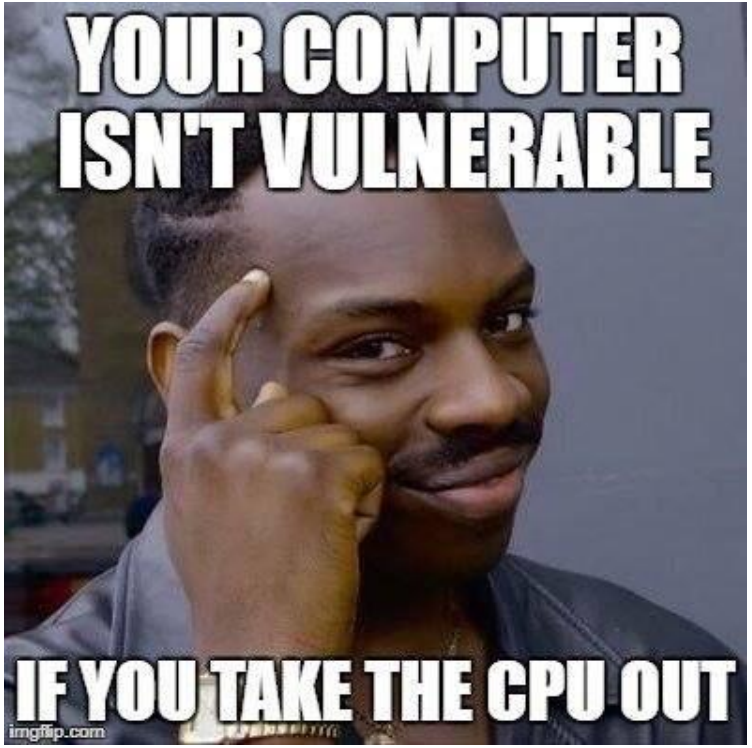
Caches Wrap-up & Side Channel Attacks

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang



Administrivia

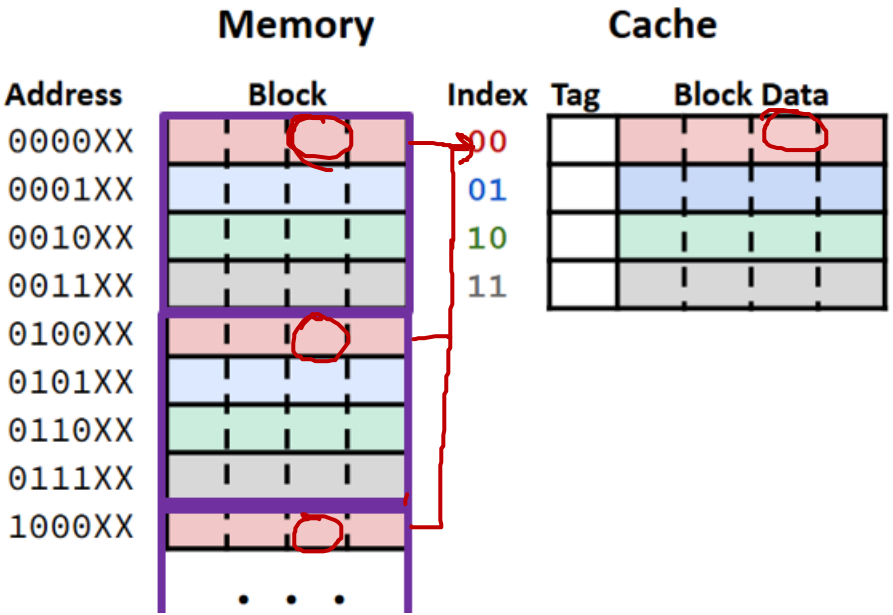
- Today
  - **Quiz 2 due (11:59pm)**
  - Lab4 released (due 8/07)
    - **Start early!** Shorter turnaround than labs 2 and 3
- Wednesday, 7/31
  - RD19 due (1pm)
  - HW17 due (11:59pm)
- Friday, 8/2
  - RD20 due (1pm)
  - No HW :)

Lesson Topics

- **Cache Wrap-up**
  - **More cache images**
- Side Channel Attacks
  - Overview
  - Cache timing attacks
  - Spectre and Meltdown

Recap: Cache Images

- A contiguous, aligned chunk of memory the same size as the cache
- *Any two addresses with the same offset within their respective cache images will map to the same location in the cache*



Cache Image Diagrams from HW17

Example: we have 16B of data

C = 8B, ***K*** = 2B

1. Show data in memory
    a. Each row = 4B

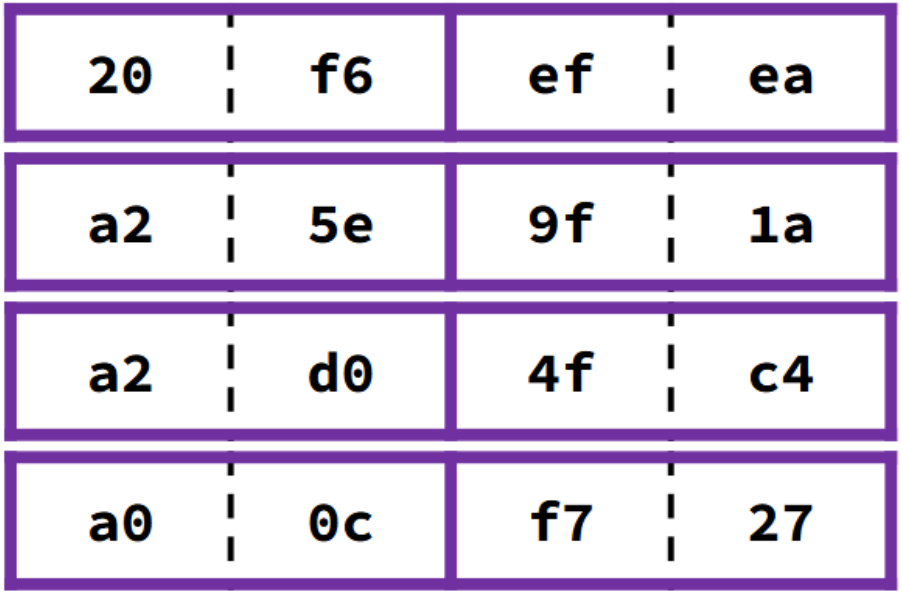
20	f6	ef	ea
a2	5e	9f	1a
a2	d0	4f	c4
a0	0c	f7	27

Cache Image Diagrams from HW17 (pt 2)

Example: we have 16B of data

C = 8B, ***K*** = 2B

1. Show data in the memory
   a. Each row = 4B
2. Purple lines represent cache blocks



Cache Image Diagrams from HW17 (pt3)

Example: we have 16B of data

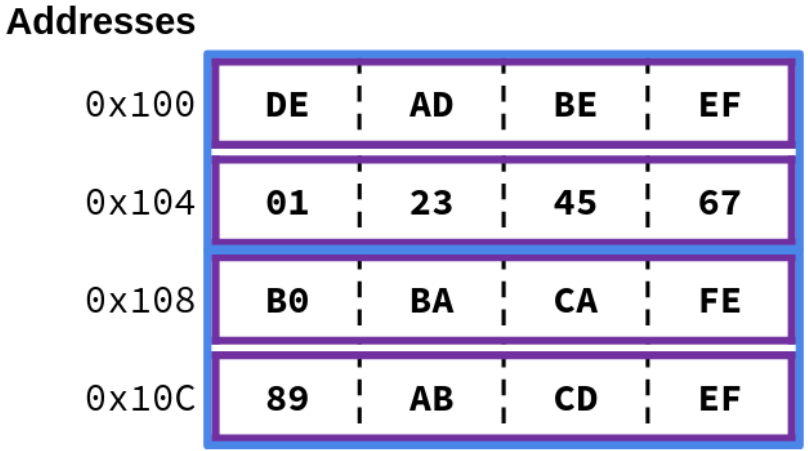
C = 8B, ***K*** = 2B

1. Show data in the memory
   a. Each row = 4B
2. Purple lines represent cache blocks
3. Blue lines represent cache images

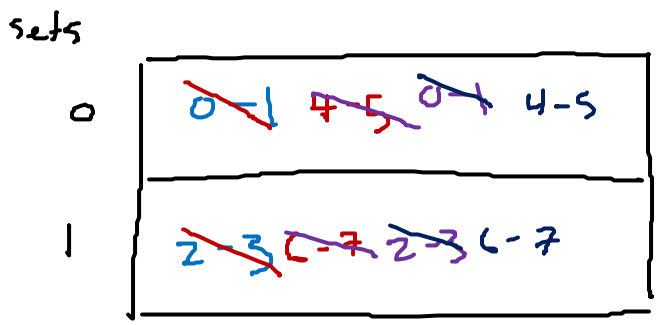
Polling Question

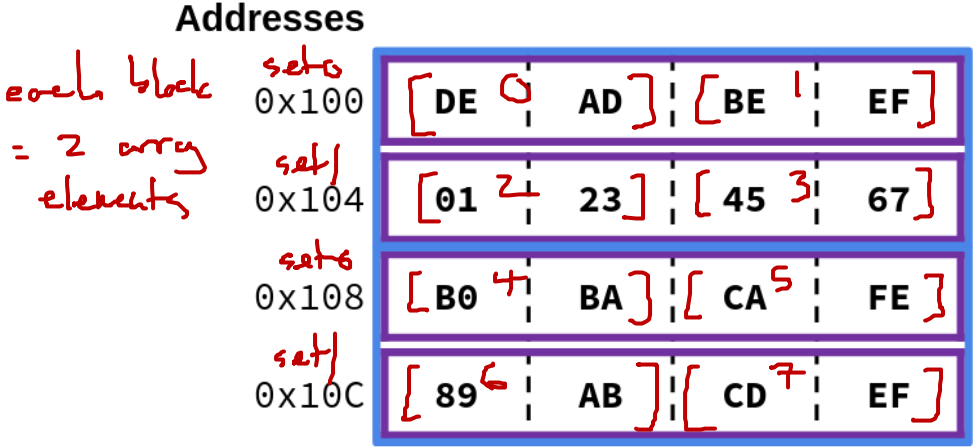
1. (Not on Ed) Based on the diagram, determine the following values:
   A) Block size 4 B
   B) Cache size 8 B
2. Assume the cache is direct-mapped, we start with a cold cache, and that `i`, `even_sum`, and `odd_sum` are stored in registers. How many times will a block be evicted from the cache?

diagram on next slide




```
short ar[8];  //&ar = 0x100
int i, even_sum = 0, odd_sum = 0;
for (i = 0; i < 8; i += 2) {
        even_sum += ar[i];
}
for (i = 0; i < 8; i += 2) {
        odd_sum += ar[i+1];
}
```





access to 4 evicts 0-1
access to 6 evicts 2-3
access to 1 evicts 4-5
access to 3 evicts 6-7
access to 5 evicts 0-1
access to 7 evicts 2-3
6 evictions total

Misses rate = 100%
1st for loop: 4 compulsory misses
2nd for loop: 4 conflict misses

```
short ar[8];  //&ar = 0x100
int i, even_sum = 0, odd_sum = 0;
for (i = 0; i < 8; i += 2) {
        even_sum += ar[i]; 0, 2, 4, 6
}
for (i = 0; i < 8; i += 2) {
        odd_sum += ar[i+1]; 1, 3, 5, 7
}
```

Lesson Topics

- Cache Wrap-up
  - More cache images
- **Side Channel Attacks**
  - **Overview**
  - **Cache timing attacks**
  - Spectre and Meltdown

Side Channel Attacks

- Attacks that use side effects of the physical implementation of a system in order to gain information

Example: can you guess the passcode for this block?

Side Channels in Computer Security

- Observable changes in hardware during program execution
  - Heat
  - Sound
  - Time

Example: how can this code be exploited?

```
// Return 1 if password is correct, 0 otherwise
int check_password(char* input, char* real) {
    for (i = 0; input[i] != 0 && real[i] != 0; i++) {
        if (input[i] != real[i])
            return 0;
    }
    return 1;
}
```

Example: Password Checker

```
// Return 1 if password is correct, 0 otherwise
int check_password(char* input, char* real) {
    for (i = 0; input[i] != 0 && real[i] != 0; i++) {
        if (input[i] != real[i])
            return 0;
    }
    return 1;
}
```

Function returns as soon as it reaches a character that doesn't match

Input	Runtime
aaaaa	1ms
baaaa	1ms
caaaa	**2ms**
cbaaa	2ms
ccaaa	2ms
cdaaa	2ms
ceaaa	**3ms**
…	…

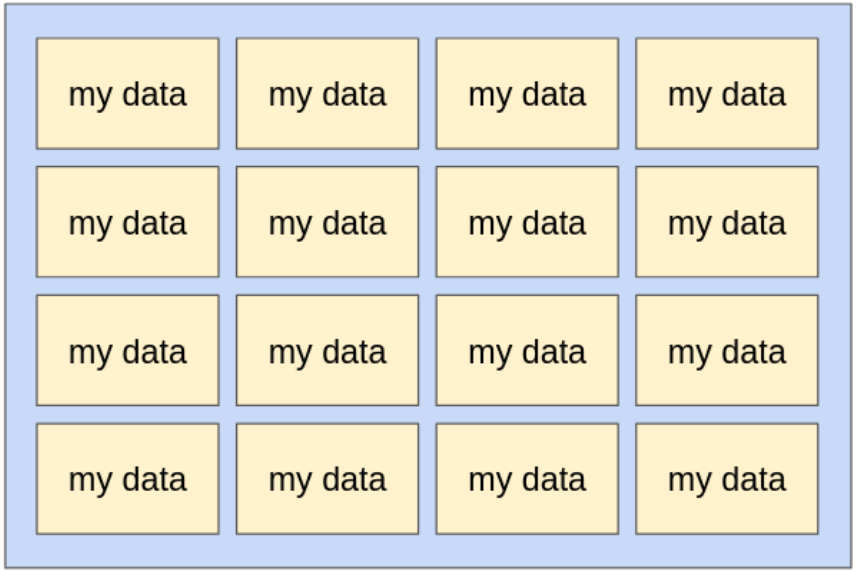
Cache Timing Attacks

- Recap:
  - Cache misses take significantly longer than hits
  - Each address will map to a single set in the cache
- Conclusion:
  - By monitoring how long a memory access takes, **we can tell whether or not it's in the cache!**
- Common method: **"Flush + Reload"**
  - Allows attacker to figure out whether some address was accessed by another program

Attack Time

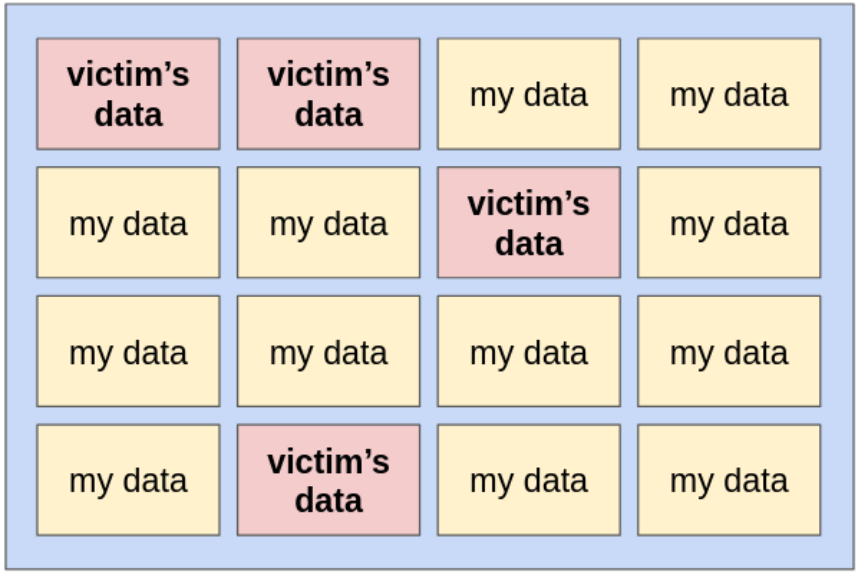
Flush + Reload

1. Fill the cache with data (“flush”)



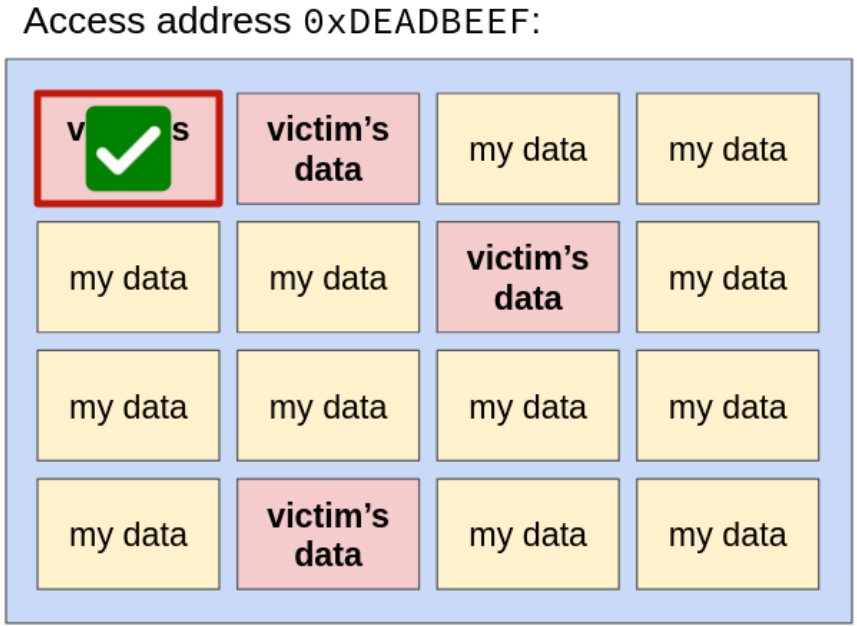
Flush + Reload (pt 2)

1. Fill the cache with data (“flush”)
2. Let victim code run
    a. It will load its data into the cache



Flush + Reload (pt 3)

1. Fill the cache with data (“flush”)
2. Let victim code run
   a. It will load its data into the cache
3. Access the address we want to know about (“reload”)
   a. Slow: it’s not in the cache
   b. Fast: it is in the cache - victim must have accessed it!



Lesson Topics

- Cache Wrap-up
  - More cache images
- **Side Channel Attacks**
  - Overview
  - Cache timing attacks
  - **Spectre and Meltdown**

Speculative Execution

- Modern CPUs use **pipelining** - execute instructions ahead of time
  - Ex: while CPU is computing the result for one instruction, start fetching data for the next one
- An analogy:
  - Imagine you have to bake 10 cakes
    - Each one takes 1 hr to make (30 min to make the batter + 30 min to bake)
    - Making each batch separately: 10hrs :(
  - What can you do to speed this up?
    - While one batch is baking, make the batter for the next one!
    - Total time: 5.5hrs :)

Branch Prediction

- Problem: when executing a conditional branch, how does the CPU know which instruction to start working on?
  - Doesn't know which instruction will come next until *after* the jump finishes
- Solution: **branch prediction**
  - CPU learns observes program behavior during branches
  - Makes an educated guess as to whether a branch will be taken based on previous behavior
    - If the guess is correct: we saved time!
    - If it's wrong: go back and execute correct branch

Branch Prediction Example:

```
for (int i = 0; i < n; i++) {
        <loop body>
}
. . .
```

```
    movl    $0, %eax      # i=0
    jmp     .condition
.loop:
    <loop body>
    addl    $1, %eax      # i++
.condition:
    cmpl    %edi, %eax    # i<n
    jl      .loop
    . . .
```

- When executing the **`jl`** instruction, is it more likely to take the jump (restart loop), or not?
  - *Hint: which scenario occurs more often?*

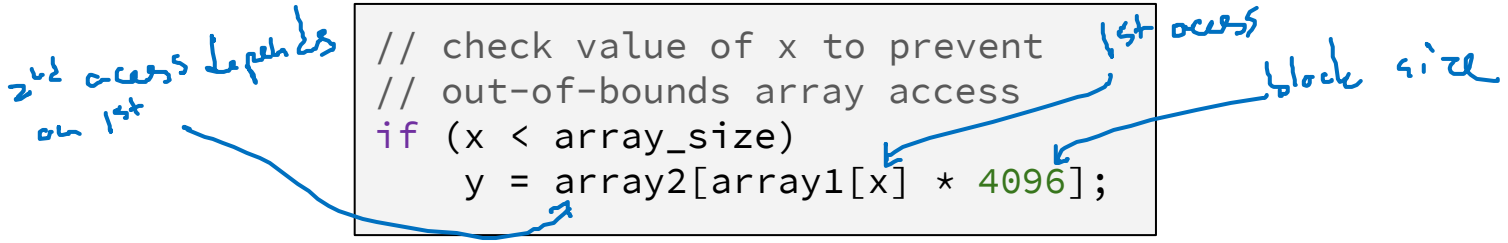
More likely to take the jump
- jump to loop body n times
- only don't jump once

Spectre Attacks

Exploit both cache timing and speculative execution

1. Train branch predictor to take a particular branch
2. Make that branch access some memory location you normally wouldn't have access to
    a. Loads that memory location into the cache
    b. Even after the CPU realizes it's in the wrong branch, the data will still be in the cache!
3. Use cache timing to figure out what was accessed

Specter Attacks Example



1. Train branch predictor to go into the if statement
2. Pass in a value of `x` that is much larger than `array_size`
   a. The CPU will speculatively execute the if statement body while the conditional jump is being computed
   b. Loads `array2[array1[x] * 4096]` into the cache
3. Use flush + reload to figure out where `array2[array1[x] * 4096]` is
   a. This will tell you what the value of `array1[x]` was

Spectre and Meltdown History

- In 2018, two different research groups found that Intel CPUs were vulnerable to attacks using cache side channels and speculative execution
  - **Spectre** exploits branch prediction to access victim's data
  - **Meltdown** exploits a race condition to gain access to OS memory
  - Affected all Intel CPUs from the previous *20 years*!
- Since then, other variants have been found
  - Vulnerabilities in ARM CPUs too

Spectre and Meltdown Mitigations

- Prevent attacker from accessing victim's data
  - **Virtual Memory**: future lecture topic!
  - Other methods not talked about in this class
- Prevent speculative execution
  - Turn off branch prediction when running vulnerable code
  - Don't let branch prediction training carry across programs
    - i.e., CPU "forgets" prior training when switching between programs
    - Prevents attacker from mistraining branch predictor

- However, these strategies not used that often. Why?

Spectre and Meltdown Mitigations (pt 2)

- Many of these mitigation strategies are ***not*** commonly used. Why?
  - High performance cost.
    - Turning off speculative execution slows down a CPU by ~30%
  - Attacks are unlikely
    - Require the attacker to already have access to the victim computer *and* some the victim program
    - There is no evidence of these attacks actually occurring in the wild (i.e. outside of a research environment)

Discussion

Discuss the following in groups of 2-4, then we'll share out as a class.

- Despite the initial panic, little has actually been done to remove the vulnerabilities that allow Spectre and Meltdown to occur. Why do you think this is? Some things to consider:
  - Would you buy a computer that had was 100% resistant to being hacked, but was 30% slower?
  - If you were a business executive at a tech company, how much would you be willing to pay to ensure your product was completely secure?
    - What if it came at the cost of other functionality (speed, features, etc.)
- Do you think we should be doing more, or are things fine the way they are?

Conclusion

- Cache side channel vulnerabilities are a result of features we've added to improve performance (caches, speculative execution, etc.)
- Completely preventing attacks would require us to give up those performance gains, so we just live with it ¯_ (ツ)_/¯
  - Tradeoff between security, functionality, and performance

Want to learn more? Take these classes:

- CSE 484 (Security)
- CSE 451 (Operating Systems)
- CSE 469 (Computer Architecture)