From bottom to top: 
Exploiting hardware side channels in web browsers

Clémentine Maurice, Graz University of Technology
July 4, 2017—RMLL, Saint-Étienne, France
Clémentine Maurice
PhD since October 2015
from Rennes, France

now postdoc at TU Graz, Austria
Secure Systems group

+ Secure Systems team: Daniel Gruss, Michael Schwarz, Peter Pessl
safe software infrastructure does not mean safe execution
• safe software infrastructure does not mean safe execution
• information leaks because of the underlying hardware
• safe software infrastructure does not mean safe execution
• information leaks because of the underlying hardware
• these vulnerabilities can also be exploited at a high level
• safe software infrastructure does not mean safe execution
• information leaks because of the **underlying hardware**
• these vulnerabilities can also be exploited at a high level
• like a **web browser**
Introduction

• safe software infrastructure does not mean safe execution
• information leaks because of the underlying hardware
• these vulnerabilities can also be exploited at a high level
• like a web browser
• because JavaScript is nothing more than code executing on your machine :)
1. What are micro-architectural side channels?
1. What are micro-architectural side channels?
2. How can I use DRAM to create a covert channel?
Outline

1. What are micro-architectural side channels?
2. How can I use DRAM to create a covert channel?
3. How can I do that in JavaScript?!
Sources of leakage

- no “bug” in the sense of a mistake $\rightarrow$ lots of performance optimizations
Sources of leakage

- no “bug” in the sense of a mistake $\rightarrow$ lots of performance optimizations
- via power consumption, electromagnetic leaks
Sources of leakage

- No “bug” in the sense of a mistake → lots of performance optimizations
- Via power consumption, electromagnetic leaks → targeted attacks, physical access
- Via shared hardware and microarchitecture → remote attacks

Controller: Rikomagic MK802 IV
Loop antenna
MicroSD card
Antenna tuning capacitor
SDR receiver: FUNcube Dongle Pro+
Power: 4xAA batteries
WiFi antenna
Pita bread
Sources of leakage

- no “bug” in the sense of a mistake → lots of performance optimizations
- via power consumption, electromagnetic leaks
  → targeted attacks, physical access
Sources of leakage

• no “bug” in the sense of a mistake → lots of performance optimizations
• via power consumption, electromagnetic leaks
  → targeted attacks, physical access
• via shared hardware and microarchitecture
Sources of leakage

• no “bug” in the sense of a mistake → lots of performance optimizations
• via power consumption, electromagnetic leaks
  → targeted attacks, physical access
• via shared hardware and microarchitecture
  → remote attacks
Shared hardware

- memory
  - DRAM
  - memory bus
- CPU
  - branch prediction unit
  - arithmetic logic unit
  - data and instruction cache
DRAM and side channels
DRAM organization
DRAM organization

channel 0

channel 1
DRAM organization

channel 0

back of DIMM: rank 1

channel 1

front of DIMM: rank 0
DRAM organization

- Channel 0:
  - Back of DIMM: rank 1
  - Front of DIMM: rank 0

- Channel 1:
  - Chip
DRAM organization

chip

bank 0

row 0
row 1
row 2
...
row 32767

row buffer
DRAM organization

bank 0

row 0
row 1
row 2
...
row 32767

row buffer

64k cells
1 capacitor, 1 transistor each
DRAM row buffer

- DRAM internally is only capable of reading entire rows
• DRAM internally is only capable of reading entire rows
• capacitors in cells discharge when you “read the bits”
• buffer the bits when reading them from the cells
• write the bits back to the cells when you’re done
DRAM row buffer

- DRAM internally is only capable of reading entire rows
- Capacitors in cells discharge when you “read the bits”
- Buffer the bits when reading them from the cells
- Write the bits back to the cells when you’re done

→ row buffer
How reading from DRAM works

DRAM bank

CPU wants to access row 1

row buffer
How reading from DRAM works

CPU wants to access row 1
→ row 1 activated
How reading from DRAM works

CPU wants to access row 1
→ row 1 activated
→ row 1 copied to row buffer
How reading from DRAM works

DRAM bank

11111111111111111
11111111111111111
11111111111111111
11111111111111111
...
11111111111111111

row buffer

CPU wants to access row 1
→ row 1 activated
→ row 1 copied to row buffer

return
How reading from DRAM works

CPU wants to access row 2

row buffer
How reading from DRAM works

CPU wants to access row 2
→ row 2 activated
How reading from DRAM works

CPU wants to access row 2
→ row 2 activated
→ row 2 copied to row buffer
How reading from DRAM works

CPU wants to access row 2
→ row 2 activated
→ row 2 copied to row buffer
How reading from DRAM works

CPU wants to access row 2
→ row 2 activated
→ row 2 copied to row buffer
→ slow (row conflict)
How reading from DRAM works

DRAM bank

11111111111111
11111111111111
11111111111111
11111111111111
...
11111111111111

row buffer

CPU wants to access row 2—again
How reading from DRAM works

**DRAM bank**

```
11111111111111
11111111111111
11111111111111
11111111111111
...         
11111111111111
```

**CPU wants to access row 2—again**

→ **row 2 already in row buffer**
How reading from DRAM works

DRAM bank

CPU wants to access row 2—again
→ row 2 already in row buffer

return
How reading from DRAM works

DRAM bank

CPU wants to access row 2—again
→ row 2 already in row buffer
→ fast (row hit)
How reading from DRAM works

DRAM bank

row buffer = cache
DRAM timing differences

Access time [CPU cycles]

Number of cases

- Cache hit
- Cache miss, row hit
- Cache miss, row conflict
DRAM side channels?

- row buffers are caches
• row buffers are **caches**
• we can observe timing differences
DRAM side channels?

- row buffers are *caches*
- we can observe timing differences
- how to *exploit* these timing differences?

---

• row buffers are caches
• we can observe timing differences
• how to exploit these timing differences?
• target addresses in the same channel, rank and bank
• row buffers are *caches*
• we can observe timing differences
• how to *exploit* these timing differences?
• *target addresses* in the same channel, rank and bank
• but DRAM mapping functions are *undocumented*
DRAM side channels?

- row buffers are **caches**
- we can observe timing differences
- how to **exploit** these timing differences?
- **target addresses** in the same channel, rank and bank
- but DRAM mapping functions are **undocumented**
  → we reverse-engineered them! 🔄 [https://github.com/IAIK/drama](https://github.com/IAIK/drama)

---

• infer behavior from memory accesses similarly to cache attacks
• infer behavior from memory accesses similarly to cache attacks
• works across VMs, across cores, across CPUs
DRAMA: DRAM Addressing attacks

- infer behavior from memory accesses similarly to cache attacks
- works across VMs, across cores, across CPUs
- covert channels and side-channel attacks
DRAMA: DRAM Addressing attacks

- infer behavior from memory accesses similarly to cache attacks
- works across VMs, across cores, **across CPUs**
- covert channels and side-channel attacks
- covert channel: two processes ***communicating*** with each other
  - not allowed to do so, e.g., across VMs
DRAMA: DRAM Addressing attacks

- infer behavior from memory accesses similarly to cache attacks
- works across VMs, across cores, across CPUs
- covert channels and side-channel attacks
  - 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., spies on keystrokes
sender and receiver agree on one bank
receiver continuously accesses a row $i$
sender and receiver agree on one bank receiver continuously accesses a row $i$
sender and receiver agree on one bank receiver continuously accesses a row $i$

**case #1: sender transmits 1**
DRAMA covert channel

sender and receiver agree on one bank
receiver continuously accesses a row $i$

case #1: sender transmits 1
sender accesses row $j \neq i$
DRAMA covert channel

sender and receiver agree on one bank
receiver continuously accesses a row $i$

case #1: sender transmits 1
sender accesses row $j \neq i$
DRAMA covert channel

sender and receiver agree on one bank
receiver continuously accesses a row $i$

**case #1: sender transmits 1**

sender accesses row $j \neq i$
next receiver access $\rightarrow$ copy row buffer
sender and receiver agree on one bank
receiver continuously accesses a row \( i \)

**case #1: sender transmits 1**

sender accesses row \( j \neq i \)
next receiver access \( \rightarrow \) copy row buffer
\( \rightarrow \text{slow} \)
DRAMA covert channel

sender and receiver agree on one bank
receiver continuously accesses a row $i$

case #2: sender transmits 0
**DRAMA covert channel**

- **sender** and **receiver** agree on **one bank**
- **receiver** continuously accesses a row \( i \)

| 00000000 | 00000000 |
| 00000000 | 00000000 |
| 00000000 | 00000000 |
| 00000000 | 00000000 |
| 00000000 | 00000000 |

**case #2: sender transmits 0**
- **sender** does nothing

row buffer
DRAMA covert channel

sender and receiver agree on one bank receiver continuously accesses a row $i$

case #2: sender transmits 0

sender does nothing

next receiver access $\rightarrow$ already in buffer
**DRAMA covert channel**

sender and receiver agree on one bank receiver continuously accesses a row $i$

**case #2: sender transmits 0**

sender does nothing next receiver access $\rightarrow$ already in buffer $\rightarrow$ fast
Two applications can covertly communicate with each other
But can we use that for spying?
DRAMA side-channel attacks

spy and victim share a row $i$
DRAMA side-channel attacks

spy and victim share a row $i$

**case #1**
spy accesses row $j \neq i$, copy to row buffer
DRAMA side-channel attacks

spy and victim share a row $i$

**case #1**
spy accesses row $j \neq i$, copy to row buffer

**victim accesses** row $i$, copy to row buffer
DRAMA side-channel attacks

spy and victim share a row $i$

**case #1**

spy accesses row $j \neq i$, copy to row buffer

**victim accesses** row $i$, copy to row buffer

spy accesses row $i$, no copy
DRAMA side-channel attacks

spy and victim share a row $i$

**case #1**

spy accesses row $j \neq i$, copy to row buffer

**victim accesses** row $i$, copy to row buffer

spy accesses row $i$, no copy

→ **fast**
DRAMA side-channel attacks

spy and victim share a row $i$

**case #2**
spy accesses row $j \neq i$, copy to row buffer
DRAMA side-channel attacks

spy and victim share a row $i$

**case #2**
spy accesses row $j \neq i$, copy to row buffer
no victim access on row $i$
DRAMA side-channel attacks

spy and victim share a row \( i \)

**case #2**
spy accesses row \( j \neq i \), copy to row buffer

**no victim access** on row \( i \)
spy accesses row \( i \), copy to row buffer
DRAMA side-channel attacks

spy and victim share a row \( i \)

**case #2**

- spy accesses row \( j \neq i \), copy to row buffer
- **no victim access** on row \( i \)
- spy accesses row \( i \), copy to row buffer
  
  \[ \rightarrow \text{slow} \]
Spying on keystrokes on the Firefox URL bar

- side-channel: template attack
  - allocate a large fraction of memory to be in a row with the victim
  - profile memory and record row-hit ratio for each address
I’m sure we’ll need to write a lot of C code
At least we’re safe with JavaScript!
Member Rowhammer.js?
DRAM covert channels in JavaScript?
Why JavaScript?

• JavaScript is code executed in a sandbox
• JavaScript is code executed in a sandbox
• can’t do anything nasty since it is in a sandbox, right?
Why 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
Why 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**
  1. accessing their own memory
Why 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
  1. accessing their own memory
  2. measuring time
Challenges with JavaScript

1. No knowledge about physical addresses
2. No instruction to flush the cache
3. No high-resolution timers
#1. 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
#1. 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?
#1. 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

---

#1. Choosing physical addresses

- we now know the last 21 bits of physical addresses
- enough for most systems, e.g., Sandy Bridge with DDR3
#2. No instruction to flush the cache

- measure DRAM timing
- only non-cached accesses reach DRAM
- no clflush instruction
  → evict data with other memory accesses
#2. Bypassing the CPU cache: Basic idea

- evicting cache line only using memory accesses

---

#2. Bypassing the CPU cache: Basic idea

- evicting **cache line** only using **memory accesses**

---

#2. Bypassing the CPU cache: Basic idea

- evicting **cache line** only using **memory accesses**

---

• evicting cache line only using memory accesses

---

#2. Bypassing the CPU cache: Basic idea

- evicting **cache line** only using **memory accesses**

---

#2. Bypassing the CPU cache: Basic idea

- evicting cache line only using memory accesses

#2. Bypassing the CPU cache: Basic idea

- evicting cache line only using memory accesses

---

#2. Bypassing the CPU cache: Basic idea

- evicting cache line only using memory accesses

---

#2. Bypassing the CPU cache: Basic idea

- evicting **cache line** only using **memory accesses**

---

#2. Bypassing the CPU cache: Basic idea

- evicting cache line only using memory accesses

- it’s a bit more complicated than that: replacement policy is not LRU

---

#2. Bypassing the CPU cache: Basic idea

- evicting **cache line** only using **memory accesses**

- it’s a bit more complicated than that: **replacement policy** is not LRU
- but we already solved this problem before :)

---

• measure small timing differences: need a high-resolution timer
• measure small timing differences: need a high-resolution timer
• native: rdtsc, timestamp in CPU cycles
#3. 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
• measure small timing differences: need a **high-resolution timer**
• native: `rdtsc`, timestamp in CPU cycles
• JavaScript: `performance.now()` has the highest resolution

```javascript
performance.now()
```

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

— Mozilla Developer Network
High-resolution timers in JavaScript
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

Microsecond precision?

Firefox < 41 (1 ns) | $1 \cdot 10^{-3}$
Microsecond precision?

Edge 38 (1 μs) 1

Firefox < 41 (1 ns) $1 \cdot 10^{-3}$

Microsecond precision?

W3C standard (5 μs) 5

Edge 38 (1 μs) 1

Firefox < 41 (1 ns) 1 \cdot 10^{-3}
Microsecond precision?

Firefox $\geq 41$/Chrome/Safari (5 µs) [5]

W3C standard (5 µs) [5]

Edge 38 (1 µs) [1]

Firefox < 41 (1 ns) $1 \cdot 10^{-3}$
<table>
<thead>
<tr>
<th>Browser</th>
<th>Precision (in MS)</th>
<th>Microseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tor (100 ms)</td>
<td></td>
<td>1 \cdot 10^5</td>
</tr>
<tr>
<td>Firefox \geq 41/Chrome/Safari (5 \mu s)</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>W3C standard (5 \mu s)</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Edge 38 (1 \mu s)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Firefox &lt; 41 (1 ns)</td>
<td>1 \cdot 10^{-3}</td>
<td></td>
</tr>
</tbody>
</table>
Microsecond precision?

<table>
<thead>
<tr>
<th>Browser</th>
<th>Precision (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuzzyfox (100 ms)</td>
<td>$1 \cdot 10^5$</td>
</tr>
<tr>
<td>Tor (100 ms)</td>
<td>$1 \cdot 10^5$</td>
</tr>
<tr>
<td>Firefox ≥ 41/Chrome/Safari (5 μs)</td>
<td>5</td>
</tr>
<tr>
<td>W3C standard (5 μs)</td>
<td>5</td>
</tr>
<tr>
<td>Edge 38 (1 μs)</td>
<td>1</td>
</tr>
<tr>
<td>Firefox &lt; 41 (1 ns)</td>
<td>$1 \cdot 10^{-3}$</td>
</tr>
</tbody>
</table>

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

---

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

Firefox/Chrome: five.osf/zero.osf/zero.osf ns, Tor: one.osf/five.osf µs/three.osf/one.osf
Recovering resolution: Clock interpolation

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

![Diagram showing clock tick intervals](image)
• measure how often we can increment a variable between two timer ticks

Firefox/Chrome: +five.osf/+zero.osf/+zero.osf ns, Tor: +one.osf/+five.osf +µs/three.osf/+one.osf
Recovering resolution: Clock interpolation

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

- Firefox/Chrome: \( \text{five.osf, zero.osf, zero.osf} \) ns, Tor: \( \text{one.osf, five.osf} \) \( \mu \text{s} \)

• 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

- Firefox/Chrome: /five.osf/zero.osf/zero.osf ns, Tor: /one.osf/five.osf

• to measure with high resolution
  - start measurement at **clock edge**
  - **increment** a variable until next clock edge
Recovering resolution: Clock interpolation

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

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

• Firefox/Chrome: 500 ns, Tor: 15 µs
• 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

- Firefox/Tor: /two.osf ns, Edge: /one.osf/zero.osf ns, Chrome: /one.osf/five.osf ns

- Percentage of correct classifications:
  - **unaligned**: 87% correct, 13% slow misclassified, 0% fast misclassified
  - **aligned**: 100% correct, 0% slow misclassified, 0% fast misclassified
  - **padded**: 82% correct, 18% slow misclassified, 0% fast misclassified
Recovering resolution: Edge thresholding

- nanosecond resolution
Recovering resolution: Edge thresholding

- Both correct
- $f_{\text{slow}}$ misclassified
- $f_{\text{fast}}$ misclassified

- Nanosecond resolution
- Firefox/Tor: 2 ns, Edge: 10 ns, Chrome: 15 ns
Building a timer

• goal: counter that does not block main thread
Building a timer

• goal: counter that does not block main thread
• baseline `setTimeout`: 4 ms (except Edge: 2 ms)
Building a timer

- goal: counter that does not block main thread
- baseline `setTimeout`: 4 ms (except Edge: 2 ms)
- CSS animation → increase width of element as fast as possible
Building a timer

- goal: counter that does not block main thread
- baseline `setTimeout`: 4 ms (except Edge: 2 ms)
- CSS animation $\rightarrow$ increase width of element as fast as possible
- timestamp = width of element
Building a timer

- goal: counter that **does not block main thread**
- baseline `setTimeout`: 4 ms (except Edge: 2 ms)
- **CSS animation** → increase width of element as fast as possible
- timestamp = width of element
- but animation limited to 60 fps → 16 ms resolution
• JavaScript can spawn new threads called web worker
Building a timer: Web worker

- JavaScript can spawn **new threads** called web worker
- web worker communicate using **message passing**
Building a timer: Web worker

- JavaScript can spawn **new threads** called web worker
- web worker communicate using **message passing**
- let **worker count** and request timestamp in main thread
Building a timer: Web worker

- JavaScript can spawn new threads called web worker
- Web worker communicate using message passing
- Let worker count and request timestamp in main thread
- Possibilities: postMessage, MessageChannel or BroadcastChannel
Building a timer: Web worker

- JavaScript can spawn new threads called web worker
- web worker communicate using message passing
- let worker count and request timestamp in main thread
- possibilities: postMessage, MessageChannel or BroadcastChannel
- microsecond resolution (even on Tor and Fuzzyfox)
Building a timer: Web worker

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

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

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

- **experimental** 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

- **experimental** 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
Building a timer: Is it good enough?

Access time [SharedArrayBuffer increments]

We can distinguish cache hits from cache misses (only \( \approx \) one cycle difference)!
→ we can distinguish cache hits from cache misses (only ≈ 150 cycles difference)!
ONE DOES NOT SIMPLY
GET RID OF CLOCKS
Bonus: What else can we do with this?

- idea is not new: Wray (1992)
- we also exploited it in other contexts
  - on ARM
  - inside an SGX enclave

---

DRAM covert channels in JavaScript!
Setup

- sender: native application in a VM
- receiver: JavaScript in a web page on the host
- sender and receiver select the same bank
- sender and receiver select a different row inside this bank
- sender transmits /zero.osf by doing nothing and /one.osf by causing row conflict
- receiver measures access time for its row: fast → /zero.osf, slow → /one.osf
Setup

- sender: native application in a VM
- receiver: JavaScript in a web page on the host
Setup

- sender: native application in a VM
- receiver: JavaScript in a web page on the host
- sender and receiver select the same bank
  - sender transmits /zero.osf by doing nothing and /one.osf by causing row conflict
  - receiver measures access time for its row: fast → /zero.osf, slow → /one.osf
Setup

- sender: native application in a VM
- receiver: JavaScript in a web page on the host
- sender and receiver select the same bank
- sender and receiver select a different row inside this bank
• sender: native application in a VM
• receiver: JavaScript in a web page on the host
• sender and receiver select the same bank
• sender and receiver select a different row inside this bank
• sender transmits 0 by doing nothing and 1 by causing row conflict
sender: native application in a VM
receiver: JavaScript in a web page on the host
sender and receiver select the same bank
sender and receiver select a different row inside this bank
sender transmits 0 by doing nothing and 1 by causing row conflict
receiver measures access time for its row: fast $\rightarrow$ 0, slow $\rightarrow$ 1
• communication based on 11-bit packets, with 5-bit of data
Sending packets

- Communication based on 11-bit packets, with 5-bit of data.
- Packet starts with a 2-bit preamble.

Diagram:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td>EDC</td>
</tr>
</tbody>
</table>

**Sequence bit** indicates whether it is a retransmission or a new packet.
Sending packets

- Communication based on 11-bit packets, with 5-bit of data
- Packet starts with a 2-bit preamble
- Data integrity checked by an error-detection code
Sending packets

- communication based on 11-bit packets, with 5-bit of data
- packet starts with a 2-bit preamble
- data integrity checked by an error-detection code
- sequence bit indicates whether it is a retransmission or a new packet
• transmission of approximately 11 bits/s
Evaluation

- transmission of approximately 11 bits/s
- can be improved using
• transmission of approximately 11 bits/s
• can be improved using
  • fewer re-transmits
• transmission of approximately 11 bits/s
• can be improved using
  • fewer re-transmits
  • error correction
Evaluation

- transmission of approximately 11 bits/s
- can be improved using
  - fewer re-transmits
  - error correction
  - multithreading $\rightarrow$ multiple banks in parallel
Evaluation

- transmission of approximately $11 \text{ bits/s}$
- can be improved using
  - fewer re-transmits
  - error correction
  - multithreading $\rightarrow$ multiple banks in parallel
- native code: 596 kbit/s cross CPU and cross VM
Conclusion
• information leaks because of the underlying hardware
Conclusion

• information leaks because of the underlying hardware
• vulnerabilities exploitable at the browser level
Conclusion

- information leaks because of the underlying hardware
- vulnerabilities exploitable at the browser level
- running arbitrary JavaScript allows building high-resolution timers
Conclusion

- information leaks because of the underlying hardware
- vulnerabilities exploitable at the browser level
- running arbitrary JavaScript allows building high-resolution timers
- hard to mitigate without reducing functionality
Thank you!

Contact

✉️ clementine@cmaurice.fr
🐦 @BloodyTangerine
From bottom to top:
Exploiting hardware side channels in web browsers

Clémentine Maurice, Graz University of Technology
July 4, 2017—RMLL, Saint-Étienne, France