Rowhammer Attacks:
A Walkthrough Guide

Daniel Gruss & Clémentine Maurice, Graz University of Technology

May 4, 2017 — RuhrSec 2017

Who are we

- **Daniel Gruss**
- PhD student @ Graz University Of Technology
- 🐦 @lavados
- ✉ daniel.gruss@iaik.tugraz.at

Who are we

- **Clémentine Maurice**
- PhD in computer science, Postdoc @ Graz University Of Technology
- 🐦 @BloodyTangerine
- ✉ clementine.maurice@iaik.tugraz.at

Goals of this talk

- you get a comprehensive overview of Rowhammer attacks
- you can run the tools on your machine
- you understand what's happening and why

→ nothing here is black magic!

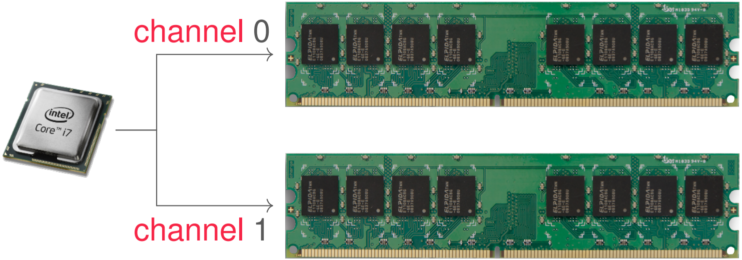
Outline

- Background
- How to flip bits?
- How to exploit them?
- How to mitigate them?
- Conclusion

1. Background

DRAM organization

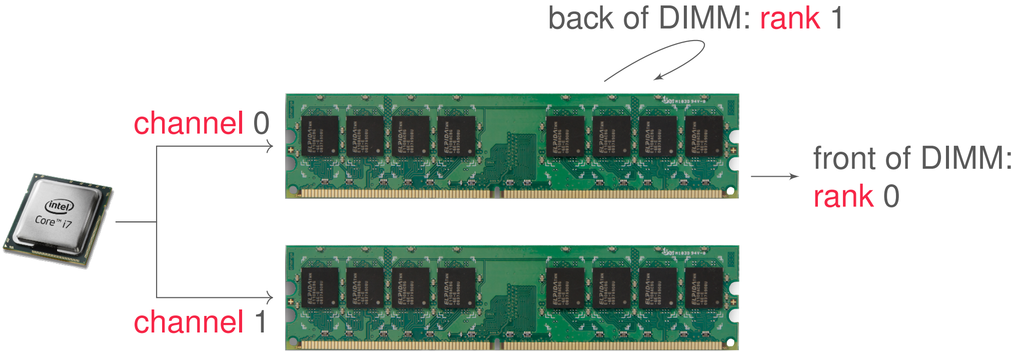
DRAM organization



channel 0

channel 1

DRAM organization



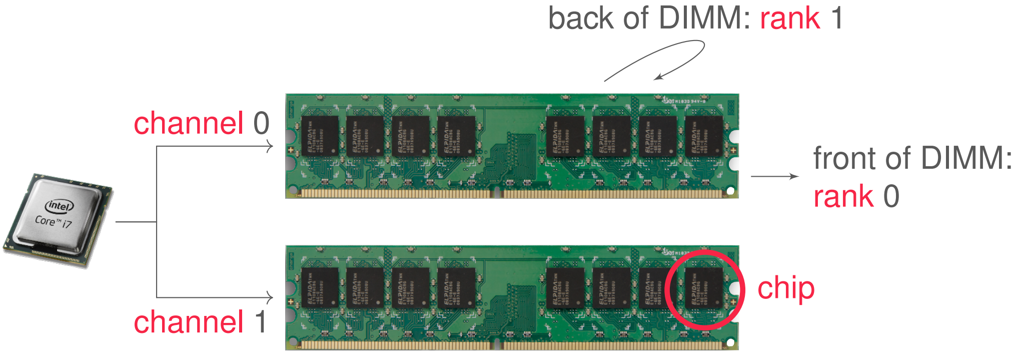
back of DIMM: rank 1

channel 0

front of DIMM: rank 0

channel 1

DRAM organization



back of DIMM: rank 1

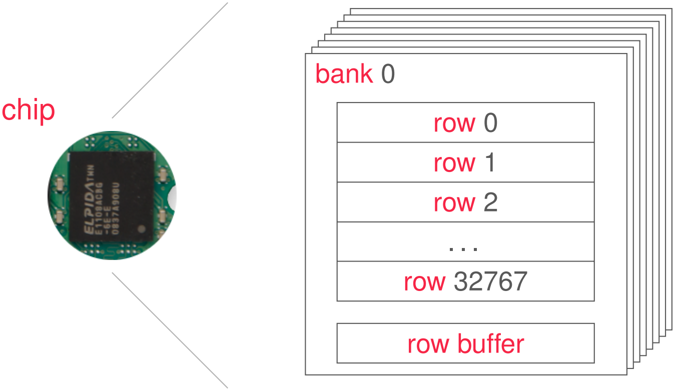
channel 0

front of DIMM: rank 0

chip

channel 1

DRAM organization



chip

bank 0

row 0

row 1

row 2

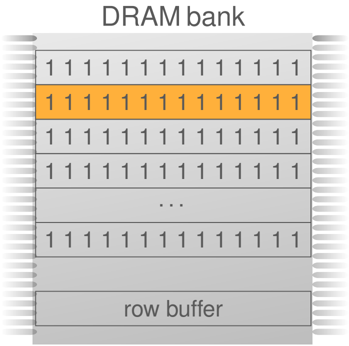
. . .

row 32767

row buffer

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

How reading from DRAM works



DRAM bank

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

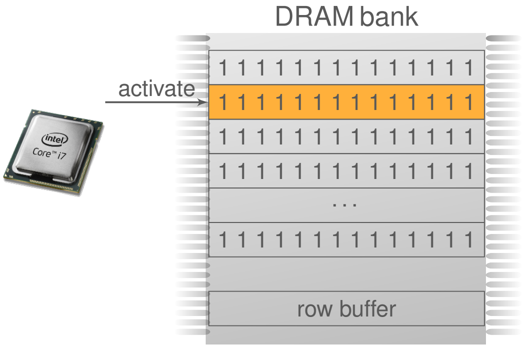
. . .

1 1 1 1 1 1 1 1 1 1 1 1 1 1

row buffer

CPU wants to access row 1

How reading from DRAM works



DRAM bank

activate

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

. . .

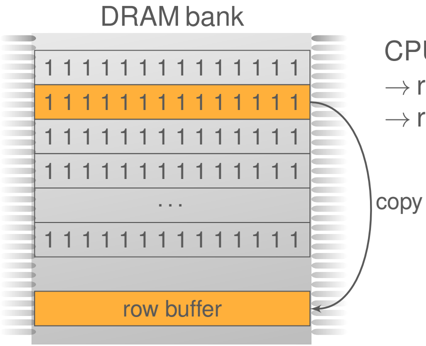
1 1 1 1 1 1 1 1 1 1 1 1 1 1

row buffer

CPU wants to access row 1

→ row 1 activated

How reading from DRAM works



DRAM bank

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

. . .

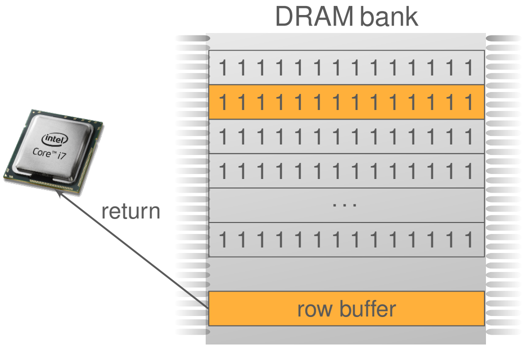
1 1 1 1 1 1 1 1 1 1 1 1 1 1

row buffer

copy

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

How reading from DRAM works



DRAM bank

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

. . .

1 1 1 1 1 1 1 1 1 1 1 1 1 1

row buffer

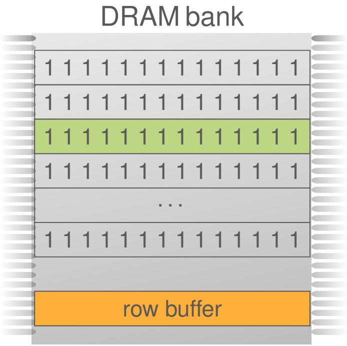
return

CPU wants to access row 1

→ row 1 activated

→ row 1 copied to row buffer

How reading from DRAM works


DRAM bank

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

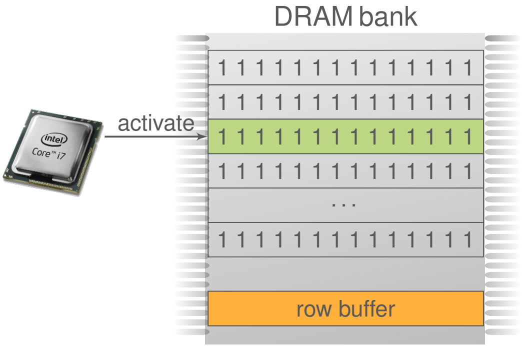
. . .

1 1 1 1 1 1 1 1 1 1 1 1 1 1

row buffer

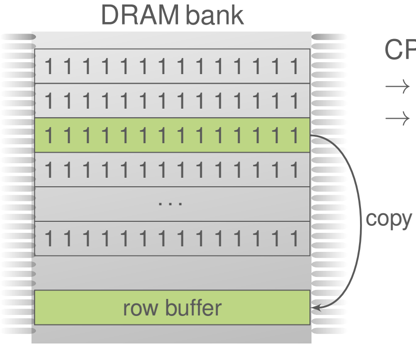
CPU wants to access row 2

How reading from DRAM works



CPU wants to access row 2
→ row 2 activated

How reading from DRAM works

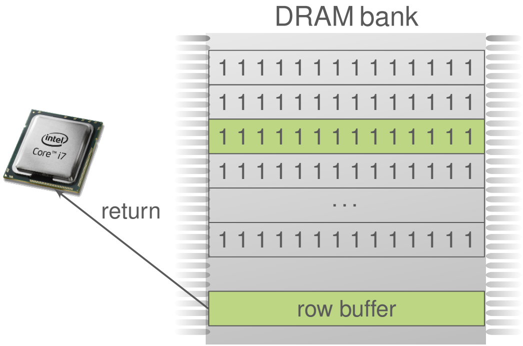


CPU wants to access row 2

→ row 2 activated

→ row 2 copied to row buffer

How reading from DRAM works



DRAM bank

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

. . .

1 1 1 1 1 1 1 1 1 1 1 1 1 1

row buffer

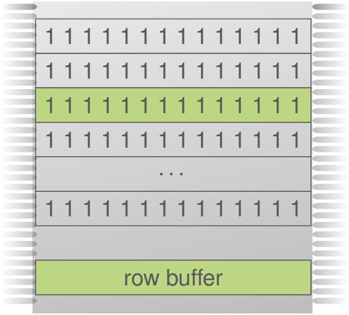
CPU wants to access row 2

→ row 2 activated

→ row 2 copied to row buffer

How reading from DRAM works

DRAM bank



1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

. . .

1 1 1 1 1 1 1 1 1 1 1 1 1 1

row buffer

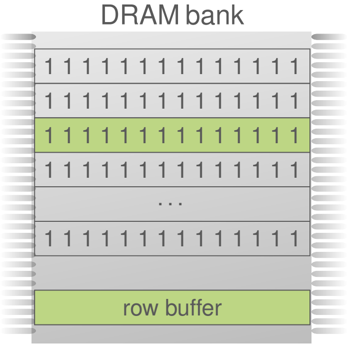
CPU wants to access row 2

→ row 2 activated

→ row 2 copied to row buffer

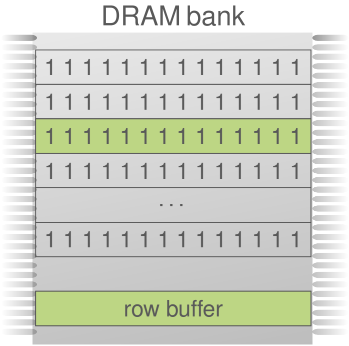
→ slow (row conflict)

How reading from DRAM works



CPU wants to access row 2—again

How reading from DRAM works



DRAM bank

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

. . .

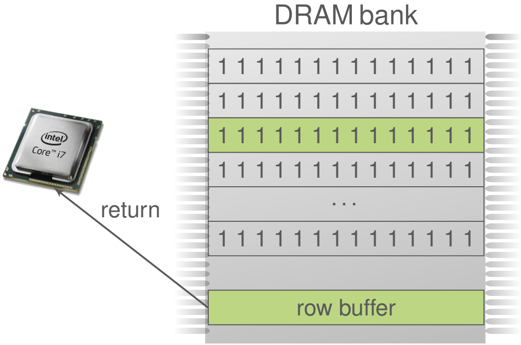
1 1 1 1 1 1 1 1 1 1 1 1 1 1

row buffer

CPU wants to access row 2—again

→ row 2 already in row buffer

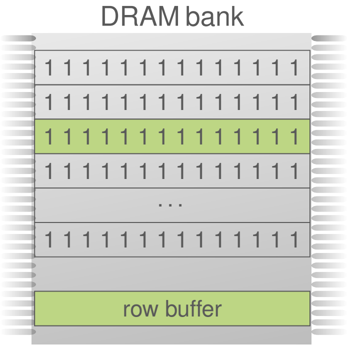
How reading from DRAM works



CPU wants to access row 2—again

→ row 2 already in row buffer

How reading from DRAM works

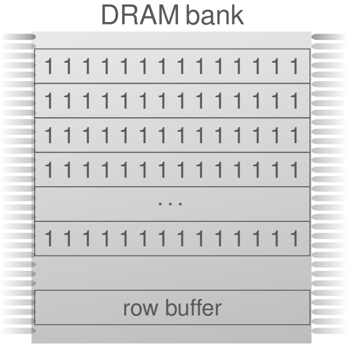


CPU wants to access row 2—again

→ row 2 already in row buffer

→ fast (row hit)

How reading from DRAM works



DRAM bank

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

. . .

1 1 1 1 1 1 1 1 1 1 1 1 1 1

row buffer

row buffer = cache

DRAM refresh

- cells leak → repetitive refresh necessary
- refresh ≈ reading (destructive) + writing same data again
- maximum interval between refreshes to guarantee data integrity

DRAM refresh

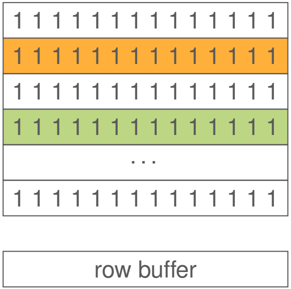
- cells leak → repetitive refresh necessary
- refresh ≈ reading (destructive) + writing same data again
- maximum interval between refreshes to guarantee data integrity

- cells leak faster upon proximate accesses → Rowhammer

Rowhammer

"It's like breaking into an apartment by repeatedly slamming a neighbor's door until the vibrations open the door you were after" – Motherboard Vice

DRAM bank



1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

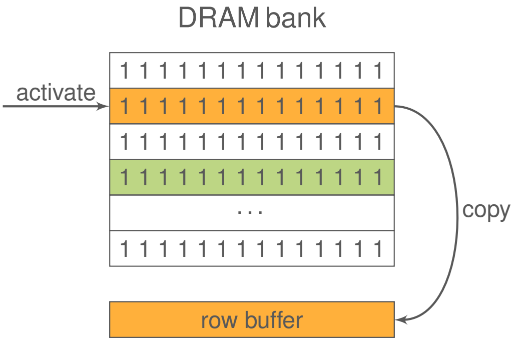
. . .

1 1 1 1 1 1 1 1 1 1 1 1 1 1

row buffer

Rowhammer

"It's like breaking into an apartment by repeatedly slamming a neighbor's door until the vibrations open the door you were after" – Motherboard Vice



DRAM bank

activate

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

. . .

1 1 1 1 1 1 1 1 1 1 1 1 1 1

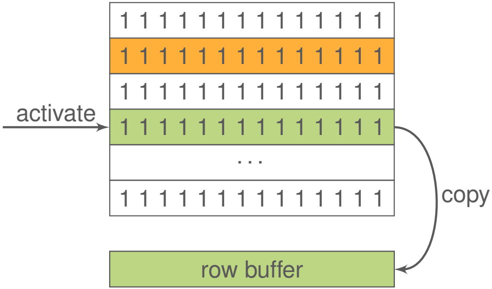
copy

row buffer

Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice

DRAM bank



1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

activate

1 1 1 1 1 1 1 1 1 1 1 1 1 1

. . .

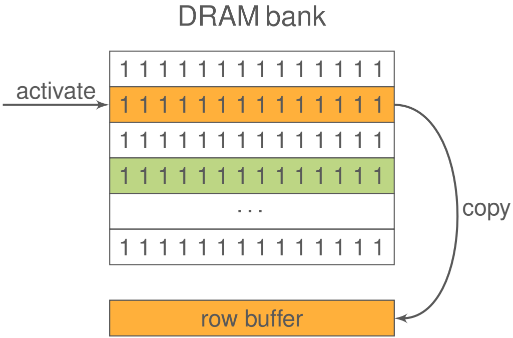
1 1 1 1 1 1 1 1 1 1 1 1 1 1

copy

row buffer

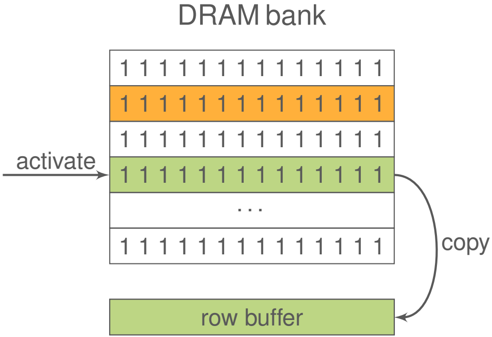
Rowhammer

"It's like breaking into an apartment by repeatedly slamming a neighbor's door until the vibrations open the door you were after" – Motherboard Vice



Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice



DRAM bank

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

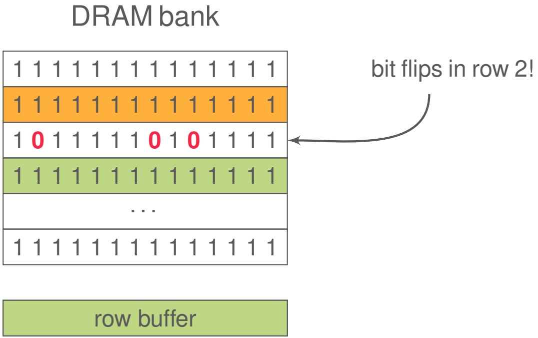
1 1 1 1 1 1 1 1 1 1 1 1 1 1

. . .

1 1 1 1 1 1 1 1 1 1 1 1 1 1

Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice



DRAM bank

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 **0** 1 1 1 1 1 **0** 1 **0** 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1

. . .

1 1 1 1 1 1 1 1 1 1 1 1 1 1

row buffer

bit flips in row 2!

2. How to flip bits?

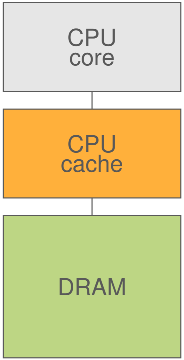
Requirements

Memory accesses must be

- uncached: reach DRAM
- fast: race against the next row refresh
- targeted: reach specific row

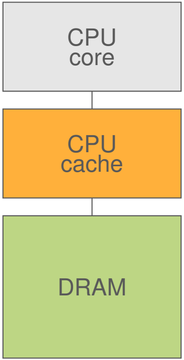
How do we get enough uncached accesses?

Impact of the CPU cache



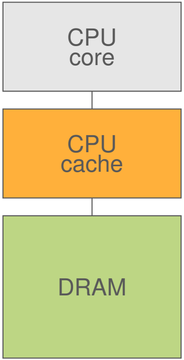
- only non-cached accesses reach DRAM

Impact of the CPU cache



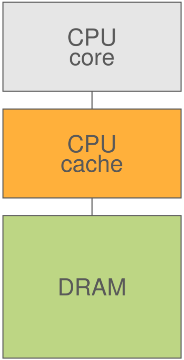
- only non-cached accesses reach DRAM
- either remove data from cache

Impact of the CPU cache



- only non-cached accesses reach DRAM
- either remove data from cache
- or don’t put it there in the first place

Impact of the CPU cache



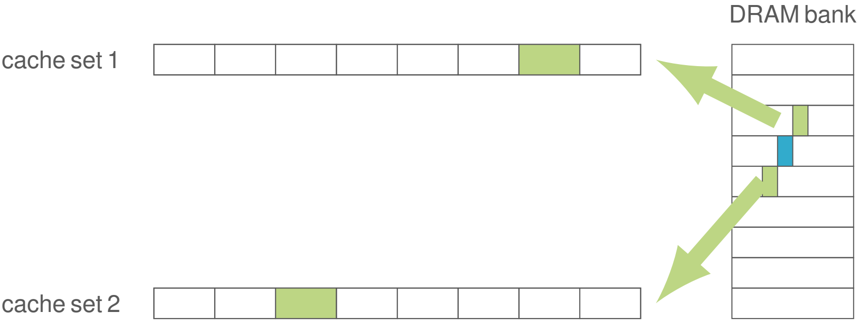
- only non-cached accesses reach DRAM
- either remove data from cache
- or don’t put it there in the first place

→ next access will be served from DRAM

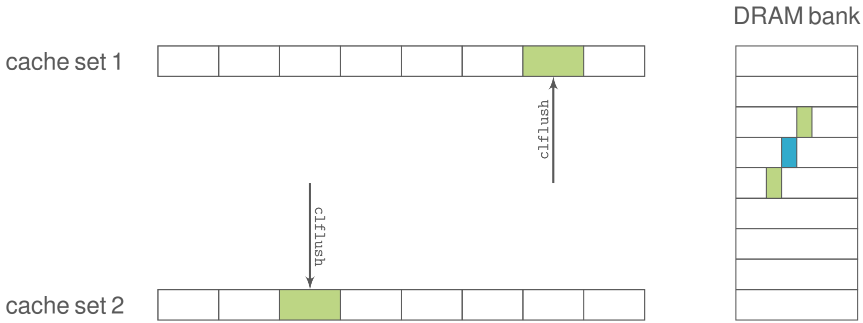
Access techniques

1. `clflush` instruction → original paper (Kim et al. 2014)
2. cache eviction (Gruss, Maurice, and Mangard 2016; Aweke et al. 2016)
3. non-temporal accesses (Qiao et al. 2016)
4. uncached memory (Veen et al. 2016)

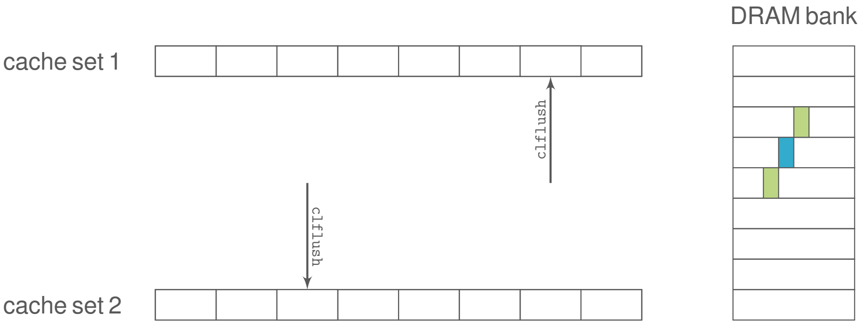
#1 Hammering with `clflush`



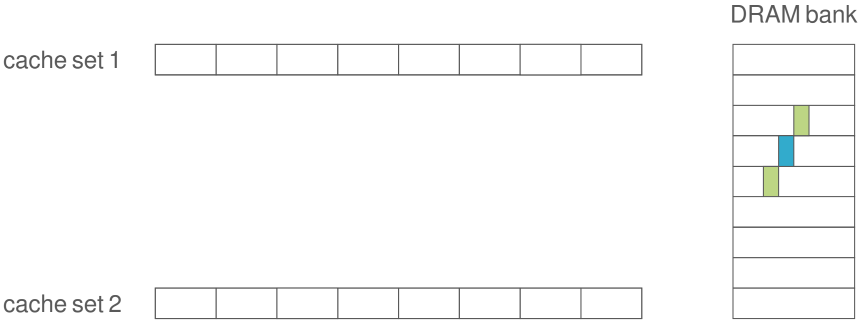
#1 Hammering with clflush



#1 Hammering with `clflush`



#1 Hammering with `clflush`

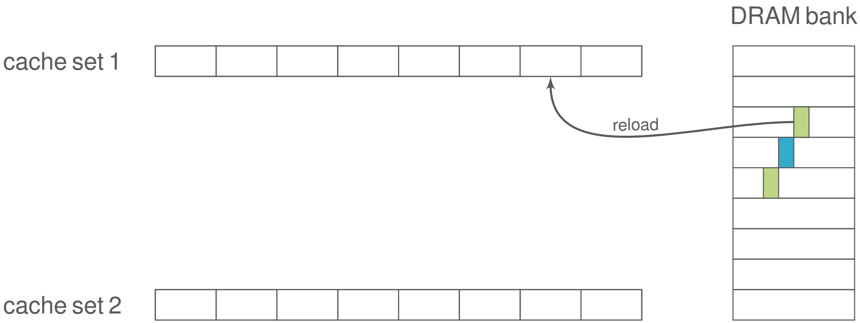


DRAM bank

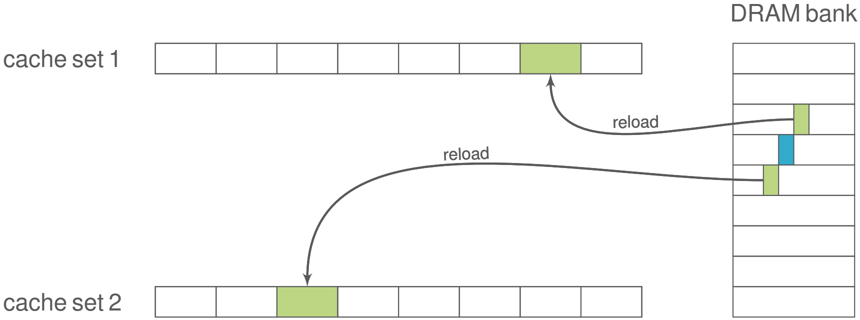
cache set 1

cache set 2

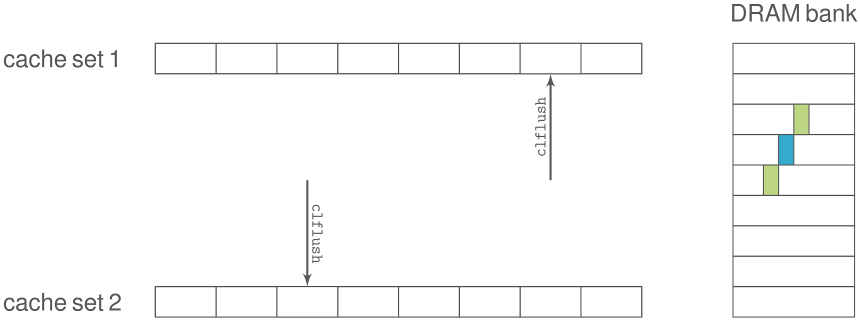
#1 Hammering with `clflush`



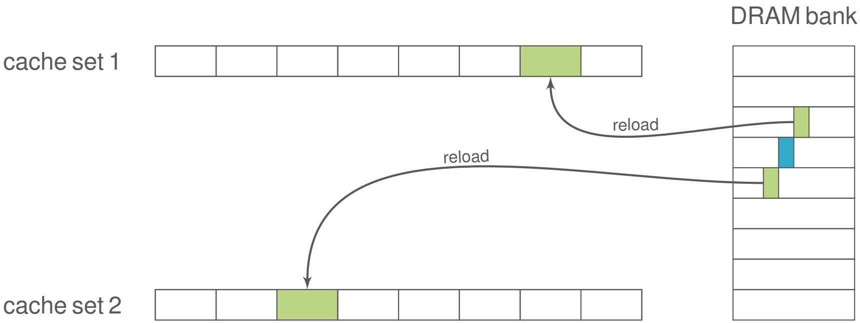
#1 Hammering with `clflush`



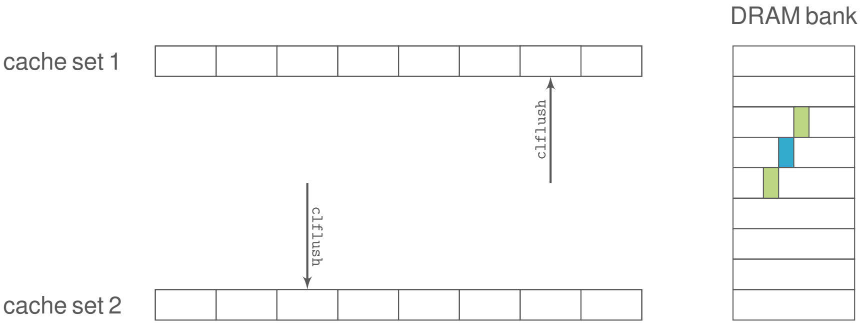
#1 Hammering with `clflush`



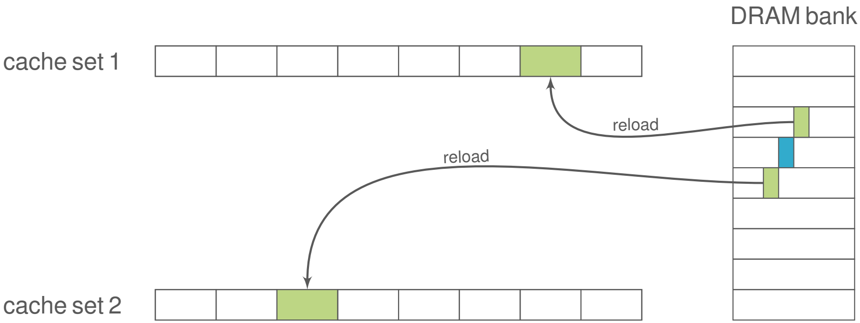
#1 Hammering with `clflush`



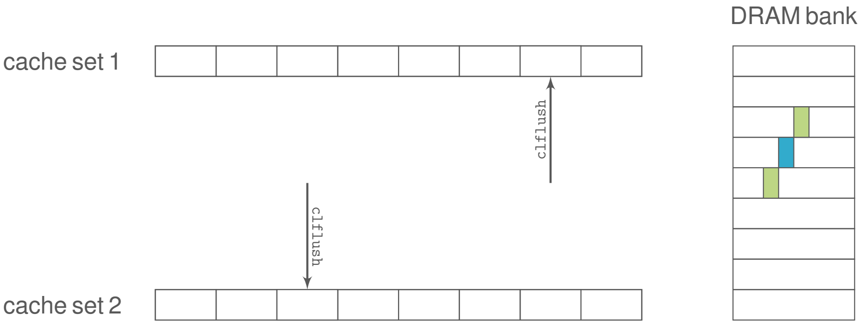
#1 Hammering with `clflush`



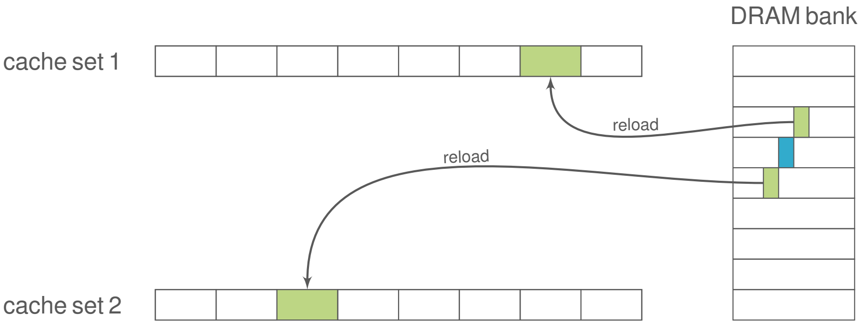
#1 Hammering with `clflush`



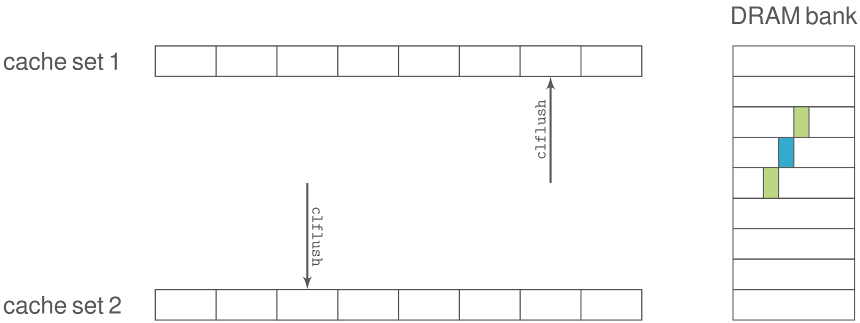
#1 Hammering with `clflush`



#1 Hammering with `clflush`

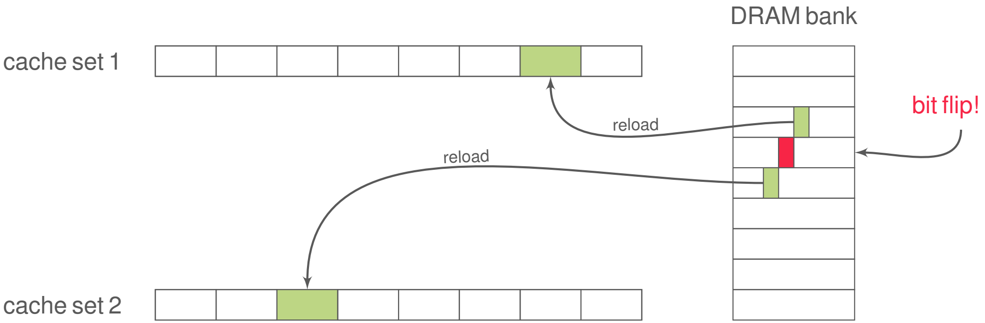


#1 Hammering with `clflush`



wait for it. . .

#1 Hammering with `clflush`



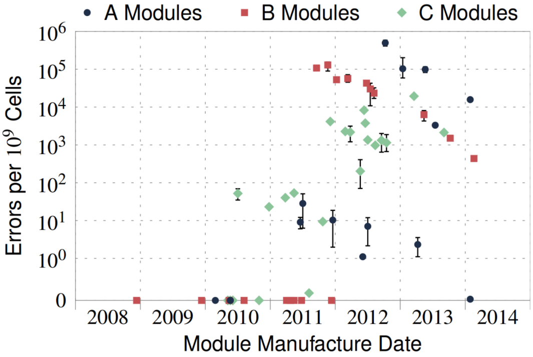
How widespread is the issue?

DDR3:

- Kim et al.: 110/129 modules from 3 vendors, all but 3 since mid-2011
- Seaborn and Dullien: 15/29 laptops

DDR4 believed to be safe:

- we showed bit flips (Pessl et al. 2016)



Prevalence, by Kim et al. 2014

Flush, reload, flush, reload. . .

- the core of Rowhammer is essentially a Flush+Reload loop
- as much an attack on DRAM as on cache

#2 Hammering with cache eviction

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

  → no `clflush` in JavaScript

#2 Hammering with cache eviction

- 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!

#2 Hammering with cache eviction

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

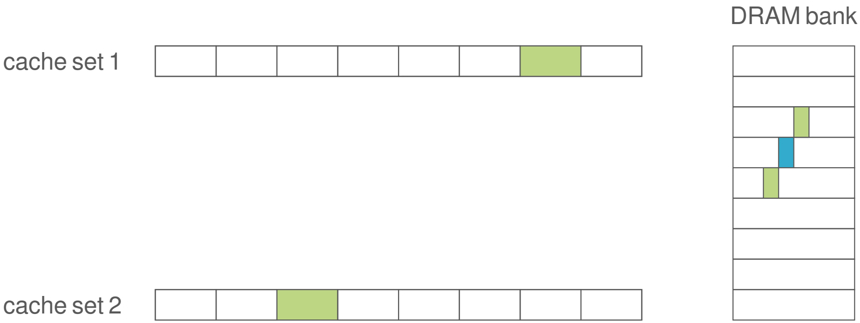
  → no `clflush` in JavaScript

- our approach: use regular memory accesses for eviction

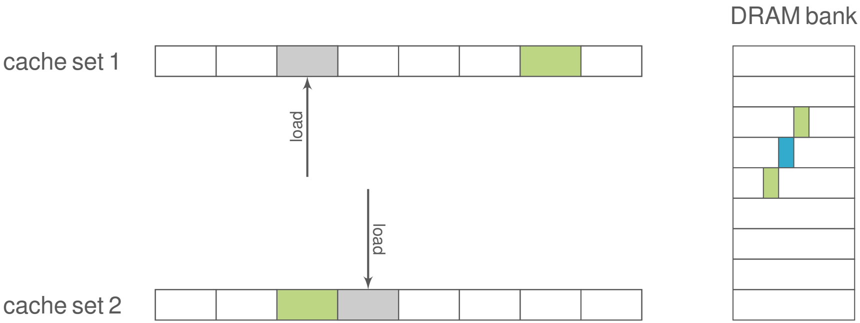
  → techniques from cache attacks!

  → Rowhammer, Prime+Probe style!

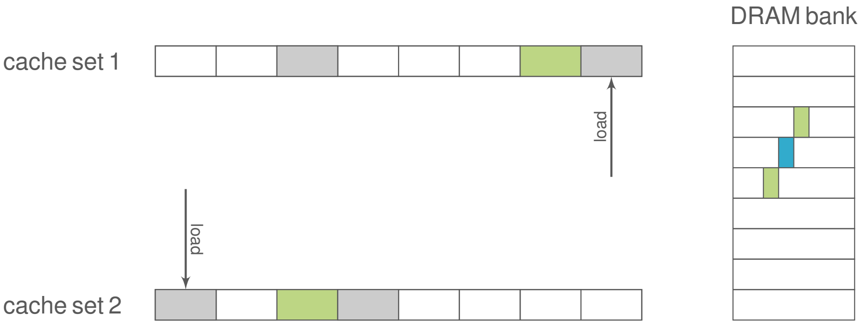
#2 Hammering with cache eviction



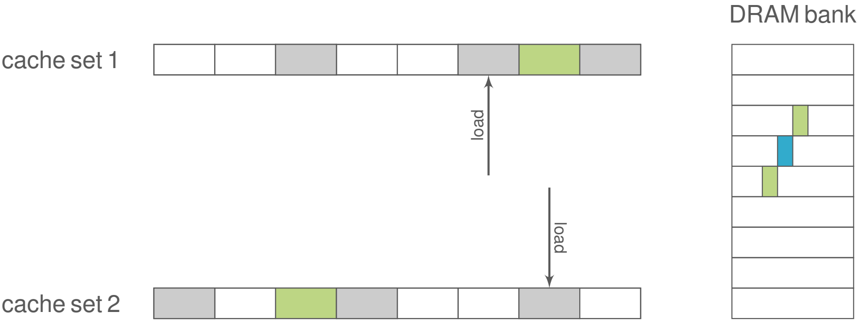
#2 Hammering with cache eviction



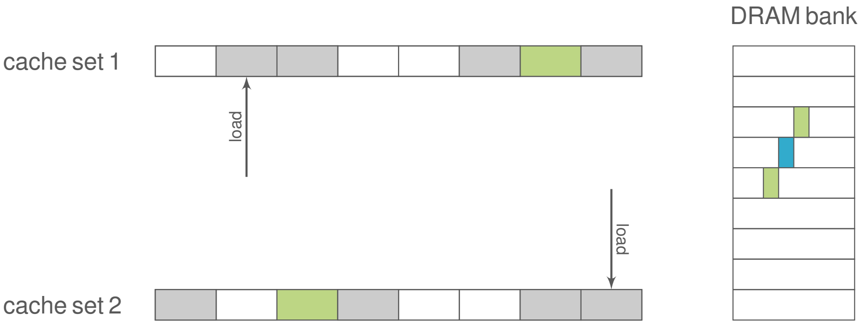
#2 Hammering with cache eviction



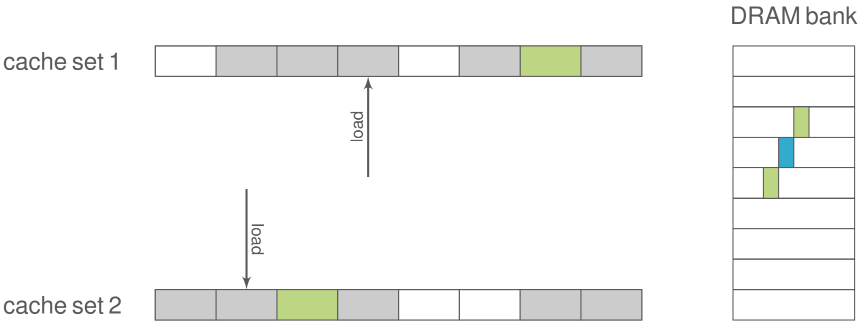
#2 Hammering with cache eviction



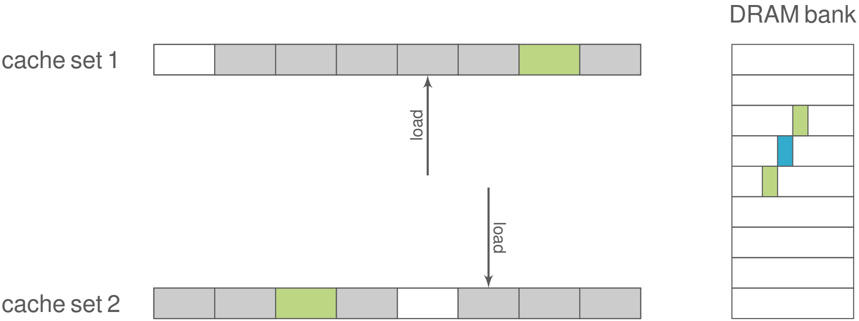
#2 Hammering with cache eviction



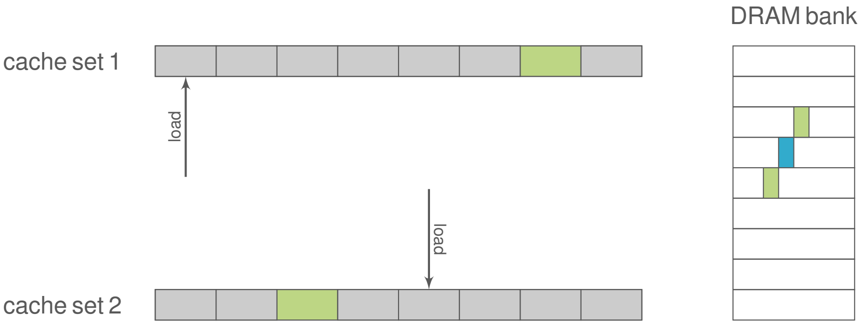
#2 Hammering with cache eviction



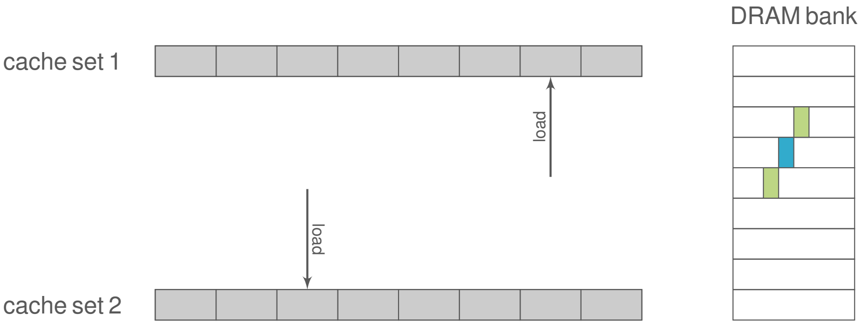
#2 Hammering with cache eviction



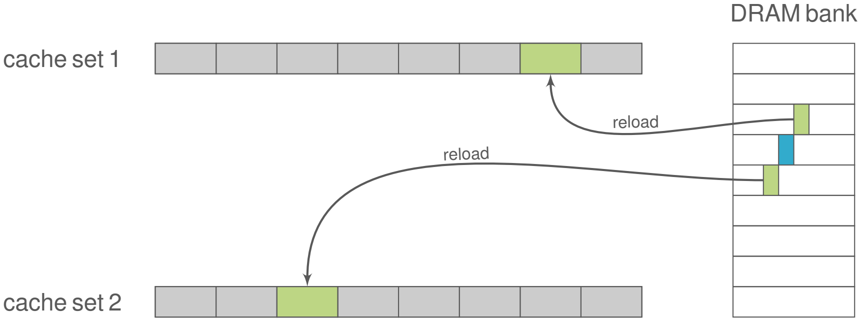
#2 Hammering with cache eviction



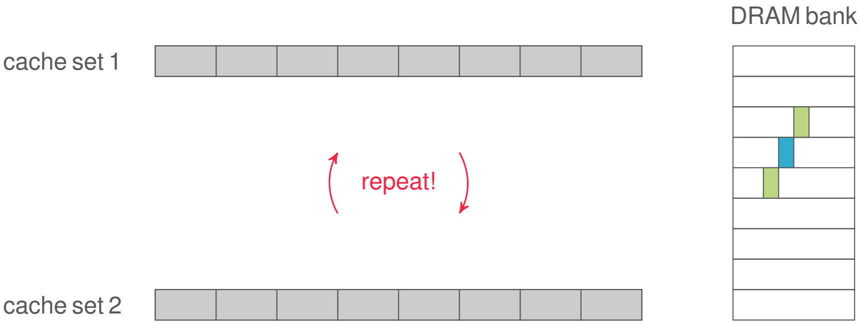
#2 Hammering with cache eviction



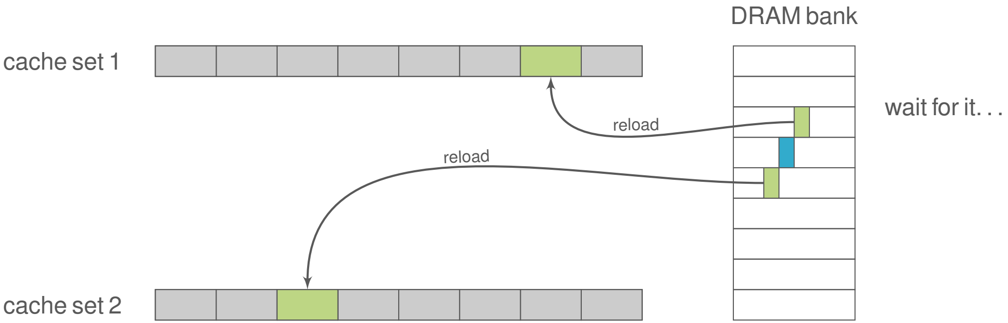
#2 Hammering with cache eviction



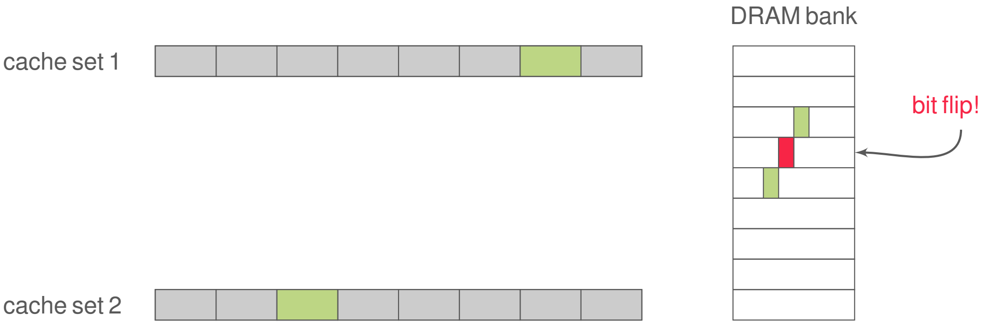
#2 Hammering with cache eviction



#2 Hammering with cache eviction



#2 Hammering with cache eviction

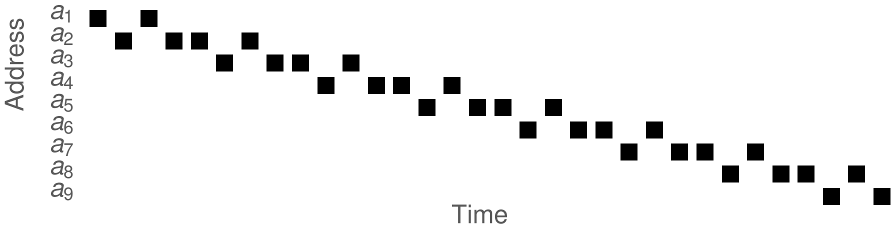


Cache eviction strategies

Not as simple as that → replacement policy is not LRU

Cache eviction strategies

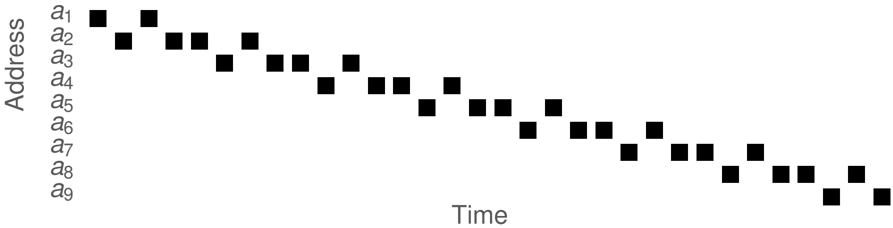
Not as simple as that → replacement policy is not LRU



→ fast and effective on Haswell: eviction rate >99.97%

Cache eviction strategies

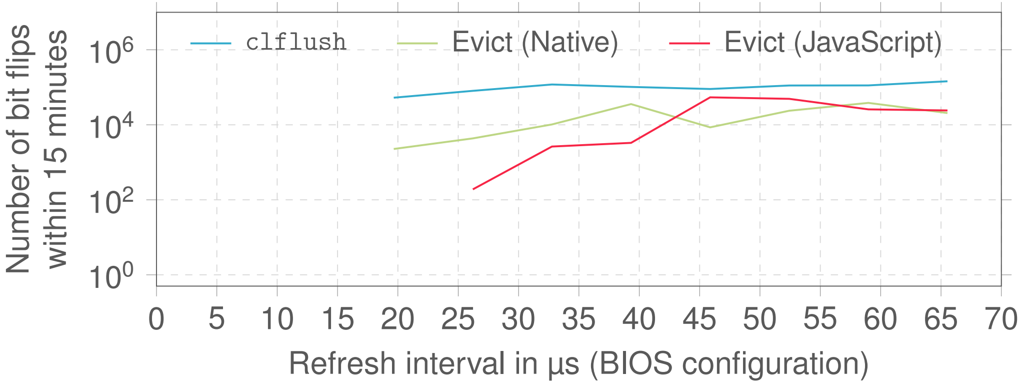
Not as simple as that → replacement policy is not LRU



→ fast and effective on Haswell: eviction rate >99.97%

→ we evaluated 10 000+ strategies to find the best one

Hammering with cache eviction on Haswell



#3 Hammering with non-temporal accesses

- non-temporal accesses: data accessed just once, not in the future
- NTA instructions → bypass cache to minimize cache pollution

#3 Hammering with non-temporal accesses

- non-temporal accesses: data accessed just once, not in the future
- NTA instructions → bypass cache to minimize cache pollution
- NT stores to 1 address are combined at WC buffer
- only last write goes to DRAM → rate not sufficient

#3 Hammering with non-temporal accesses

- non-temporal accesses: data accessed just once, not in the future
- NTA instructions → bypass cache to minimize cache pollution
- NT stores to 1 address are combined at WC buffer
- only last write goes to DRAM → rate not sufficient
- following cached access to same address (Qiao et al. 2016)

#3 Hammering with non-temporal accesses

```
begin:
  movnti %eax, (X)
  movnti %eax, (Y)
  mov %eax, (X)
  mov %eax, (Y)
  jmp begin
```

#4 Hammering with uncached memory

Sometimes, everything fails,

#4 Hammering with uncached memory

Sometimes, everything fails, e.g., on mobile devices

#4 Hammering with uncached memory

Sometimes, everything fails, e.g., on mobile devices

- ARMv7 flush instruction is privileged

#4 Hammering with uncached memory

Sometimes, everything fails, e.g., on mobile devices

- ARMv7 flush instruction is privileged
- cache eviction seems to be too slow

#4 Hammering with uncached memory

Sometimes, everything fails, e.g., on mobile devices

- ARMv7 flush instruction is privileged
- cache eviction seems to be too slow
- ARMv8 non-temporal stores are still cached in practice

#4 Hammering with uncached memory

- ION: memory management since Android 4.0
- apps can use `/dev/ion` for uncached, physically contiguous memory
- no privilege and no permission needed (Veen et al. 2016)

How do we target accesses?

Physical addresses and DRAM

- fixed map: physical addresses → DRAM cells
- undocumented for Intel
- reverse-engineering for Sandy Bridge (Seaborn 2015)
- and by us for Sandy, Ivy, Haswell, Skylake, . . . (Pessl et al. 2016)
- using the timing difference between row hits and row conflicts

How do I reverse my own DRAM?

https://github.com/IAIK/DRAMA

```
taskset 0x4 sudo ./measure -p 0.5 -s 16
# taskset core for stability
# sudo for pagemap access
# -p 0.5 allocate 50% of memory, the more the better
# -s I expect at least 16 sets (I have 32)
```

How do I flip bits?

https://github.com/IAIK/rowhammerjs

Copy functions from `measure` result

```
make ivy # or your microarchitecture
sudo ./rowhammer-ivy -d 2
# sudo for pagemap
# -d 2, for 2 DIMMs
sudo ./rowhammer-ivy -d 2 -f 0
# -f 0, only test offset 0 of every row
```

Demo

Demo!

3. How to exploit bit flips?

How to exploit random bit flips?

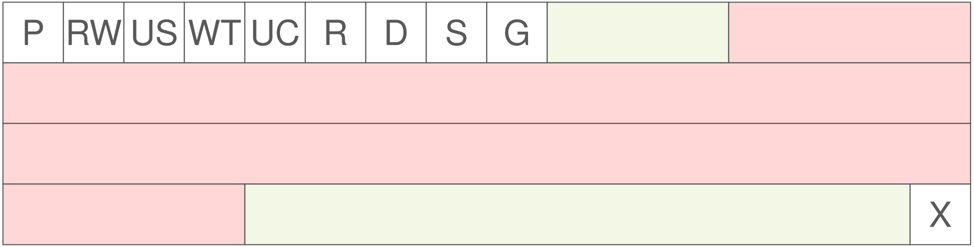
- They are not random → highly reproducible flip pattern!

  1. choose a data structure that you can place at arbitrary memory locations
  2. scan for “good” flips
  3. place data structure there
  4. trigger bit flip again

Strategy: Modify instructions

- idea from Seaborn and Dullien 2015
- x86 op codes are variable length
  - unsafe op codes (syscall) $\in$ safe but long multi-byte op codes
  - only a problem with jumps to arbitrary addresses
- flip a bit in a validated NaCl instruction sequence
  - safe + validated jump $\rightarrow$ arbitrary jump

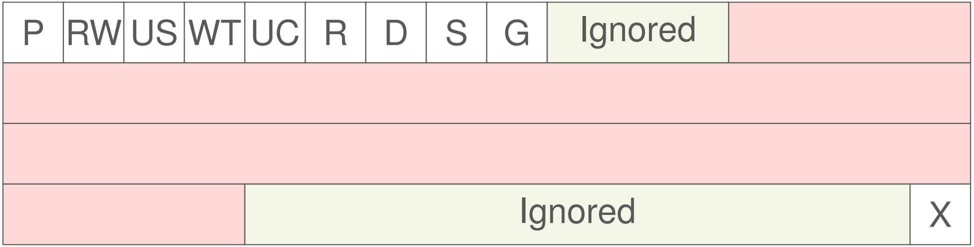
Page Table Entries



P RW US WT UC R D S G

X

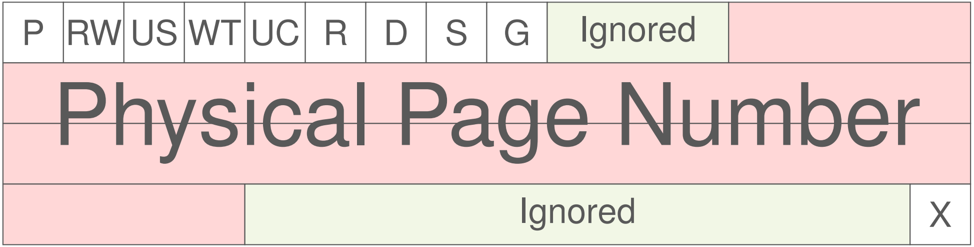
Page Table Entries



P RW US WT UC R D S G Ignored

Ignored X

Page Table Entries

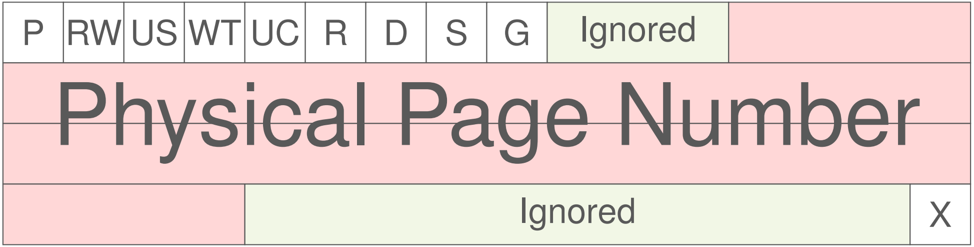


P RW US WT UC R D S G Ignored

Physical Page Number

Ignored X

Page Table Entries



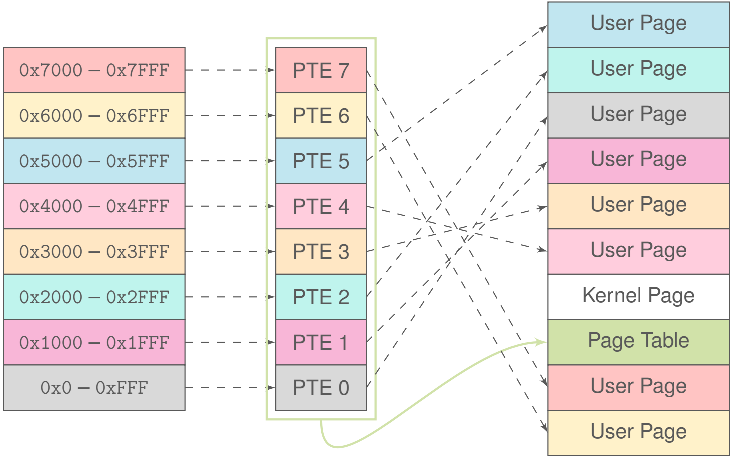
P RW US WT UC R D S G Ignored

Physical Page Number

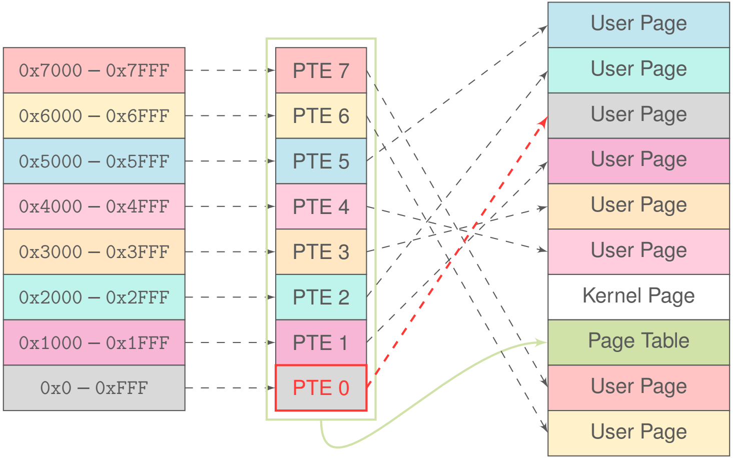
Ignored X

Each 4 KB page table consists of 512 such entries

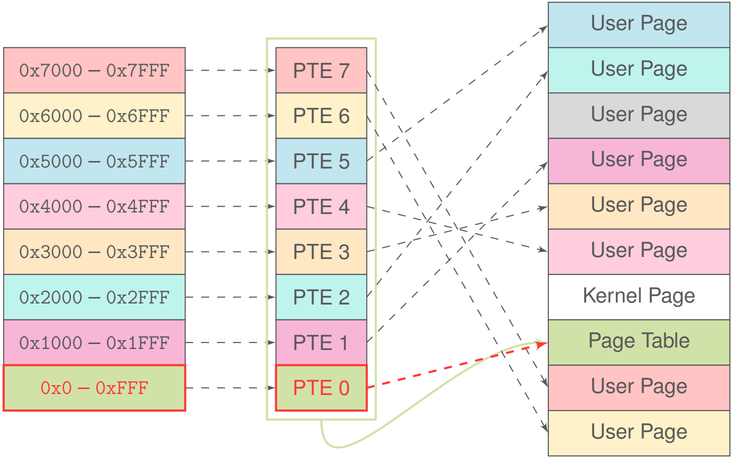
Page Table Manipulation



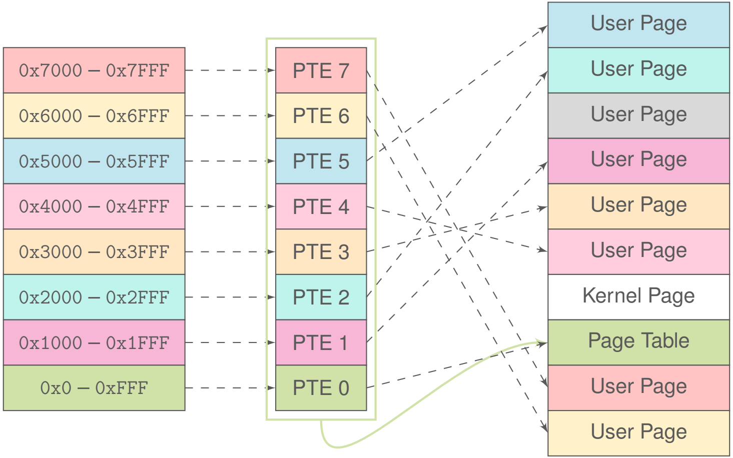
Page Table Manipulation



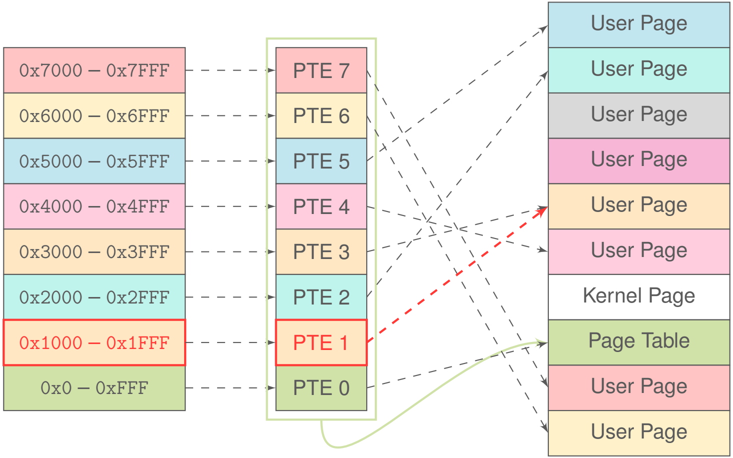
Page Table Manipulation



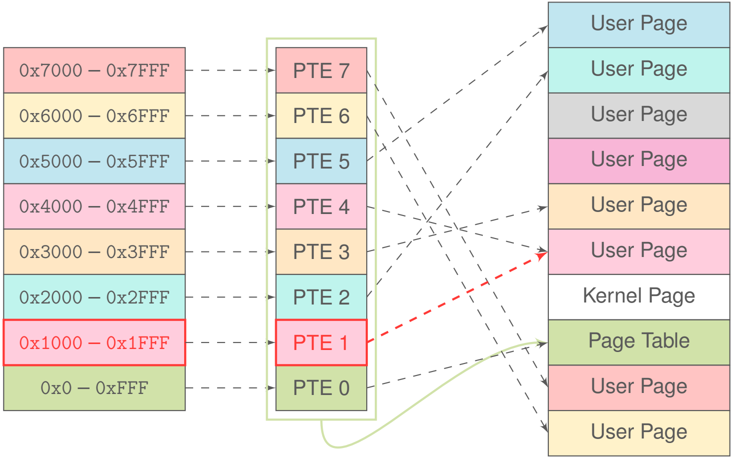
Page Table Manipulation



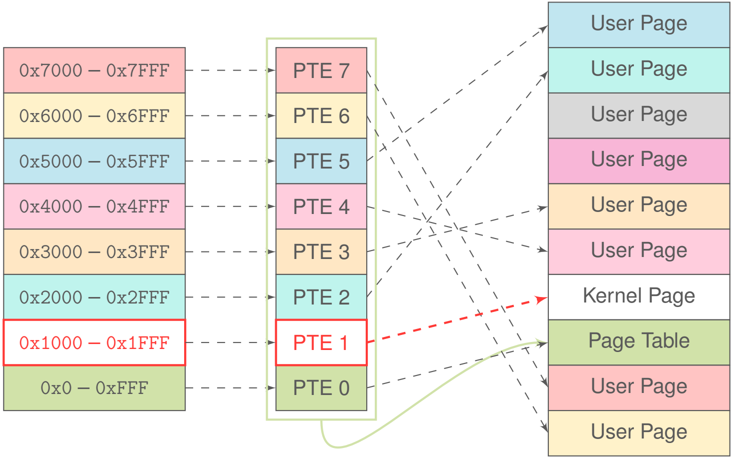
Page Table Manipulation



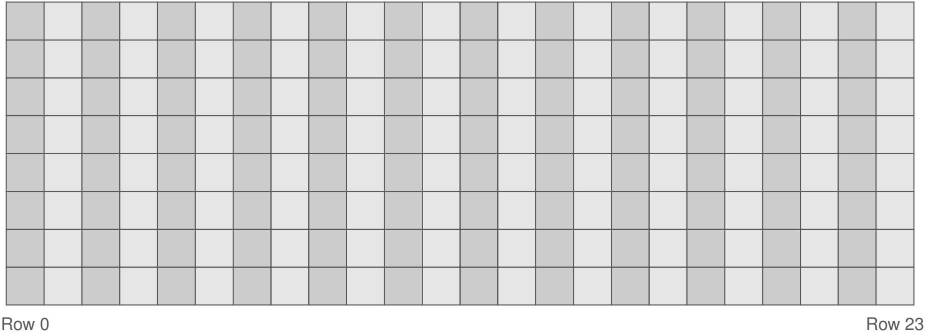
Page Table Manipulation



Page Table Manipulation

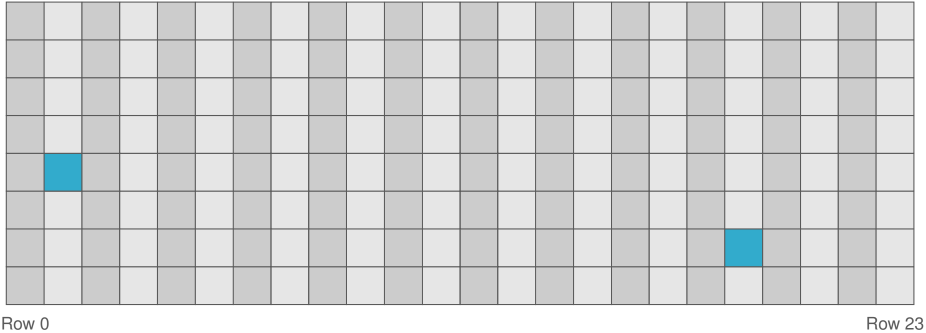


Search for page with flip



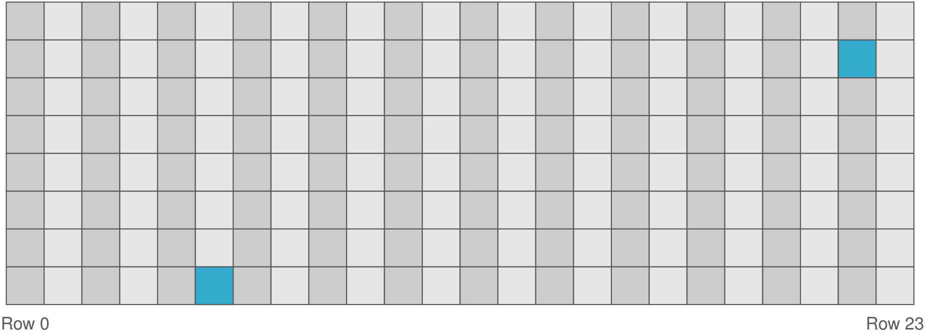
Hammering memory locations in different rows

Search for page with flip



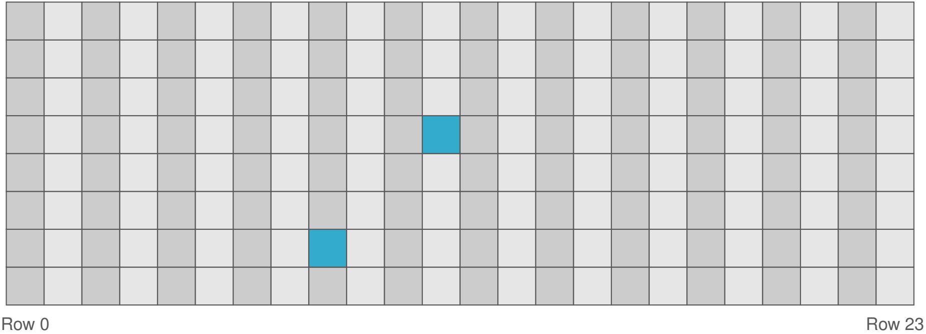
Hammering memory locations in different rows

Search for page with flip



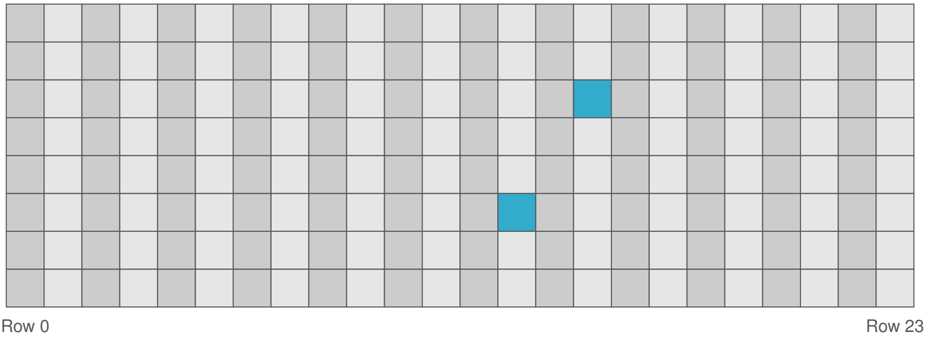
Hammering memory locations in different rows

Search for page with flip



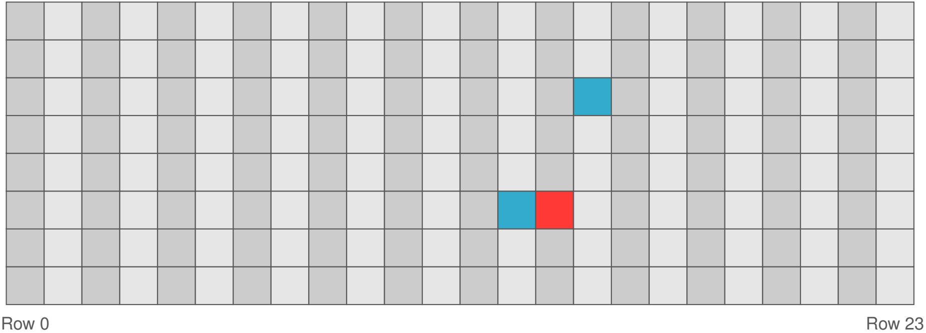
Hammering memory locations in different rows

Search for page with flip



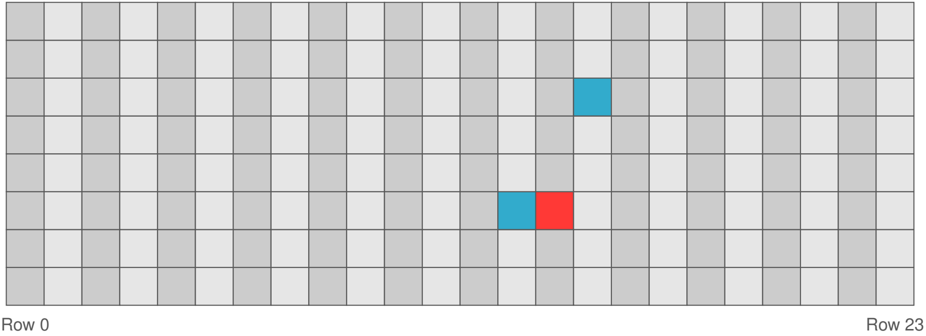
Hammering memory locations in different rows

Search for page with flip

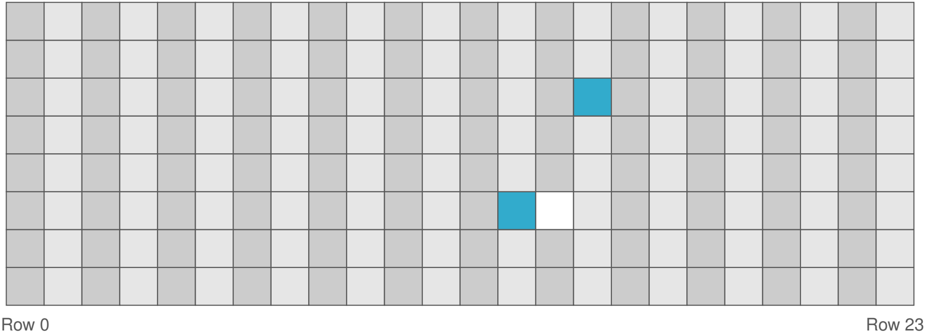


Hammering memory locations in different rows

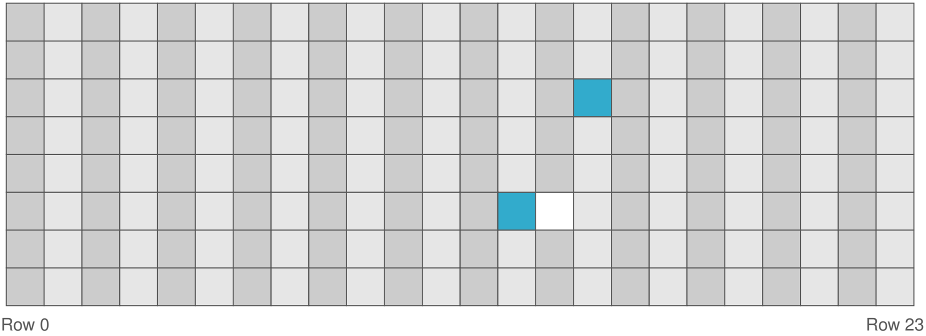
Release page with flip



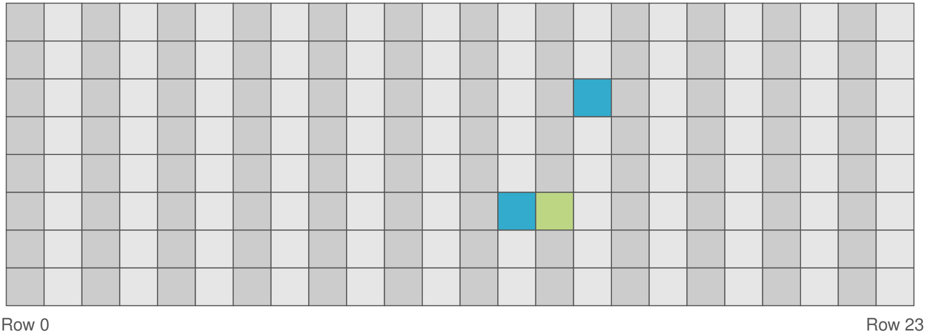
Release page with flip



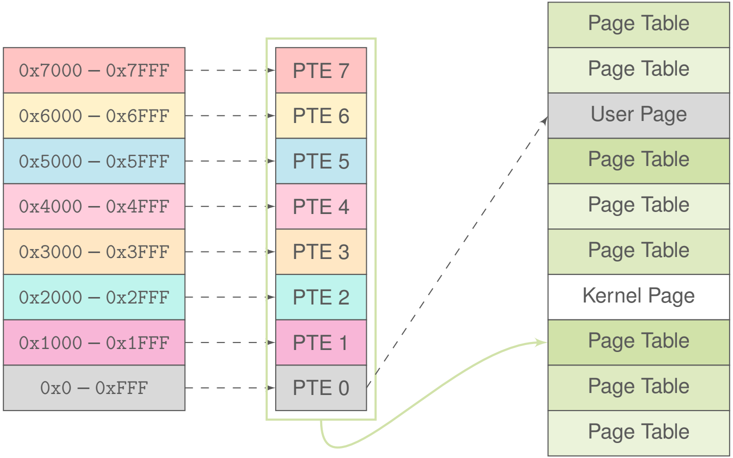
Fill all remaining memory with page tables



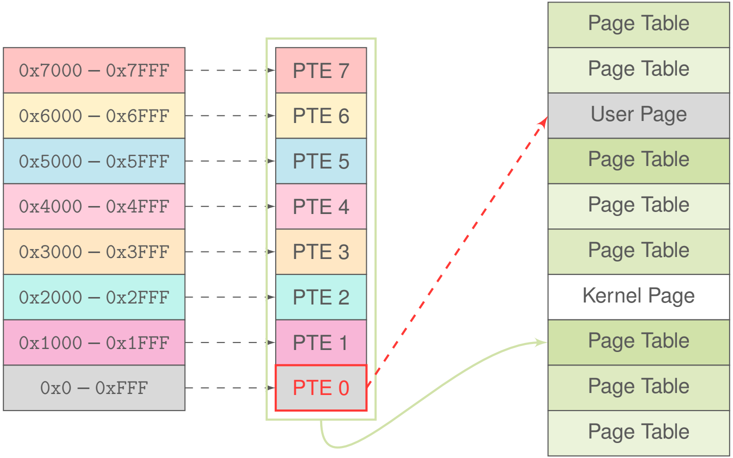
Fill all remaining memory with page tables



Page Table Manipulation



Page Table Manipulation

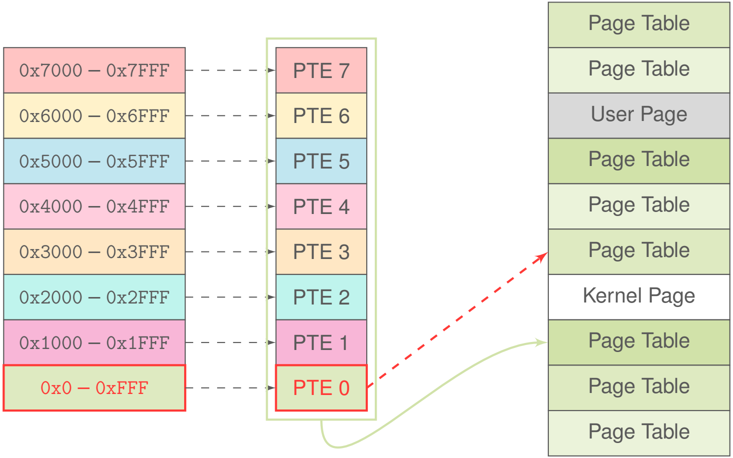


0x7000 – 0x7FFF
0x6000 – 0x6FFF
0x5000 – 0x5FFF
0x4000 – 0x4FFF
0x3000 – 0x3FFF
0x2000 – 0x2FFF
0x1000 – 0x1FFF
0x0 – 0xFFF

PTE 7
PTE 6
PTE 5
PTE 4
PTE 3
PTE 2
PTE 1
PTE 0

Page Table
Page Table
User Page
Page Table
Page Table
Page Table
Kernel Page
Page Table
Page Table
Page Table

Page Table Manipulation



Strategy: Flipping Page Table PPN bits

1. scan for flips
2. exhaust or massage memory to place a page table at target location
3. gain access to your own page table → kernel privileges

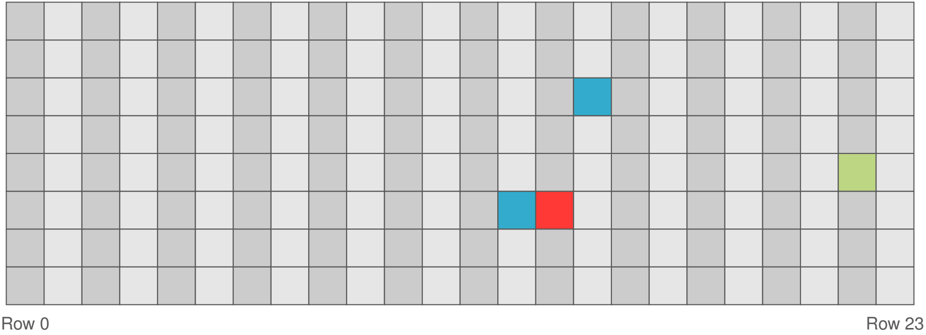
Flipping Page Table PPN bits

- idea from Seaborn and Dullien 2015
- same idea applied in several other works:
  - Rowhammer.js (Gruss, Maurice, and Mangard 2016)
  - One bit flips, one cloud flops (Xiao et al. 2016)
  - Drammer (Veen et al. 2016)

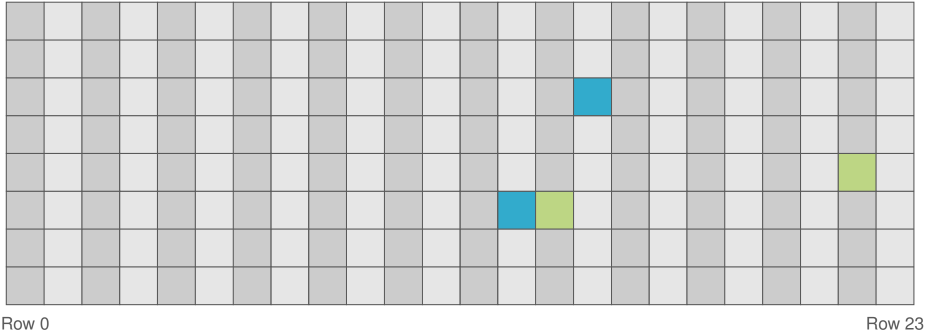
Post-Rowhammer Exploitation

- scan entire physical memory (very fast) and:
  - modify binary pages executed in root privileges (Xiao et al. 2016)
  - modify credential structs (Veen et al. 2016)
  - read keys (Xiao et al. 2016)
  - corrupt RSA signatures (Bhattacharya et al. 2016)
  - modify certificates
  - configurations
  - etc.
- pages are pretty unique: 32768 bits per page

Bit Flips + Page Deduplication

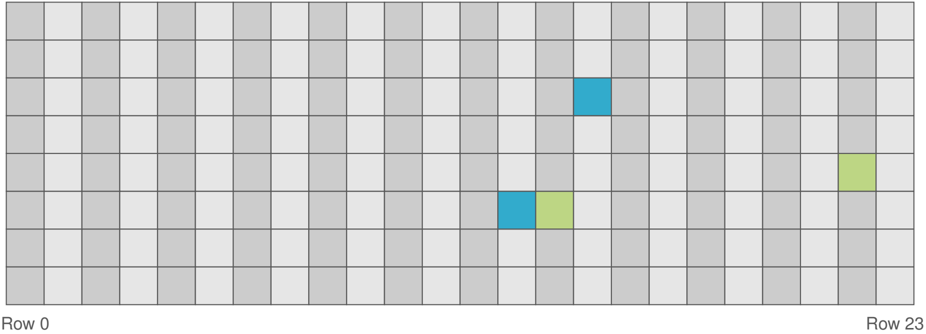


Bit Flips + Page Deduplication



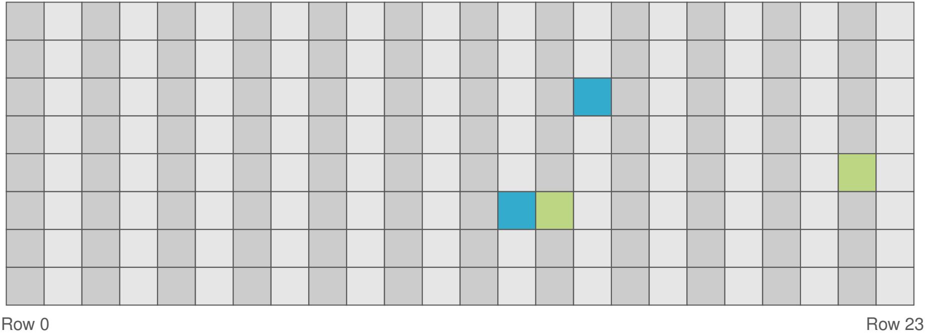
Page with bit flip is filled with target content

Bit Flips + Page Deduplication



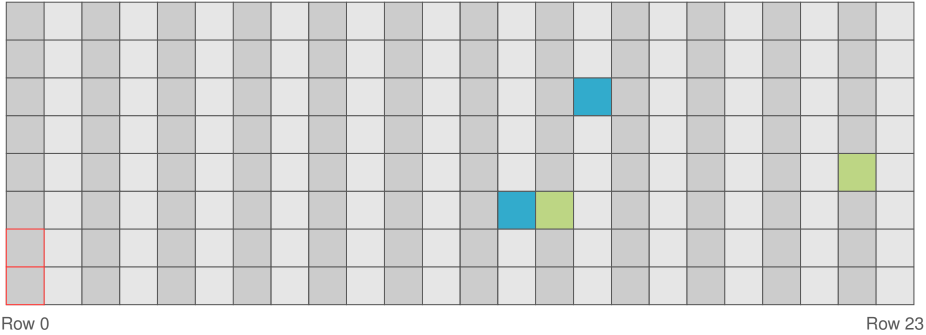
OS or hypervisor searches for duplicate pages

Bit Flips + Page Deduplication



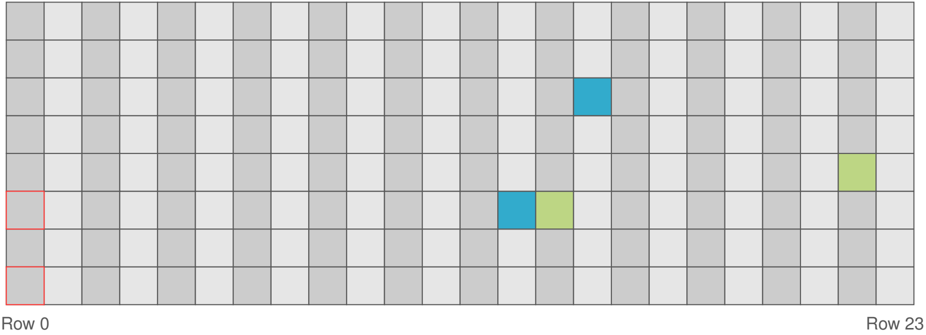
OS or hypervisor searches for duplicate pages

Bit Flips + Page Deduplication



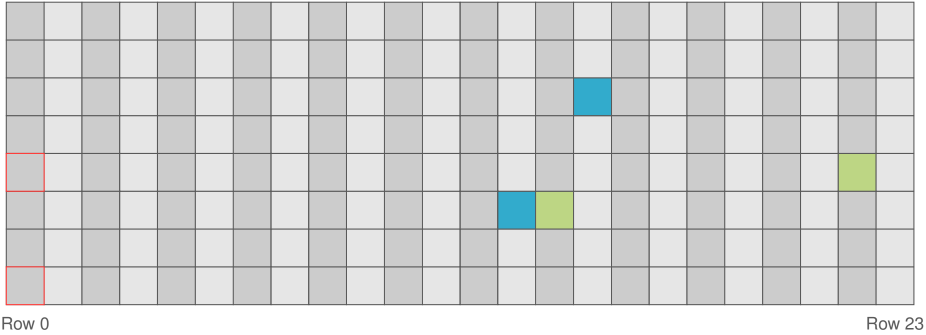
OS or hypervisor searches for duplicate pages

Bit Flips + Page Deduplication



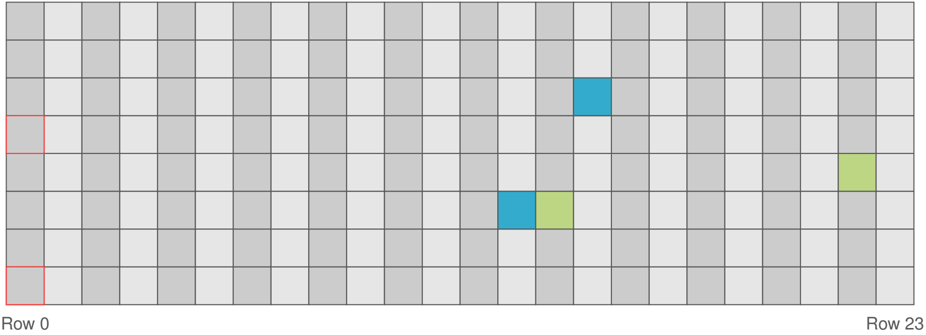
OS or hypervisor searches for duplicate pages

Bit Flips + Page Deduplication



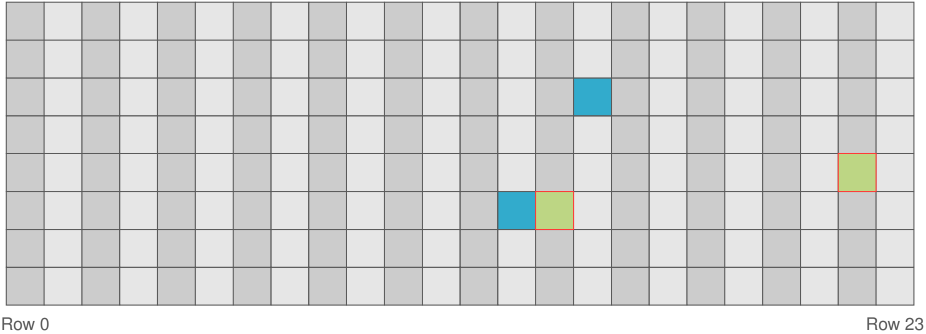
OS or hypervisor searches for duplicate pages

Bit Flips + Page Deduplication



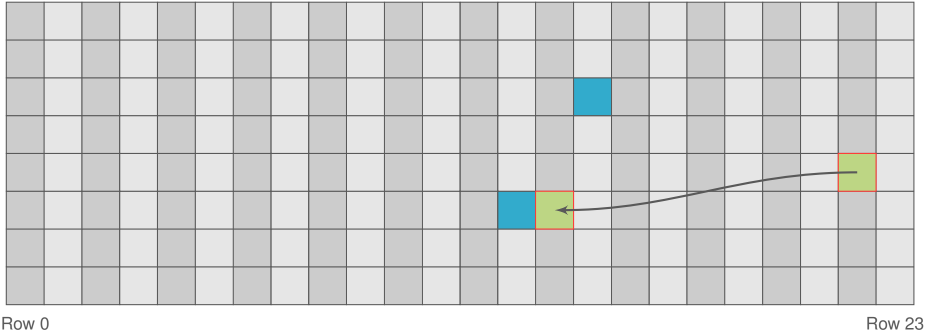
OS or hypervisor searches for duplicate pages

Bit Flips + Page Deduplication



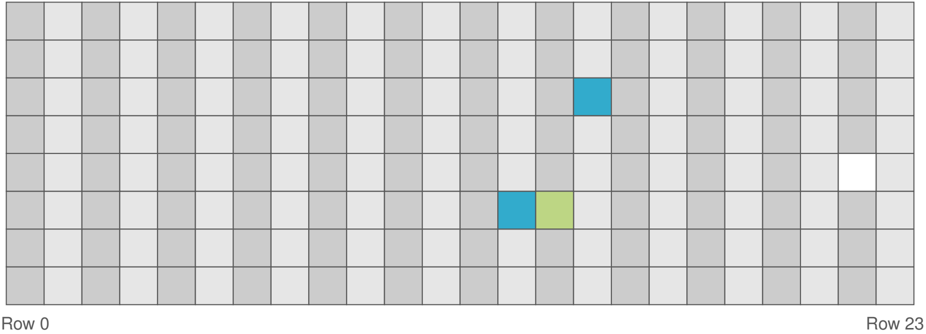
OS or hypervisor searches for duplicate pages

Bit Flips + Page Deduplication



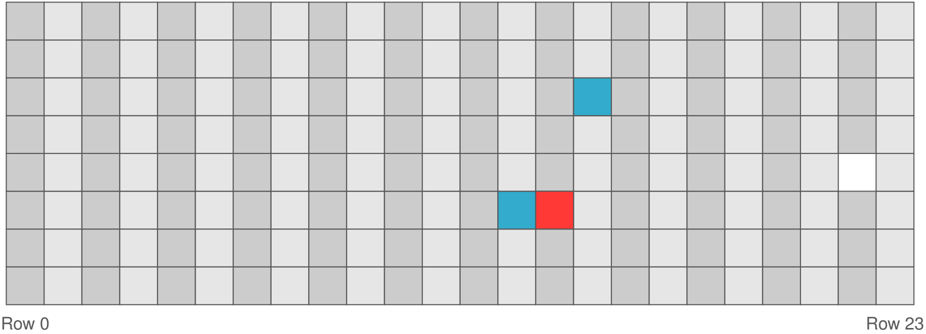
OS or hypervisor searches for duplicate pages

Bit Flips + Page Deduplication



Hammer again + flip again

Bit Flips + Page Deduplication



Strategy: Flipping in Deduplicated Pages

1. scan for flips
2. place content for deduplication so that flip can be exploited
3. perform the bit change through Rowhammer

Flipping in Deduplicated Pages

- idea from Bosman et al. 2016
  - change data type (double → pointer)
  - change pointer to good object → counterfeit object
- and from Razavi et al. 2016
  - corrupt authorized SSH keys
  - corrupt Debian update URLs + RSA public key file

4. How to mitigate Rowhammer?

Mitigations

Different mitigations have been proposed:

- detection vs prevention
- software vs hardware
- short-term vs long-term

Quick fixes

- no `clflush` instruction

Quick fixes

- no `clflush` instruction → Rowhammer.js

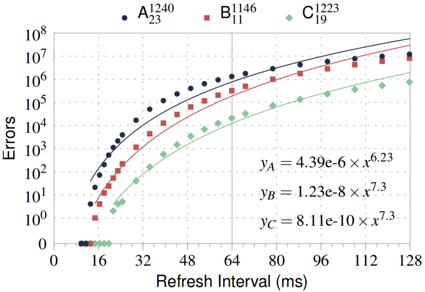
Quick fixes

- no `clflush` instruction → Rowhammer.js
- increase the refresh rate

Quick fixes

- no `clflush` instruction → Rowhammer.js
- increase the refresh rate

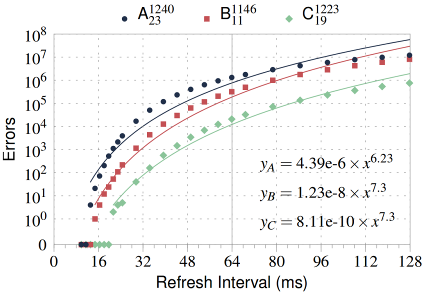
  → would need to be increased by 7× to eliminate all bit flips



Errors depending on refresh interval (Kim et al. 2014)

Quick fixes

- no `clflush` instruction → Rowhammer.js
- increase the refresh rate
  - → would need to be increased by 7× to eliminate all bit flips
  - → implementation: increased by 2× by BIOS vendors



Errors depending on refresh interval (Kim et al. 2014)

What about ECC?

- ECC protection: server can handle or correct single bit errors

What about ECC?

- ECC protection: server can handle or correct single bit errors
- no standard for event reporting

What about ECC?

- ECC protection: server can handle or correct single bit errors
- no standard for event reporting
- in practice (Lanteigne 2016)
  - common: server counts ECC errors and report only if they reach a threshold (e.g., $>$ 100 bit flips / hour)

What about ECC?

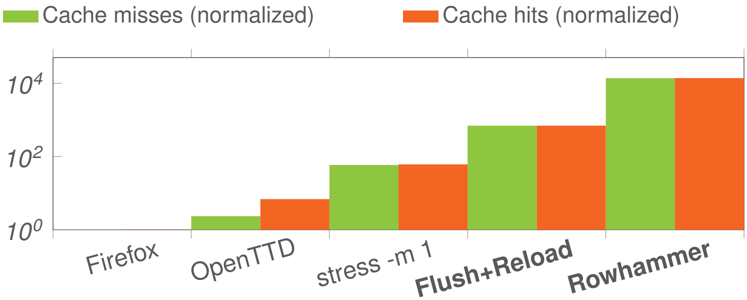
- ECC protection: server can handle or correct single bit errors
- no standard for event reporting
- in practice (Lanteigne 2016)
  - common: server counts ECC errors and report only if they reach a threshold (e.g., $>$ 100 bit flips / hour)
  - some server vendors never report errors to the OS

What about ECC?

- ECC protection: server can handle or correct single bit errors
- no standard for event reporting
- in practice (Lanteigne 2016)
  - common: server counts ECC errors and report only if they reach a threshold (e.g., $>$ 100 bit flips / hour)
  - some server vendors never report errors to the OS
  - one server did not even halt when bit flips were non-correctable

Detecting Rowhammer attacks

- Rowhammer: lots of cache misses that can be monitored with hardware performance counters (Herath et al. 2015; Gruss, Maurice, Wagner, et al. 2016; Chiappetta et al. 2015; Payer 2016)



Preventing Rowhammer attacks in hardware (1/3)

Original ideas from Kim et al. 2014

- making better DRAM chips that are not vulnerable,
- using error correcting codes (ECC)
- increasing the refresh rate
- remapping/retiring faulty cells after manufacturing
- identifying hammered rows at runtime and refreshing neighbors

Preventing Rowhammer attacks in hardware (1/3)

Original ideas from Kim et al. 2014

- making better DRAM chips that are not vulnerable,
- using error correcting codes (ECC)
- increasing the refresh rate
- remapping/retiring faulty cells after manufacturing
- identifying hammered rows at runtime and refreshing neighbors

→ expensive, performance overhead, or increased power consumption

Preventing Rowhammer attacks in hardware (2/3)

PARA - Probabilistic Adjacent Row Activation (Kim et al. 2014)

- one row closed → one adjacent row opened with low probability p

Preventing Rowhammer attacks in hardware (2/3)

PARA - Probabilistic Adjacent Row Activation (Kim et al. 2014)

- one row closed → one adjacent row opened with low probability p
- Rowhammer: one row opened and closed a high number of times N_{th}

Preventing Rowhammer attacks in hardware (2/3)

PARA - Probabilistic Adjacent Row Activation (Kim et al. 2014)

- one row closed → one adjacent row opened with low probability p
- Rowhammer: one row opened and closed a high number of times N_{th}
- statistically, neighbor rows are refreshed → no bit flip

Preventing Rowhammer attacks in hardware (2/3)

PARA - Probabilistic Adjacent Row Activation (Kim et al. 2014)

- one row closed → one adjacent row opened with low probability p
- Rowhammer: one row opened and closed a high number of times N_{th}
- statistically, neighbor rows are refreshed → no bit flip
- implementation at the memory controller level

Preventing Rowhammer attacks in hardware (2/3)

PARA - Probabilistic Adjacent Row Activation (Kim et al. 2014)

- one row closed → one adjacent row opened with low probability p
- Rowhammer: one row opened and closed a high number of times N_{th}
- statistically, neighbor rows are refreshed → no bit flip
- implementation at the memory controller level
- advantage: stateless → not expensive

Preventing Rowhammer attacks in hardware (2/3)

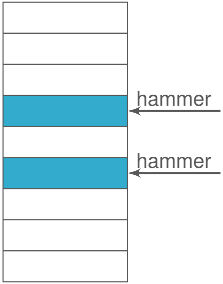
PARA - Probabilistic Adjacent Row Activation (Kim et al. 2014)

- one row closed → one adjacent row opened with low probability p
- Rowhammer: one row opened and closed a high number of times N_{th}
- statistically, neighbor rows are refreshed → no bit flip
- implementation at the memory controller level
- advantage: stateless → not expensive
- for $p = 0.001$ and $N_{th} = 100K$, experiencing one error in one year has a probability 9.4×10^{-14}

Preventing Rowhammer attacks in hardware (3/3)

Target Row Refresh (TRR)

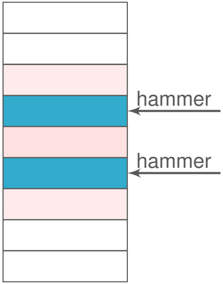
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold



Preventing Rowhammer attacks in hardware (3/3)

Target Row Refresh (TRR)

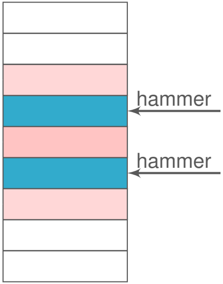
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold



Preventing Rowhammer attacks in hardware (3/3)

Target Row Refresh (TRR)

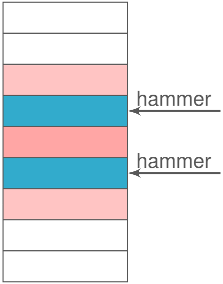
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold



Preventing Rowhammer attacks in hardware (3/3)

Target Row Refresh (TRR)

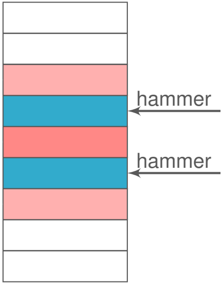
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold



Preventing Rowhammer attacks in hardware (3/3)

Target Row Refresh (TRR)

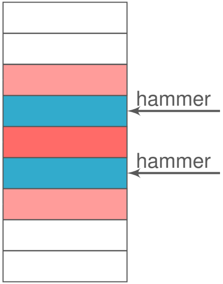
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold



Preventing Rowhammer attacks in hardware (3/3)

Target Row Refresh (TRR)

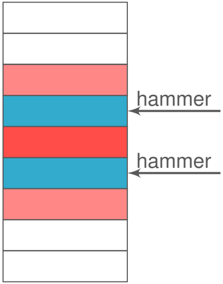
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold



Preventing Rowhammer attacks in hardware (3/3)

Target Row Refresh (TRR)

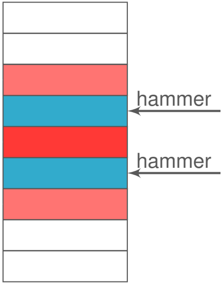
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold



Preventing Rowhammer attacks in hardware (3/3)

Target Row Refresh (TRR)

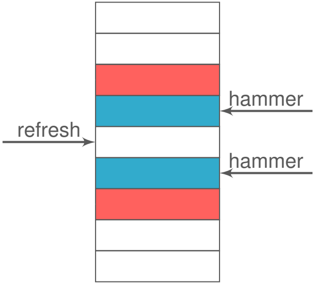
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold



Preventing Rowhammer attacks in hardware (3/3)

Target Row Refresh (TRR)

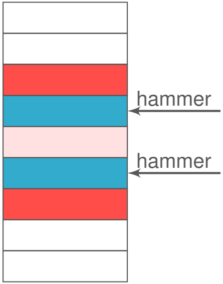
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold



Preventing Rowhammer attacks in hardware (3/3)

Target Row Refresh (TRR)

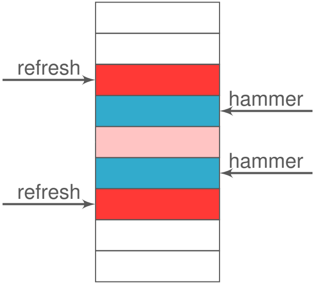
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold



Preventing Rowhammer attacks in hardware (3/3)

Target Row Refresh (TRR)

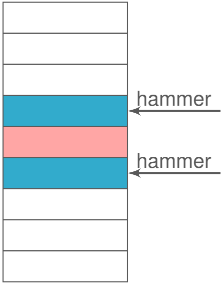
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold



Preventing Rowhammer attacks in hardware (3/3)

Target Row Refresh (TRR)

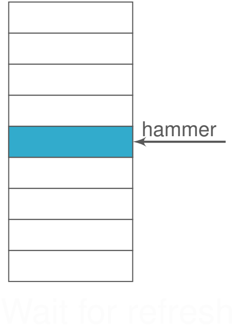
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold



Preventing Rowhammer attacks in software

“nohammer” kernel module Corbet 2016

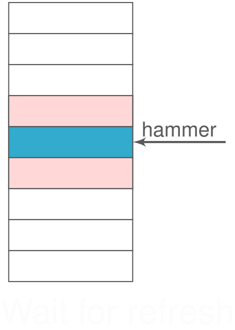
- refresh rate of 8 ms would prevent Rowhammer on most systems
- use PMC to measure cache misses per 64 ms interval
- limit cache miss rate to 1/8 of maximum



Preventing Rowhammer attacks in software

“nohammer” kernel module Corbet 2016

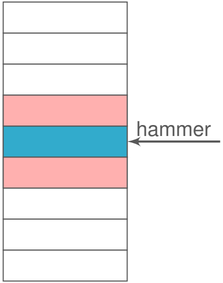
- refresh rate of 8 ms would prevent Rowhammer on most systems
- use PMC to measure cache misses per 64 ms interval
- limit cache miss rate to 1/8 of maximum



Preventing Rowhammer attacks in software

"nohammer" kernel module Corbet 2016

- refresh rate of 8 ms would prevent Rowhammer on most systems
- use PMC to measure cache misses per 64 ms interval
- limit cache miss rate to 1/8 of maximum

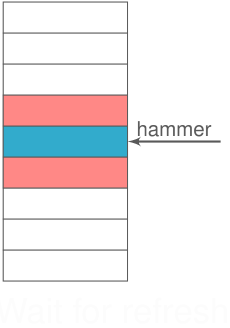


Wait for refresh

Preventing Rowhammer attacks in software

“nohammer” kernel module Corbet 2016

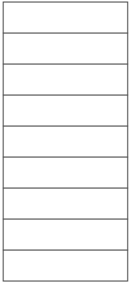
- refresh rate of 8 ms would prevent Rowhammer on most systems
- use PMC to measure cache misses per 64 ms interval
- limit cache miss rate to 1/8 of maximum



Preventing Rowhammer attacks in software

“nohammer” kernel module Corbet 2016

- refresh rate of 8 ms would prevent Rowhammer on most systems
- use PMC to measure cache misses per 64 ms interval
- limit cache miss rate to 1/8 of maximum



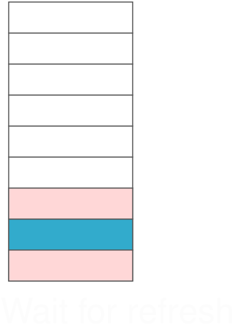
Wait for refresh

Wait for refresh

Preventing Rowhammer attacks in software

"nohammer" kernel module Corbet 2016

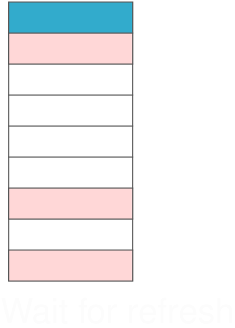
- refresh rate of 8 ms would prevent Rowhammer on most systems
- use PMC to measure cache misses per 64 ms interval
- limit cache miss rate to 1/8 of maximum



Preventing Rowhammer attacks in software

“nohammer” kernel module Corbet 2016

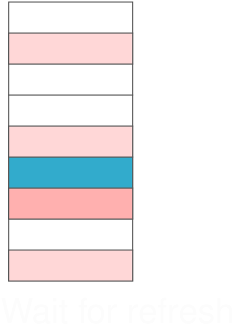
- refresh rate of 8 ms would prevent Rowhammer on most systems
- use PMC to measure cache misses per 64 ms interval
- limit cache miss rate to 1/8 of maximum



Preventing Rowhammer attacks in software

“nohammer” kernel module Corbet 2016

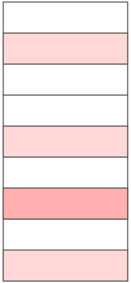
- refresh rate of 8 ms would prevent Rowhammer on most systems
- use PMC to measure cache misses per 64 ms interval
- limit cache miss rate to 1/8 of maximum



Preventing Rowhammer attacks in software

“nohammer” kernel module Corbet 2016

- refresh rate of 8 ms would prevent Rowhammer on most systems
- use PMC to measure cache misses per 64 ms interval
- limit cache miss rate to 1/8 of maximum



Wait for refresh

Wait for refresh

Preventing Rowhammer attacks in software

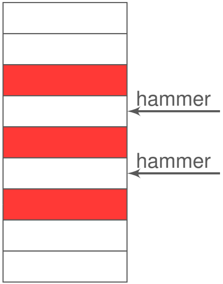
MASCAT - Stopping Microarchitectural Attacks Before Execution
(Irazoqui et al. 2016)

- static analysis of the binary
- detect suspicious instruction sequences (`clflush`, `rdtsc`, fences, . . .)
- open problem: false positives

Preventing Rowhammer attacks in software

ANVIL (Aweke et al. 2016)

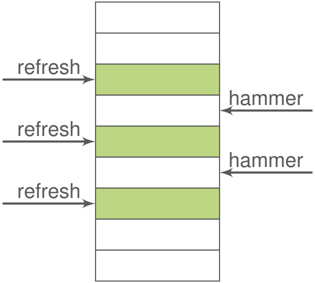
- uses performance counters to detect rowhammer
- activate rows neighbor rows to prevent flips
- similar as PARA, but in software



Preventing Rowhammer attacks in software

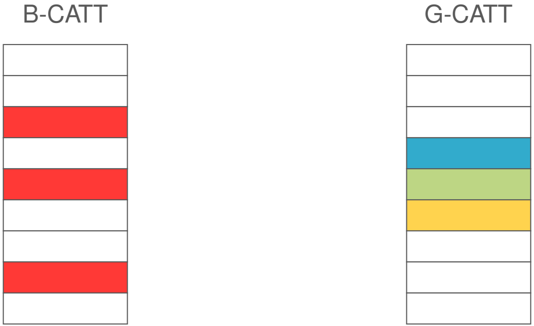
ANVIL (Aweke et al. 2016)

- uses performance counters to detect rowhammer
- activate rows neighbor rows to prevent flips
- similar as PARA, but in software



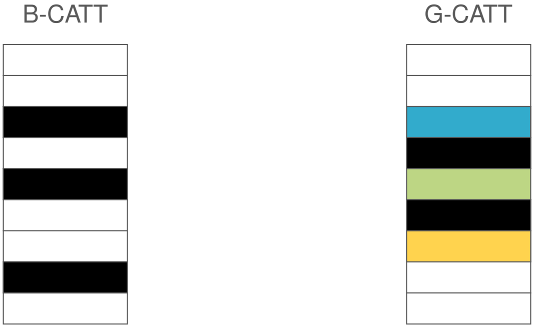
Preventing Rowhammer attacks in software

- B-CATT: disable vulnerable physical memory (Brasser et al. 2016)
- G-CATT: isolate security domains in physical memory based on potential vulnerability (Brasser et al. 2016)



Preventing Rowhammer attacks in software

- B-CATT: disable vulnerable physical memory (Brasser et al. 2016)
- G-CATT: isolate security domains in physical memory based on potential vulnerability (Brasser et al. 2016)



5. Conclusion

Conclusion

- Rowhammer attacks are easy to mount
- works on most systems (if you know the DRAM mapping)
- most countermeasures are too expensive or ineffective

I want to try!

- `https://github.com/IAIK/DRAMA`
  Reverse-engineering tool for DRAM addressing
- `https://github.com/IAIK/rowhammerjs`
  Adaptation of double-sided hammering + hammering in JavaScript
- `https://github.com/IAIK/armageddon`
  `libflush` provides performant eviction strategies
- `https://github.com/vusec/drammer`
  Hammering with ION on ARM

Thank you!

Contact

- 🐦 @lavados
- 🐦 @BloodyTangerine

Rowhammer Attacks: A Walkthrough Guide

Daniel Gruss & Clémentine Maurice, Graz University of Technology

May 4, 2017 — RuhrSec 2017

References I

Aweke, Zelalem Birhanu, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin (2016). “ANVIL: Software-based protection against next-generation rowhammer attacks”. In: ACM SIGPLAN Notices 51.4, pp. 743–755.

Bhattacharya, Sarani and Debdeep Mukhopadhyay (2016). “Curious case of Rowhammer: Flipping Secret Exponent Bits using Timing Analysis”. In: CHES’16.

Bosman, Erik, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida (2016). “Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector”. In: S&P’16.

References II

Brasser, Ferdinand, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi (2016). “CAn’t Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks”. In: arXiv:1611.08396.

Chiappetta, Marco, Erkay Savas, and Cemal Yilmaz (2015). Real time detection of cache-based side-channel attacks using Hardware Performance Counters. Cryptology ePrint Archive, Report 2015/1034.

Corbet, Jonathan (Oct. 2016). Defending against Rowhammer in the kernel. URL: `https://lwn.net/Articles/704920/`.

Gruss, Daniel, Clémentine Maurice, and Stefan Mangard (2016). “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: DIMVA’16.

Gruss, Daniel, Clémentine Maurice, Klaus Wagner, and Stefan Mangard (2016). “Flush+Flush: A Fast and Stealthy Cache Attack”. In: DIMVA’16.

References III

Herath, Nishad and Anders Fogh (Aug. 2015). “These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security”. In: Black Hat 2015 Briefings. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf.

Irazoqui, Gorka, Thomas Eisenbarth, and Berk Sunar (2016). “MASCAT: Stopping Microarchitectural Attacks Before Execution”. In: Cryptology ePrint Archive: Report 2016/1196.

Kim, Yoongu, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu (2014). “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors”. In: ISCA’14.

References IV

Lanteigne, Mark (2016). How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. URL: http://www.thirdio.com/rowhammer.pdf.

Payer, Matthias (2016). “HexPADS: a platform to detect “stealth” attacks”. In: ESSoS’16.

Pessl, Peter, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard (2016). “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: USENIX Security Symposium.

Qiao, Rui and Mark Seaborn (2016). “A new approach for rowhammer attacks”. In: HOST 2016.

Razavi, Kaveh, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos (2016). “Flip Feng Shui: Hammering a Needle in the Software Stack”. In: USENIX Security Symposium.

References V

Seaborn, Mark (May 2015). How physical addresses map to rows and banks in DRAM. URL: http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html.

Seaborn, Mark and Thomas Dullien (2015). “Exploiting the DRAM rowhammer bug to gain kernel privileges”. In: Black Hat 2015 Briefings.

Veen, Victor van der, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida (2016). “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms”. In: CCS’16.

Xiao, Yuan, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu (2016). “One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation ”. In: USENIX Security Symposium.