

9/9/26

COMP 435:

Computer Security Concepts



Lecture 10: Hardware Bugs, Errata and CWEs



The University
of North Carolina
at Chapel Hill

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

Logistics Update



- First lab will go out later today
- We will go over the material needed to get started 😊

Plan for Class Today



- 10:10-10:25 – Information Flow Verification
- 10:25-10:35 – IF Property Writing
- 10:35-11:00 – Intro to Side Channel Attacks

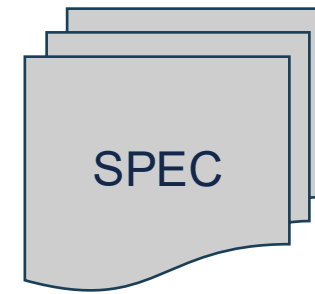
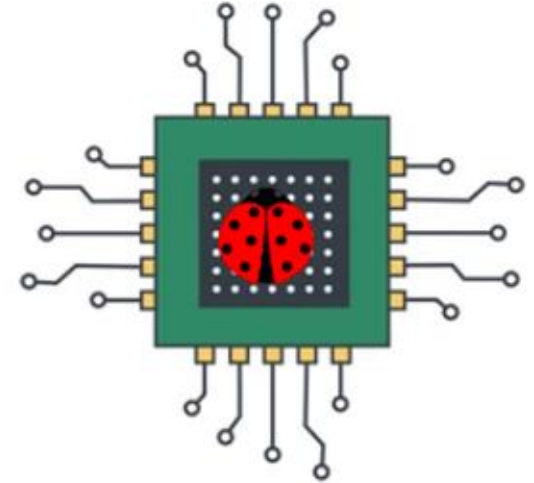
Guiding Question 😊

How can attacker learn secret information without directly accessing it?

Recall: Hardware Bugs



- Hardware **Errata**: design defects or errors.
 - Errata may cause the processor's behavior to **deviate** from published **specifications**.
 - Companies and organizations **document** errata and **changes to designs** to help hardware (and software) engineers.
- We have an **incorrect specification**
 - Bug in the spec leads to bug in the HW
- Or an **imprecise specification**
 - Vagueness in the spec leads to buggy/unwanted behavior



Quick Background: Information Flow

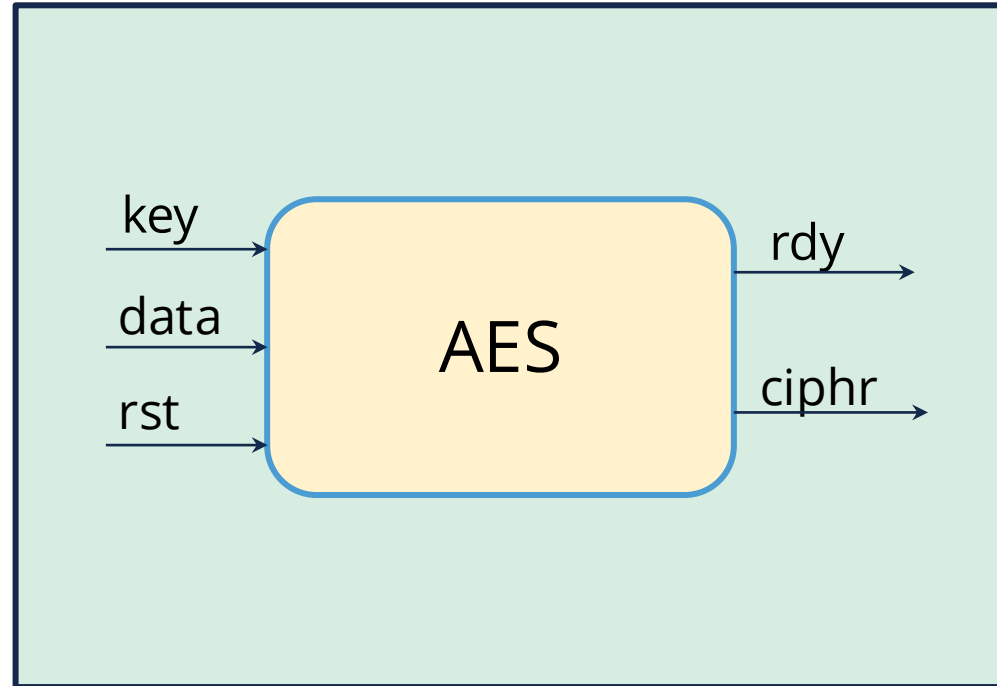


```
1 input  A
2 output E
3 reg B, C, D;
4 assign E = D;
5 if (A)
6     D <= B;
7 else
8     D <= C;
```

Line 6 shows an **explicit flow** from B to D.

Lines 5-6 shows an **implicit flow** from A to D.

Information Flow Property Verification



$\phi:$

key $\nrightarrow$ rdy

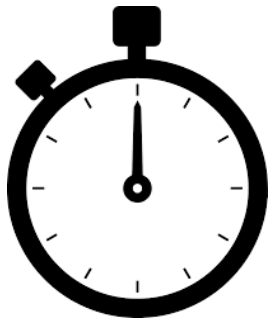
Information
Flow
Property

Information Flow Property Writing Activity



Beyond Program Logic: Side Channels

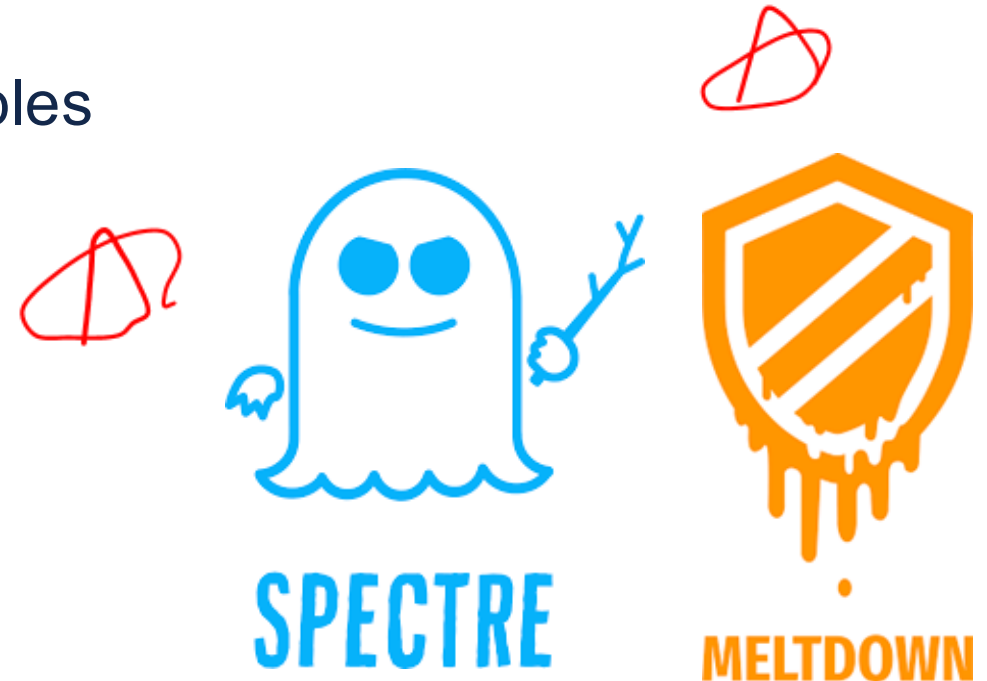
- Information leaks through how a program executes
 - *not just what it computes*
- Attacker observes:
 - Timing
 - Memory/cache behavior
 - Control flow/speculation
 - Power Usage, too!



Side Channel Attacks



- **Side-Channel Attack:** Exploit generated by collecting information during program execution
 - Ex: Memory access patterns, cache state, branch predictor state
- Spectre and Meltdown are two recent examples

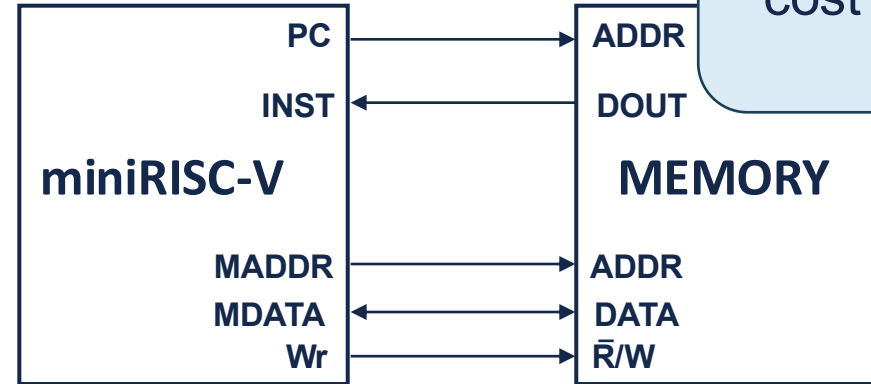


UNIT 1: TRUSTING THE MACHINE

Background: Caches!

What Do We Want in a Memory?

Observation: As capacity goes up and cost does down, latency gets worse.



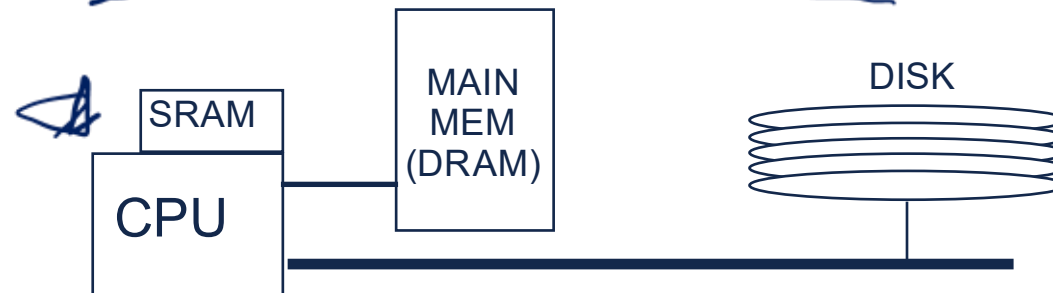
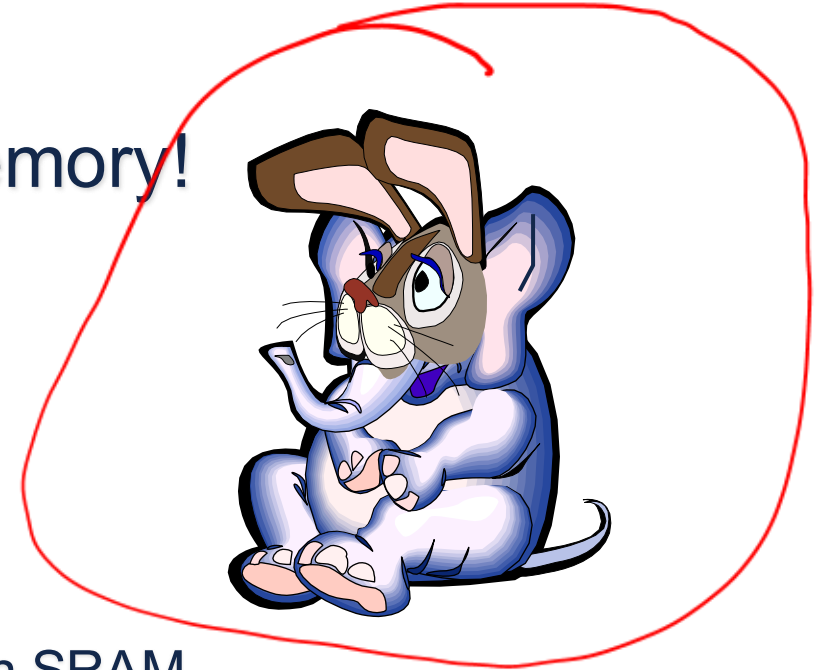
	<i>Capacity</i>	<i>Latency</i>	<i>Cost</i>
Register	1000s of bits	10 ps	\$\$\$\$
SRAM	1-8 MBytes	0.5-1 ns	\$2K/GB
DRAM	1-8 GBytes	50 ns	\$20/GB
Hard disk*	100s GBytes	10 ms	20¢/GB
Want?	Huge	Low (fast)	Cheap!

*non-volatile: Typically magnetic disk drive, but also SSD/flash now

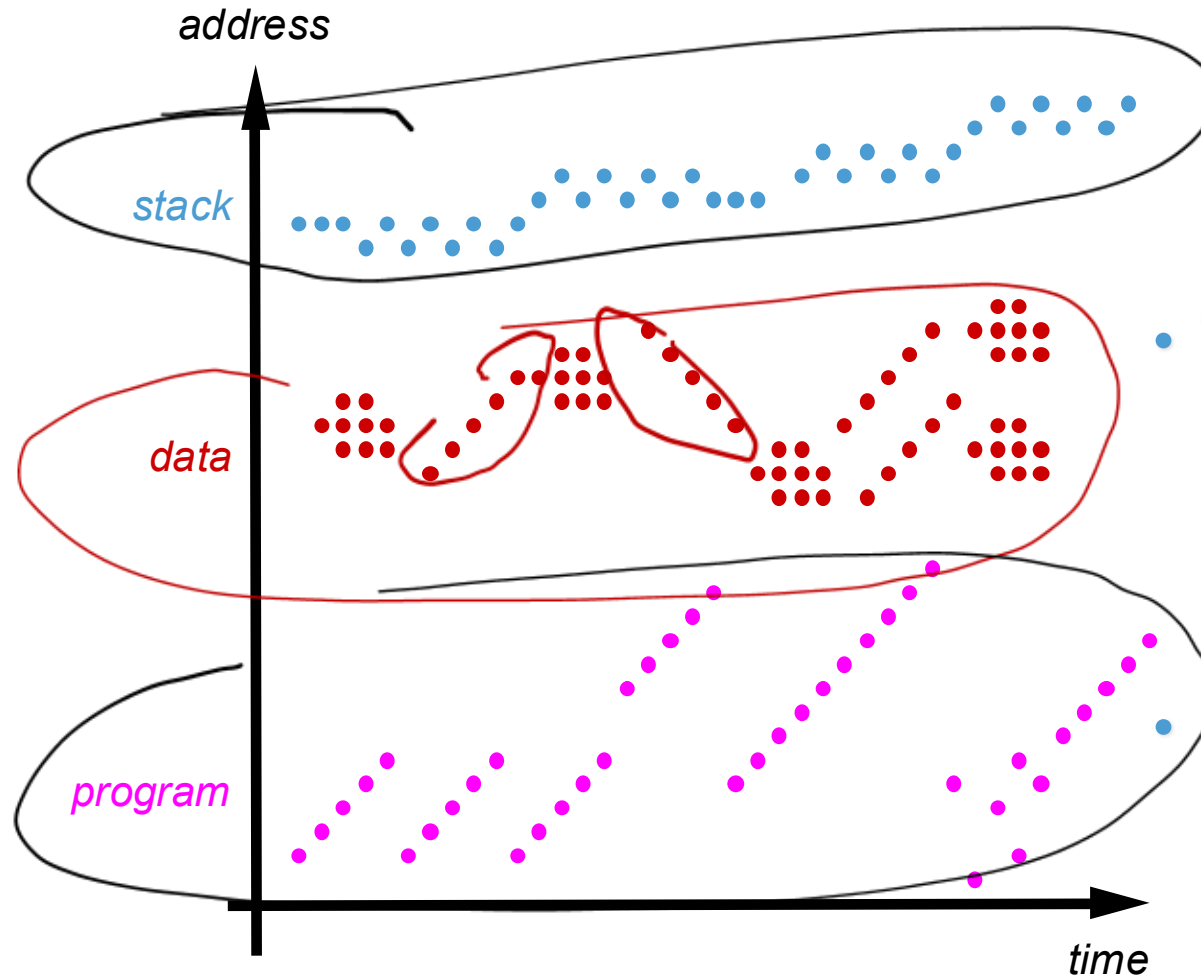
Best of Both Worlds



- What we REALLY want: A BIG, FAST memory!
 - Keep everything within instant access
- We'd like to have a memory system that
 - Stores 2-16 GB of data like DRAM would
 - typical SRAM sizes are in MB, not GB
 - Performs fast like SRAM would
 - typical DRAMs are order of magnitude slower than SRAM
 - SURPRISE: We can (nearly) get our wish!
 - Key Idea: Use a hierarchy of memory technologies:



Typical Memory Reference Patterns



- Memory Trace
 - A temporal sequence of memory references (addresses) from a real program.
- Temporal Locality
 - If an item is referenced, it will tend to be referenced again soon
- Spatial Locality
 - If an item is referenced, nearby items will tend to be referenced soon.

Principle of Locality



- Key Idea: Exploit “Principle of Locality”
 - Keep data used often in a small fast SRAM
 - called “CACHE”, often on the same chip as the CPU
 - Keep all data in a bigger but slower DRAM
 - called “main memory”, usually separate chip
 - Access Main Memory only rarely, for remaining data
 - The reason this strategy works: LOCALITY
 - if you access something now, you will likely access it again (or its neighbors) soon

Locality of Reference:

If a location is accessed in memory, the **same location** or its **neighbors** will likely be accessed soon

Cache

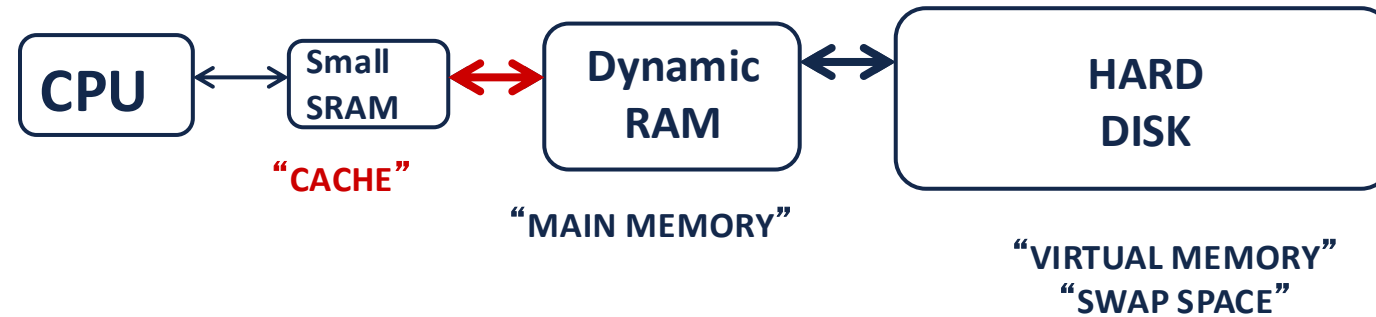


- cache (kash)
n.
 - A hiding place used especially for storing provisions.
 - A place for concealment and safekeeping, as of valuables.
 - The store of goods or valuables concealed in a hiding place.
 - Computer Science. A fast storage buffer in the central processing unit of a computer. In this sense, also called cache memory.
- *v. tr.* cached, cach·ing, cach·es.
 - To hide or store in a cache.

Exploiting the Memory Hierarchy



- CPU speed is dominated by memory performance



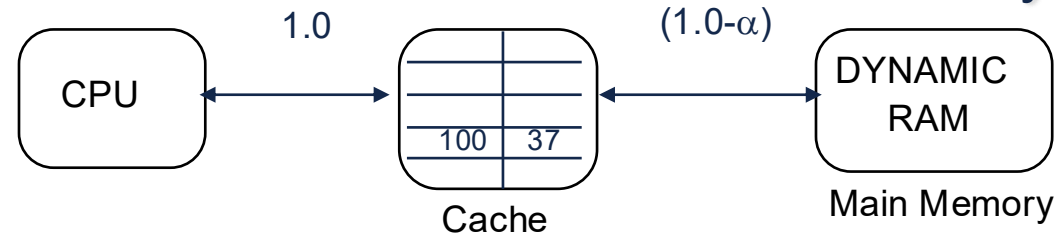
- Make slow MAIN MEMORY appear faster with caching! 😊

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 (α): Fraction of refs found in cache
 - MISS RATIO ($1 - \alpha$): Remaining references
 - avg total access time depends on these parameters



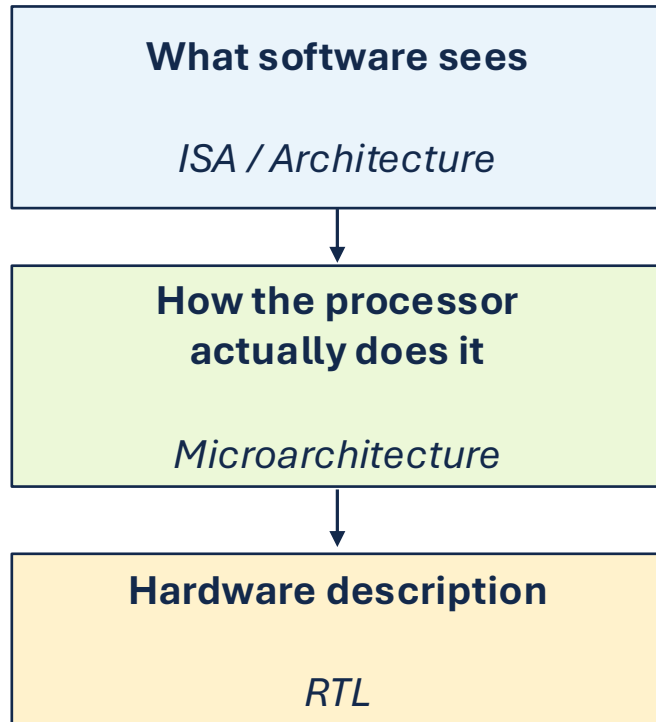
Challenge:
To make the hit ratio as high as possible.

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

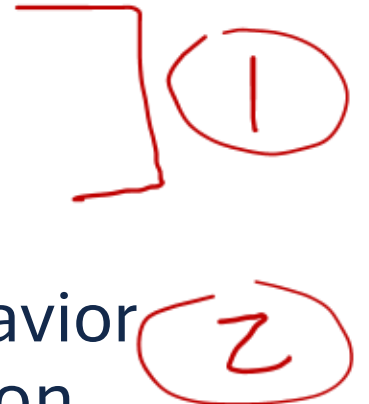
t_c = time to access cache

t_m = time to access main memory

Recall: Architecture vs. Implementation



- A design can be functionally correct at one level and still create security problems at another.
- **Verification** asks whether an implementation satisfies stated properties.
- **Side channels** often exploit behavior below the architectural abstraction.
 - i.e.; microarchitectural timing, cache state



Caches are Microarchitectural Optimiazation



- Architectural Level:
 - Load $x \rightarrow$ 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.

A little thought experiment...



Which would be faster, the right or the left?

```
array[0];  
time(array[0]);
```

```
time(array[1]);
```

A little thought experiment...



Which would be faster, the right or the left?

```
array[0];
```

```
time(array[0]);
```

```
time(array[1]);
```

Memory access time reveals something about cache state.

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

Q: What can the attacker infer here?

A Simple Timing Side Channel



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

Victim Program

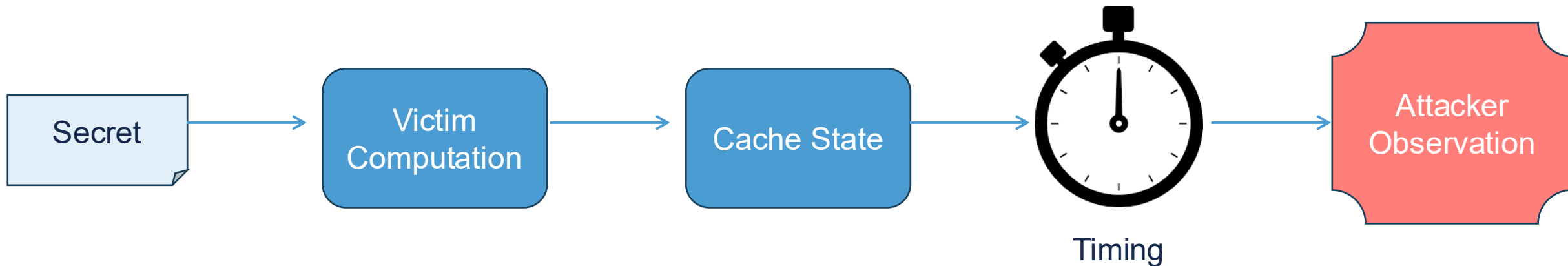
- Suppose a **victim** program looks like this
- The **attacker** can't read **secret**
 - BUT.... ***** They can measure timing!!! *******
- So they may see something like:
 - array[0] → FAST
 - array[1000] → SLOW

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:



What is leaking?



```
if (password[0] == 'A')  
    access(table[0]);  
else  
    access(table[1]);
```

Victim Program

- What is the secret?
- What is observable by the attacker?
- What can the attacker infer?

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

Q: If the CPU discards the result, does it undo every effect of that 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!