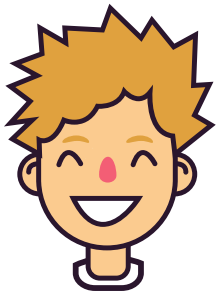


ANOTHER FLIP IN THE ROW

DANIEL GRUSS, MORITZ LIPP, MICHAEL SCHWARZ

AUGUST 9, 2018

GRAZ UNIVERSITY OF TECHNOLOGY

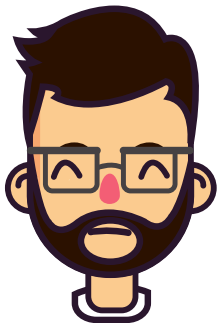


Daniel Gruss

PostDoc @ Graz University of Technology

🐦 @lavados

✉ daniel.gruss@iaik.tugraz.at

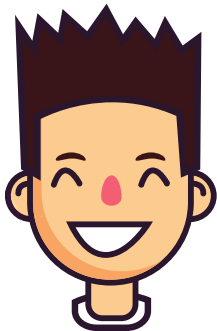


Moritz Lipp

PhD student @ Graz University of Technology

🐦 @mlxyz

✉ moritz.lipp@iaik.tugraz.at



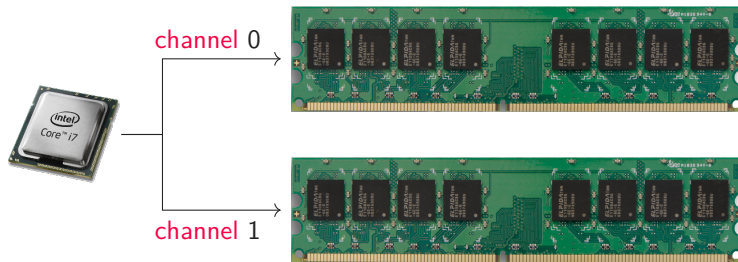
Michael Schwarz

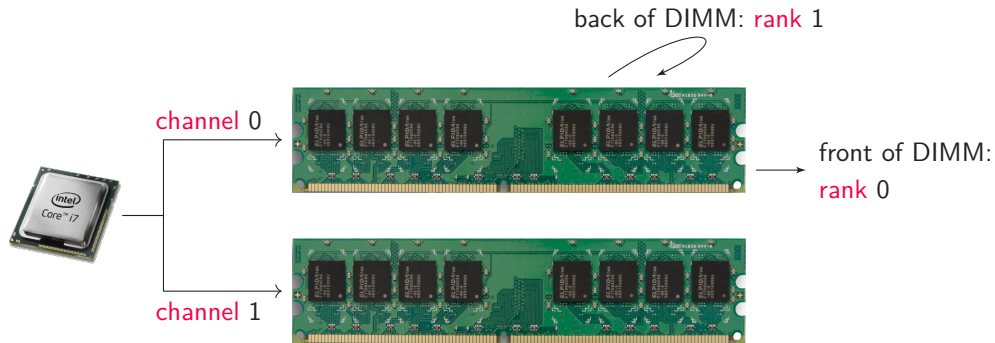
PhD student @ Graz University of Technology

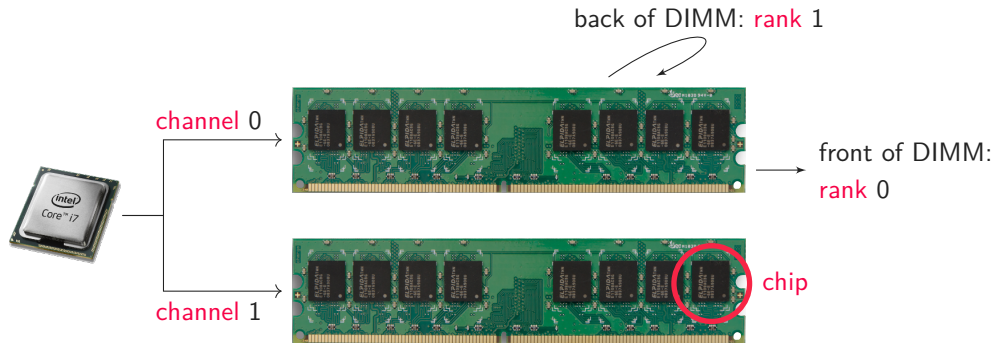
🐦 @misc0110

✉ michael.schwarz@iaik.tugraz.at

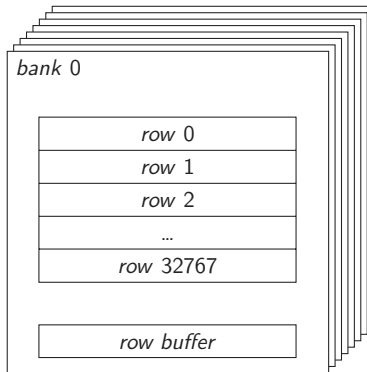








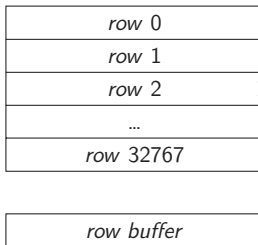
chip



chip

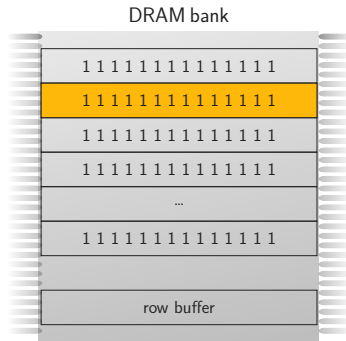


bank 0

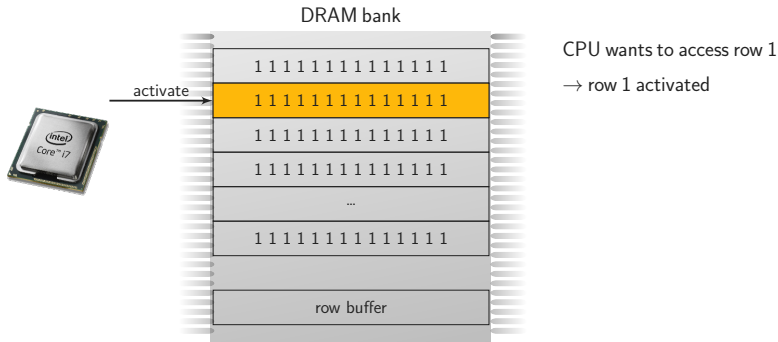


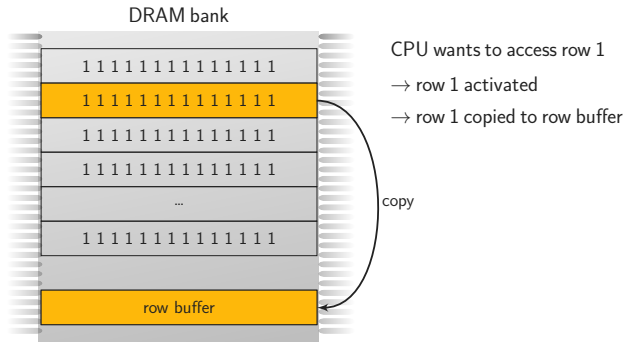
64k cells
1 capacitor,
1 transistor each

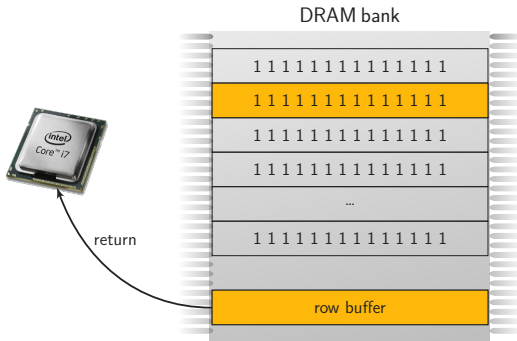
How reading from DRAM works



CPU wants to access row 1



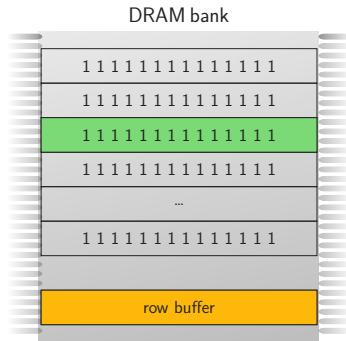




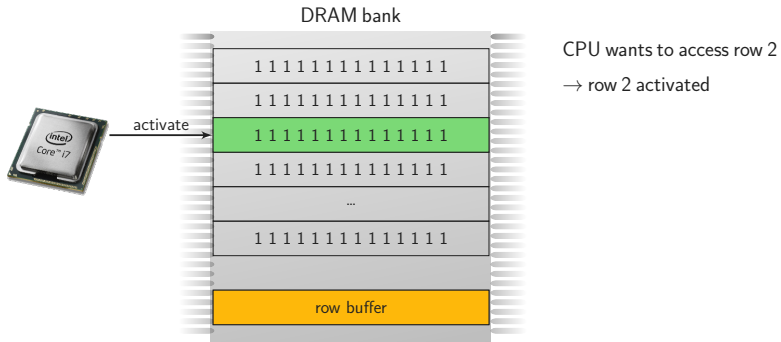
CPU wants to access row 1

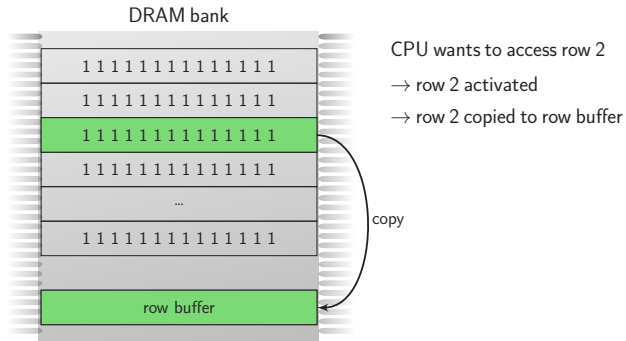
→ row 1 activated

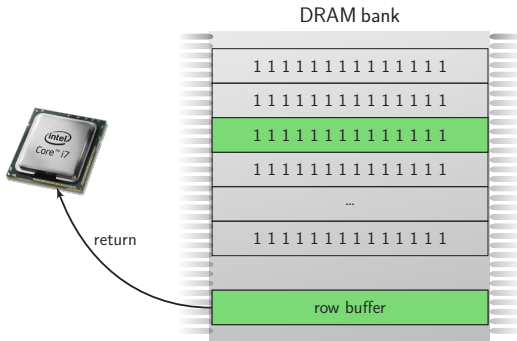
→ row 1 copied to row buffer



CPU wants to access row 2



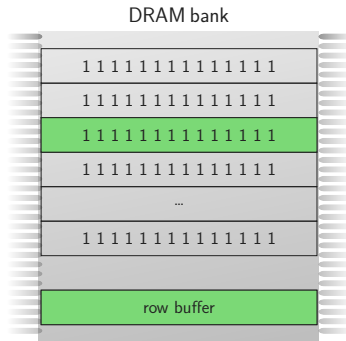




CPU wants to access row 2

→ row 2 activated

→ row 2 copied to row buffer

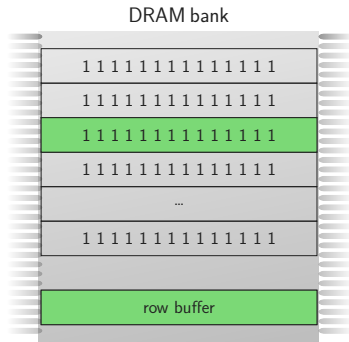


CPU wants to access row 2

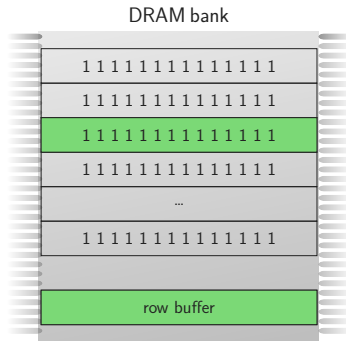
→ row 2 activated

→ row 2 copied to row buffer

→ **slow** (row conflict)

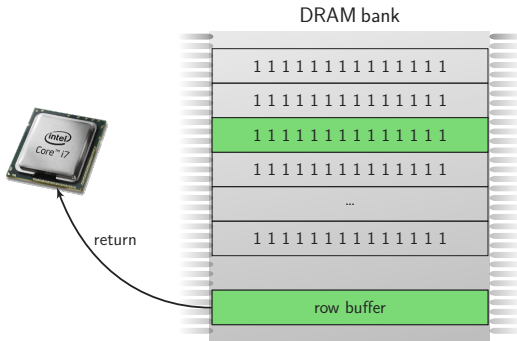


CPU wants to access row 2—again



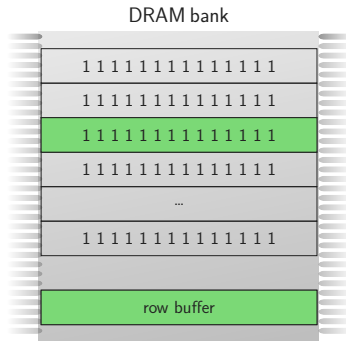
CPU wants to access row 2—again

→ row 2 already in row buffer



CPU wants to access row 2—again

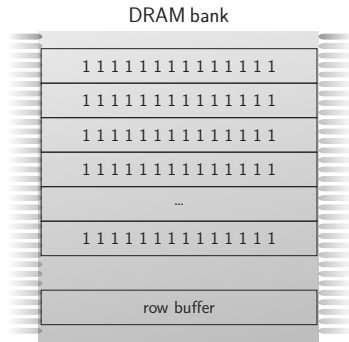
→ row 2 already in row buffer



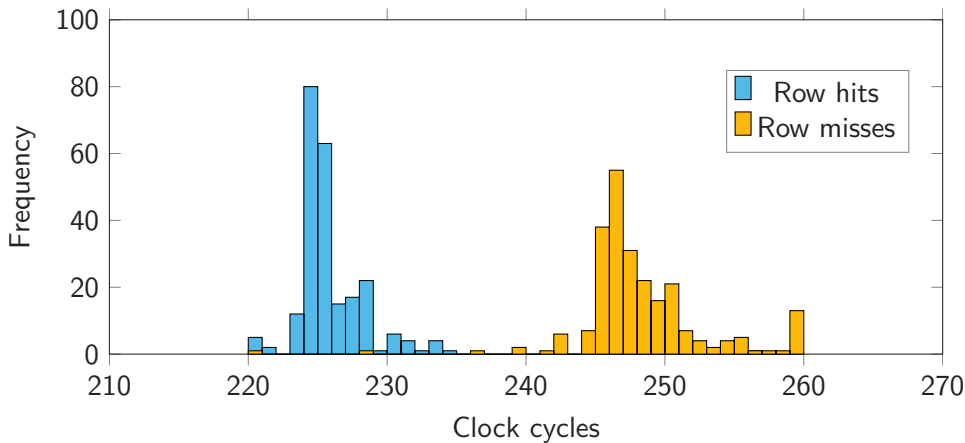
CPU wants to access row 2—again

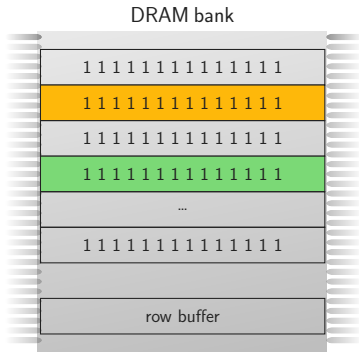
→ row 2 already in row buffer

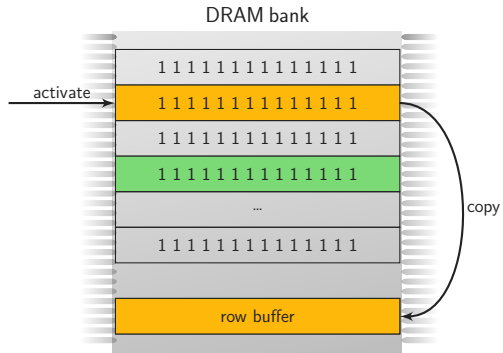
→ **fast** (row hit)

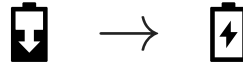
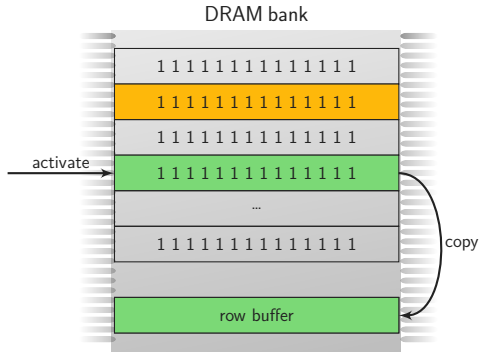


row buffer = cache

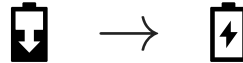
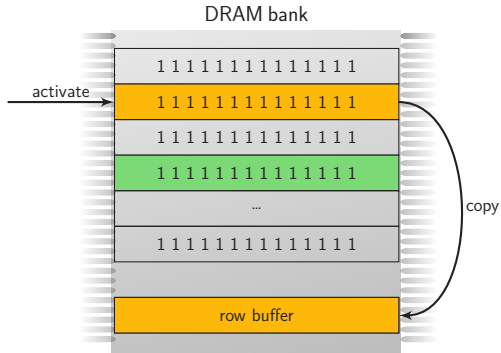




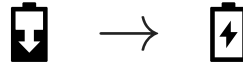
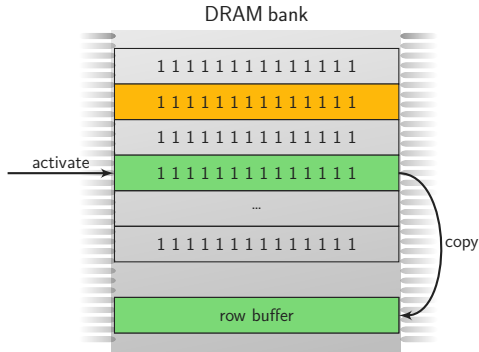




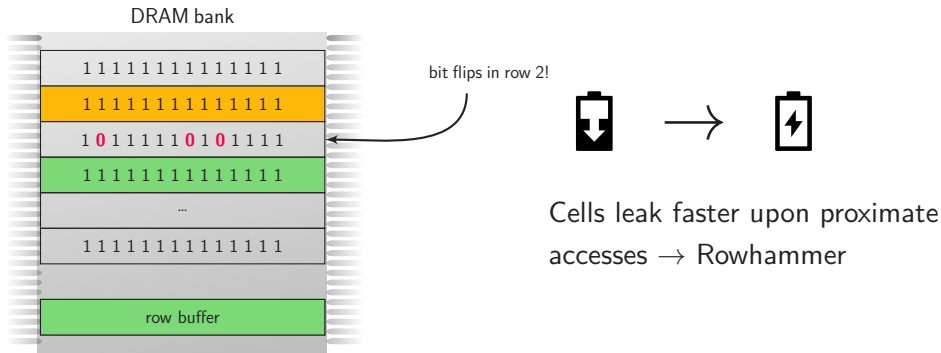
Cells leak faster upon proximate accesses → Rowhammer



Cells leak faster upon proximate accesses → Rowhammer



Cells leak faster upon proximate accesses → Rowhammer



How widespread is the issue?





- 85% affected [Kim+14] (see Figure)





- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]



DDR3

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

DDR4

- First believed to be safe

DDR3

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

DDR4

- First believed to be safe
- We showed bit flips [Pes+16]

DDR3

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

DDR4

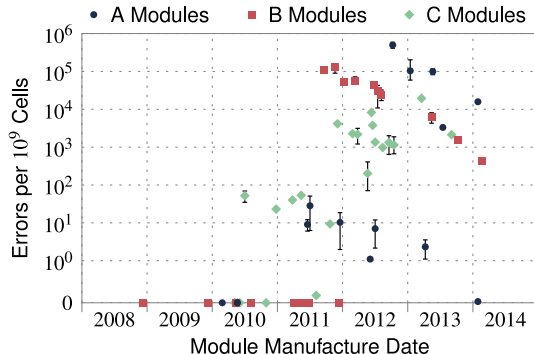
- First believed to be safe
- We showed bit flips [Pes+16]
- 67% affected [Lan16]

DDR3

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

DDR4

- First believed to be safe
- We showed bit flips [Pes+16]
- 67% affected [Lan16]

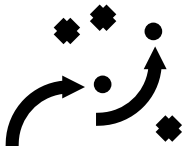




Memory accesses must be

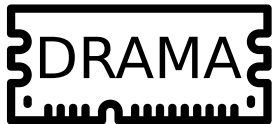
- **uncached**: reach DRAM
- **fast**: race against the next row refresh
- **targeted**: reach specific row

How do we get enough uncached accesses?



- `clflush` instruction → original paper [Kim+14]
- cache eviction [GMM16; Awe+16]
- non-temporal accesses [QS16]
- uncached memory [Vee+16]

How do we target accesses?



DRAMA: How your DRAM becomes a security problem

Anders Fogh & Michael Schwarz

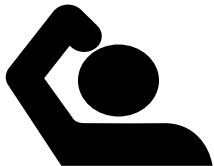
Black Hat Europe 2016

- They are not random → highly reproducible flip pattern!

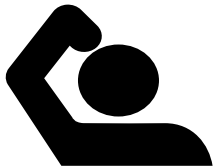




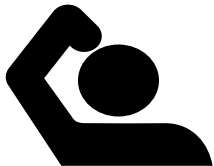
- They are not random → highly reproducible flip pattern!
 1. Choose a data structure that you can place at arbitrary memory locations



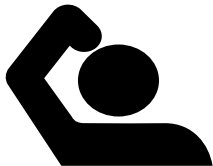
- They are not random → highly reproducible flip pattern!
 1. Choose a data structure that you can place at arbitrary memory locations
 2. Scan for “good” flips



- 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



- 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



- 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
- Alternatively: Build a PUF [Ana+18]







- Idea from [SD15]



- Idea from [SD15]
- x86 op codes are variable length



- Idea from [SD15]
- x86 op codes are variable length
 - Unsafe op codes (syscall) \in safe but long multi-byte op codes



- Idea from [SD15]
- 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



- Idea from [SD15]
- 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



- Idea from [SD15]
- 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

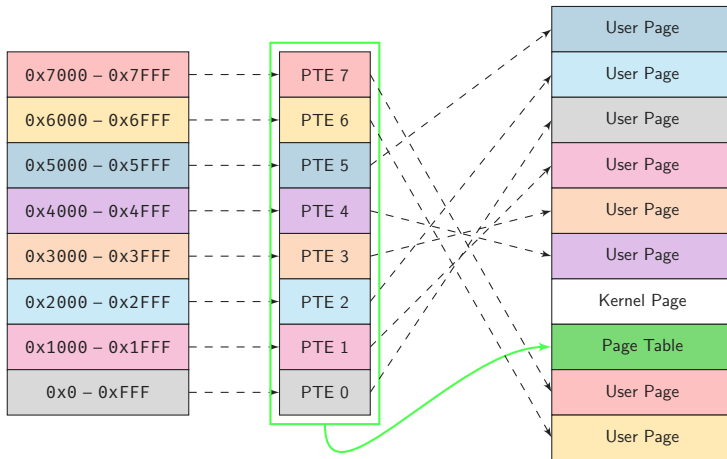
P	RW	US	WT	UC	R	D	S	G		
										X

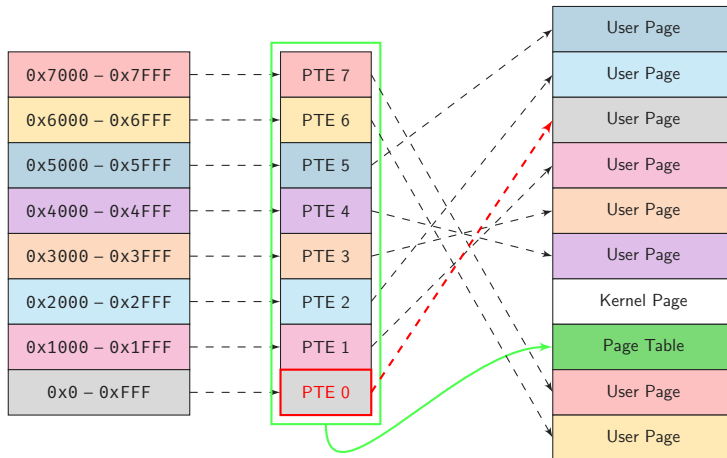
P	RW	US	WT	UC	R	D	S	G	Ignored	
				Ignored						X

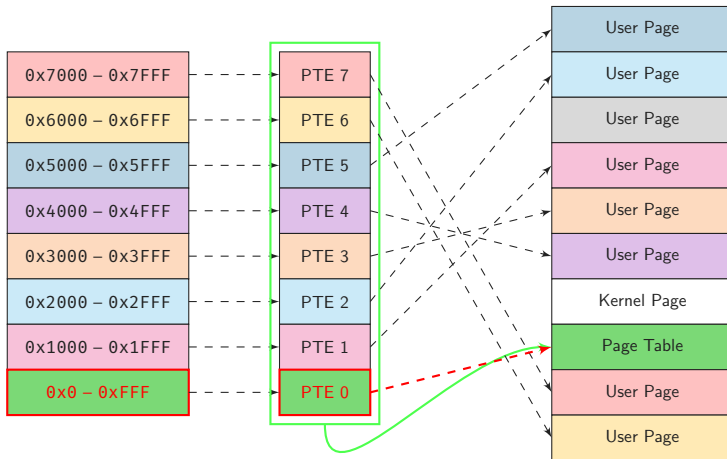
P	RW	US	WT	UC	R	D	S	G	Ignored	
Physical Page Number										
				Ignored						X

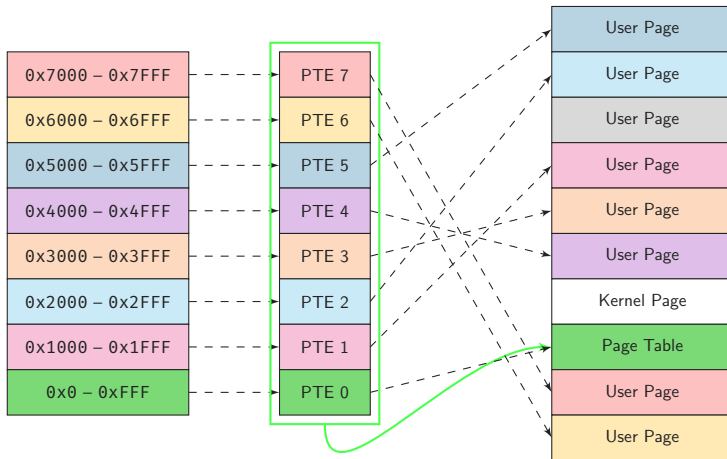
P	RW	US	WT	UC	R	D	S	G	Ignored	
Physical Page Number										
				Ignored						X

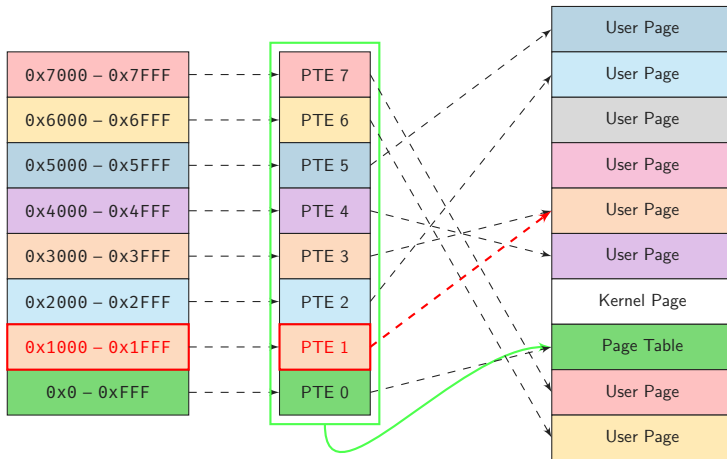
Each 4 KB page table consists of 512 such entries

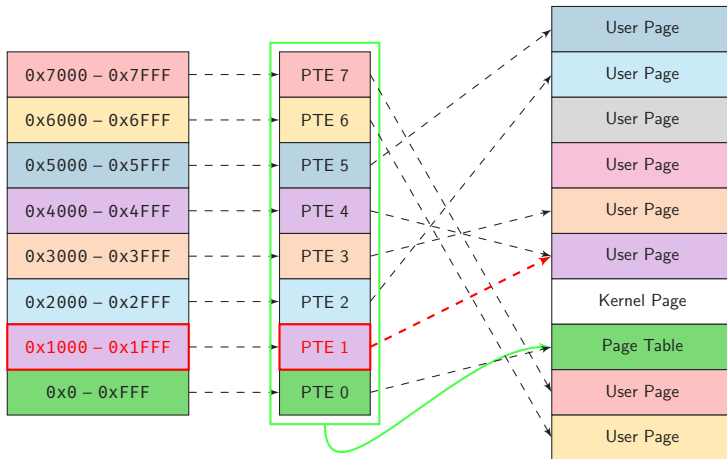


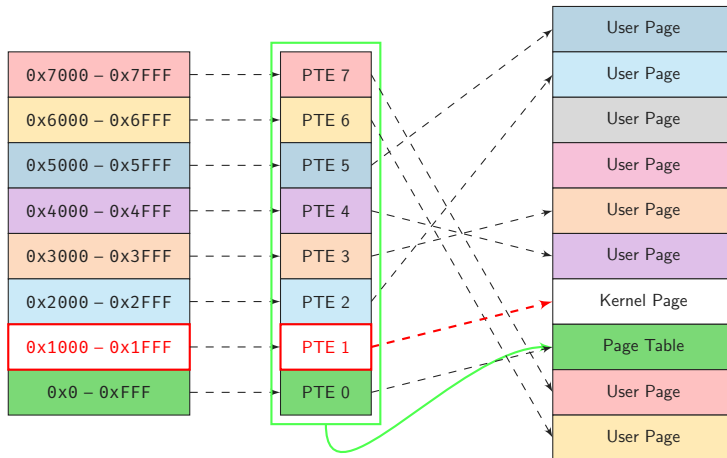


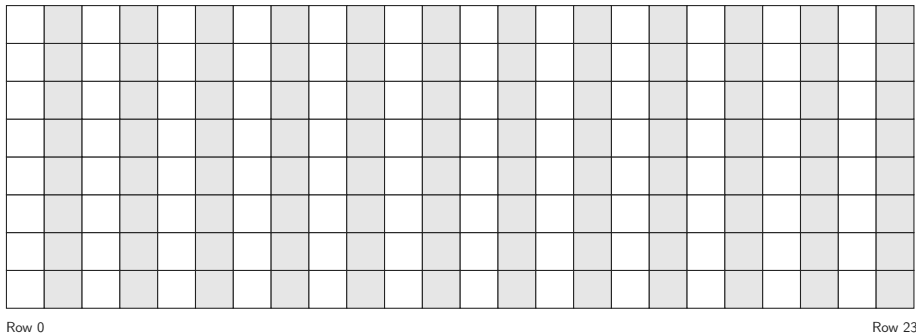




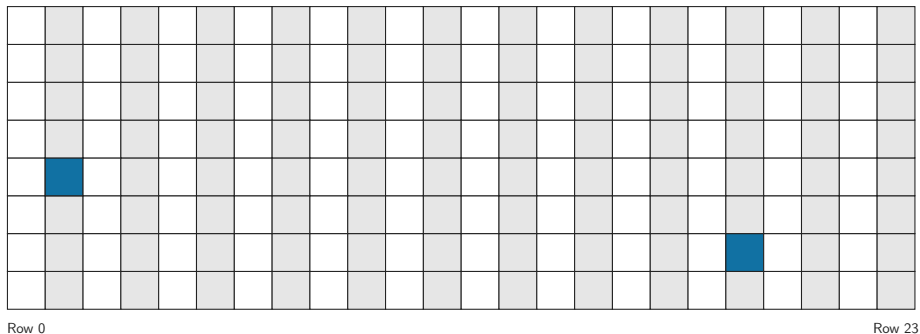




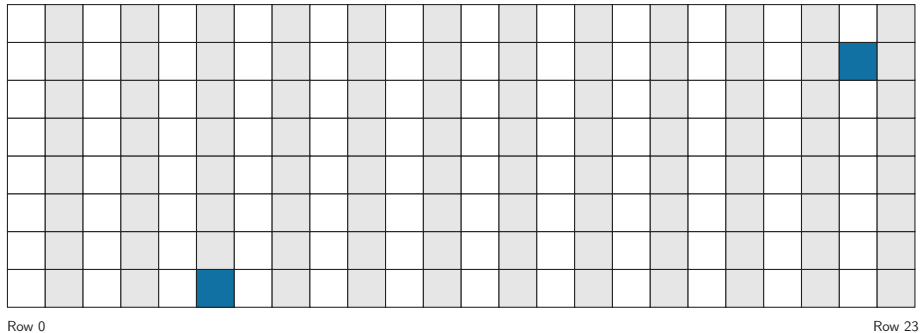




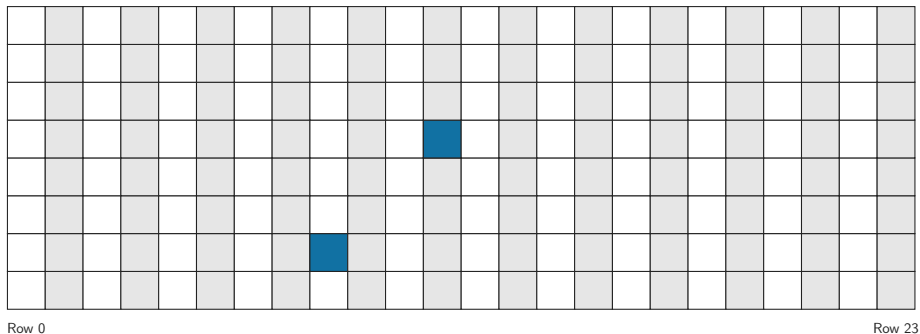
Hammering memory locations in different rows



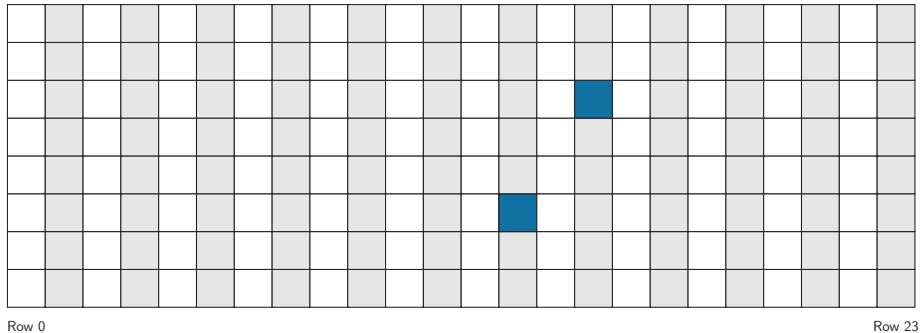
Hammering memory locations in different rows



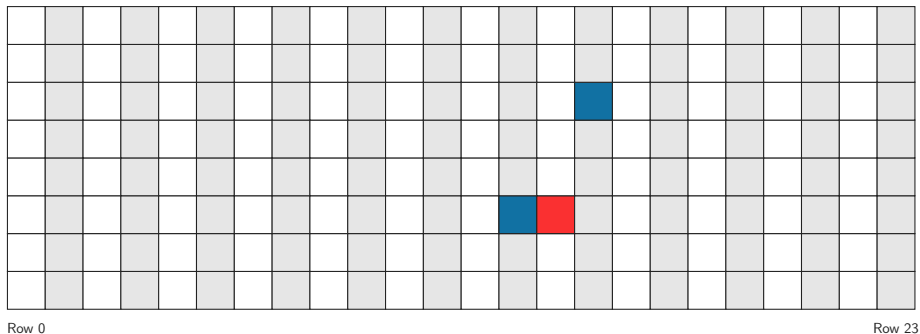
Hammering memory locations in different rows



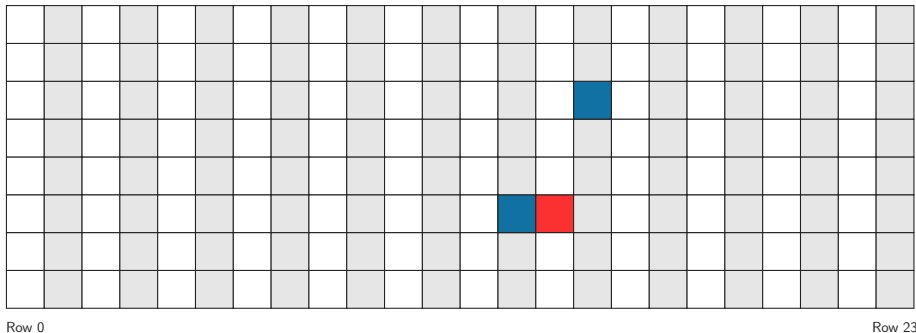
Hammering memory locations in different rows

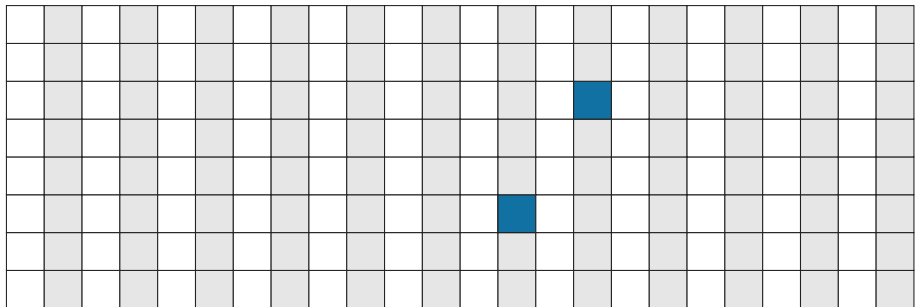


Hammering memory locations in different rows



Hammering memory locations in different rows

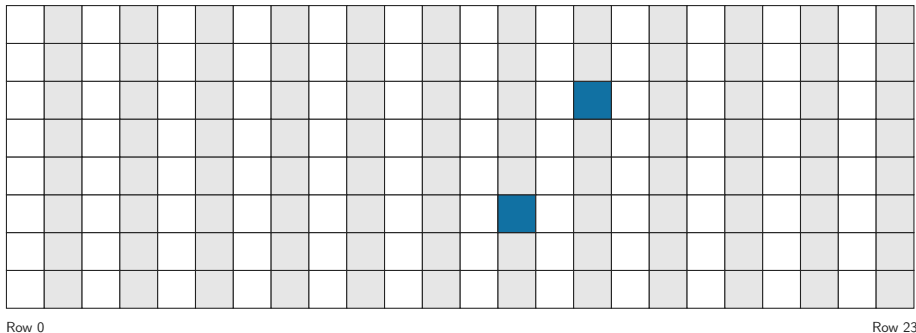




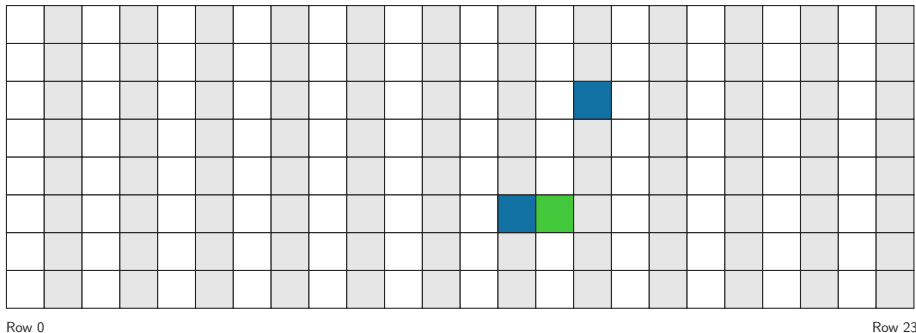
Row 0

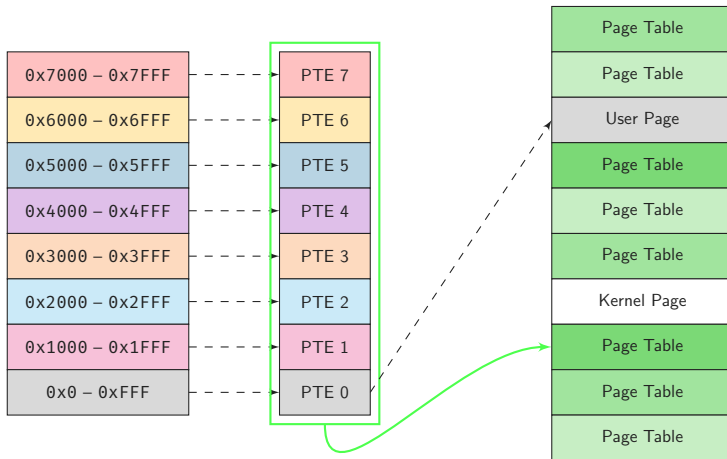
Row 23

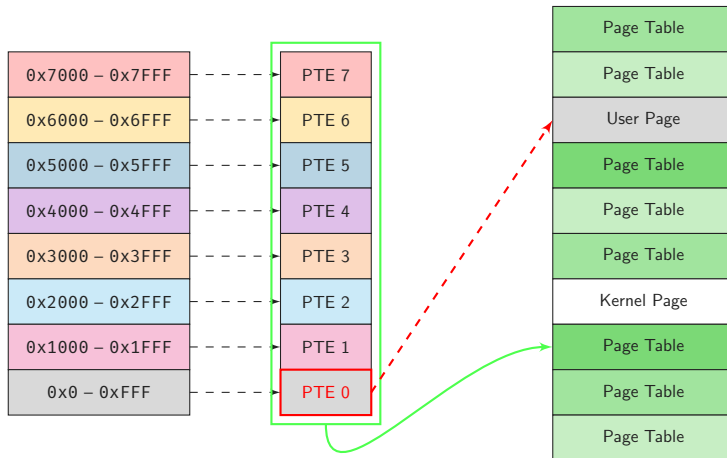
Fill all remaining memory with page tables

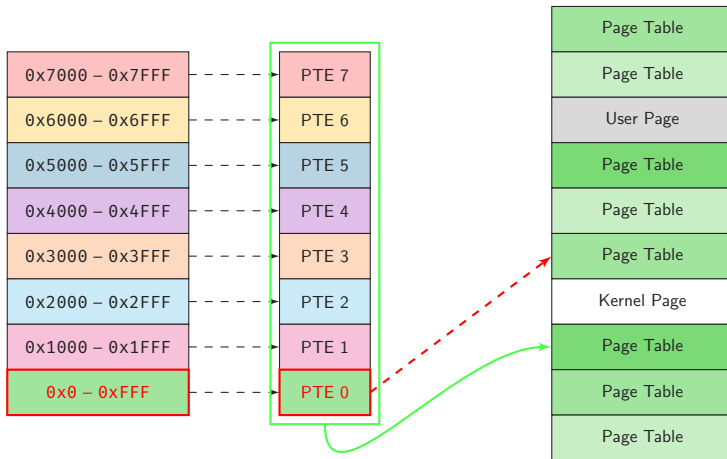


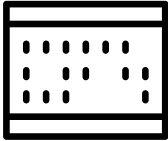
Fill all remaining memory with page tables

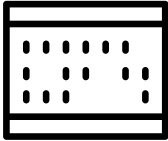


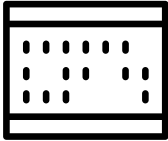




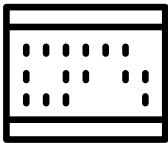




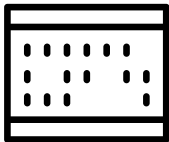




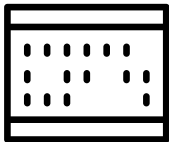
1. Scan for flips

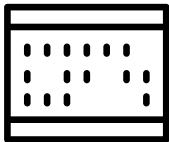


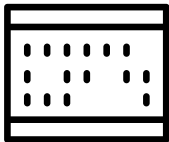
1. Scan for flips
2. Exhaust or massage memory to place a page table at target location



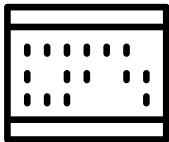
1. Scan for flips
2. Exhaust or massage memory to place a page table at target location
3. Gain access to your own page table → kernel privileges



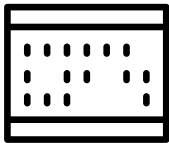




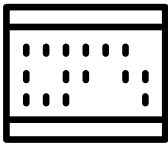
- Idea from [SD15]



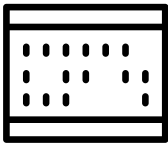
- Idea from [SD15]
- Same idea applied in several other works:



- Idea from [SD15]
- Same idea applied in several other works:
 - Rowhammer.js [GMM16]



- Idea from [SD15]
- Same idea applied in several other works:
 - Rowhammer.js [GMM16]
 - One bit flips, one cloud flops [Xia+16]



- Idea from [SD15]
- Same idea applied in several other works:
 - Rowhammer.js [GMM16]
 - One bit flips, one cloud flops [Xia+16]
 - Drammer [Vee+16]





- Scan entire physical memory (very fast) and:





- Scan entire physical memory (very fast) and:
 - Modify binary pages executed in root privileges [Xia+16]



- Scan entire physical memory (very fast) and:
 - Modify binary pages executed in root privileges [Xia+16]
 - Modify credential structs [Vee+16]



- Scan entire physical memory (very fast) and:
 - Modify binary pages executed in root privileges [Xia+16]
 - Modify credential structs [Vee+16]
 - Read keys [Xia+16]



- Scan entire physical memory (very fast) and:
 - Modify binary pages executed in root privileges [Xia+16]
 - Modify credential structs [Vee+16]
 - Read keys [Xia+16]
 - Corrupt signatures [BM16; Pod+18]



- Scan entire physical memory (very fast) and:
 - Modify binary pages executed in root privileges [Xia+16]
 - Modify credential structs [Vee+16]
 - Read keys [Xia+16]
 - Corrupt signatures [BM16; Pod+18]
 - Modify certificates



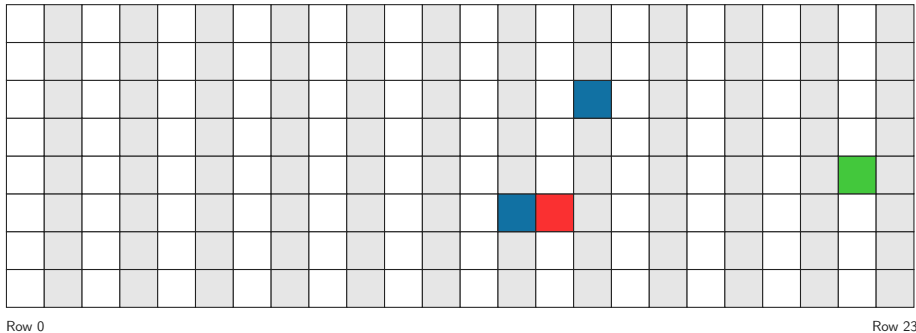
- Scan entire physical memory (very fast) and:
 - Modify binary pages executed in root privileges [Xia+16]
 - Modify credential structs [Vee+16]
 - Read keys [Xia+16]
 - Corrupt signatures [BM16; Pod+18]
 - Modify certificates
 - Configurations

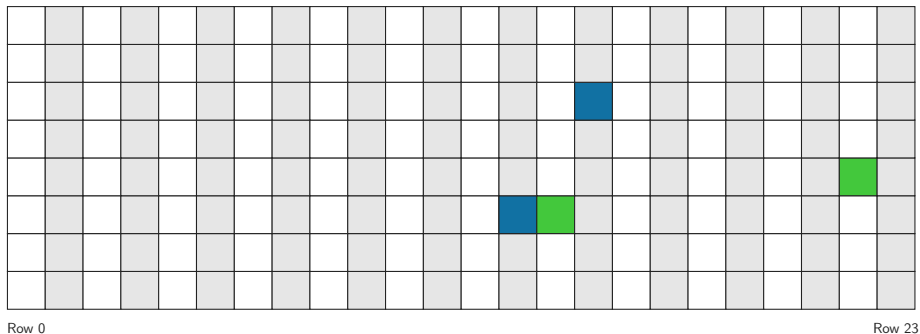


- Scan entire physical memory (very fast) and:
 - Modify binary pages executed in root privileges [Xia+16]
 - Modify credential structs [Vee+16]
 - Read keys [Xia+16]
 - Corrupt signatures [BM16; Pod+18]
 - Modify certificates
 - Configurations
 - etc.

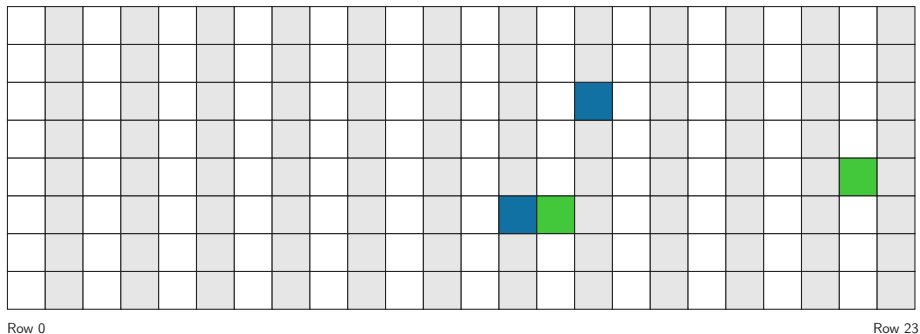


- Scan entire physical memory (very fast) and:
 - Modify binary pages executed in root privileges [Xia+16]
 - Modify credential structs [Vee+16]
 - Read keys [Xia+16]
 - Corrupt signatures [BM16; Pod+18]
 - Modify certificates
 - Configurations
 - etc.
- pages are pretty unique: 32768 bits per page

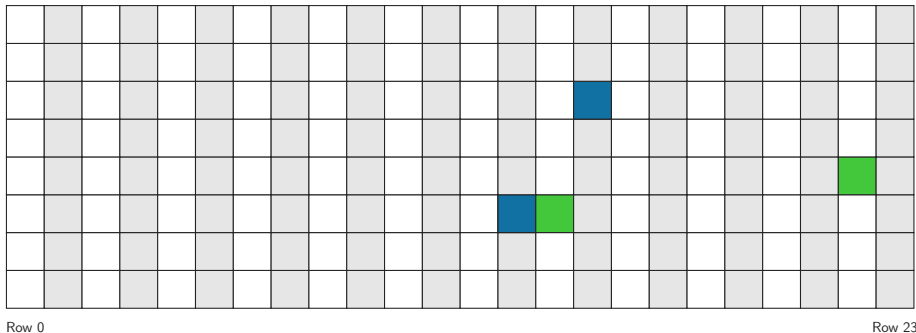




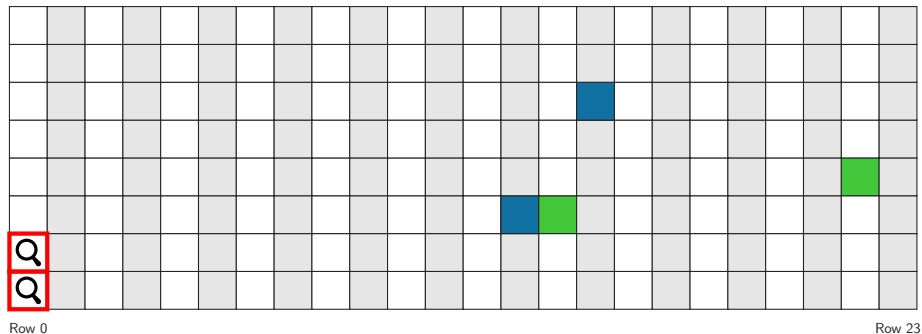
Page with bit flip is filled with target content



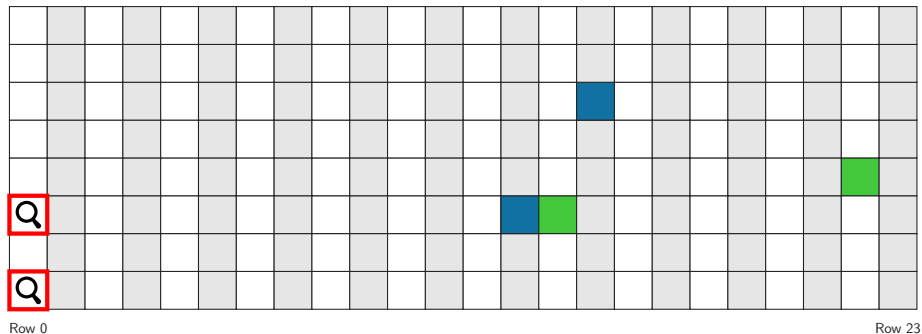
OS or hypervisor searches for duplicate pages



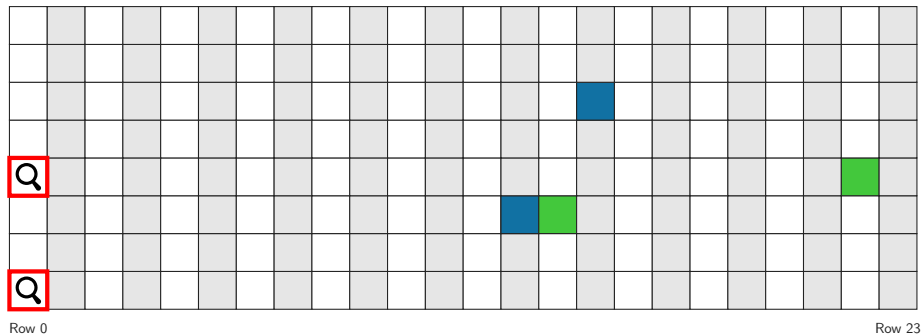
OS or hypervisor searches for duplicate pages



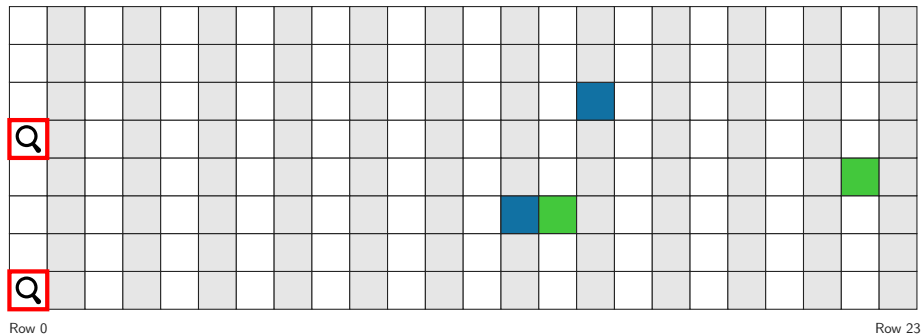
OS or hypervisor searches for duplicate pages



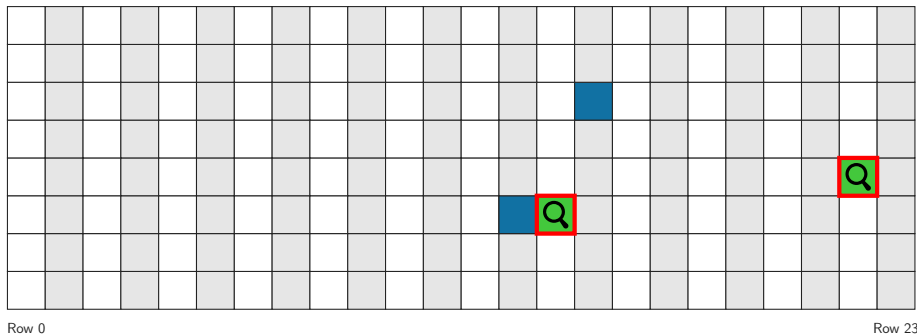
OS or hypervisor searches for duplicate pages



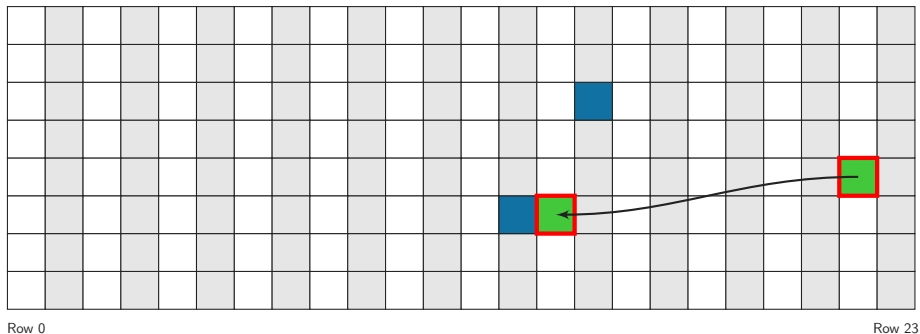
OS or hypervisor searches for duplicate pages



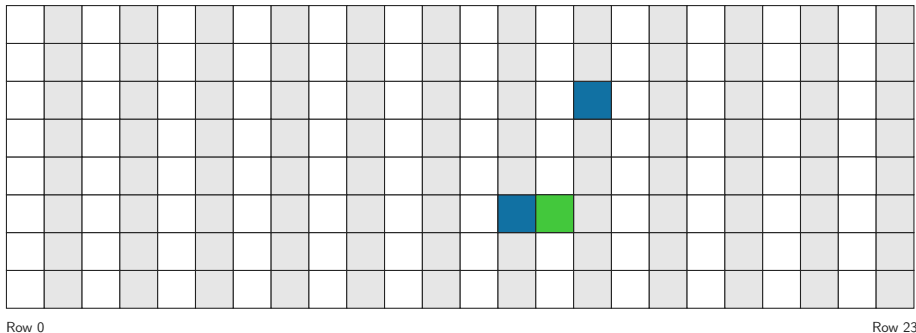
OS or hypervisor searches for duplicate pages



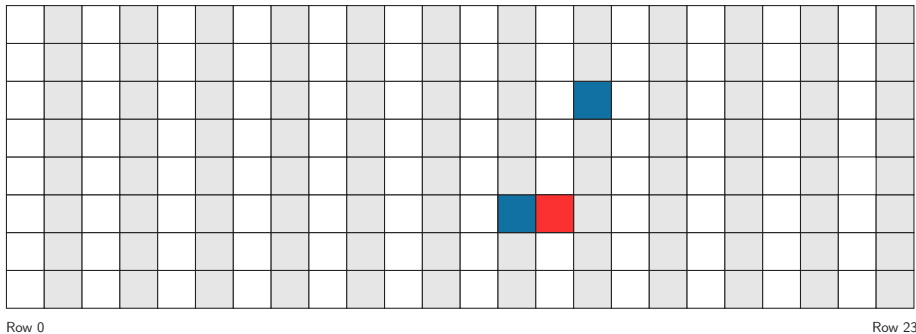
OS or hypervisor searches for duplicate pages

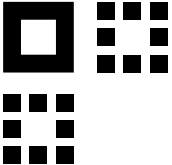


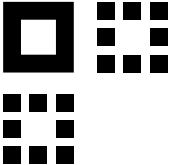
OS or hypervisor searches for duplicate pages

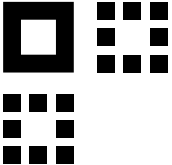


Hammer again + flip again

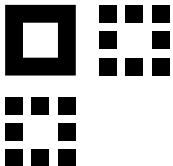




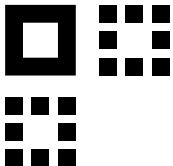




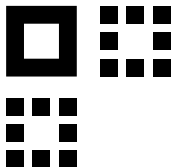
1. Scan for flips

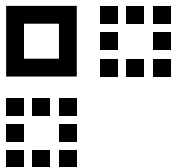


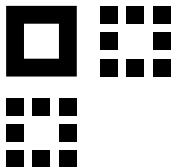
1. Scan for flips
2. Place content for deduplication so that flip can be exploited



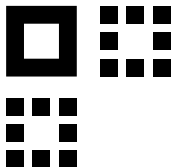
1. Scan for flips
2. Place content for deduplication so that flip can be exploited
3. Perform the bit change through Rowhammer



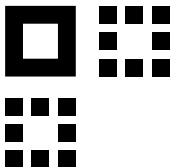




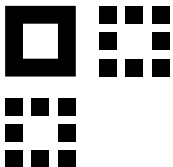
- Idea from [Bos+16]



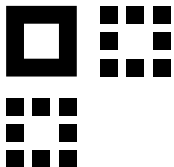
- Idea from [Bos+16]
 - Change data type (double \rightarrow pointer)



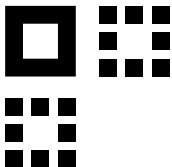
- Idea from [Bos+16]
 - Change data type (double \rightarrow pointer)
 - Change pointer to good object \rightarrow counterfeit object



- Idea from [Bos+16]
 - Change data type (double \rightarrow pointer)
 - Change pointer to good object \rightarrow counterfeit object
- and from [Raz+16]



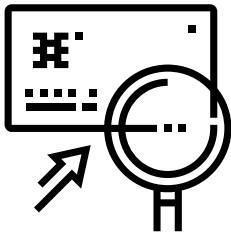
- Idea from [Bos+16]
 - Change data type (double \rightarrow pointer)
 - Change pointer to good object \rightarrow counterfeit object
- and from [Raz+16]
 - Corrupt authorized SSH keys



- Idea from [Bos+16]
 - Change data type (double → pointer)
 - Change pointer to good object → counterfeit object
- and from [Raz+16]
 - Corrupt authorized SSH keys
 - Corrupt Debian update URLs + RSA public key file

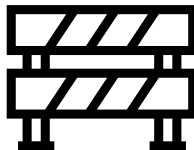
How to mitigate Rowhammer?

Different mitigations have been proposed:



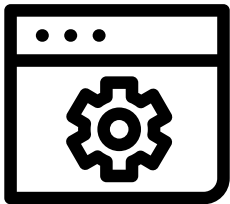
Detection

vs



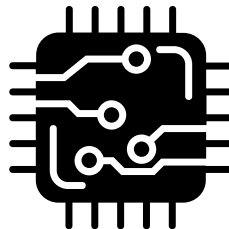
Prevention

Different mitigations have been proposed:



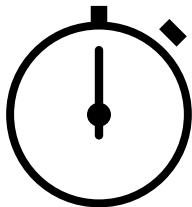
Software

vs



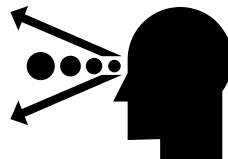
Hardware

Different mitigations have been proposed:



Short Term

vs



Long Term

- No `clflush` instruction

✕ ✕



- No `clflush` instruction →
Rowhammer.js

✕ ✕



✘ ✘

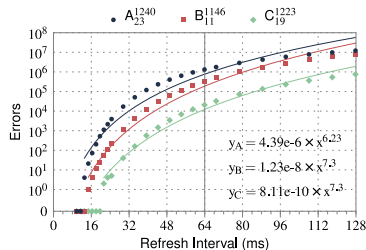
————

- No `clflush` instruction →
Rowhammer.js
- Increase the refresh rate

✗ ✗

—

- No `clflush` instruction → Rowhammer.js
- Increase the refresh rate
 - Would need to be **increased** by **7×** to eliminate all bit flips

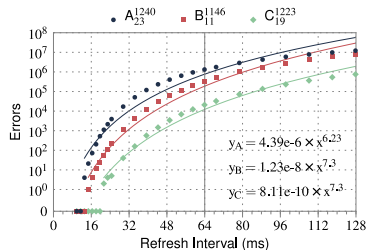


Errors depending on refresh interval [Kim+14]

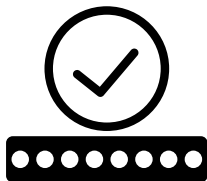
✗ ✗

—

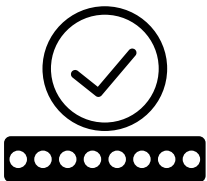
- No `clflush` instruction → Rowhammer.js
- Increase the refresh rate
 - Would need to be **increased by 7×** to eliminate all bit flips
 - Implementation: increased by 2× by BIOS vendors



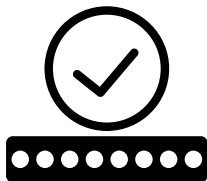
Errors depending on refresh interval [Kim+14]



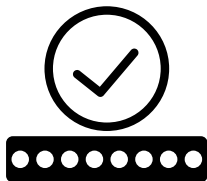
- ECC protection: server can handle or correct single bit errors



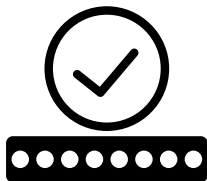
- ECC protection: server can handle or correct single bit errors
- **No standard** for event reporting



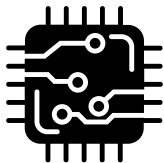
- ECC protection: server can handle or correct single bit errors
- **No standard** for event reporting
- In practice [Lan16]
 - Common: server counts ECC errors and report only if they reach a threshold (e.g., > 100 bit flips / hour)



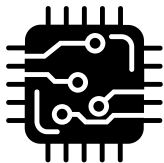
- ECC protection: server can handle or correct single bit errors
- **No standard** for event reporting
- In practice [Lan16]
 - Common: server counts ECC errors and report only if they reach a threshold (e.g., > 100 bit flips / hour)
 - Some server vendors **never report errors** to the OS



- ECC protection: server can handle or correct single bit errors
- **No standard** for event reporting
- In practice [Lan16]
 - 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

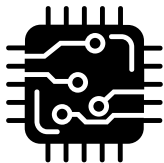


Original ideas from [Kim+14]



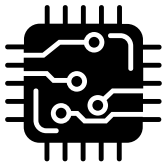
Original ideas from [Kim+14]

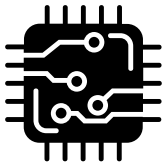
- Making better DRAM chips that are not vulnerable



Original ideas from [Kim+14]

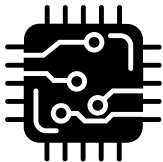
- Making better DRAM chips that are not vulnerable
- Using error correcting codes (ECC)





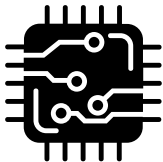
Original ideas from [Kim+14]

- Making better DRAM chips that are not vulnerable
- Using error correcting codes (ECC)
- Increasing the refresh rate



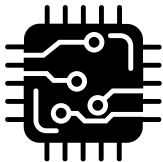
Original ideas from [Kim+14]

- Making better DRAM chips that are not vulnerable
- Using error correcting codes (ECC)
- Increasing the refresh rate
- Remapping/retiring faulty cells after manufacturing



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



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

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 \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



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

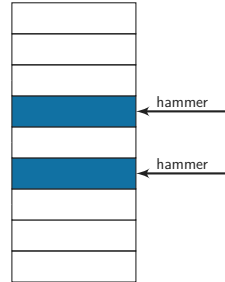


PARA - Probabilistic Adjacent Row Activation [Kim+14]

- 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
- For $p = 0.001$ and $N_{th} = 100K$, experiencing one error in one year has a probability 9.4×10^{-14}

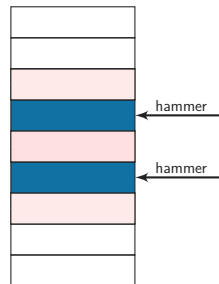


Target Row Refresh (TRR)



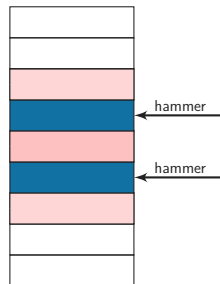
Target Row Refresh (TRR)

- Counter per row



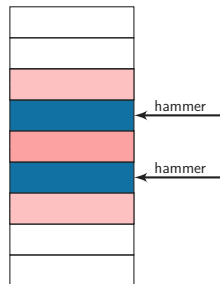
Target Row Refresh (TRR)

- Counter per row
- Increment neighbor rows



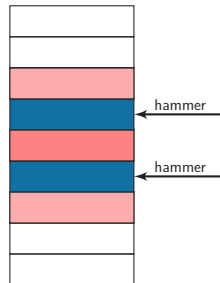
Target Row Refresh (TRR)

- Counter per row
- Increment neighbor rows



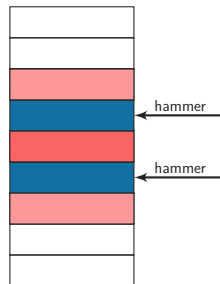
Target Row Refresh (TRR)

- Counter per row
- Increment neighbor rows



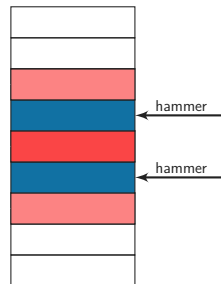
Target Row Refresh (TRR)

- Counter per row
- Increment neighbor rows



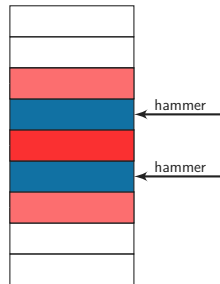
Target Row Refresh (TRR)

- Counter per row
- Increment neighbor rows



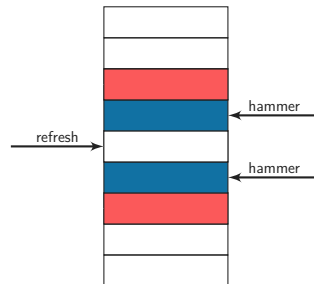
Target Row Refresh (TRR)

- Counter per row
- Increment neighbor rows



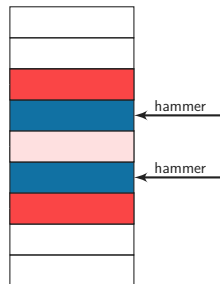
Target Row Refresh (TRR)

- Counter per row
- Increment neighbor rows
- Refresh when counter reaches a threshold



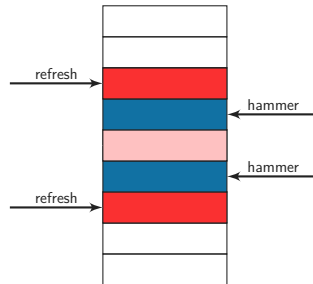
Target Row Refresh (TRR)

- Counter per row
- Increment neighbor rows
- Refresh when counter reaches a threshold



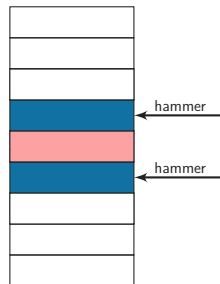
Target Row Refresh (TRR)

- Counter per row
- Increment neighbor rows
- Refresh when counter reaches a threshold



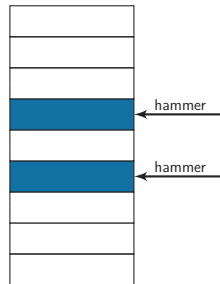
Target Row Refresh (TRR)

- 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



We flipped bits on DDR4 with TRR activated!

“nohammer” kernel module [Cor16]

“nohammer” kernel module [Cor16]

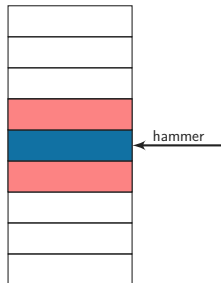
- Refresh rate of 8 ms would prevent Rowhammer on most systems

“nohammer” kernel module [Cor16]

- Refresh rate of 8 ms would prevent Rowhammer on most systems
- Use PMC to measure cache misses per 64 ms interval

“nohammer” kernel module [Cor16]

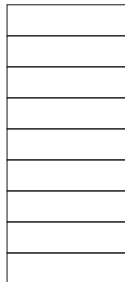
- Refresh rate of 8 ms would prevent Rowhammer on most systems
- Use PMC to measure cache misses per 64 ms interval
- Limit cache miss rate to $1/8$ of maximum



Wait for refresh

“nohammer” kernel module [Cor16]

- 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

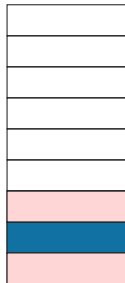


Wait for refresh

Wait for refresh

“nohammer” kernel module [Cor16]

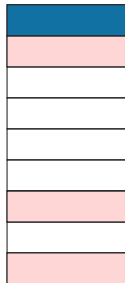
- Refresh rate of 8 ms would prevent Rowhammer on most systems
- Use PMC to measure cache misses per 64 ms interval
- Limit cache miss rate to $1/8$ of maximum



Wait for refresh

“nohammer” kernel module [Cor16]

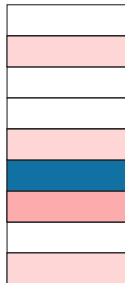
- Refresh rate of 8 ms would prevent Rowhammer on most systems
- Use PMC to measure cache misses per 64 ms interval
- Limit cache miss rate to $1/8$ of maximum



Wait for refresh

“nohammer” kernel module [Cor16]

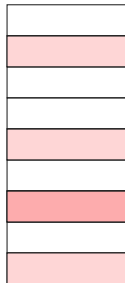
- Refresh rate of 8 ms would prevent Rowhammer on most systems
- Use PMC to measure cache misses per 64 ms interval
- Limit cache miss rate to $1/8$ of maximum



Wait for refresh

“nohammer” kernel module [Cor16]

- 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

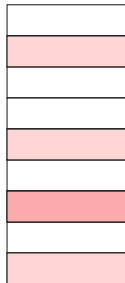


Wait for refresh

Wait for refresh

“nohammer” kernel module [Cor16]

- 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

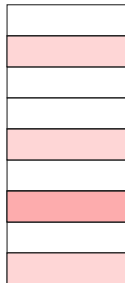


Wait for refresh

Performance?

“nohammer” kernel module [Cor16]

- 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

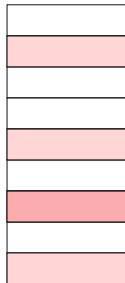


Wait for refresh

Performance? Grand Pwning Unit [Fri+18],

“nohammer” kernel module [Cor16]

- 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

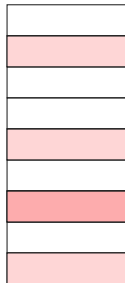


Wait for refresh

Performance? Grand Pwning Unit [Fri+18], ThrowHammer [Tat+18],

“nohammer” kernel module [Cor16]

- 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

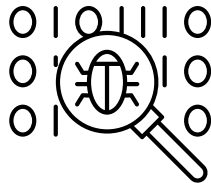
Performance? Grand Pwning Unit [Fri+18], ThrowHammer [Tat+18], NetHammer [Lip+17].

MASCAT - Stopping Microarchitectural Attacks Before Execution [IES17]

- Static analysis of the binary

MASCAT - Stopping Microarchitectural Attacks Before Execution [IES17]

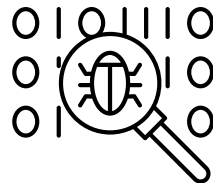
- Static analysis of the binary
- Detect suspicious instruction sequences
(`clflush`, `rdtsc`, fences, ...)
- Open problem: false positives



MASCAT - Stopping Microarchitectural Attacks Before Execution [IES17]

- Static analysis of the binary
- Detect suspicious instruction sequences (`clflush`, `rdtsc`, fences, ...)
- Open problem: false positives

ThrowHammer [Tat+18], NetHammer [Lip+17].

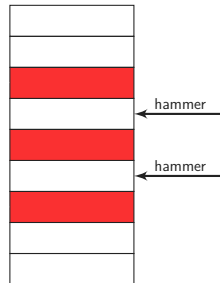


ANVIL [Awe+16]

- Uses performance counters to detect rowhammer

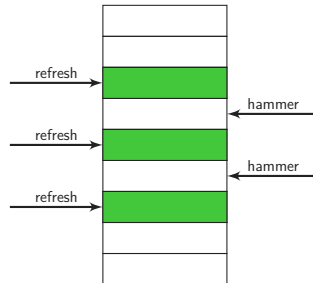
ANVIL [Awe+16]

- Uses performance counters to detect rowhammer
- Activate rows neighbor rows to prevent flips
- Similar as PARA, but in software



ANVIL [Awe+16]

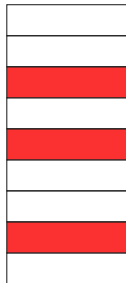
- Uses performance counters to detect rowhammer
- Activate rows neighbor rows to prevent flips
- Similar as PARA, but in software



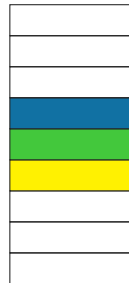
What if performance counters do not work? [Gru+18; Jan+17]

- B-CATT: disable vulnerable physical memory [Bra+17]
- G-CATT: isolate security domains in physical memory based on potential vulnerability [Bra+17]

B-CATT

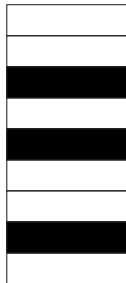


G-CATT



- B-CATT: disable vulnerable physical memory [Bra+17]
- G-CATT: isolate security domains in physical memory based on potential vulnerability [Bra+17]

B-CATT



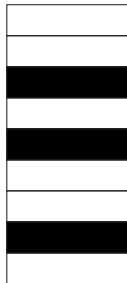
G-CATT



B-CATT: Might block 95% of RAM [Gru+18; Vee+18]

- B-CATT: disable vulnerable physical memory [Bra+17]
- G-CATT: isolate security domains in physical memory based on potential vulnerability [Bra+17]

B-CATT



G-CATT

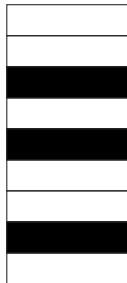


B-CATT: Might block 95% of RAM [Gru+18; Vee+18]

G-CATT: What about non-kernel or shared pages? [Gru+18; CZN18]

- B-CATT: disable vulnerable physical memory [Bra+17]
- G-CATT: isolate security domains in physical memory based on potential vulnerability [Bra+17]

B-CATT



G-CATT

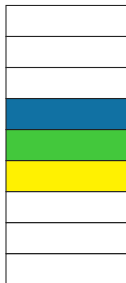


B-CATT: Might block 95% of RAM [Gru+18; Vee+18]

G-CATT: What about non-kernel or shared pages? [Gru+18; CZN18]

G-CATT: Bit flips more than 8 “rows” apart [Kim+14; Gru+18]

- Isolate DMA buffers in physical memory [Vee+18]



- Isolate DMA buffers in physical memory [Vee+18]

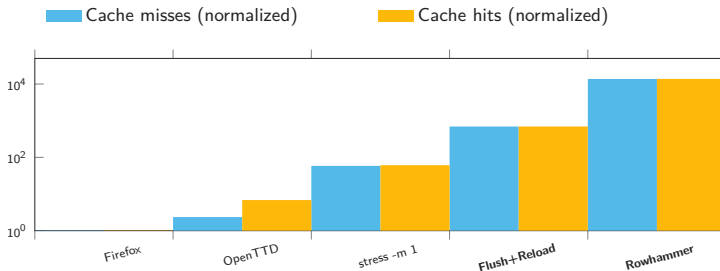


- Isolate DMA buffers in physical memory [Vee+18]

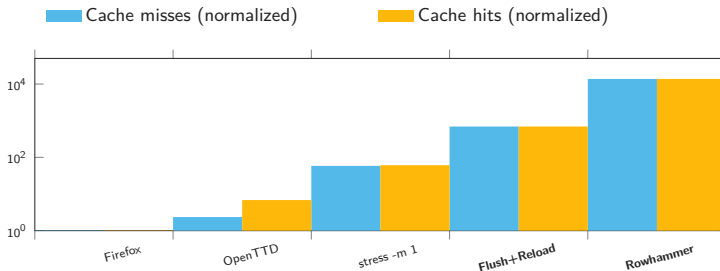


Bit flips more than 8 “rows” apart [Kim+14; Gru+18]

- Rowhammer: lots of **cache misses** that can be monitored with **hardware performance counters** [HF15; Gru+16; CSY15; Pay16]



- Rowhammer: lots of **cache misses** that can be monitored with **hardware performance counters** [HF15; Gru+16; CSY15; Pay16]



What if performance counters do not work because we run in SGX? [Gru+18; Jan+17]

Methodology		Defense																
		MASCAT	Chiapetta et al. [CSY15]	CloudRadar	Herath and Fogh [HF15]	HexPADS	perF	ANVIL	nohammer	No OOM	G-CATT	B-CATT	TRR	MAC	PARA/CRA/PRA	ARMOR	ECC/Chipkill	Refresh Rate
DETECTION																		
	Static Analysis	●	○	○	◐	○	○	○	○	○	○	○	○	○	○	○	○	○
	Performance Counters	○	●	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○
	Memory Access Pattern	○	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○
NEUTRALIZATION																		
	Physical Proximity	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○
	Memory Footprint	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○
ELIMINATION																		
	Bootloader	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○
	Hardware Modification	○	○	○	○	○	○	○	○	○	○	○	●	●	●	●	○	○
	BIOS Update	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●

What if you don't need to hammer two or more rows?

What if you don't need to hammer two or more rows?

One-location hammering



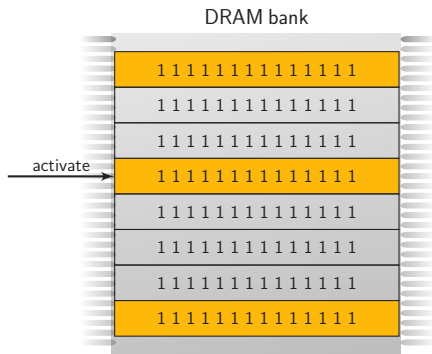
- There are two different hammering techniques

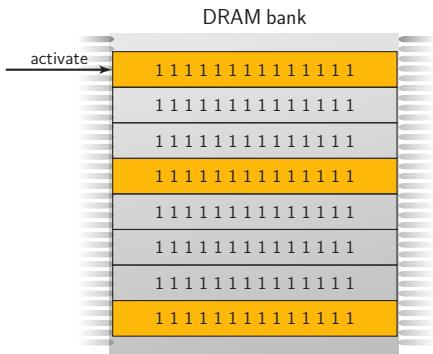


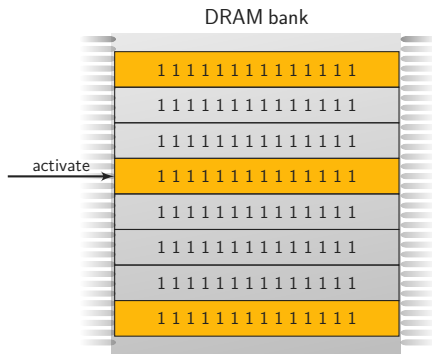
- There are two different hammering techniques
- #1: Hammer one row next to victim row and other random rows

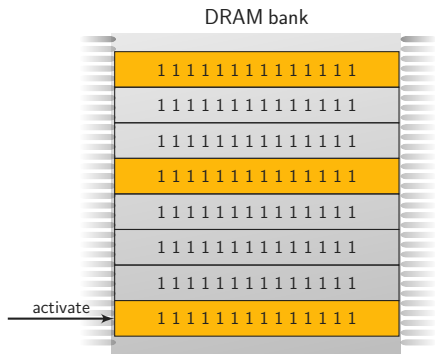


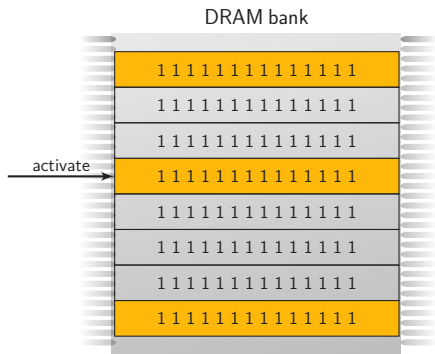
- There are two different hammering techniques
- #1: Hammer one row next to victim row and other random rows
- #2: Hammer two rows neighboring victim row

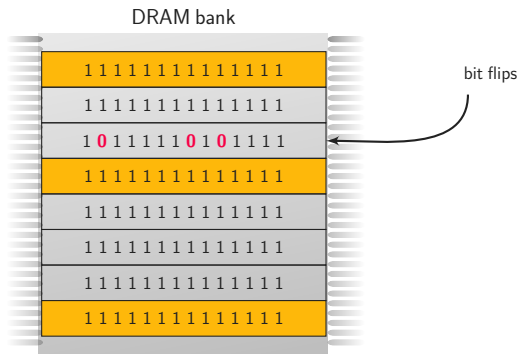


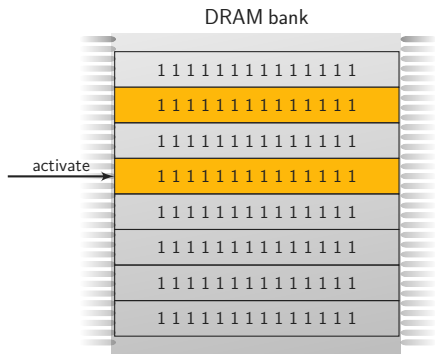


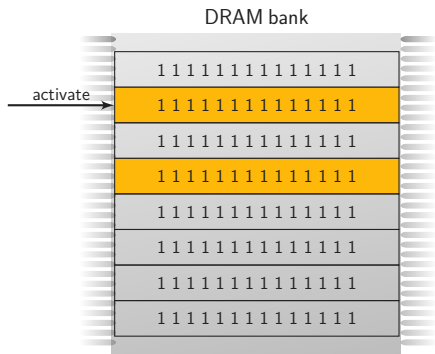


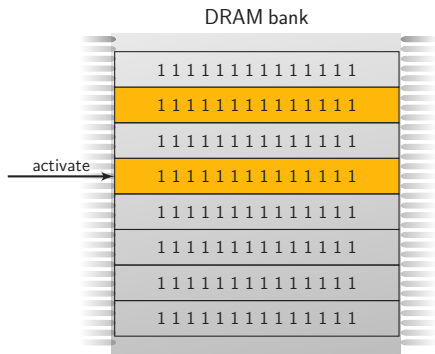


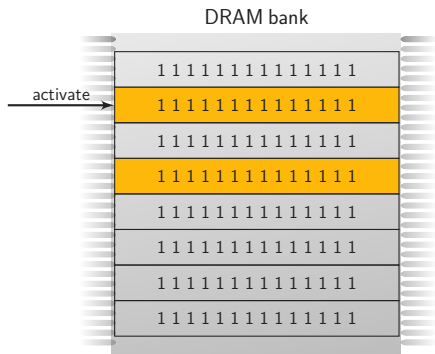


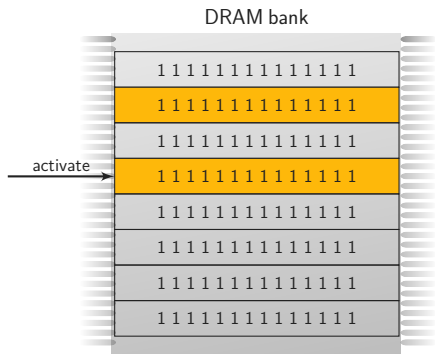


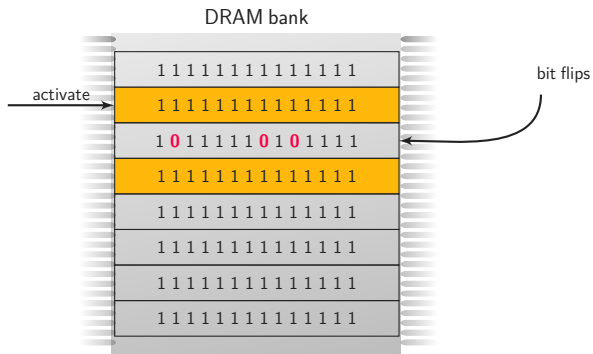


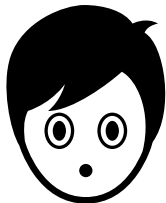




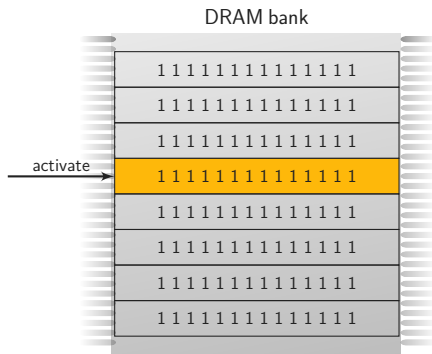


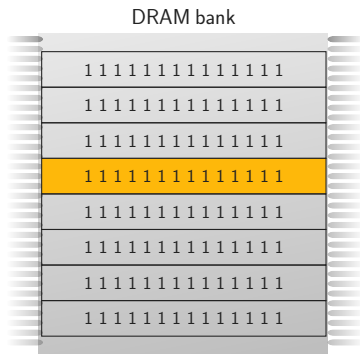


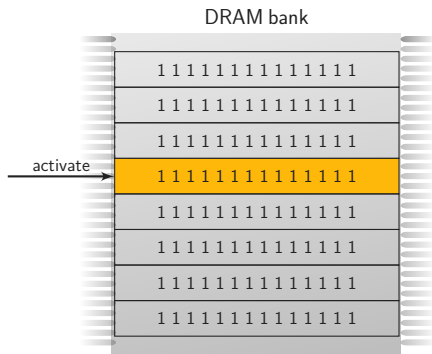


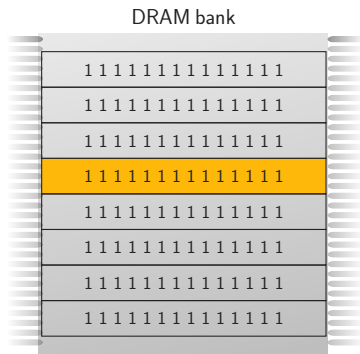


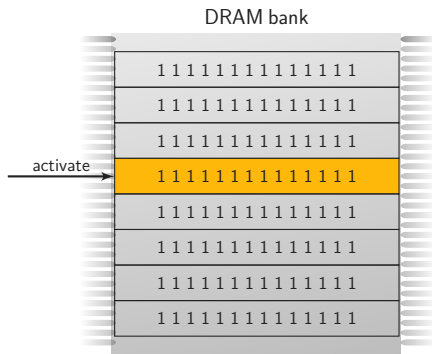
- There are **three** different hammering techniques
- #1: Hammer one row next to victim row and other random rows
- #2: Hammer two rows neighboring victim row
- **#3: Hammer only one row next to victim row**

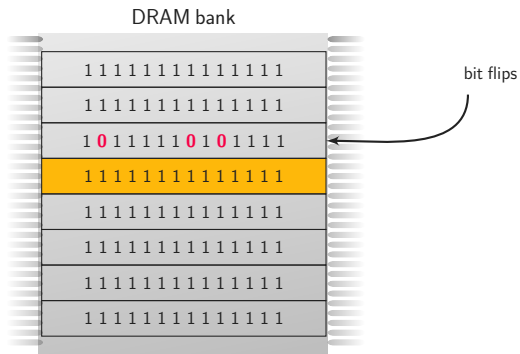


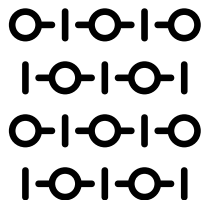




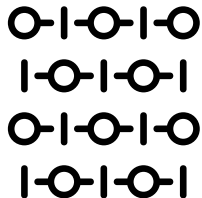




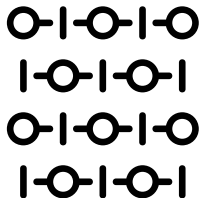




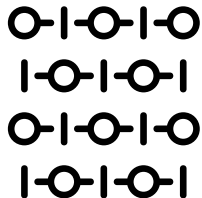
- **Open-page policy:** Keep row opened and buffered



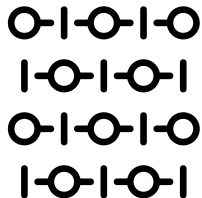
- **Open-page policy:** Keep row opened and buffered
 - Low latency for subsequent accesses to same row
 - High latency for accesses to any other row



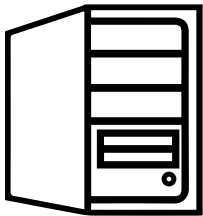
- **Open-page policy:** Keep row opened and buffered
 - Low latency for subsequent accesses to same row
 - High latency for accesses to any other row
- **Close-page policy:** Immediately close row, ready to open a new row



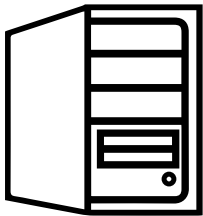
- **Open-page policy:** Keep row opened and buffered
 - Low latency for subsequent accesses to same row
 - High latency for accesses to any other row
- **Close-page policy:** Immediately close row, ready to open a new row
 - Medium latency for accesses to any row



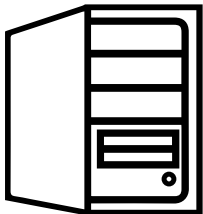
- **Open-page policy:** Keep row opened and buffered
 - Low latency for subsequent accesses to same row
 - High latency for accesses to any other row
- **Close-page policy:** Immediately close row, ready to open a new row
 - Medium latency for accesses to any row
 - Perform better on multi-core systems [Dav+11]



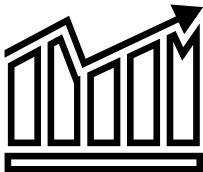
- Policies that preemptively close rows, would allow one-location hammering



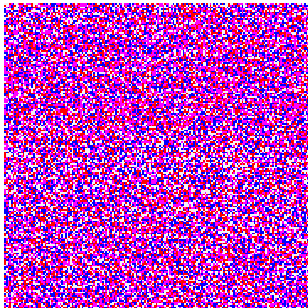
- Policies that preemptively close rows, would allow one-location hammering
- We observed close-page policies on desktop computers



- Policies that preemptively close rows, would allow one-location hammering
- We observed close-page policies on desktop computers
- Mobile devices (e.g., laptops) seem to use mostly open-page policies



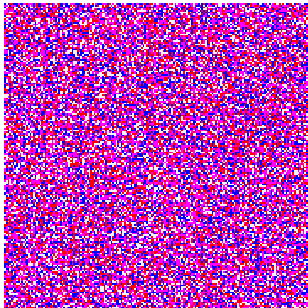
- Distribution of bit flips over 4 kB-aligned memory regions
- Test each technique for 8 hours
- Scanned for bit flips after every hammering attempt
 - Hammering a random location of more than 100 000 randomly-chosen 4 kB pages



Double-sided

77.0 % bit offsets

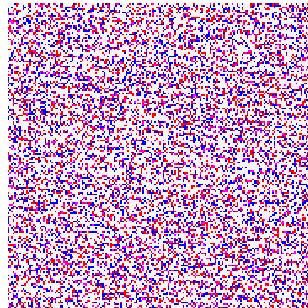
51.7 % 0→1 bit flips



Single-sided

78.5 % bit offsets

54.1 % 0→1 bit flips



One-location

36.5 % bit offsets

51.6 % 0→1 bit flips

What if we cannot target kernel pages?

What if we cannot target kernel pages?

Opcode Flipping



- Many applications perform actions as root



- Many applications perform actions as root
- They can be used by unprivileged users as well



- Many applications perform actions as root
- They can be used by unprivileged users as well
- Implicitly: e.g., ping or mount



- Many applications perform actions as root
- They can be used by unprivileged users as well
- Implicitly: e.g., ping or mount
- Explicitly: **sudo**



- Many applications perform actions as root
- They can be used by unprivileged users as well
- Implicitly: e.g., ping or mount
- Explicitly: **sudo**
- Target sudo (easy to exploit)



















- Conditional jumps are not the only targets



- Conditional jumps are not the only targets
- Other targets include



- Conditional jumps are not the only targets
- Other targets include
 - Comparisons
 - Addresses of memory loads/stores
 - Address calculations
 - ...



- Conditional jumps are not the only targets
- Other targets include
 - Comparisons
 - Addresses of memory loads/stores
 - Address calculations
 - ...
- Manual analysis of `sudo` revealed 29 possible bit flips

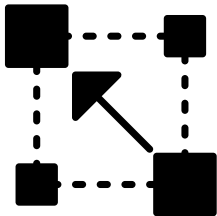


- Conditional jumps are not the only targets
- Other targets include
 - Comparisons
 - Addresses of memory loads/stores
 - Address calculations
 - ...
- Manual analysis of `sudo` revealed 29 possible bit flips
- They all somehow skipped the password check

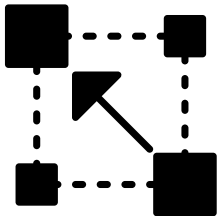
How to get the target virtual page to the target physical location?

How to get the target virtual page to the target physical location?

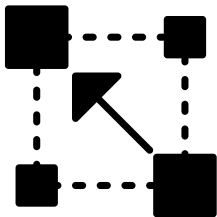
Memory Waylaying



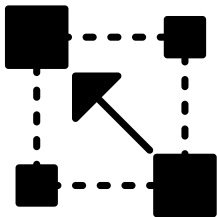
- Maneuver target binary page to a vulnerable physical page



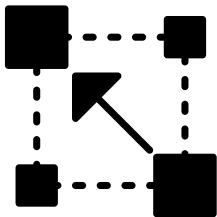
- Maneuver target binary page to a vulnerable physical page
- Not as easy as with page tables



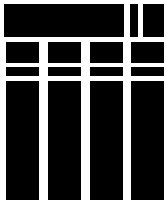
- Maneuver target binary page to a vulnerable physical page
- Not as easy as with page tables
- Binary only once in memory + stays in memory (in the page cache) even after termination



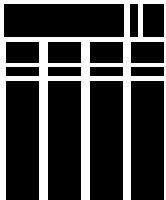
- Maneuver target binary page to a vulnerable physical page
- Not as easy as with page tables
- Binary only once in memory + stays in memory (in the page cache) even after termination
- Only evicted if page cache is full



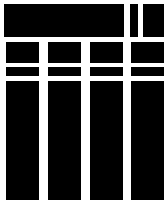
- Maneuver target binary page to a vulnerable physical page
- Not as easy as with page tables
- Binary only once in memory + stays in memory (in the page cache) even after termination
- Only evicted if page cache is full
- Page cache usually occupies all unused memory



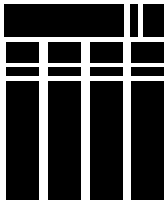
- If a binary is loaded the first time, it is loaded to the memory



- If a binary is loaded the first time, it is loaded to the memory
- It stays in memory (in the page cache) even after execution



- If a binary is loaded the first time, it is loaded to the memory
- It stays in memory (in the page cache) even after execution
- Only evicted if page cache is full



- If a binary is loaded the first time, it is loaded to the memory
- It stays in memory (in the page cache) even after execution
- Only evicted if page cache is full
- Page cache is huge - usually all unused memory



- We propose **memory waylaying** to move a victim page to a target page



- We propose **memory waylaying** to move a victim page to a target page
- Fill page cache with executable pages → evict victim binary



- We propose **memory waylaying** to move a victim page to a target page
- Fill page cache with executable pages → evict victim binary
- Use `mincore` to check for victim eviction

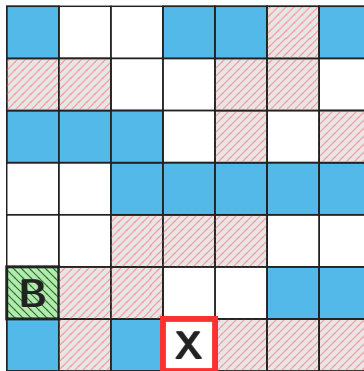


- We propose **memory waylaying** to move a victim page to a target page
- Fill page cache with executable pages → evict victim binary
- Use `mincore` to check for victim eviction
- Reload victim binary → new physical page

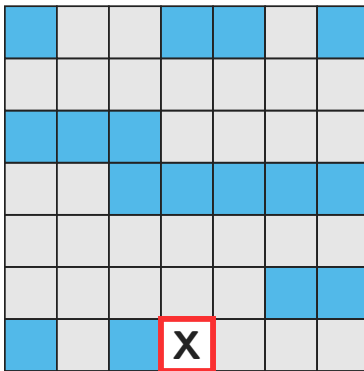


- We propose **memory waylaying** to move a victim page to a target page
- Fill page cache with executable pages → evict victim binary
- Use `mincore` to check for victim eviction
- Reload victim binary → new physical page
- Continue until it is at the target page

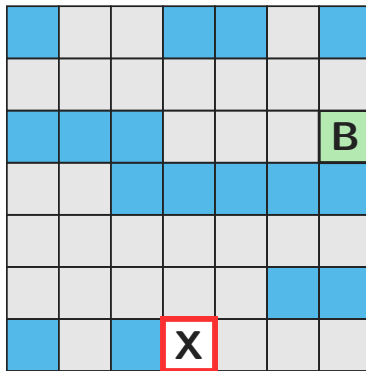
(1) Start



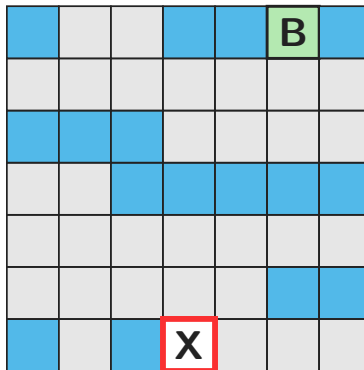
(2) Evict Page Cache



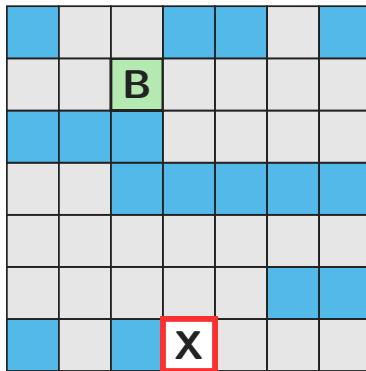
(3) Access Binary



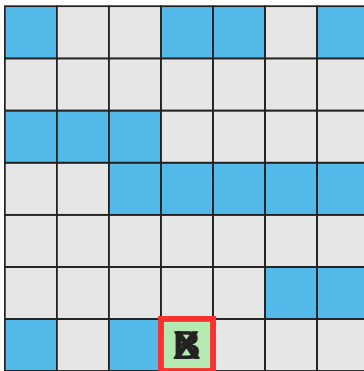
(4) Evict + Access



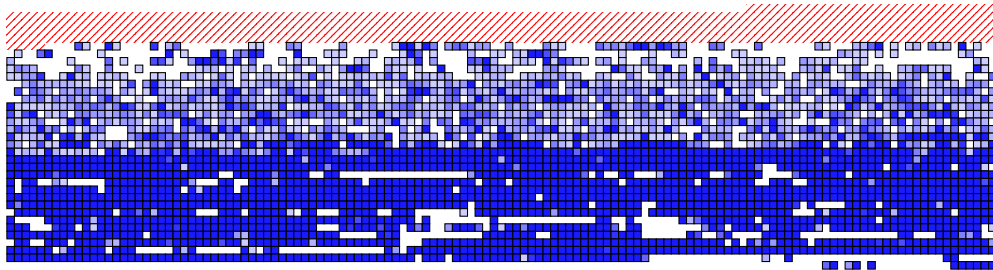
(5) Evict + Access



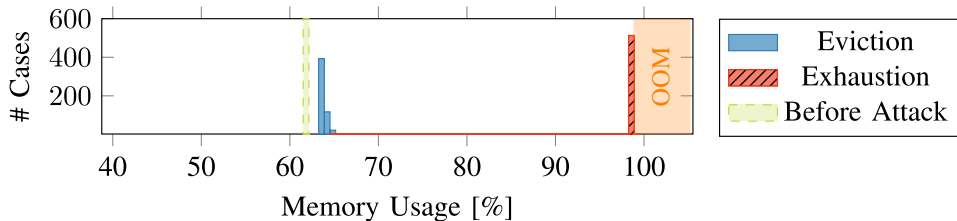
(6) Stop if target reached



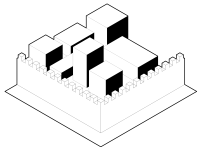
- New pages cover most of the physical memory



- Great advantage over memory massaging: only negligible memory footprint



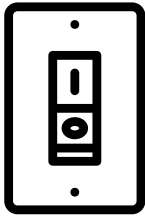
Rowhammer + SGX = Cheap Denial of Service



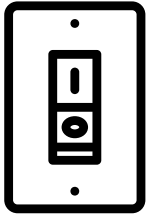
- Instruction-set extension
- Integrity and confidentiality of code and data in untrusted environments
- Run with user privileges and restricted, e.g., no system calls
- Run programs in enclaves using protected areas of memory



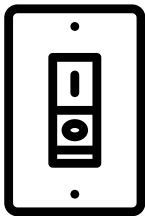




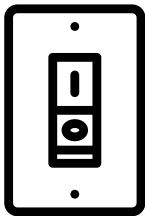
- What happens if a bit flips in the EPC?



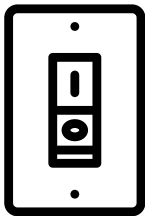
- What happens if a bit flips in the EPC?
- Integrity check will fail!



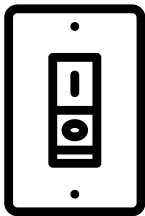
- What happens if a bit flips in the EPC?
 - Integrity check will fail!
- Locks up the memory controller



- What happens if a bit flips in the EPC?
 - Integrity check will fail!
- Locks up the memory controller
- Not a single further memory access!

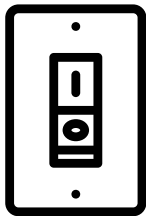


- What happens if a bit flips in the EPC?
- Integrity check will fail!
- Locks up the memory controller
- Not a single further memory access!
- System halts immediately



- What happens if a bit flips in the EPC?
- Integrity check will fail!
- Locks up the memory controller
- Not a single further memory access!
- System halts immediately

Sounds unsafe?



- What happens if a bit flips in the EPC?
- Integrity check will fail!
- Locks up the memory controller
- Not a single further memory access!
- System halts immediately

Sounds unsafe? **It is unsafe!**



- If a malicious enclave induces a bit flip, ...



- If a malicious enclave induces a bit flip, ...
- ...the entire machine halts



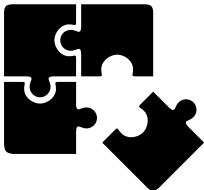
- If a malicious enclave induces a bit flip, ...
- ...the entire machine halts
- ...including co-located tenants

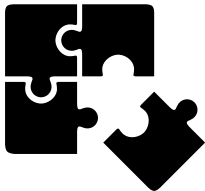


- If a malicious enclave induces a bit flip, ...
- ...the entire machine halts
- ...including co-located tenants
- Denial-of-Service Attacks in the Cloud [Gru+18; Jan+17]

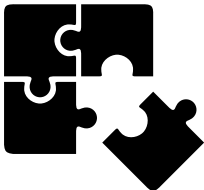
**SGX + One-location Hammering + Opcode Flipping =
Undetectable Exploit**

- SGX protects software from malicious environments





- SGX protects software from malicious environments
- Thwarts static and dynamic (= performance counters) analysis



- SGX protects software from malicious environments
- Thwarts static and dynamic (= performance counters) analysis
- Hammering from SGX defeats countermeasures relying on this:
 - MASCAT
 - ANVIL
 - HexPADS
 - Herath and Fogh
 - Gruss et al.
 - Zhang et al.
 - Chiappetta et al.

Bypass	Defense Class	Static Analysis	Performance Counters	Memory Access Pattern	Physical Proximity	Memory footprint
	Intel SGX	●	●	○	○	○
	One-location hammering	○	○	●	○	○
	Opcode flipping	○	○	○	●	○
	Memory waylaying	○	○	○	○	●
	Defense class defeated	●	●	●	●	●

Luckily, all of these are local attacks...

- $\geq 43\,000$ hammering attempts (within 64 ms) for a bit flip [GMM16]

- $\geq 43\,000$ hammering attempts (within 64 ms) for a bit flip [GMM16]
= 671 875 accesses per second

- $\geq 43\,000$ hammering attempts (within 64 ms) for a bit flip [GMM16]
- = 671 875 accesses per second
- Network packets access memory location up to 6 times (depending on kernel)

- $\geq 43\,000$ hammering attempts (within 64 ms) for a bit flip [GMM16]
= 671 875 accesses per second
- Network packets access memory location up to 6 times
(depending on kernel)
→ 111 979 packets per second

- $\geq 43\,000$ hammering attempts (within 64 ms) for a bit flip [GMM16]
- = 671 875 accesses per second
- Network packets access memory location up to 6 times
(depending on kernel)
- 111 979 packets per second
- Network packets are at least 64 B

- $\geq 43\,000$ hammering attempts (within 64 ms) for a bit flip [GMM16]

= 671 875 accesses per second

- Network packets access memory location up to 6 times
(depending on kernel)

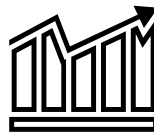
→ 111 979 packets per second

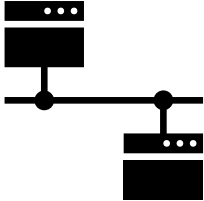
- Network packets are at least 64 B

= 7 166 656 B/s = 7 MB/s = 57 Mb/s

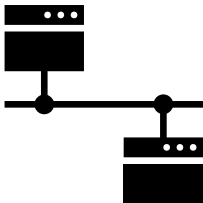
- $\geq 43\,000$ hammering attempts (within 64 ms) for a bit flip [GMM16]
= 671 875 accesses per second
- Network packets access memory location up to 6 times
(depending on kernel)
→ 111 979 packets per second
- Network packets are at least 64 B
= 7 166 656 B/s = 7 MB/s = 57 Mb/s
→ That sounds doable on “modern” networks

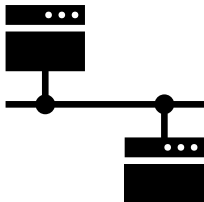
- $\geq 43\,000$ hammering attempts (within 64 ms) for a bit flip [GMM16]
= 671 875 accesses per second
- Network packets access memory location up to 6 times
(depending on kernel)
→ 111 979 packets per second
- Network packets are at least 64 B
= 7 166 656 B/s = 7 MB/s = 57 Mb/s
→ That sounds doable on “modern” networks





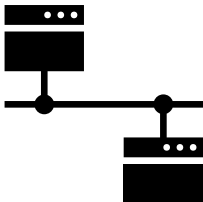
Inducing bit flips:





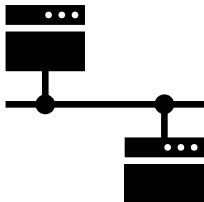
Inducing bit flips:

- Network stacks on ARM often use uncached memory (perfect for hammering)



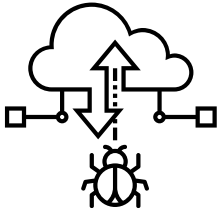
Inducing bit flips:

- Network stacks on ARM often use uncached memory (perfect for hammering)
- Intel recommends Intel CAT for QoS (perfect for hammering)

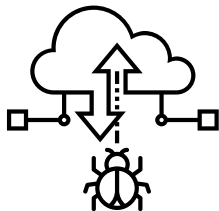


Inducing bit flips:

- Network stacks on ARM often use uncached memory (perfect for hammering)
- Intel recommends Intel CAT for QoS (perfect for hammering)
- Some device drivers / network reachable code uses `clflush` or non-temporal stores (good for hammering)

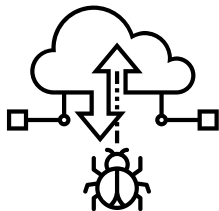


Nethammer on ...



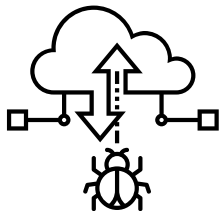
Nethammer on ...

- SGX = powerful DoS



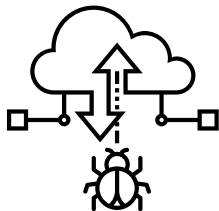
Nethammer on ...

- SGX = powerful DoS
- File system = persistent DoS



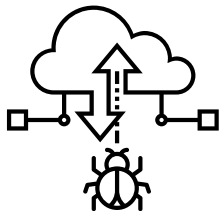
Nethammer on ...

- SGX = powerful DoS
- File system = persistent DoS
- DNS entries on a DNS server = bit-squatting attack



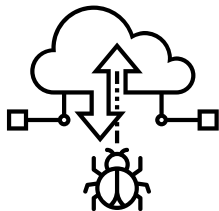
Nethammer on ...

- SGX = powerful DoS
- File system = persistent DoS
- DNS entries on a DNS server = bit-squatting attack
- OCSP servers = make invalid certificates great again

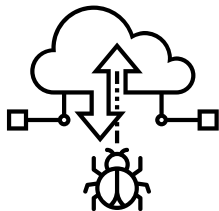


Nethammer on ...

- SGX = powerful DoS
- File system = persistent DoS
- DNS entries on a DNS server = bit-squatting attack
- OCSP servers = make invalid certificates great again
- Crypto = generate private keys for broken public keys

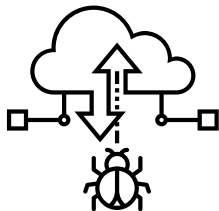


Nethammer on ...



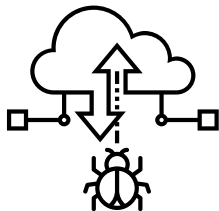
- SGX = powerful DoS
- File system = persistent DoS
- DNS entries on a DNS server = bit-squatting attack
- OCSP servers = make invalid certificates great again
- Crypto = generate private keys for broken public keys
- Crypto + GitLab = manipulate repositories in the name of someone else

Nethammer on ...

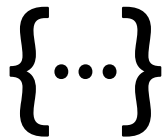


- SGX = powerful DoS
- File system = persistent DoS
- DNS entries on a DNS server = bit-squatting attack
- OCSP servers = make invalid certificates great again
- Crypto = generate private keys for broken public keys
- Crypto + GitLab = manipulate repositories in the name of someone else
 - Bonus: evict the broken key and all traces are gone!

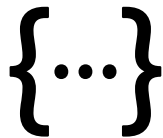
Nethammer on ...



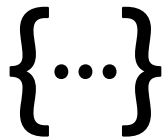
- SGX = powerful DoS
- File system = persistent DoS
- DNS entries on a DNS server = bit-squatting attack
- OCSP servers = make invalid certificates great again
- Crypto = generate private keys for broken public keys
- Crypto + GitLab = manipulate repositories in the name of someone else
 - Bonus: evict the broken key and all traces are gone!
 - Original key owner will have a hard time proving that this was an attacker



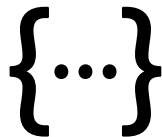
- Many (academic) countermeasures were proposed to mitigate Rowhammer



- Many (academic) countermeasures were proposed to mitigate Rowhammer
- We showed that all of them can be circumvented [Gru+18]



- Many (academic) countermeasures were proposed to mitigate Rowhammer
- We showed that all of them can be circumvented [Gru+18]
- We cannot design countermeasures without completely understanding the attack



- Many (academic) countermeasures were proposed to mitigate Rowhammer
- We showed that all of them can be circumvented [Gru+18]
- We cannot design countermeasures without completely understanding the attack
- Otherwise we only patch concrete exploits, but do not solve the problem

Apple had a great idea:

- Lower refresh rate \rightarrow save energy + more flips





Apple had a great idea:

- Lower refresh rate \rightarrow save energy + more flips
- \rightarrow ECC memory \rightarrow fewer flips



Apple had a great idea:

- Lower refresh rate \rightarrow save energy + more flips
- \rightarrow ECC memory \rightarrow fewer flips
- It's an optimization problem.



Apple had a great idea:

- Lower refresh rate \rightarrow save energy + more flips
- \rightarrow ECC memory \rightarrow fewer flips
- It's an optimization problem.
 - Too aggressive? \rightarrow bit flips



Apple had a great idea:

- Lower refresh rate → save energy + more flips
- ECC memory → fewer flips
- It's an optimization problem.
 - Too aggressive? → bit flips
 - Too cautious? → waste of energy



Apple had a great idea:

- Lower refresh rate → save energy + more flips
- ECC memory → fewer flips
- It's an optimization problem.
 - Too aggressive? → bit flips
 - Too cautious? → waste of energy
 - What if the “too aggressive” changes over time?



Apple had a great idea:

- Lower refresh rate → save energy + more flips
- ECC memory → fewer flips
- It's an optimization problem.
 - Too aggressive? → bit flips
 - Too cautious? → waste of energy
 - What if the “too aggressive” changes over time?
 - What if attackers come up with slightly better attacks?



Apple had a great idea:

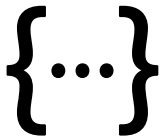
- Lower refresh rate → save energy + more flips

→ ECC memory → fewer flips

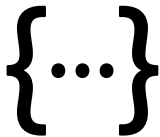
- It's an optimization problem.
 - Too aggressive? → bit flips
 - Too cautious? → waste of energy
 - What if the “too aggressive” changes over time?
 - What if attackers come up with slightly better attacks?
- Difficult to optimize with an adversary working against you

{...}

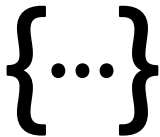
- We have to invest more into researching attacks



- We have to invest more into researching attacks
- There are still aspects of Rowhammer we do not fully understand



- We have to invest more into researching attacks
- There are still aspects of Rowhammer we do not fully understand
- However, this is required to design effective countermeasures



- We have to invest more into researching attacks
- There are still aspects of Rowhammer we do not fully understand
- However, this is required to design effective countermeasures
- Moreover, new features might introduce new attack vectors (e.g., SGX)



- We underestimated side-channel attacks for a long time



- We underestimated side-channel attacks for a long time
- Industry and customers have to reconsider priorities → focus more on security instead of performance



- We underestimated side-channel attacks for a long time
- Industry and customers have to reconsider priorities → focus more on security instead of performance
- Reliability issues (Rowhammer) can have security impacts



- We underestimated side-channel attacks for a long time
- Industry and customers have to reconsider priorities → focus more on security instead of performance
- Reliability issues (Rowhammer) can have security impacts
- More research is required to understand attacks to ultimately mitigate them

ANOTHER FLIP IN THE ROW

DANIEL GRUSS, MORITZ LIPP, MICHAEL SCHWARZ

AUGUST 9, 2018

GRAZ UNIVERSITY OF TECHNOLOGY

References



N. A. Anagnostopoulos, T. Arul, Y. Fan, C. Hatzfeld, A. Schaller, W. Xiong, M. Jain, M. U. Saleem, J. Lotichius, S. Gabmeyer, et al. Intrinsic Run-Time Row Hammer PUFs: Leveraging the Row Hammer Effect for Run-Time Cryptography and Improved Security. In: *Cryptography* 2.3 (2018), p. 13.



Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin. ANVIL: Software-based protection against next-generation Rowhammer attacks. In: *ACM SIGPLAN Notices* 51.4 (2016), pp. 743–755.



S. Bhattacharya and D. Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In: *Conference on Cryptographic Hardware and Embedded Systems (CHES)*. 2016.



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



F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In: USENIX Security Symposium. 2017.



J. Corbet. Defending against Rowhammer in the kernel. Oct. 2016. URL: <https://lwn.net/Articles/704920/>.



M. Chiappetta, E. Savas, and C. Yilmaz. Real time detection of cache-based side-channel attacks using Hardware Performance Counters. Cryptology ePrint Archive, Report 2015/1034. 2015.



Y. Cheng, Z. Zhang, and S. Nepal. Still Hammerable and Exploitable: on the Effectiveness of Software-only Physical Kernel Isolation. In: arXiv:1802.07060 (2018).



H. David, C. Fallin, E. Gorbato, U. R. Hanebutte, and O. Mutlu. Memory power management via dynamic voltage/frequency scaling. In: ACM International Conference on Autonomic Computing. 2011.



P. Frigo, C. Giuffrida, H. Bos, and K. Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In: IEEE S&P. 2018.



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



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



D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoecl, and Y. Yarom. Another Flip in the Wall of Rowhammer Defenses. In: S&P. 2018.



N. Herath and A. Fogh. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In: Black Hat Briefings. Aug. 2015. 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>.



G. Irazoqui, T. Eisenbarth, and B. Sunar. MASCAT: Stopping Microarchitectural Attacks Before Execution. Cryptology ePrint Archive, Report 2016/1196. 2017.



Y. Jang, J. Lee, S. Lee, and T. Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In: SysTEX. 2017.



Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: ISCA'14. 2014.



M. Lanteigne. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. Mar. 2016. URL: <http://www.thirdio.com/rowhammer.pdf>.



M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster. Nethammer: Inducing Rowhammer Faults through Network Requests. In: arXiv:1711.08002 (2017).



M. Payer. HexPADS: a platform to detect “stealth” attacks. In: ESSoS’16. 2016.



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



D. Poddebniak, J. Somorovsky, S. Schinzel, M. Lochter, and P. Rösler. Attacking deterministic signature schemes using fault attacks. In: EuroS&P. 2018.



R. Qiao and M. Seaborn. A New Approach for Rowhammer Attacks. In: International Symposium on Hardware Oriented Security and Trust. 2016.



K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In: USENIX Security Symposium. 2016.



M. Seaborn and T. Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In: Black Hat Briefings. 2015.



A. Tatar, R. Krishnan, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi. Throwhammer: Rowhammer Attacks over the Network and Defenses. In: USENIX ATC. 2018.



V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In: CCS'16. 2016.



V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In: DIMVA. 2018.



Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In: USENIX Security Symposium. 2016.

Method	Bit flips	Templating	Waylaying	Total
Double-sided, waylaying	91	26.1 h	69.4 h	95.5 h
Single-sided, waylaying	87	27.5 h	70.6 h	98.1 h
One-location, waylaying	50	47.3 h	90.5 h	137.8 h
<hr/>				
Double-sided, chasing	1	0.7 h	43.7 h	44.4 h
Single-sided, chasing	1	0.7 h	43.7 h	44.4 h
One-location, chasing	1	1.3 h	44.0 h	45.4 h