

# Evolution des attaques sur la micro-architecture

---

Clémentine Maurice, Chargée de Recherche CNRS, IRISA

31 Mai 2018–Journées Nationales 2018 Pré-GDR Sécurité Informatique

## Attacks on micro-architecture

- **hardware** usually modeled as an abstract layer behaving correctly

## Attacks on micro-architecture

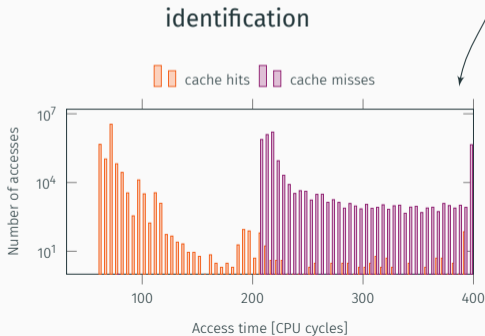
- **hardware** usually modeled as an abstract layer behaving correctly, but possible attacks

# Attacks on micro-architecture

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

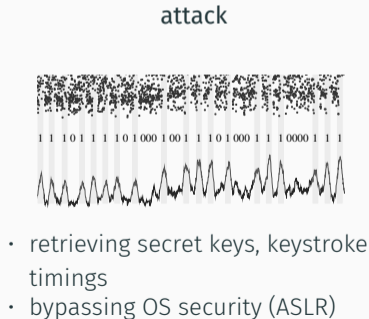
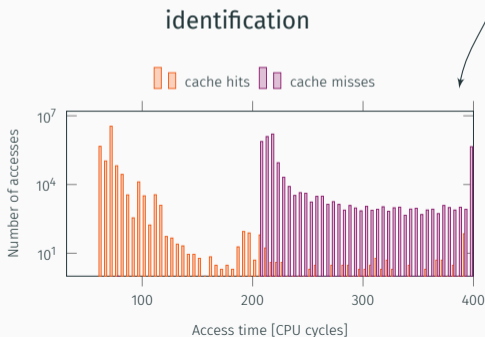
# Attacks on micro-architecture

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

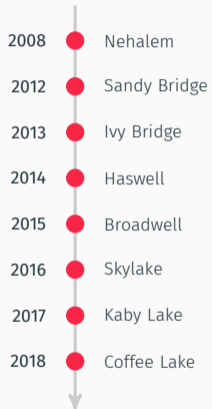


# Attacks on micro-architecture

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

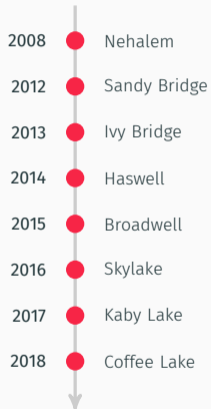


# From small optimizations...



- new microarchitectures yearly

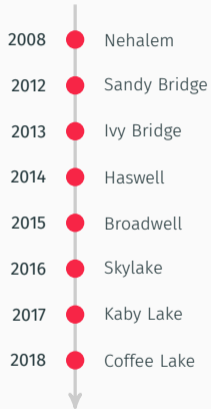
# From small optimizations...



- new microarchitectures yearly
- performance improvement  $\approx 5\%$



# From small optimizations...



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

## ... To microarchitectural side-channel attacks

- microarchitectural side channels come from these optimizations

## ... To microarchitectural side-channel attacks

- microarchitectural side channels come from these optimizations
- several processes are **sharing microarchitectural** components

## ... To microarchitectural side-channel attacks

- microarchitectural side channels come from these optimizations
- several processes are **sharing microarchitectural** components
- attacker infers information from a victim process via hardware usage

## ... To microarchitectural side-channel attacks

- microarchitectural side channels come from these optimizations
- several processes are **sharing microarchitectural** components
- attacker infers information from a victim process via hardware usage
- **pure-software** attacks by **unprivileged** processes

## ... To microarchitectural side-channel attacks

- microarchitectural side channels come from these optimizations
- several processes are **sharing microarchitectural** components
- attacker infers information from a victim process via hardware usage
- **pure-software** attacks by **unprivileged** processes
- sequences of benign-looking actions → hard to detect

Historical recap of past attacks

Historical recap of past attacks

Recent advances



Historical recap of past attacks

Recent advances

Future and challenges

# Historical Recap

---

# From theoretical to practical cache attacks

- first **theoretical** attack in **1996** by Kocher
- first **practical** attack on RSA in **2005** by Percival, on AES in 2006 by Osvik et al.
- **renewed interest** for the field in **2014** after Flush+Reload by Yarom and Falkner

---

P. C. Kocher. "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems". In: *Crypto'96*. 1996.

C. Percival. "Cache missing for fun and profit". In: *Proceedings of BSDCan*. 2005.

D. A. Osvik, A. Shamir, and E. Tromer. "Cache Attacks and Countermeasures: the Case of AES". In: *CT-RSA 2006*. 2006.

Y. Yarom and K. Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security Symposium*. 2014.

# Hyper-threading: Same-core attacks

- threads sharing one core **share resources**: L1, L2 cache, branch predictor

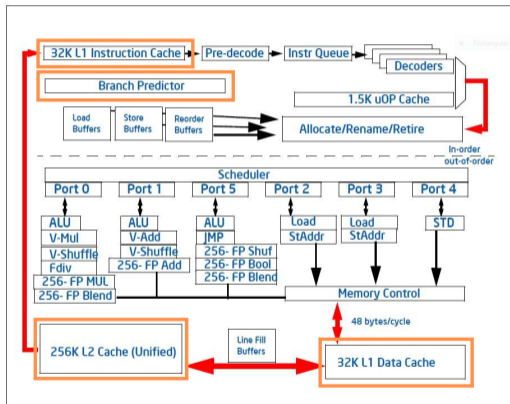


Figure 2-1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

# Branch prediction

- conditional branches → taking the branch or not depends on some condition
- the condition has to be evaluated
- instead of stalling the pipeline → speculative execution of one possible path
- **branch prediction unit** predicts the most likely execution path
  - branch target buffer (BTB): cache that stores the target addresses of previously executed branches
  - branch predictor: makes the prediction on the outcome of the branch

# Branch prediction side channels

- two things can go wrong
  1. **BTB** miss
  2. **misprediction** of the branch
- can be observed by **timing penalty** or **hardware performance counters**
- algorithms with secret-dependent path → detect whether specific branches are taken or not taken

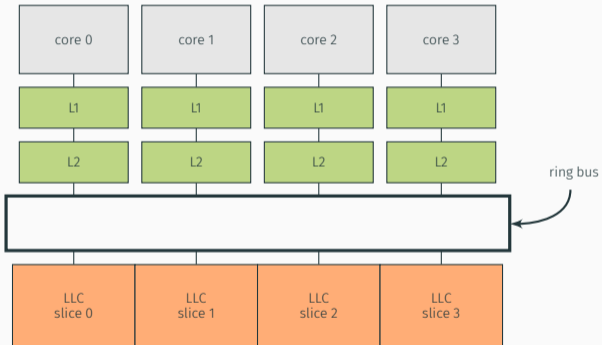
Possible side channels using  
components shared by a core?

Possible side channels using  
components shared by a core?

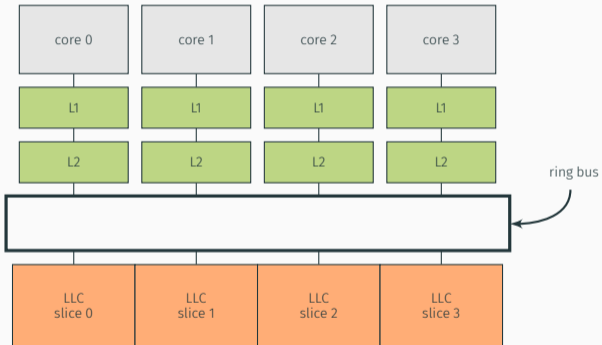
Stop sharing a core!



# Caches on Intel CPUs

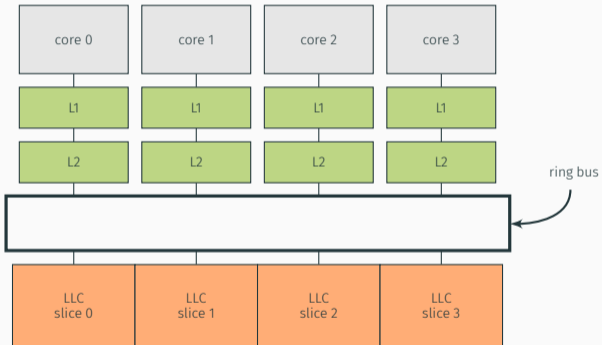


# Caches on Intel CPUs



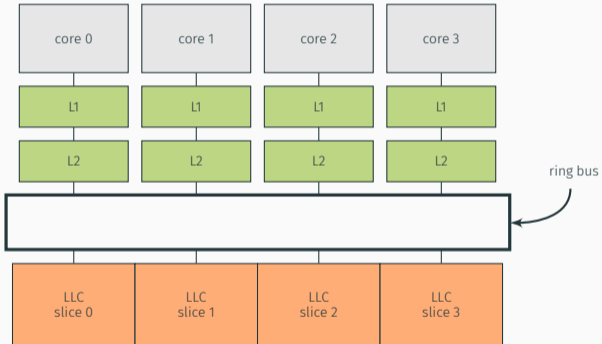
- L1 and L2 are private

# Caches on Intel CPUs



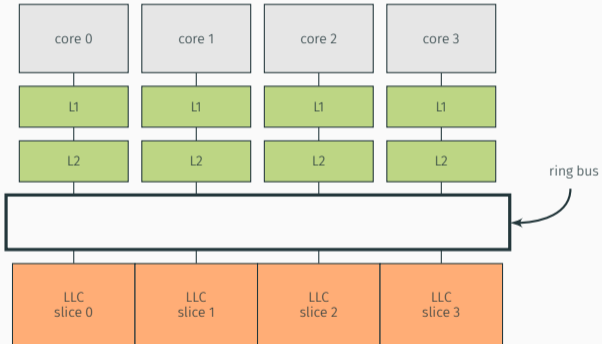
- L1 and L2 are private
- last-level cache

# Caches on Intel CPUs



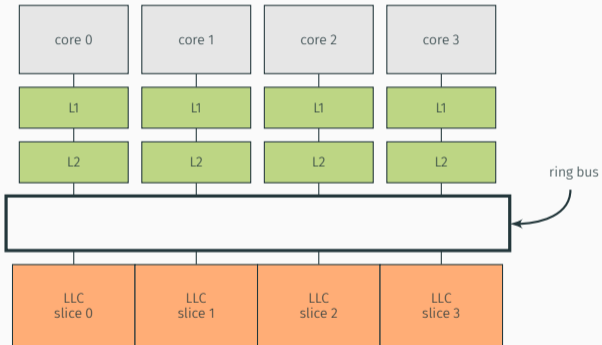
- L1 and L2 are private
- last-level cache
  - divided in **slices**

# Caches on Intel CPUs



- L1 and L2 are private
- last-level cache
  - divided in **slices**
  - **shared** across cores

# Caches on Intel CPUs



- L1 and L2 are private
- last-level cache
  - divided in **slices**
  - **shared** across cores
  - **inclusive**

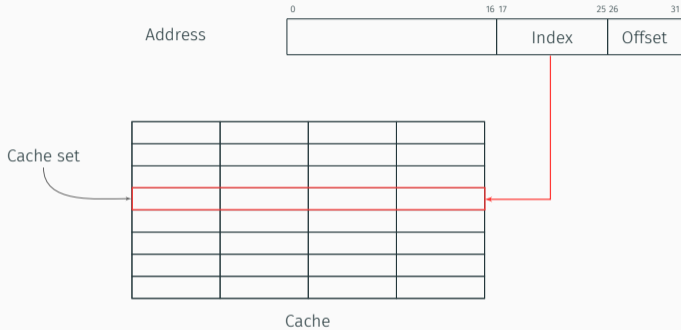
# Set-associative caches



|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Cache

# Set-associative caches



Data loaded in a specific **set** depending on its address



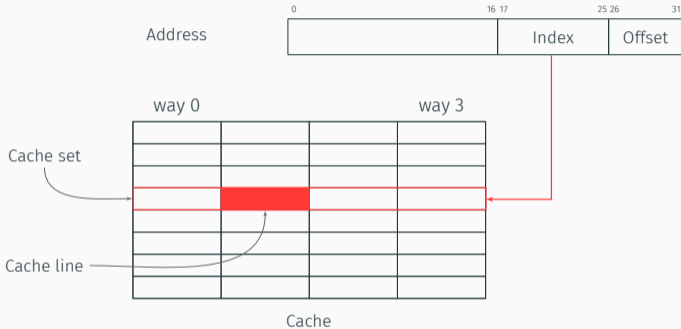
# Set-associative caches



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

- caches improve performance

# Cache attacks

- caches improve performance
- SRAM is expensive → small caches

# Cache attacks

- caches improve performance
- SRAM is expensive → small caches
- different timings for memory accesses

# Cache attacks

- caches improve performance
- SRAM is expensive → small caches
- different timings for memory accesses
  1. data is **cached** → cache hit → **fast**

# Cache attacks

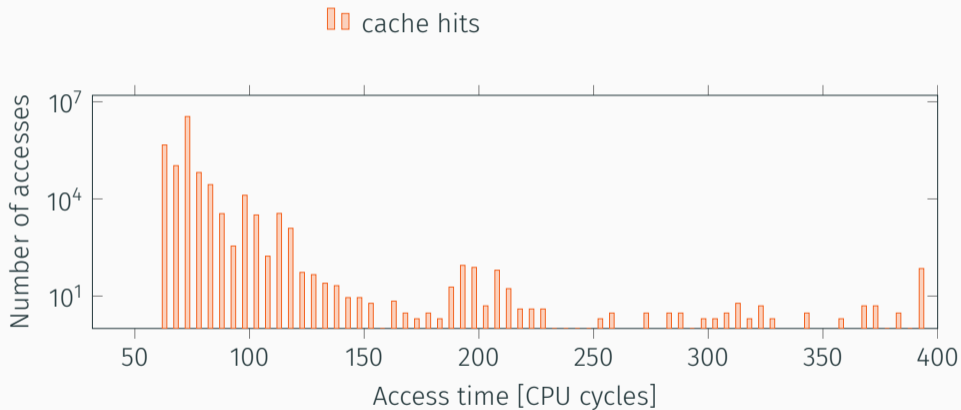
- caches improve performance
- SRAM is expensive → small caches
- different timings for memory accesses
  1. data is **cached** → cache hit → **fast**
  2. data is **not cached** → cache miss → **slow**

# Cache attacks

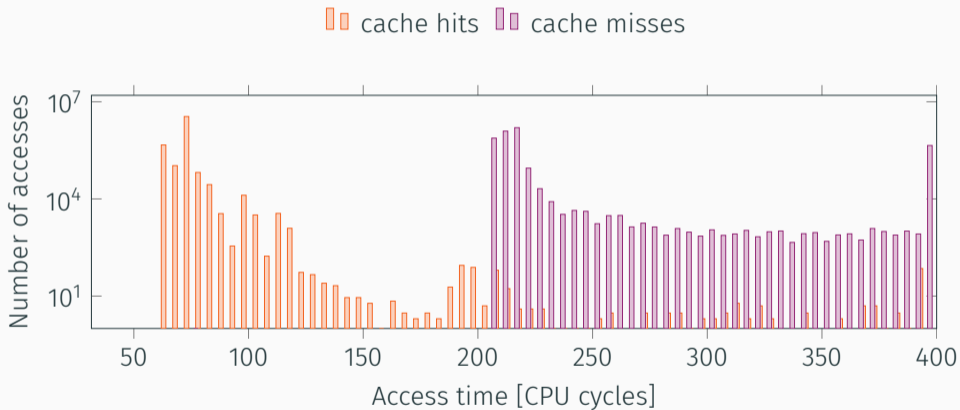
- caches improve performance
- SRAM is expensive → small caches
- different timings for memory accesses
  1. data is **cached** → cache hit → **fast**
  2. data is **not cached** → cache miss → **slow**
- **cache attacks** leverage this timing difference



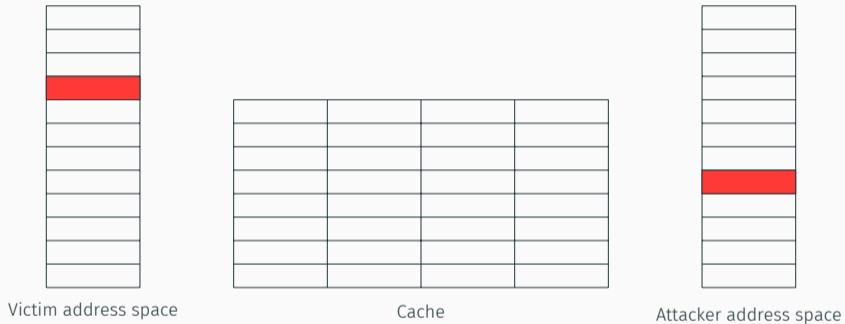
# Timing differences



# Timing differences

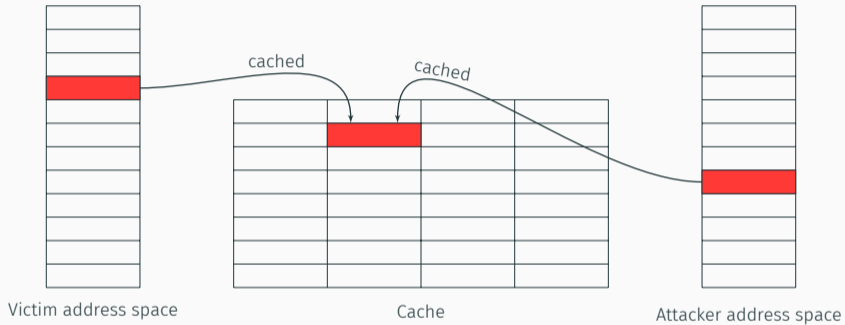


# Cache attacks: Flush+Reload



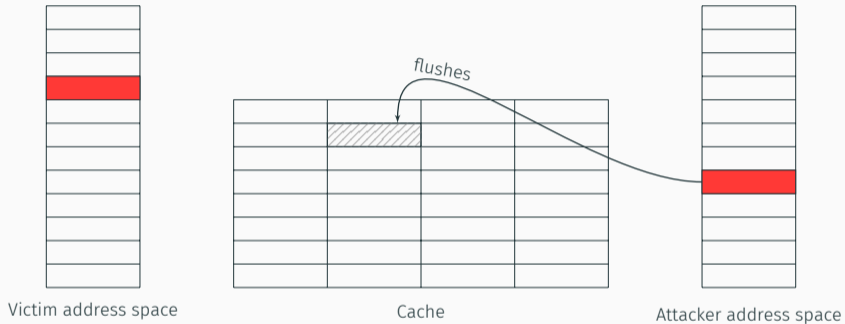
**Step 1:** Attacker maps shared library (shared memory, in cache)

# Cache attacks: Flush+Reload



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

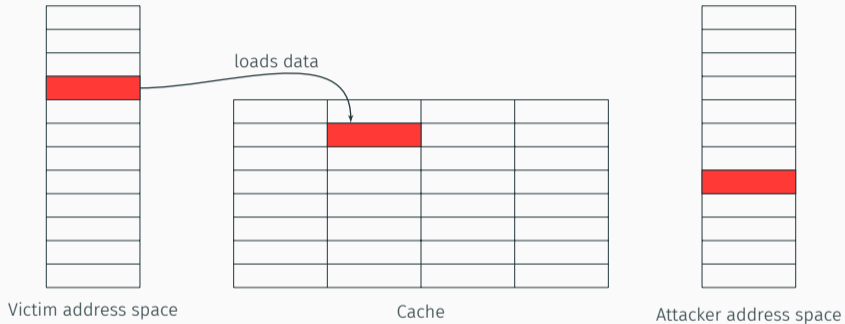
# Cache attacks: Flush+Reload



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

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

# Cache attacks: Flush+Reload

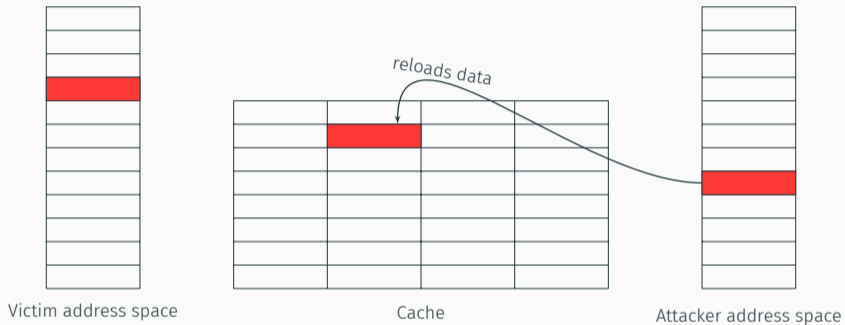


**Step 1:** Attacker maps shared library (shared memory, in cache)

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

**Step 3:** Victim loads the data

# Cache attacks: 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** side channel attacks on **crypto** algorithms
  - RSA: 96.7% of secret key bits in a single signature
  - AES: full key recovery in 30000 dec. (a few seconds)

---

Y. Yarom and K. Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security Symposium*. 2014

B. Gülmezoglu, M. S. Inci, T. Eisenbarth, and B. Sunar. "A Faster and More Realistic Flush+Reload Attack on AES". In: *Constructive Side-Channel Analysis and Secure Design (COSADE)*. 2015

D. Gruss, R. Spreitzer, and S. Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches". In: *USENIX Security Symposium*. 2015

[https://github.com/IAIK/cache\\_template\\_attacks](https://github.com/IAIK/cache_template_attacks)



# Flush+Reload: Applications

- **cross-VM** side channel attacks on **crypto** algorithms
  - RSA: 96.7% of secret key bits in a single signature
  - AES: full key recovery in 30000 dec. (a few seconds)
- Cache Template Attacks: **automatically** finds information leakage
  - side channel on **keystrokes** and AES T-tables implementation

---

Y. Yarom and K. Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security Symposium*. 2014

B. Gülmezoglu, M. S. Inci, T. Eisenbarth, and B. Sunar. "A Faster and More Realistic Flush+Reload Attack on AES". In: *Constructive Side-Channel Analysis and Secure Design (COSADE)*. 2015

D. Gruss, R. Spreitzer, and S. Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches". In: *USENIX Security Symposium*. 2015

[https://github.com/IAIK/cache\\_template\\_attacks](https://github.com/IAIK/cache_template_attacks)

## Flush+Reload: Pros and cons

- **fine granularity**: 1 cache line (64 Bytes)

## Flush+Reload: Pros and cons

- **fine granularity**: 1 cache line (64 Bytes)
- but requires shared memory

## Flush+Reload: Pros and cons

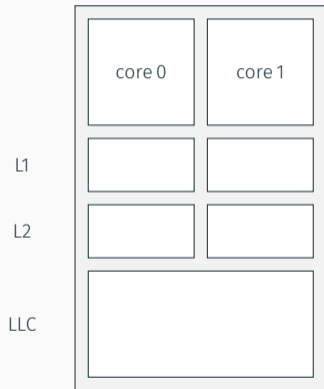
- **fine granularity**: 1 cache line (64 Bytes)
  - but requires shared memory
- **memory deduplication** between VMs

Possible side channels using  
memory deduplication?

Possible side channels using  
memory deduplication?

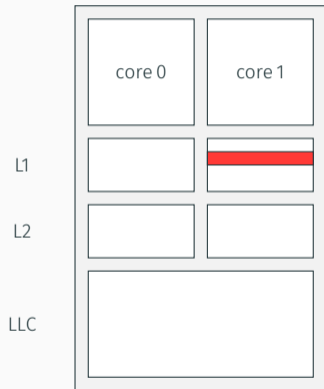
Disable memory deduplication!

# Inclusive property



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

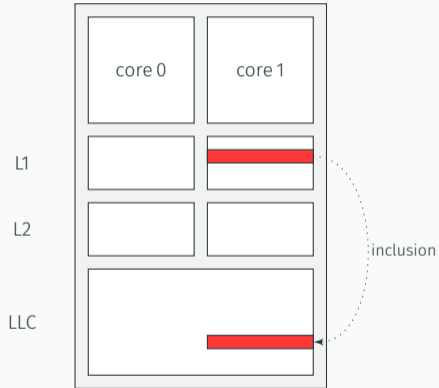
# Inclusive property



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

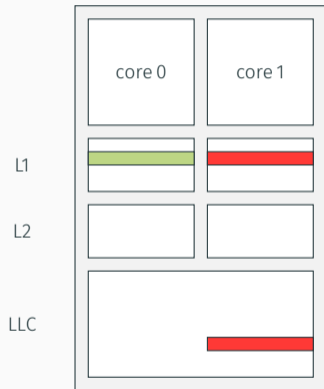


# Inclusive property



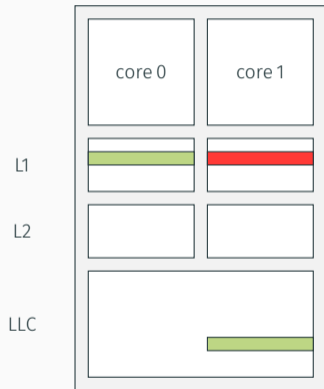
- **inclusive** LLC: superset of L1 and L2

# Inclusive property



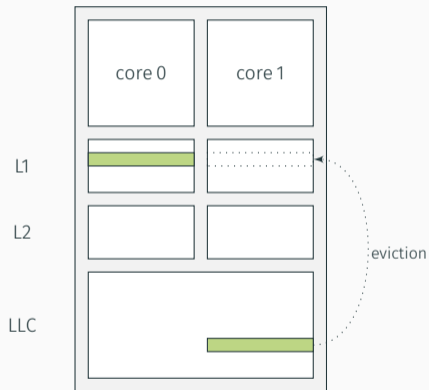
- **inclusive** LLC: superset of L1 and L2

# Inclusive property



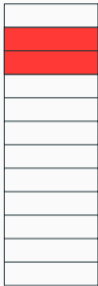
- **inclusive** LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2

# Inclusive property

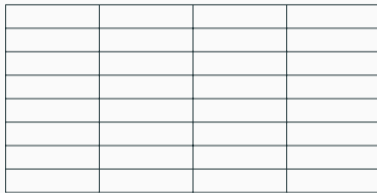


- **inclusive** LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2
- a core can **evict lines** in the private L1 of another core

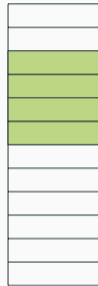
# Cache attacks: Prime+Probe



Victim address space

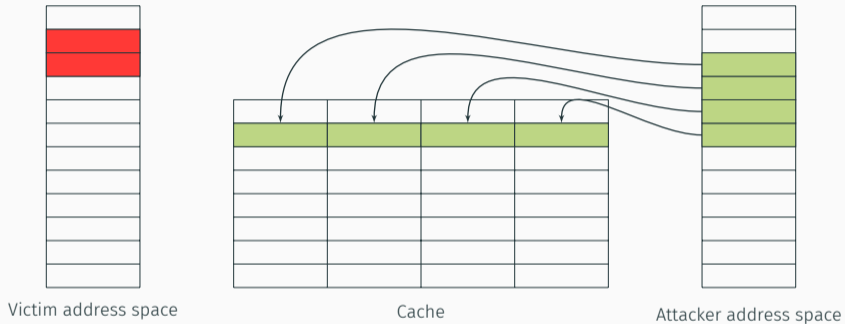


Cache



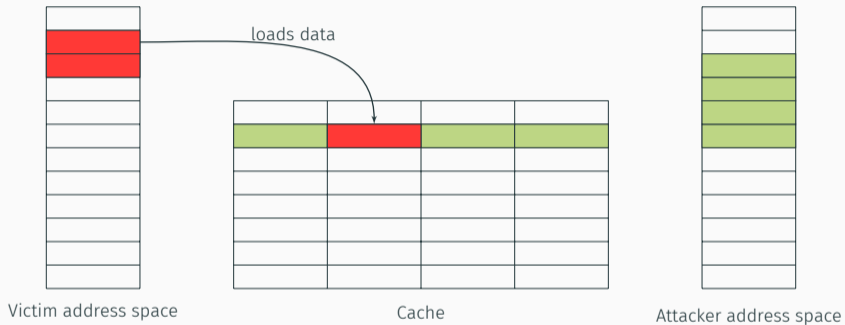
Attacker address space

# Cache attacks: Prime+Probe



**Step 1:** Attacker **primes**, *i.e.*, fills, the cache (no shared memory)

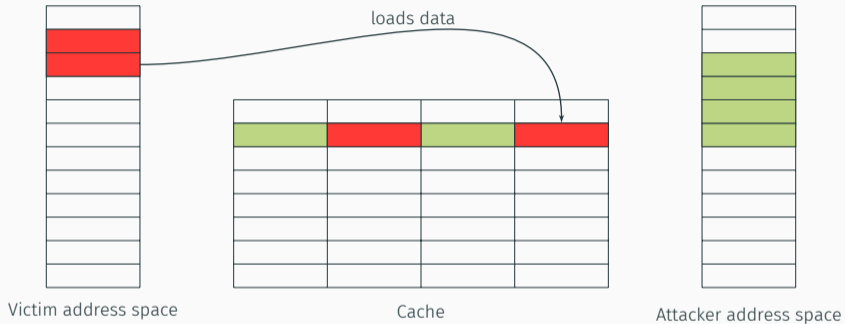
# Cache attacks: Prime+Probe



**Step 1:** Attacker **primes**, *i.e.*, fills, the cache (no shared memory)

**Step 2:** Victim evicts cache lines while running

# Cache attacks: Prime+Probe

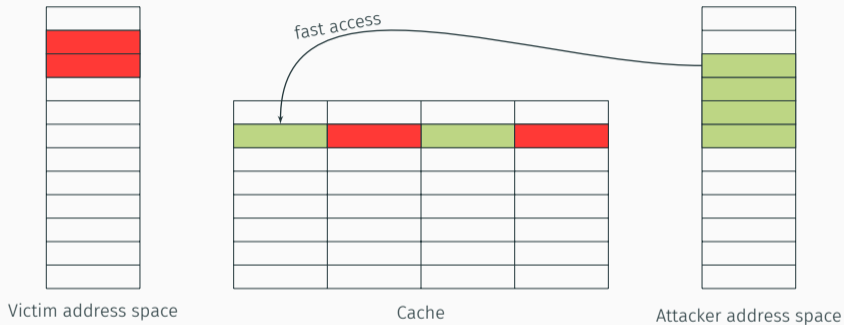


**Step 1:** Attacker **primes**, *i.e.*, fills, the cache (no shared memory)

**Step 2:** Victim evicts cache lines while running



# Cache attacks: Prime+Probe

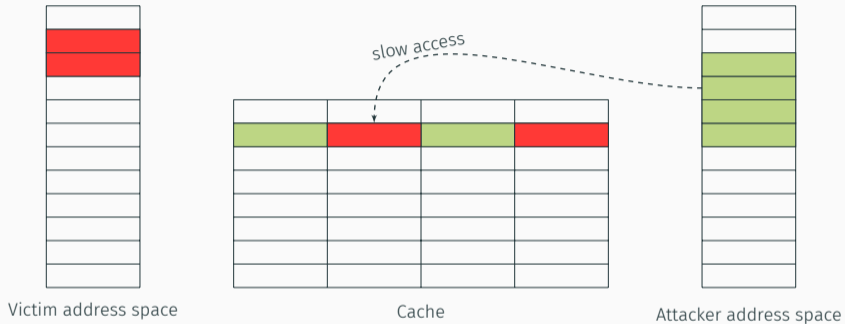


**Step 1:** Attacker **primes**, *i.e.*, fills, the cache (no shared memory)

**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

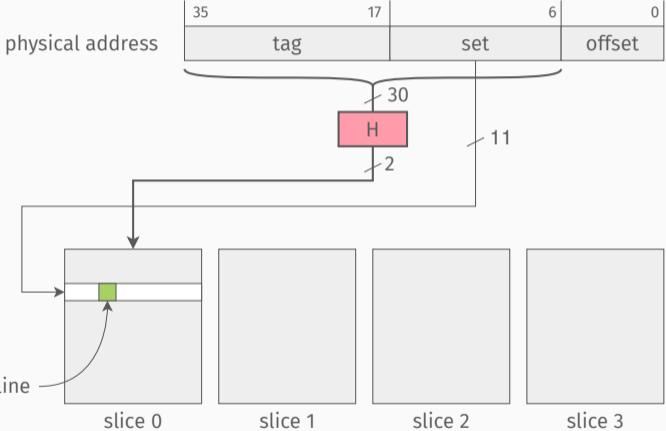
**Step 3:** Attacker **probes** data to determine if set has been accessed

## Challenges with Prime+Probe

We need to evict caches 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?

# Last-level cache addressing



# Last-level cache addressing

- last-level cache → as many slices as cores
- **undocumented** hash function that maps a physical address to a slice
- designed for performance

For  $2^k$  slices:

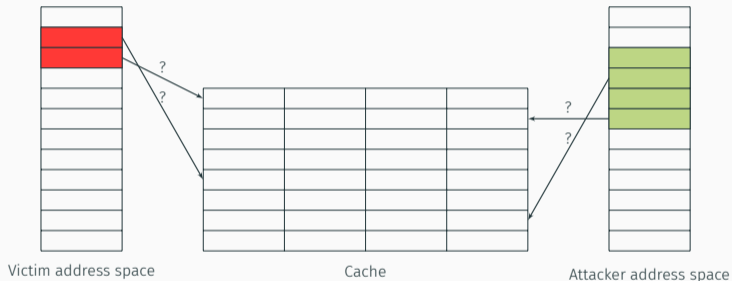
physical address  
30 bits



slice  $(o_0, \dots, o_{k-1})$   
 $k$  bits

# Prime+Probe on recent processors?

Undocumented function → impossible to **target a set**

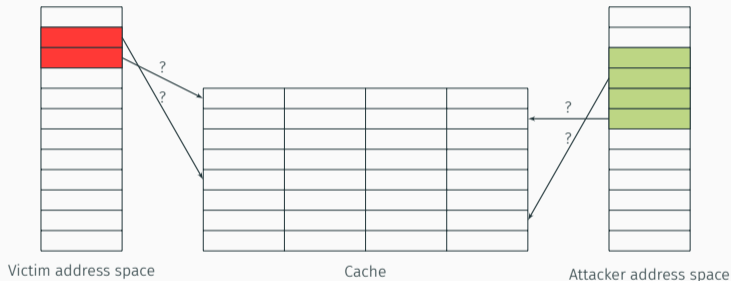


---

C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. "Reverse Engineering Intel Complex Addressing Using Performance Counters". In: RAID'15. 2015

# Prime+Probe on recent processors?

Undocumented function → impossible to **target a set**



→ We reverse-engineered the function!

---

C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. "Reverse Engineering Intel Complex Addressing Using Performance Counters". In: *RAID'15*. 2015

# Prime+Probe: Applications

- **cross-VM** side channel attacks on **crypto** algorithms:
  - 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**

---

F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: *S&P'15*. 2015.

Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications". In: *CCS'15*. 2015.

C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud". In: *NDSS'17*. to appear. 2017.



Possible side channels using  
components shared by a CPU?

Possible side channels using  
components shared by a CPU?

Stop sharing a CPU!?

## Recent Advances

---

## Building practical attacks

## Covert channels in the cloud

- covert channel: two processes **communicating** with each other
  - **not allowed** to do so, e.g., across VMs

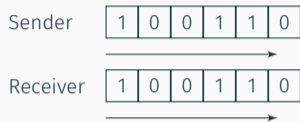
## Covert channels in the cloud

- covert channel: two processes **communicating** with each other
  - **not allowed** to do so, e.g., across VMs
- literature: stops working with **noise** on the machine

## Covert channels in the cloud

- covert channel: two processes **communicating** with each other
  - **not allowed** to do so, e.g., across VMs
- literature: stops working with **noise** on the machine
- solution? “Just use error-correcting codes”

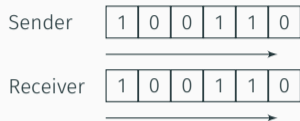
# Why can't we just use error correcting codes?



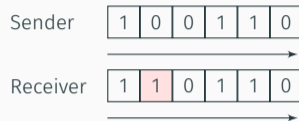
**(a)** Transmission without errors



# Why can't we just use error correcting codes?

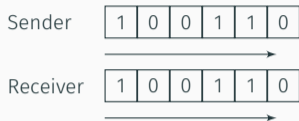


(a) Transmission without errors

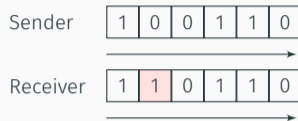


(b) Noise: **substitution** error

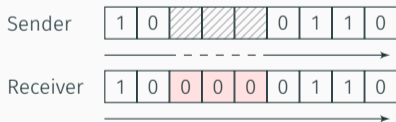
# Why can't we just use error correcting codes?



(a) Transmission without errors

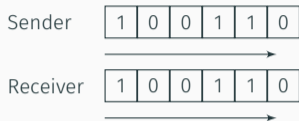


(b) Noise: **substitution** error

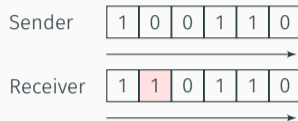


(c) Sender descheduled: **insertions**

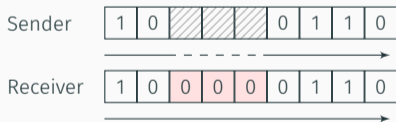
# Why can't we just use error correcting codes?



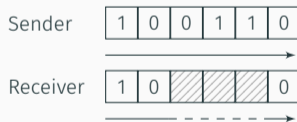
(a) Transmission without errors



(b) Noise: **substitution** error



(c) Sender descheduled: **insertions**



(d) Receiver descheduled: **deletions**

# Our robust covert channel

- **physical** layer:
  - transmits words as a sequence of '0's and '1's
  - deals with synchronization errors
- **data-link** layer:
  - divides data to transmit into packets
  - corrects the remaining errors

---

C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud". In: *NDSS'17*. to appear. 2017

## Physical layer: Sending '0's and '1's

- sender and receiver agree on one set

## Physical layer: Sending '0's and '1's

- sender and receiver agree on one set
- receiver probes the set continuously

## Physical layer: Sending '0's and '1's

- sender and receiver agree on one set
- receiver probes the set continuously
- sender **transmits '0'** doing nothing
  - lines of the receiver still in cache → **fast access**

## Physical layer: Sending '0's and '1's

- sender and receiver agree on one set
- receiver probes the set continuously
- sender **transmits '0'** doing nothing
  - lines of the receiver still in cache → **fast access**
- sender **transmits '1'** accessing addresses in the set
  - evicts lines of the receiver → **slow access**



## Eviction set generation

- need a set of addresses in the **same cache set** and **same slice**

## Eviction set generation

- need a set of addresses in the **same cache set** and **same slice**
- problem: slice number depends on all bits of the physical address

## Eviction set generation

- need a set of addresses in the **same cache set** and **same slice**
- problem: slice number depends on all bits of the physical address



- we can build a set of addresses in the **same cache set** and **same slice**

# Eviction set generation

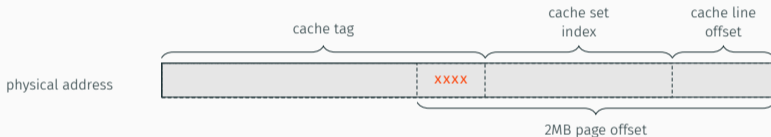
- need a set of addresses in the **same cache set** and **same slice**
- problem: slice number depends on all bits of the physical address



- we can build a set of addresses in the **same cache set** and **same slice**
- without knowing **which slice**

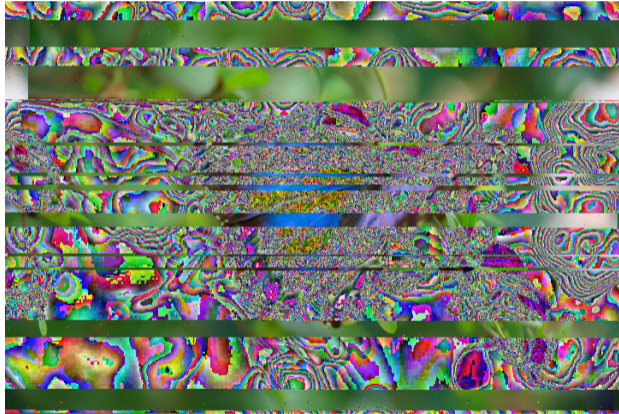
# Eviction set generation

- need a set of addresses in the **same cache set** and **same slice**
- problem: slice number depends on all bits of the physical address



- we can build a set of addresses in the **same cache set** and **same slice**
  - without knowing **which slice**
- we use a **jamming agreement**

## Sending the first image



# Handling synchronization errors



# Handling synchronization errors

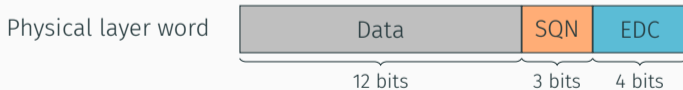
- deletion errors: **request-to-send scheme** that also serves as ack
  - 3-bit sequence number
  - request: encoded sequence number (7 bits)



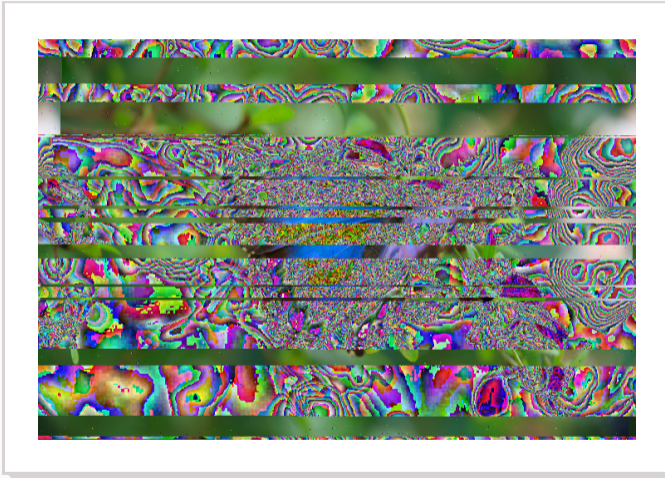


# Handling synchronization errors

- deletion errors: **request-to-send scheme** that also serves as ack
  - 3-bit sequence number
  - request: encoded sequence number (7 bits)
- '0'-insertion errors: **error detection code** → Berger codes
  - appending the number of '0's in the word to itself→ property: a word cannot consist solely of '0's



## Synchronization (before)



## Synchronization (after)



## Synchronization (after)



## Synchronization (after)

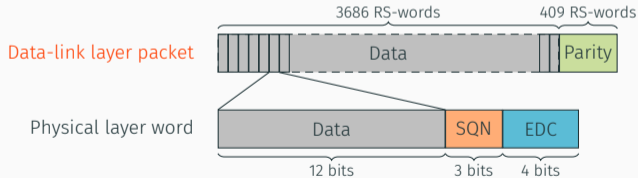


## Data-link layer: Error correction

- Reed-Solomon codes to correct the remaining errors

# Data-link layer: Error correction

- **Reed-Solomon** codes to correct the remaining errors
- RS word size = physical layer word size = 12 bits
- packet size =  $2^{12} - 1 = 4095$  RS words
- 10% error-correcting code: 409 parity and 3686 data RS words



## Error correction (after)





# Evaluation

---

| Environment | Bit rate   | Error rate | Noise |
|-------------|------------|------------|-------|
| Native      | 75.10 KBps | 0.00%      | -     |

---

# Evaluation

---

| Environment | Bit rate   | Error rate | Noise              |
|-------------|------------|------------|--------------------|
| Native      | 75.10 KBps | 0.00%      | -                  |
| Native      | 36.03 KBps | 0.00%      | <b>stress -m 1</b> |

---

# Evaluation

---

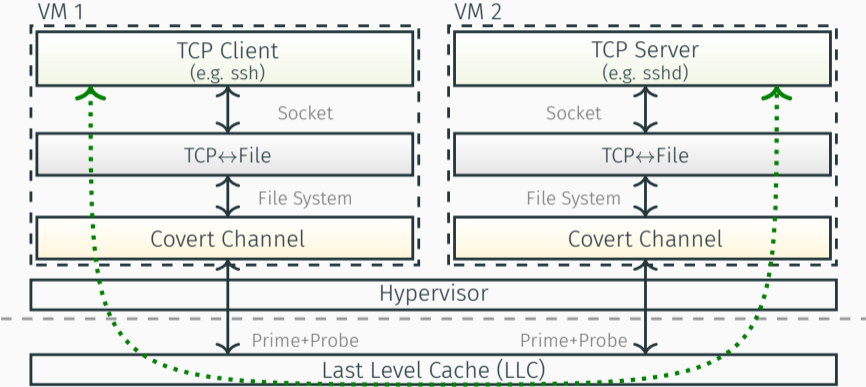
| Environment | Bit rate   | Error rate | Noise                    |
|-------------|------------|------------|--------------------------|
| Native      | 75.10 KBps | 0.00%      | -                        |
| Native      | 36.03 KBps | 0.00%      | <code>stress -m 1</code> |
| Amazon EC2  | 45.25 KBps | 0.00%      | -                        |

---

# Evaluation

| Environment | Bit rate   | Error rate | Noise  |
|-------------|------------|------------|--|
| Native      | 75.10 KBps | 0.00%      | –  |
| Native      | 36.03 KBps | 0.00%      | <b>stress -m 1</b>   |
| Amazon EC2  | 45.25 KBps | 0.00%      | –  |
| Amazon EC2  | 45.09 KBps | 0.00%      | web server serving files on sender VM  |
| Amazon EC2  | 42.96 KBps | 0.00%      | <b>stress -m 2</b> on sender VM  |
| Amazon EC2  | 42.26 KBps | 0.00%      | <b>stress -m 1</b> on receiver VM  |
| Amazon EC2  | 37.42 KBps | 0.00%      | web server on all 3 VMs, <b>stress -m 4</b> on 3rd VM, <b>stress -m 1</b> on sender and receiver VMs |
| Amazon EC2  | 34.27 KBps | 0.00%      | <b>stress -m 8</b> on third VM   |

# Building an SSH connection



# SSH evaluation

Between two instances on Amazon EC2

| Noise                                   | Connection |
|---|------------|
| No noise                                | ✓          |
| <code>stress -m 8</code> on third VM    | ✓          |
| Web server on third VM                  | ✓          |
| Web server on SSH server VM             | ✓          |
| Web server on all VMs                   | ✓          |
| <code>stress -m 1</code> on server side | unstable   |

# SSH evaluation

Between two instances on Amazon EC2

| Noise                                   | Connection |
|---|------------|
| No noise                                | ✓          |
| <code>stress -m 8</code> on third VM    | ✓          |
| Web server on third VM                  | ✓          |
| Web server on SSH server VM             | ✓          |
| Web server on all VMs                   | ✓          |
| <code>stress -m 1</code> on server side | unstable   |

Telnet also works with occasional corrupted bytes with `stress -m 1`

Increasing the attack surface





## Not just caches: also DRAM, MMU, TLB, GPUs...

- **DRAM** [Pessl et al., DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks (USENIX Security 2016)]
- **GPU** [Frigo et al., Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU (S&P 2018)]
- **MMU** [Van Schaik et al., Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think (USENIX Security 2018)]
- **TLB** [Gras et al., Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks (USENIX Security 2018)]

## Not just native code on x86: mobile and web too

- Oren et al., The Spy in the Sandbox: Practical **Cache Attacks in JavaScript** and their Implications (CCS 2015)
- Lipp et al., ARMageddon: **Cache Attacks on Mobile Devices** (USENIX Security 2016)
- Gras et al., ASLR on the Line: Practical Cache Attacks on the MMU (NDSS 2017)
- Schwarz et al., Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript (FC 2017)
- Lipp et al., Practical Keystroke Timing Attacks in Sandboxed JavaScript (ESORICS 2017)

## Not just side channels: software fault attacks too

- Kim et al., Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors (ISCA 2014)
- Bosman et al., Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector (S&P 2016)
- Gruss et al., Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript (DIMVA 2016)
- Van der Veen et al., Drammer: Deterministic Rowhammer Attacks on Mobile Platforms (CCS 2016)
- Tang et al., CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management (USENIX Security 2017)

## Future and Challenges

---

# Challenges and questions

- lack of documentation on microarchitectural components
- which components are vulnerable to these attacks?
- which software is vulnerable to these attacks?
- how to **prevent attacks** based on performance optimizations **without removing performance**?

## Future: Towards remote attacks?

- all past and recent attacks use some code execution (native or JavaScript)
- can we perform attacks **without a single attacker-controlled line of code** on the targeted system?
- very recent remote Rowhammer attacks:
  - using RDMA [Tatar et al., Throwhammer: Rowhammer Attacks over the Network and Defenses (USENIX ATC 2018)]
  - using uncached memory and quality-of-service techniques [Lipp et al., Nethammer: Inducing Rowhammer Faults through Network Requests (arXiv:1805.04956)]
- more to come?

## Future: More speculative execution side channels?



- **Meltdown** breaks isolation between applications and kernel by exploiting Out-of-Order execution
- **Spectre** mistrains branch prediction to speculatively execute code that should not be executed
- 3 initial variants in January, a 4th one on May 21
- more to come?




# Conclusion

- first paper by Kocher in 1996: 22 years of research in this area
- domain still in expansion: increasing number of papers published since 2015
- adopted countermeasures only target cryptographic implementations
- still a lot more to discover on this iceberg :)
- quick fixes don't work
- still a lot more work needed to find satisfying countermeasures

# Thank you!

Contact

 clementine.maurice@irisa.fr

 @BloodyTangerine

# Evolution des attaques sur la micro-architecture

---

Clémentine Maurice, Chargée de Recherche CNRS, IRISA

31 Mai 2018–Journées Nationales 2018 Pré-GDR Sécurité Informatique