



Project 1: **26** days left

# Offense-Based Cybersecurity: Exploitation of Hardware Vulnerabilities

CS 459/559: Science of Cyber Security  
7<sup>th</sup> Lecture

**Instructor:**

Guanhua Yan

# Agenda

- Quiz 1: September 29
- Project 1 (offense): October 10
- Project 2 (defense): December 5
- Presentations: 11/17, 11/19, 11/24, 12/1, 12/3
- Final report: December 15



# Report submission requirements

## ■ Project report

- At least five pages long, excluding references

## ■ PLEASE, NO PLAGIARISM

- Heavy penalty

## ■ Supplementary materials that you have done the real work

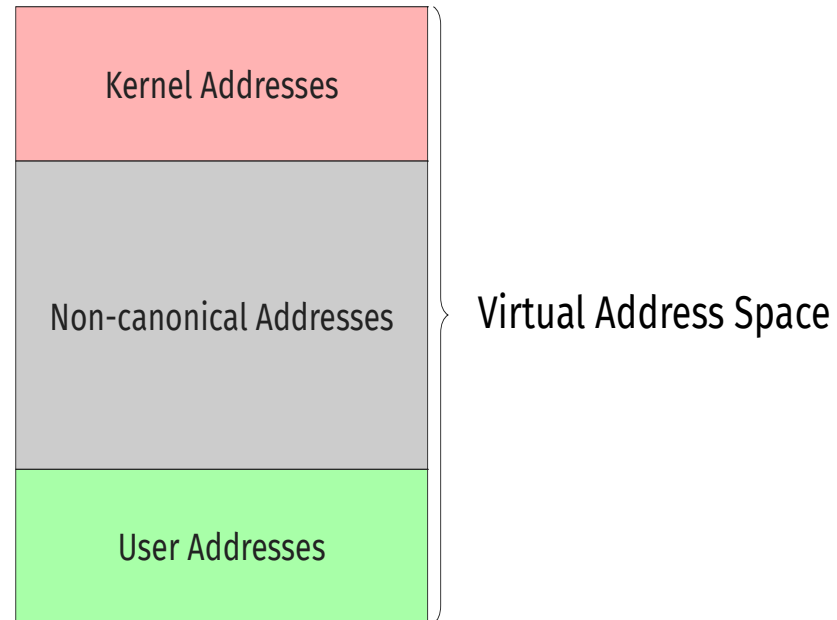
- Any code you have written
- Any data you have collected for your project
- The TA will use them to decide for grading (along with report):
  - Difficulty, novelty, presentation, and results

# What we will learn in this lecture?

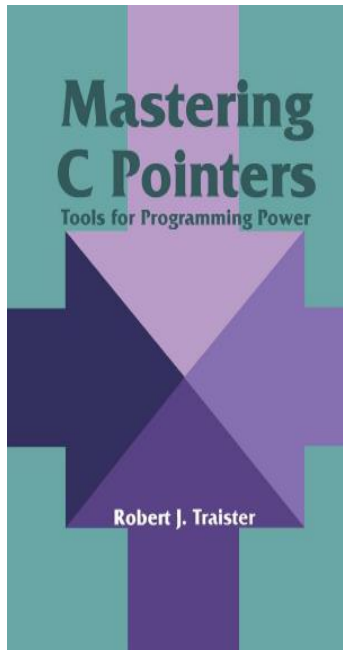
- Basics of computer architecture
- Side channel attacks based on caches
- Meltdown & Spectre

# *Basics of computer architecture*

# Virtual Memory



# Building the Code



```
char data = *(char*) 0xffffffff81a000e0;  
printf("%c\n", data);
```

# Building the Code

```
char data = *(char*) 0xffffffff81a000e0;  
printf("%c\n", data);
```

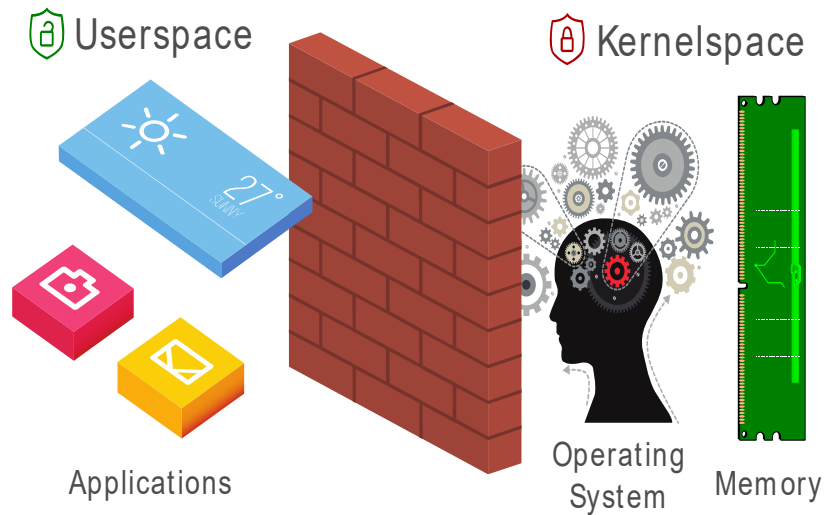


- Compile and run

```
segfault at ffffffff81a000e0 ip 0000000000400535  
sp 00007ffce4a80610 error 5 in reader
```

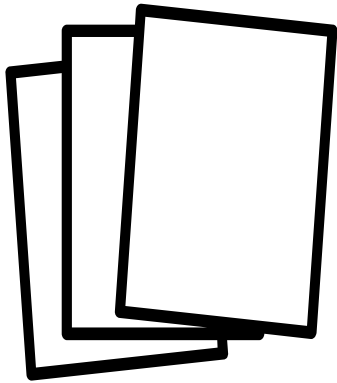
- Kernel addresses are of course **not accessible**
- Any invalid access throws an exception ! **segmentation fault**

# Memory Isolation



- Kernel is isolated from user space
- This **isolation** is a combination of hardware and software
- User applications cannot access anything from the kernel

# Paging

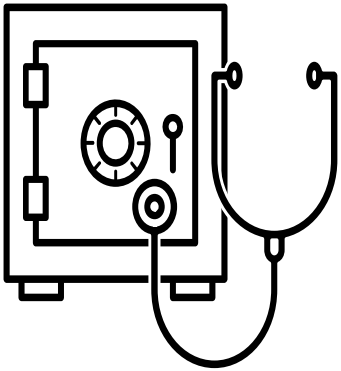


- CPU support **virtual address spaces** to isolate processes
- Physical memory is organized in **page frames**
- Virtual memory pages are **mapped** to page frames **using page tables**

## *Side channel attacks based on caches*

# Side-channel Attacks

- Safe software infrastructure does not mean safe execution
- Information leaks because of the **underlying hardware**
- Exploit **unintentional information leakage by side-effects**



Power  
consumption



Execution  
time

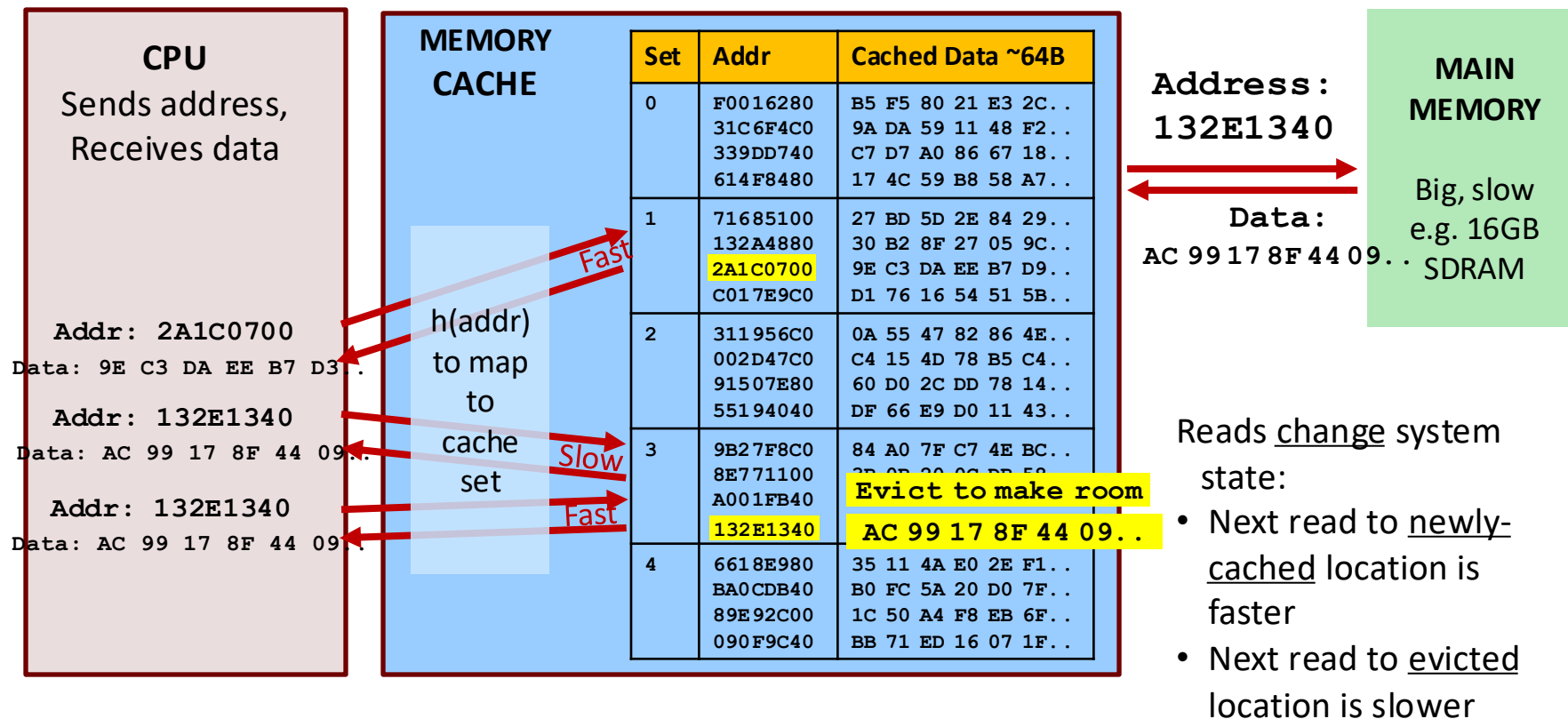


CPU caches



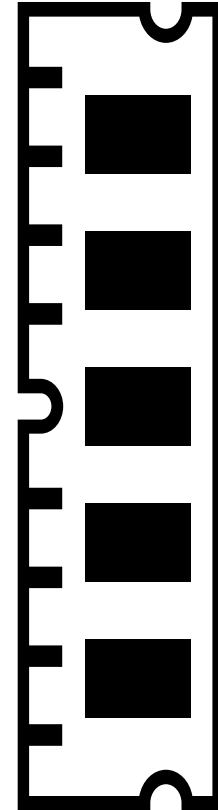
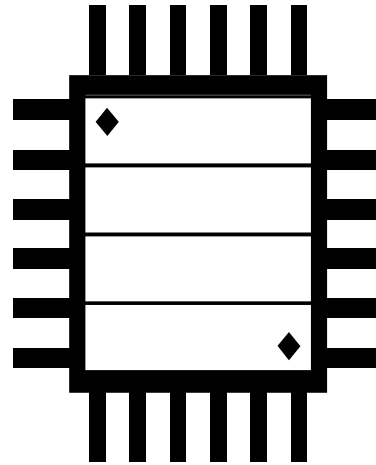
# Memory caches for dummies

- Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



# CPU Cache

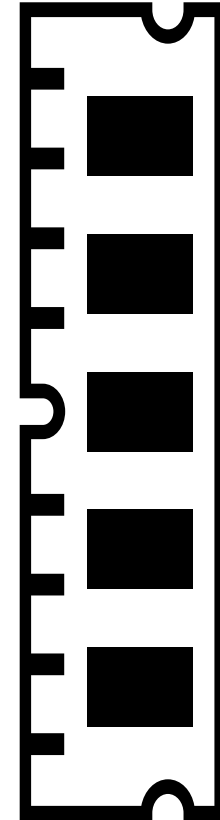
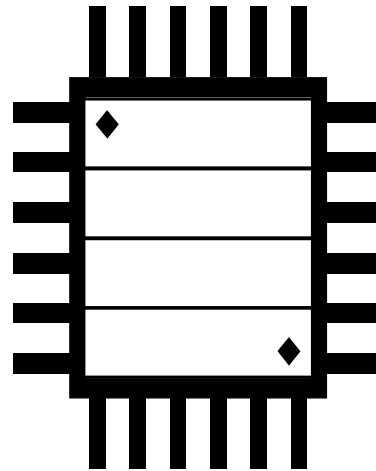
```
printf("%d", i);  
printf("%d", i);
```



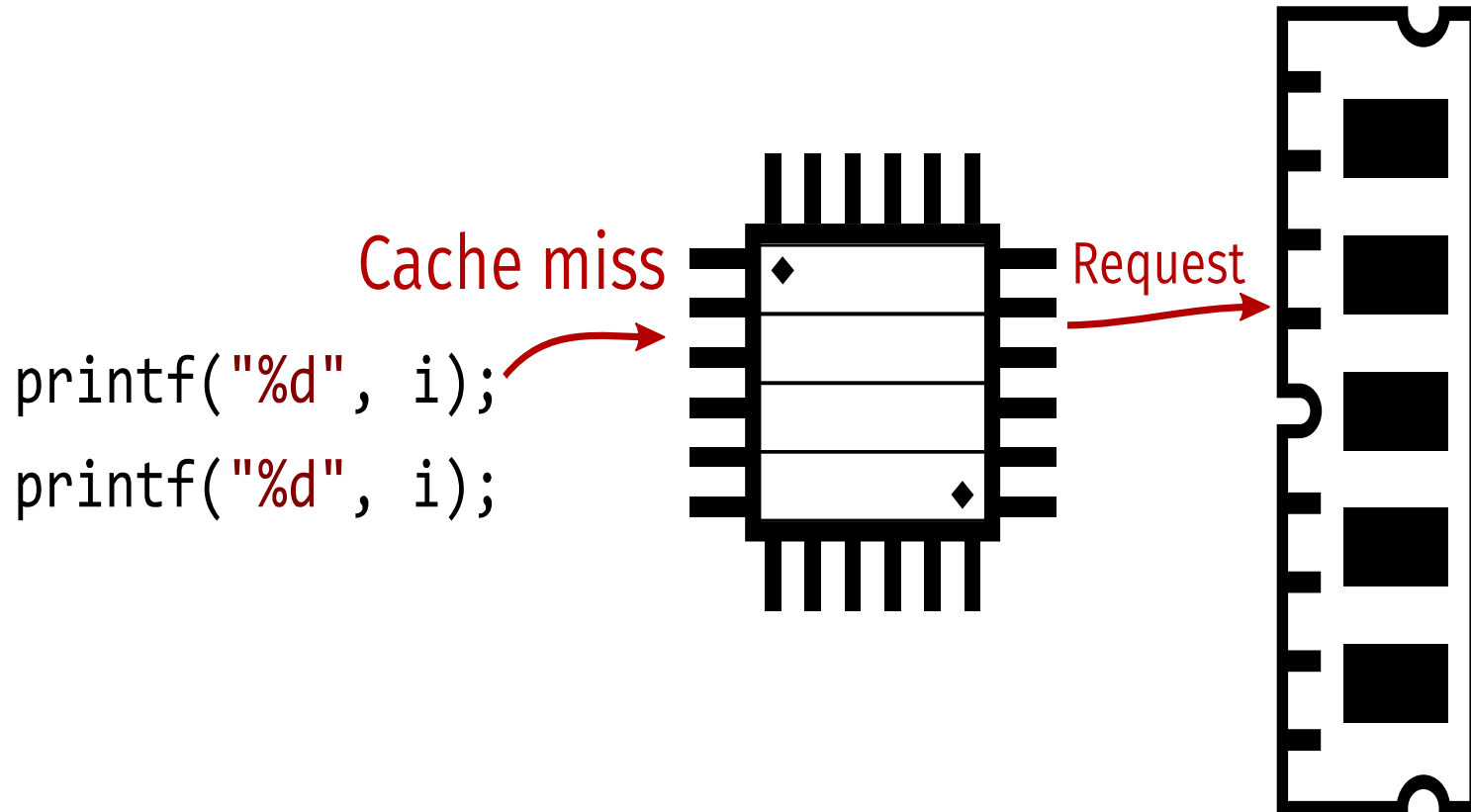
# CPU Cache

```
printf("%d", i);  
printf("%d", i);
```

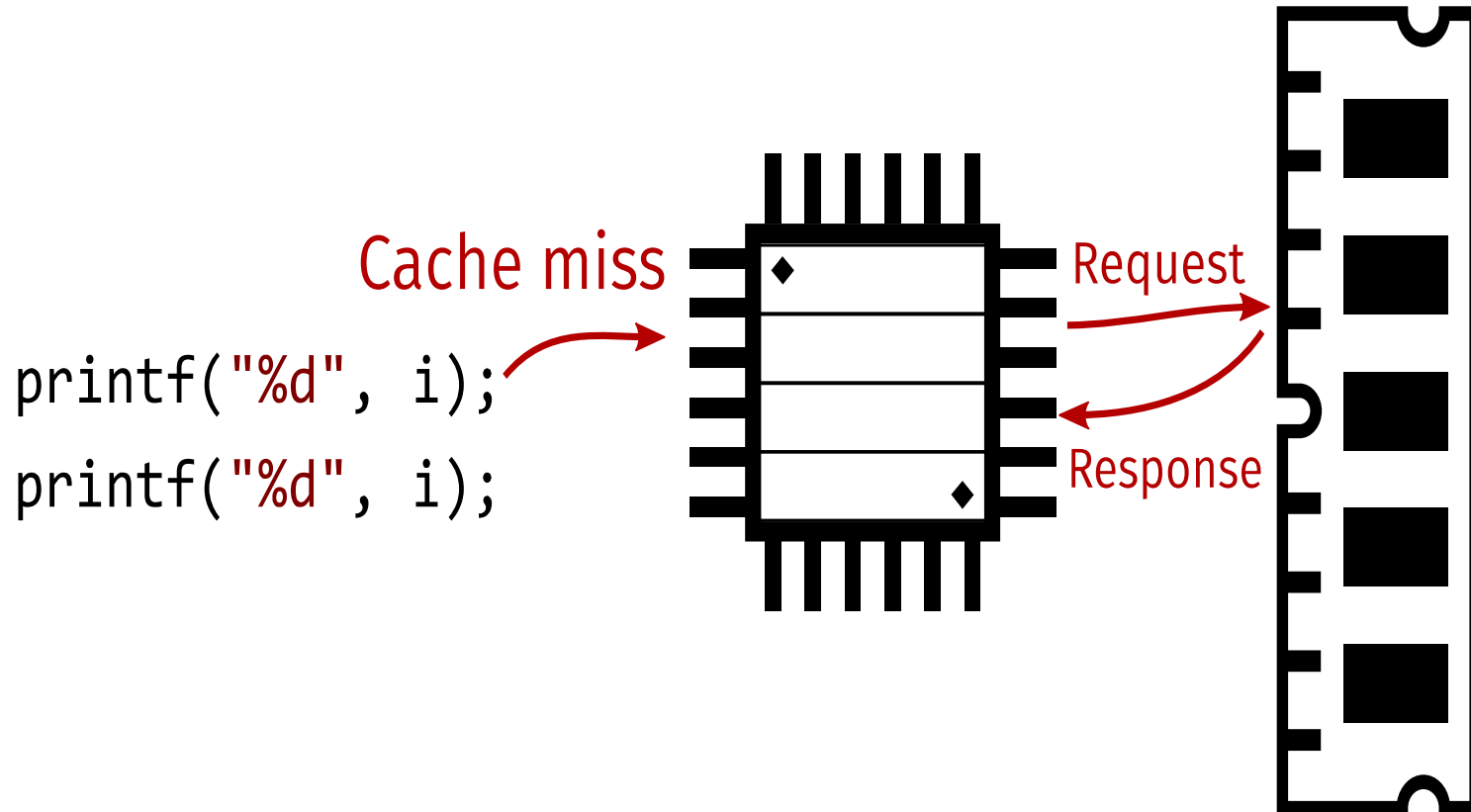
Cache miss



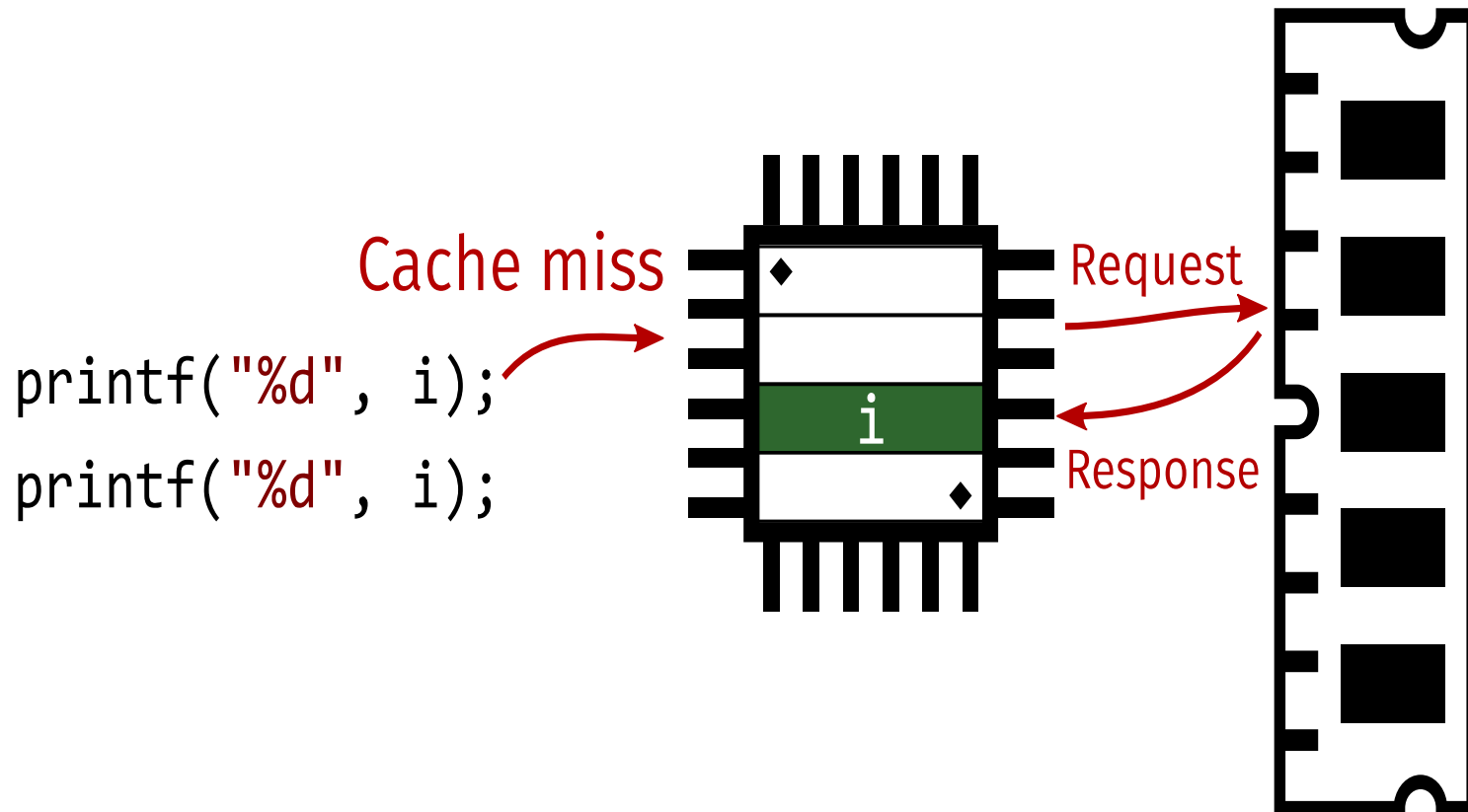
# CPU Cache



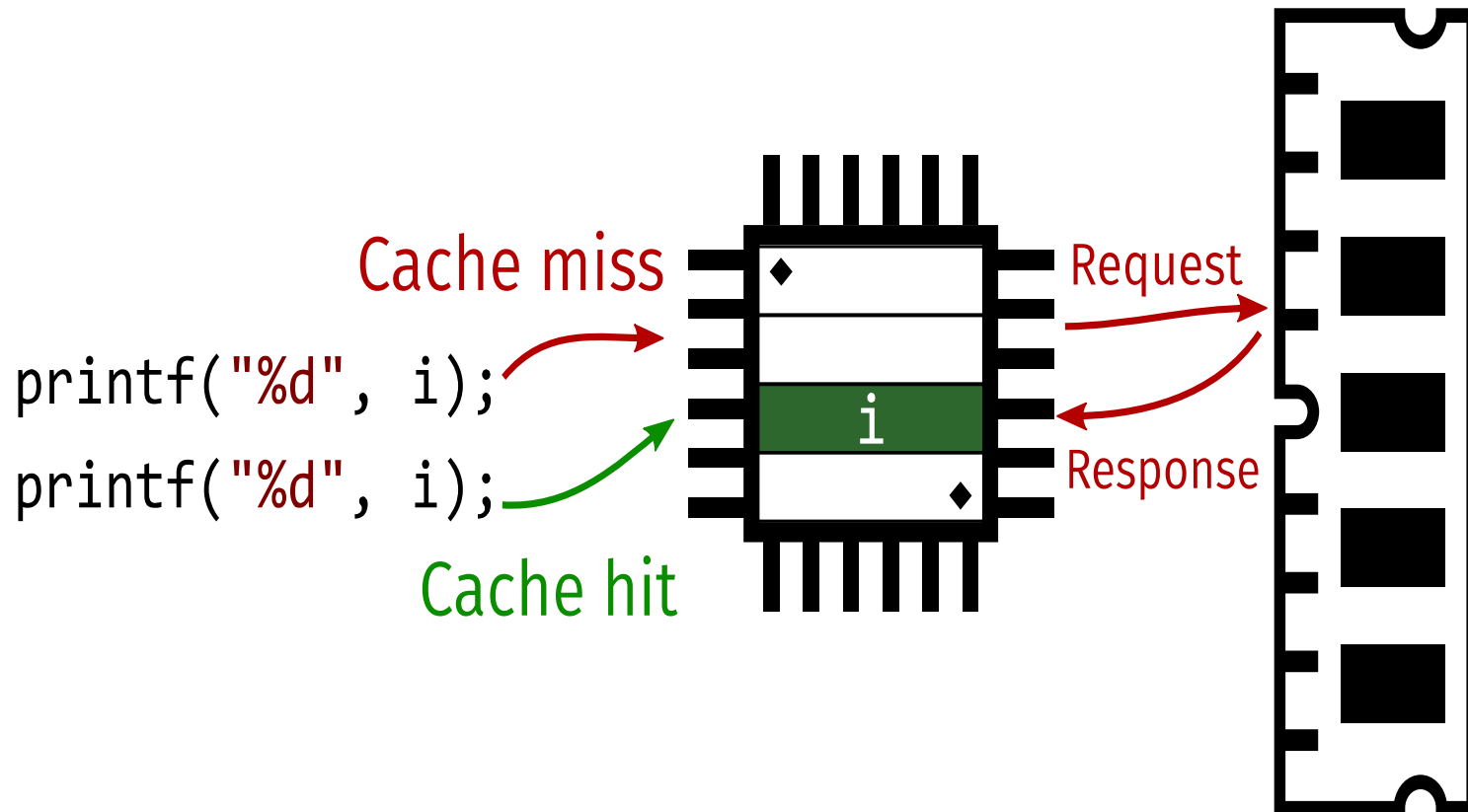
# CPU Cache



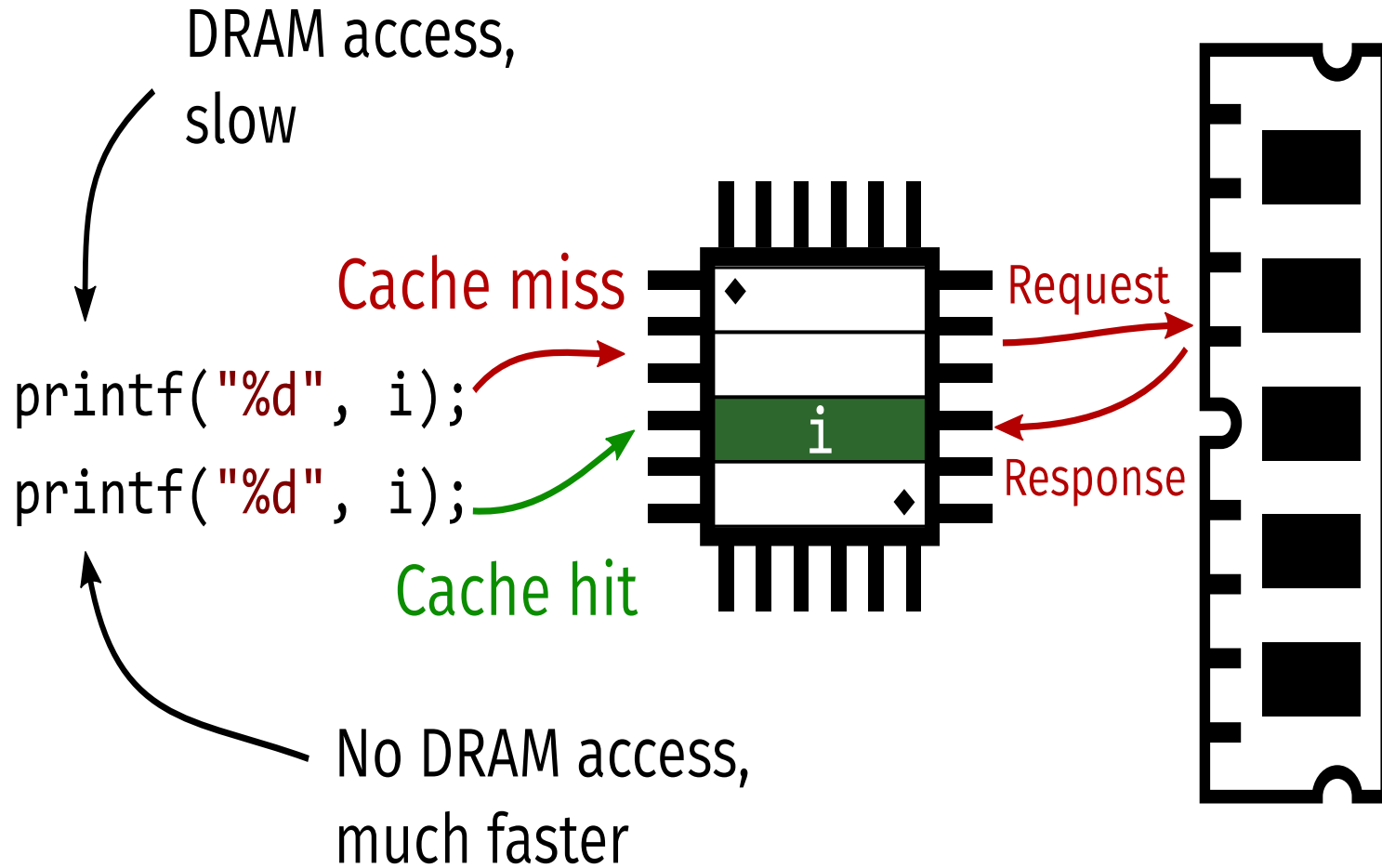
# CPU Cache



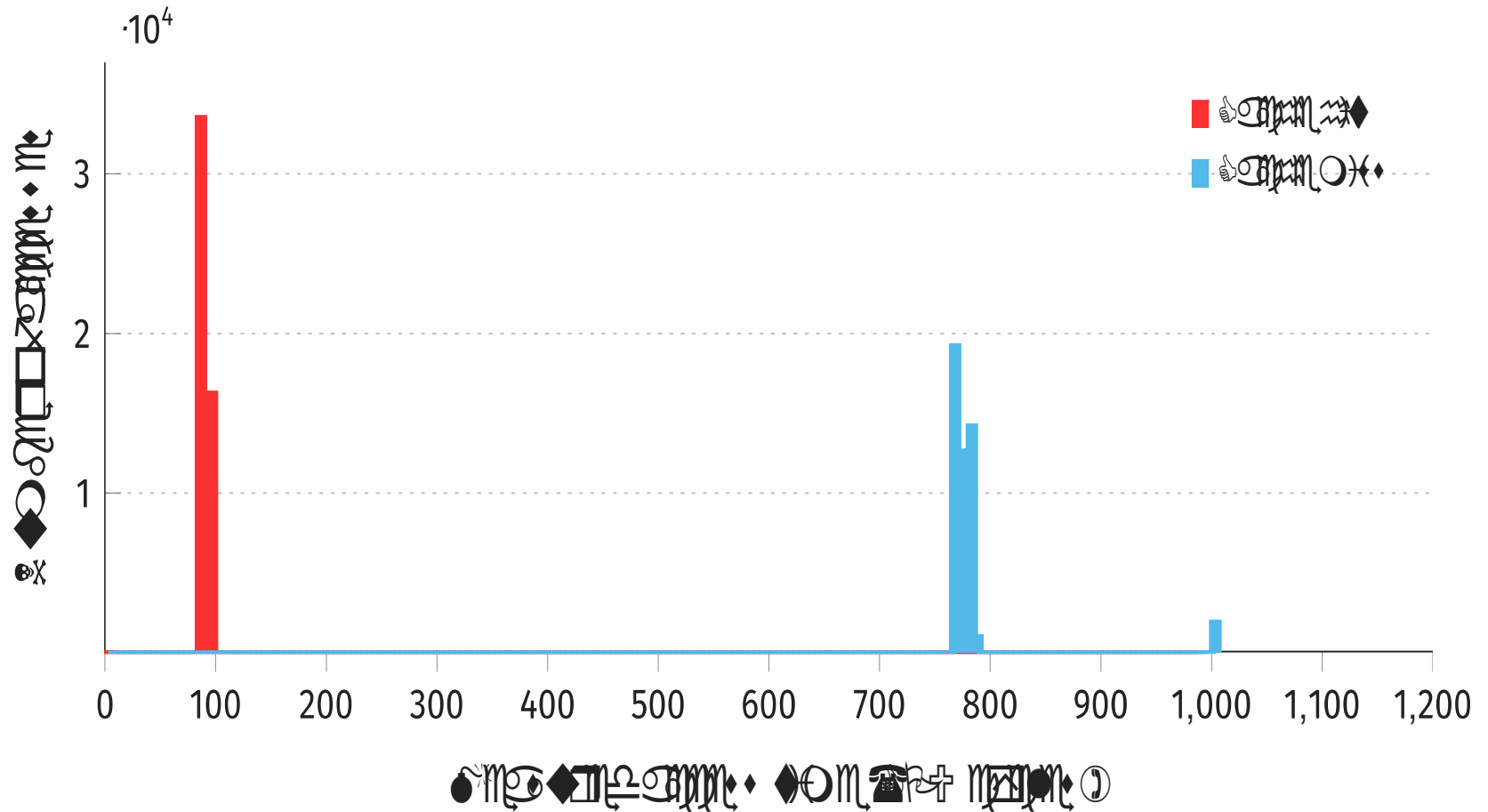
# CPU Cache



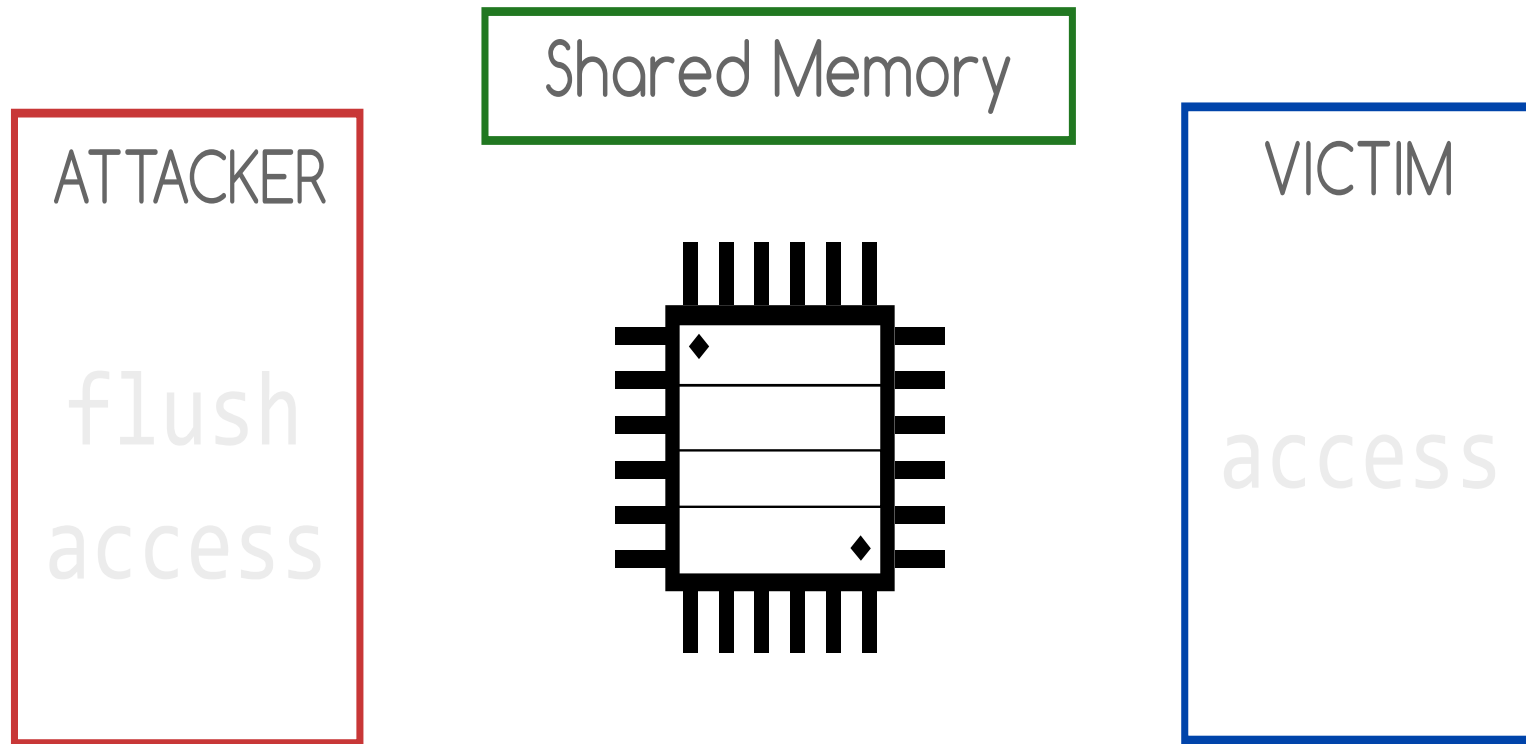
# CPU Cache



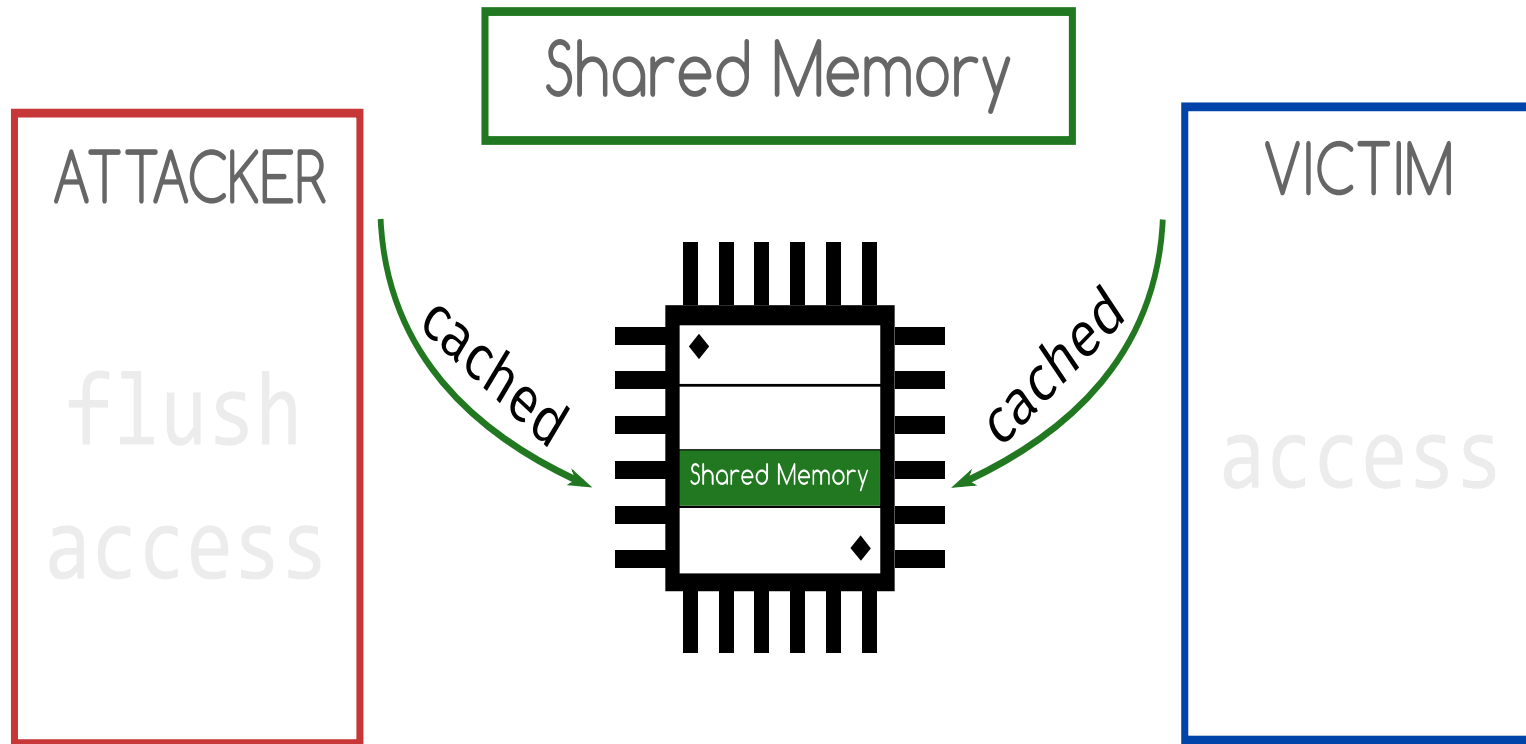
# Memory Access Latency

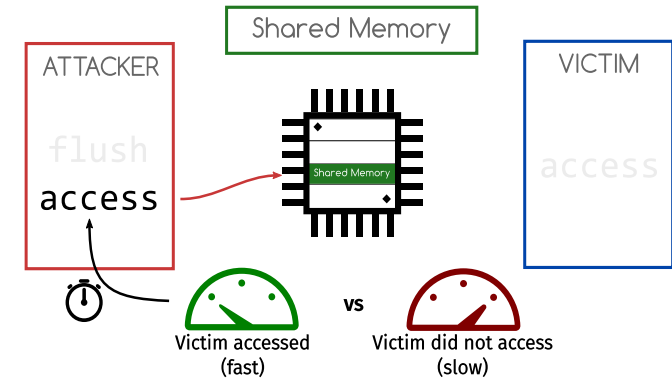
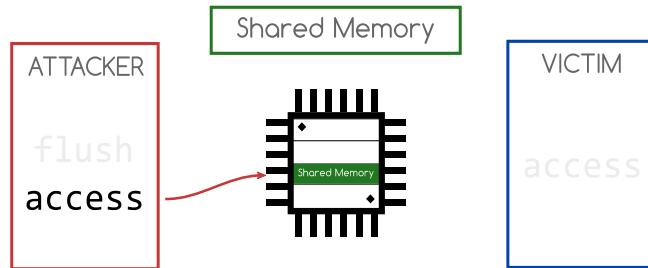
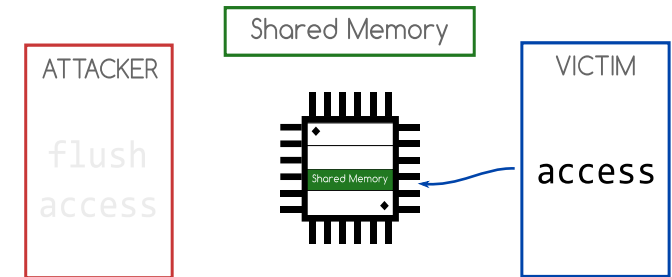
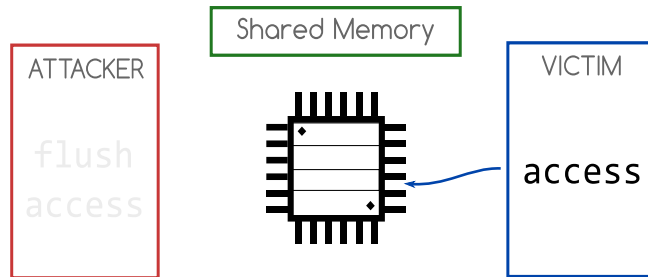
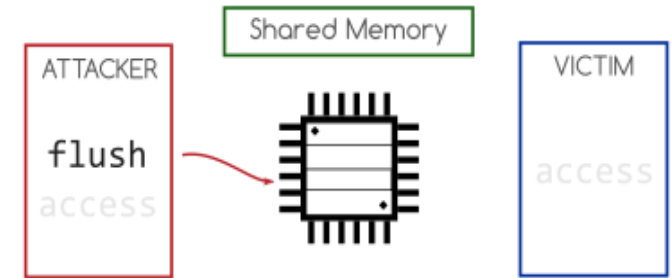
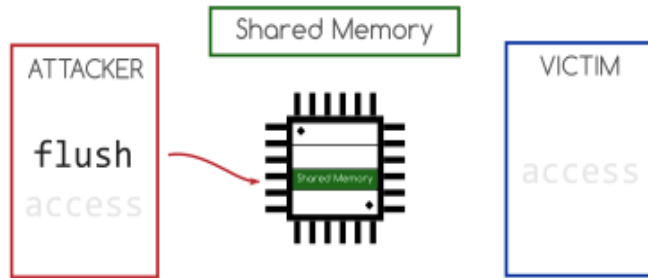


# Flush+Reload



# Flush+Reload

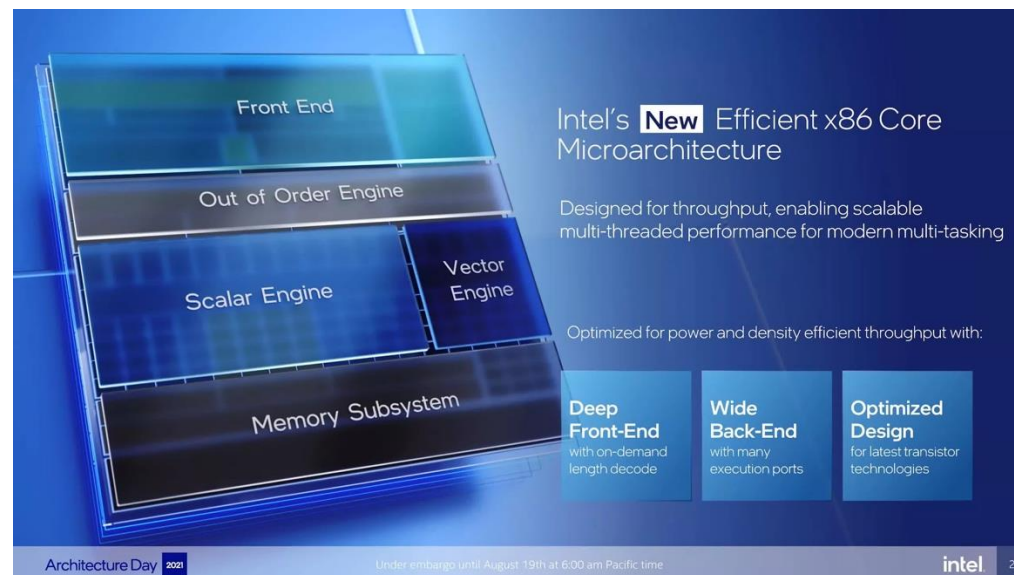




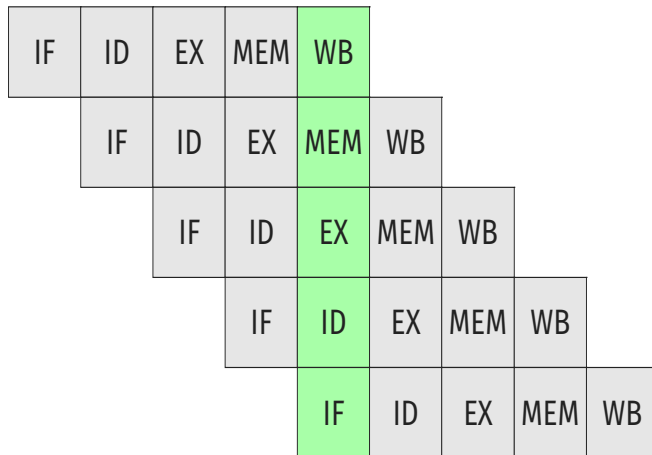
# *Out of order execution*

# Architecture and Microarchitecture

- Instruction Set Architecture (ISA) is an **abstract model** of a computer (x86, ARMv8, SPARC, ...)
- Serves as the **interface** between hardware and software
- Microarchitecture is an **actual implementation** of the ISA



# In-Order Execution



- Instructions are...
  - **fetch**ed (IF) from the L1 Instruction Cache
  - **dec**oded (ID)
  - **exec**uted (EX) by execution units
- Memory **access** is performed (MEM)
- Architectural **register file** is **update**d (WB)

# In-Order Execution



- Instructions are executed **in-order**
- Pipeline **stalls** when stages are not ready
- If data is **not cached**, we need to wait

# Out-of-order Execution

```
int width = 10, height = 5;

float diagonal = sqrt(width * width
                      + height * height);
int area = width * height;

printf("Area %d x %d = %d\n", width, height, area);
```

# Out-of-order Execution

Dependency



```
int width = 10, height = 5;
```

```
float diagonal = sqrt(width * width  
                      + height * height);
```

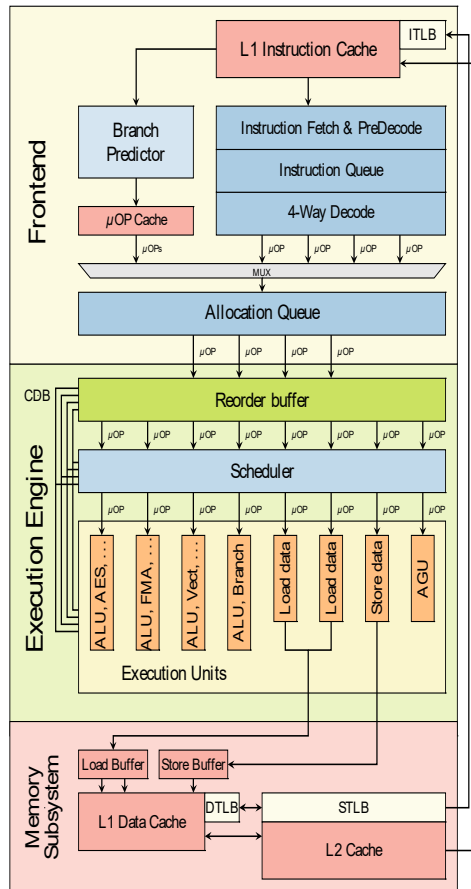
```
int area = width * height;
```

```
printf("Area %d x %d = %d\n", width, height, area);
```

Parallelize



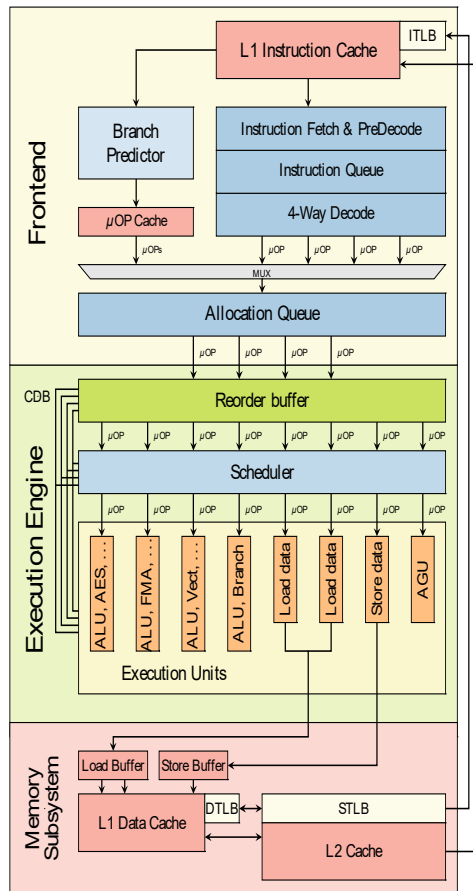
# Out-of-Order Execution



Instructions are

- fetched and decoded in the **front-end**
- dispatched to the **backend**
- processed by **individual execution units**

# Out-of-Order Execution



## Instructions

- are executed **out-of-order**
- wait until their **dependencies are ready**
  - Later instructions might execute prior earlier instructions
- **retire in-order**
  - State becomes architecturally visible
- **Exceptions** are checked during retirement
  - Flush pipeline and recover state

An instruction ([microoperation](#),  $\mu$ op) leaving the "retirement unit" means that in the [out-of-order](#) CPU pipeline the instruction is finally executed and its results are correct and visible in the [architectural state](#) as if they execute in-order.

# *Meltdown*

# Meltdown – the 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 **KAISER/KPTI**

# Toy example (data is a secret)

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

Listing 1: A toy example to illustrate side-effects of out-of-order execution.

# Out of order execution

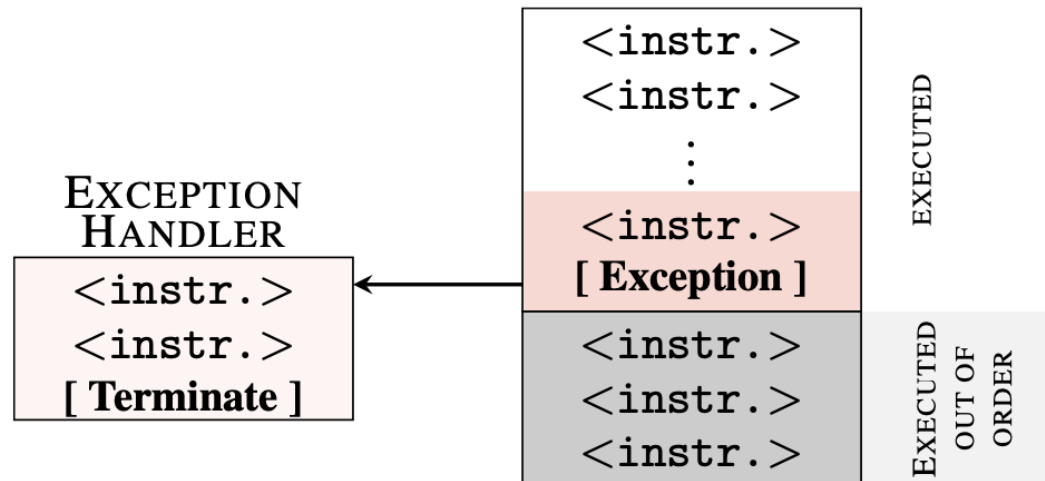


Figure 3: If an executed instruction causes an exception, diverting the control flow to an exception handler, the subsequent instruction must not be executed. Due to out-of-order execution, the subsequent instructions may already have been partially executed, but not retired. However, architectural effects of the execution are discarded.

# However

- The instructions executed out of order do **not** have any visible **architectural effect** on registers or memory
- But they have **microarchitectural side effects**. During the out-of-order execution, **the referenced memory is fetched into a register and also stored in the cache.**

# Array access time (secret is 84)

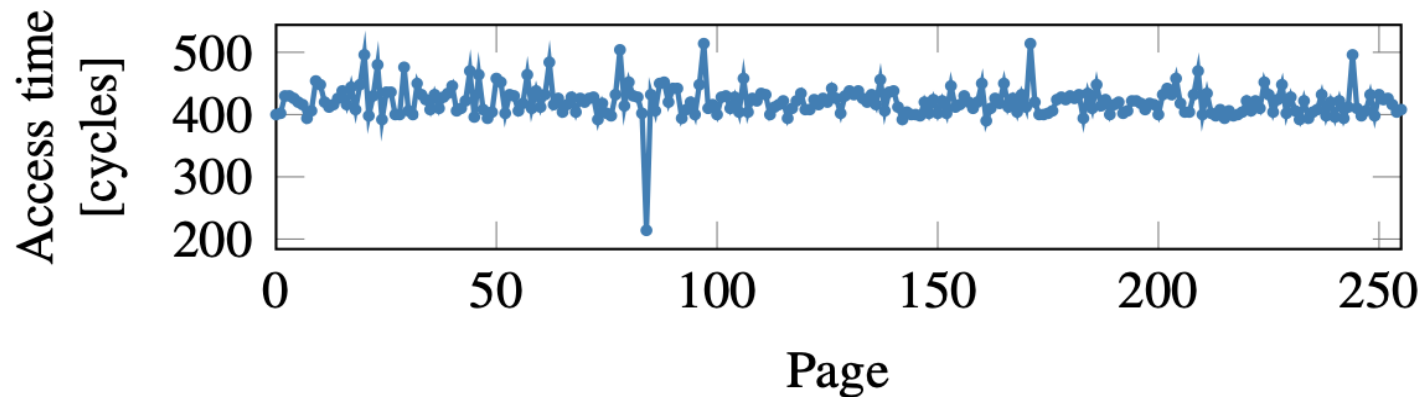


Figure 4: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of `probe_array` shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

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

# Meltdown (single bit example)

- A malicious attacker arranges for exploit code similar to the following to speculatively execute:


```
if (spec_cond) {  
    unsigned char value = *(unsigned char *)ptr;  
    unsigned long index2 = (((value>>bit)&1)*0x100)+0x200;  
    maccess(&data[index2]);  
}
```

- “data” is a user controlled array to which the attacker has access, “ptr” contains privileged data

# Meltdown (continued)

- A malicious attacker arranges for exploit code similar to the following to speculatively execute:

```
if (spec_cond) {  
    unsigned char value = *(unsigned char *)ptr;  
    unsigned long index2 = (((value > bit) & 1) * 0x100) + 0x200;  
    maccess(&data[index2]);  
}
```



load a pointer to  
which we don't have access (exception)

# Meltdown (continued)

- A malicious attacker arranges for exploit code similar to the following to speculatively execute:

```
if (spec_cond) {  
    unsigned char value = *(unsigned char *)ptr;  
    unsigned long index2 = (((value>>bit)&1)*0x100)+0x200;  
    maccess(&data[index2]);  
}
```



Out-of-order execution

bit shift extracts  
a single bit of data

# Meltdown (continued)

- A malicious attacker arranges for exploit code similar to the following to speculatively execute:

```
if (spec_cond) {  
    unsigned char value = *(unsigned char *)ptr;  
    unsigned long index2 = (((value>>bit)&1)*0x100)+0x200;  
    maccess(&data[index2]);  
}
```



generate address  
from data value

# Meltdown (continued)

- A malicious attacker arranges for exploit code similar to the following to speculatively execute:

```
if (spec_cond) {  
    unsigned char value = *(unsigned char *)ptr;  
    unsigned long index2 = (((value>>bit)&1)*0x100)+0x200;  
    maccess(&data[index2]);  
}
```



use address as offset to  
pull in cache line  
that we control

# Meltdown (continued)

```
char value = *SECRET_KERNEL_PTR;
```



mask out bit I want to read



calculate offset in "data"  
(that I do have access to)

```
char data[];
```

0x000	
0x100	
0x200	
0x300	



# Meltdown (continued)

- Access to “data” element 0x100 pulls the corresponding entry into the cache

char data[];

0x000	
0x100	
0x200	
0x300	DATA



Cache



# Meltdown (continued)

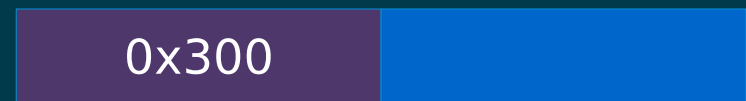
- Access to “data” element 0x300 pulls the corresponding entry into the cache

char data[];

0x000	
0x100	
0x200	
0x300	DATA



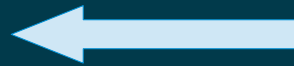
Cache



# Meltdown (continued)

- We use the cache as a side channel to determine which element of “data” is in the cache
  - Access both elements and time the difference in access (we previously flushed them)

```
time = rdtsc();  
maccess(&data[0x300]);  
delta3 = rdtsc() - time;
```



Execution time taken for instruction is proportional to whether it is in cache(s)

```
time = rdtsc();  
maccess(&data[0x100]);  
delta2 = rdtsc() - time;
```

# *Spectre*

# Speculative Execution

- Modern processors perform speculative execution
- They execute instructions in parallel that are likely to be executed after a branch in code (e.g. if/else)
- Of course these instructions may never really be executed, so a sort of CPU snapshot is taken at the branch so execution can be “rolled back” if needed

# Branch Prediction

- How does the CPU know which side of a branch (“if/else”) to speculatively execute?
- Branch prediction algorithms are trained based on current execution
- The CPU “learns” which branch will be executed from previous executions of the same code

# Speculative execution

Instead of idling, CPUs can *guess* likely program path and do speculative execution

‣ Example:

```
if (uncached_value_usually_1 == 1)
    foo()
```

- Branch predictor: if() will probably be 'true' (based on prior history)
- CPU starts foo() speculatively -- but doesn't commit changes
- When value arrives from memory, if() can be evaluated definitively -- check if guess was correct:
  - Correct: Commit speculative work – performance gain
  - Incorrect: Discard speculative work

# Attacker's goal

Violates software security requirement that the CPU runs instructions correctly.

## Regular execution

Set up the conditions so the processor will make a desired mistake

Fetch the sensitive data from the covert channel

## Erroneous speculative execution

Mistake leaks sensitive data into a covert channel (e.g. state of the cache)



# Conditional branch attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

**Assume code in kernel API, where unsigned int  $x$  comes from untrusted caller**

**Execution without speculation is safe**

- CPU will not evaluate `array2[array1[x]*4096]` unless `x < array1_size`

**What about with speculative execution?**

# Conditional branch attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Before attack:

- Train branch predictor to expect if() is true (e.g. call with `x < array1_size`)
- Evict `array1_size` and `array2[]` from cache

## Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base address:

8 bytes of data (value doesn't matter)

[... lots of memory up to `array1` base+N...]

**09** F1 98 CC 90... (something secret)

```
array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
```

Contents don't  
matter

only care about cache  
**status**

Uncached

Cached

# Conditional branch attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with  $x=N$  (where  $N > 8$ )

- Speculative exec while waiting for `array1_size`
  - Predict that `if()` is true
  - Read address (`array1 base + x`) w/ out-of-bounds  $x$
  - Read returns secret byte = **09**
  - Request memory at (`array2 base + 09*4096`)
  - Brings `array2[09*4096]` into the cache
  - Realize `if()` is false: discard speculative work
- Finish operation & return to caller

Attacker measures read time for `array2[i*4096]`

- Read for  $i=09$  is fast (cached), revealing secret byte
- Repeat with many  $x$  (eg  $\sim 10\text{KB/s}$ )

## Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base address:

8 bytes of data (value doesn't matter)

[... lots of memory up to `array1 base+N...`]

**09** F1 98 CC 90... (something secret)

array2[ 0\*4096]  
array2[ 1\*4096]  
array2[ 2\*4096]  
array2[ 3\*4096]  
array2[ 4\*4096]  
array2[ 5\*4096]  
array2[ 6\*4096]  
array2[ 7\*4096]  
array2[ 8\*4096]  
**array2[ 9\*4096]**  
array2[10\*4096]  
array2[11\*4096]

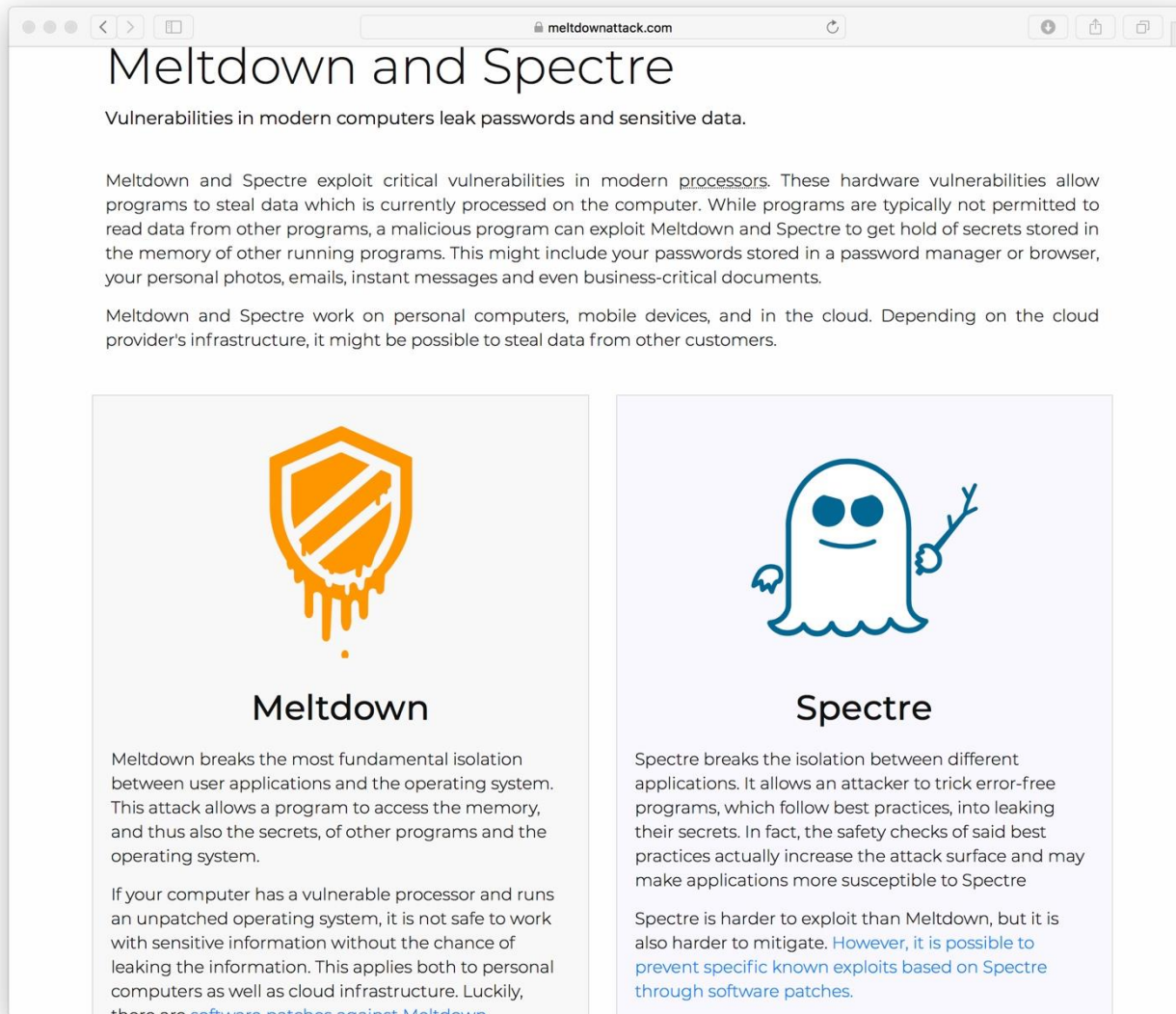
Contents don't  
matter

only care about cache  
**status**

Uncached

Cached

# Meltdown and Spectre

A screenshot of a web browser displaying the website meltdownattack.com. The browser's address bar shows the URL. The page has a white background with black text. At the top, the title 'Meltdown and Spectre' is followed by a subtitle 'Vulnerabilities in modern computers leak passwords and sensitive data.' Below this, there are three paragraphs of text explaining the nature of the vulnerabilities. The page is divided into two columns. The left column features an orange shield icon with a diagonal crack and the title 'Meltdown' in bold. Below the title is a paragraph explaining that Meltdown breaks the isolation between user applications and the operating system, and another paragraph mentioning that there are software patches against Meltdown. The right column features a blue ghost icon holding a branch and the title 'Spectre' in bold. Below the title is a paragraph explaining that Spectre breaks the isolation between different applications, and another paragraph mentioning that Spectre is harder to exploit and mitigate, but that it is possible to prevent specific known exploits based on Spectre through software patches.


meltdownattack.com

## Meltdown and Spectre

Vulnerabilities in modern computers leak passwords and sensitive data.

Meltdown and Spectre exploit critical vulnerabilities in modern processors. These hardware vulnerabilities allow programs to steal data which is currently processed on the computer. While programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Spectre to get hold of secrets stored in the memory of other running programs. This might include your passwords stored in a password manager or browser, your personal photos, emails, instant messages and even business-critical documents.


Meltdown and Spectre work on personal computers, mobile devices, and in the cloud. Depending on the cloud provider's infrastructure, it might be possible to steal data from other customers.



### Meltdown

Meltdown breaks the most fundamental isolation between user applications and the operating system. This attack allows a program to access the memory, and thus also the secrets, of other programs and the operating system.

If your computer has a vulnerable processor and runs an unpatched operating system, it is not safe to work with sensitive information without the chance of leaking the information. This applies both to personal computers as well as cloud infrastructure. Luckily, there are [software patches against Meltdown](#).



### Spectre

Spectre breaks the isolation between different applications. It allows an attacker to trick error-free programs, which follow best practices, into leaking their secrets. In fact, the safety checks of said best practices actually increase the attack surface and may make applications more susceptible to Spectre

Spectre is harder to exploit than Meltdown, but it is also harder to mitigate. [However, it is possible to prevent specific known exploits based on Spectre through software patches.](#)

*End of Lecture 7*