Microarchitectural Incontinence

You would leak too if you were so fast!

Daniel Gruss
Graz University of Technology

October 18, 2016 — Hacktivity

You know water races?

Going too fast

- CPU frequency i386 $\rightarrow$ Skylake: $\times$ 160
- DRAM module capacity KB $\rightarrow$ GB: $\times$ 1 million
- DRAM manufacturing size µm $\rightarrow$ nm

Going too fast

- CPU frequency i386 $\rightarrow$ Skylake: $\times$ 160
- DRAM module capacity KB $\rightarrow$ GB: $\times$ 1 million
- DRAM manufacturing size µm $\rightarrow$ nm

Try a water race at 160$\times$ speed with tiny cups

Intel CPUs

2008	Nehalem
2011	Sandy Bridge
2012	Ivy Bridge
2013	Haswell
2014	Broadwell
2015	Skylake

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

Intel CPUs

2008	Nehalem
2011	Sandy Bridge
2012	Ivy Bridge
2013	Haswell
2014	Broadwell
2015	Skylake

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

$\rightarrow$ more and more leakage

Whoami

- Daniel Gruss
- PhD Student, Graz University of Technology
- Twitter: `@lavados`
- Email: `daniel.gruss@iaik.tugraz.at`

Side channels

- safe software infrastructure → no bugs, e.g., Heartbleed

Side channels

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

Side channels

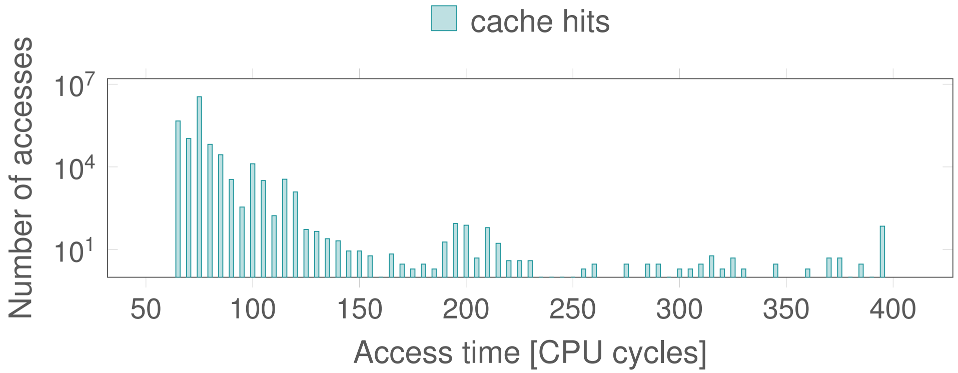
- safe software infrastructure $\rightarrow$ no bugs, e.g., Heartbleed
- does not mean safe execution
- information leaks because of the hardware it runs on
- no “bug” in the sense of a mistake $\rightarrow$ lots of performance optimizations

Side channels

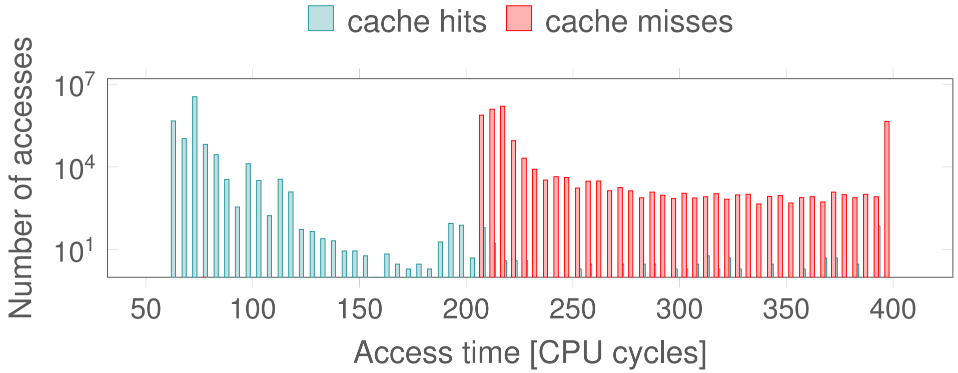
- safe software infrastructure → no bugs, e.g., Heartbleed
- does not mean safe execution
- information leaks because of the hardware it runs on
- no “bug” in the sense of a mistake → lots of performance optimizations

→ crypto and other sensitive info, e.g., keystrokes and mouse movements

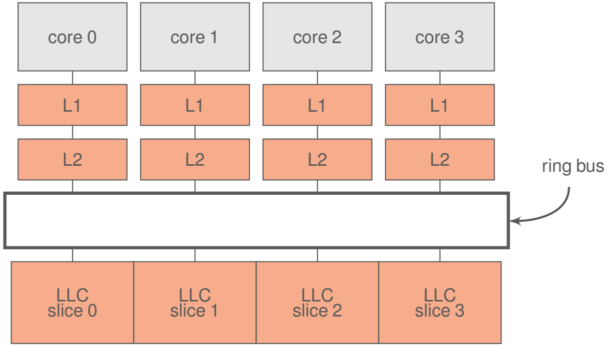
Timing differences



Timing differences

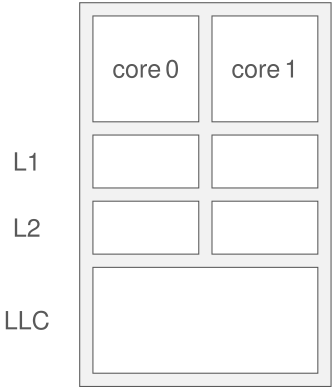


Caches on Intel CPUs



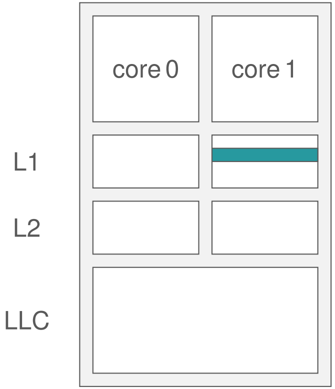
- L1 and L2 are private
- last-level cache:
  - divided in slices
  - shared across cores
  - inclusive

Inclusive property



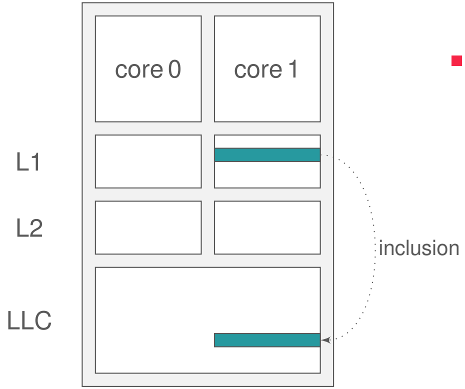
- inclusive LLC: superset of L1 and L2

Inclusive property



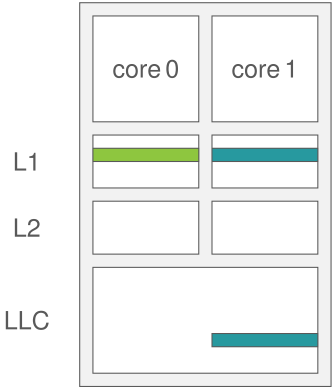
- inclusive LLC: superset of L1 and L2

Inclusive property



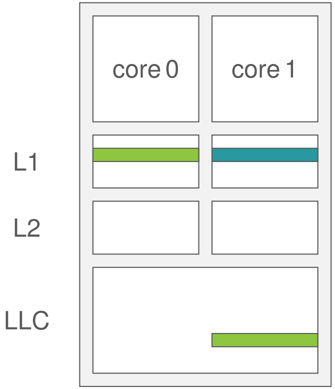
- inclusive LLC: superset of L1 and L2

Inclusive property



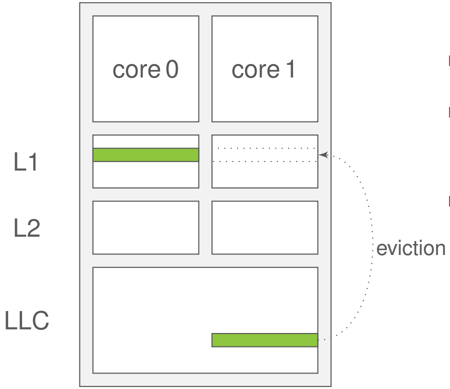
- inclusive LLC: superset of L1 and L2

Inclusive property



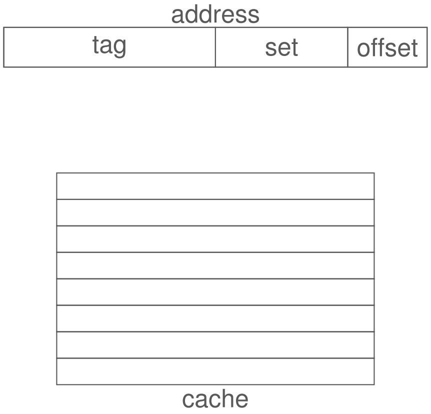
- inclusive LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2

Inclusive property



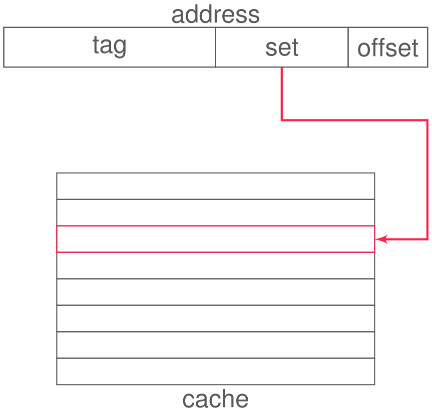
- inclusive LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2
- a core can evict lines in the private L1 of another core

Set-associative caches



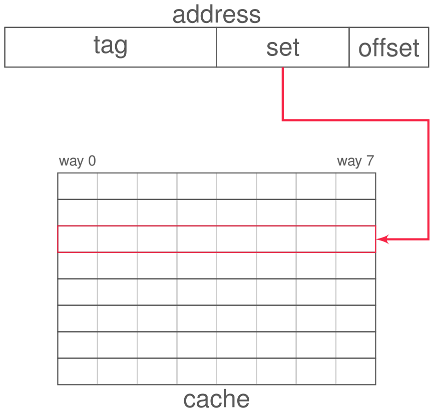
- line loaded in a specific set depending on its address
  - L1: virtually indexed
  - L2, LLC: physically indexed

Set-associative caches



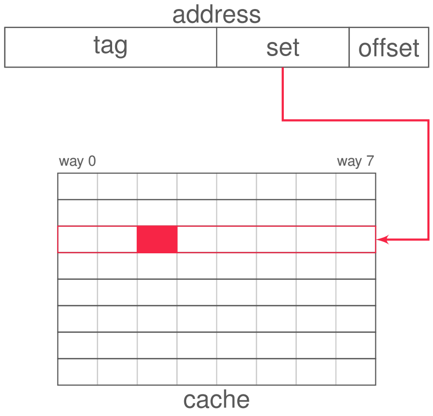
- line loaded in a specific set depending on its address
  - L1: virtually indexed
  - L2, LLC: physically indexed

Set-associative caches



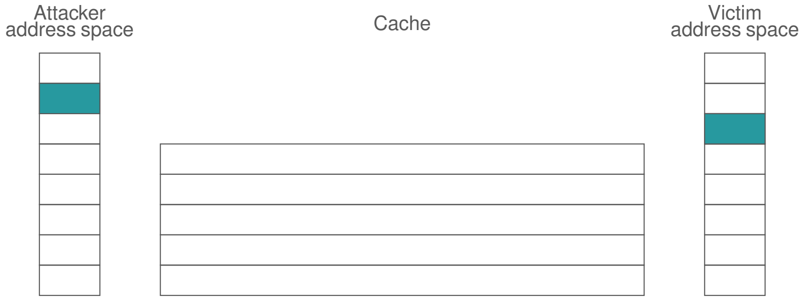
- line loaded in a specific set depending on its address
  - L1: virtually indexed
  - L2, LLC: physically indexed
- several ways per set

Set-associative caches



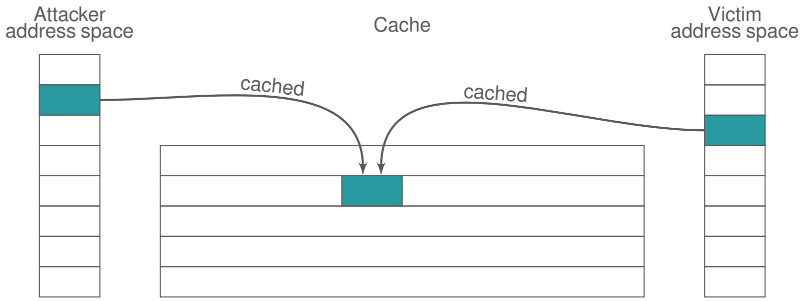
- line loaded in a specific set depending on its address
  - L1: virtually indexed
  - L2, LLC: physically indexed
- several ways per set
- replacement policy decides line to evict to store a new one

Flush+Reload



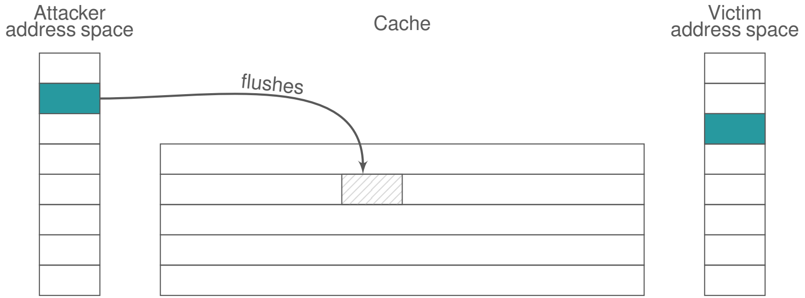
step 0: attacker maps shared library → shared memory, shared in cache

Flush+Reload



step 0: attacker maps shared library → shared memory, shared in cache

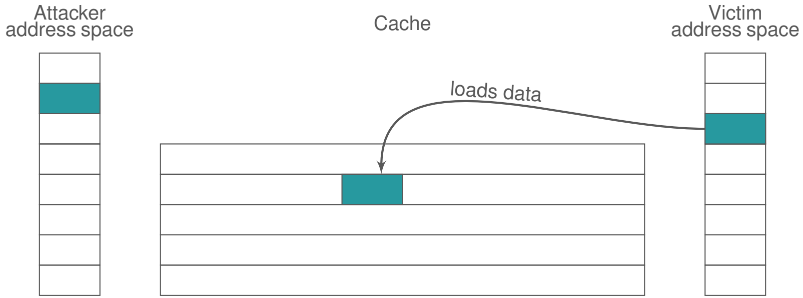
Flush+Reload



step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line with `clflush`

Flush+Reload

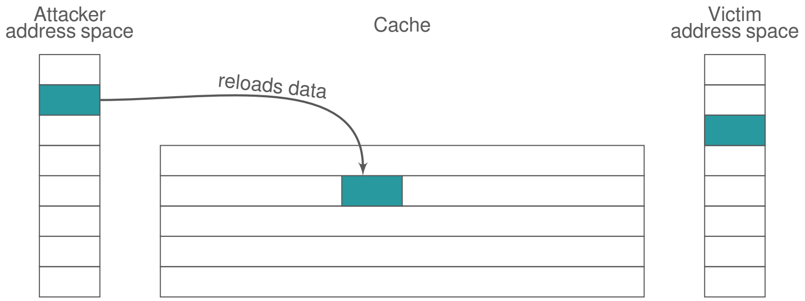


step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line with `clflush`

step 2: victim loads data while performing encryption

Flush+Reload



step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line with `clflush`

step 2: victim loads data while performing encryption

step 3: attacker reloads data → fast access if the victim loaded the line

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. Gülmezoğlu, M. S. Inci, T. Eisenbarth, and B. Sunar. “A Faster and More Realistic Flush+Reload Attack on AES”. . In: COSADE'15. 2015.

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

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 exploits cache-based information leakage

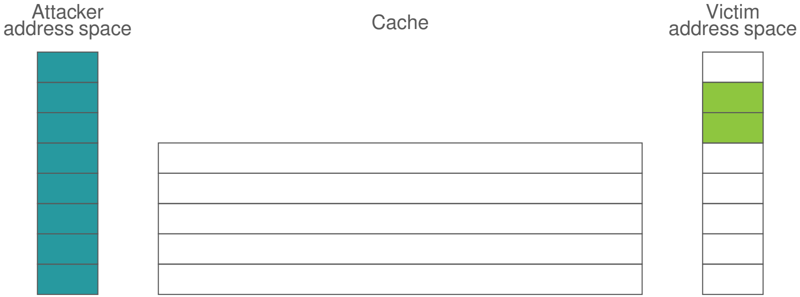
Y. Yarom and K. Falkner. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: USENIX Security Symposium. 2014.

B. Gülmezoğlu, M. S. Inci, T. Eisenbarth, and B. Sunar. “A Faster and More Realistic Flush+Reload Attack on AES”. . In: COSADE'15. 2015.

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

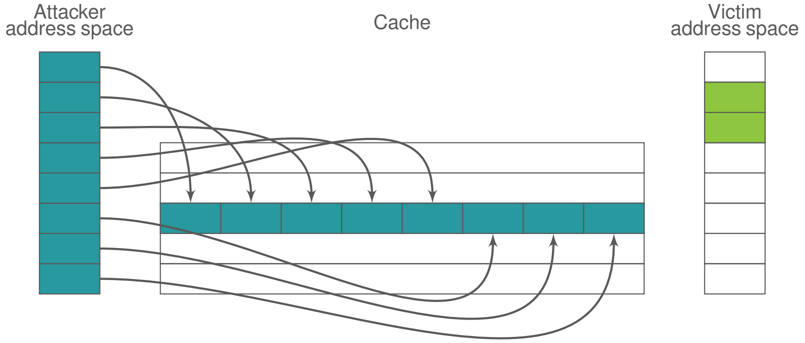
`https://github.com/IAIK/cache_template_attacks`

Prime+Probe



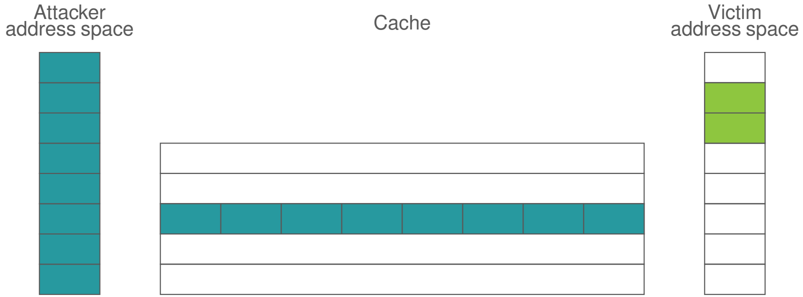
step 0: attacker fills the cache (prime)

Prime+Probe



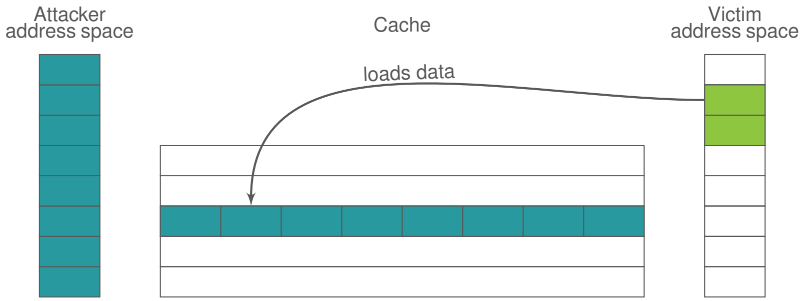
step 0: attacker fills the cache (prime)

Prime+Probe



step 0: attacker fills the cache (prime)

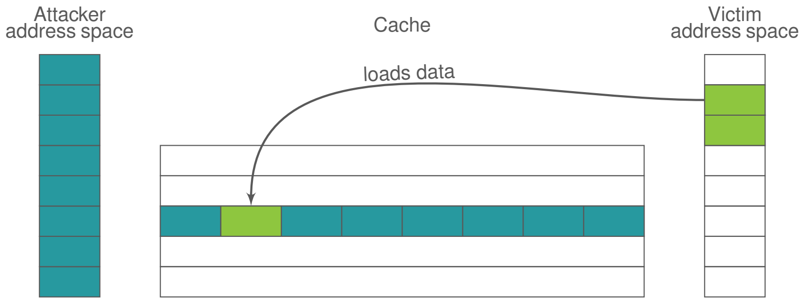
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

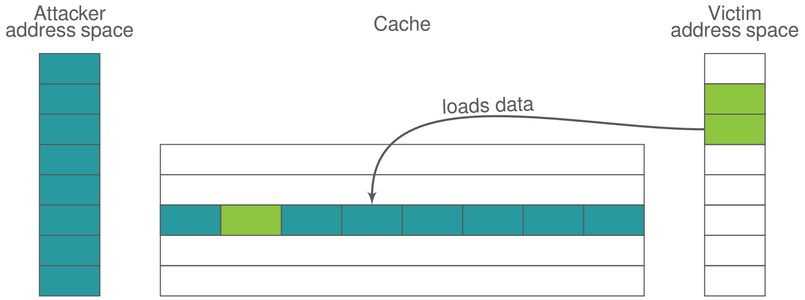
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

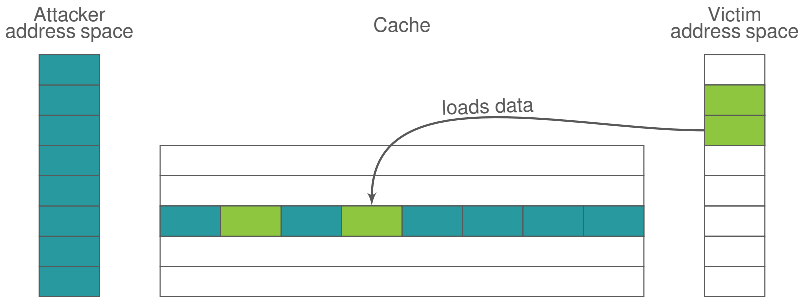
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

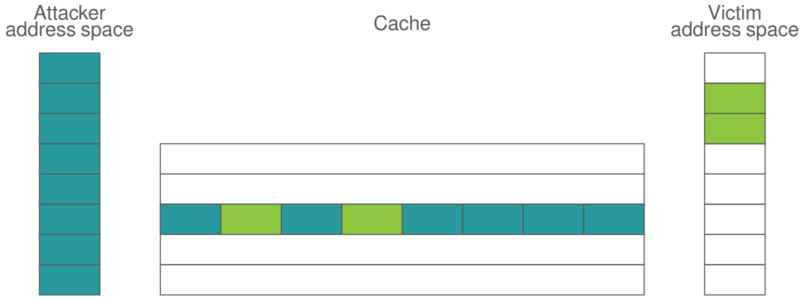
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

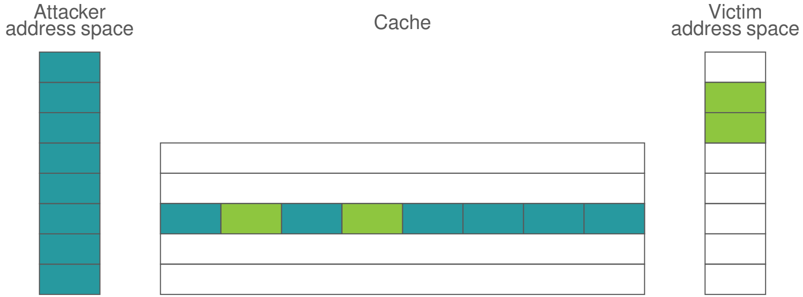
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

Prime+Probe

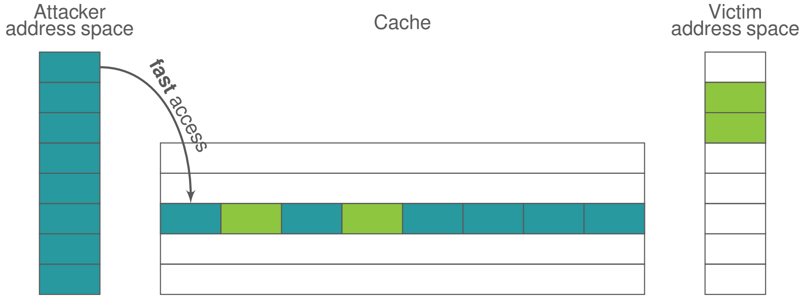


step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

step 2: attacker probes data to determine if the set was accessed

Prime+Probe

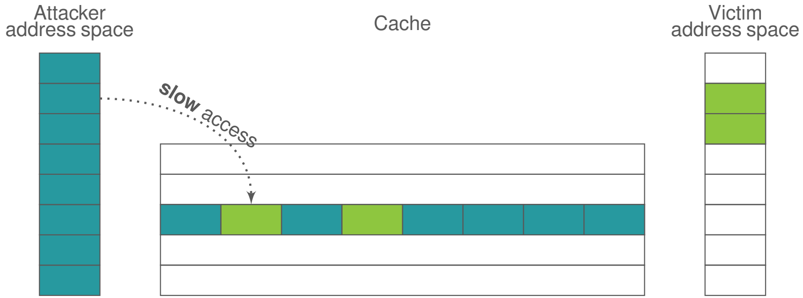


step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

step 2: attacker probes data to determine if the set was accessed

Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

step 2: attacker probes data to determine if the set was accessed

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

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.

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?

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?

Stealthier cache attack: Flush+Flush

- motivation: detecting cache attacks with perf counters is not enough

→ Flush+Flush: new cache attack, based on `clflush` timing leakage

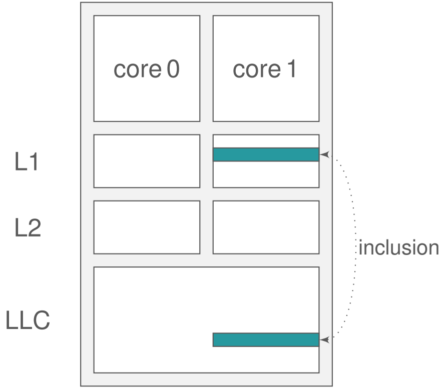
  → stealthier than Prime+Probe and Flush+Reload

  → faster than Prime+Probe and Flush+Reload

D. Gruss, C. Maurice, K. Wagner, and S. Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack". In: DIMVA'16. 2016.

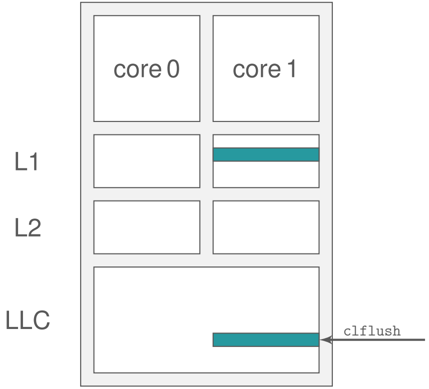
https://github.com/IAIK/flush_flush

`clflush` timing leakage (1)



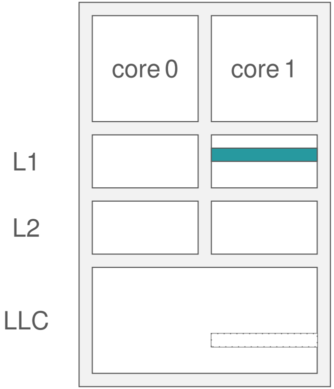
- `clflush` on cached data

`clflush` timing leakage (1)



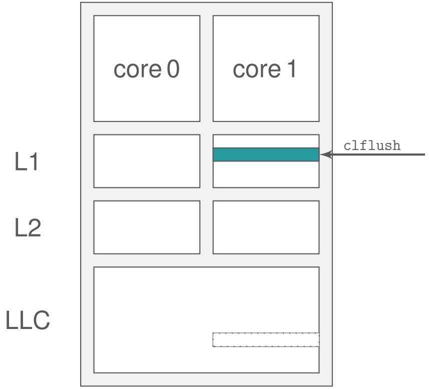
- `clflush` on cached data
  - goes to LLC, flushes line

`clflush` timing leakage (1)



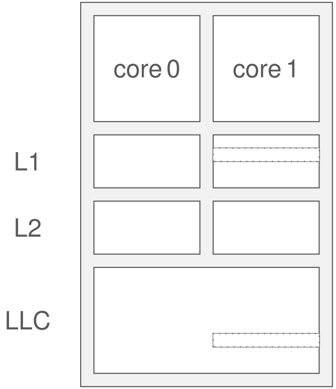
- `clflush` on cached data
    - goes to LLC, flushes line

`clflush` timing leakage (1)



- `clflush` on cached data
    - goes to LLC, flushes line
    - flushes line in L1-L2

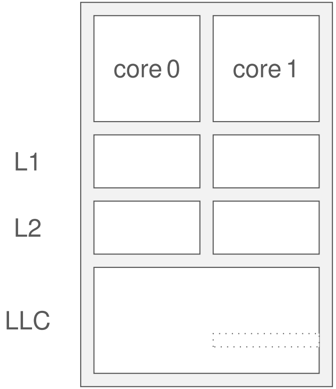
`clflush` timing leakage (1)



- `clflush` on cached data
  - goes to LLC, flushes line
  - flushes line in L1-L2

  $\rightarrow$ slow

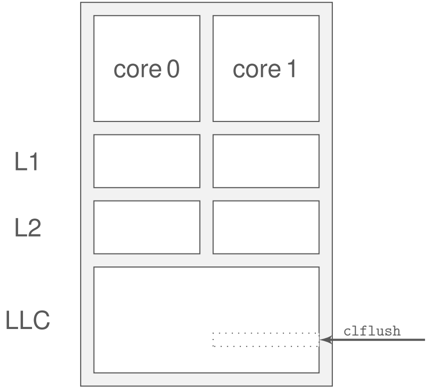
`clflush` timing leakage (1)



- `clflush` on cached data
  - goes to LLC, flushes line
  - flushes line in L1-L2

  $\rightarrow$ slow
- `clflush` on non-cached data

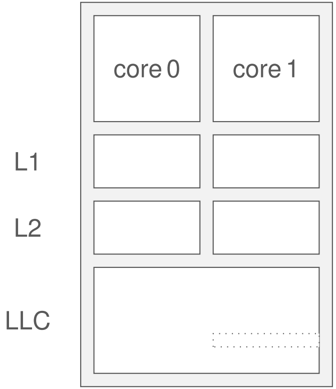
`clflush` timing leakage (1)



- `clflush` on cached data
  - goes to LLC, flushes line
  - flushes line in L1-L2

  → slow
- `clflush` on non-cached data
  - goes to LLC, does nothing

`clflush` timing leakage (1)

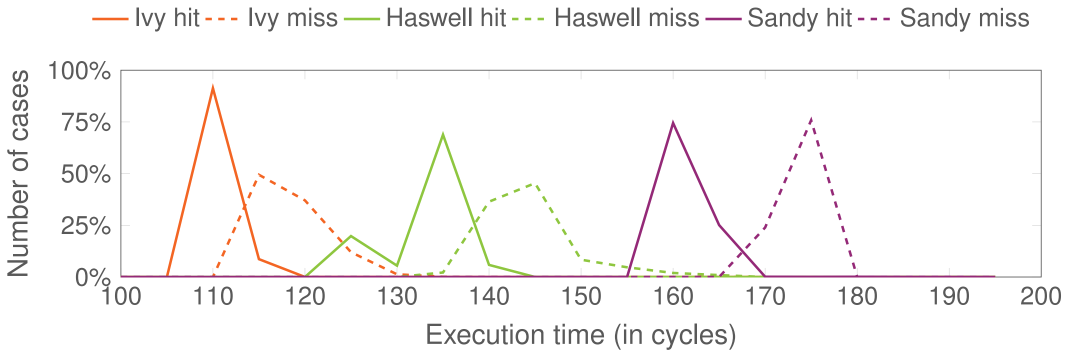


- `clflush` on cached data
  - goes to LLC, flushes line
  - flushes line in L1-L2

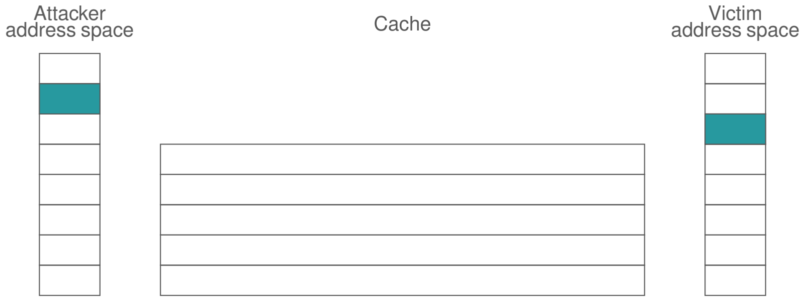
  → slow
- `clflush` on non-cached data
  - goes to LLC, does nothing

  → fast

`clflush` timing leakage (2)

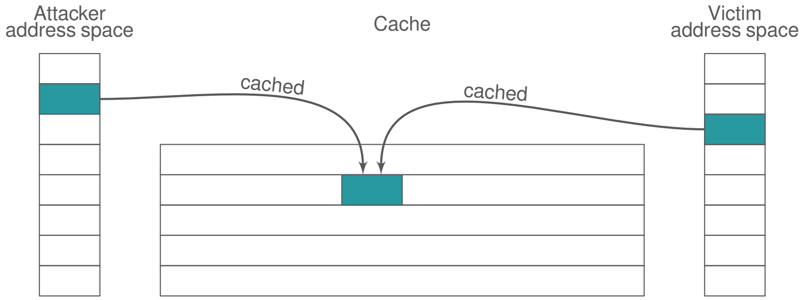


Flush+Reload



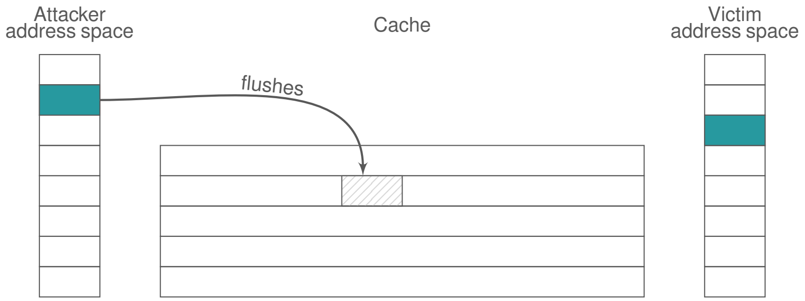
step 0: attacker maps shared library → shared memory, shared in cache

Flush+Reload



step 0: attacker maps shared library → shared memory, shared in cache

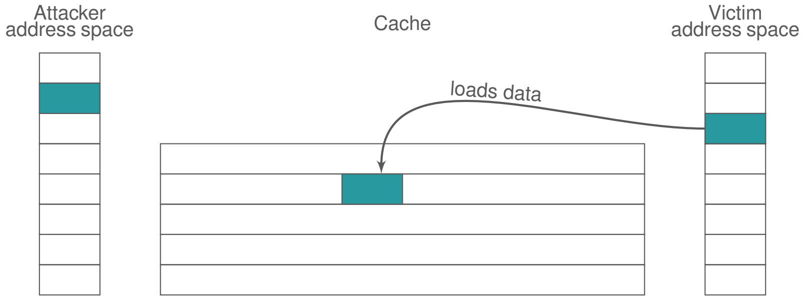
Flush+Reload



step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line with `clflush`

Flush+Reload

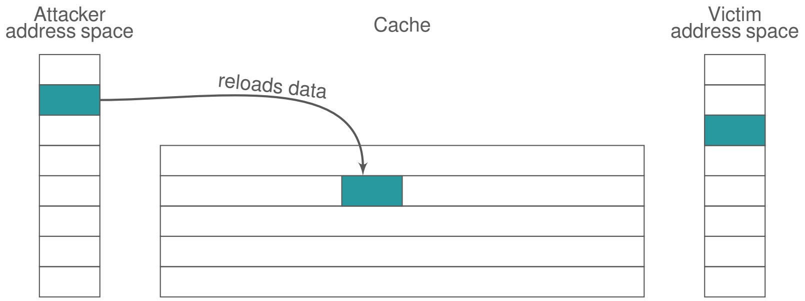


step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line with `clflush`

step 2: victim loads data while performing encryption

Flush+Reload



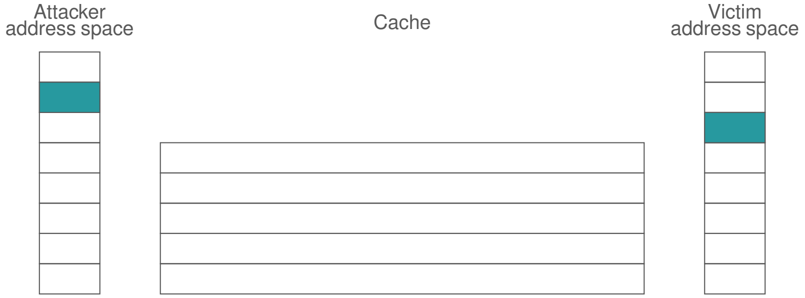
step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line with `clflush`

step 2: victim loads data while performing encryption

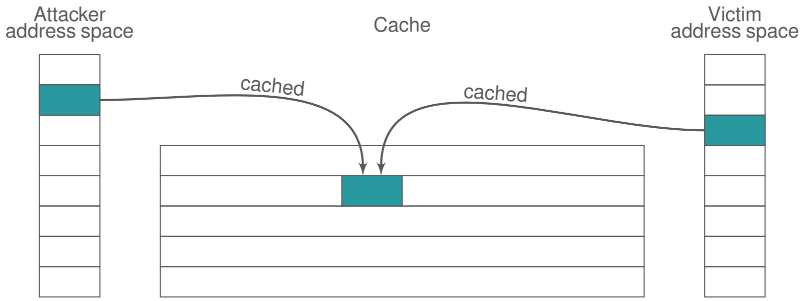
step 3: attacker reloads data → fast access if the victim loaded the line

Flush+Flush



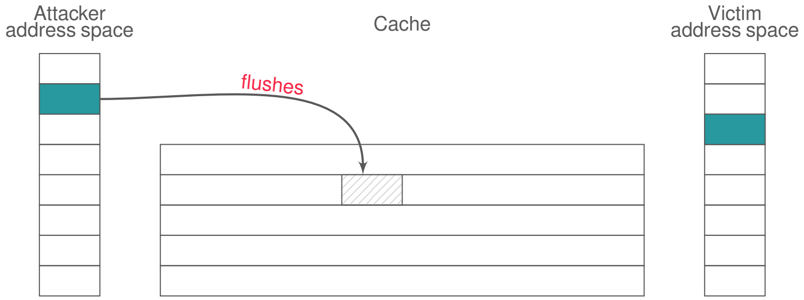
step 0: attacker maps shared library $\rightarrow$ shared memory, shared in cache

Flush+Flush



step 0: attacker maps shared library → shared memory, shared in cache

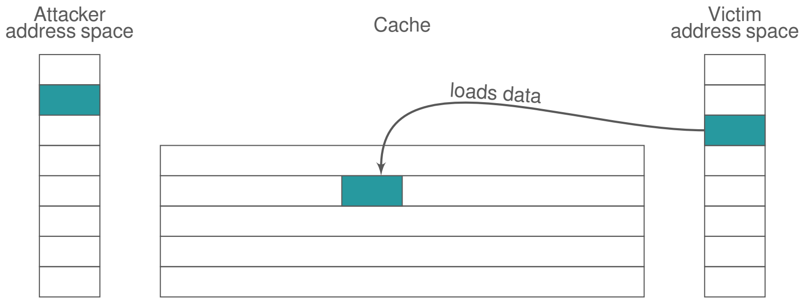
Flush+Flush



step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line

Flush+Flush

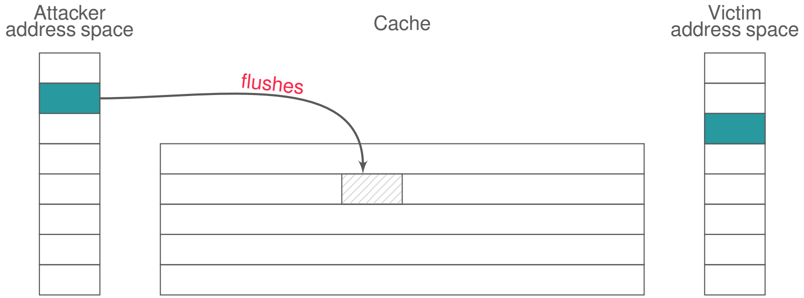


step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line

step 2: victim loads data while performing encryption

Flush+Flush



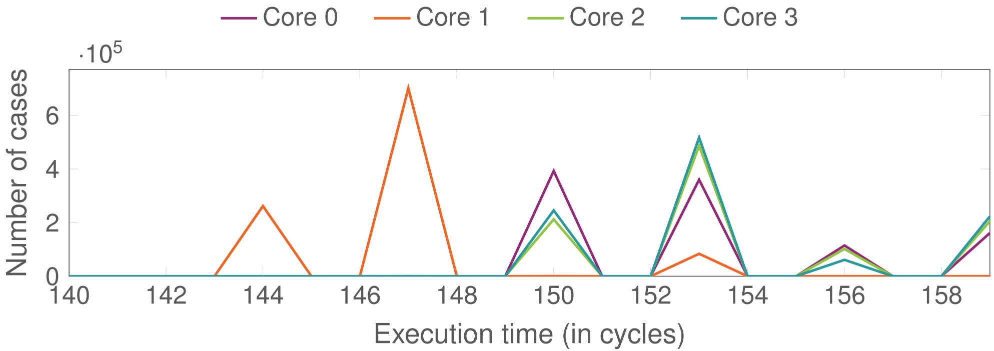
step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line

step 2: victim loads data while performing encryption

step 3: attacker flushes data → high execution time if the victim loaded the line

Even more timing leakage with `clflush`



ARMageddon: Challenges of ARM

1. ARM v7 CPUs have no flush instruction
2. replacement policy is pseudo-random
3. cycle-accurate timings require root
4. last-level caches are not inclusive
5. multiple CPUs do not share a cache

M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. "ARMageddon: Last-Level Cache Attacks on Mobile Devices". In: USENIX Security Symposium. 2016.

ARMageddon

All cache attacks from Intel x86 applicable are to ARM devices

- covert channel up to 1 Mbps

  $\rightarrow$ 2-3 orders of magnitude faster than previous work

- side channels
  - monitor taps and swipe events, keystrokes
  - AES T-table implementation of Bounty Castle 1.5

What about...

... other caches?
Yes, they leak too.

Intel being overspecific

NOTE

Intel being overspecific

NOTE

Using the PREFETCH instruction is **recommended** only if data does not fit in cache.

Intel being overspecific

NOTE

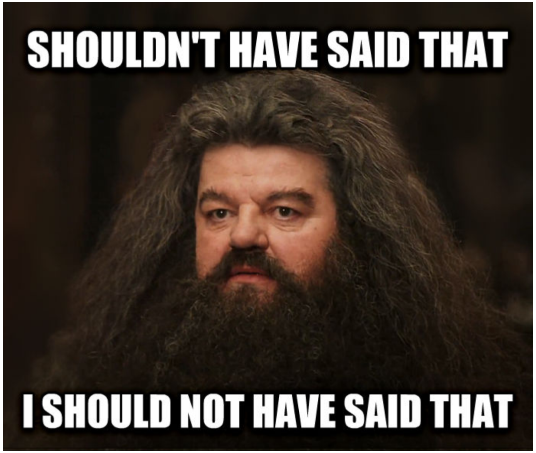
Using the PREFETCH instruction is **recommended** only if data does not fit in cache. Use of software prefetch **should** be limited to memory addresses that are managed or owned within the application context.

Intel being overspecific

NOTE

Using the PREFETCH instruction is **recommended** only if data does not fit in cache. Use of software prefetch **should** be limited to memory addresses that are managed or owned within the application context. Prefetching to addresses that are **not mapped to physical** pages can experience **non-deterministic** performance penalty.

Intel being overspecific

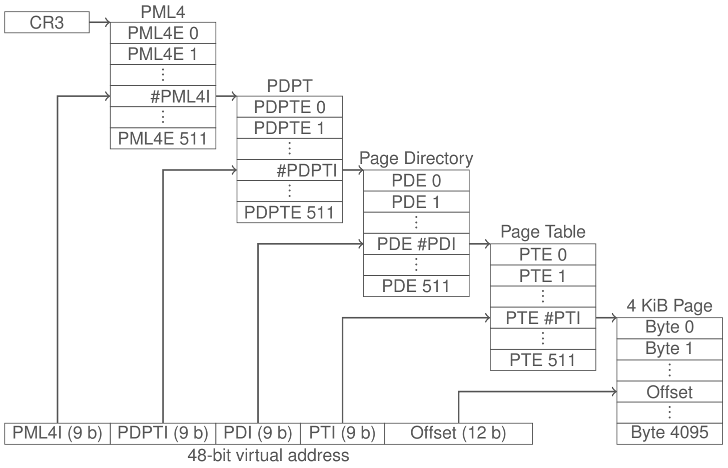


Software prefetching

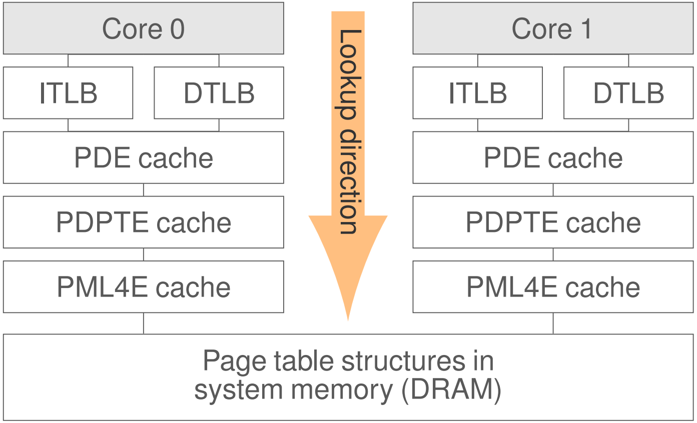
`prefetch` instructions are somewhat unusual

- Hints – can be ignored by the CPU
- Do not check privileges or cause exceptions

Address translation on x86-64

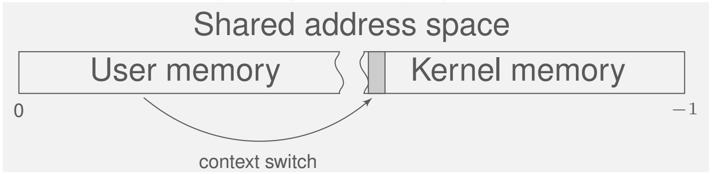


Solution: Address Translation Caches



Kernel is mapped in every process

Today’s operating systems:



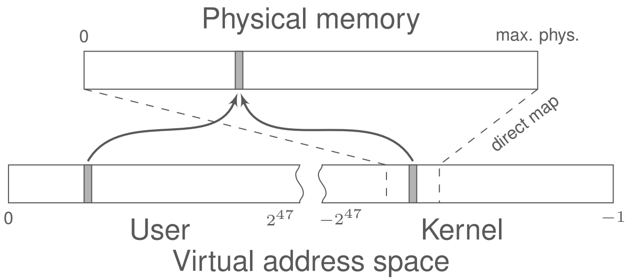
Address-Space Layout Randomization (ASLR)

- Kernel and drivers at randomized offsets in virtual memory
- Mitigates code reuse attacks e.g. return-oriented-programming
- Attacks based on read primitives or write primitives

Address-Space Layout Randomization (ASLR)

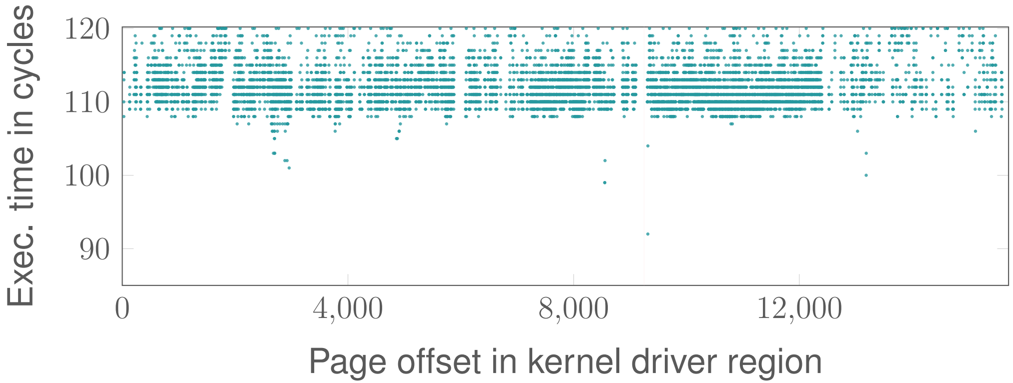
- Kernel and drivers at randomized offsets in virtual memory
- Mitigates code reuse attacks e.g. return-oriented-programming
- Attacks based on read primitives or write primitives
- But: leaking kernel/driver addresses defeats ASLR

Kernel direct-physical map



OS X, Linux, BSD, Xen PVM (Amazon EC2)

Locate Kernel Driver (defeat KASLR)



Defeating SMAP/SMEP

- Get direct-physical-map address of userspace address

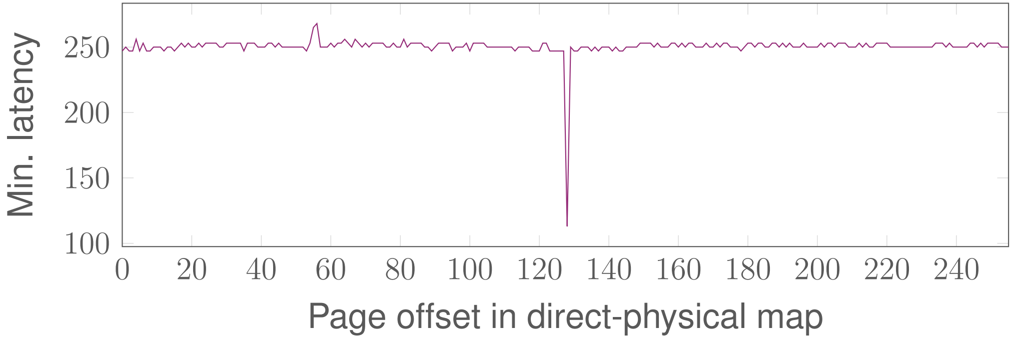
$\rightarrow$ jump there (it's executable)

$\rightarrow$ or: switch to stack there

Known as “ret2dir” attacks

V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. “ret2dir: Rethinking kernel isolation”. In: USENIX Security Symposium. 2014, pp. 957–972.

Prefetching via direct-physical map



Beyond cache attacks

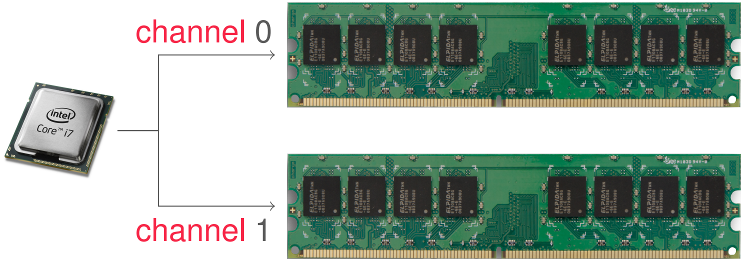
- talking about DRAM:
  - Rowhammer.js
  - DRAM side-channel attacks

D. Gruss, C. Maurice, and S. Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: DIMVA'16. 2016.

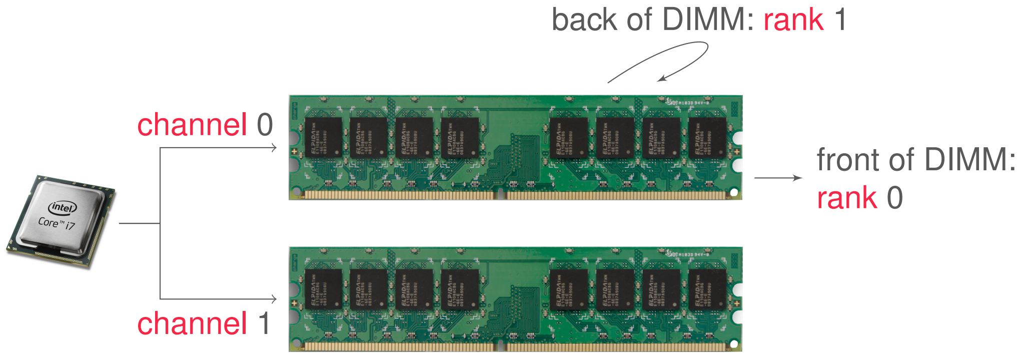
P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: USENIX Security Symposium. 2016.

DRAM organization example

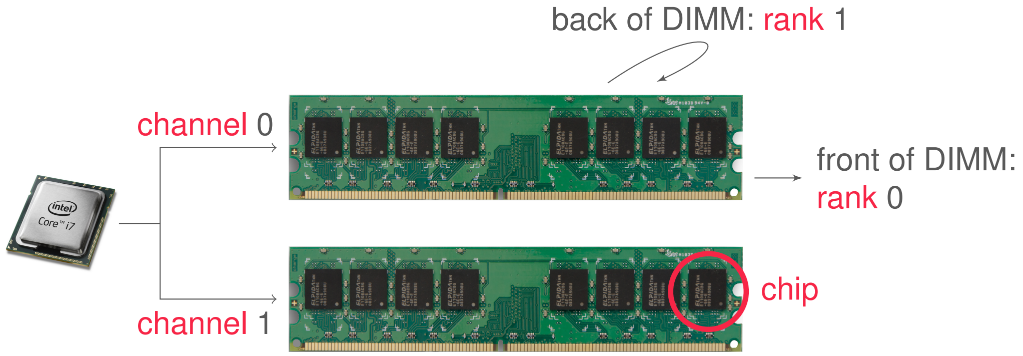
DRAM organization example



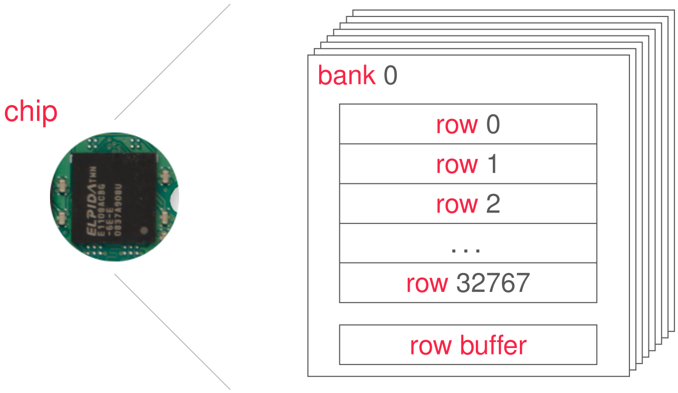
DRAM organization example



DRAM organization example



DRAM organization example



- bits in cells in rows
- access: activate row, copy to row buffer

DRAM refresh

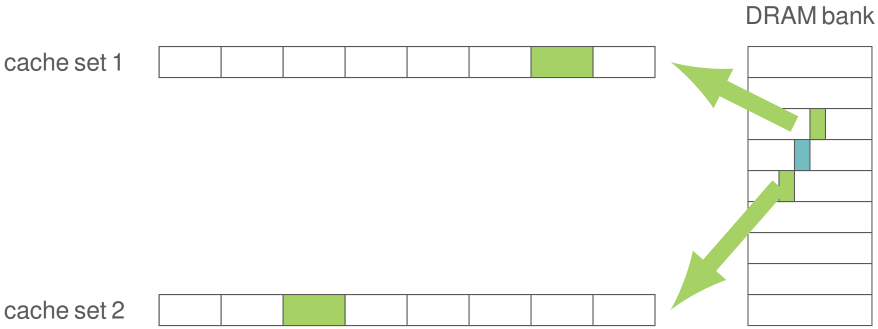
- cells leak $\rightarrow$ repetitive refresh necessary
- refresh $\approx$ reading (destructive) + writing same data again
- maximum interval between refreshes to guarantee data integrity

DRAM refresh

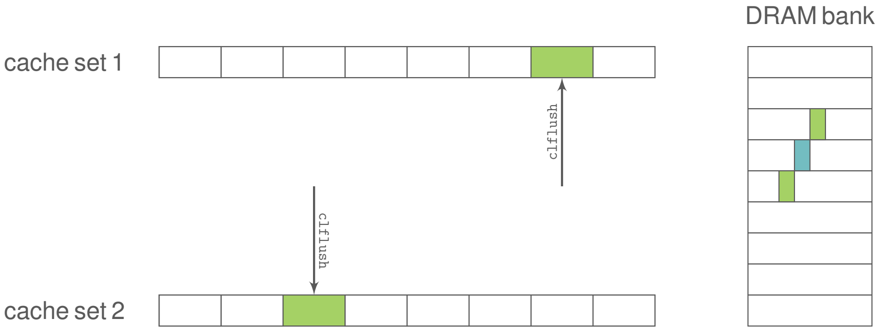
- cells leak $\rightarrow$ repetitive refresh necessary
- refresh $\approx$ reading (destructive) + writing same data again
- maximum interval between refreshes to guarantee data integrity

- cells leak faster upon proximate accesses $\rightarrow$ Rowhammer

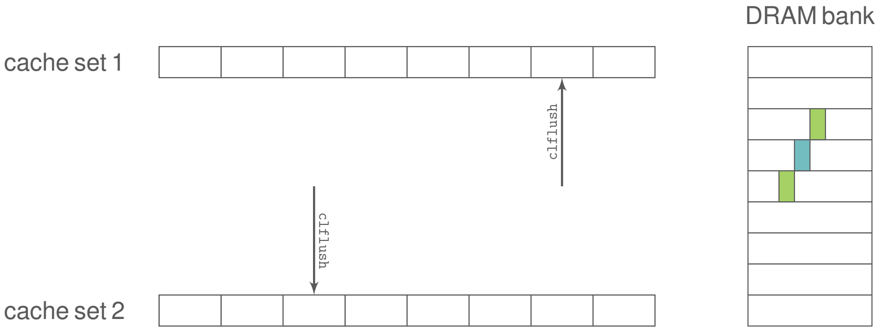
Rowhammer (with `clflush`)



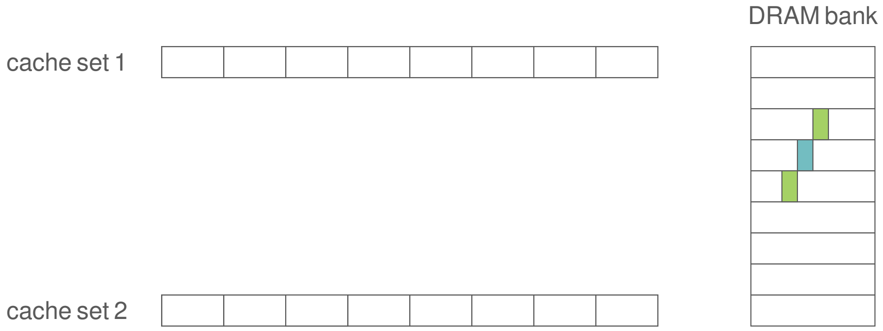
Rowhammer (with `clflush`)



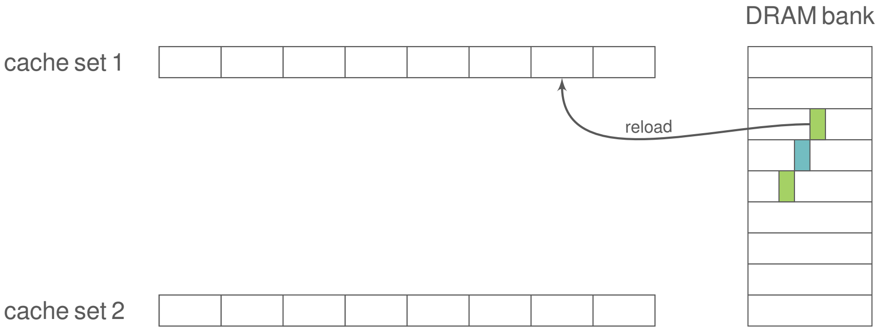
Rowhammer (with `clflush`)



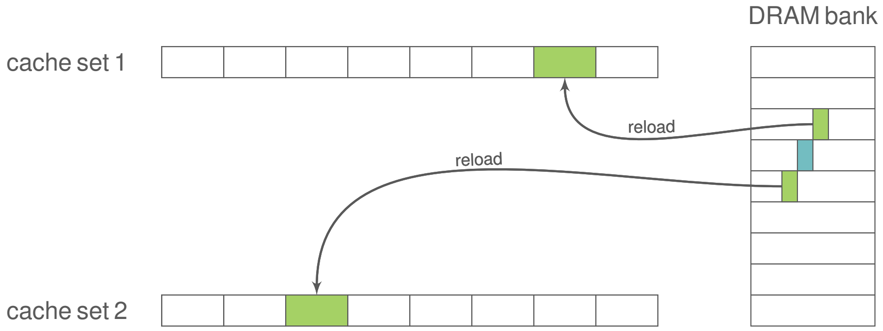
Rowhammer (with `clflush`)



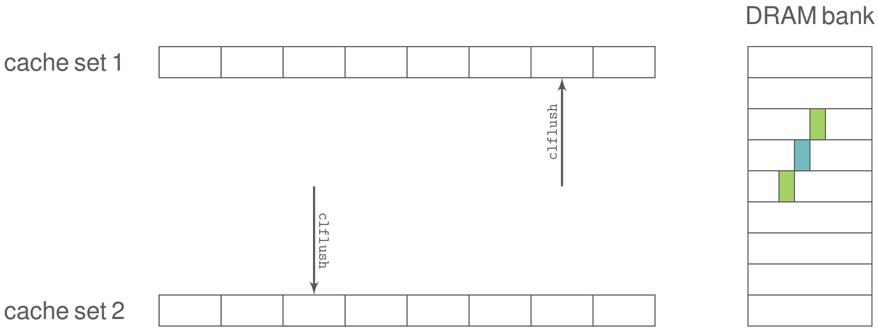
Rowhammer (with `clflush`)



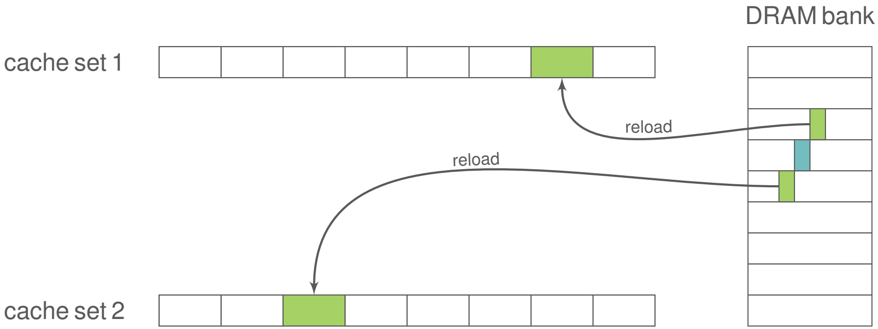
Rowhammer (with `clflush`)



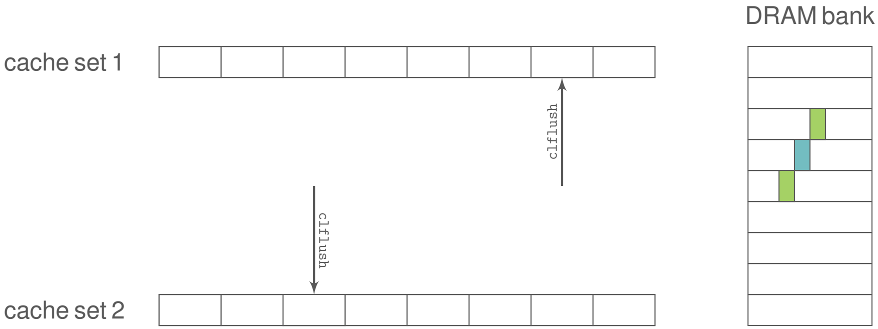
Rowhammer (with `clflush`)



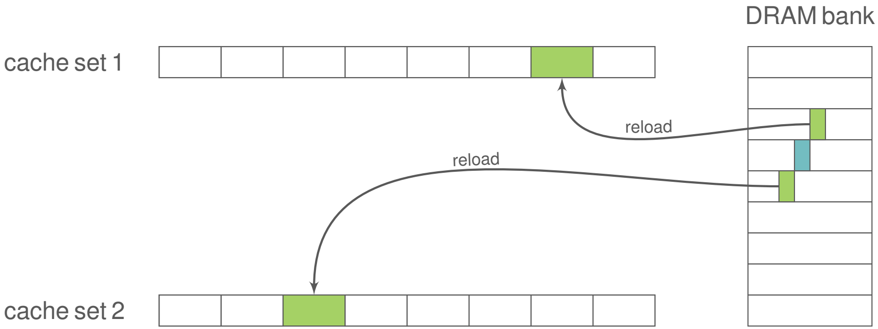
Rowhammer (with `clflush`)



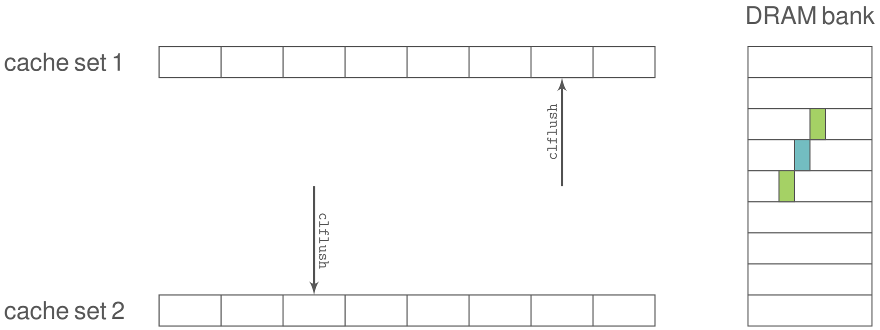
Rowhammer (with `clflush`)



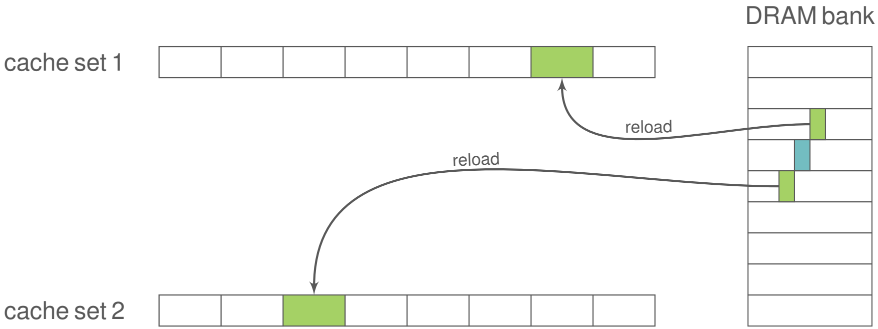
Rowhammer (with `clflush`)



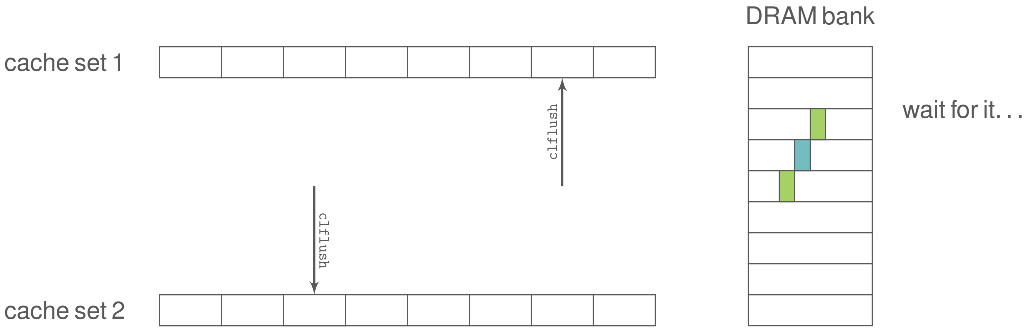
Rowhammer (with `clflush`)



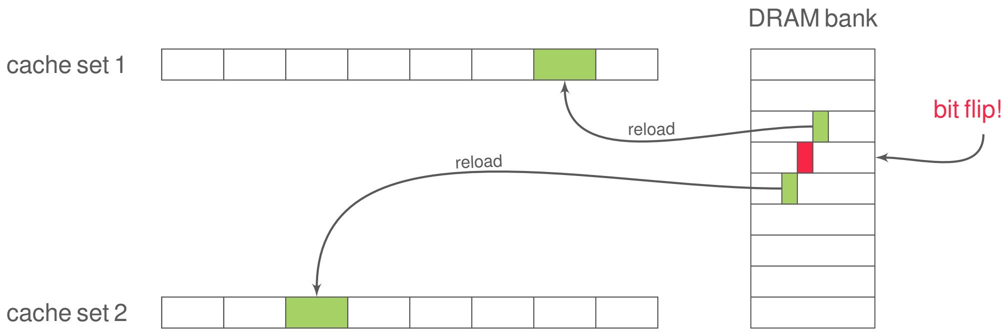
Rowhammer (with `clflush`)



Rowhammer (with `clflush`)



Rowhammer (with `clflush`)



Rowhammer without `clflush`?

- idea: avoid `clflush` to be independent of specific instructions

  $\rightarrow$ no `clflush` in JavaScript

Rowhammer without `clflush`?

- idea: avoid `clflush` to be independent of specific instructions

  $\rightarrow$ no `clflush` in JavaScript

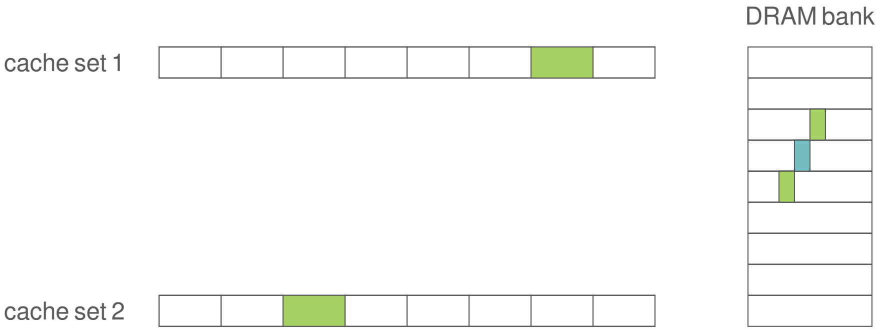
- our approach: use regular memory accesses for eviction

  $\rightarrow$ techniques from cache attacks!

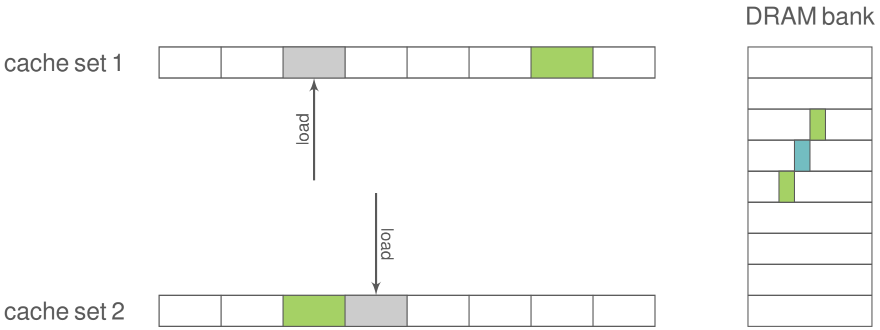
Rowhammer without `clflush`?

- idea: avoid `clflush` to be independent of specific instructions
  - $\rightarrow$ no `clflush` in JavaScript
- our approach: use regular memory accesses for eviction
  - $\rightarrow$ techniques from cache attacks!
  - $\rightarrow$ Rowhammer, Prime+Probe style!

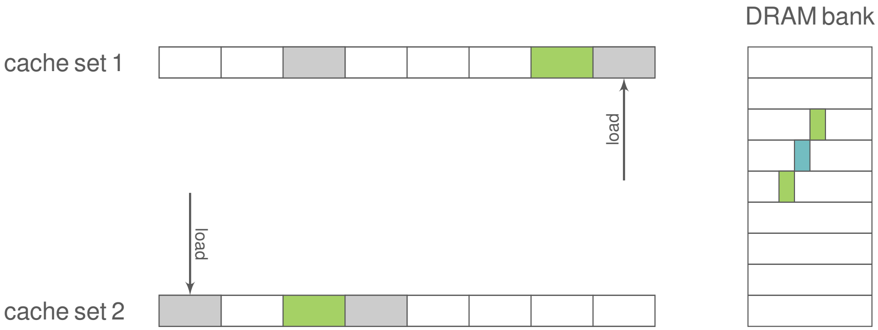
Rowhammer without `clflush`



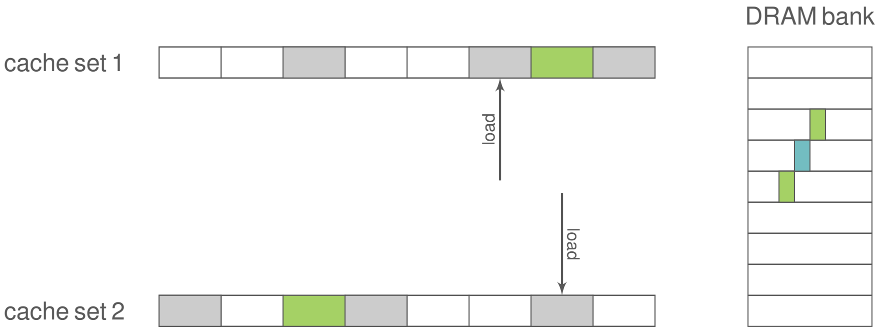
Rowhammer without `clflush`



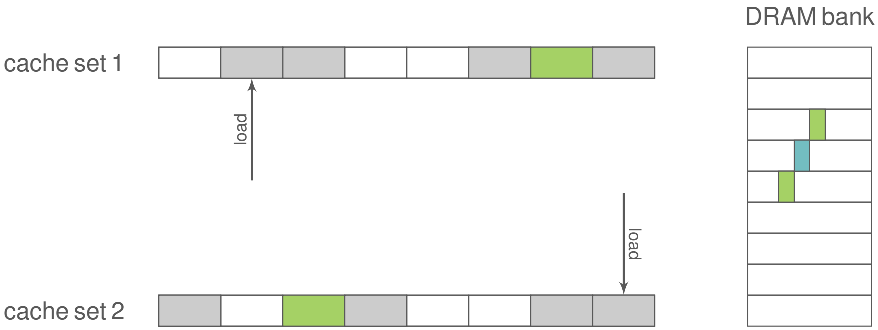
Rowhammer without `clflush`



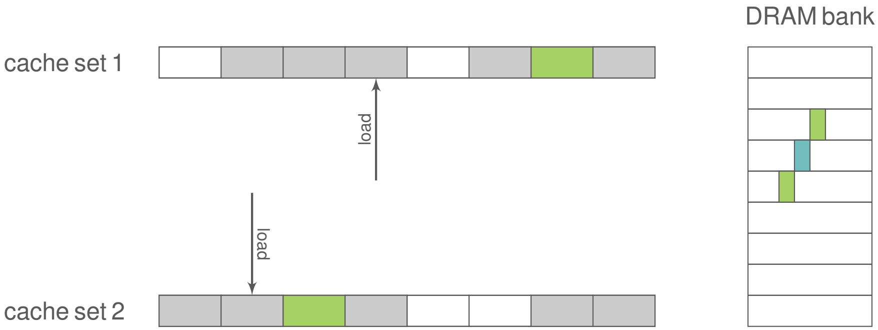
Rowhammer without `clflush`



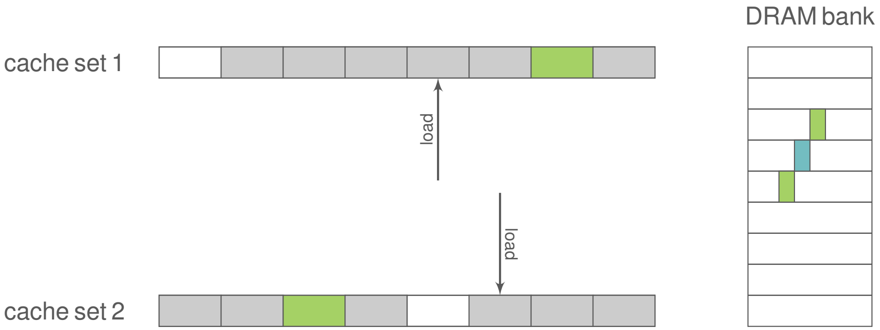
Rowhammer without `clflush`



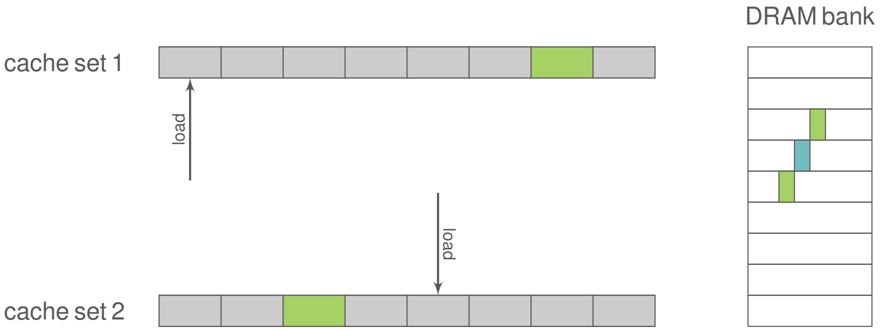
Rowhammer without `clflush`



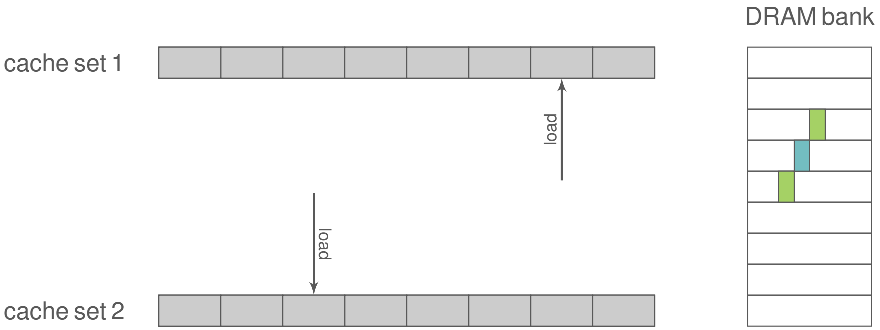
Rowhammer without `clflush`



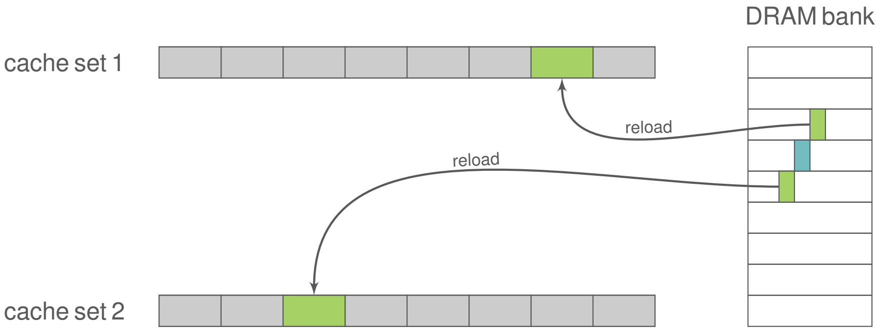
Rowhammer without `clflush`



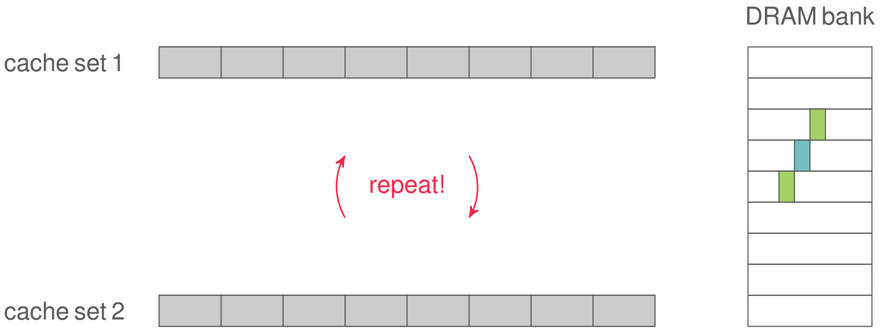
Rowhammer without `clflush`



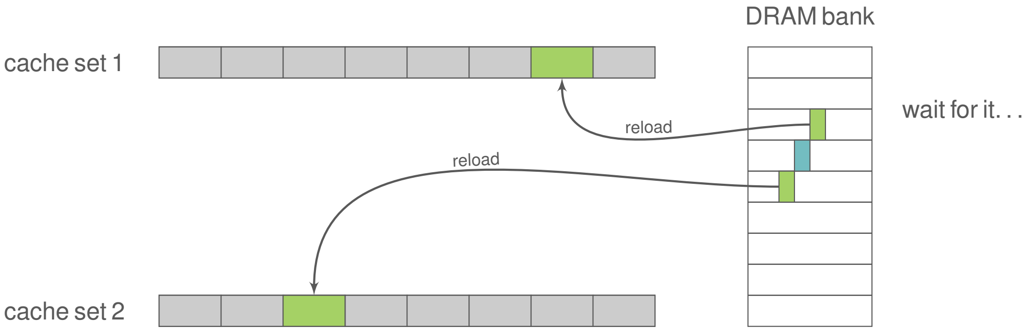
Rowhammer without `clflush`



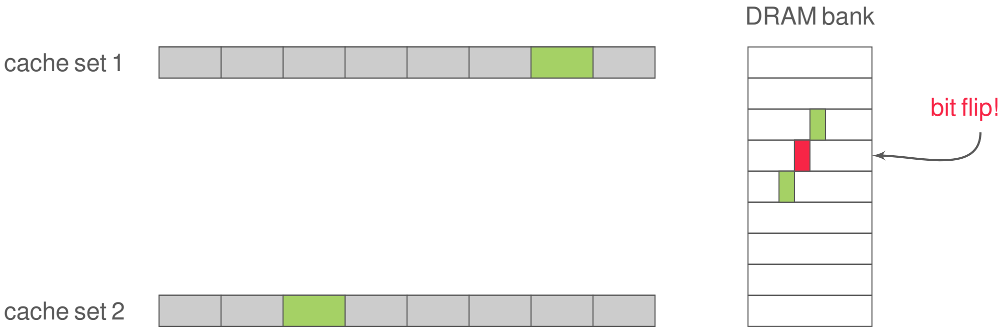
Rowhammer without `clflush`



Rowhammer without `clflush`



Rowhammer without `clflush`



Requirements for Rowhammer

1. uncached memory accesses: need to reach DRAM
2. fast memory accesses: race against the next row refresh

Requirements for Rowhammer

1. uncached memory accesses: need to reach DRAM
2. fast memory accesses: race against the next row refresh

$\rightarrow$ optimize the eviction rate and the timing

Rowhammer.js: the challenges

1. how to get accurate timing in JS?
2. how to get physical addresses in JS?
3. which physical addresses to access?
4. in which order to access them?

Rowhammer.js: the challenges

1. how to get accurate timing in JS? $\rightarrow$ easy
2. how to get physical addresses in JS?
3. which physical addresses to access?
4. in which order to access them?

Rowhammer.js: the challenges

1. how to get accurate timing in JS? $\rightarrow$ easy
2. how to get physical addresses in JS? $\rightarrow$ we solved this
3. which physical addresses to access?
4. in which order to access them?

Rowhammer.js: the challenges

1. how to get accurate timing in JS? $\rightarrow$ easy
2. how to get physical addresses in JS? $\rightarrow$ we solved this
3. which physical addresses to access? $\rightarrow$ we solved this
4. in which order to access them?

Rowhammer.js: the challenges

1. how to get accurate timing in JS? $\rightarrow$ easy
2. how to get physical addresses in JS? $\rightarrow$ we solved this
3. which physical addresses to access? $\rightarrow$ we solved this
4. in which order to access them? $\rightarrow$ we solved this

How to get accurate timing in JavaScript?

- native code: `rdtsc`
- JavaScript: `window.performance.now()`

How to get accurate timing in JavaScript?

- native code: `rdtsc`
- JavaScript: `window.performance.now()`

- recent patch: time rounded to 5 microseconds
- still works: we measure millions of accesses

Physical addresses and DRAM

- fixed map: physical addresses $\rightarrow$ DRAM cells
- undocumented for Intel
- reverse-engineering for Sandy Bridge
- and by us for Sandy, Ivy, Haswell, Skylake,. . .

M. Seaborn. How physical addresses map to rows and banks in DRAM. . http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html. Retrieved on July 20, 2015. 2015.

P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: USENIX Security Symposium. 2016.

Physical addresses and JavaScript

- OS optimization: use 2MB pages
- last 21 bits (2MB) of physical address
- = last 21 bits (2MB) of virtual address

D. Gruss, D. Bidner, and S. Mangard. “Practical Memory Deduplication Attacks in Sandboxed Javascript”. In: ESORICS’15. 2015.

Physical addresses and JavaScript

- OS optimization: use 2MB pages
- last 21 bits (2MB) of physical address
- = last 21 bits (2MB) of virtual address
- = last 21 bits (2MB) of JS array indices

D. Gruss, D. Bidner, and S. Mangard. “Practical Memory Deduplication Attacks in Sandboxed Javascript”. In: ESORICS’15. 2015.

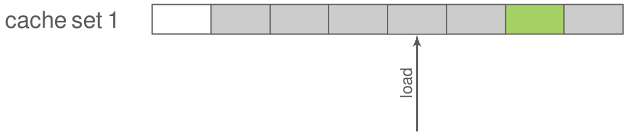
Physical addresses and JavaScript

- OS optimization: use 2MB pages
- last 21 bits (2MB) of physical address
- = last 21 bits (2MB) of virtual address
- = last 21 bits (2MB) of JS array indices

- several DRAM rows per 2MB page
- several congruent addresses per 2MB page

D. Gruss, D. Bidner, and S. Mangard. “Practical Memory Deduplication Attacks in Sandboxed Javascript”. In: ESORICS'15. 2015.

Which physical addresses to access?



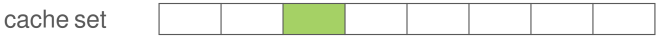
“LRU eviction”:

- assume that cache uses LRU replacement
- accessing n addresses from the same cache set to evict an n-way set
- using the reverse-engineered last-level cache addressing function

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

Replacement policy on older CPUs

"LRU eviction" memory accesses



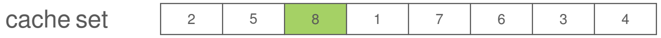
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first

Replacement policy on older CPUs

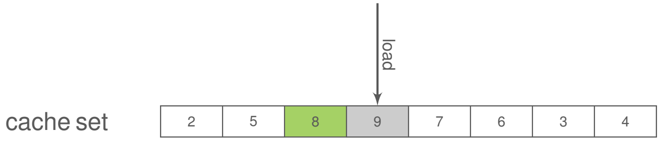
"LRU eviction" memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line

Replacement policy on older CPUs

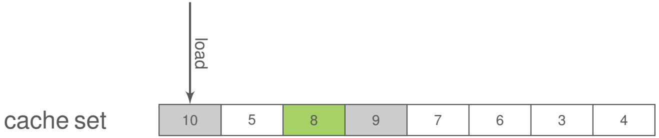
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

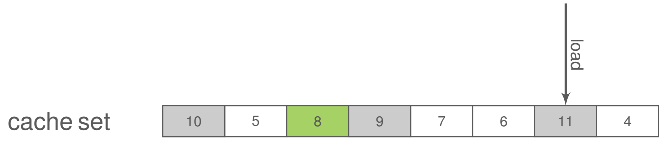
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

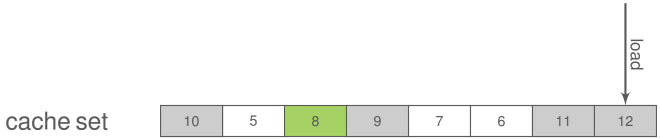
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

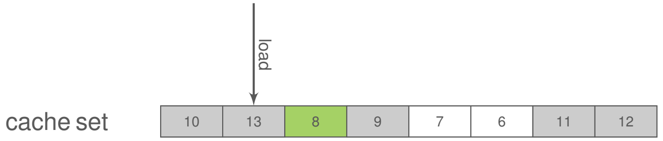
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

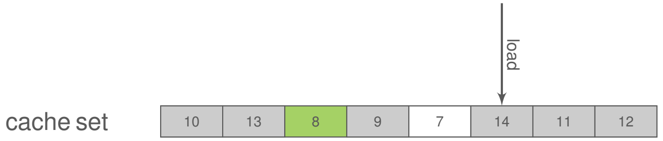
"LRU eviction" memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

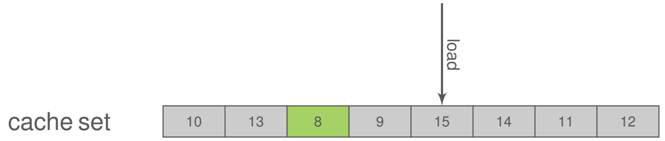
"LRU eviction" memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

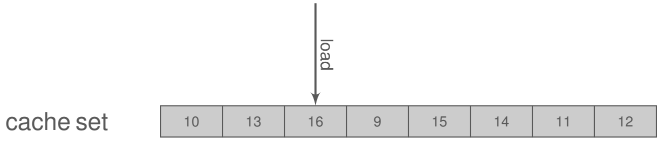
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

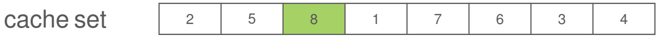
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on recent CPUs

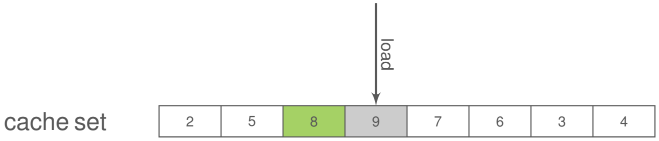
“LRU eviction” memory accesses



- no LRU replacement on recent CPUs

Replacement policy on recent CPUs

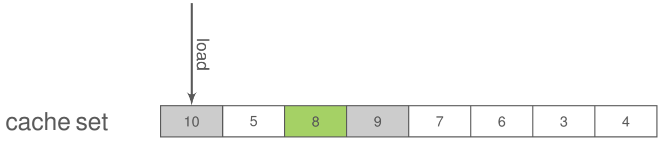
“LRU eviction” memory accesses



- no LRU replacement on recent CPUs

Replacement policy on recent CPUs

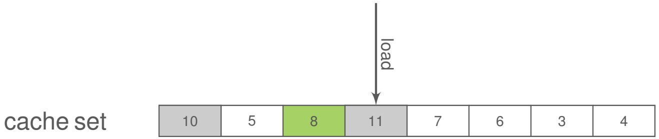
“LRU eviction” memory accesses



- no LRU replacement on recent CPUs

Replacement policy on recent CPUs

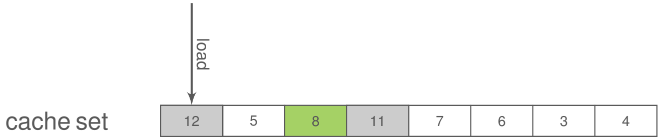
"LRU eviction" memory accesses



- no LRU replacement on recent CPUs

Replacement policy on recent CPUs

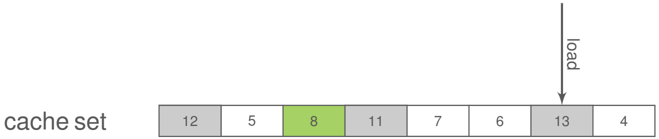
“LRU eviction” memory accesses



- no LRU replacement on recent CPUs

Replacement policy on recent CPUs

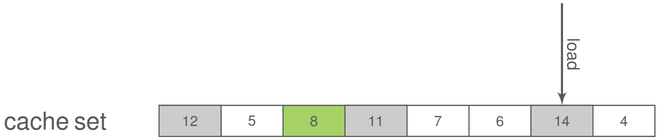
“LRU eviction” memory accesses



- no LRU replacement on recent CPUs

Replacement policy on recent CPUs

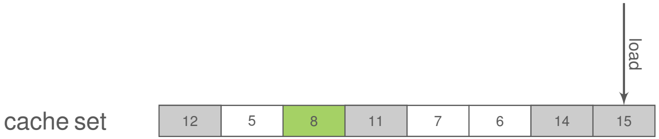
“LRU eviction” memory accesses



- no LRU replacement on recent CPUs

Replacement policy on recent CPUs

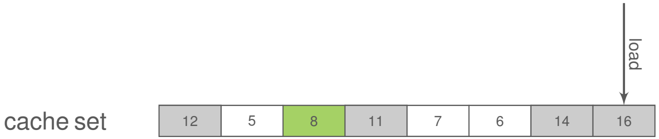
“LRU eviction” memory accesses



- no LRU replacement on recent CPUs

Replacement policy on recent CPUs

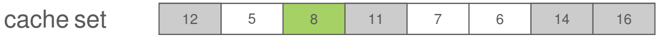
“LRU eviction” memory accesses



- no LRU replacement on recent CPUs

Replacement policy on recent CPUs

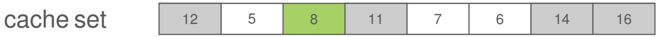
“LRU eviction” memory accesses



- no LRU replacement on recent CPUs
- only 75% success rate on Haswell

Replacement policy on recent CPUs

"LRU eviction" memory accesses



- no LRU replacement on recent CPUs
- only 75% success rate on Haswell
- more accesses → higher success rate, but too slow

Cache eviction strategy: Notation (1)

Write eviction strategies as: $\mathcal{P}$-C-D-L-S

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

Cache eviction strategy: Notation (1)

Write eviction strategies as: $\mathcal{P}$-C-D-L-S

S: total number of different addresses (= set size)

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

Cache eviction strategy: Notation (1)

Write eviction strategies as: $\mathcal{P}$-C-D-L-S

S: total number of different addresses (= set size)

D: different addresses per inner access loop

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

Cache eviction strategy: Notation (1)

Write eviction strategies as: $\mathcal{P}$-C-D-L-S

S: total number of different addresses (= set size)

D: different addresses per inner access loop

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

L: step size of the inner access loop

Cache eviction strategy: Notation (1)

Write eviction strategies as: $\mathcal{P}$-C-D-L-S

S: total number of different addresses (= set size)

D: different addresses per inner access loop

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

L: step size of the inner access loop

C: number of repetitions of the inner access loop

Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S - D ; s += L )
  for (c = 1; c <= C ; c += 1)
    for (d = 1; d <= D ; d += 1)
      *a[s+d];
```

Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S - D ; s += L )
  for (c = 1; c <= C ; c += 1)
    for (d = 1; d <= D ; d += 1)
      *a[s+d];
```

- $\mathcal{P}$-2-2-1-4 $\rightarrow 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4$

Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S-D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```

- $\mathcal{P}$-2-2-1-4 $\rightarrow 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4$ $\leftarrow$ $S = 4$

Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S - D ; s += L )
  for (c = 1; c <= C ; c += 1)
    for (d = 1; d <= D ; d += 1)
      *a[s+d];
```

- $\mathcal{P}$-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4 $\quad S = 4$

Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S-D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```

- $\mathcal{P}$-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4 $\quad S = 4 \quad D = 2$

Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S-D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```

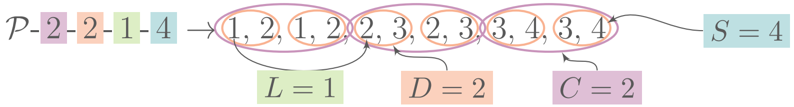
- $\mathcal{P}$-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4

$S = 4$

$D = 2$ $C = 2$

Cache eviction strategy: Notation (2)

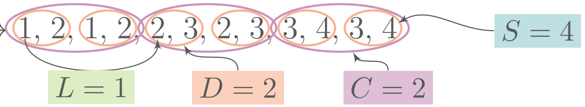
```
for (s = 0; s <= S-D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```


$\mathcal{P}$-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4
$L = 1$
$D = 2$
$C = 2$
$S = 4$

Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S-D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```

- $\mathcal{P}$-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4



- $\mathcal{P}$-1-1-1-4 $\rightarrow 1, 2, 3, 4 \rightarrow$ LRU eviction with set size 4

Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

strategy	# accesses	eviction rate	loop time
$\mathcal{P}$-1-1-1-17	17		
$\mathcal{P}$-1-1-1-20	20		

Executed in a loop, on a Haswell with a 16-way last-level cache

Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

strategy	# accesses	eviction rate	loop time
$\mathcal{P}$-1-1-1-17	17	74.46% ✗	
$\mathcal{P}$-1-1-1-20	20	99.82% ✓	

Executed in a loop, on a Haswell with a 16-way last-level cache

Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

strategy	# accesses	eviction rate	loop time
$\mathcal{P}$-1-1-1-17	17	74.46% ✗	307 ns ✓
$\mathcal{P}$-1-1-1-20	20	99.82% ✓	934 ns ✗

Executed in a loop, on a Haswell with a 16-way last-level cache

Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

strategy	# accesses	eviction rate	loop time
$\mathcal{P}$-1-1-1-17	17	74.46% ✗	307 ns ✓
$\mathcal{P}$-1-1-1-20	20	99.82% ✓	934 ns ✗
$\mathcal{P}$-2-1-1-17	34		

Executed in a loop, on a Haswell with a 16-way last-level cache

Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

strategy	# accesses	eviction rate	loop time
$\mathcal{P}$-1-1-1-17	17	74.46% ✗	307 ns ✓
$\mathcal{P}$-1-1-1-20	20	99.82% ✓	934 ns ✗
$\mathcal{P}$-2-1-1-17	34	99.86% ✓	

Executed in a loop, on a Haswell with a 16-way last-level cache

Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

strategy	# accesses	eviction rate	loop time
$\mathcal{P}$-1-1-1-17	17	74.46% ✗	307 ns ✓
$\mathcal{P}$-1-1-1-20	20	99.82% ✓	934 ns ✗
$\mathcal{P}$-2-1-1-17	34	99.86% ✓	191 ns ✓

Executed in a loop, on a Haswell with a 16-way last-level cache

Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

strategy	# accesses	eviction rate	loop time
$\mathcal{P}$-1-1-1-17	17	74.46% ✗	307 ns ✓
$\mathcal{P}$-1-1-1-20	20	99.82% ✓	934 ns ✗
$\mathcal{P}$-2-1-1-17	34	99.86% ✓	191 ns ✓
$\mathcal{P}$-2-2-1-17	64		

Executed in a loop, on a Haswell with a 16-way last-level cache

Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

strategy	# accesses	eviction rate	loop time
$\mathcal{P}$-1-1-1-17	17	74.46% ✗	307 ns ✓
$\mathcal{P}$-1-1-1-20	20	99.82% ✓	934 ns ✗
$\mathcal{P}$-2-1-1-17	34	99.86% ✓	191 ns ✓
$\mathcal{P}$-2-2-1-17	64	99.98% ✓	

Executed in a loop, on a Haswell with a 16-way last-level cache

Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

strategy	# accesses	eviction rate	loop time
$\mathcal{P}$-1-1-1-17	17	74.46% ✗	307 ns ✓
$\mathcal{P}$-1-1-1-20	20	99.82% ✓	934 ns ✗
$\mathcal{P}$-2-1-1-17	34	99.86% ✓	191 ns ✓
$\mathcal{P}$-2-2-1-17	64	99.98% ✓	180 ns ✓

Executed in a loop, on a Haswell with a 16-way last-level cache

Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

strategy	# accesses	eviction rate	loop time
$\mathcal{P}$-1-1-1-17	17	74.46% ✗	307 ns ✓
$\mathcal{P}$-1-1-1-20	20	99.82% ✓	934 ns ✗
$\mathcal{P}$-2-1-1-17	34	99.86% ✓	191 ns ✓
$\mathcal{P}$-2-2-1-17	64	99.98% ✓	180 ns ✓

$\rightarrow$ more accesses, smaller execution time?

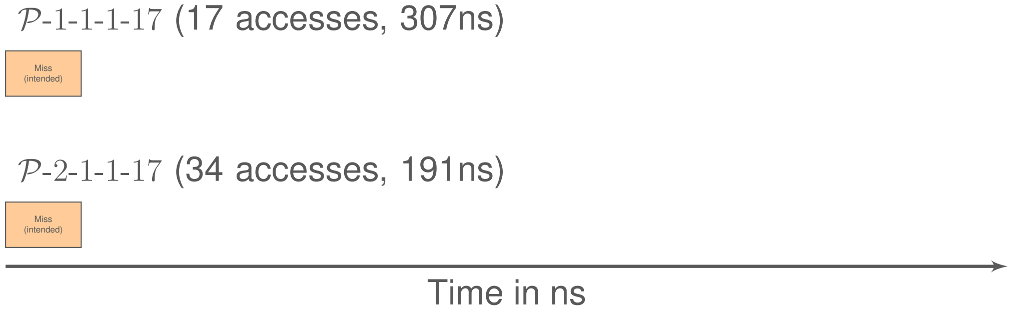
Executed in a loop, on a Haswell with a 16-way last-level cache

Cache eviction strategies: Illustration

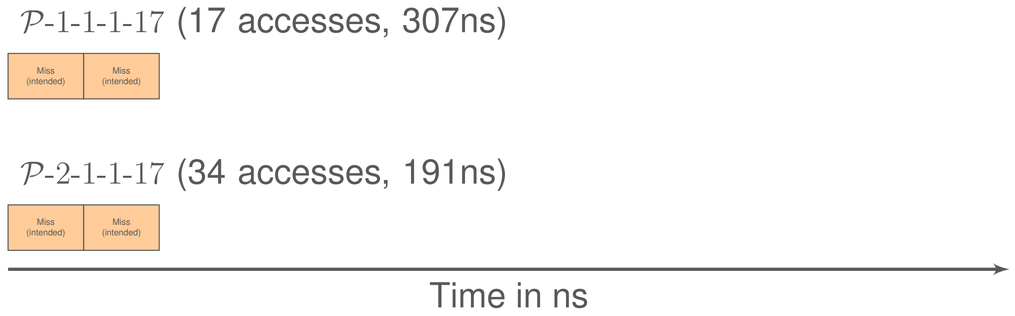
$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

Cache eviction strategies: Illustration


$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)
Miss (intended)
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)
Miss (intended)
Time in ns

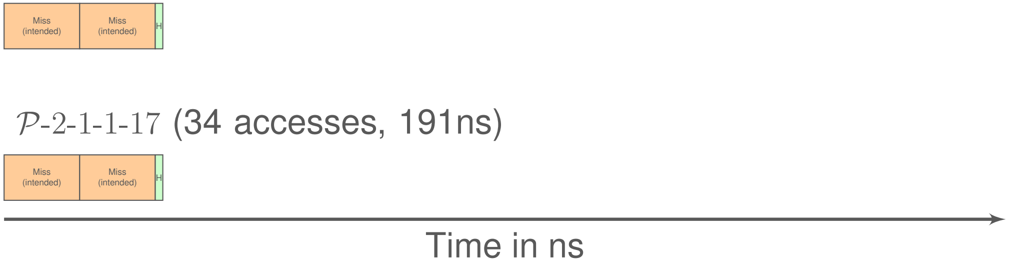
Cache eviction strategies: Illustration


$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)
Miss (intended)
Miss (intended)
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)
Miss (intended)
Miss (intended)
Time in ns

Cache eviction strategies: Illustration

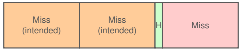
$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

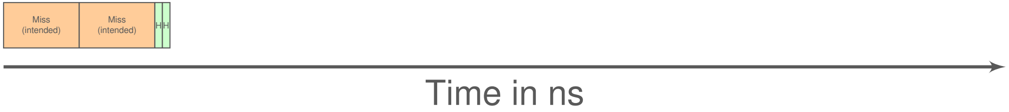


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

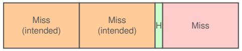


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

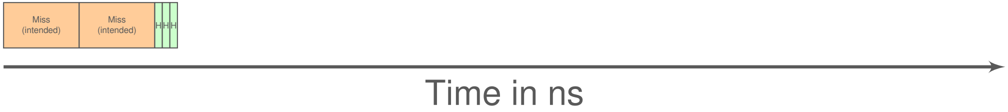


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

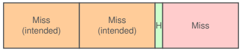


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

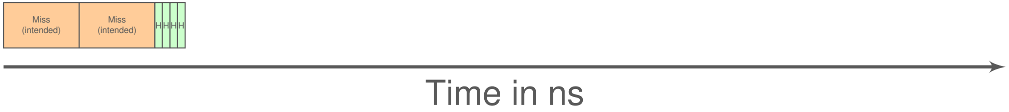


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

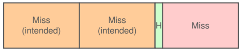


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

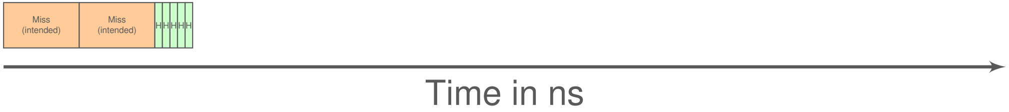


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)



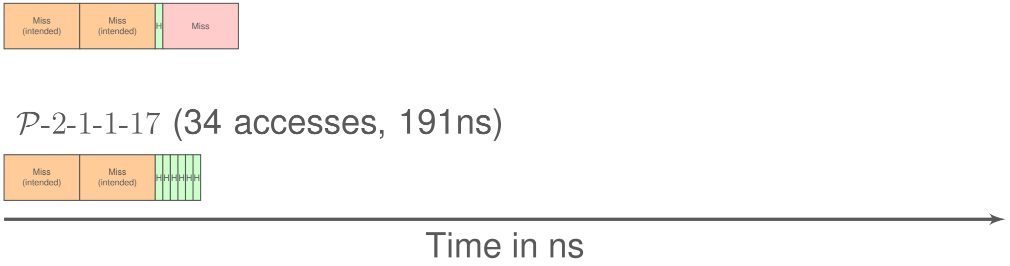
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

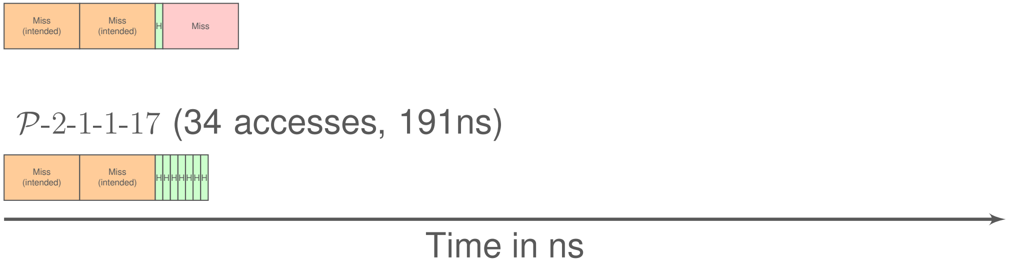
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



Cache eviction strategies: Illustration

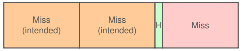
$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

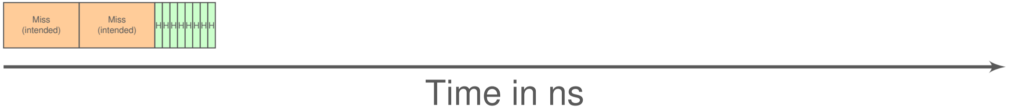


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

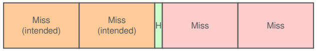


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

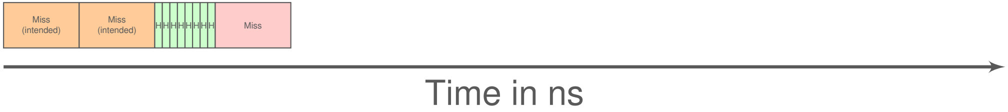


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

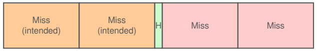


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

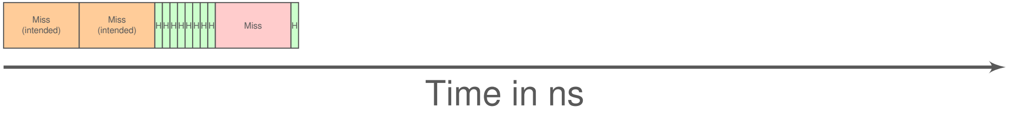


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)



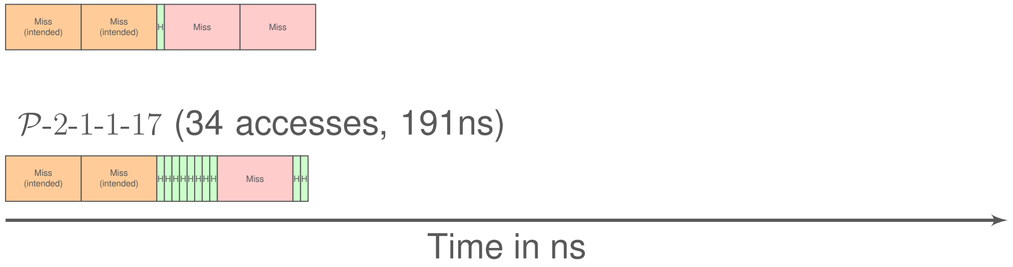
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

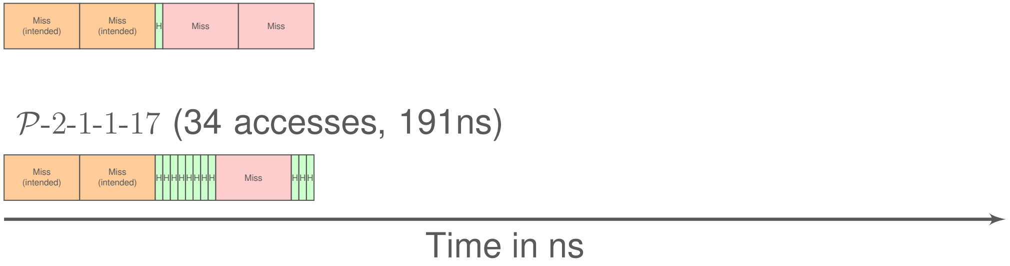
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



Cache eviction strategies: Illustration

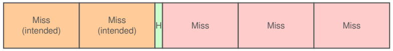
$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

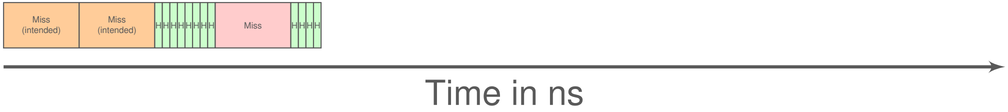


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

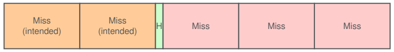


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

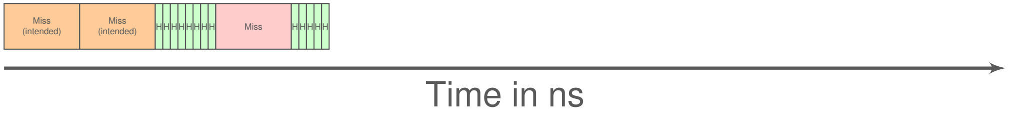


Cache eviction strategies: Illustration

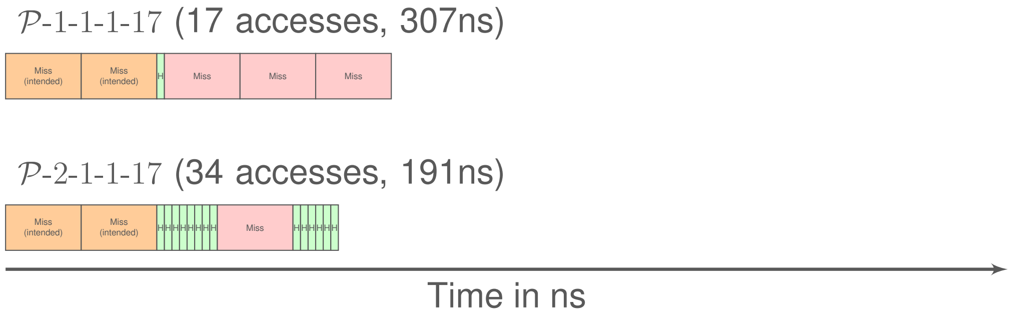
$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)



$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

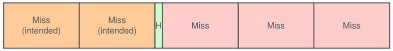


Cache eviction strategies: Illustration

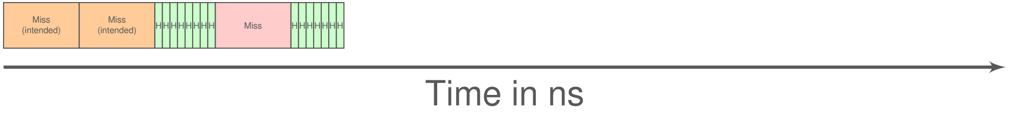


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

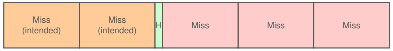


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

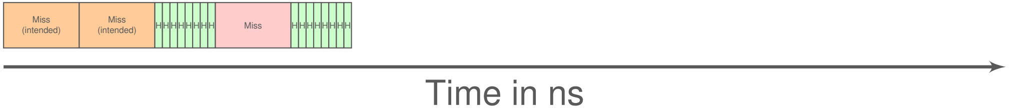


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

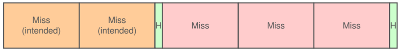


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

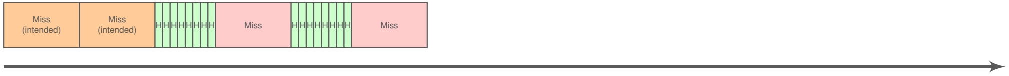


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)



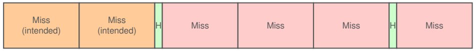
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



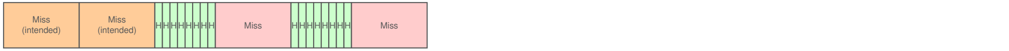
Time in ns

Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)



$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

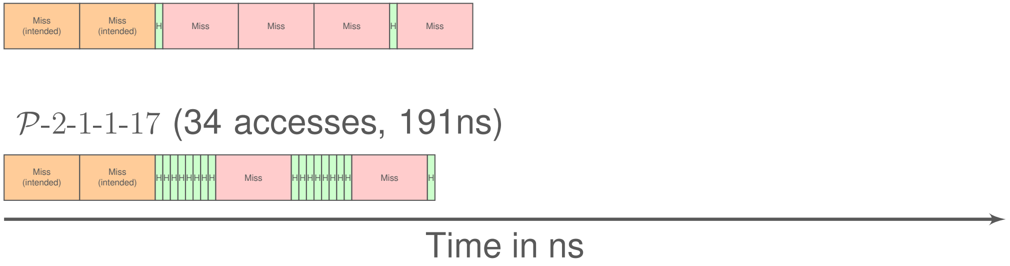


Time in ns

Cache eviction strategies: Illustration

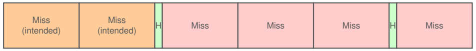
$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

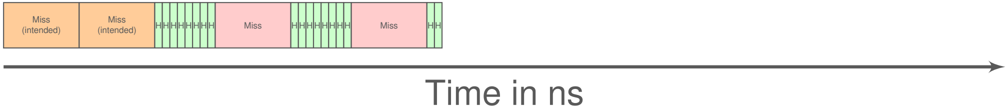


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

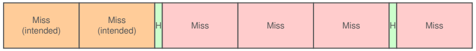


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

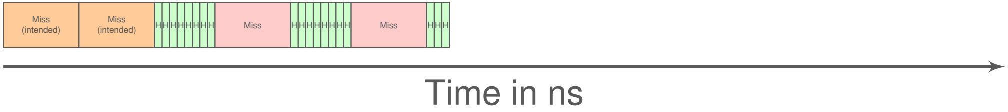


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

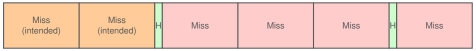


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

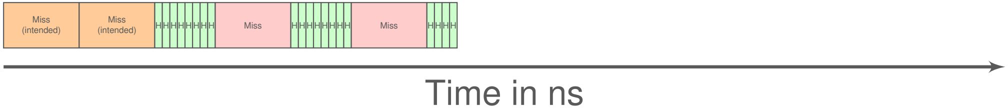


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

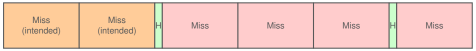


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

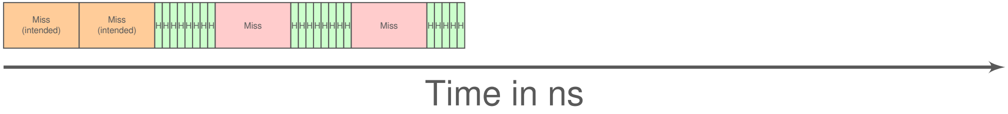


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

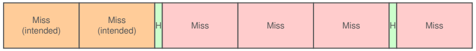


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

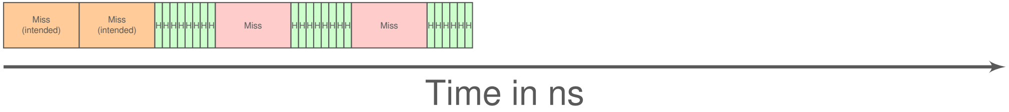


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

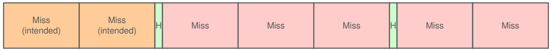


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

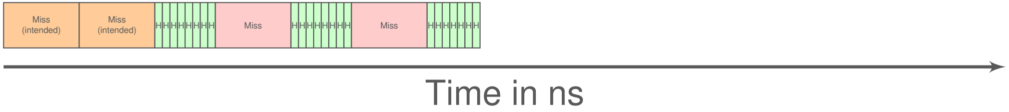


Cache eviction strategies: Illustration

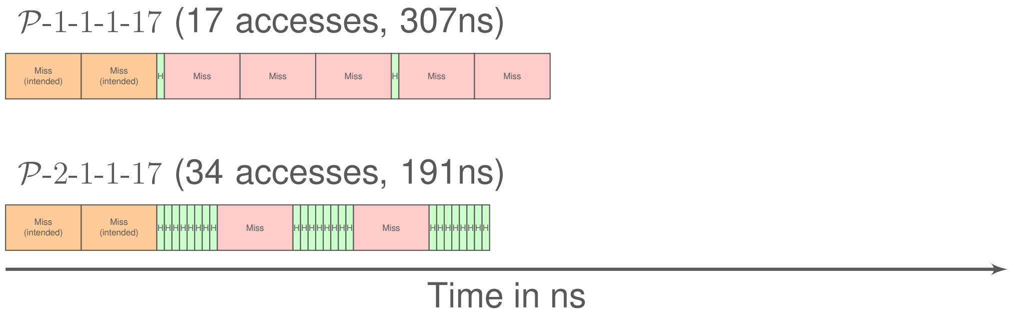
$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)



$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



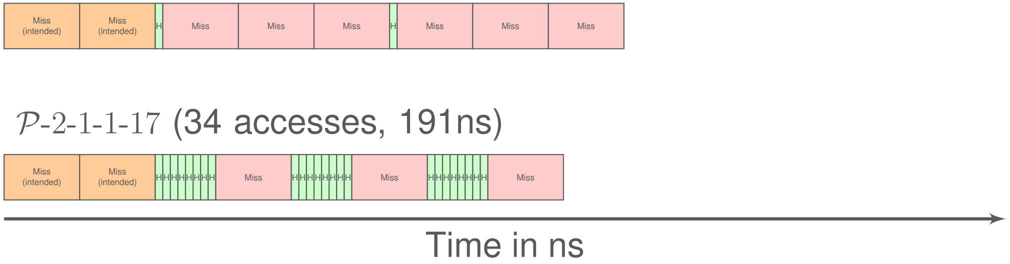
Cache eviction strategies: Illustration



Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

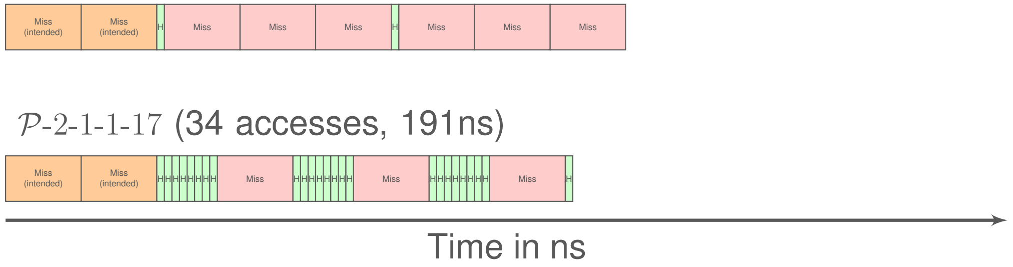
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

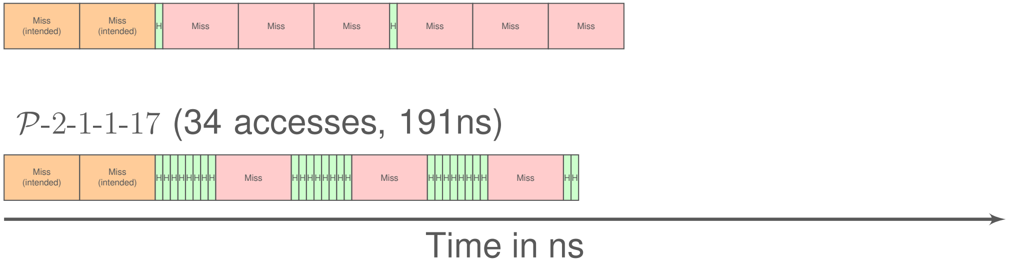
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

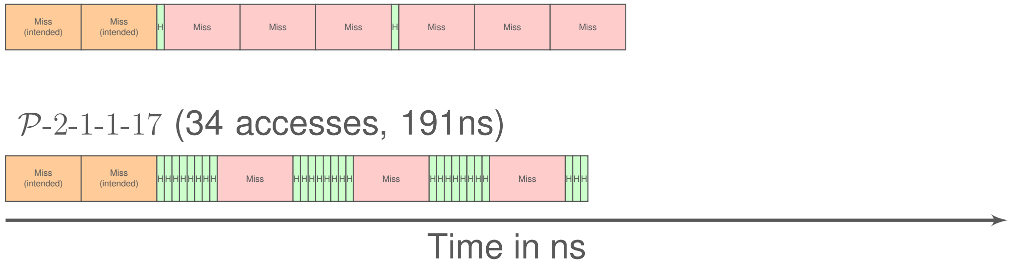
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



Cache eviction strategies: Illustration

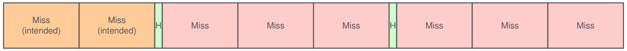
$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

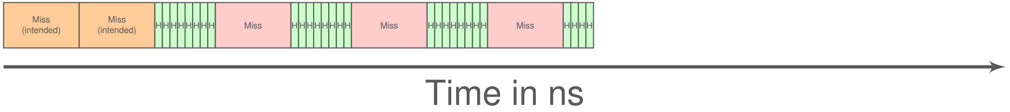


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

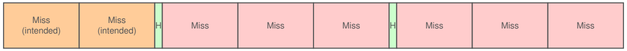


$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

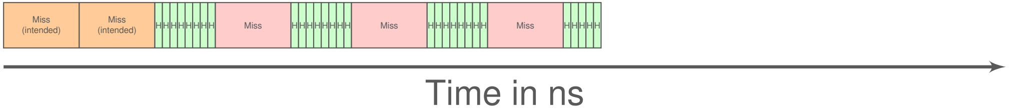


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)



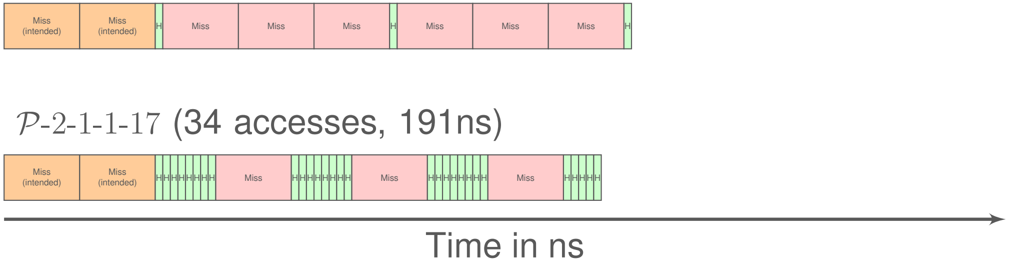
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



Cache eviction strategies: Illustration

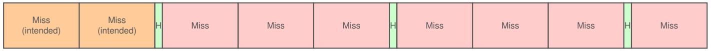
$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

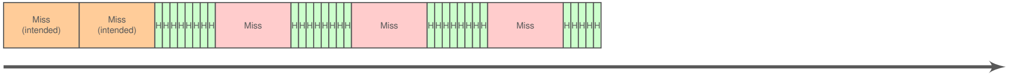


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)



$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

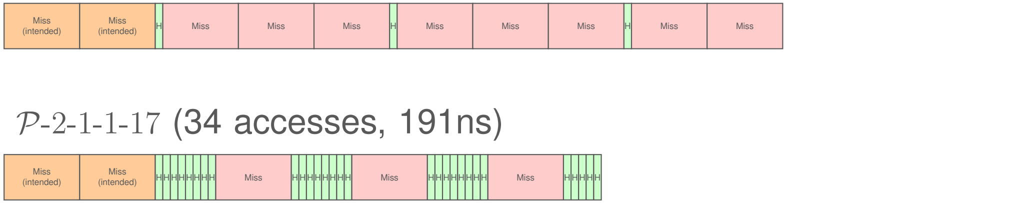


Time in ns

Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

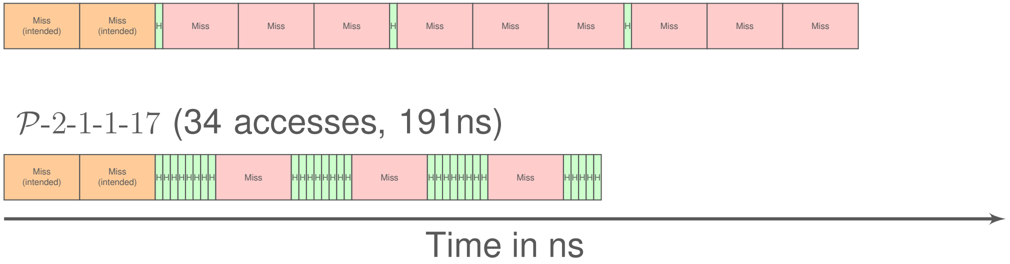


Time in ns

Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

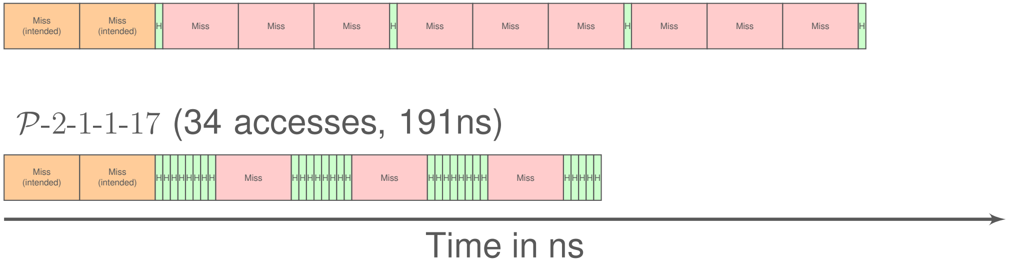
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

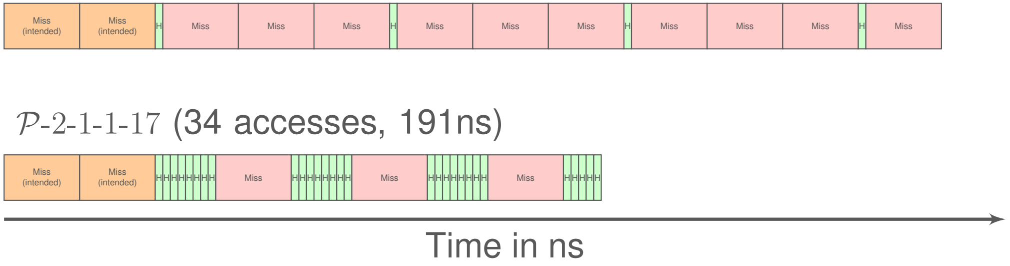
$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



Cache eviction strategies: Illustration

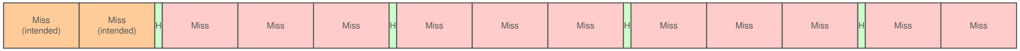
$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)

$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)

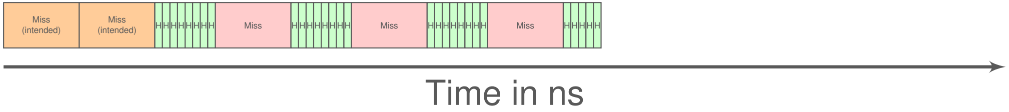


Cache eviction strategies: Illustration

$\mathcal{P}$-1-1-1-17 (17 accesses, 307ns)



$\mathcal{P}$-2-1-1-17 (34 accesses, 191ns)



Evaluation on Haswell

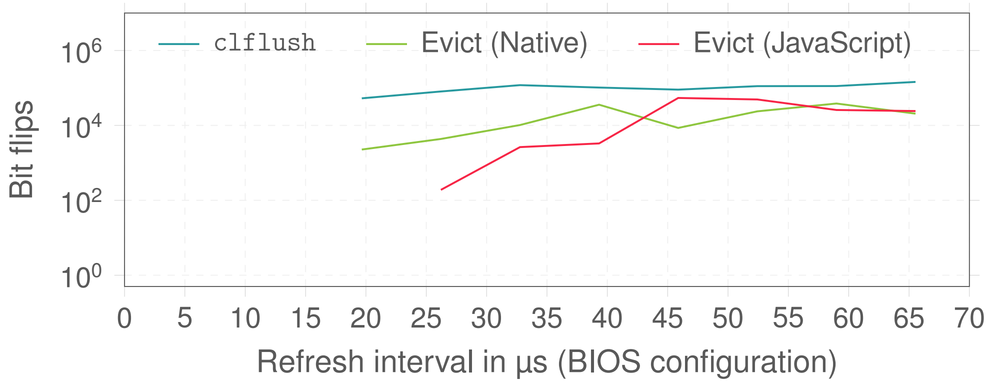


Figure: Number of bit flips within 15 minutes.

Rowhammer.js: Take-Away

- cache eviction fast enough to replace `clflush`
- independent of programming language and available instructions
- first remote fault attack, from a browser

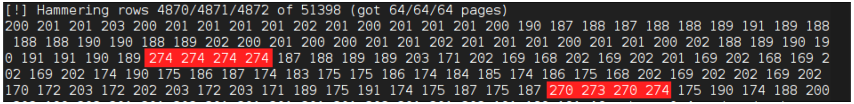
E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. “Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector”. In: S&P'16. 2016.

Rowhammer.js: Take-Away

- cache eviction fast enough to replace `clflush`
- independent of programming language and available instructions
- first remote fault attack, from a browser
- if you think a fault is not exploitable, think again

E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. “Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector”. In: S&P'16. 2016.

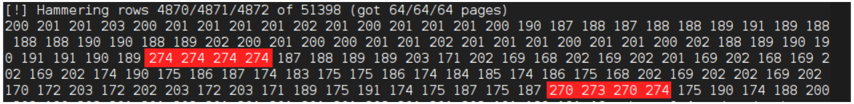
DRAMA: Motivation (1)



```
[!] Hammering rows 4870/4871/4872 of 51398 (got 64/64/64 pages)
200 201 201 203 200 201 201 201 201 202 201 200 201 201 201 200 190 187 188 187 188 188 189 191 189 188
 188 188 190 190 188 189 202 200 201 200 200 201 201 202 201 201 201 200 201 201 200 202 188 189 190 19
0 191 191 190 189 274 274 274 274 187 188 189 189 203 171 202 169 168 202 169 202 201 169 202 168 169 2
02 169 202 174 190 175 186 187 174 183 175 175 186 174 184 185 174 186 175 168 202 169 202 202 169 202
170 172 203 172 202 203 172 203 171 189 175 191 174 175 187 175 187 270 273 270 274 175 190 174 188 200
```

a lot of wasted time

DRAMA: Motivation (1)



```
[!] Hammering rows 4870/4871/4872 of 51398 (got 64/64/64 pages)
200 201 201 203 200 201 201 201 201 202 201 200 201 201 201 200 190 187 188 187 188 188 189 191 189 188
 188 188 190 190 188 189 202 200 201 200 200 201 201 202 201 201 201 200 201 201 200 202 188 189 190 19
0 191 191 190 189 274 274 274 274 187 188 189 189 203 171 202 169 168 202 169 202 201 169 202 168 169 2
02 169 202 174 190 175 186 187 174 183 175 175 186 174 184 185 174 186 175 168 202 169 202 202 169 202
170 172 203 172 202 203 172 203 171 189 175 191 174 175 187 175 187 270 273 270 274 175 190 174 188 200
```

a lot of wasted time

or a side channel?

DRAMA: Motivation (2)

- cache attacks: either not across CPUs, or need shared memory
- limits attacks in restrictive environments

P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: USENIX Security Symposium. 2016.

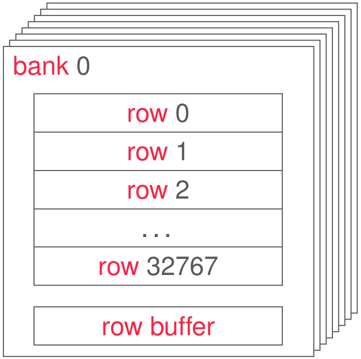
DRAMA: Motivation (2)

- cache attacks: either not across CPUs, or need shared memory
- limits attacks in restrictive environments

$\rightarrow$ exploiting the DRAM, across CPUs and without shared memory

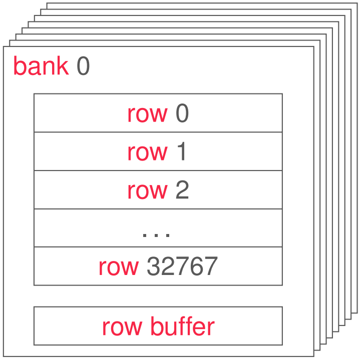
P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: USENIX Security Symposium. 2016.

DRAM organization example



- bits in cells in rows
- access: activate row, copy to row buffer
- row buffer $\rightarrow$ cache!

DRAM organization example



- bits in cells in rows
- access: activate row, copy to row buffer
- row buffer $\rightarrow$ cache!

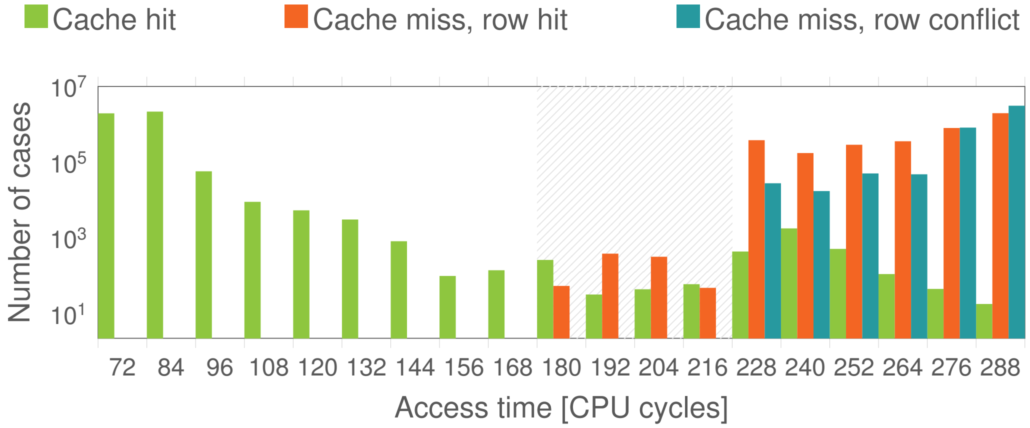
$\rightarrow$ how to exploit these caches?

Row hit and row conflict

When accessing a row i in a bank:

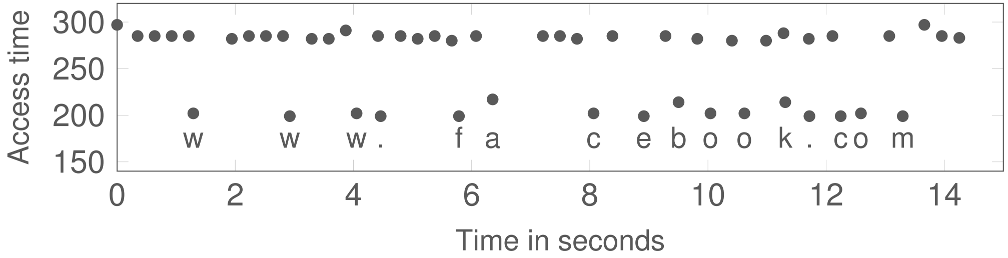
- row hit: row i already opened in row buffer $\rightarrow$ fast
- row conflict: row $j \neq i$ opened in the same bank $\rightarrow$ slow

DRAM timing differences



Example attack

- 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



Take-away

- performance optimizations $\rightarrow$ side channels
- caches $\rightarrow$ leakage
- today's computers are fast because: lots of small optimizations

$\rightarrow$ computers won't stop leaking

Microarchitectural Incontinence

You would leak too if you were so fast!

Daniel Gruss
Graz University of Technology

October 18, 2016 — Hacktivity

References I

[BRBG16] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. “Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector”. In: S&P’16. 2016.

[GBM15] D. Gruss, D. Bidner, and S. Mangard. “Practical Memory Deduplication Attacks in Sandboxed Javascript”. In: ESORICS’15. 2015.

[GIES15] B. Gülmezoğlu, M. S. Inci, T. Eisenbarth, and B. Sunar. “A Faster and More Realistic Flush+Reload Attack on AES”. In: COSADE’15. 2015.

[GMM16] D. Gruss, C. Maurice, and S. Mangard. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: DIMVA’16. 2016.

[GMWM16] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: DIMVA’16. 2016.

[GSM15] D. Gruss, R. Spreitzer, and S. Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: USENIX Security Symposium. 2015.

References II

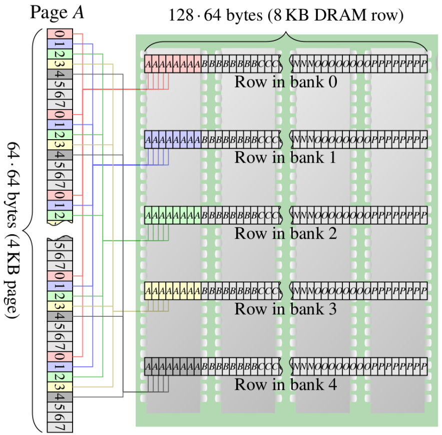
[KPK14]	V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. “ret2dir: Rethinking kernel isolation”. In: USENIX Security Symposium. 2014, pp. 957–972.
[LGSMM16]	M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. “ARMageddon: Last-Level Cache Attacks on Mobile Devices”. In: USENIX Security Symposium. 2016.
[LYGHL15]	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.
[MLSNHF15]	C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. “Reverse Engineering Intel Complex Addressing Using Performance Counters”. In: RAID. 2015.
[OKSK15]	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.
[PGMSM16]	P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: USENIX Security Symposium. 2016.

References III

[Sea15] M. Seaborn. How physical addresses map to rows and banks in DRAM. http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html. Retrieved on July 20, 2015. 2015.

[YF14] Y. Yarom and K. Falkner. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: USENIX Security Symposium. 2014.

Granularity of the attacks



- 8 out of 64 regions ($= 512\,B$) map to the same bank.
- each row is divided among 16 different pages ($A - P$)
- occupying 1 page B to P enough to spy on the eight 64-byte regions of page A in the same bank

$\rightarrow$ granularity: $512\,B$ = 2 cache lines