Micro-Architectural Fault Attacks

Clémentine Maurice, CNRS
May 5, 2019—FICHSA, Ben-Gurion University of the Negev, Beer Sheva, Israel
Fault attacks

- side channels: non-legitimate "read primitive"

- can we have a "write primitive"?
- yes! fault attacks!
- why is that a problem?
  - hardware can bypass software security mechanisms
  - software cannot trust hardware anymore
Fault attacks

- side channels: non-legitimate “read primitive”
- can we have a “write primitive”?
Fault attacks

- side channels: non-legitimate “read primitive”
- can we have a “write primitive”? 
- yes! fault attacks!
Fault attacks

- side channels: non-legitimate “read primitive”
- can we have a “write primitive”?
- yes! fault attacks!
- why is that a problem?
Fault attacks

- side channels: non-legitimate “read primitive”
- can we have a “write primitive”? 
- yes! fault attacks!
- why is that a problem? hardware can bypass software security mechanisms
  → software cannot trust hardware anymore
Software-based fault attacks?

- until 2014, fault attacks required physical access: changes in temperature, clock, voltage, electric/magnetic fields, ...

---

Software-based fault attacks?

- until 2014, fault attacks required physical access: changes in temperature, clock, voltage, electric/magnetic fields, ...
- core idea of fault attacks: **pushing hardware beyond nominal operating conditions**

Software-based fault attacks?

- until 2014, fault attacks required physical access: changes in temperature, clock, voltage, electric/magnetic fields, ...
- core idea of fault attacks: pushing hardware beyond nominal operating conditions
- software can do that too!

---

Software-based fault attacks?

- until 2014, fault attacks required physical access: changes in temperature, clock, voltage, electric/magnetic fields, ...
- core idea of fault attacks: pushing hardware beyond nominal operating conditions
- software can do that too!
- 2014: Rowhammer (Kim et al.)

---

Software-based fault attacks?

• until 2014, fault attacks required physical access: changes in temperature, clock, voltage, electric/magnetic fields, ...
• core idea of fault attacks: pushing hardware beyond nominal operating conditions
• software can do that too!
• 2014: Rowhammer (Kim et al.)
• first reaction: ”it’s a reliability issue, not a security issue”

Software-based fault attacks?

• until 2014, fault attacks required physical access: changes in temperature, clock, voltage, electric/magnetic fields, ...
• core idea of fault attacks: pushing hardware beyond nominal operating conditions
• software can do that too!
• 2014: Rowhammer (Kim et al.)
• first reaction: ”it’s a reliability issue, not a security issue”
• 2015: Google Project Zero showed a sandbox escape and a privilege escalation attack using Rowhammer

• Background on DRAM and Rowhammer
• How do we get bip flips?
• How do we target memory accesses?
• Can we exploit these bit flips?
• Countermeasures
Background on DRAM and Rowhammer
DRAM organization

channel 0

channel 1
DRAM organization

channel 0
back of DIMM: rank 1

channel 1
front of DIMM: rank 0
DRAM organization

channel 0

back of DIMM: rank 1

channel 1

front of DIMM: rank 0

chip
DRAM organization

chip

bank 0

row 0

row 1

row 2

...

row 32767

row buffer
DRAM organization

chip

bank 0

row 0
row 1
row 2
...
row 32767

row buffer

64k cells
1 capacitor,
1 transistor each
• DRAM internally is only capable of reading entire rows
• DRAM internally is only capable of reading entire rows
• capacitors in cells discharge when you “read the bits”
• buffer the bits when reading them from the cells
• write the bits back to the cells when you’re done
• DRAM internally is only capable of reading entire rows
• capacitors in cells discharge when you “read the bits”
• buffer the bits when reading them from the cells
• write the bits back to the cells when you’re done
→ row buffer
How reading from DRAM works

CPU wants to access row 1
How reading from DRAM works

DRAM bank

CPU wants to access row 1
→ row 1 activated
How reading from DRAM works

CPU wants to access row 1
→ row 1 activated
→ row 1 copied to row buffer
How reading from DRAM works

CPU wants to access row 1
→ row 1 activated
→ row 1 copied to row buffer
How reading from DRAM works

CPU wants to access row 2
How reading from DRAM works

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

... row buffer

activate

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

return
How reading from DRAM works

- CPU wants to access row 2
  - row 2 activated
  - row 2 copied to row buffer
  - slow (row conflict)
How reading from DRAM works

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
...
1 1 1 1 1 1 1 1 1 1 1 1 1 1

row buffer

CPU wants to access row 2—again
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
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

1111111111111
1111111111111
1111111111111
1111111111111
...
1111111111111

row buffer

row buffer = cache
• cells leak $\rightarrow$ repetitive refresh necessary
• refresh $\approx$ reading (destructive) + writing same data again
• maximum interval between refreshes to guarantee data integrity
DRAM refresh

- cells leak → repetitive refresh necessary
- refresh ≈ reading (destructive) + writing same data again
- maximum interval between refreshes to guarantee data integrity
- cells leak faster upon proximate accesses → fault attack
“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

<table>
<thead>
<tr>
<th>DRAM bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>111111111111111</td>
</tr>
<tr>
<td>111111111111111</td>
</tr>
<tr>
<td>111111111111111</td>
</tr>
<tr>
<td>111111111111111</td>
</tr>
<tr>
<td>111111111111111</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>111111111111111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>row buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>111111111111111</td>
</tr>
</tbody>
</table>
“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
“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
“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
"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
“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
Requirements

Memory accesses must be

- **uncached**: reach DRAM
- **fast**: race against the next row refresh
- **targeted**: reach specific row
Issue #1: How do we get bit flips?
Impact of the CPU cache

- only non-cached accesses reach DRAM
- original attacks use `clflush` instruction
  - flush line from cache
  - next access will be served from DRAM
How to reach DRAM?

1. **clflush** instruction → original paper (Kim et al.)
2. cache eviction (Gruss et al., Aweke et al.)
3. non-temporal accesses (Qiao et al.)
4. uncached memory (van der Veen et al.)
5. remotely (Lipp et al., Tatar et al.)

begin:
    mov (X), %eax // read from address X
    mov (Y), %ebc // read from address Y
    clflush (X) // flush cache for address X
    clflush (Y) // flush cache for address Y
    jmp begin

#1 Hammering with clflush

cache set 1

cache set 2

DRAM bank
#1 Hammering with clflush

- cache set 1
- cache set 2
- clflush
- clflush
- clflush
- DRAM bank
#1 Hammering with clflush

cache set 1

cache set 2

DRAM bank
#1 Hammering with clflush

Cache set 1

Cache set 2

DRAM bank
#1 Hammering with clflush

cache set 1

cache set 2

DRAM bank

reload
#1 Hammering with clflush

cache set 1

cache set 2

DRAM bank

reload

reload

reload

reload
#1 Hammering with clflush

cache set 1

clflush

cache set 2

clflush

DRAM bank
#1 Hammering with clflush

- cache set 1
- cache set 2
- DRAM bank

reload

reload
#1 Hammering with clflush

- cache set 1
- cache set 2
- clflush
- clflush
- clflush
- DRAM bank
#1 Hammering with clflush

- Cache set 1
  - Cache set 2
  - DRAM bank

Reload
#1 Hammering with `clflush`

cache set 1

cache set 2

DRAM bank
#1 Hammering with clflush

cache set 1

cache set 2

DRAM bank
#1 Hammering with clflush

Cache set 1

Cache set 2

DRAM bank

wait for it...
Hammering with clflush

#1

- cache set 1
- cache set 2
- DRAM bank
- reload
- bit flip!
Flush, reload, flush, reload...

- the core of Rowhammer is essentially a Flush+Reload loop
- as much an attack on DRAM as on cache
• idea: avoid `clflush` to be independent of specific instructions
  → no `clflush` in JavaScript

The idea is to avoid the `clflush` instruction to be independent of specific instructions. In JavaScript, there is no `clflush` instruction.

What can we do?

---

• idea: avoid \texttt{clflush} to be independent of specific instructions
  → no \texttt{clflush} in JavaScript

• what can we do?

• our approach: use regular memory accesses for eviction
  → techniques from cache attacks
  → Rowhammer using \texttt{Prime+Probe}

#2 Hammering with cache eviction

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

- what can we do?

- our approach: use regular memory accesses for eviction
  → techniques from cache attacks
  → Rowhammer using Prime+Probe

- beware of the replacement policy!

#2 Hammering with cache eviction

- Cache set 1
- Cache set 2

DRAM bank
#2 Hammering with cache eviction

[Diagram showing cache sets and load operations]
#2 Hammering with cache eviction

DRAM bank

cache set 1

cache set 2

load

prefetch
#2 Hammering with cache eviction

- DRAM bank
- cache set 1
- cache set 2
- load
#2 Hammering with cache eviction

- **Cache set 1**
  -DRAM bank
- **Cache set 2**
  -DRAM bank
#2 Hammering with cache eviction

- DRAM bank
- cache set 1
- cache set 2
- load
- load
#2 Hammering with cache eviction

- Cache set 1
- Cache set 2
- Load

DRAM bank
#2 Hammering with cache eviction

DRAM bank

cache set 1

load

cache set 2

load
#2 Hammering with cache eviction
#2 Hammering with cache eviction

cache set 1

repeat!

cache set 2

DRAM bank
#2 Hammering with cache eviction

- cache set 1
- cache set 2
- DRAM bank
- reload
- wait for it...
#2 Hammering with cache eviction

![Diagram showing cache sets and DRAM bank with bit flip.]

- cache set 1
- cache set 2
- DRAM bank

bit flip!
#2 Hammering with cache eviction: Evaluation on Haswell

![Graph](image)

- **clflush**
- **Evict (Native)**
- **Evict (JavaScript)**

- **x-axis:** Refresh interval in µs (BIOS configuration)
- **y-axis:** Number of bit flips within 15 minutes

- The graph compares the number of bit flips between different refresh intervals for various eviction methods.
non-temporal accesses: data accessed just once, not in the future

NTA instructions → **bypass cache** to minimize cache pollution

---

• non-temporal accesses: data accessed just once, not in the future
• NTA instructions $\rightarrow$ **bypass cache** to minimize cache pollution
• issue: NT stores to 1 address are combined at write-combining buffer
• only last write goes to DRAM $\rightarrow$ 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
- issue: NT stores to 1 address are combined at write-combining buffer
- only last write goes to DRAM → rate not sufficient
- solution: following cached access to same address

---

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

begin:
    movnti %eax, (X)
    movnti %eax, (Y)
    mov %eax, (X)
    mov %eax, (Y)
    jmp begin
Sometimes, everything fails, e.g., on mobile devices

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

- ARMv7 flush instruction is privileged

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

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

• ION: memory management since Android 4.0
• apps can use /dev/ion for uncached, physically contiguous memory
• no privilege and no permission needed
• previous work: some code execution (even JS)
• how about remote attacks, i.e., triggered by network packets?
  • Tatar et al. use RDMA, fast network communication that does not involve the CPU
  • Lipp et al. use Intel CAT that restricts cache allocation to a subset of cache ways for QoS

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:

- Still bit flips (Pessl et al.)

---


Issue #2: How do we target accesses?
Physical addresses and DRAM

- fixed map: physical addresses → DRAM cells
- undocumented for Intel

Physical addresses and DRAM

- fixed map: physical addresses → DRAM cells
- undocumented for Intel → reverse-engineered by Pessl et al.

Memory controller policies

• **open-page policy**: keep row opened and buffered
  • low latency for subsequent accesses to same row
  • high latency for accesses to any other row

---

Memory controller policies

- **open-page policy**: keep row opened and buffered
  - low latency for subsequent accesses to same row
  - high latency for accesses to any other row

- **close-page policy**: immediately close row, ready to open a new row
  - medium latency for accesses to any row
  - perform better on multi-core systems

“Double-sided hammering”

With an open-page policy, you need to alternate accesses to two rows

<table>
<thead>
<tr>
<th>DRAM bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

row buffer
With an open-page policy, you need to **alternate accesses to two rows**
**“Double-sided hammering”**

With an open-page policy, you need to **alternate accesses to two rows**

![Diagram showing DRAM bank and row buffer with binary data and activation and copy arrows]
With an open-page policy, you need to alternate accesses to two rows.
“Double-sided hammering”

With an open-page policy, you need to alternate accesses to two rows.
"Double-sided hammering"

With an open-page policy, you need to alternate accesses to two rows.

![Diagram showing DRAM bank with alternating accesses to two rows, indicating bit flips in row 2! and row buffer.]
With a close-page policy, you can **hammer a single row**
“One-location hammering”

With a close-page policy, you can hammer a single row

[Diagram of DRAM bank and row buffer]

activate

copy
“One-location hammering”

With a close-page policy, you can hammer a single row.

DRAM bank

111111111111111
111111111111111
111111111111111
111111111111111
111111111111111
...
111111111111111

row buffer
“One-location hammering”

With a close-page policy, you can hammer a single row

![Diagram of DRAM bank and row buffer with binary data]

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

...
With a close-page policy, you can hammer a single row

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 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
```
With a close-page policy, you can **hammer a single row**

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 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
```

row buffer

bit flips in row 2!
Issue #3: Can we exploit these bit flips?
How to exploit random bit flips?

- Rowhammer was deemed non-exploitable and only a reliability issue
How to exploit random bit flips?

- Rowhammer was deemed non-exploitable and only a reliability issue
- bit flips are not random → highly reproducible flip pattern!
- ideas for exploitation
  1. bit flip in data structure, e.g., page table
  2. bit flip in instruction opcode
  3. bit flip in signature (→ fault-based cryptanalysis)
Bit flips in page tables

General idea

1. allocate a large chunk of memory
2. scan for “good” flips with your own buffer
3. return that particular area of memory to the OS
4. force OS to place data structure there → page tables are good for this
5. trigger bit flip again
6. profit

Bit flips in page tables

- x86 page tables entries (PTEs) control access to physical memory
- bit flip in a PTE’s physical page number can give a process access to a different physical page
Bit flips in page tables

- x86 page tables entries (PTEs) control access to physical memory
- bit flip in a PTE’s physical page number can give a process access to a different physical page
- aim of exploit: get access to a page table → gives access to all of physical memory
Bit flips in page tables

- x86 page tables entries (PTEs) control access to physical memory
- bit flip in a PTE’s **physical page number** can give a process access to a different physical page
- aim of exploit: get access to a page table → gives access to **all of physical memory**
- maximize chances that a bit flip is useful by “**spraying**” physical memory with page tables
Bit flips in page tables

Page table: 4k page containing array of 512 PTEs (64 bits each)
Bit flips in page tables

Page table: 4k page containing array of 512 PTEs (64 bits each)

Could flip

• “writable” permission bit (RW): 1 bit → 2% chance
Bit flips in page tables

Page table: 4k page containing array of 512 PTEs (64 bits each)

Could flip

- “writable” permission bit (RW): 1 bit → 2% chance
- physical page number: 20 bits on 4GB system → 31% chance
Bit flips in page tables

- mapping a file with read-write permissions?
Bit flips in page tables

- mapping a file with read-write permissions?
  → indirection via page tables
Bit flips in page tables

- mapping a file with read-write permissions?
  → indirection via page tables
- repeatedly mapping a file with read-write permissions?
Bit flips in page tables

- mapping a file with read-write permissions?
  → indirection via page tables
- repeatedly mapping a file with read-write permissions?
  → more PTEs in physical memory!
Bit flips in page tables

- mapping a file with read-write permissions?
  → indirection via page tables
- repeatedly mapping a file with read-write permissions?
  → more PTEs in physical memory!
- we can fill physical memory with PTEs
Bit flips in page tables

- physical memory is filled with PTEs

virtual address space

physical memory
Bit flips in page tables

- physical memory is filled with PTEs
Bit flips in page tables

- Physical memory is filled with PTEs
- If a bit flips in the right place in the PTE...

• Use that to map any memory read/write, including kernel memory!
Bit flips in page tables

- physical memory is filled with PTEs
- if a bit flips in the right place in the PTE...
- ... the corresponding virtual address now points to a wrong physical page, with RW access, with a great chance the page contains a PT itself
Bit flips in page tables

- physical memory is filled with PTEs
- if a bit flips in the right place in the PTE...
- ... the corresponding virtual address now points to a wrong physical page, with RW access, with a great chance the page contains a PT itself
- use that to map any memory read/write
Bit flips in page tables

- physical memory is filled with PTEs
- if a bit flips in the right place in the PTE...
- ... the corresponding virtual address now points to a wrong physical page, with RW access, with a great chance the page contains a PT itself
- use that to map any memory read/write
- including kernel memory
  → privilege escalation
Bit flips in instruction opcode

- some applications perform actions as root
- can be used by unprivileged users

Bit flips in instruction opcode

- some applications perform actions as root
- can be used by unprivileged users
- ping, mount, sudo

Bit flips in instruction opcode

JE

0 1 1 1 0 1 0 0

HLT

1 1 1 1 0 1 0 0

bit flip in conditional jump
bypass password check
Bit flips in instruction opcode

JE

0 1 1 1 0 1 0 0

XORB

0 0 1 1 0 1 0 0

bit flip in conditional jump
bypass password check
Bit flips in instruction opcode

JE

PUSHQ

0 1 1 1 0 1 0 0

0 1 0 1 0 1 0 0

bit flip in conditional jump
bypass password check
Bit flips in instruction opcode

JE

0 1 1 1 0 1 0 0

<prefix>

0 1 1 0 0 1 0 0
Bit flips in instruction opcode

JE

0 1 1 1 0 1 0 0

→

JL

0 1 1 1 1 1 1 0 0
Bit flips in instruction opcode

JE

JO

0 1 1 1 0 1 0 0

0 1 1 1 1 0 0 0

bit flip in conditional jump
bypass password check
Bit flips in instruction opcode

JE

0 1 1 1 0 1 0 0

JBE

0 1 1 1 0 1 1 0

• bit flip in conditional jump
• bypass password check
Bit flips in instruction opcode

- Bit flip in conditional jump → bypass password check
Bit flips in instruction opcode

- not just conditional jump
- other targets include:
  - comparisons
  - addresses of memory loads/stores
- analysis of sudo → 29 possible bit flips to bypass password check
Countermeasures
Different countermeasures have been proposed:

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

1. no `clflush` instruction
Quick fixes

1. no `clflush` instruction →
   Rowhammer.js
Quick fixes

1. no clflush instruction → Rowhammer.js
2. increase the refresh rate
Quick fixes

1. no **clflush** instruction → Rowhammer.js
2. increase the refresh rate
   → would need to be increased by $7 \times$ to eliminate all bit flips

Errors depending on refresh interval ([Kim+14])

\[
y_A = 4.39e-6 \times x^{6.23} \\
y_B = 1.23e-8 \times x^{7.3} \\
y_C = 8.11e-10 \times x^{7.3}
\]
Quick fixes

1. no **clflush** instruction $\rightarrow$ Rowhammer.js
2. increase the refresh rate
   $\rightarrow$ would need to be increased by $7\times$ to eliminate all bit flips
   $\rightarrow$ implementation: increased by $2\times$ by BIOS vendors

Errors depending on refresh interval ([Kim+14])
• ECC protection: server can handle or correct single bit errors

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

---

ECC

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

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

• ECC protection: server can handle or correct single bit errors
• **no standard** for event reporting
• in practice
  • 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 ([Her+15; Gru+16a; Chi+15; Pay16])
Preventing Rowhammer attacks in hardware (1/3)

Original ideas from [Kim+14]

- 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+14]

- 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+14])

- one row closed $\rightarrow$ one adjacent row opened with low probability $p$
PARA - Probabilistic Adjacent Row Activation ([Kim+14])

- one row closed $\rightarrow$ one adjacent row opened with low probability $p$
- Rowhammer: one row opened and closed a high number of times $N_{th}$
PARA - Probabilistic Adjacent Row Activation ([Kim+14])

- one row closed $\rightarrow$ 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 $\rightarrow$ no bit flip
PARA - Probabilistic Adjacent Row Activation ([Kim+14])

- 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
PARA - Probabilistic Adjacent Row Activation ([Kim+14])

- one row closed $\rightarrow$ 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 $\rightarrow$ no bit flip
- implementation at the memory controller level
- advantage: stateless $\rightarrow$ not expensive
Preventing Rowhammer attacks in hardware (2/3)

PARA - Probabilistic Adjacent Row Activation ([Kim+14])

- 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 $9.4 \times 10^{-14}$
Target Row Refresh (TRR)

- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold
Target Row Refresh (TRR)

- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold
Target Row Refresh (TRR)

- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold
Target Row Refresh (TRR)

- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold
Target Row Refresh (TRR)

- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold
Target Row Refresh (TRR)
- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold
Target Row Refresh (TRR)

- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold
Target Row Refresh (TRR)

- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold
Target Row Refresh (TRR)

- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold
Target Row Refresh (TRR)

- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold
Target Row Refresh (TRR)

- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold
Target Row Refresh (TRR)

- counter per row
- increment neighbor rows
- refresh when counter reaches a threshold
MASCAT: Stopping Microarchitectural Attacks Before Execution (Irazoqui et al.)

- static analysis of the binary
- detect suspicious instruction sequences (\texttt{clflush}, \texttt{rdtsc}, fences, ...)
- open problem: false positives
- since then: remote exploits from network (Lipp et al., Tatar et al.)

---

ANVIL

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

ANVIL

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

---

• B-CATT: disable vulnerable physical memory
• G-CATT: isolate security domains in physical memory based on potential vulnerability

Ferdinand Brasser et al. “CAn’t Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks”. In: arXiv:1611.08396 (2016).
• B-CATT: disable vulnerable physical memory
• G-CATT: isolate security domains in physical memory based on potential vulnerability

Ferdinand Brasser et al. “Can’t Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks”. In: arXiv:1611.08396 (2016).
• B-CATT: disable vulnerable physical memory
• G-CATT: isolate security domains in physical memory based on potential vulnerability

B-CATT: might block 95% of RAM
G-CATT: what about non-kernel or shared pages?
G-CATT: bit flips more than 8 “rows” apart

Ferdinand Brasser et al. “CAn’t Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks”. In: arXiv:1611.08396 (2016).
Conclusion

- software fault attacks are real, and can be triggered by various techniques in various environments
- no physical access → different countermeasures than physical fault attacks
- difficult to replace hardware at a large scale → software countermeasures are our best short-term/mid-term hope
Thank you!

Contact

✉️ clementine.maurice@irisa.fr
🐦 @BloodyTangerine
Micro-Architectural Fault Attacks

Clémentine Maurice, CNRS
May 5, 2019—FICHSA, Ben-Gurion University of the Negev, Beer Sheva, Israel

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


References


