# Rowhammer

# CS642: Computer Security

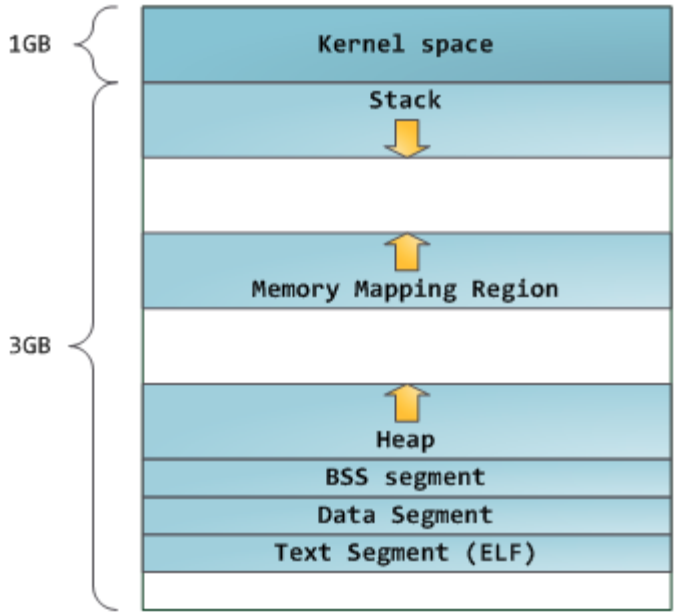
I can't talk loudly
Please sit close to the front

Office hours today 11:15-12:15

# Outline

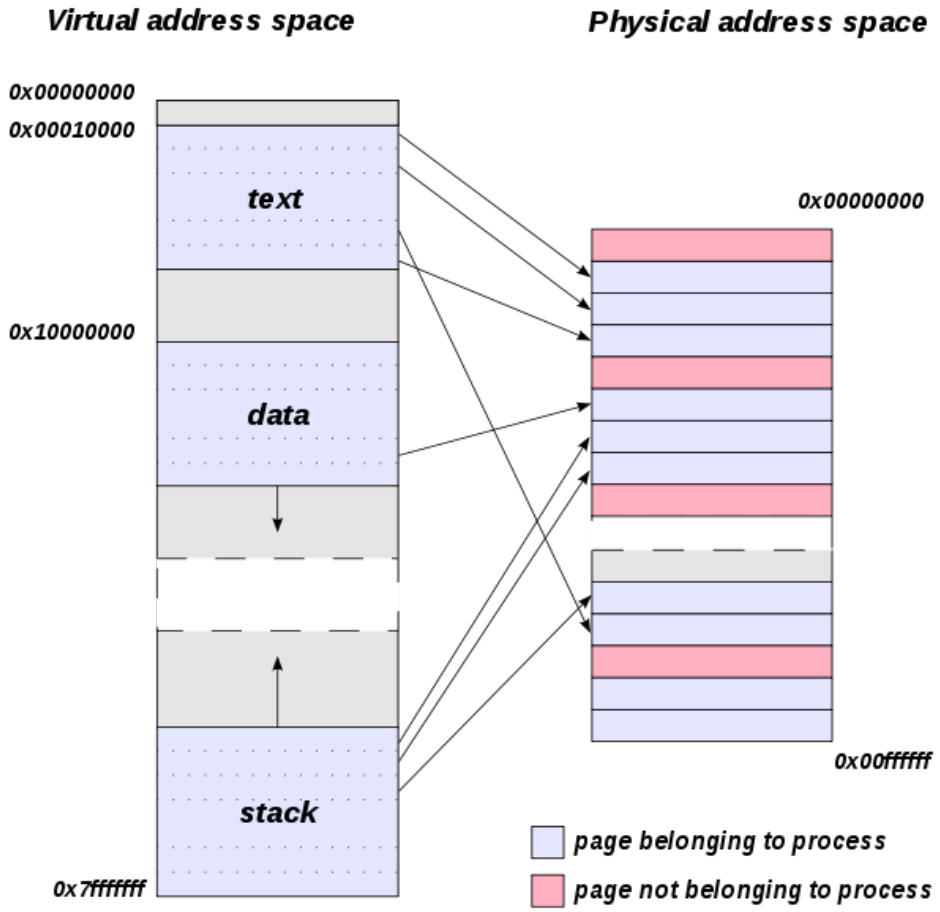
- Spectre/meltdown review
- Rowhammer
- Hardware security summary

# Isolating kernel memory from userland




- Userland code must not directly access kernel memory.
- Kernel contains sensitive info:
  - Info about other processes.
  - Typically, all of physical memory is mapped into the kernel address space.
- If userland code attempts to directly access kernel memory, hardware triggers an exception.

# Virtual and physical memory




- The OS maps each process' virtual address space to physical memory via per-process page tables.
- Pages tables for all user processes are managed by the kernel, i.e., kernel knows virtual to physical mappings for all processes.
- Kernel itself is mapped into process address space: Kernel's own virtual to physical mappings are part of the page table.

# Metldown bug

- CPU will speculatively use page table entries with valid bit == 0 and user/kernel bit set to kernel

- Result:
  - inaccessible data loaded into cache
    - but can’t be read
  - Inaccessible data used in processor pipeline for subsequent instructions
    - Can be used to decide what to speculatively load into cache

# Meltdown: The Covert Channel Setup

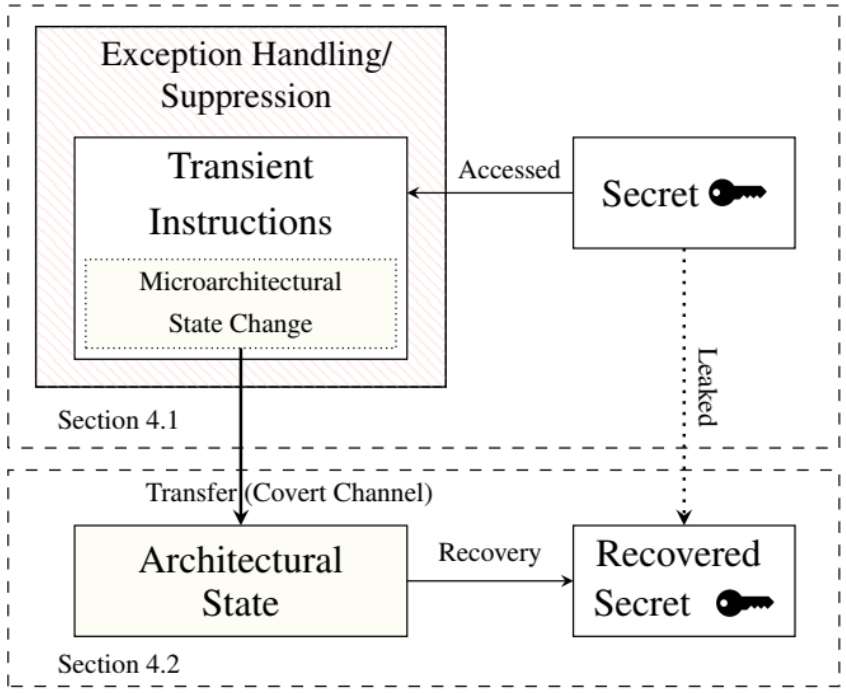
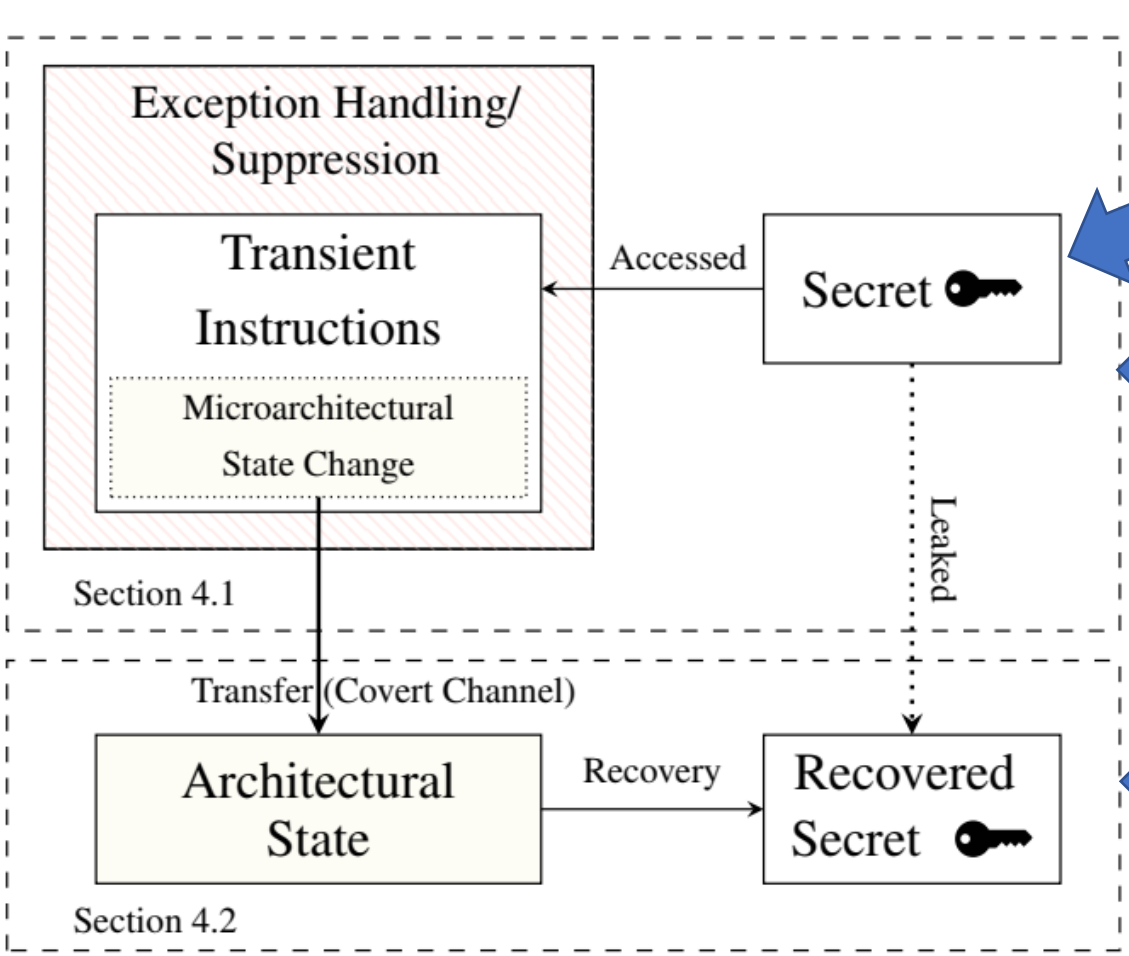



Figure 5: The Meltdown attack uses exception handling or suppression, e.g., TSX, to run a series of transient instructions. These transient instructions obtain a (persistent) secret value and change the microarchitectural state of the processor based on this secret value. This forms the sending part of a microarchitectural covert channel. The receiving side reads the microarchitectural state, making it architectural and recovering the secret

# Meltdown: The Covert Channel Setup

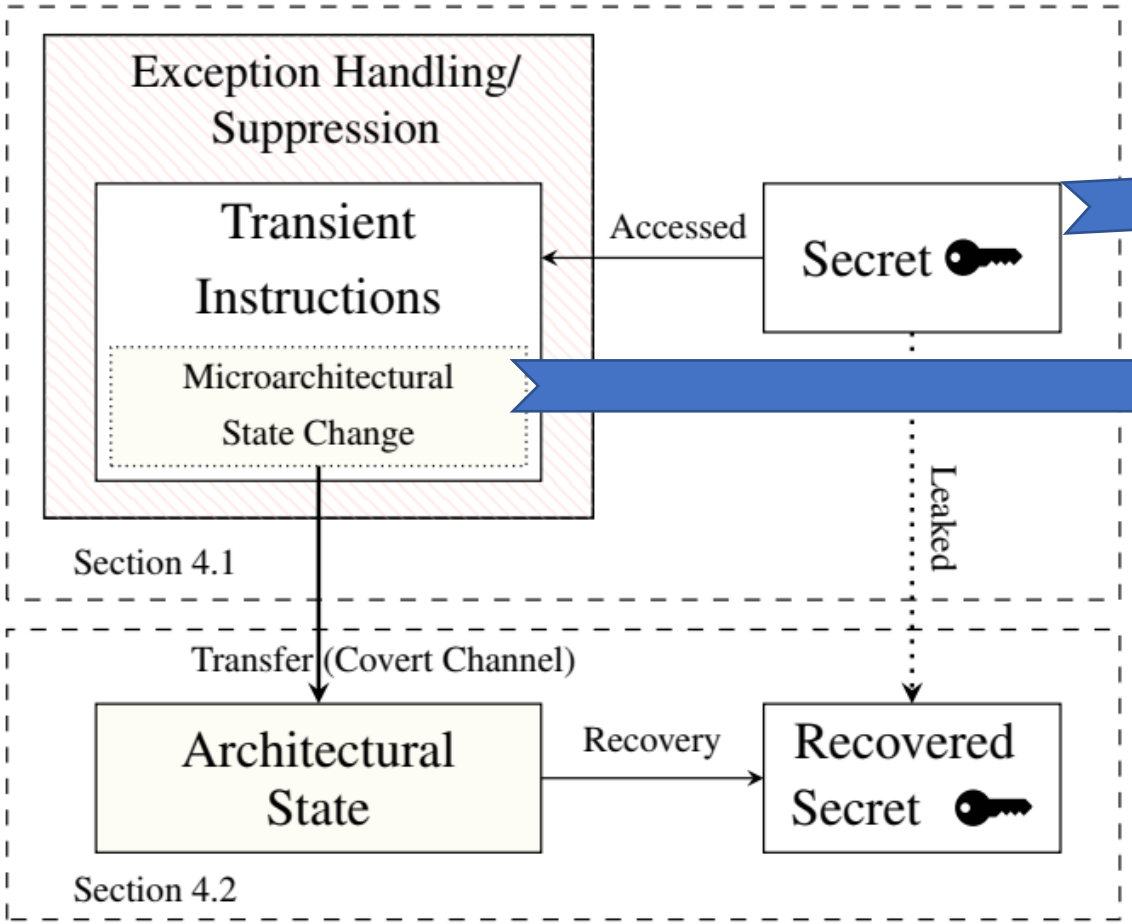



A value of interest, stored in kernel memory (typically inaccessible to user processes)

Forked child process (dies on exception, but affects cache state)

Parent process

# Meltdown: Transient Instruction Sequence




```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

# The Meltdown Attack

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- **Goal**: Attacker wants to learn the value of the byte stored at a particular kernel memory address (address in the `rcx` register).
- **Step 1: Reading the secret (Line 4) mov al, byte [rcx]**
  - Loads the byte value stored at the address RCX into AL (LSB of RAX)
  - This instruction should cause an exception if executed in userland. Subsequent instructions should never be executed.
  - **BUT,** due to OOO, subsequent instructions may already be executed speculatively.
  - Exceptions handled only when **Line 4 is retired.** By then, microarchitectural state is already affected by subsequent OOO instruction execution.

# The Meltdown Attack

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- **Goal**: Attacker wants to learn the value of the byte stored at a particular kernel memory address (address in the `rcx` register).
- **Step 2: Transmitting the secret (Line 5) shl rax, 0xc**
  - Multiply the byte value **X** by the page size (4K).
  - This will be used to index into a **probe array** (base address in RBX)**.**
  - A large spatial distance ensures that neighboring locations of the probe array are not loaded into the cache (due to spatial locality optimizations).
  - Probe array is of size 256 * 4K bytes, since we only have 256 possible byte values.

# The Meltdown Attack

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- **Goal**: Attacker wants to learn the value of the byte stored at a particular kernel memory address (address in the `rcx` register).
- **Step 2: Transmitting the secret (Line 7) mov rbx, qword [rbx + rax]**
  - Read Probe_Array[X] (each entry is 4K bytes long).
  - The value will be stored into the corresponding cache line
- **Step 3: Receiving the secret (Parent process)**
  - Parent process probes the cache by iterating through Probe_Array[].
  - Only the read of **Probe_Array[X]** will be a hit in the cache.
  - Attacker learns the value of **X**

# The Meltdown Attack

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- What is the role of line 3 and line 6?
  - Race conditions!
  - The attacker is racing against the hardware: Must get transient instructions to execute and affect microarchitectural state before the exception for **line 4** is thrown.
  - In some machines, exception is not handled, and process crashes, but processor zeroes out registers before crashing the process.
  - If zeroing out happens faster than the operation in **line 5,** attacker will read the wrong value for **X.** So the code retries.

# Meltdown attack application: Memory dumps

- Can iterate attack across a range of memory addresses to obtain a complete memory dump of the kernel.
- Physical memory on modern machines mapped at an offset within the kernel. So complete dump of physical memory is possible.

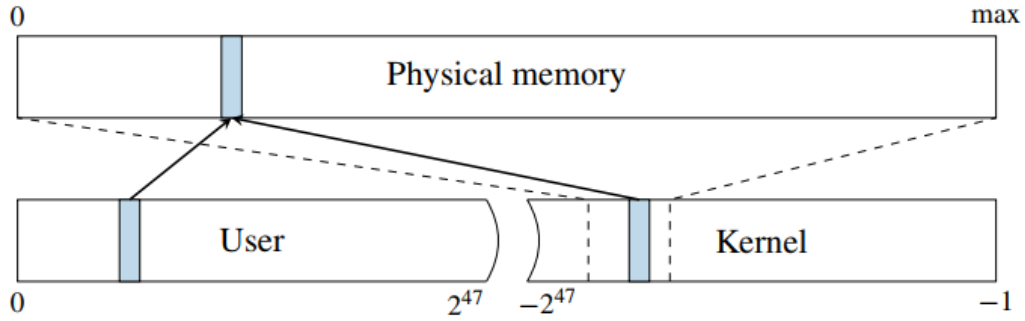



Figure 2: The physical memory is directly mapped in the kernel at a certain offset. A physical address (blue) which is mapped accessible for the user space is also mapped in the kernel space through the direct mapping.

# Meltdown attack application: Memory dumps

- Attack against Firefox56 running atop a Ubunto 16.10/Linux-4.8.0 machine on Intel Corei7-6700K

```
f94b7690: e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 e5 |................|
f94b76a0: e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 e5 |................|
f94b76b0: 70 52 b8 6b 96 7f XX XX XX  XX XX XX XX XX XX XX |pR.k............|
f94b76c0: 09 XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b76d0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b76e0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX 81 |................|
f94b76f0: 12 XX e0 81 19 XX e0 81 44  6f 6c 70 68 69 6e 31 |........Dolphin1|
f94b7700: 38 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |8...............|
f94b7710: 70 52 b8 6b 96 7f XX XX XX  XX XX XX XX XX XX XX |pR.k............|
f94b7720: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7730: XX XX XX XX 4a XX XX XX XX  XX XX XX XX XX XX XX |....J...........|
f94b7740: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7750: XX XX XX XX XX XX XX XX XX  XX e0 81 69 6e 73 74 |............inst|
f94b7760: 61 5f 30 32 30 33 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |a_0203..........|
f94b7770: 70 52 18 7d 28 7f XX XX XX  XX XX XX XX XX XX XX |pR.}(...........|
f94b7780: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7790: XX XX XX XX 54 XX XX XX XX  XX XX XX XX XX XX XX |....T...........|
f94b77a0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b77b0: XX XX XX XX XX XX XX XX XX  XX XX XX 73 65 63 72 |............secr|
f94b77c0: 65 74 70 77 64 30 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |etpwd0..........|
f94b77d0: 30 b4 18 7d 28 7f XX XX XX  XX XX XX XX XX XX XX |0..}(...........|
f94b77e0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b77f0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7800: e5 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |................|
f94b7810: 68 74 74 70 73 3a 2f 2f 61  64 64 6f 6e 73 2e 63 |https://addons.c/|
f94b7820: 64 6e 2e 6d 6f 7a 69 6c 6c  61 2e 6e 65 74 2f 75 |dn.mozilla.net/u|
f94b7830: 73 65 72 2d 6d 65 64 69 61  2f 61 64 64 6f 6e 5f |ser-media/addon_|
f94b7840: 69 63 6f 6e 73 2f 33 35 34  2f 33 35 34 33 39 39 |icons/354/354399|
f94b7850: 2d 36 34 2e 70 6e 67 3f 6d  6f 64 69 66 69 65 64 |-64.png?modified|
f94b7860: 3d 31 34 35 32 32 34 34 38  31 35 XX XX XX XX XX |=1452244815.....|
```

Listing 4: Memory dump of Firefox 56 on Ubuntu 16.10 on a Intel Core i7-6700K disclosing saved passwords (cf.

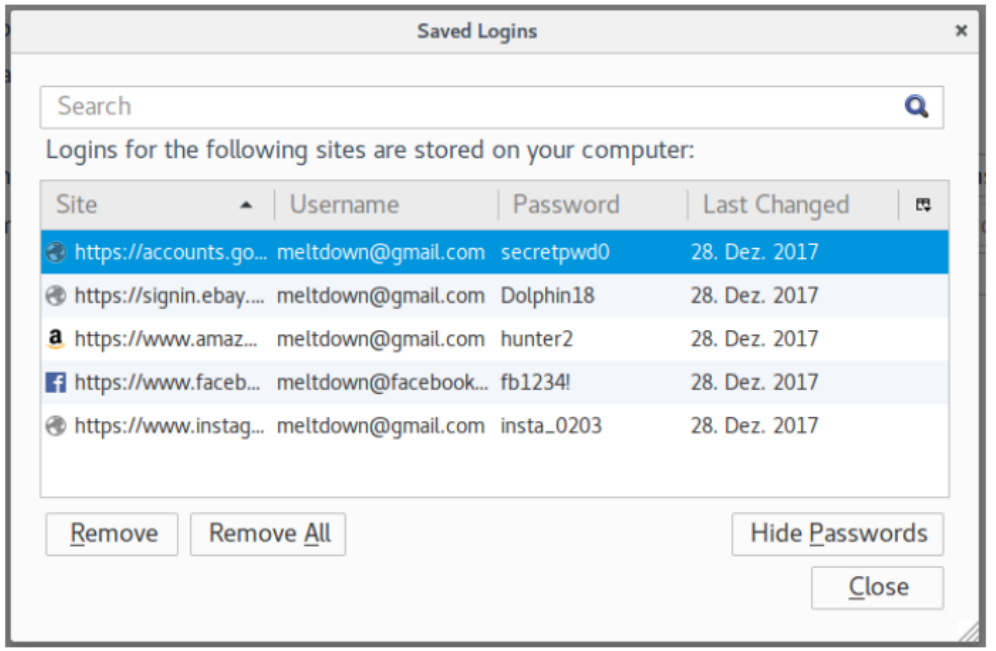
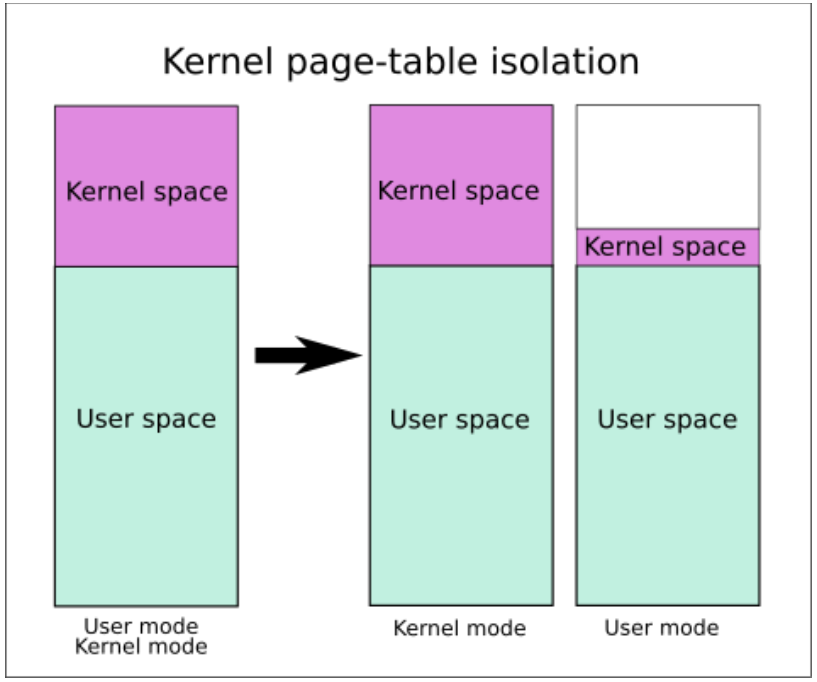



Figure 6: Firefox 56 password manager showing the stored passwords that are leaked using Meltdown in Listing 4.

# Meltdown attack status

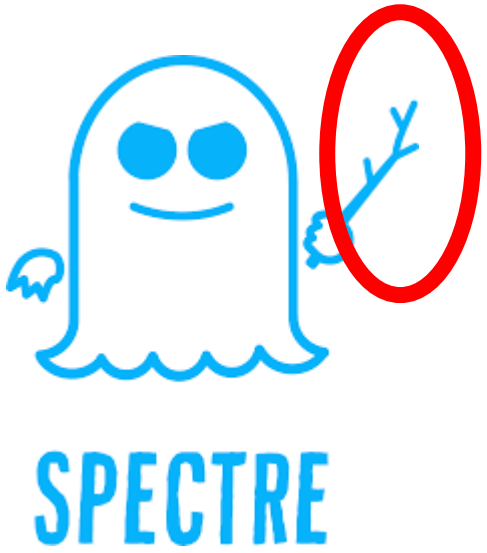
- Applied successfully on several Intel processors on various OSes (Linux-2.6.32 to 4.13.0), Windows 10, Docker, LXC, and OpenVZ.
- Proposed defense: **KPTI (Kernel Page Table Isolation).**
  - Being integrated into various OSes.
  - Long-term effectiveness is unclear.
  - Also, still seems controversial:

**Linus Torvalds declares Intel fix for Meltdown/Spectre 'COMPLETE AND UTTER GARBAGE'**



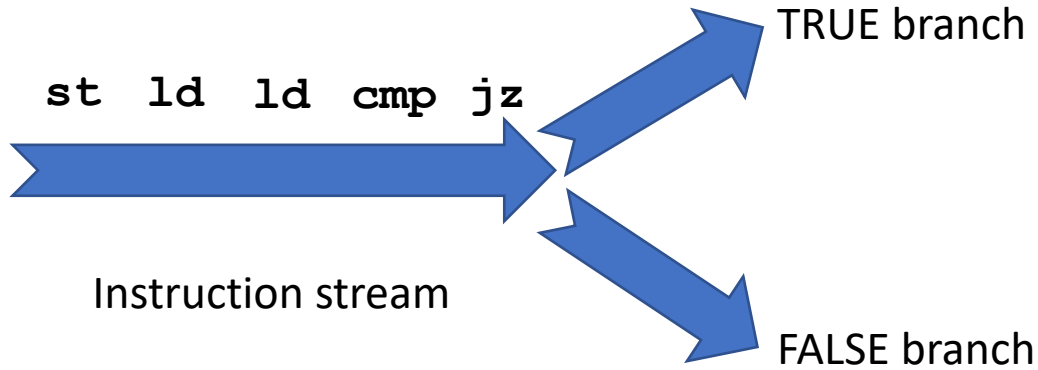
# Spectre

- Affects a wide variety of processors (Intel, ARM, AMD).
- Uses another form of speculative execution: **branch prediction.**
- Slightly harder to deploy than Meltdown, in that a “host” program is required, which contains certain instruction sequences that can be misused.



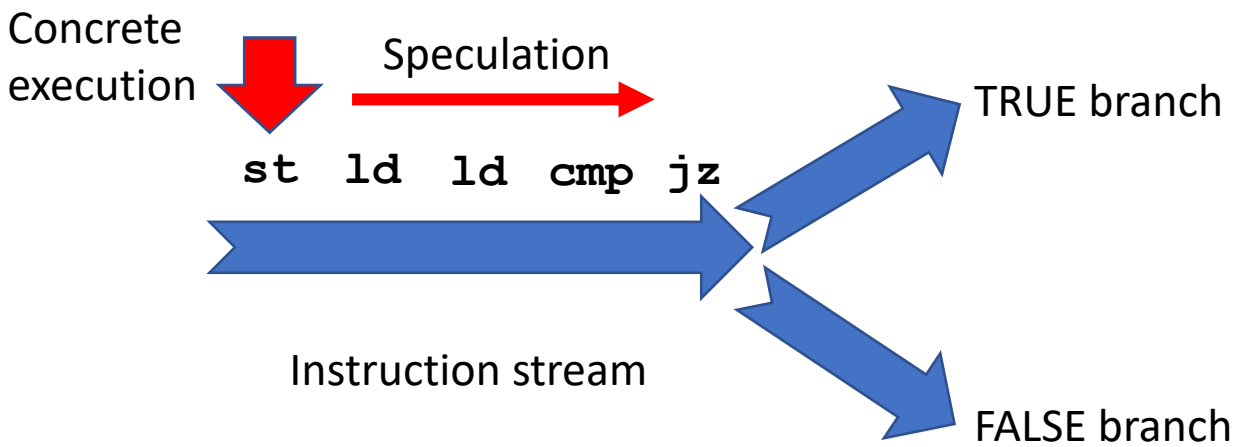
# Building block: Branch prediction

- In OOO, what happens when the speculative execution engine reaches a branch?
- **Hardware branch predictor** predicts a likely outcome of the branch (based on past history), and continues to speculate along the (likely) taken branch.



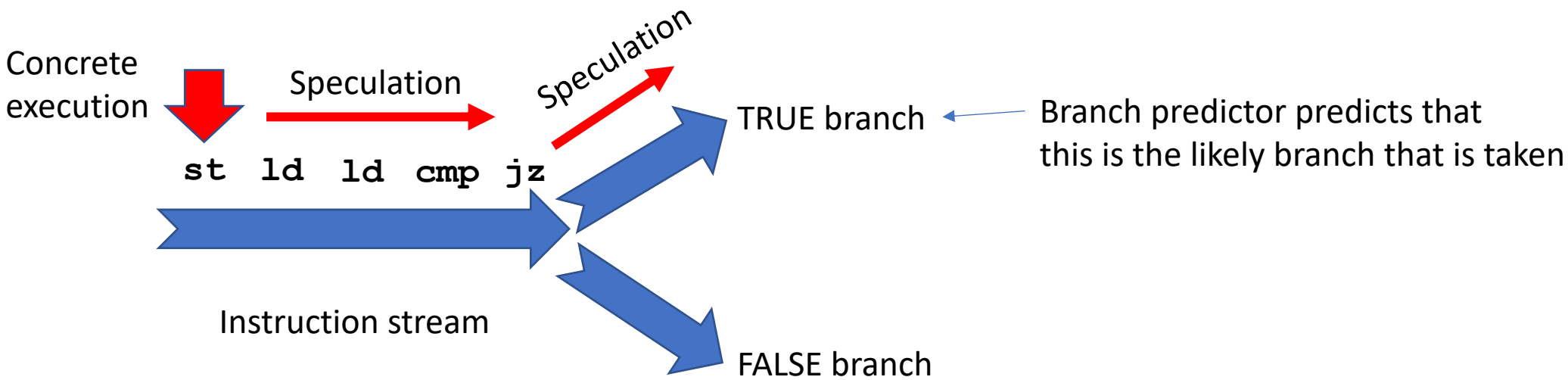
# Building block: Branch prediction

- In OOO, what happens when the speculative execution engine reaches a branch?
- **Hardware branch predictor** predicts a likely outcome of the branch (based on past history), and continues to speculate along the (likely) taken branch.



# Building block: Branch prediction

- In OOO, what happens when the speculative execution engine reaches a branch?
- **Hardware branch predictor** predicts a likely outcome of the branch (based on past history), and continues to speculate along the (likely) taken branch.



# Basic setup of Spectre attack

- In a “host” program (the victim of the attack), find an instruction sequence with a branch.
- **Preparation**: Execute the program to train the branch predictor to go in one direction (say, TRUE)
- **Attack**: Feed it a malicious input that would cause the branch to go the other direction (i.e., FALSE), but rely on branch predictor to execute the TRUE branch. Use the speculatively executed TRUE branch to extract data from the victim program.

# Consider a host program with this snippet

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

`array1` is a unsigned byte array of size `array1_size`

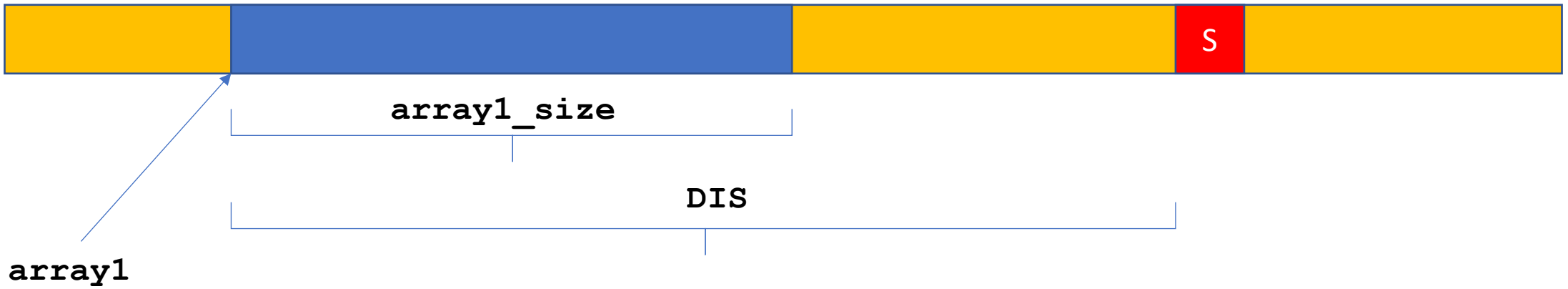
`array2` is of size 64KB (256*256)

Suppose the value of `x` is derived from user input to the program (and can therefore be controlled by attacker).

In this program, there is some secret data **S** that you wish to access

# Consider a host program with this snippet

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```




**Observe: `array1[DIS]` obtains `S`**

# Attack preparation

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

1. Execute the program long enough with a number of values of x, so that the branch predictor is trained to take the true branch.
2. Arrange for cache to **not contain** `array2` and `array1_size`.
3. Arrange for cache to **contain** secret value S. How? E.g., S could be a cryptographic key you want to learn. Arrange for a cryptographic computation to happen that uses S.

# Actual Spectre attack

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

Now execute the program with `x = DIS`

1. x < array1_size will lead to a cache miss. Leads to a delay in fetching array1_size. Processor speculates on branch.
2. Speculative code reads array1[DIS]. A hit in the cache (the value S)
3. Code then proceeds to read array2[S*256]. A miss in the cache.

However, array1_size may have arrived by then. Processor realizes mistake in speculation. But too late…the speculative read array2[S*256] already affects cache state

# Actual Spectre attack

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

However, array1_size may have arrived by then. Processor realizes mistake in speculation. But too late…the speculative read array2[S*256] already affects cache state.

If array2 is accessible to the attacker, just probe all its elements and use cache-timing to figure out the value of S. (Many options possible here to “transmit” the microarchitectural state to the attacker).

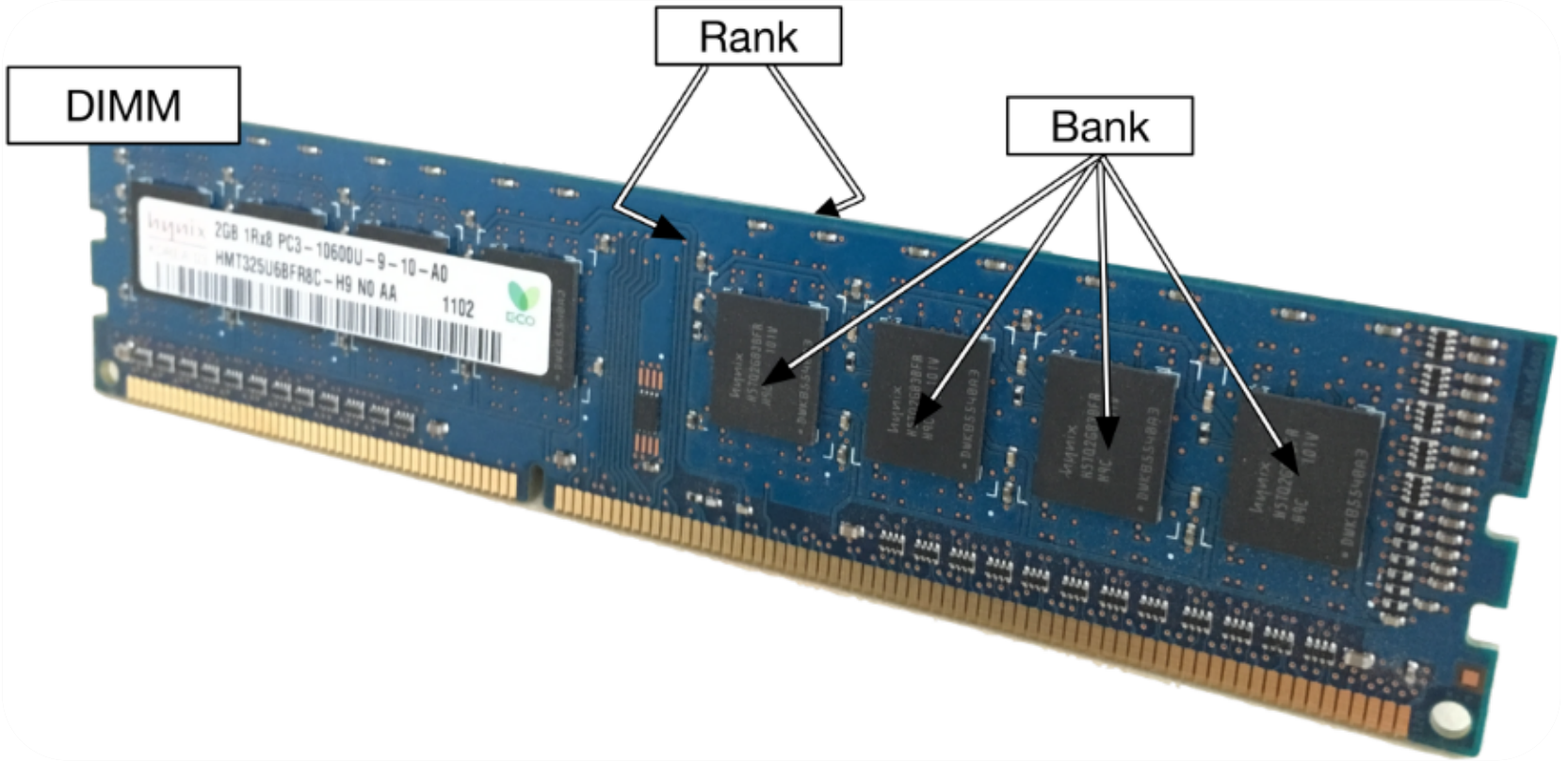
# Notes about Spectre

- Not restricted to host programs that have such a convenient code sequence built in.
  - Can search for “gadgets” (short instruction sequences) that can be “weaved” together to achieve desired effect (“Return-oriented Programming”, for those students who took my E0-256 course)
- Not restricted to conditional branches. Attack also adapted to work with indirect branches.

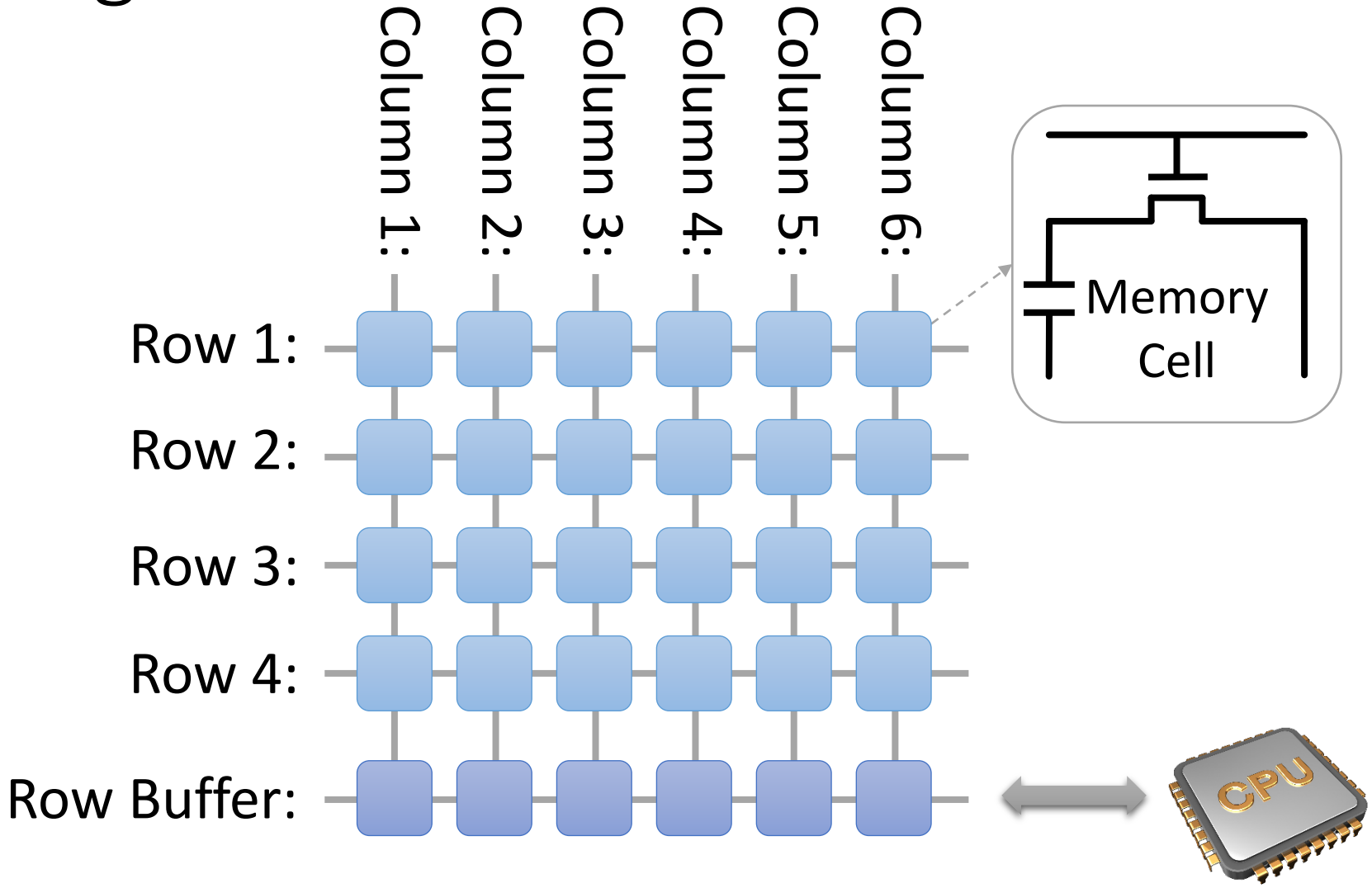
# Rowhammer

- An ***unspeakably cool*** security vulnerability…
- DRAM (unless you pay for error correcting (ECC) memory) is actually unreliable
  - Can repeatedly read/write the same location ("hammer the row" and eventually cause an error in ***some physically distinct memory location***
- Can tell the OS "I want to map this same block of memory at multiple addresses in my process…"
  - Which creates additional page table entries
- Enter ***Rowhammer***
  - It seems all vunerabilities get named now, but this one is cool enough to deserve a name!

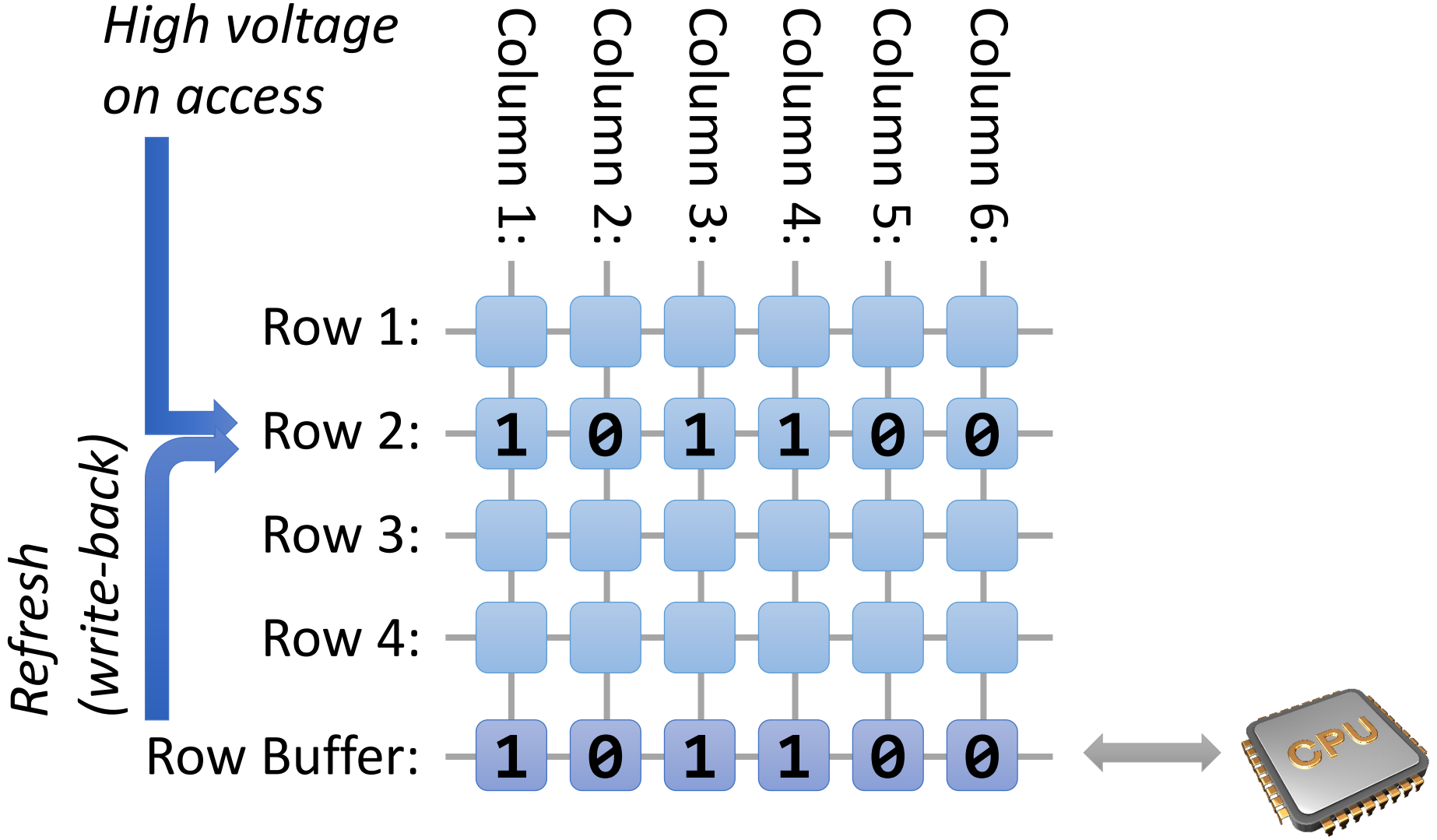
# Dynamic Random Access Memory (DRAM)



# DRAM Organization Inside a Bank



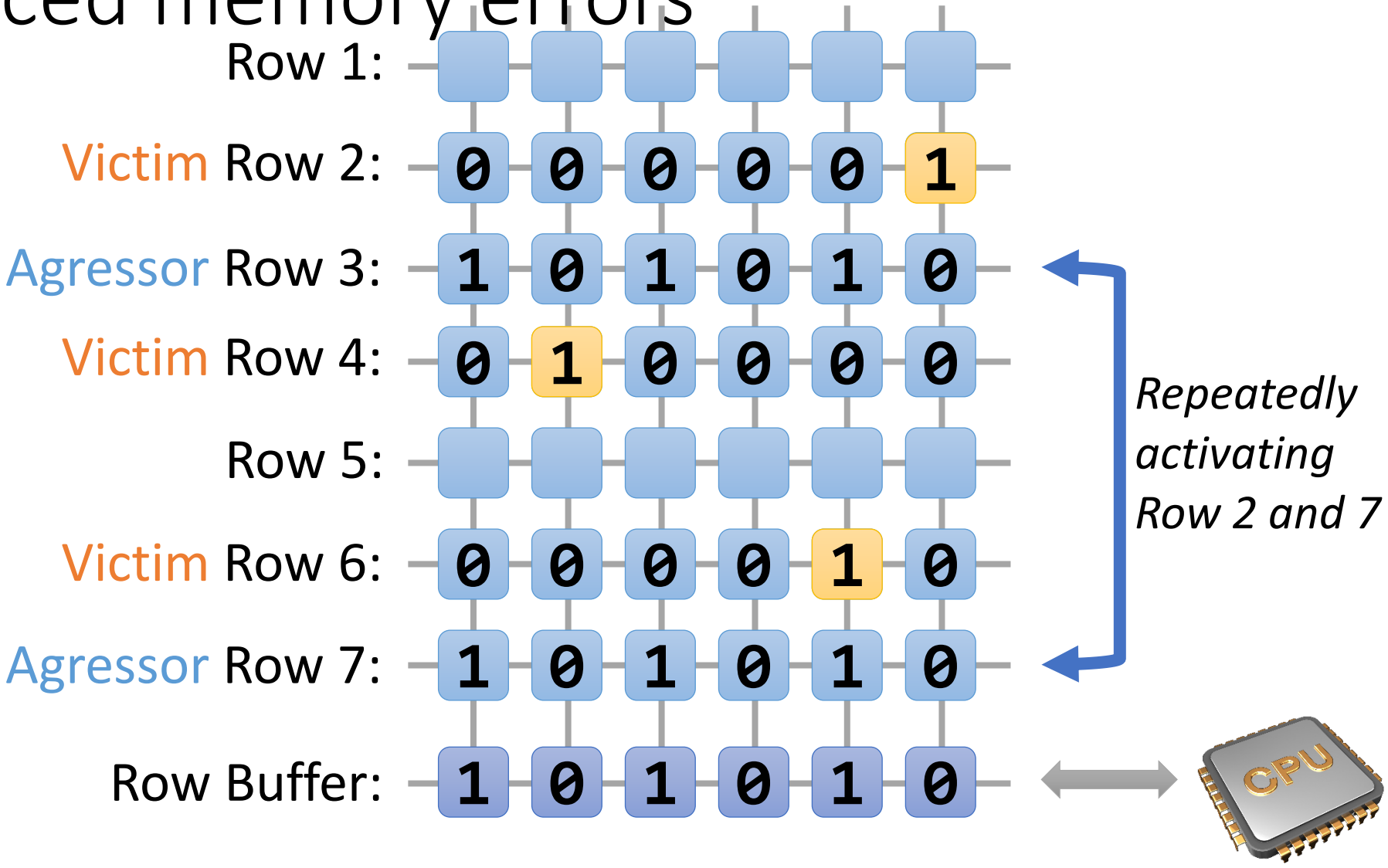
# DRAM: Read Access



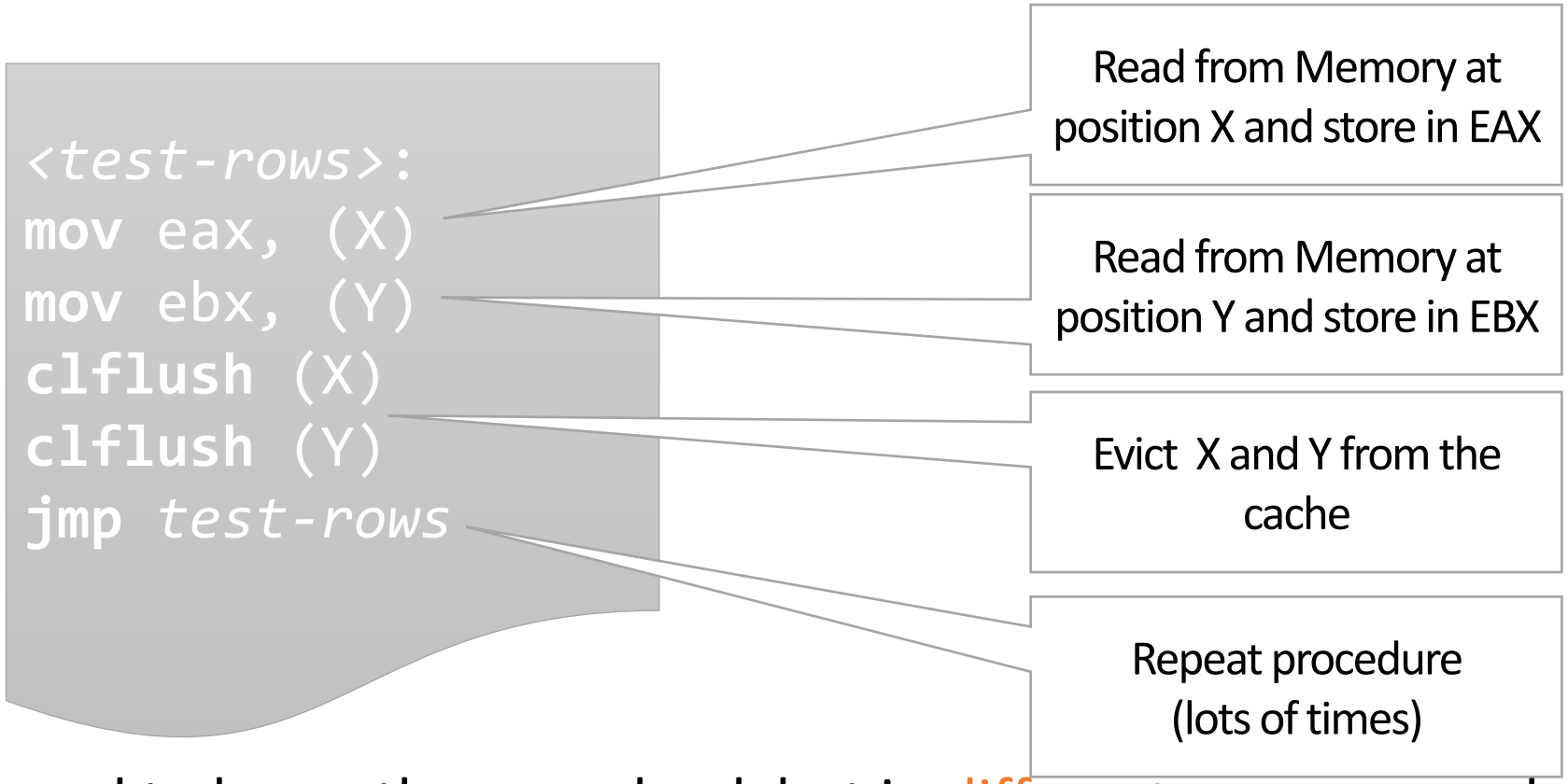
# DRAM unreliability

- Reading/rewriting bits leaks a bit of voltage to neighboring rows
- If done enough, can cause errors --- bit flips

# Induced memory errors



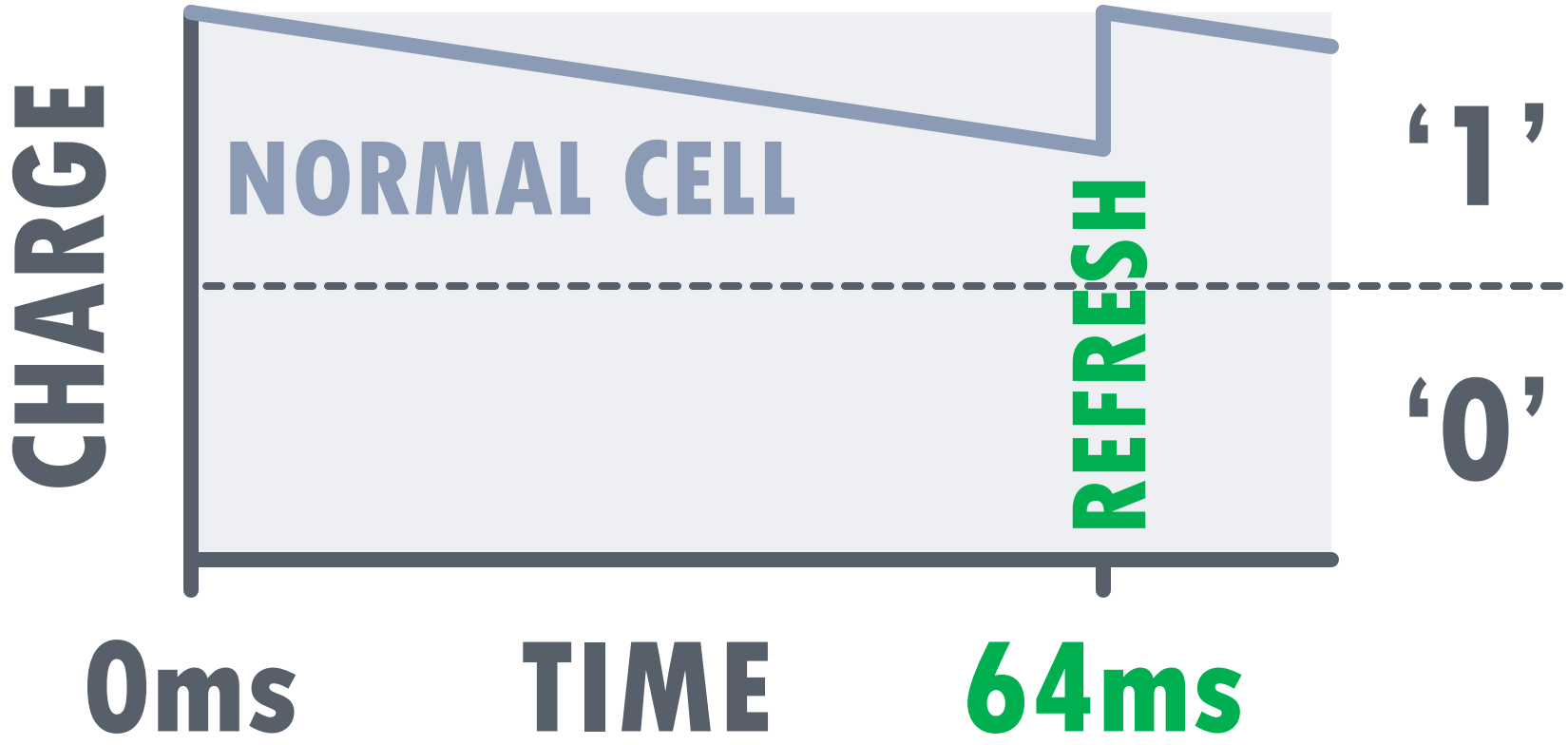
# Triggering code



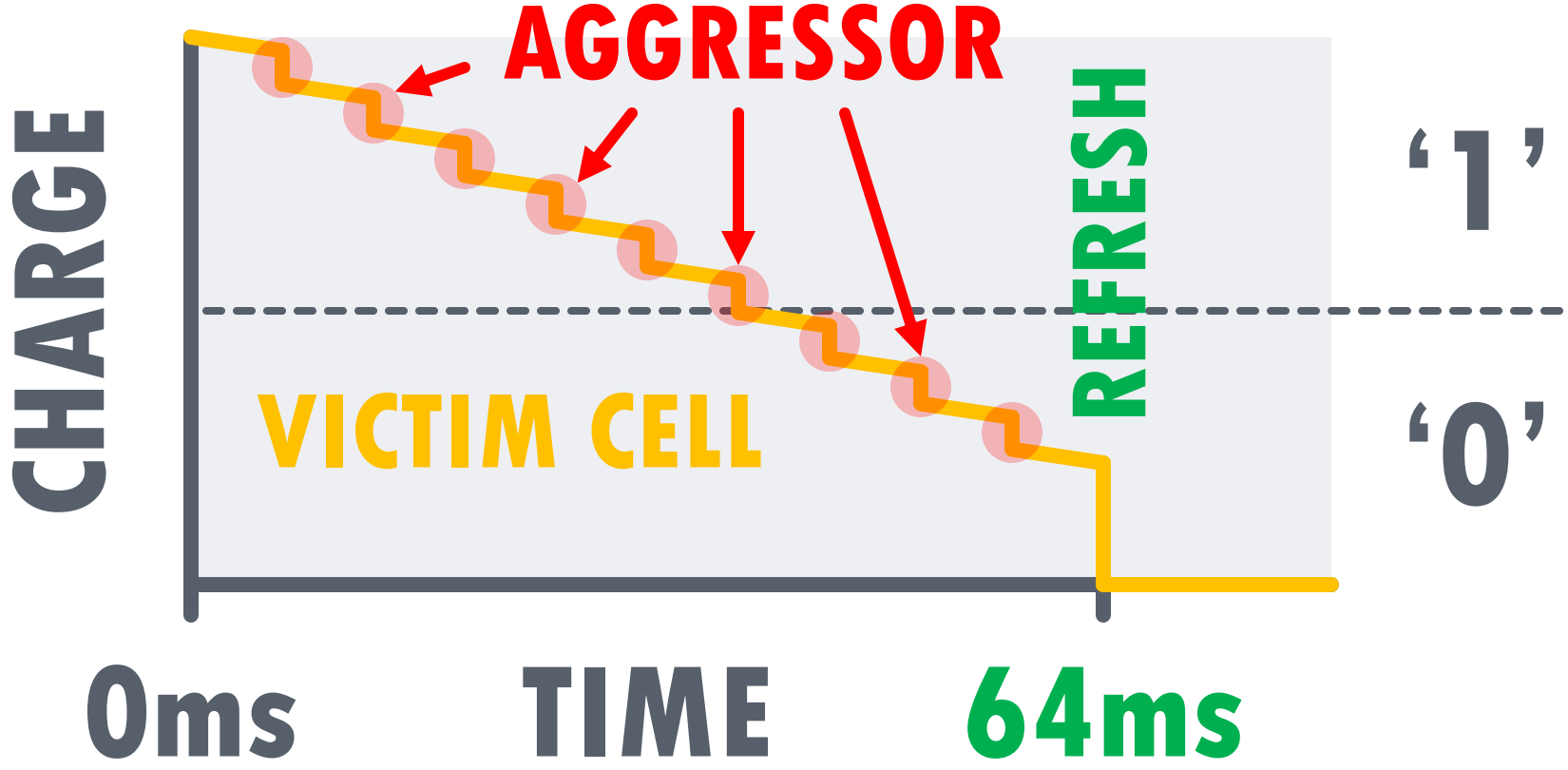

- X and Y need to be on the same bank but in different rows; general pattern: Y = X + 8MB

# WHY DO THE ERRORS OCCUR?

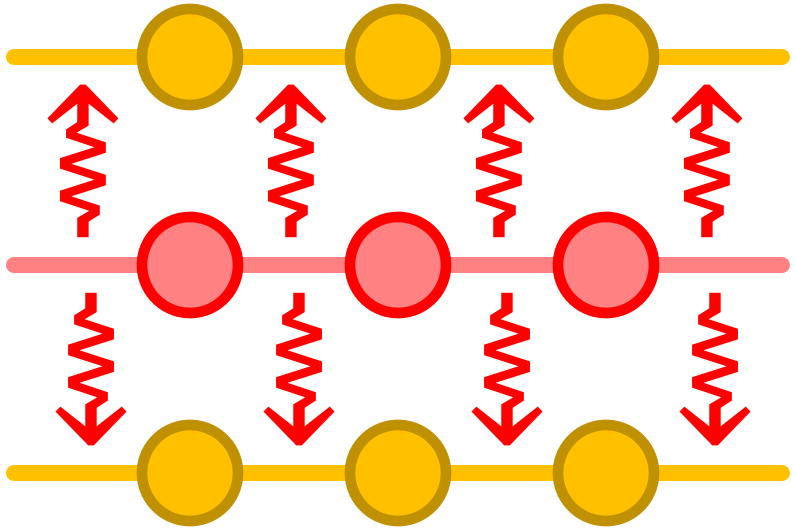
# DRAM CELLS ARE LEAKY



# DRAM CELLS ARE LEAKY



# ROOT CAUSE?



## COUPLING

- Electromagnetic
- Tunneling

## ACCELERATES CHARGE LOSS

# AS DRAM SCALES …

- **CELLS BECOME SMALLER**
  Less tolerance to coupling effects
- **CELLS BECOME PLACED CLOSER**
  Stronger coupling effects

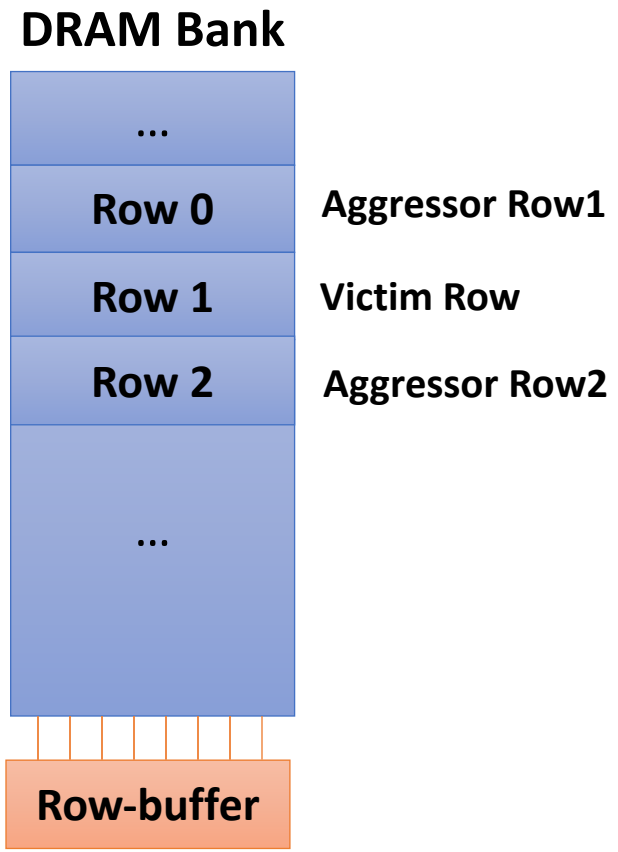
**COUPLING ERRORS MORE LIKELY**

# Increase strength: Double-Sided Rowhammer

[Seaborn+, Black Hat 2015]

**Repeat N times:**

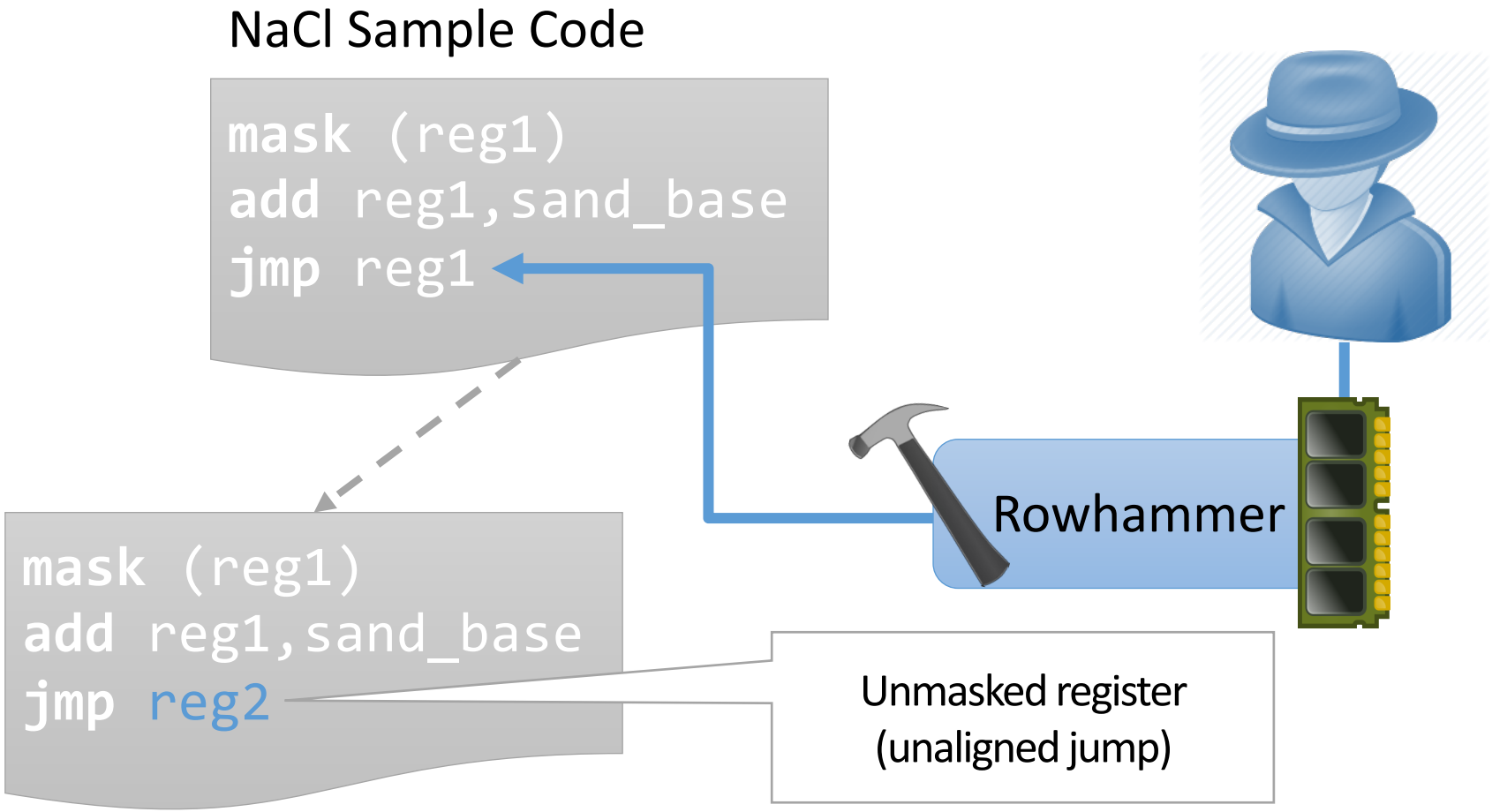
mov $A0_{(Row\ 0)}$, eax
mov $A1_{(Row\ 2)}$, eax
clflush $A0_{(Row\ 0)}$
clflush $A1_{(Row\ 2)}$




- *Victim row* lies between two aggressor rows
- All accesses have cross-talk, no need to flush row buffer

# Rowhammer Native Client (NaCl) Exploit

- Google‘s Native Client (NaCl) limits indirect jumps to target a 32-Byte aligned address inside the sandbox

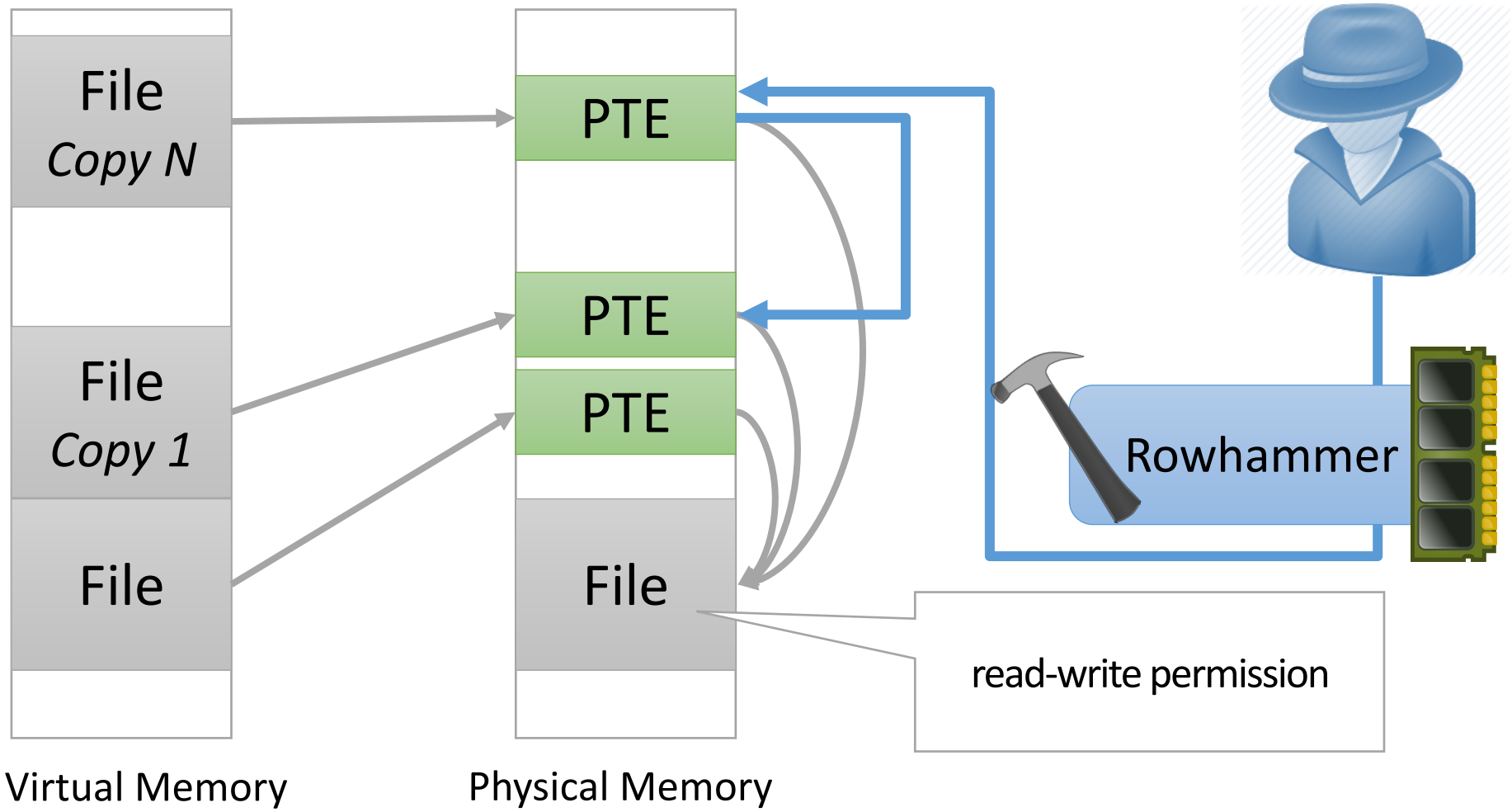


# Native Client attack

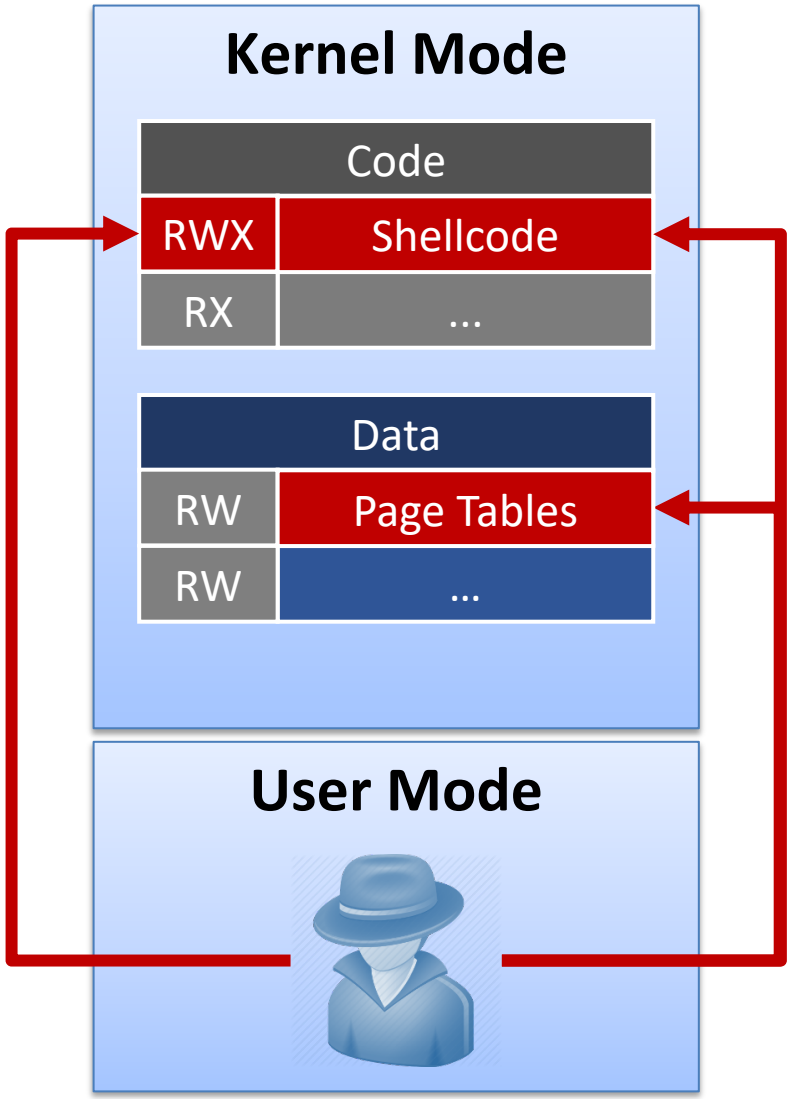
- Allocate lots of memory
- Fill with vulnerable instructions (load code)
- Apply Rowhammer
- Read memory to find desired bit flip
- Exploit flip

# Rowhammer Kernel Exploit

- Spray the memory with page table entries (PTEs)
- Launch rowhammer to corrupt a PTE



# Privilege Escalation with Rowhammer



# Extensions

- Throwhammer: exploit over RDMA network
- Nethammer: exploit over normal network
- GLitch: exploit from GPU code
- …

# Solutions

- Test DRAM for vulnerable rows, black-list
- Increase refresh rate
  - Hurts performance, takes power
- Add Error-correcting codes
  - Better, but can still be broken
- Conceal physical addresses
  - Turn off huge pages (hurts performance)
- Use vulnerable rows as cache, add checksum to detect errors

# Summary of Hardware Vulnerabilities

- Optimizations change timing of instructions
  - Timing leaks information
- Speculation changes microarchitectural (internal state)
  - Timing can reveal these changes
- Reliability problems become security problems if they can be intentionally triggered
  - OS, program security mechanisms assume memory is correct