µs
- **Chrome 44** (2015): resolution 5 µs
- **Chrome 64** (2016): resolution + jitter 100 µs
- **Chrome 72** (2018): resolution + jitter 5 µs

Thomas Rokicki et al. “Sok: In search of lost time: A review of javascript timers in browsers”. In: EuroS&P. 2021
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!

- microphone 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

---

• 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
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

- Firefox/Chrome: 500 ns, Tor: 15 µs

• 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

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
Recovering resolution: Edge thresholding

- often sufficient to just see which of two functions takes longer
• often sufficient to just see which of two functions takes longer
• often sufficient to just see which of two functions takes longer
• often sufficient to just see which of two functions takes longer

→ padding so the slow function crosses one more clock edge than the fast one
Recovering resolution: Edge thresholding

- **nanosecond** resolution
- **Firefox/Tor**: 2 ns, **Edge**: 10 ns, **Chrome**: 15 ns
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 and edge thresholding inefficient (need to redo the measurements to get rid of noise)
• lowering timer resolution is not enough
• adding jitter → makes clock interpolation and edge thresholding inefficient (need to redo the measurements to get rid of noise)
→ has no impact on SharedArrayBuffers!

Thomas Rokicki et al. “Sok: In search of lost time: A review of javascript timers in browsers”. In: EuroS&P. 2021
Jitter?

- lowering timer resolution is not enough
- adding jitter → makes clock interpolation and edge thresholding 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!

---

Thomas Rokicki et al. “Sok: In search of lost time: A review of javascript timers in browsers”. In: EuroS&P. 2021
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
#1. No high-resolution timers

We just solved this problem :)
#2. No instruction to flush the cache

We already solved this problem earlier :)
#2. No instruction to flush the cache

We already solved this problem earlier :) 

Let’s use Prime+Probe!
#3. No knowledge about physical addresses

- OS optimization: use Transparent Huge Pages (THP, 2MB pages)
- = last 21 bits (2MB) of physical address
- = last 21 bits (2MB) of virtual address
#3. No knowledge about physical addresses

- OS optimization: use Transparent Huge Pages (THP, 2MB pages)
  - = last 21 bits (2MB) of physical address
  - = last 21 bits (2MB) of virtual address

→ which JS array indices?
#3. Obtaining the beginning of a THP

- physical pages for these THPs are mapped on-demand
  → **page fault** when an allocated THP is accessed for the first time
#3. Choosing physical addresses

- we now know the last 21 bits of physical addresses
  → enough to get cache set indexes
  → enough to get DRAM information for some systems, e.g., Sandy Bridge with DDR3
Eviction sets in JavaScript

→ 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
- covert channel cross-VM
- website fingerprinting

Port contention attacks in browsers
Port contention attacks: Challenges with JavaScript

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

- 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

Sender

- Virtual machine
- JS sandbox

Receiver

**Cross-browser**

- User applications
- OS
- Hardware: CPU Ports

Sender

- JS sandbox

Receiver

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

80 bit/s bandwidth
Other micro-architectural attacks in browsers?
Other micro-architectural attacks in browsers

ASLR on the Line: Practical Cache Attacks on the MMU

Spectre Attacks: Exploiting Speculative Execution
Bonus: you don’t even need JavaScript!

Attack 5: CSS Prime+Probe

```
<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

Resolve non existing image

==

TIMER

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

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

Conclusions
Conclusions

- any shared component is a potential side-channel vector
Conclusions

• any shared component is a potential side-channel vector
• it’s really hard not to share a component
Conclusions

- 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

• 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!
Micro-architectural attacks: from CPU to browser

Clémentine Maurice, CNRS
@BloodyTangerine
July 7 2022—Summer School “Cyber in Nancy”, Nancy, France


Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. “SMoTherSpectre: Exploiting Speculative Execution through Port Contention”. In: CCS. 2019.


Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: S&P. 2015.

Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. “Reverse Engineering Intel Complex Addressing Using Performance Counters”. In: RAID. 2015.


These slides have been designed using resources from Flaticon.com