

9/14/26

# COMP 435:



# *Computer Security Concepts*

**Lecture 11:** Putting it all together – Spectre Attacks!



The University  
of North Carolina  
at Chapel Hill

<https://tinyurl.com/comp435-fa26>

# Logistics Update

---



- First lab went out Friday!
  - Due Thurs 9/24
- 2nd Written Assignment due Thursday at 11:59pm.
  - In-class component 70%. Work in small groups to trace a Spectre attack from start to finish
  - Pre-class component 30%. I'm passing out today!

# Plan for Class Today

---



- 10:10-10:45 – Spectre V1
- 10:45-11:00 – Intro to OS Security

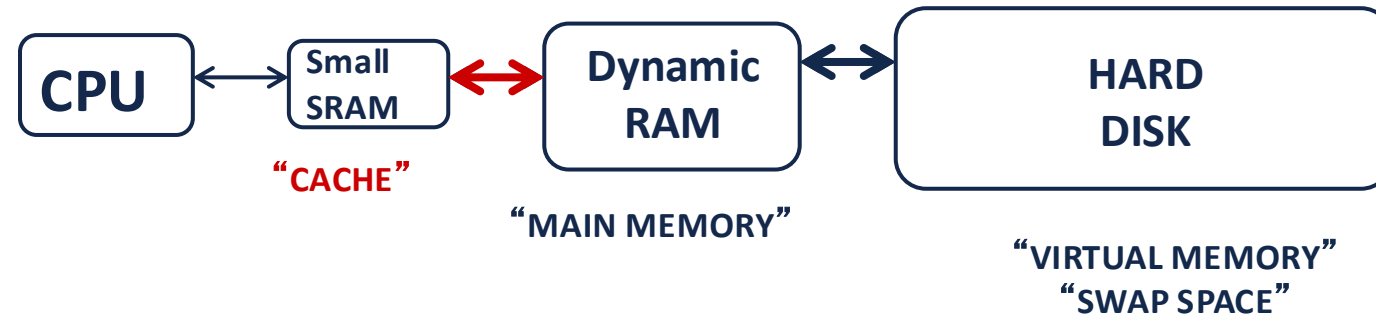
## Guiding Question 😊

*How can cache timing  
+ speculative execution  
create a complete  
attack ?*

# Recall: Caches!



- CPU speed is dominated by memory performance



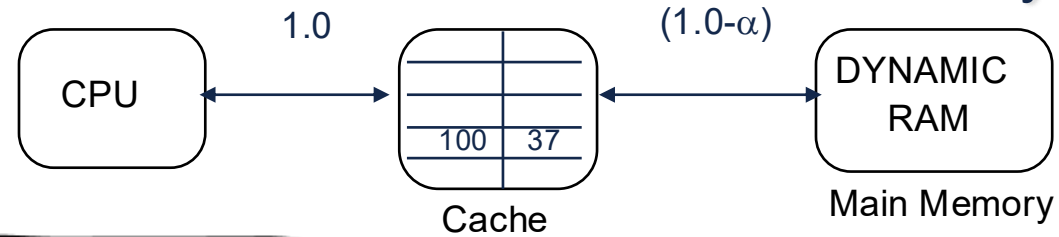
- Make slow MAIN MEMORY appear faster with caching! 😊

# The Cache Idea



## Program-Transparent Memory Hierarchy:

- Cache contains TEMPORARY COPIES of selected main memory locations...
  - e.g. Mem[100] = 37



- Main Goal:

- Improve the average memory access time

HIT RATIO ( $\alpha$ ): Fraction of refs found in cache

- MISS RATIO ( $1 - \alpha$ ): Remaining references
- avg total access time depends on these parameters

$$t_{ave} = \alpha t_c + (1 - \alpha)(t_c + t_m) = t_c + (1 - \alpha)t_m$$

$t_c$  = time to access cache

$t_m$  = time to access main memory

Challenge:  
To make the hit ratio as high as possible.

# Caches are Microarchitectural Optimization

---



- Architectural Level: *defined by ISA*
  - Load x → returns x
- Microarch Level:
  - Was x cached?
  - Which cache line changed?
  - How long did the access take?
- Differences don't matter for functionality so much...
  - **But they do for security!!**

---

Side-channel attacks turn implementation details into information.



UNIT 1: TRUSTING THE MACHINE

# Basic Cache Attacks

# A Simple Timing Side Channel



```
if (secret)
    access(array[0]);
else
    access(array[1000]);
```

Victim Program

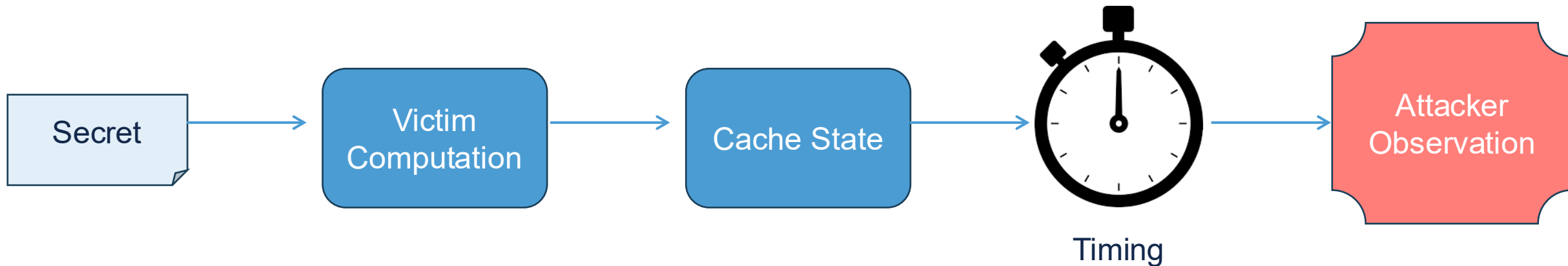
- Suppose a **victim** program looks like this
- The **attacker** can't read **secret**
  - BUT.... \*\*\* They can measure **timing!!!** \*\*\*\*\*
- So they may see something like:
  - array[0] → FAST
  - array[1000] → SLOW

**The secret was probably true!!**

# Let's debrief... what just happened?



- **Timing side channel:** secret-dependent behavior changes execution time in a way an attacker can observe.
- The attacker never directly reads the secret!!
- The steps/flow of an attack:



# Speculative Execution

# Why speculate?



CPU encounters:

```
if (condition) {  
    do_A();  
} else {  
    do_B();  
}
```

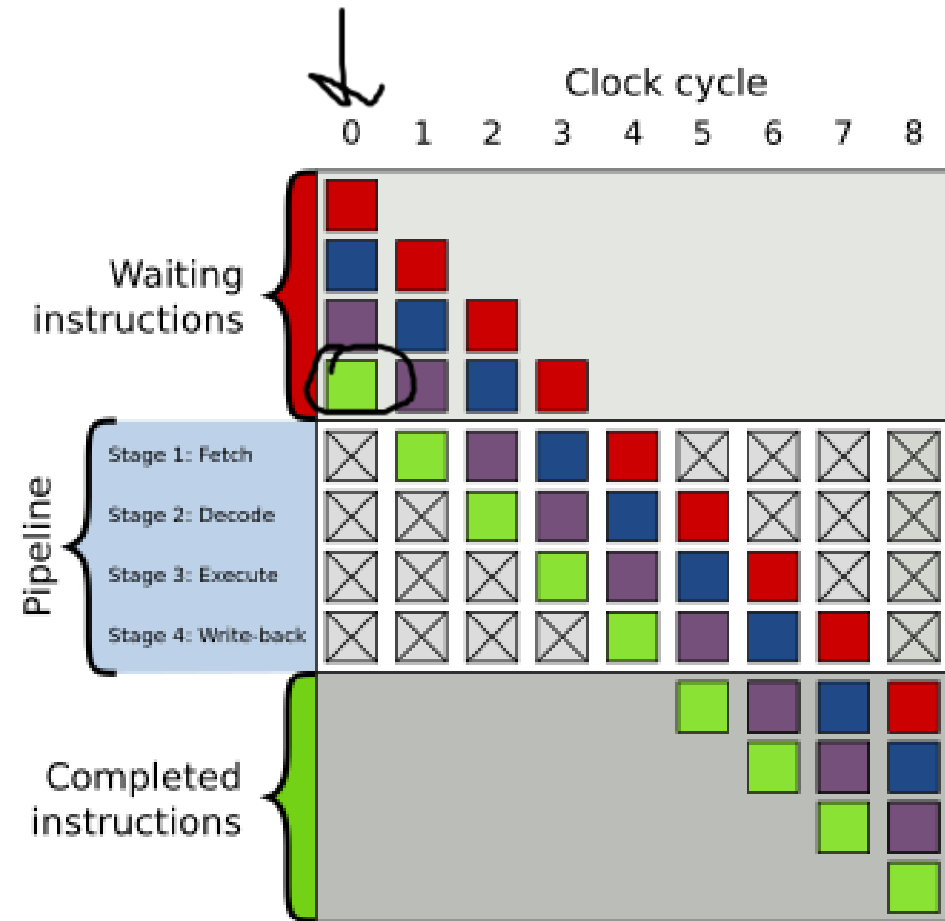
- Instead of waiting to know the result of condition, the processor will **predict** which path do execute and start doing the work **early!**
- If correct → FASTER
- If wrong → must discard microarchitectural results

Some microarchitectural side effects can remain, such as changes to the cache.... Enter: SPECTRE!

# The Branch Predictor: Enabling Speculation



- Hardware in a CPU that guesses the outcome of a conditional jump
- Not known if the prediction is correct until conditional jump has cleared execution stage of pipeline.
- If wrong, speculatively (or partially) executed instructions are discarded + we start over.
- A wrong prediction costs modern processors between 10-20 cycles



## UNIT 1: TRUSTING THE MACHINGE

# Spectre



# Spectre: Simply Put ☺

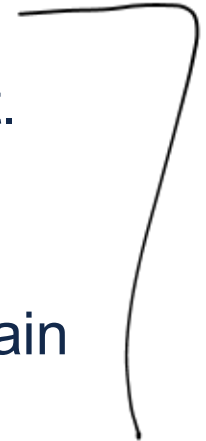


- Spectre tricks a processor into temporarily executing code that should not have run, then uses **cache timing** to infer what that temporary execution touched.



- Key points

- The attacker does not receive the secret as a normal program output.
- The attack creates an **indirect path** (aka implicit flow!!) from secret → cache state → timing.
- The architectural mistake is rolled back; the cache footprint may remain



# The Pieces We Need



1

**A bounds check**

Code meant to stop invalid accesses



2

**A trained  
prediction**

CPU expects the check to pass

3

**A secret-  
dependent access**

Secret chooses which cache line changes

4

**A timing probe**

Attacker finds which line is fast

# Steps in a Spectre Attack

def. r5+ good inputs  
→ malicious at 4 bds



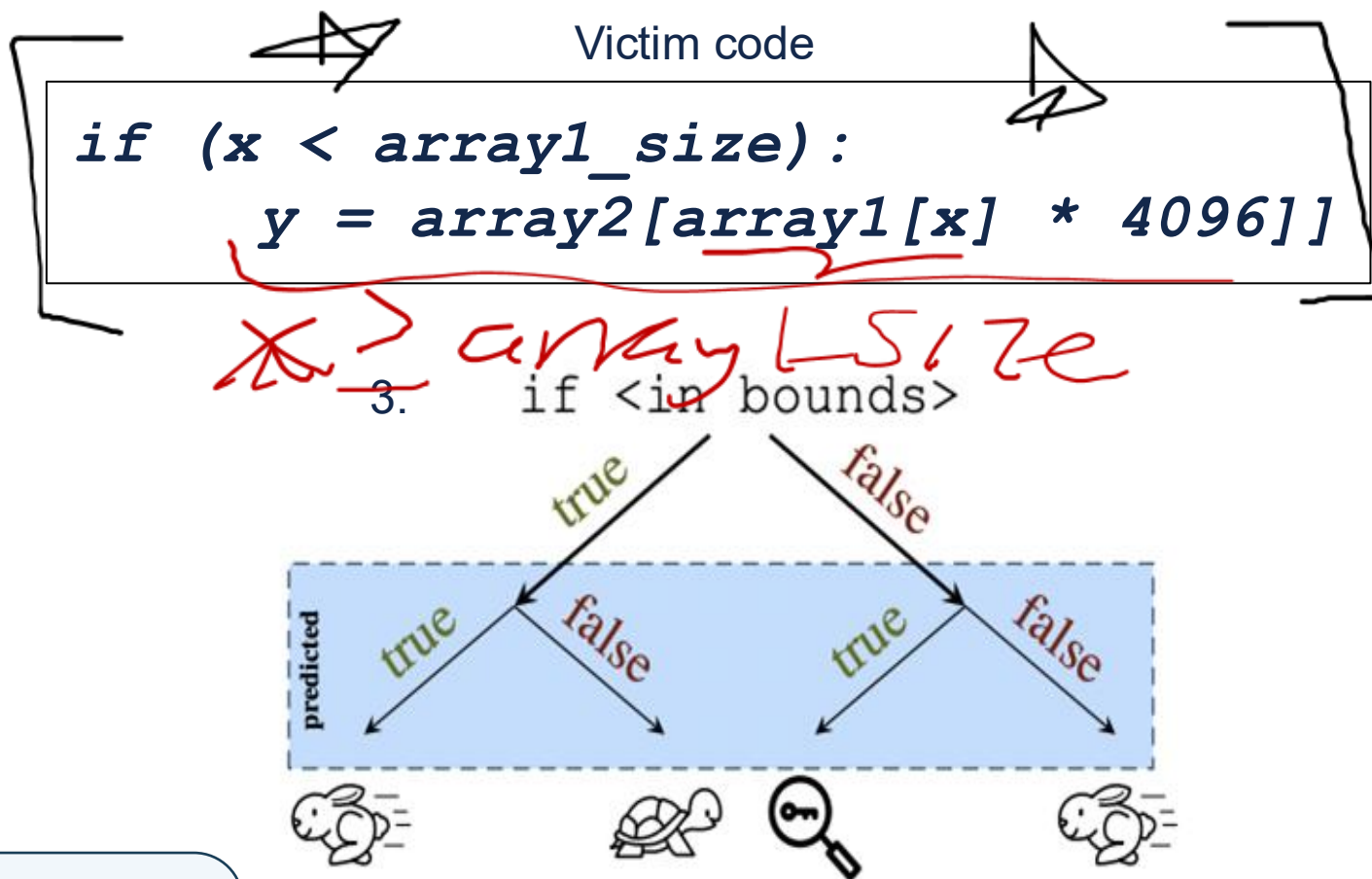
1. Mistrain branch predictor to predict a path that will leak info over a covert channel

2. Invoke code with malicious inputs

3. Extract information from covert channel (AKA the cache!!)

1. Mistrain branch predictor to assume x is in bounds (invoke code many times with valid x inputs)

2. Call with  $x = (\text{addr of some secret}) - (\text{array1 base})$



# Step 0: Make the Cache Measurable



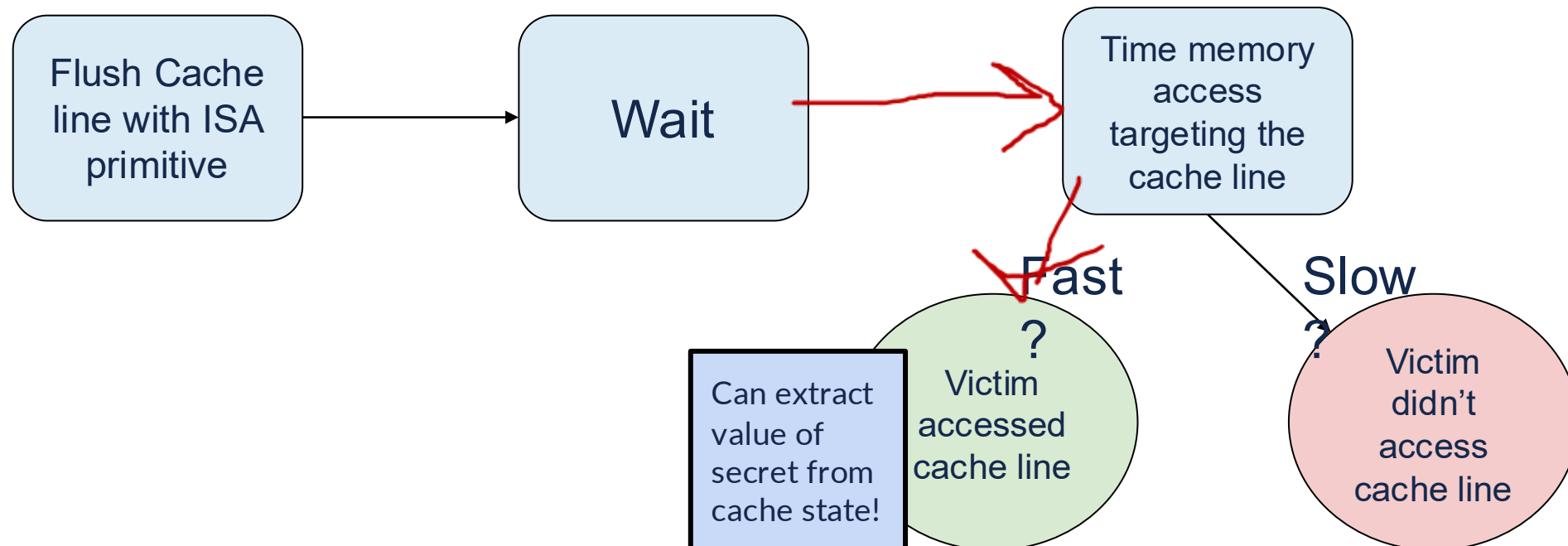
- Flush & Reload Mechanics: ISA Primitive, make all the array2 locations slow
- Then, if one becomes fast later, that is evidence something touched it!

```
flush(array2[0 * 4096]);  
flush(array2[1 * 4096]);  
...  
flush(array2[255 * 4096]);
```

# Step 0: Make the Cache Measurable



- Flush & Reload Mechanics: ISA Primitive, make all the array2 locations slow
- Then, if one becomes fast later, that is evidence something touched it!



# Step 1: Train the Branch Predictor



- Branch predictors are performance features!
- The attacker uses normal, valid calls to make the processor expect that this branch is usually taken.
- Key idea: past behavior influences future predictions!

```
victim_function(valid_x);  
victim_function(valid_x);  
victim_function(valid_x);  
victim_function(valid_x);
```

```
// CPU learns: the bounds check  
usually passes
```

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

Branch predictor will predict:  
x < array1\_size is probably TRUE



## Step 2: Use a Malicious Index

---

- The malicious index points to data outside the allowed array range.
- If the CPU waited for the check, the load should not happen.
- The problem is that speculation may proceed before the check is resolved.

Out-of-bounds  $x$  should not let the program read `array1[x]....`  
But ...

Branch predictor will predict:  
`x < array1_size` is probably TRUE

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

## Step 3: Speculatively Read the Secret



- This is the key! The CPU is not supposed to commit this read.
- During speculative execution, the secret value can be used transiently to **compute the address of another memory access**.

The CPU temporarily follows the predicted path.

The program never gets the secret as a committed result.

But the secret can still influence what happens next.

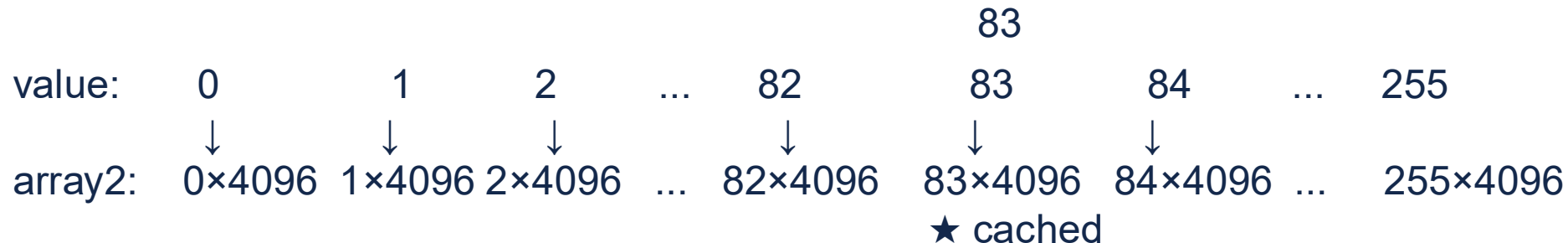
```
if (x < array1_size):  
    y = array2[array1[x] * 4096]
```

## Step 4: Encode the Secret into the Cache



- Suppose the secret byte is 83
- The secret is not printed. Instead, it selects an array2 location.
- If value=83, then array2[83\*4096] is touched and **becomes fast to access later.**

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



**Only one cache-sized region is fast afterward.**

# Step 5: Rollback is not sufficient!



- Correct architectural behavior does not necessarily mean secure behavior.
- Architecturally: the secret read is rolled back.
- Microarchitecturally: the cache footprint may remain and can be measured.

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

## Architectural State:

Wrong-path results are discarded. At ISA level, we redo the instructions, and the program is not allowed to read the secret



**SPECTRE**

Spectre lives in the gap between these two levels!

## Microarchitectural State:

Some effects may remain!  
The cache may still remember which array2 location was touched.

## Step 6: Secret Recovery/Measurement with Timing



- It is not magic: the attacker is just timing accesses and looking for the unusually fast one.
- The fast index corresponds to the secret value that was used speculatively.

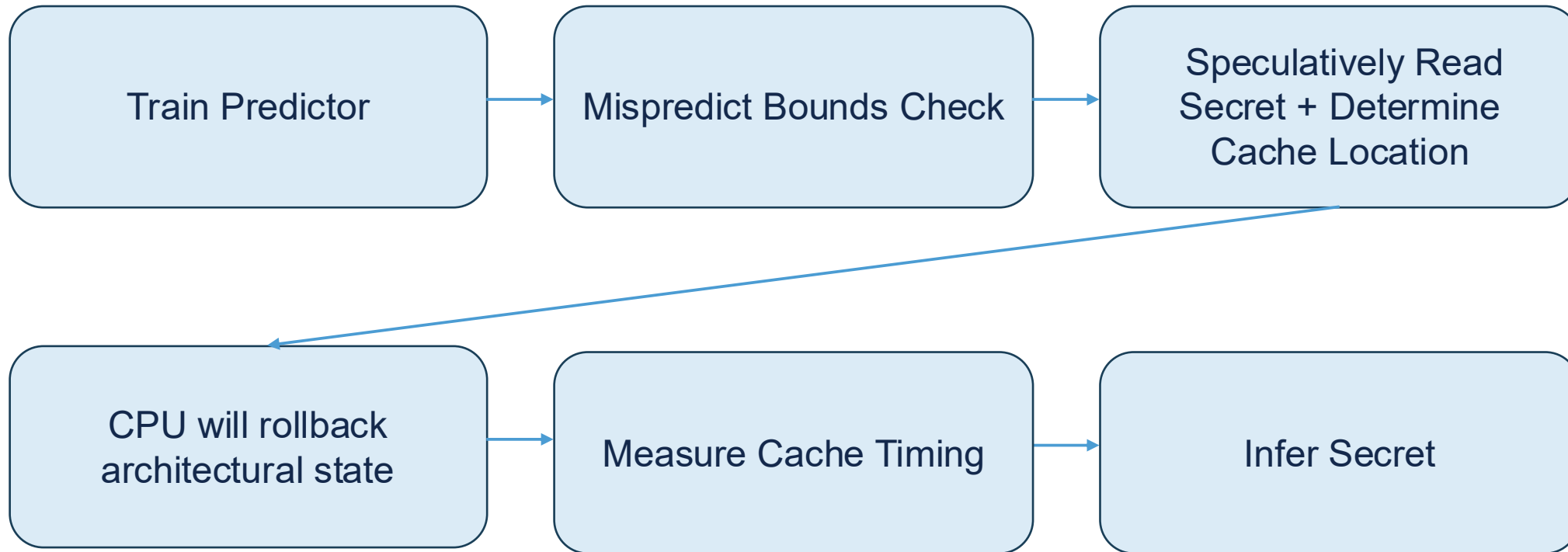
```
time(array2[0 * 4096]);  
time(array2[1 * 4096]);  
...  
time(array2[83 * 4096]); // FAST  
...  
time(array2[255 * 4096]);
```

The attacker probes each possible location

The fast one reveals which location was probed!

If `array2[83*4096]` is fast, infer secret  $\approx$  **83**.

# Full attack path!



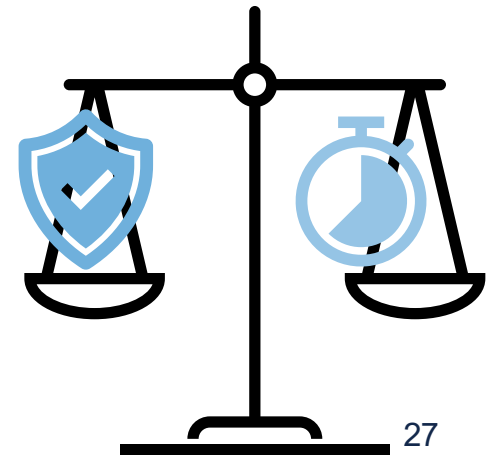
**Information Flow:**  
secret → speculative access → cache state → timing → attacker

# Summary + Impact

---



- Spectre shows that “the program was not allowed to read the secret” is not the same as “the secret could not influence anything observable.”
- Speculation is a performance feature, not a bug you can not simply delete!
- Potential mitigations: add checks, fences, isolation, or compiler transformations.
- Many defenses trade security against performance and compatibility.

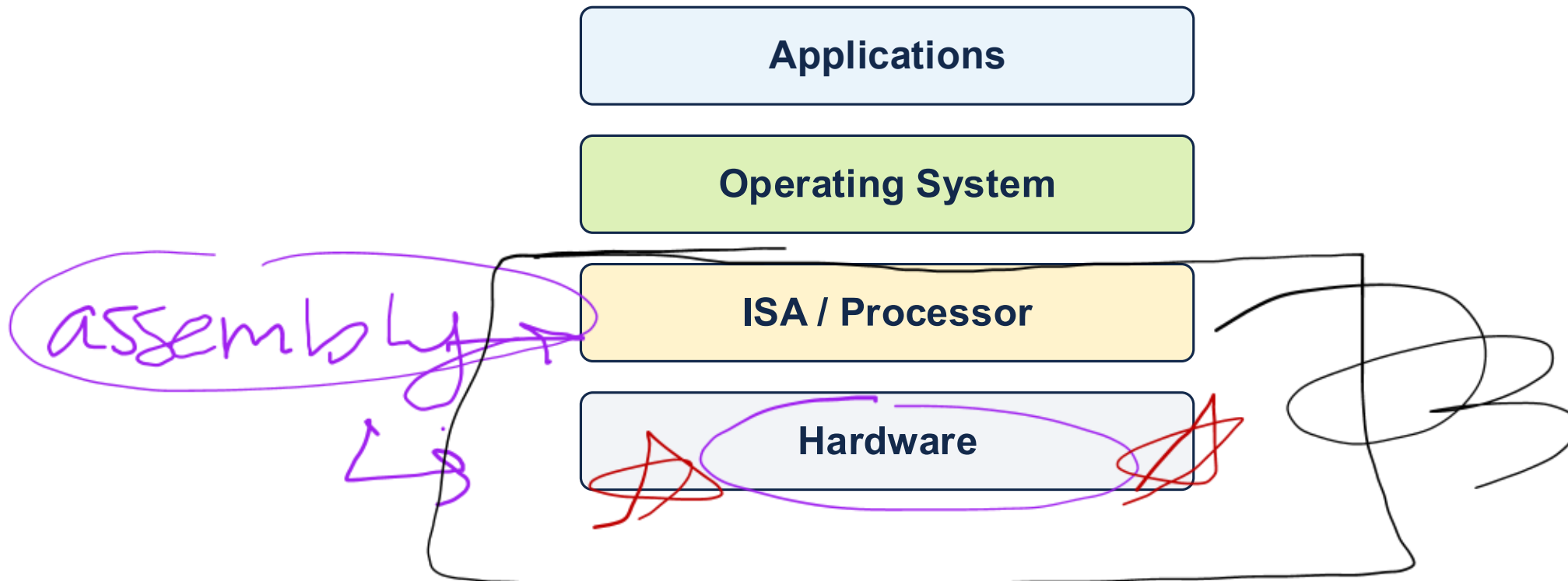


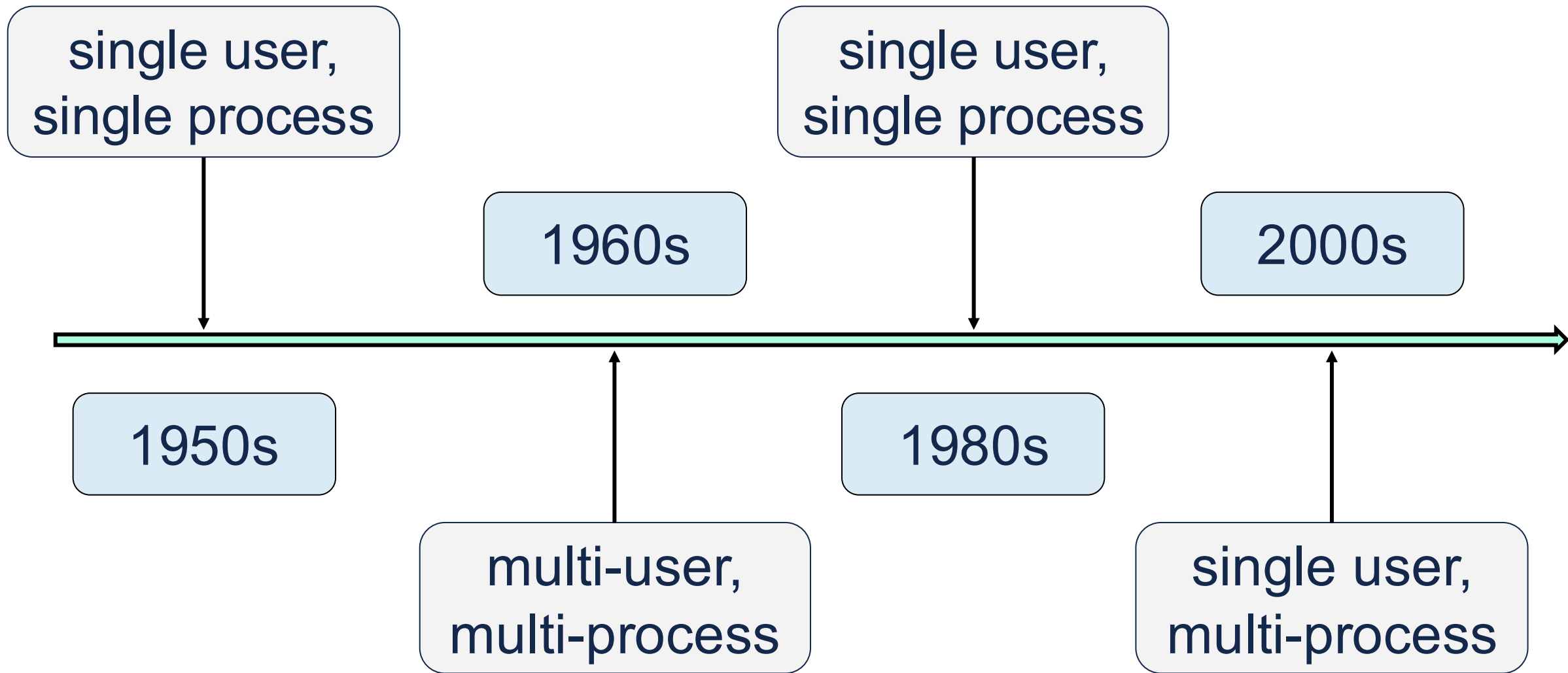
# Operating Systems Security!

# The Stack We Are Trusting



- Every software security mechanism rests on assumptions about the layers below it.

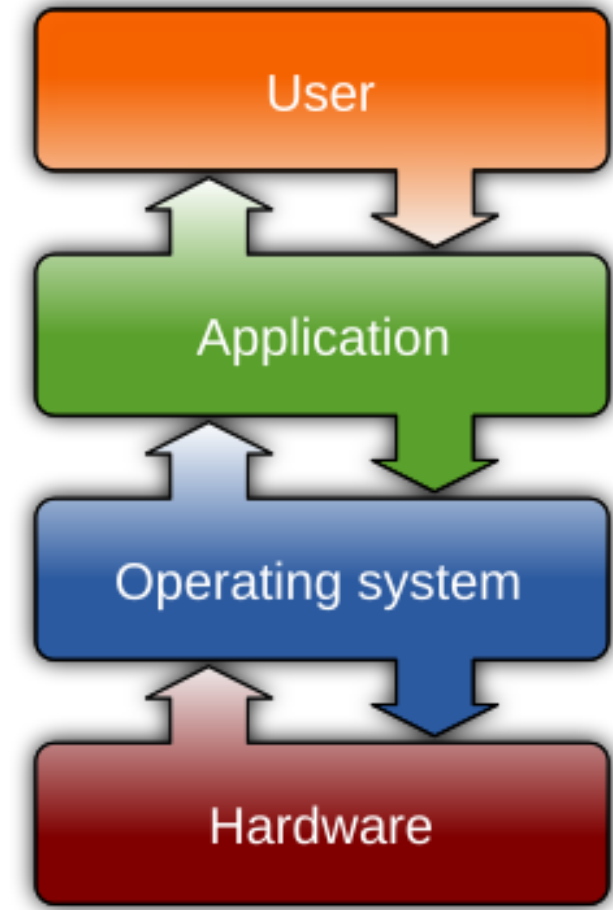




# What is the Operating System??

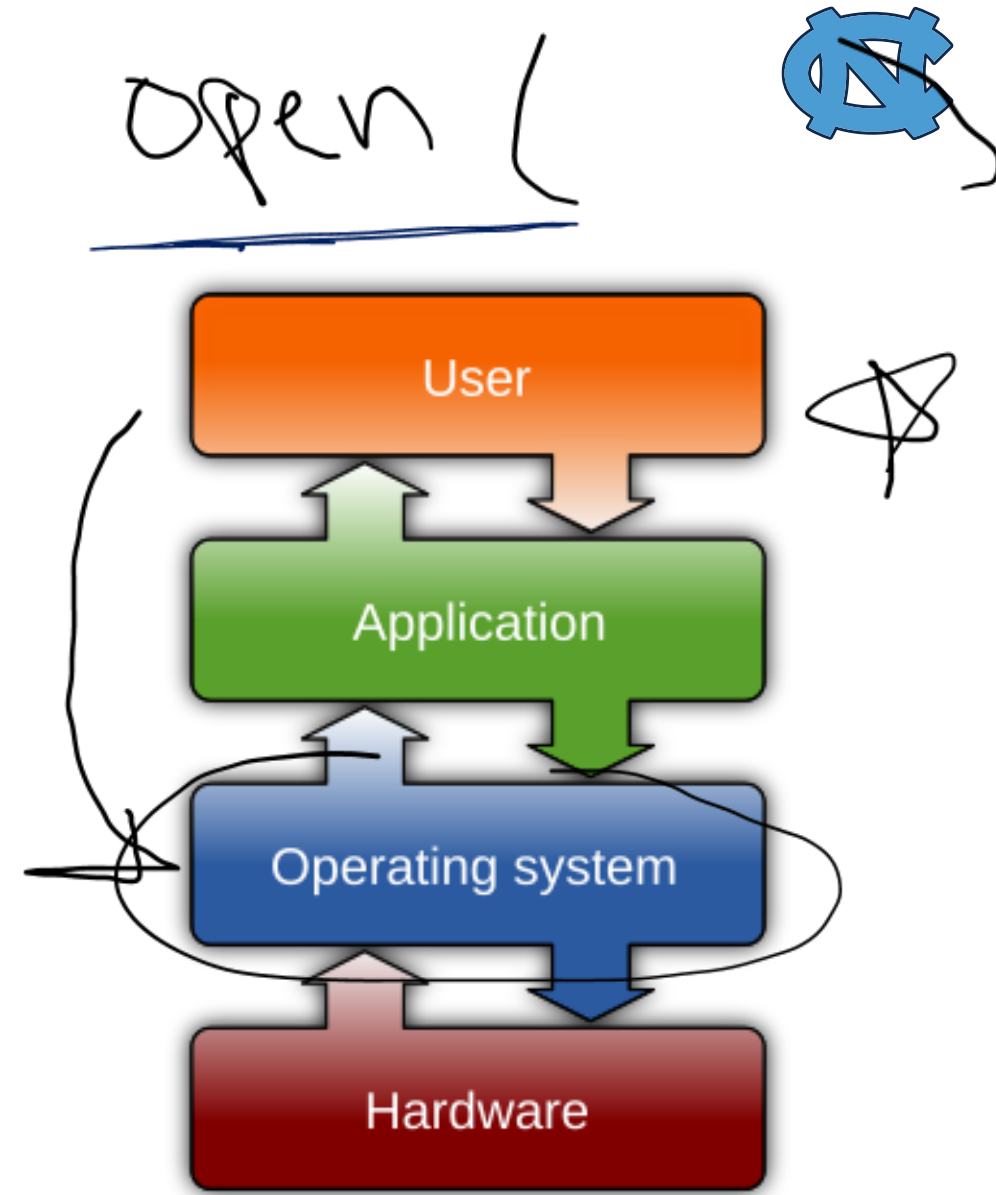


**Software** that **manages** hardware and software **resources**, and **provides common services** for software programs



# What is a System Call??

A system call (syscall) is the mechanism by which a program in user space asks the operating system to do something on its behalf.



# OS Responsibilities



- ~~manage~~ resources
  - fork(), mmap()
- provide interface to HW
  - ~~read()~~, write()
- provide user authentication
  - ~~getuid()~~, setuid()
- mediate interprocess communication
  - pipe()
- protect itself
  - kill(), mprotect()

cat a.txt



~~root~~ == 0  
user



# Manage Resources



- allocation
- sharing
- protection — authorization



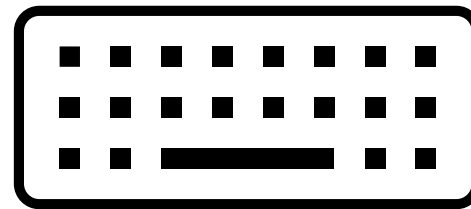
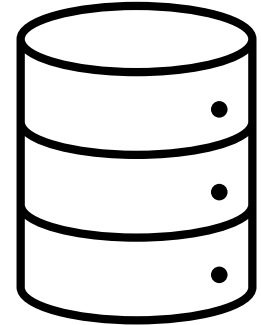
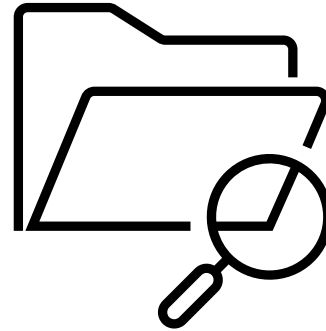
memory,  
disk space  
access to device

✓ Access condition ✓

# Provide Interface to Hardware



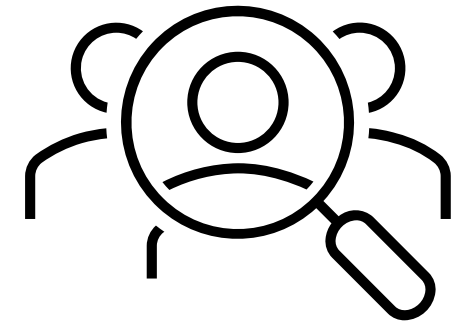
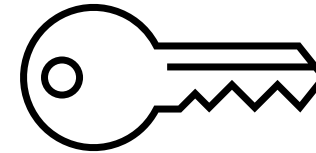
- memory
- file system
- device I/O



# Provide User Authentication



- user accounts
- roles
- passwords

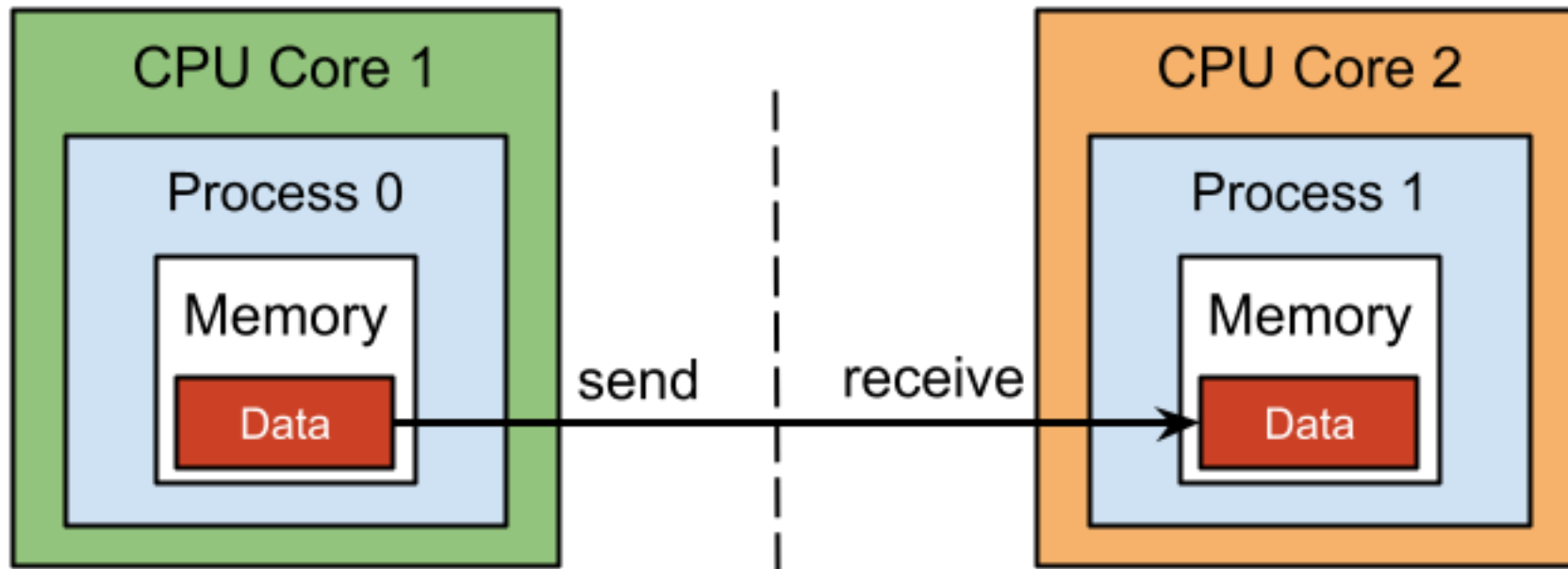


password  
file!



# Mediate Interprocess Communication

- *Interprocess communication facilitated by OS ✓*
- shared memory
  - message passing



# Protect the Operating System

---



- data
- code
- permissions