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

Introduction

- safe software infrastructure does not mean safe execution

Introduction

- safe software infrastructure does not mean safe execution
- information leaks because of the underlying hardware

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

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

Introduction

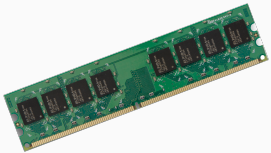
- safe software infrastructure does not mean safe execution
- information leaks because of the underlying hardware
- these vulnerabilities can also be exploited at a high level
- like a web browser
- because JavaScript is nothing more than code executing on your machine :)

Outline

1. What are micro-architectural side channels?

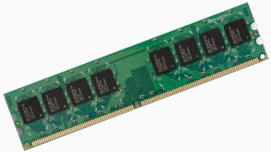
Outline

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?!



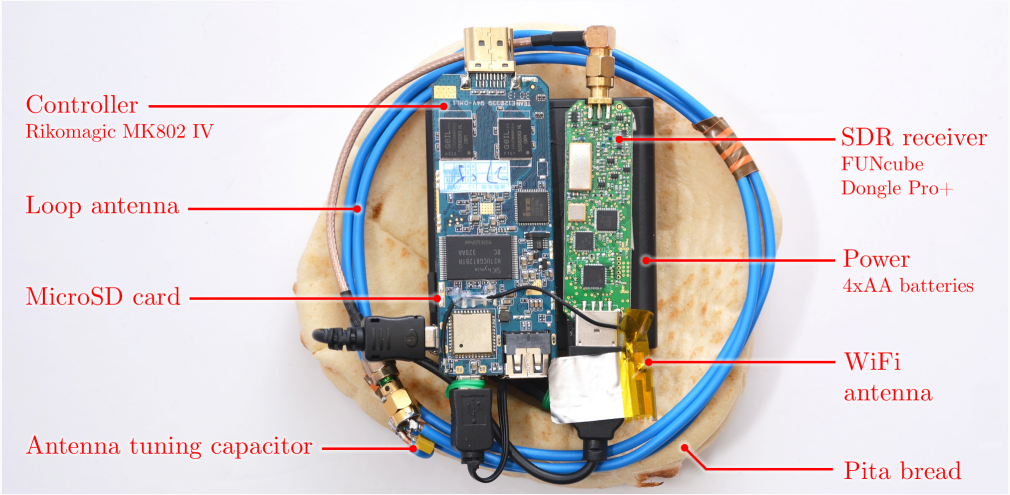
Sources of leakage

- 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



Sources of leakage

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

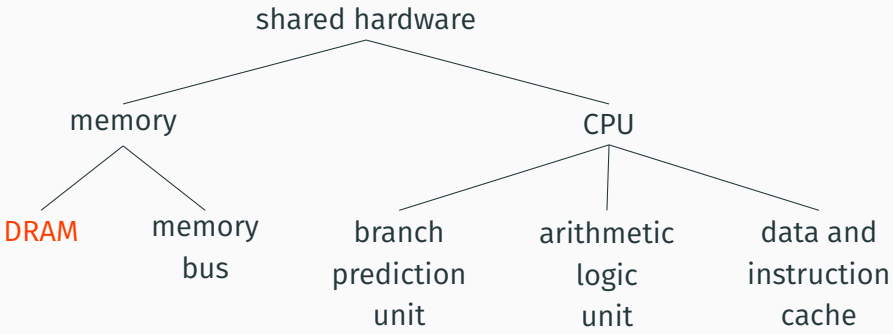
Sources of leakage

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

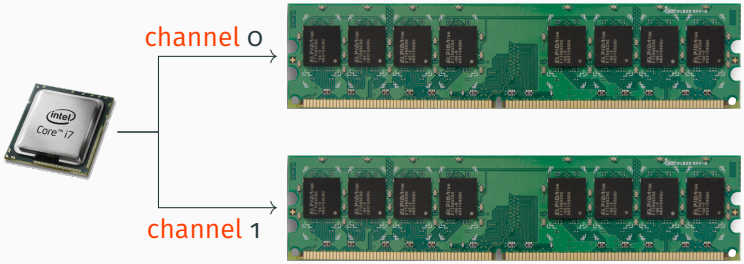
Shared hardware



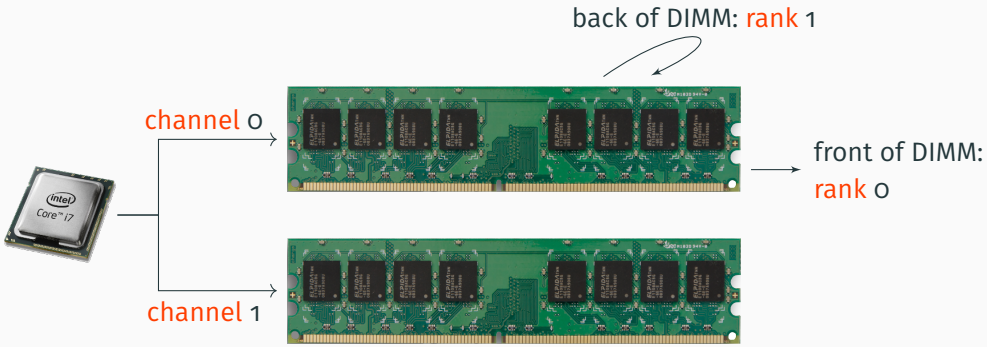
DRAM and side channels

DRAM organization

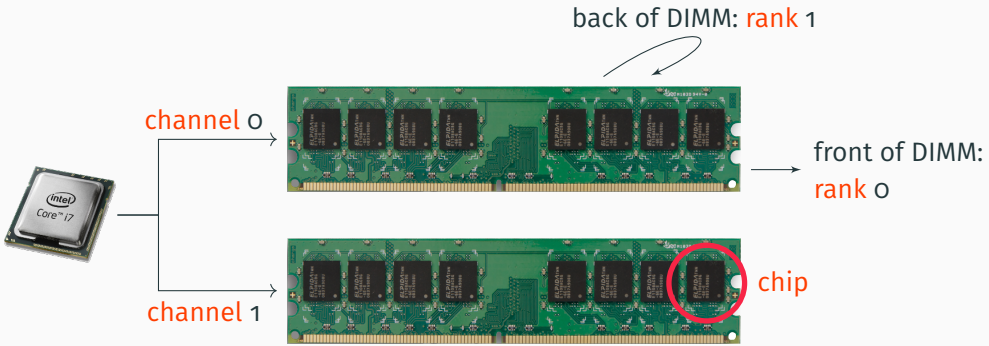
DRAM organization



DRAM organization

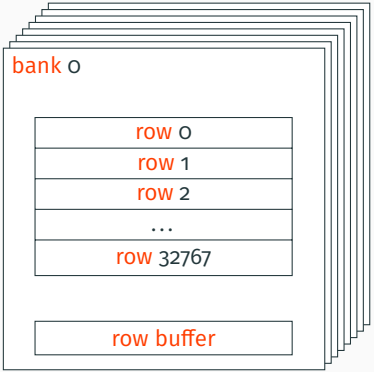


DRAM organization



DRAM organization

chip



DRAM organization

chip

bank 0

row 0
row 1
row 2
…
row 32767

row buffer

64k cells
1 capacitor,
1 transitor each

DRAM row buffer

- DRAM internally is only capable of reading entire rows

DRAM row buffer

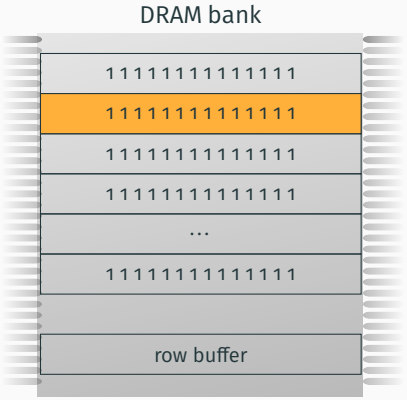
- DRAM internally is only capable of reading entire rows
- capacitors in cells discharge when you “read the bits”
- buffer the bits when reading them from the cells
- write the bits back to the cells when you’re done

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

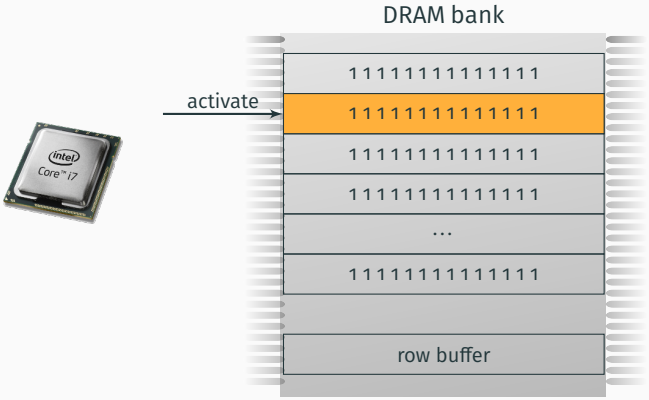
→ row buffer

How reading from DRAM works



CPU wants to access row 1

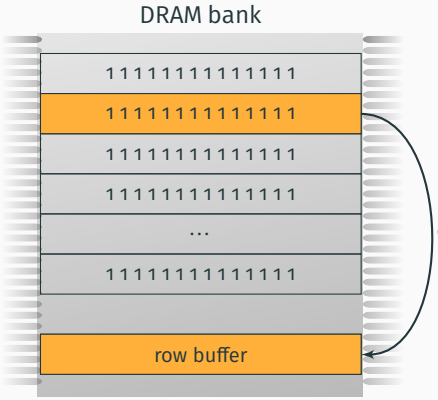
How reading from DRAM works



CPU wants to access row 1

$\rightarrow$ row 1 activated

How reading from DRAM works

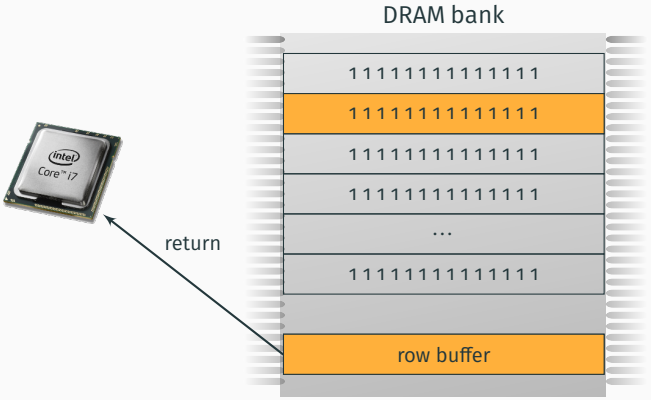


CPU wants to access row 1

→ row 1 activated

→ row 1 copied to row buffer

How reading from DRAM works

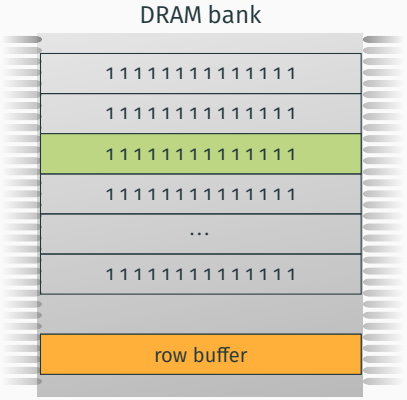


CPU wants to access row 1

→ row 1 activated

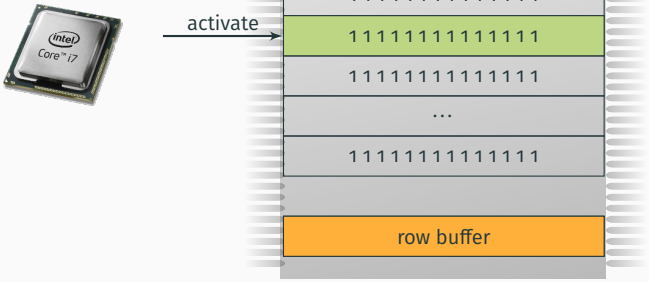
→ row 1 copied to row buffer

How reading from DRAM works



CPU wants to access row 2

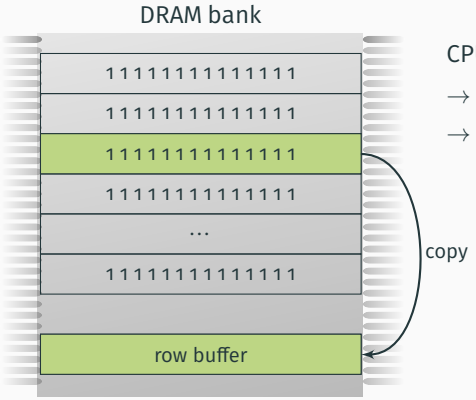
How reading from DRAM works



CPU wants to access row 2

$\rightarrow$ row 2 activated

How reading from DRAM works

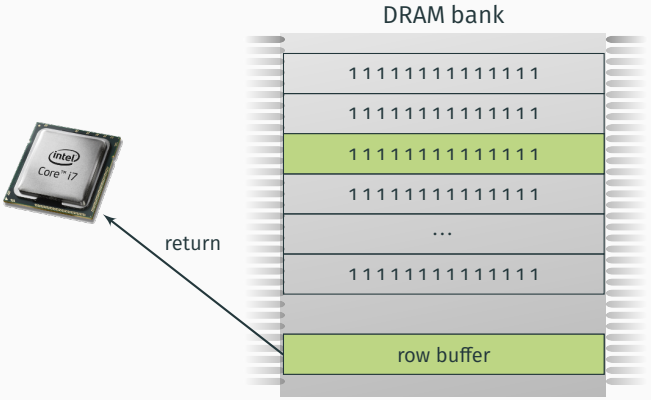


CPU wants to access row 2

$\rightarrow$ row 2 activated

$\rightarrow$ row 2 copied to row buffer

How reading from DRAM works

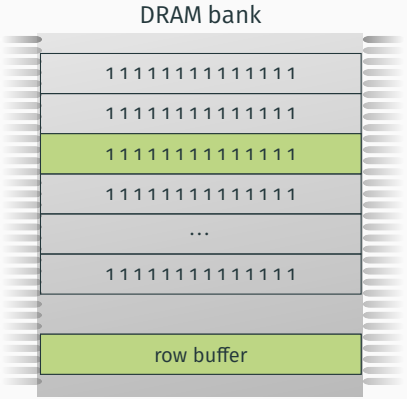


CPU wants to access row 2

→ row 2 activated

→ row 2 copied to row buffer

How reading from DRAM works



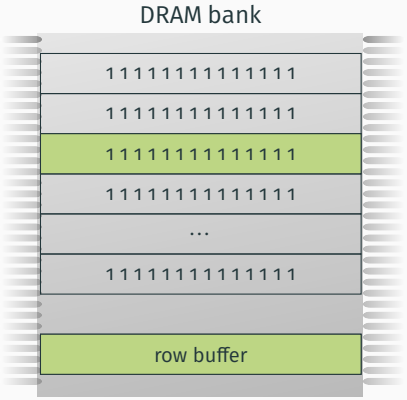
CPU wants to access row 2

→ row 2 activated

→ row 2 copied to row buffer

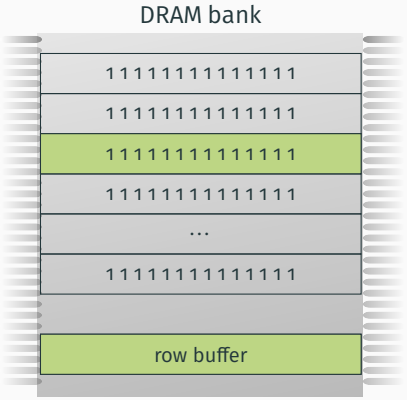
→ slow (row conflict)

How reading from DRAM works



CPU wants to access row 2—again

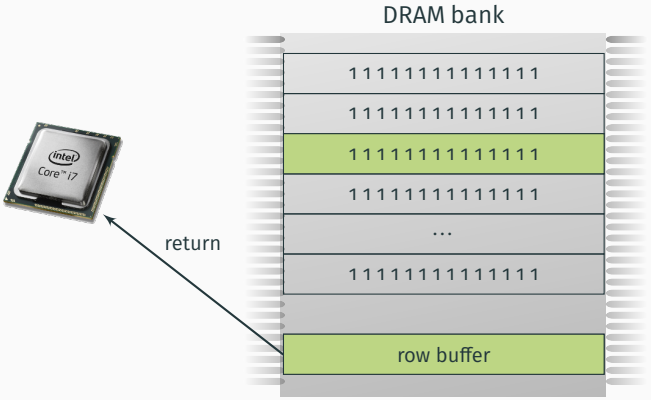
How reading from DRAM works



CPU wants to access row 2—again

$\rightarrow$ row 2 already in row buffer

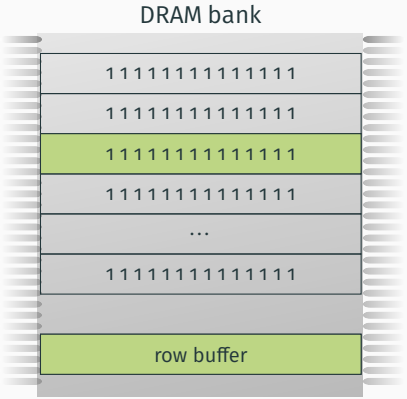
How reading from DRAM works



CPU wants to access row 2—again

→ row 2 already in row buffer

How reading from DRAM works

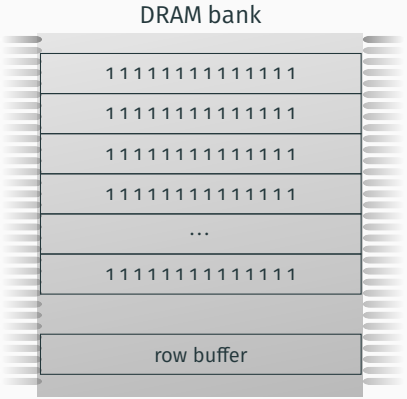


CPU wants to access row 2—again

→ row 2 already in row buffer

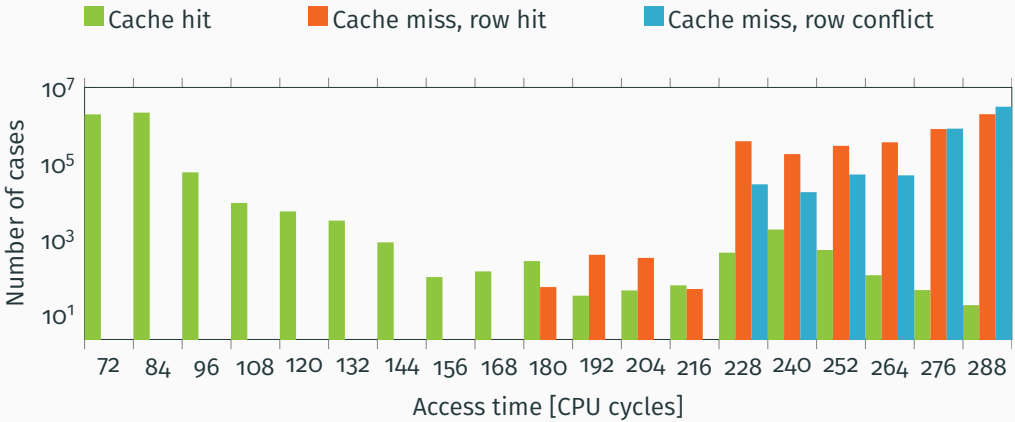
→ fast (row hit)

How reading from DRAM works

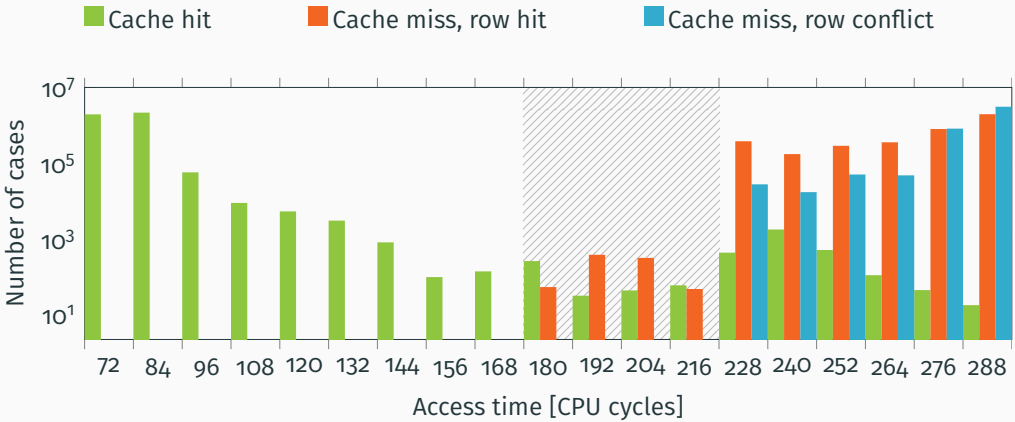


row buffer = cache

DRAM timing differences



DRAM timing differences



DRAM side channels?

- row buffers are caches

DRAM side channels?

- 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?

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

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

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`

P. Pessl et al. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *USENIX Security Symposium*. 2016

DRAMA: DRAM Addressing attacks

- infer behavior from memory accesses similarly to cache attacks

DRAMA: DRAM Addressing 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

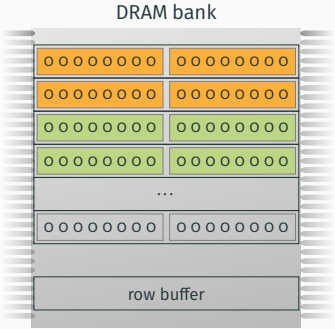
DRAMA: DRAM Addressing attacks

- 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

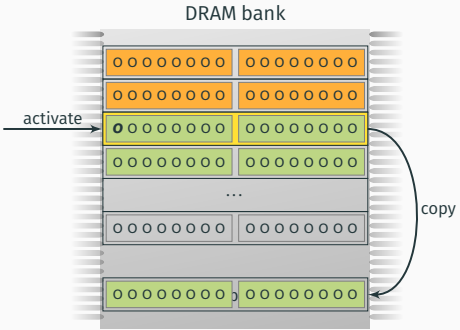
DRAMA covert channel



sender and receiver agree on one bank

receiver continuously accesses a row *i*

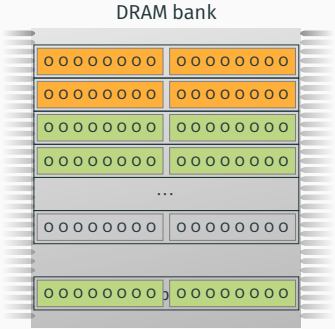
DRAMA covert channel



sender and receiver agree on one bank

receiver continuously accesses a row *i*

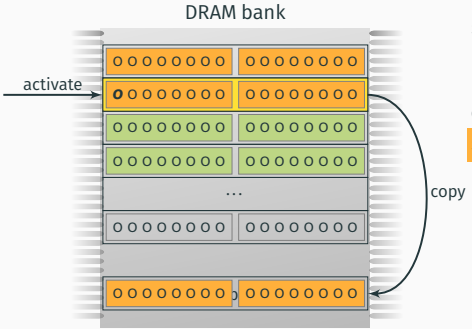
DRAMA covert channel



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

case #1: sender transmits 1

DRAMA covert channel



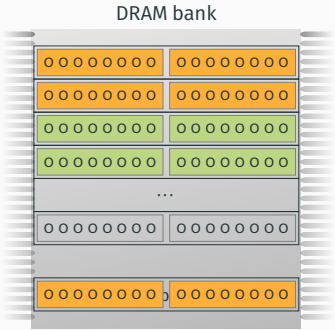
sender and receiver agree on one bank

receiver continuously accesses a row i

case #1: sender transmits 1

sender accesses row $j \neq i$

DRAMA covert channel



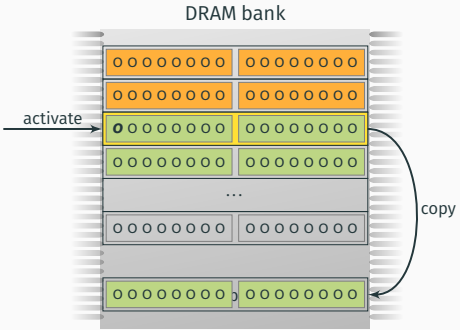
sender and receiver agree on one bank

receiver continuously accesses a row i

case #1: sender transmits 1

sender accesses row $j \neq i$

DRAMA covert channel



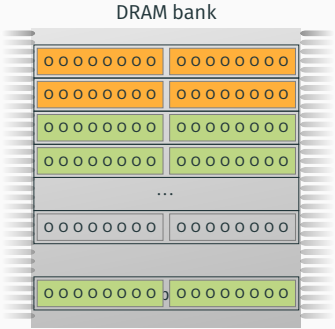
sender and receiver agree on one bank

receiver continuously accesses a row i

case #1: sender transmits 1

sender accesses row $j \neq i$
next receiver access $\rightarrow$ copy row buffer

DRAMA covert channel



sender and receiver agree on one bank

receiver continuously accesses a row i

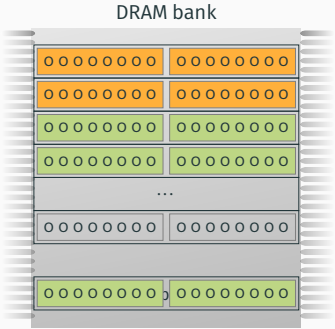
case #1: sender transmits 1

sender accesses row $j \neq i$

next receiver access $\rightarrow$ copy row buffer

$\rightarrow$ **slow**

DRAMA covert channel

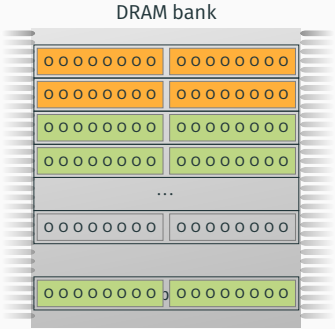


sender and receiver agree on one bank

receiver continuously accesses a row *i*

case #2: sender transmits o

DRAMA covert channel



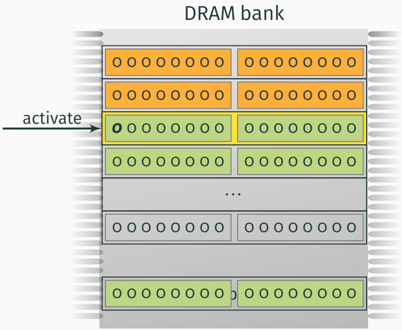
sender and receiver agree on one bank

receiver continuously accesses a row *i*

case #2: sender transmits 0

sender does nothing

DRAMA covert channel



sender and receiver agree on one bank

receiver continuously accesses a row *i*

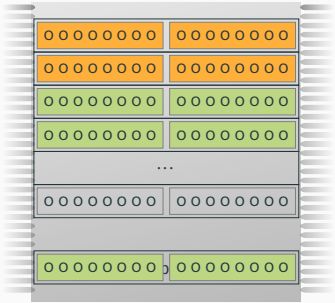
case #2: sender transmits 0

sender does nothing

next receiver access → already in buffer

DRAMA covert channel

DRAM bank



sender and receiver agree on one bank

receiver continuously accesses a row *i*

case #2: sender transmits o

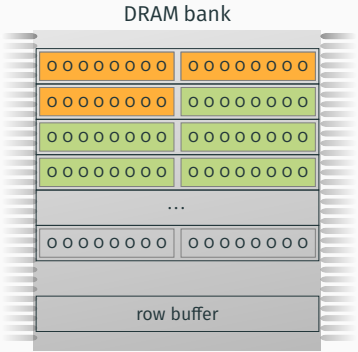
sender does nothing

next receiver access → already in buffer

→ **fast**

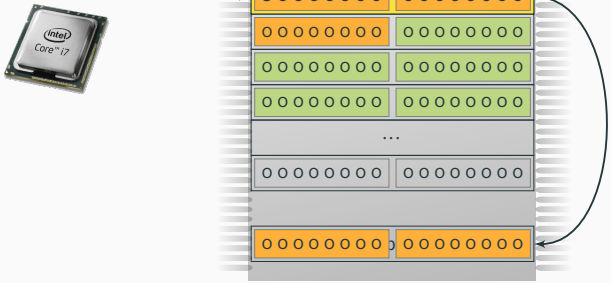
Two applications can covertly communicate with each other
But can we use that for spying?

DRAMA side-channel attacks



spy and victim share a row *i*

DRAMA side-channel attacks

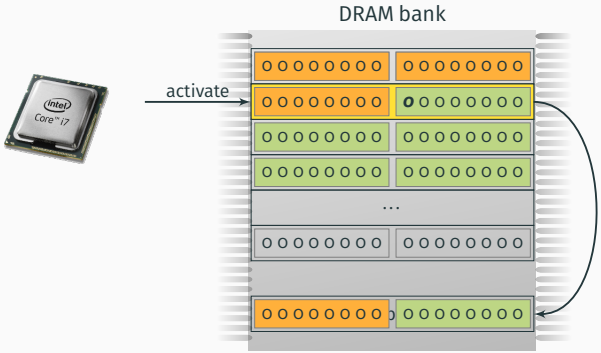


spy and victim share a row i

case #1

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

DRAMA side-channel attacks



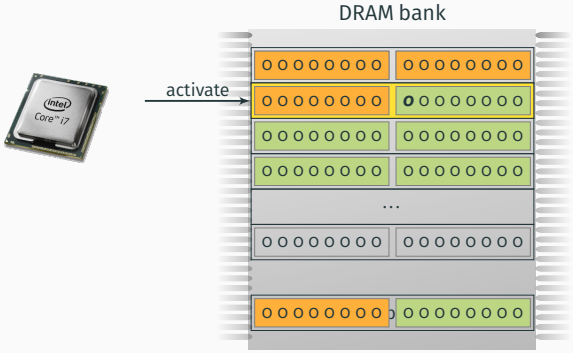
spy and victim share a row *i*

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

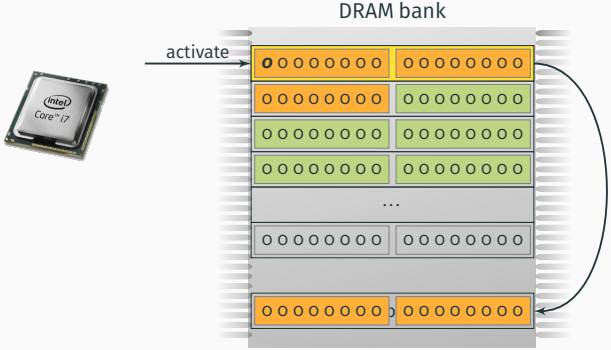
spy and victim share a row i

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

$\rightarrow$ **fast**

DRAMA side-channel attacks



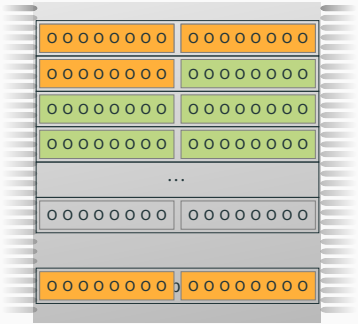
spy and victim share a row i

case #2

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

DRAMA side-channel attacks

DRAM bank



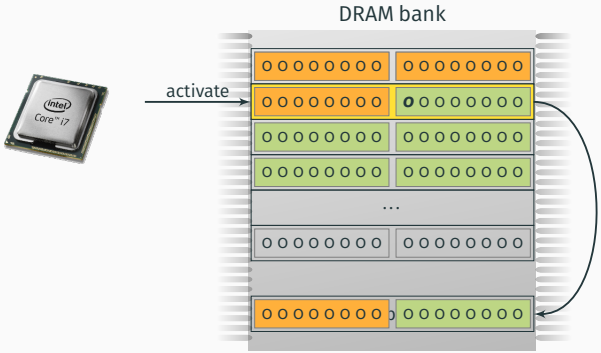
spy and victim share a row *i*

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

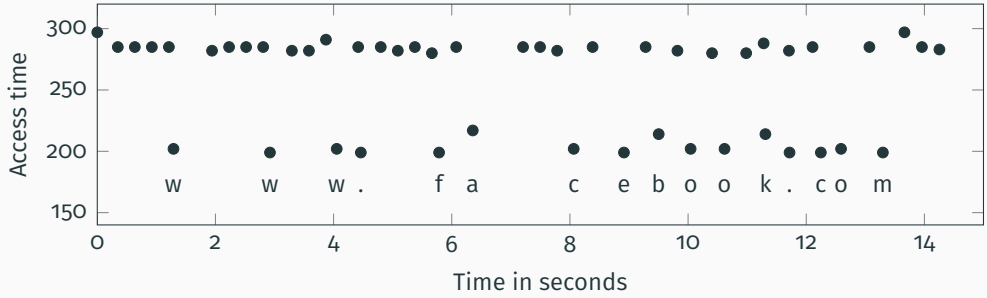
no victim access on row *i*

spy accesses row *i*, copy to row buffer

$\rightarrow$ **slow**

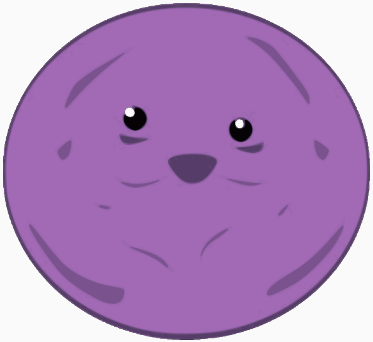
Spying on keystrokes on the Firefox URL bar

- 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

Why JavaScript?

- 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

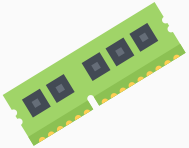
Why JavaScript?

- 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

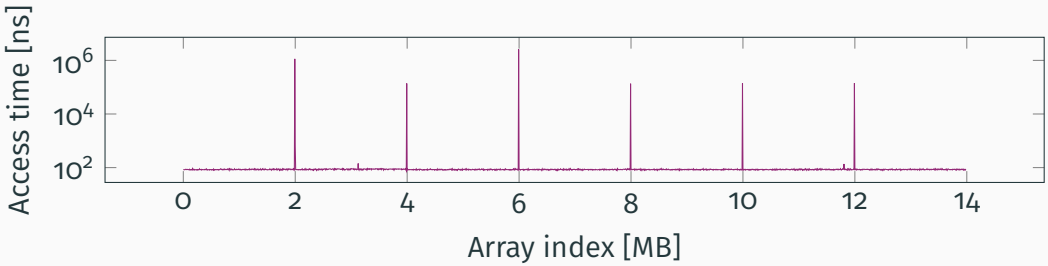
- OS optimization: use Transparent Huge Pages (THP, 2MB pages)
- last 21 bits (2MB) of physical address
- = last 21 bits (2MB) of virtual address

#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



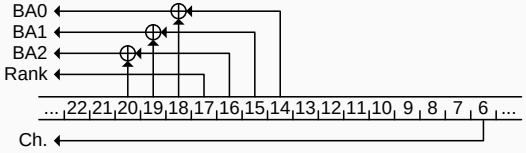
- physical pages for these THPs are mapped on-demand

→ page fault when an allocated THP is accessed for the first time

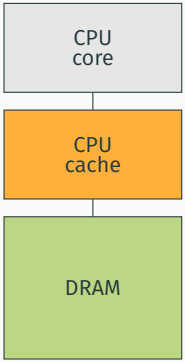
D. Gruss et al. “Practical Memory Deduplication Attacks in Sandboxed JavaScript”. In: *ESORICS’15*. 2015.

#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

$\rightarrow$ evict data with other memory accesses

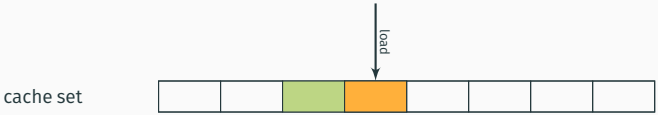
#2. Bypassing the CPU cache: Basic idea

- evicting cache line only using memory accesses

D. Gruss et al. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA’16*. 2016.

#2. Bypassing the CPU cache: Basic idea

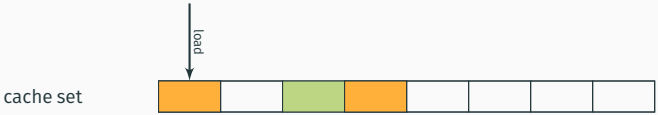
- evicting cache line only using memory accesses



D. Gruss et al. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA’16*. 2016.

#2. Bypassing the CPU cache: Basic idea

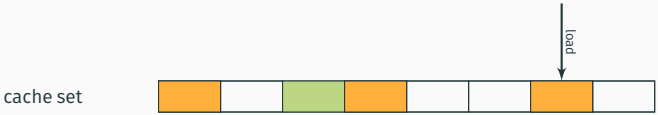
- evicting cache line only using memory accesses



D. Gruss et al. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA’16*. 2016.

#2. Bypassing the CPU cache: Basic idea

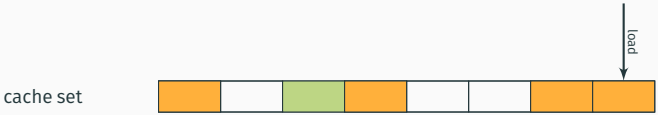
- evicting cache line only using memory accesses



D. Gruss et al. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA’16*. 2016.

#2. Bypassing the CPU cache: Basic idea

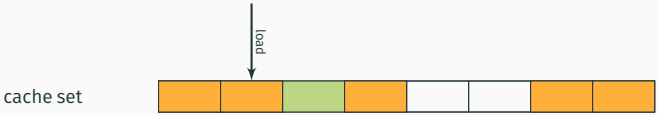
- evicting cache line only using memory accesses



D. Gruss et al. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA’16*. 2016.

#2. Bypassing the CPU cache: Basic idea

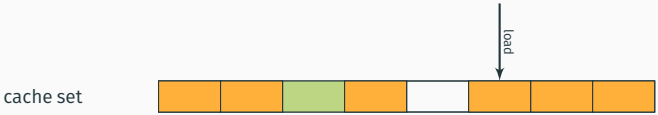
- evicting cache line only using memory accesses



D. Gruss et al. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA’16*. 2016.

#2. Bypassing the CPU cache: Basic idea

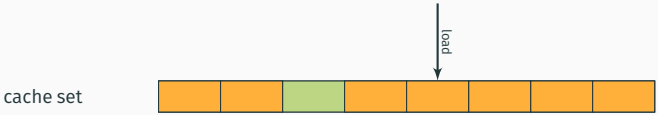
- evicting cache line only using memory accesses



D. Gruss et al. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA’16*. 2016.

#2. Bypassing the CPU cache: Basic idea

- evicting cache line only using memory accesses



D. Gruss et al. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA’16*. 2016.

#2. Bypassing the CPU cache: Basic idea

- evicting cache line only using memory accesses

D. Gruss et al. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA’16*. 2016.

#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

D. Gruss et al. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: *DIMVA'16*. 2016.

#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 :)

D. Gruss et al. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: *DIMVA'16*. 2016.

#3. High-resolution timers?

- measure small timing differences: need a high-resolution timer

#3. High-resolution timers?

- 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

#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

`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

Y. Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS’15*. 2015.

`https://www.mozilla.org/en-US/security/advisories/mfsa2015-114/`

It was better before

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

Y. Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS'15*. 2015.

`https://www.mozilla.org/en-US/security/advisories/mfsa2015-114/`

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

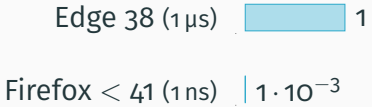
Y. Oren et al. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS’15*. 2015.

`https://www.mozilla.org/en-US/security/advisories/mfsa2015-114/`

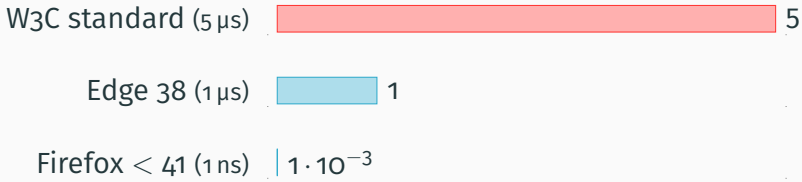
Microsecond precision?

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

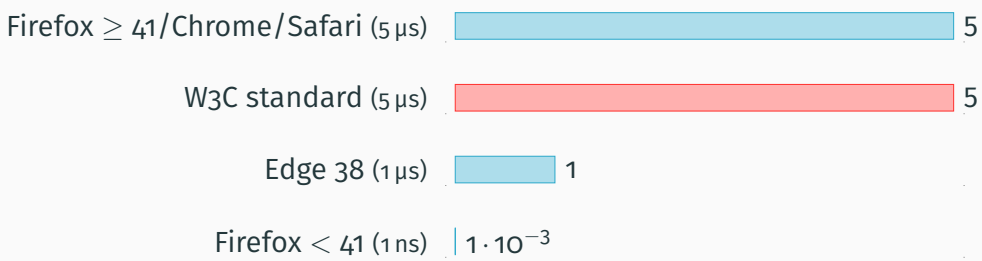
Microsecond precision?



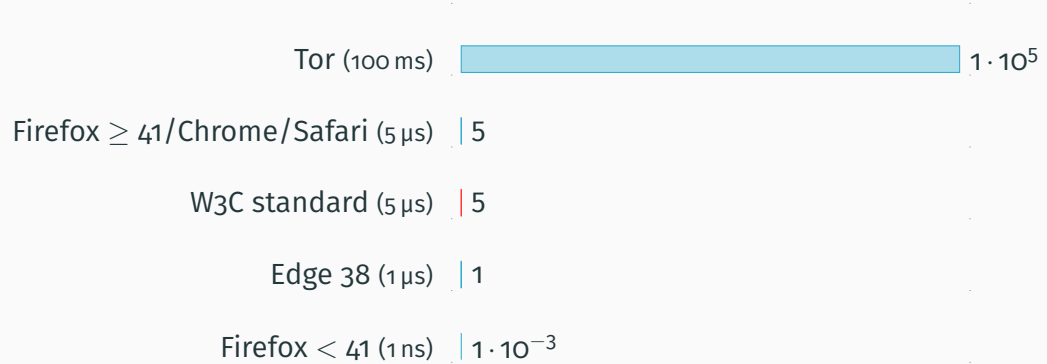
Microsecond precision?



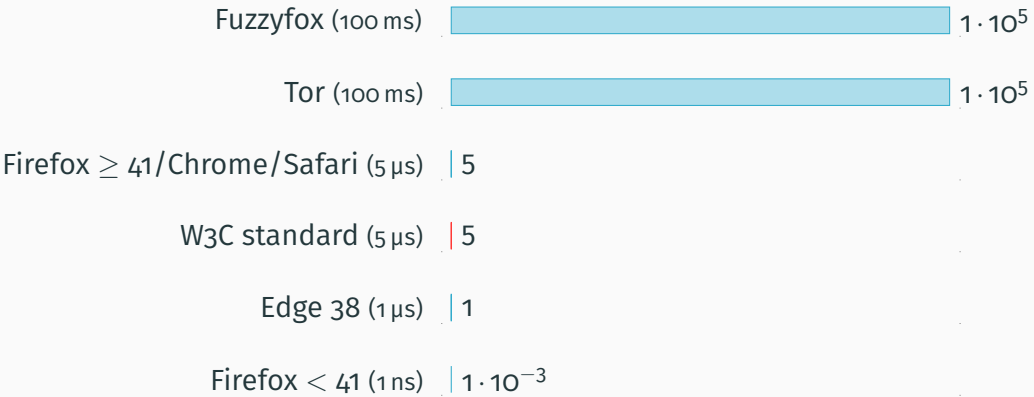
Microsecond precision?



Microsecond precision?



Microsecond precision?



D. Kohlbrenner et al. “Trusted Browsers for Uncertain Times”. In: *USENIX Security Symposium*. 2016

We can do better!

- microsecond resolution is not enough

M. Schwarz et al. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: *FC’17*. 2017.

We can do better!

- microsecond resolution is not enough
- two approaches

M. Schwarz et al. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: *FC’17*. 2017.

We can do better!

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

M. Schwarz et al. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: *FC’17*. 2017.

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

M. Schwarz et al. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: *FC’17*. 2017.

Recovering resolution: Clock interpolation

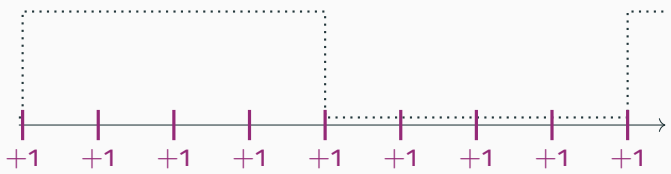
- measure how often we can increment a variable between two timer ticks

Recovering resolution: Clock interpolation

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

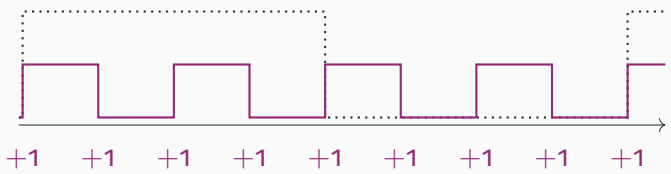
Recovering resolution: Clock interpolation

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



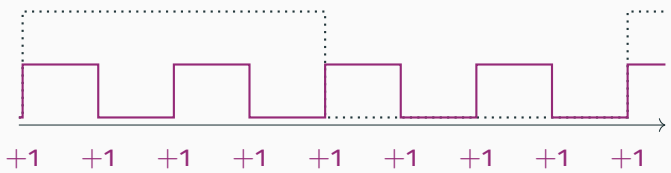
Recovering resolution: Clock interpolation

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



Recovering resolution: Clock interpolation

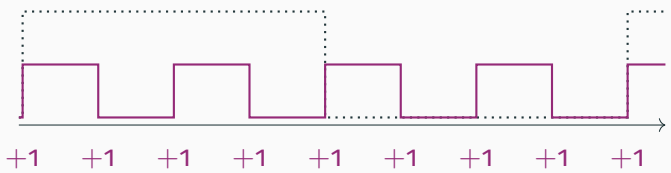
- measure how often we can increment a variable between two timer ticks



- to measure with high resolution

Recovering resolution: Clock interpolation

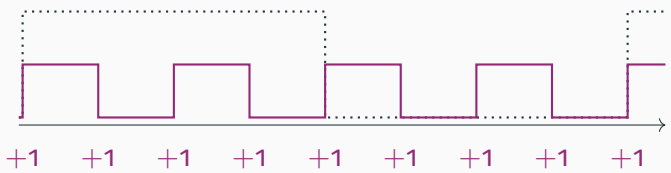
- measure how often we can increment a variable between two timer ticks



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

Recovering resolution: Clock interpolation

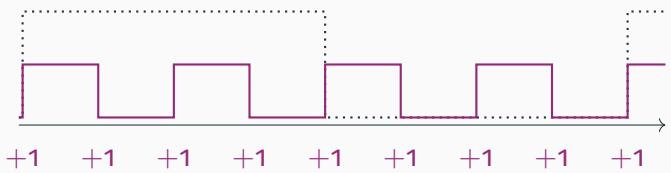
- measure how often we can increment a variable between two timer ticks



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

Recovering resolution: Clock interpolation

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



- to measure with high resolution
  - start measurement at clock edge
  - increment a variable until next clock edge
- Firefox/Chrome: 500 ns, Tor: 15 µs

Recovering resolution: Edge thresholding

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

Recovering resolution: Edge thresholding

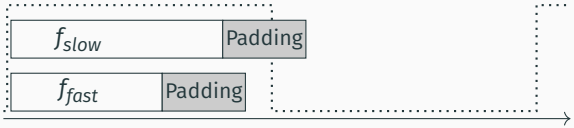
- often sufficient to just see which of two functions takes longer

Recovering resolution: Edge thresholding

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

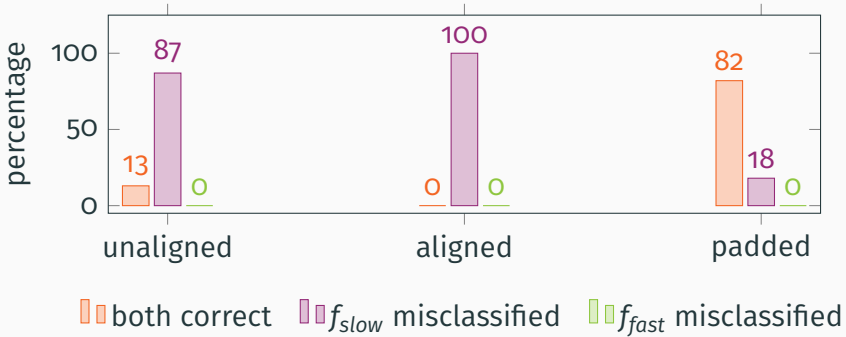
Recovering resolution: Edge thresholding

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

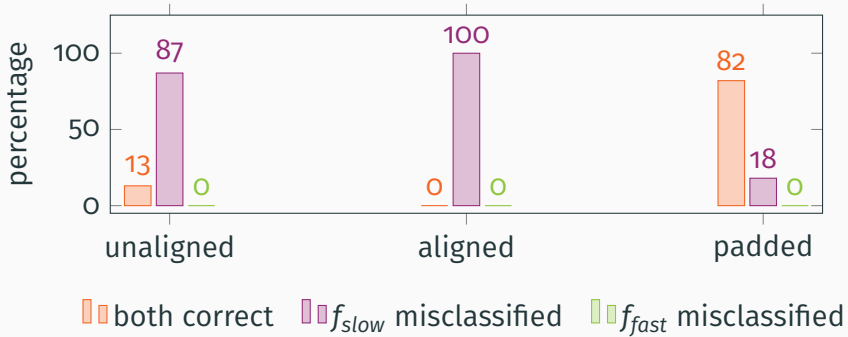


→ padding so the slow function crosses one more clock edge than the fast one

Recovering resolution: Edge thresholding

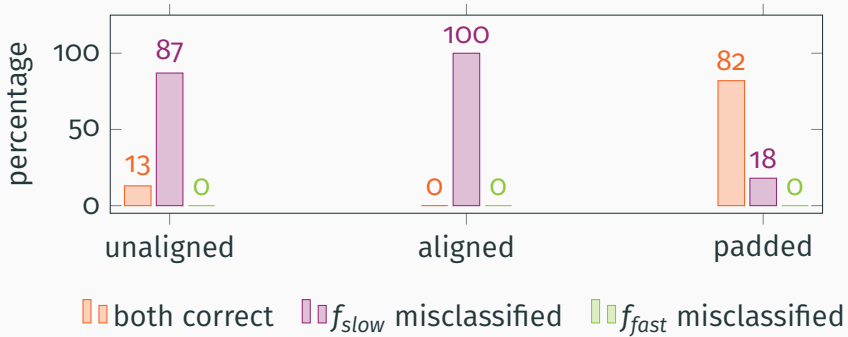


Recovering resolution: Edge thresholding



- nanosecond resolution

Recovering resolution: Edge thresholding



- 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 $\rightarrow$ 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 $\rightarrow$ increase width of element as fast as possible
- timestamp = width of element
- but animation limited to 60 fps $\rightarrow$ 16 ms resolution

Building a timer: Web worker

- 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

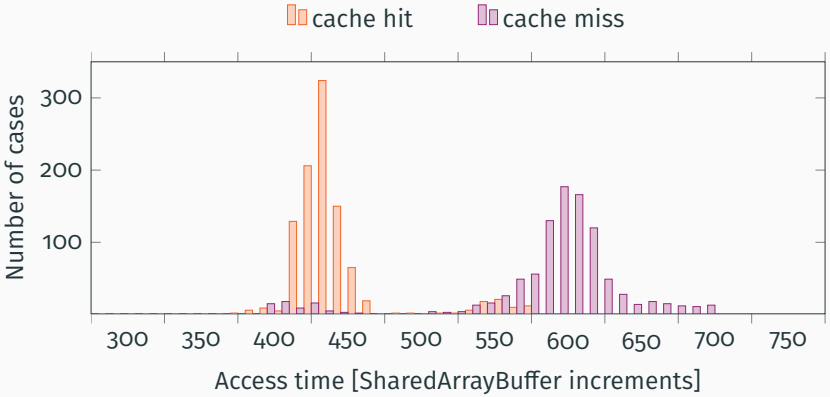
Building a timer: Web worker

- 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

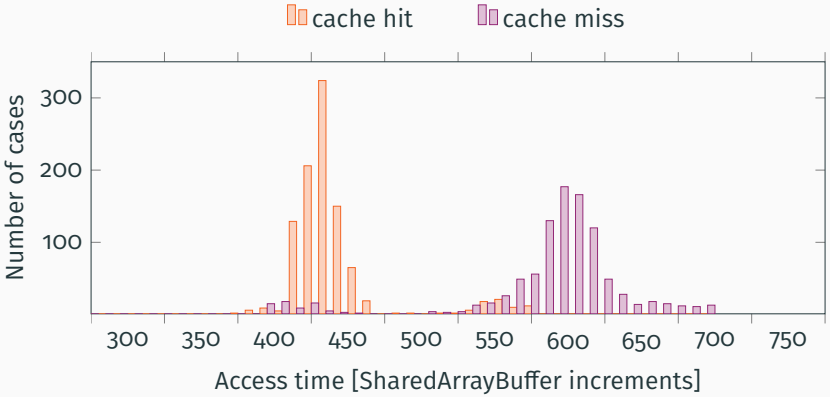
Building a timer: Web worker

- 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?



Building a timer: Is it good enough?



→ we can distinguish cache hits from cache misses (only ≈ 150 cycles difference)!

Take-away

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

J. C. Wray. “An analysis of covert timing channels”. In: *Journal of Computer Security* 1.3-4 (1992), pp. 219–232.

M. Lipp et al. “ARMageddon: Cache Attacks on Mobile Devices”. In: *USENIX Security Symposium*. 2016.

M. Schwarz et al. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: *DIMVA’17*. 2017.

DRAM covert channels in JavaScript!

Setup

- sender: native application in a VM

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

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

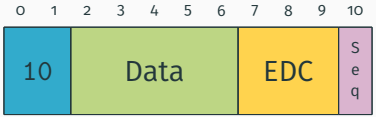
Setup

- 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 0 by doing nothing and 1 by causing row conflict

Setup

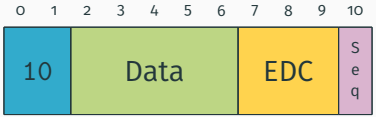
- sender: native application in a VM
- receiver: JavaScript in a web page on the host
- sender and receiver select the same bank
- sender and receiver select a different row inside this bank
- sender transmits 0 by doing nothing and 1 by causing row conflict
- receiver measures access time for its row: fast $\rightarrow$ 0, slow $\rightarrow$ 1

Sending packets



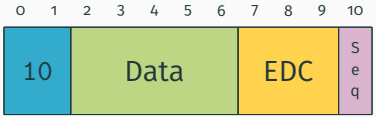
- communication based on 11-bit packets, with 5-bit of data

Sending packets



- communication based on 11-bit packets, with 5-bit of data
- packet starts with a 2-bit preamble

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

Evaluation

- transmission of approximately 11 bits/s

Evaluation

- transmission of approximately 11 bits/s
- can be improved using

Evaluation

- transmission of approximately 11 bits/s
- can be improved using
  - fewer re-transmits

Evaluation

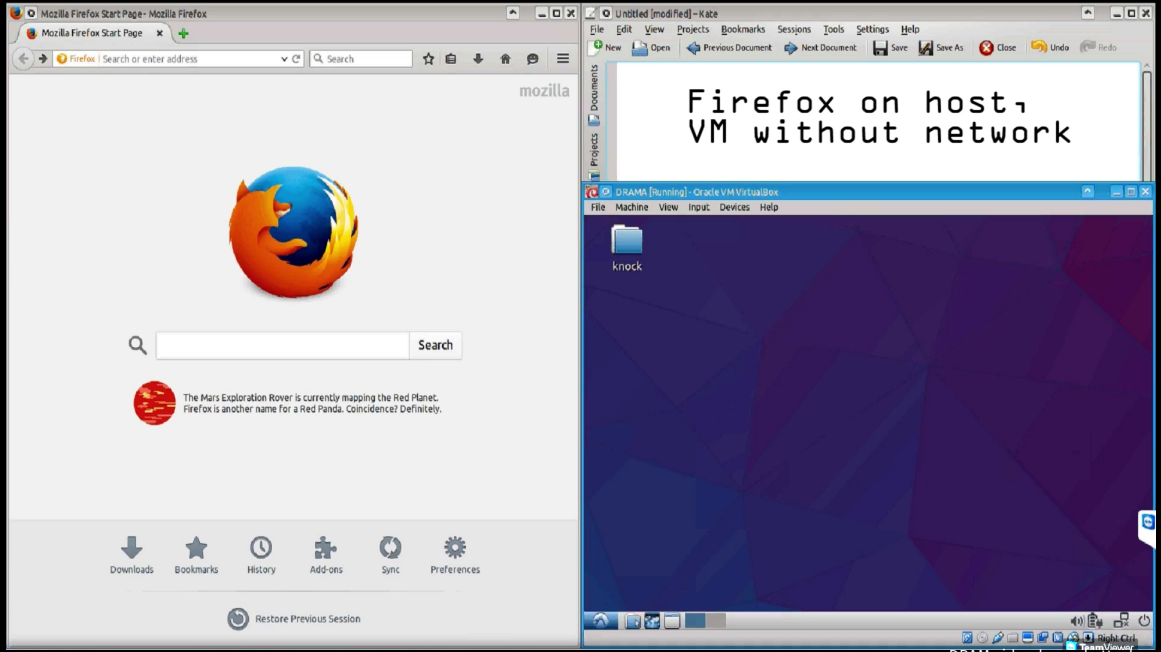
- transmission of approximately 11 bits/s
- can be improved using
  - fewer re-transmits
  - error correction

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 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



DRAM side-channel attack

Conclusion

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