CSE127, Computer Security

System Security II: Side channels, Covert channels, Caches, Meltdown, Spectre

UC San Diego

Housekeeping

General course things to know

- PA2 due at **1/29** at 11:59
  - This one is a toughie. Good luck!
- PA3 released **1/30** at midnight
  - Web attacks (we start talking about the web on Thursday)
  - My read: not as bad at PA2, but note less time (~1.5 weeks instead of 2 weeks)
  - Things to get ready for: SQL injection, XSS, and… JavaScript (not my favorite language)

Previously on CSE127…

Systems + Privilege

- Last class we talked about **isolation** and **privilege**
  - How we implement least privilege, privilege separation, and complete mediation in operating systems + processes
  - Basic idea: **protect the sensitive or secret stuff so it can't be access across a trust boundary** (e.g., recall protected kernel memory reads from user mode)
- Assumption: we know what the trust boundaries are and, specifically, that ***access*** to something is easy to identify

Today's lecture — Side Channels

Learning Objectives

- Understand the basic concept of a side channel, how side channels work in practice, and where we might find interesting side channels
- Remind ourselves the basics of a CPU cache and understand the risks and dangers of cache side channels
- Get into the details of very famous architectural side channel attacks: Rowhammer, Meltdown, and Spectre

Side Channels

A hypothetical

```
passwd = “abcdefghijklmnop”

def check_passwd(inp):
    for x in range(len(inp)):
        if inp[x] != passwd[x]:
            return False
    return True
```

- What does this code do?
- What is the time complexity of the function check_passwd?

A hypothetical

```
passwd = “abcdefghijklmnop”

def check_passwd(inp):
    for x in range(len(inp)):
        if inp[x] != passwd[x]:
            return False
    return True
```

- What does this code do?
- What is the time complexity of the function check_passwd?
- Does this function take the same amount of time every time you call it?

A hypothetical

```
passwd = "abcdefghijklmnop"

def check_passwd(inp):
    for x in range(len(inp)):
        if inp[x] != passwd[x]:
            return False
    return True
```

- What does this code do?
- What is the time complexity of the function check_passwd?
- Does this function take the same amount of time every time you call it?

No! It depends on where the first mismatch is.

Breaking our beautiful password checker

```
passwd = "abcdefghijklmnop"

def check_passwd(inp):
    for x in range(len(inp)):
        if inp[x] != passwd[x]:
            return False
    return True
```

- Let's say we want to learn what the password is, we have control of `inp` and we have infinite guesses. How might we do this?

Breaking our beautiful password checker

```
passwd = "abcdefghijklmnop"

def check_passwd(inp):
    for x in range(len(inp)):
        if inp[x] != passwd[x]:
            return False
    return True
```

- One strategy…
  - Start with a random password, *time* how long it takes to get a response
  - Start changing one byte at a time, starting with first character (a, b, c… etc.), measure time taken
    - ***Longer responses mean more correct letters!***

Side Channels

Yeesh

- We are taught to think of systems, functions, and algorithms as **black boxes**
  - AKA abstractions that consume input and produce output
  - We assume that all *side effects* are about output (e.g., values in memory, I/O, etc.)
- But sometimes…. critical information can be revealed in ***how*** the output is produced
  - E.g., timing… but many others, how *loud*, how *hot*… these are artifacts of the *implementation* rather than the *abstraction*
- **Side channel:** A source of information beyond the output specified by the abstraction

Types of Side Channels

Consumption vs. Emission

- **Consumption:** how much of a resource is being utilized to perform the operation?
  - Time is one example… others?

Types of Side Channels

Consumption vs. Emission

- **Consumption:** how much of a resource is being utilized to perform the operation?
  - Time is one example… others?
  - Power, memory, network, etc.
- **Emissions:** what out-of-band signal is generated in the course of performing the operation?
  - EM radiation, sound, movement, error messages, etc.

Types of Side

Consumption vs. Em

- **Consumption:** how o perform the operation?
  - Time is one exam
  - Power, memory,
- **Emissions:** what ou urse of performing the operation?
  - EM radiation, sou

How might you read the computer screen from the window?

Figure 2. The basic setting: The monitor faces away from the window in an attempt to hide the screen's content.

From: Compromising Reflections – or – How to Read LCD Monitors Around the Corner

Reflections are your friend!

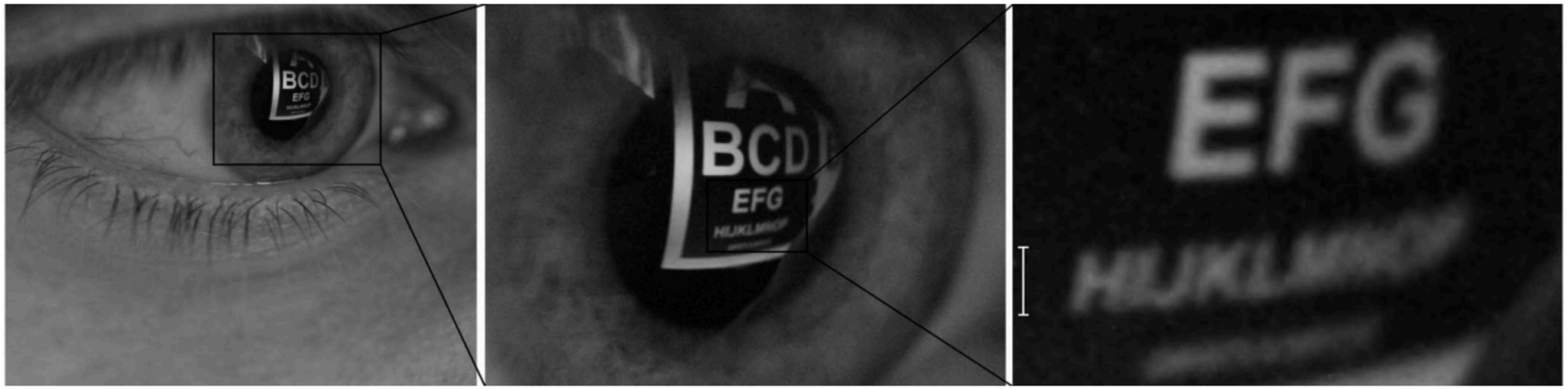


Figure 1. Image taken with a macro lens from short distance; the distance between the eye and the monitor was reduced for demonstration. Readability is essentially limited by the camera resolution.

From: Compromising Reflections – or – How to Read LCD Monitors Around the Corner

Reflections are your friend!

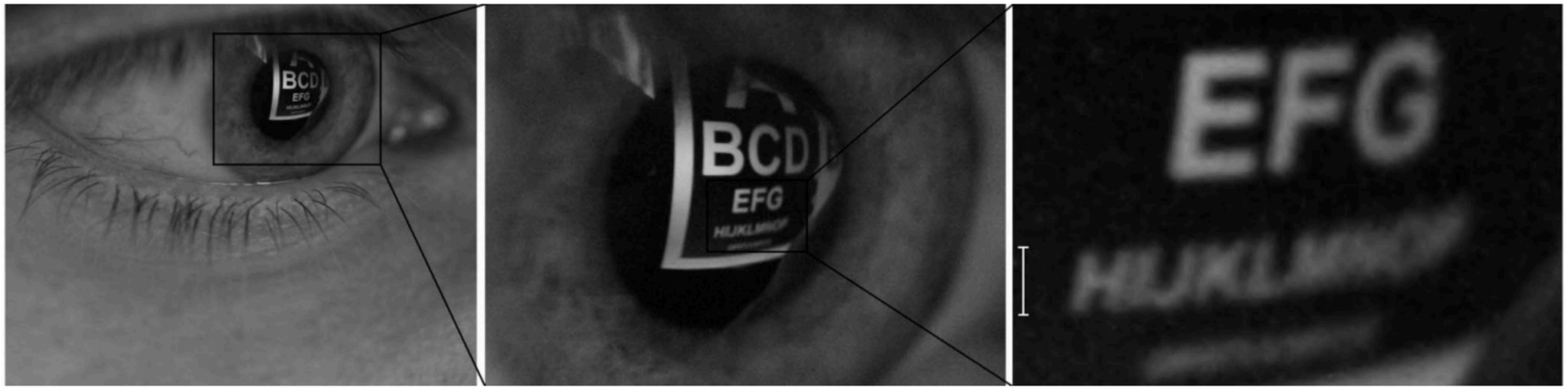


Figure 1. Image taken with a macro lens from short distance; the distance between the eye and the monitor was reduced for demonstration. Readability is essentially limited by the camera resolution.

Figure 5. Reflections in a tea pot, taken from a distance of 10m. The 18pt font is readable from the reflection.

The craziest reflection of them all…

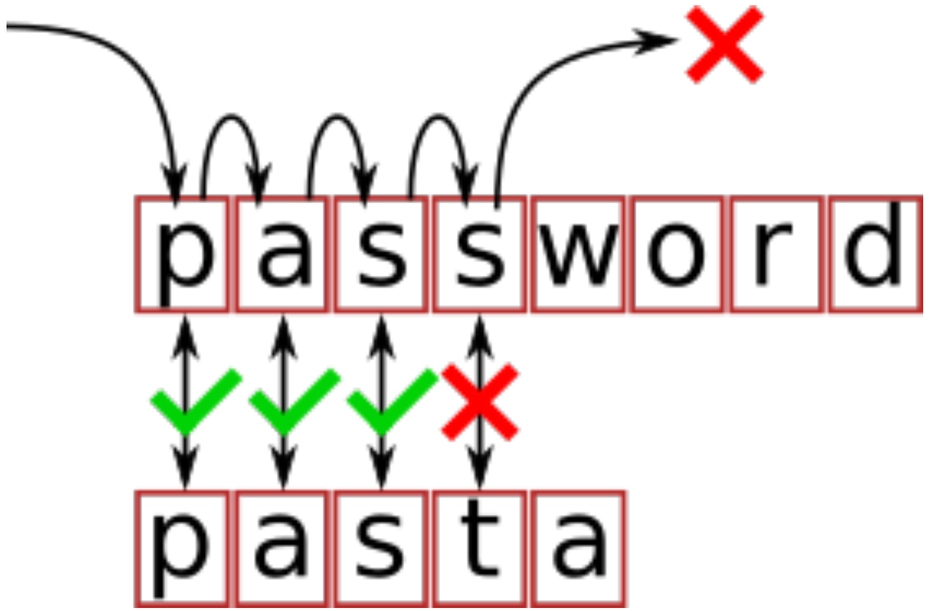
Figure 12. Reflections in a 0.5l plastic Coca-Cola bottle, taken from a distance of 5m. Because of the irregular surface, only parts of the text are readable.

From: Compromising Reflections – or – How to Read LCD Monitors Around the Corner

Side Channel Examples

Tenex password verification

- Old OS named Tenex; vulnerability was discovered by Alan Bell in 1974
- Basic software bug was the same… but how to exploit?
  - Tenex offered userspace programs a lot of control over paging
  - Basic idea: split user input password over multiple memory pages, forcing system to fault if next page is not in memory
- System will not fault if the check fails, allowing user to recover full password in linear time

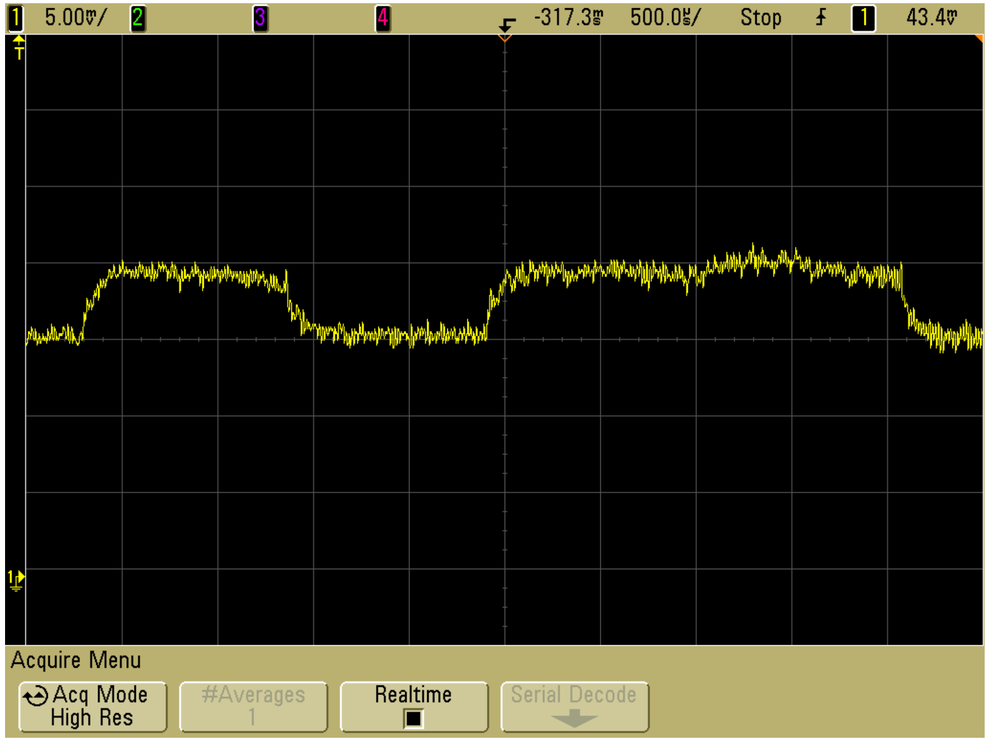


https://www.sjoerdlangkemper.nl/2016/11/01/tenex-password-bug/

Side Channel Examples

Power as a side channel

- Let's say there's a cryptographic secret maintained in hardware
  - Can never be read, only used (e.g., your phone)
- Simple Power Analysis (SPA)
  - Change in power draw can correspond to underlying values or operations
- Differential Power Analysis (DPA)
  - Use signal processing techniques to subtract out noise cause by other activity you aren't interested in



https://en.wikipedia.org/wiki/Power_analysis

Side Channel Examples

Keyboard acoustic emanations

What is the attack the authors want to conduct?

Side Channel Examples

Keyboard acoustic emanations

What is the attack the authors want to conduct?

Covertly or overtly record keystrokes

Side Channel Examples

Keyboard acoustic emanations

What is the attack the authors want to conduct?

Attacker can use keystrokes to recover passwords or other secrets

From: *Keyboard Acoustic Emanations Revisited*

Side Channel Examples

Decrypting RSA keys through sound

Aside: Covert Channels

- Side channels are inadvertent artifacts of the implementation that can be analyzed to extract information across a trust boundary
- **Covert channels** are the same idea, but actually *on purpose*
  - One party is trying to leak information in a way that won't be obvious
  - They *encode* that information into some side channel (e.g., variation in time, memory usage, etc.)
  - Information is extracted on the other end
- <u>These are really hard to implement, but also really hard to protect against.</u>

Rowhammer

How does RAM actually work?

Nothing is safe

- If you load something in memory, how long can you expect it to stay there for?

How does RAM actually work?

Nothing is safe

- If you load something in memory, how long can you expect it to stay there for?
    - What happens if the computer gets unplugged or dies?

How does RAM actually work?

Nothing is safe

- If you load something in memory, how long can you expect it to stay there for?
  - What happens if the computer gets unplugged or dies?
- RAM is what's called volatile memory: data retained only as long as power is on
  - As opposed to persistent (or non-volatile) memory which retains data even without power (e.g., flash, magnetic disks)
- Why do we have RAM in the first place, if it's so *volatile*?

How does RAM actually work?

Nothing is safe

- If you load something in memory, how long can you expect it to stay there for?
  - What happens if the computer gets unplugged or dies?
- RAM is what's called volatile memory: data retained only as long as power is on
  - As opposed to persistent (or non-volatile) memory which retains data even without power (e.g., flash, magnetic disks)
- Why do we have RAM in the first place, if it's so *volatile*?
  - It's fast. We like fast.

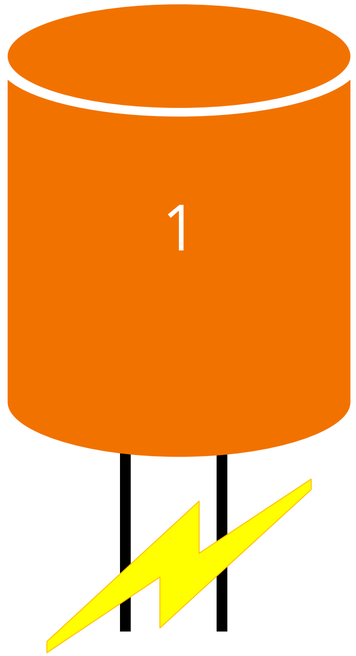
Two types of RAM

SRAM and DRAM

- Static RAM (SRAM) vs. Dynamic RAM (DRAM)
  - SRAM: Retains bit values in memory so long as there is power!
    - Typically faster
    - Lower density (SRAM doesn’t get so big; requires 6 transistors per bit)
    - More expensive
  - DRAM: requires a periodic refresh to maintain a stored value
    - Refresh happens ~64ms
    - Higher capacity
    - Lower cost… DRAM is what’s in all of our machines

How does DRAM work?

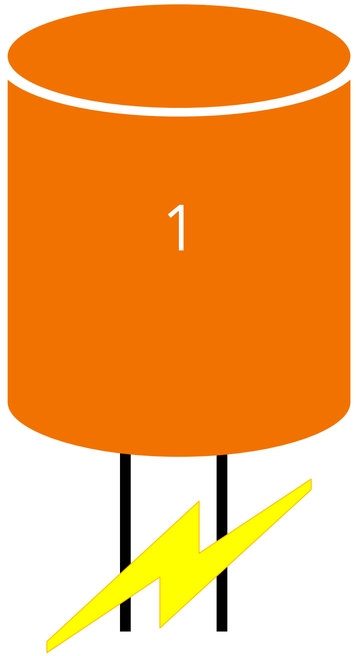
DRAM cells are essentially just capacitors. What is a capacitor?



Write “1”

How does DRAM work?

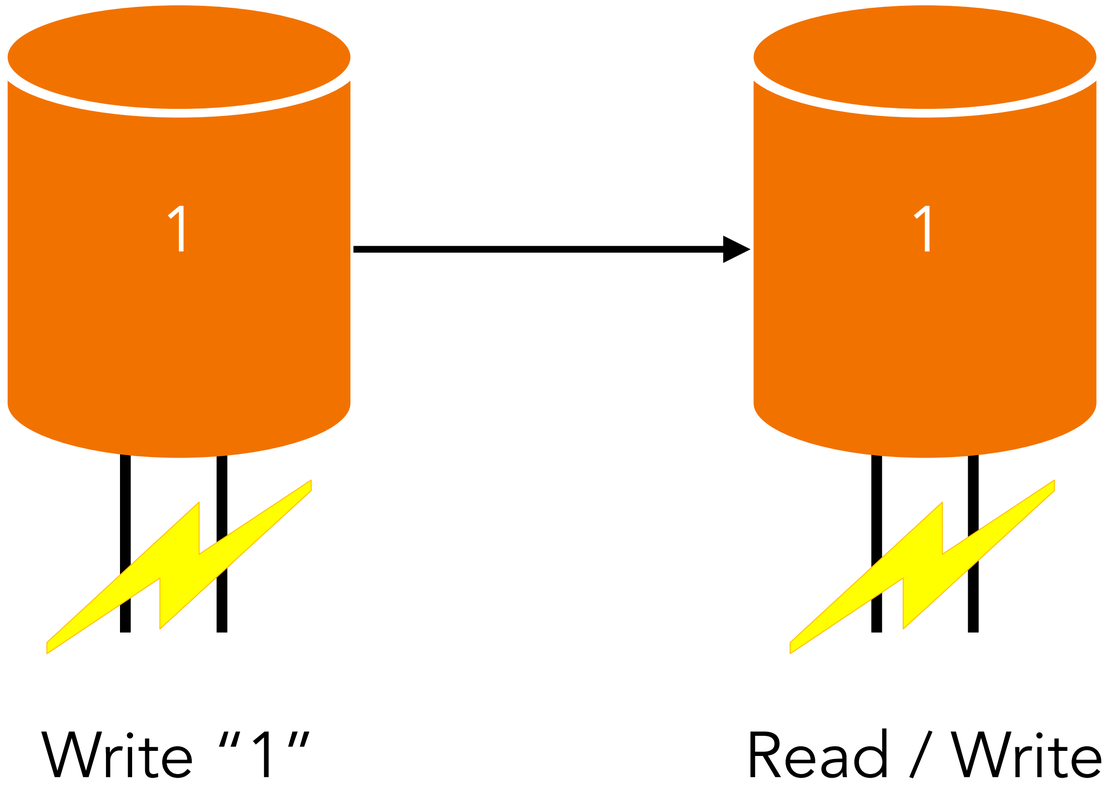
DRAM must be refreshed, so how does that work?



Write “1”

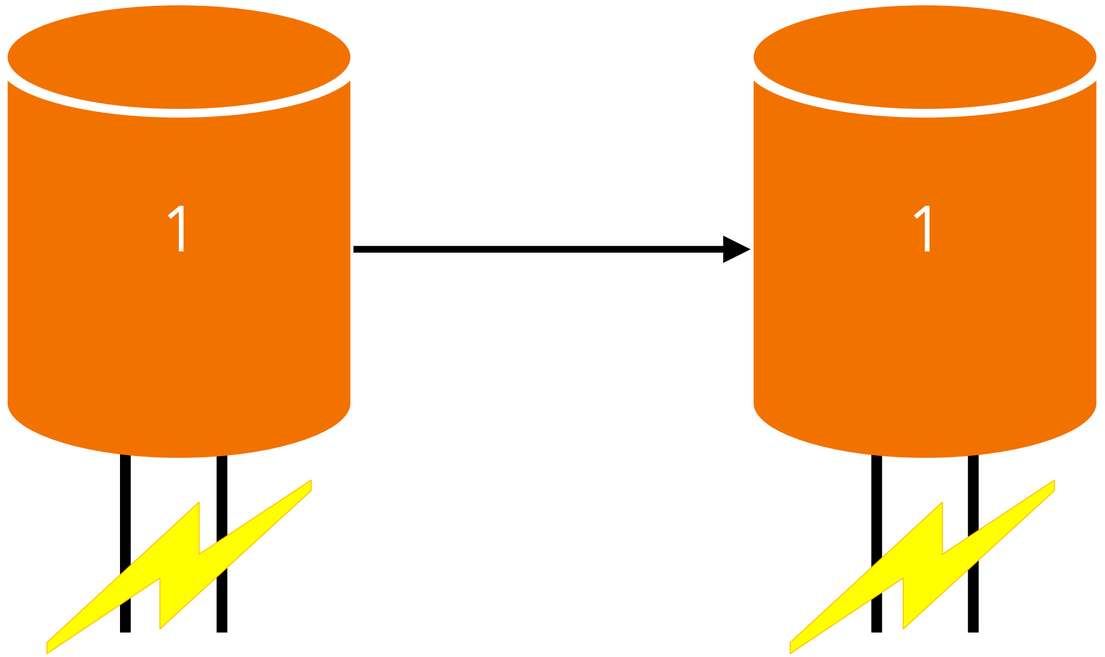
How does DRAM work?

Every refresh period, all cells are read from and then written to



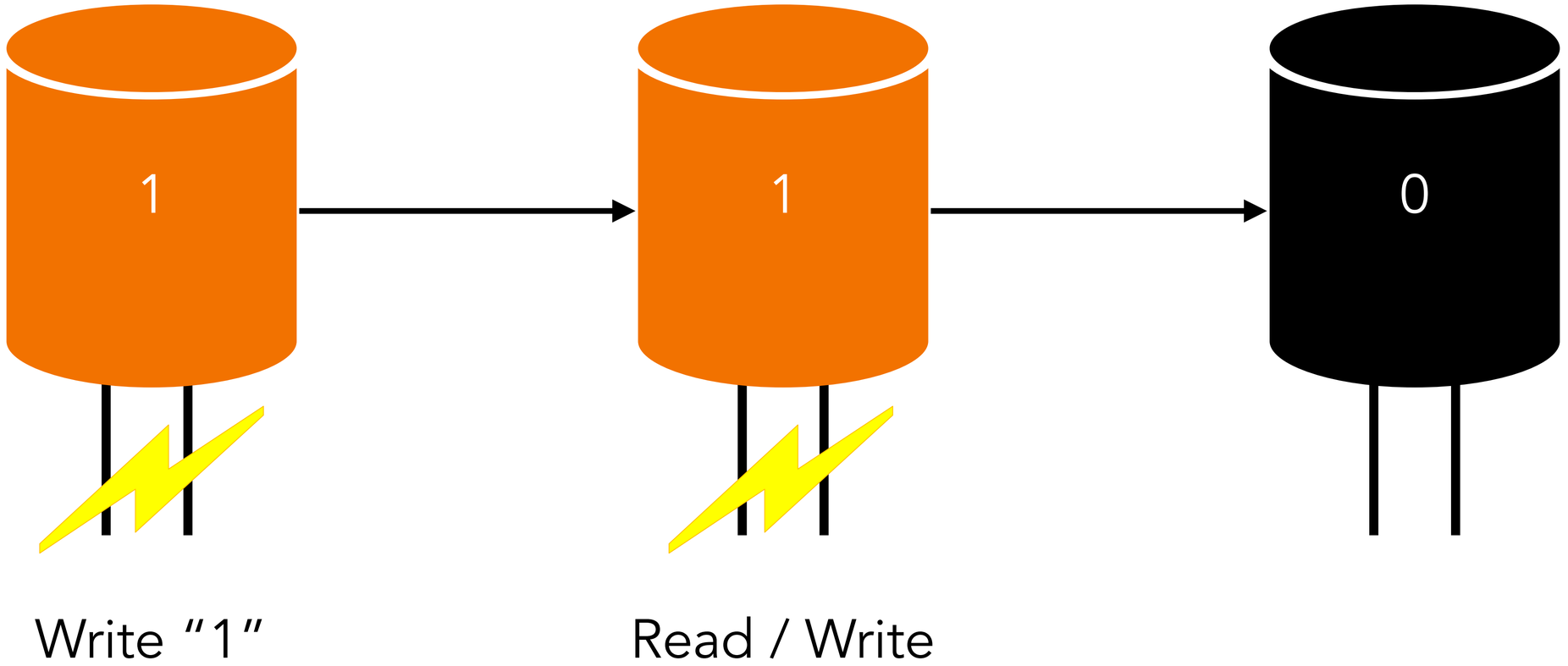
How does DRAM work?

What happens if we don’t refresh the cell?



How does DRAM work?

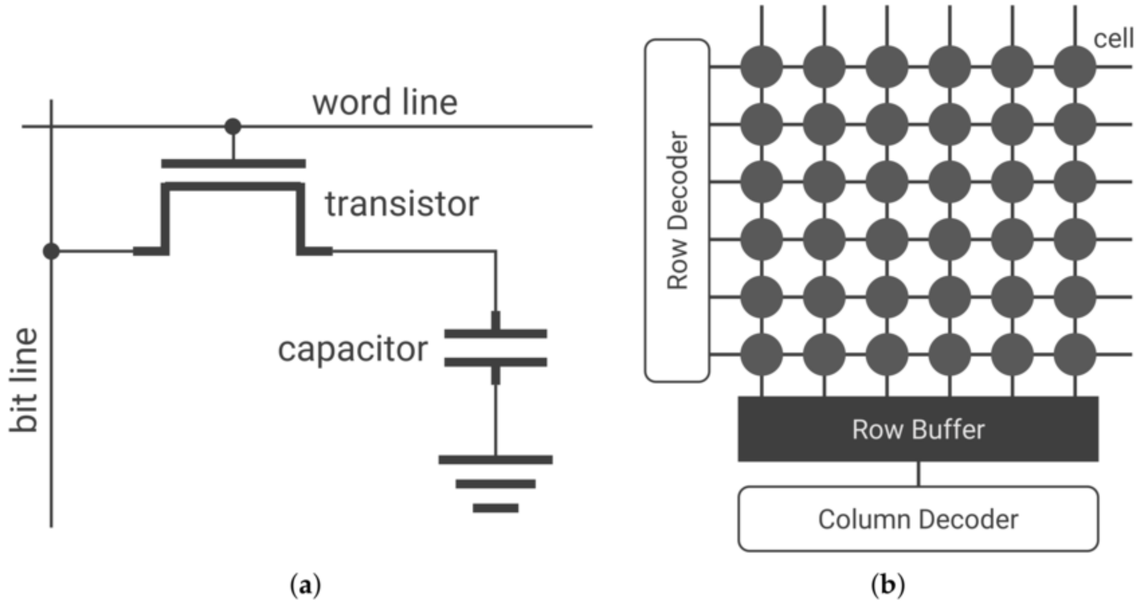
What happens if we don’t refresh the cell?



How is DRAM organized?

Organizing capacitors

- DRAM cells are groups into *rows*
  - ~1KB per row
  - All cells in a row are refreshed together

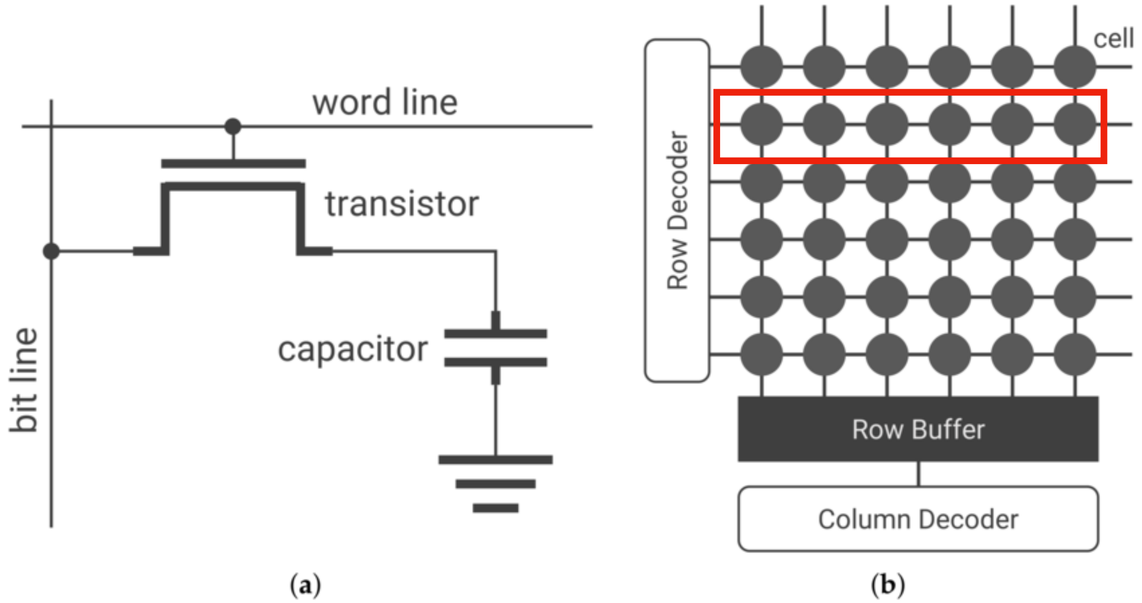


https://www.researchgate.net/figure/Structure-of-dynamic-random-access-memory-DRAM-DRAM-consists-of-cells-that-hold-one_fig1_360496423

How is DRAM organized?

Organizing capacitors

- DRAM cells are groups into *rows*
  - ~1KB per row
  - All cells in a row are refreshed together
- To read a single bit, we read the row into a "row buffer" and index

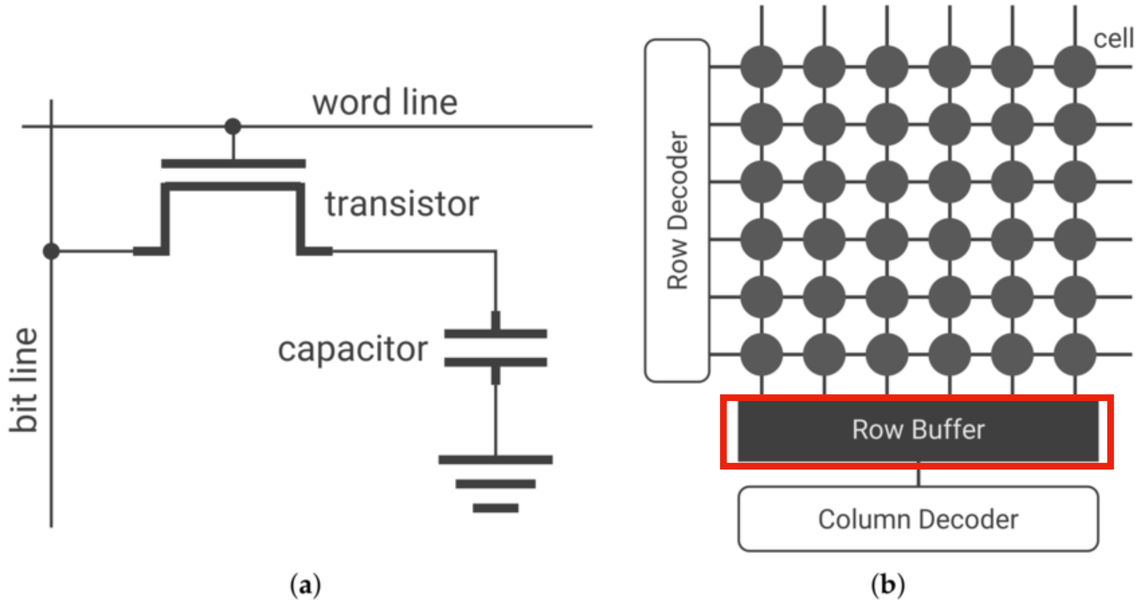


https://www.researchgate.net/figure/Structure-of-dynamic-random-access-memory-DRAM-DRAM-consists-of-cells-that-hold-one_fig1_360496423

How is DRAM organized?

Organizing capacitors

- DRAM cells are groups into *rows*
  - ~1KB per row
  - All cells in a row are refreshed together
- To read a single bit, we read the row into a "row buffer" and index

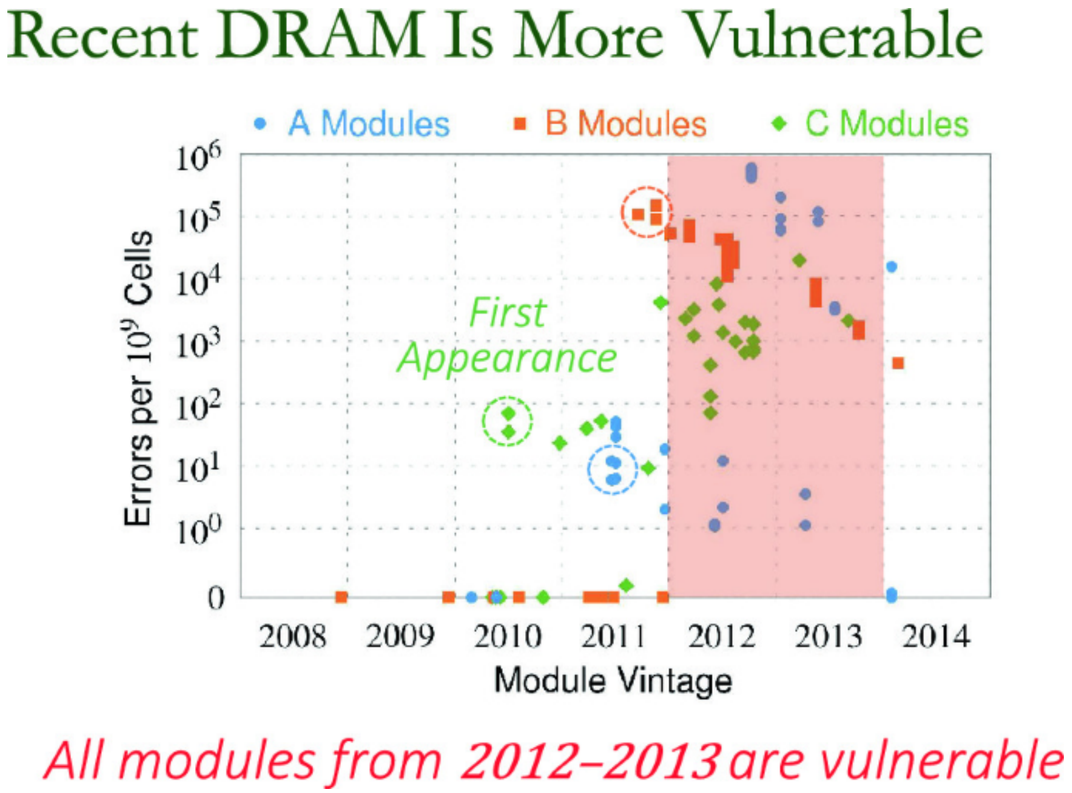


https://www.researchgate.net/figure/Structure-of-dynamic-random-access-memory-DRAM-DRAM-consists-of-cells-that-hold-one_fig1_360496423

DRAM reliability

DRAM getting less reliable over time

- As DRAM has gotten more and more dense (4GB —> 32GB over roughly same area footprint)… many issues of reliability
- Graph on right shows errors over time. Why do these errors occur?

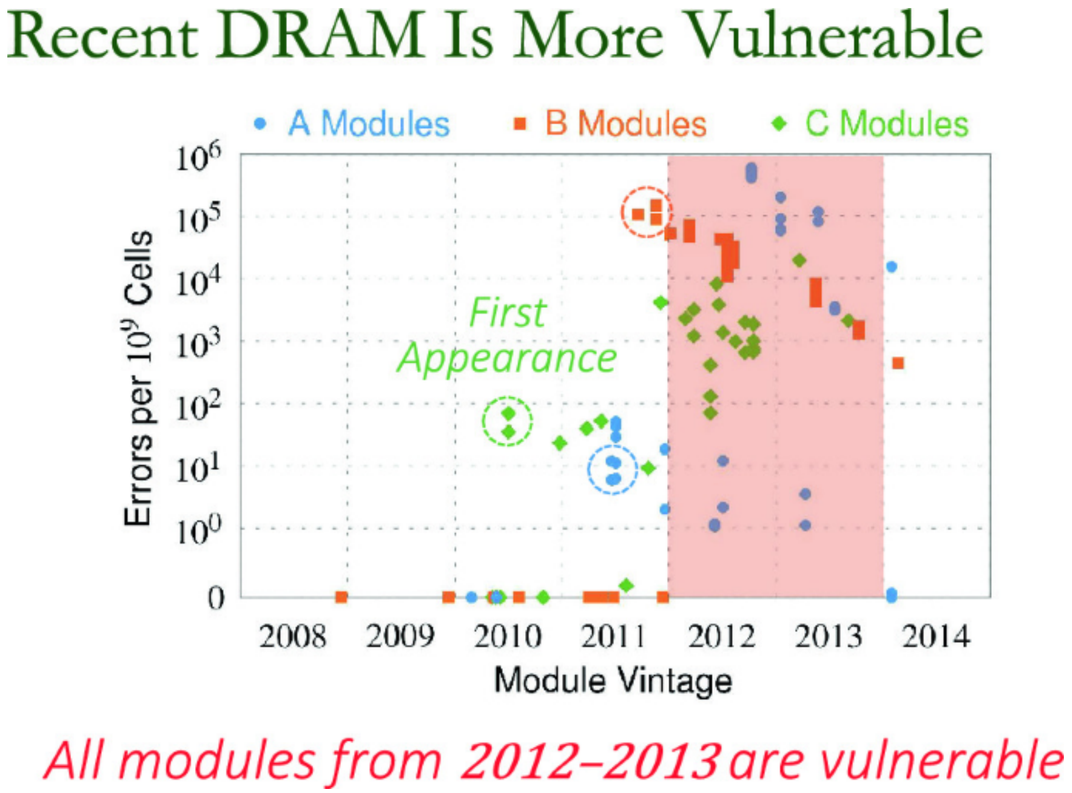


https://www.researchgate.net/figure/Structure-of-dynamic-random-access-memory-DRAM-DRAM-consists-of-cells-that-hold-one_fig1_360496423

DRAM reliability

DRAM getting less reliable over time

- As DRAM has gotten more and more dense (4GB —> 32GB over roughly same area footprint)… many issues of reliability
- Graph on right shows errors over time. Why do these errors occur?
- As rows get closer and closer together, *sending power to a row* has a nontrivial probability of "leaking" charge to nearby rows, potentially flipping bits

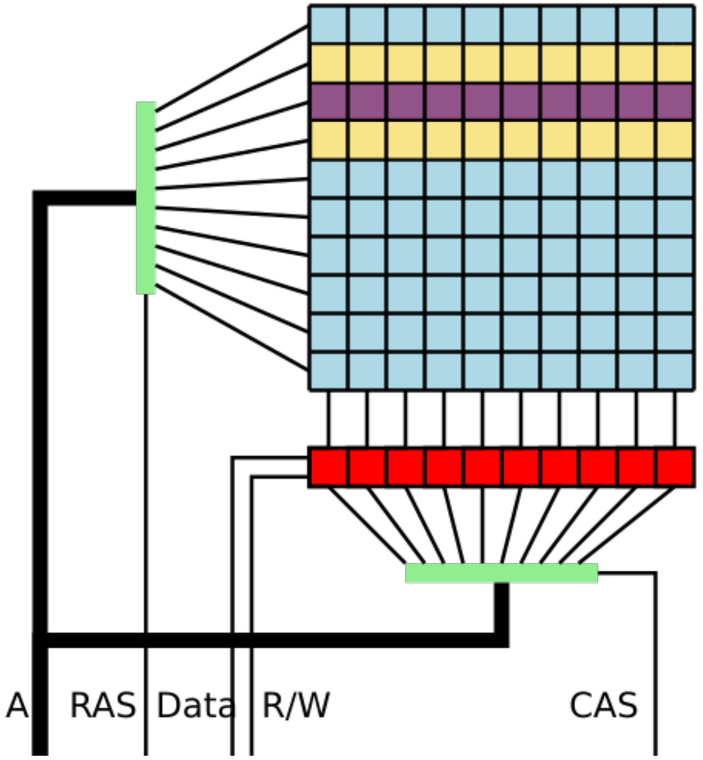


https://www.researchgate.net/figure/Structure-of-dynamic-random-access-memory-DRAM-DRAM-consists-of-cells-that-hold-one_fig1_360496423

Enter: Rowhammer

Make your own bit flips

- Basic idea: *induce bit flips in between refresh periods by hammering memory lines that sandwich important pages*
  - Identify a target row (in this case, the purple row)
  - Sandwich it between two rows you control (e.g., page in everything else yourself, but more clever ways to do this too)
  - Repeatedly read from rows you control (forcing power through the row)

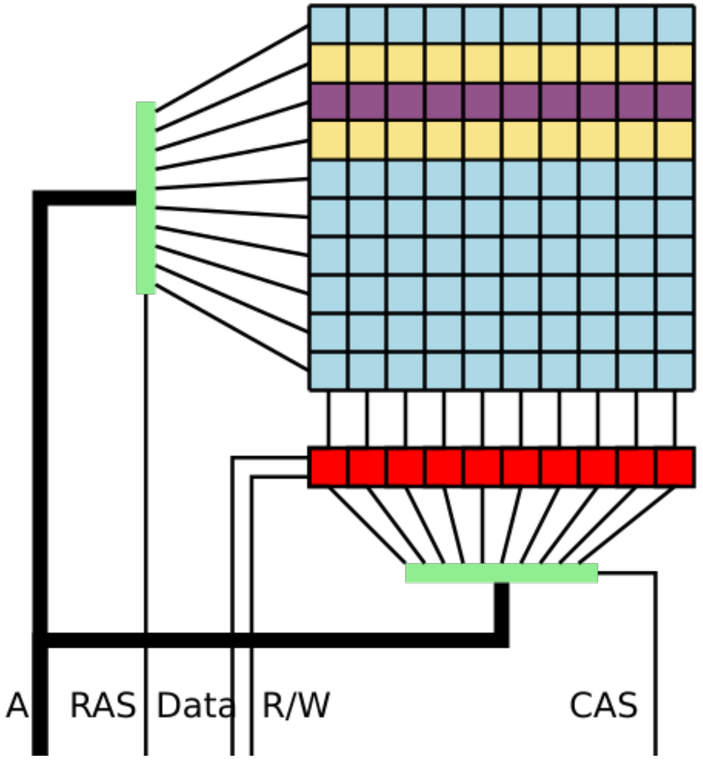



```
hammer:
  mov (X), %eax  // read from address X
  mov (Y), %ebx  // read from address Y
  clflush (X)    // flush cache for address
X
  clflush (Y)    // flush cache for address
Y
  jmp hammer
```

Rowhammer attack model

Threat model

- Attacker code is executing on same machine as the victim, but with *less privileges*
  - When might this happen in practice?

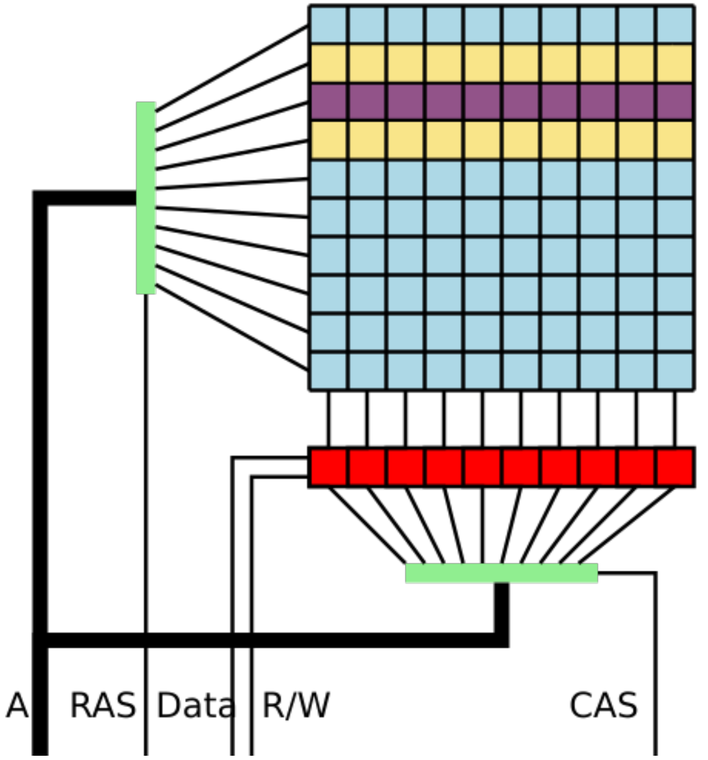



```
hammer:
  mov (X), %eax  // read from address X
  mov (Y), %ebx  // read from address Y
  clflush (X)    // flush cache for address
X
  clflush (Y)    // flush cache for address
Y
  jmp hammer
```

Rowhammer attack model

Threat model

- Attacker code is executing on same machine as the victim, but with *less privileges*
  - When might this happen in practice?
- All the time!
  - Userland attacking kernel
  - JavaScript attacking browser
  - Guest OS on host OS




```
hammer:
  mov (X), %eax  // read from address X
  mov (Y), %ebx  // read from address Y
  clflush (X)    // flush cache for address
X
  clflush (Y)    // flush cache for address
Y
  jmp hammer
```

Who cares?

Some bits are important

Exploiting the DRAM rowhammer bug to gain kernel privileges

How to cause and exploit
single bit errors

Mark Seaborn and Thomas Dullien

Who cares?

Some bits are important

x86-64 Page Table Entries (PTEs)

- Page table is a 4k page containing array of 512 PTEs
- Each PTE is 64 bits, containing:

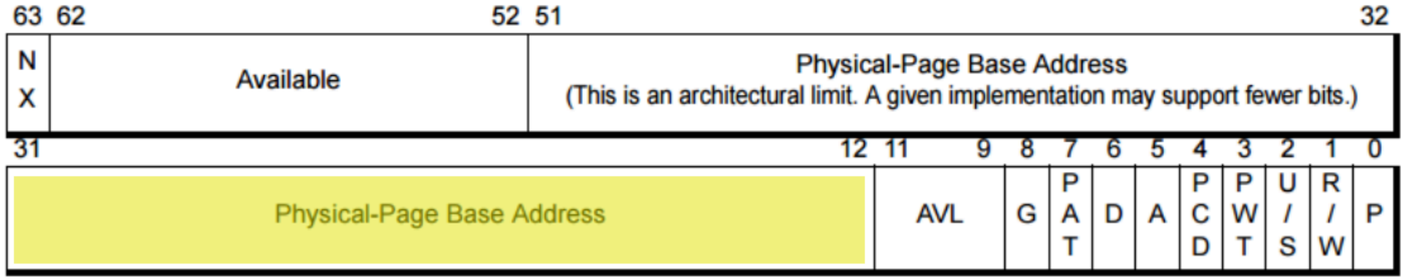


Figure 5-21. 4-Kbyte PTE—Long Mode

- Could flip:
  - "Writable" permission bit (RW): 1 bit → 2% chance
  - Physical page number: 20 bits on 4GB system → 31% chance

What do we do about rowhammer?

- ECC memory
  - Compute error correcting code on write, check on read
  - Significant mitigation to rowhammer attacks, but still, some attacks will work
  - Somewhat costly to do this check
- Memory controller limitations on “hammering” or additional adjacent line refresh
  - Memory controller needs to keep state; could have impact
- Issue still persists to this day.

Cache Side Channels

What is a CPU cache?

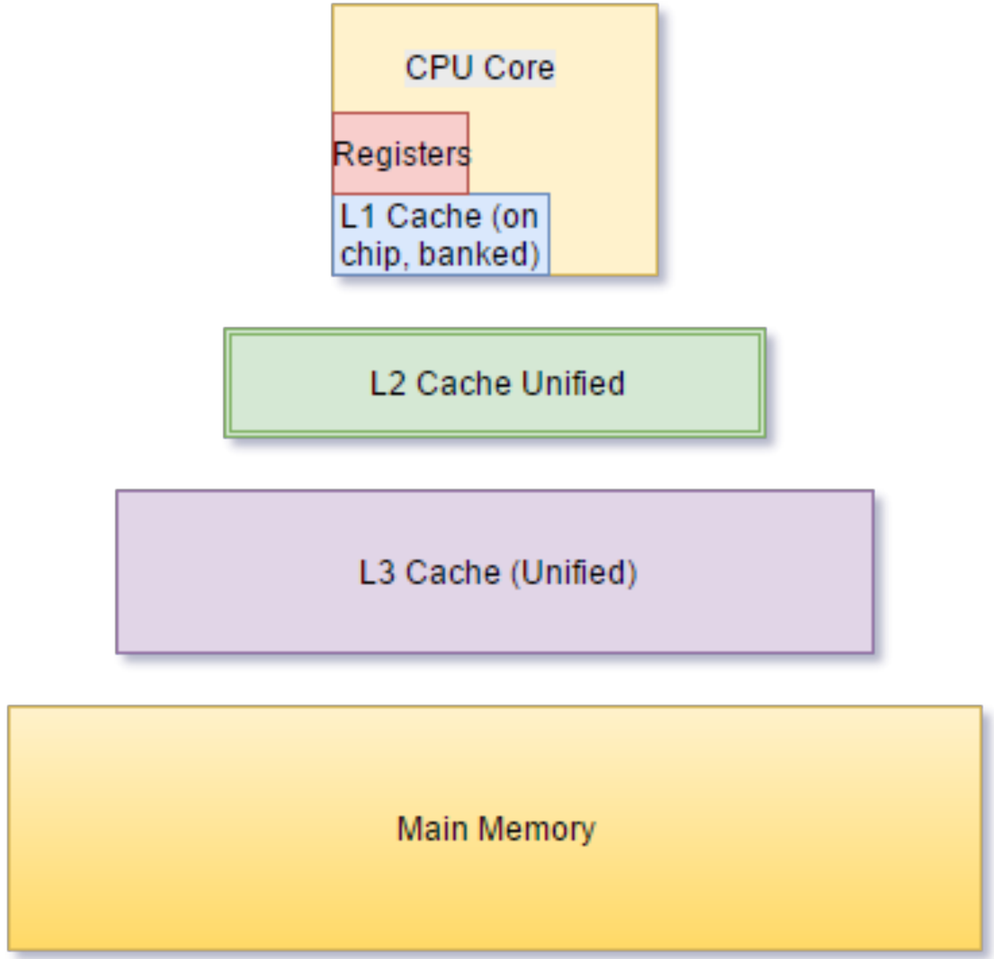
What is a CPU cache?

- Main memory is dense (high capacity) … but slow
  - 1 – 4 clock cycles for cache read
  - Hundreds of clock cycles for memory read
- Processors will try to “cache” recently used memory
  - Cache typically implemented at SRAM (notably much smaller capacity than DRAM)

Cache Hierarchy

Caches all the way down

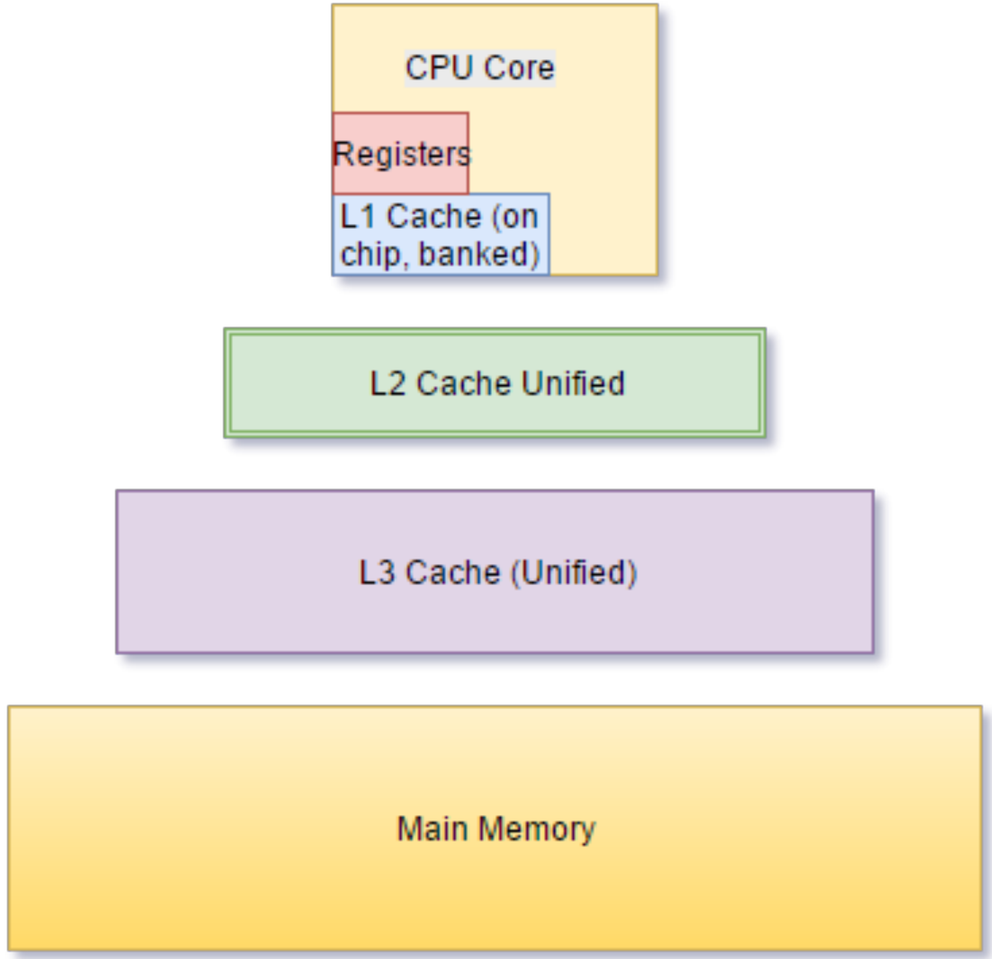
- There are multiple layers of caches…
  - L1 (on chip), L2, L3; each increasing in size but slightly decreasing in speed
  - If it's not in L1, L2, or L3, then we go to DRAM



Cache Hierarchy

Caches all the way down

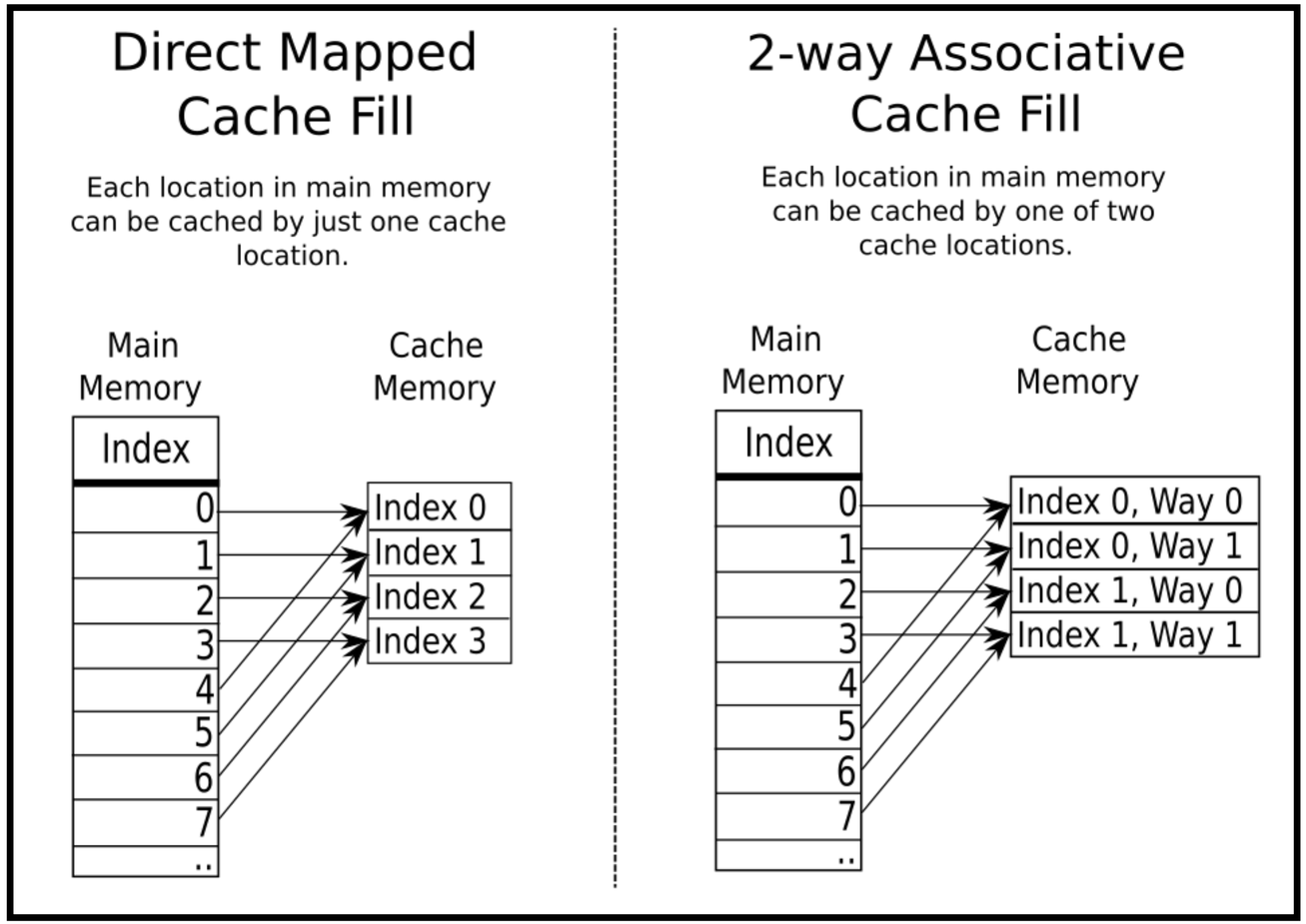
- There are multiple layers of caches…
  - L1 (on chip), L2, L3; each increasing in size but slightly decreasing in speed
  - If it's not in L1, L2, or L3, then we go to DRAM
- Note the cache is a **shared** system resource
  - "Just a performance optimization" —> has no impact on reliability… but it does change ***time***



Cache Organization

How do caches *actually* work?

- Cache line
  - Unit of cache granularity, e.g., 64 bytes
  - Smallest unit of putting something “in the cache”
- Set associativity
  - Cache lines are grouped into sets
  - Each memory address is mapped to a set of cache lines; (associativity), reducing potential cache misses without causing too much eviction



https://en.wikipedia.org/wiki/CPU_cache

Cache Side Channel Attacks

- Threat model:
  - We want to protect victim memory
  - Attacker and victim are two different execution domains (e.g., processes or privilege levels) on the same physical system
  - Attacker is able to invoke (directly or indirectly) functionality exposed by the victim
    - Sometimes with attacker-supplied parameters

Attacker capabilities

- Prime
  - Place a known address in the cache. How?

Attacker capabilities

- Prime
  - Place a known address in the cache. How?
- Evict
  - Remove something from the cache. How?

Attacker capabilities

- Prime
  - Place a known address in the cache. How?
- Evict
  - Remove something from the cache. How?
- Flush
  - Remove a given address from the cache (`cflush` on x86)
- Measure
  - Identify how long it takes to do something

Cache Side Channel Attack Strategy

1. Manipulate cache into a known state
2. Make victim run
3. Try to infer what has changed in the cache as a result of victim code running

Cache Side Channel Attack Strategy

- Three basic techniques…
  - **Evict & Time**
    - Kick stuff out of the cache and see if the victim slows down as a result
  - **Prime & Probe**
    - Put stuff in the cache, run the victim and see if accesses are still fast (no conflict) or slowed down (have been displaced by memory accesses)
  - **Flush & Reload**
    - Flush a particular line from the cache, run the victim and see if your access are still fast as a result

Evict & Time

- Run the victim code several times and time it
  - Get a baseline
- Evict (portions of) the cache
  - Access conflicting memory locations so previous cache contents are replaced with recently-accessed data
- Run the victim code again and retime it
- If it is slower than before, cache lines evicted by the attacker *must have been used by the victim*
  - We now know something about the addresses used by victim code

Prime & Probe

- Prime the cache
  - Access may memory locations (covering all cache lines of interest) so previous cache contents are replaced with attacker addresses
  - Time access to each cache line to establish speed for “in cache” references
- Run victim code
- Attacker retimes access to its own memory locations
  - If any are slower, it means the corresponding cache line was used by the victim
  - We again, know something now about addresses used by the victim

Flush & Reload

- Specifically, for *shared memory*
  - E.g., shared libraries, fork() sharing, deduplication in VMs
- Time memory access to (potentially) shared regions
- Flush specific lines from the cache
- Invoke victim code
- Retime access to flushed addresses, if still fast, then was used by the victim
  - Because we flushed it, it should be slow, unless the victim reloaded it
  - Again, addresses used by the victim.

Things to note about these attacks

- The error on any individual measurement is high
  - Repeat many times and use central tendency statistics (e.g., medians) to filter outliers
- Do they… work?
  - *"Our error-correcting and error-handling high-throughput covert channel can sustain transmission rates of more than 45 KBps on Amazon EC2."*

Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud

Meltdown

What does Meltdown enable?

Meltdown: Reading Kernel Memory from User Space

Moritz Lipp[1], Michael Schwarz[1], Daniel Gruss[1], Thomas Prescher[2],
Werner Haas[2], Anders Fogh[3], Jann Horn[4], Stefan Mangard[1],
Paul Kocher[5], Daniel Genkin[6,9], Yuval Yarom[7], Mike Hamburg[8]
[1]*Graz University of Technology,* [2]*Cyberus Technology GmbH,*
[3]*G-Data Advanced Analytics,* [4]*Google Project Zero,*
[5]*Independent (www.paulkocher.com),* [6]*University of Michigan,*
[7]*University of Adelaide & Data61,* [8]*Rambus, Cryptography Research Division*

Recall from last time…

- In order to avoid context switching for syscalls, kernel virtual memory is mapped onto every process
  - Otherwise, system calls would take forever, lots of context switching, etc.
- Remember isolation guarantees:
  - Page table access control ensures kernel pages are only read when processor register is in privileged mode!

Abstraction vs. Implementation in Architecture

- Instruction Set Architecture (ISA)
  - Defines interface between hardware and software… “in a perfect world”
- Microarchitecture is the *implementation* of the ISA on a chip
- Oftentimes, there are subtle differences in implementation and interface, leading to what are called **microarchitectural side channel attacks.**

Abstraction vs. Implementation in Architecture

- Instruction Set Architecture (ISA)
  - Defines interface between hardware and software… “in a perfect world”
- Microarchitecture is the *implementation* of the ISA on a chip
- Oftentimes, there are subtle differences in implementation and interface, leading to what are called **microarchitectural side channel attacks.**
  - Meltdown (and Spectre) are such attacks.

Example: Instruction Pipelining

- Processors often break up instructions into smaller parts so parts can be processed in parallel
- Instructions appear to be executed one at a time and in order
  - But under the hood, dependencies get resolved through pipelining effects

Instr. No. \ Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

https://en.wikipedia.org/wiki/Instruction_pipelining

Example: Out-of-Order Execution

- Sometimes, instructions can be safely executed out of order
  - Avoid unnecessary pipeline stalls
- But, architecturally, it appears instructions are executed in order

Another example: Speculative Execution

- Sometimes control flow depends on output of an earlier instruction
  - E.g., conditional branch, function pointer
- Rather than wait to know for sure which way to go, the processor may ***speculate*** about the direction/target of a branch
  - **Guess** based on the past
  - If guess is correct, performance is improved
  - If guess is wrong, speculated computation is discarded and everythign is re-computed using the correct value
- No impact on correctness… so what's the issue?

Microarchitectural side effects can leak privileged information.

Meltdown

Speculative execution to our downfall

```
if (x < array.length()) {

    value = array[x];

}
```

Meltdown

Speculative execution to our downfall

```
if (x < array.length()) {
    value = array[x];
}
```

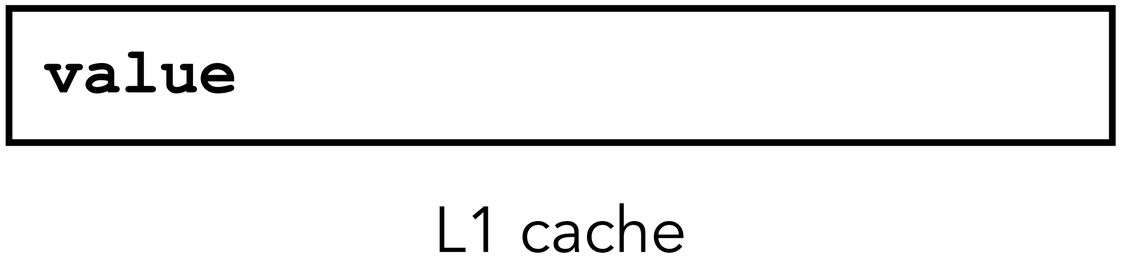
- Checking array length can take some time… so processor will speculatively fetch **value**

Meltdown

Speculative execution to our downfall

```
if (x < array.length()) {
    value = array[x];
}
```

- Checking array length can take some time… so processor will speculatively fetch **value**
- Will store that value in the cache



Meltdown

Speculative execution to our downfall

```
if (x < array.length()) {
    value = array[x];
}
```

L1 cache

- Checking array length can take some time… so processor will speculatively fetch **value**
- Will store that value in the cache
- If `x` ends up larger than the array length, we obviously don't take the branch
  - But… value has been loaded into the cache
  - **No privilege checks happen because the branch is not taken.**
- What happens if `array[x]` is protected memory?

Building blocks of Meltdown

- Out of order instructions that have microarchitectural side effects are called **transient instructions**

- Through side channels, we can *read arbitrary memory* from kernel without even exploiting any bugs!

Meltdown to read kernel memory

- What does the following code do?

```
*(char *) 0;

array[0] = 0;
```

Meltdown to read kernel memory

- What does the following code do?

```
*(char *) 0;

array[0] = 0;
```

- Null pointer dereference! Should throw an exception right away. Except…

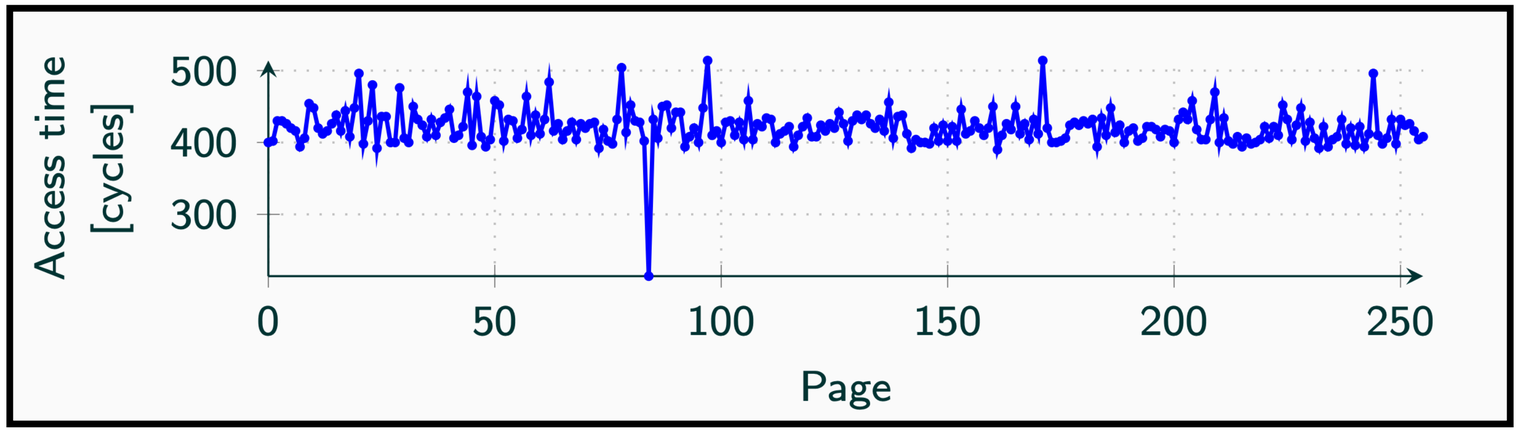
Meltdown to read kernel memory

- What does the following code do?

```
*(char *) 0;

array[0] = 0;
```

- Null pointer dereference! Should throw an exception right away. Except…



- Somehow, array[0] was cached, and the privilege check doesn't happen until some time later… uh oh

Meltdown — it’s a huge problem

- Goal: read one byte of kernel memory
  - Basic idea: Combine speculative execution + flush & reload + late privilege check

```
char data = *(char*)0xffff00e0

array[data * 4096] = 0;
```


Example adapted from Security & Privacy Academy

Meltdown — it's a huge problem

- Goal: read one byte of kernel memory
  - Basic idea: Combine speculative execution + flush & reload + late privilege check
- This line reads a byte from an address in memory and stores it in `data` (putting it in cache)

```
char data = *(char*)0xffff00e0

array[data * 4096] = 0;
```


Example adapted from Security & Privacy Academy

Meltdown — it's a huge problem

- Goal: read one byte of kernel memory
  - Basic idea: Combine speculative execution + flush & reload + late privilege check
- This line reads a byte from an address in memory and stores it in `data` (putting it in cache)
- This line uses that byte as an index in a big array

```
char data = *(char*)0xffff00e0

array[data * 4096] = 0;
```


Example adapted from Security & Privacy Academy

Meltdown — it's a huge problem

- Goal: read one byte of kernel memory
  - Basic idea: Combine speculative execution + flush & reload + late privilege check
- This line reads a byte from an address in memory and stores it in `data` (putting it in cache)
- This line uses that byte as an index in a big array
- Sweep over all pages of array

```
char data = *(char*)0xffff00e0

array[data * 4096] = 0;
```

Meltdown — it's a huge problem

- Goal: read one byte of kernel memory
  - Basic idea: Combine speculative execution + flush & reload + late privilege check
- This line reads a byte from an address in memory and stores it in `data` (putting it in cache)
- This line uses that byte as an index in a big array
- Flush + reload all pages of the array

```
char data = *(char*)0xffff00e0

array[data * 4096] = 0;
```

Is data = 0?

Meltdown — it's a huge problem

- Goal: read one byte of kernel memory
  - Basic idea: Combine speculative execution + flush & reload + late privilege check
- This line reads a byte from an address in memory and stores it in `data` (putting it in cache)
- This line uses that byte as an index in a big array
- Flush + reload all pages of the array

```
char data = *(char*)0xffff00e0

array[data * 4096] = 0;
```

Is data = 1?

Meltdown — it's a huge problem

- Goal: read one byte of kernel memory
  - Basic idea: Combine speculative execution + flush & reload + late privilege check
- This line reads a byte from an address in memory and stores it in `data` (putting it in cache)
- This line uses that byte as an index in a big array
- Flush + reload all pages of the array

```
char data = *(char*)0xffff00e0

array[data * 4096] = 0;
```

Is data = 2?

Meltdown — it's a huge problem

- Goal: read one byte of kernel memory
  - Basic idea: Combine speculative execution + flush & reload + late privilege check
- This line reads a byte from an address in memory and stores it in `data` (putting it in cache)
- This line uses that byte as an index in a big array
- Flush + reload all pages of the array
- If you do this fast enough…

Meltdown enables reading arbitrary kernel memory from any userland process.

Meltdown Bug

- Using out-of-order execution, attacker can read any data at any address
- Privilege checks for the kernel are sometimes too slow to stop this
- Kernel memory is leaked
- Entire physical memory is typically also accessible in kernel space… meaning you can potentially leak other processes private memory as well

Meltdown Demo (embed was not working)

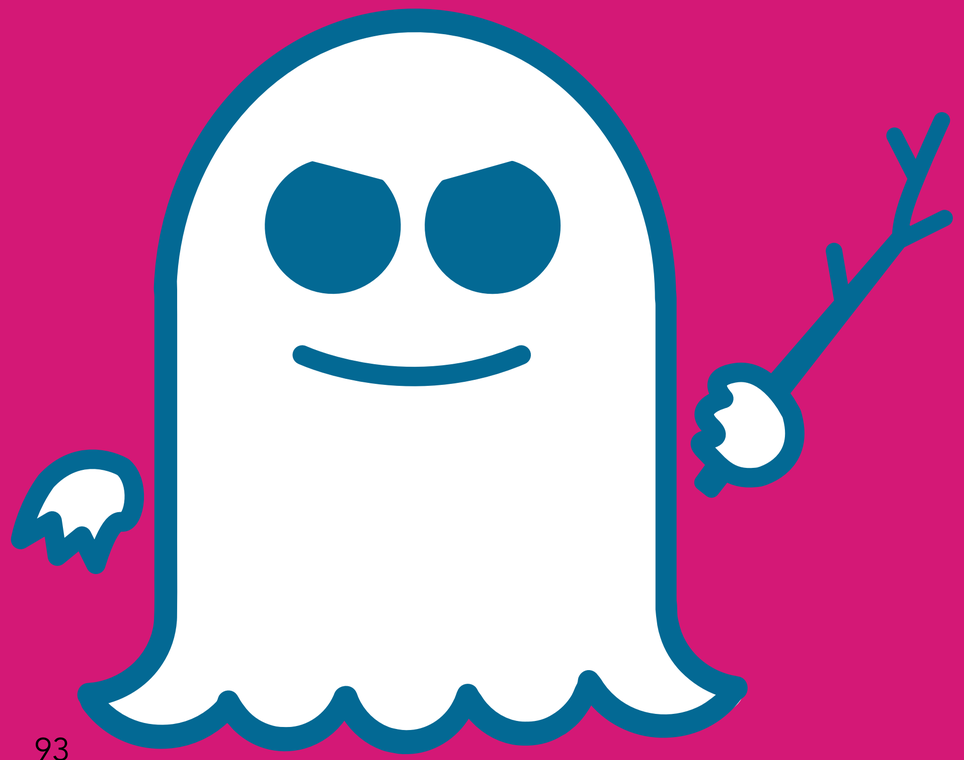
https://www.youtube.com/watch?v=RbHbFkh6eeE

What do we do about Meltdown?

- Ideas?

What do we do about Meltdown?

- Ideas?
- Immediate software only fix: stop mapping kernel memory pages in user processes
  - But this would make programs so slow…
- Instead, we do something called **Kernel Page Table Isolation**
  - Key idea: Map kernel memory in userspace still, but **page everything out** unless you're running in kernel mode
  - 5% — 30% slowdown for most workload
- Newer processes have a microarchitectural fix where the privilege check happens fast, but many are still vulnerable



Spectre

Branch Prediction

- Branch prediction is a feature in CPUs that tries to predict pathways taken to conduct efficient speculative execution

Branch Prediction

- Branch prediction is a feature in CPUs that tries to predict pathways taken to conduct efficient speculative execution

```
for (i = 0; i < 500; i++){
    array[i] = 'h'
}
array2[0] = 0;
```

Branch Prediction

- Branch prediction is a feature in CPUs that tries to predict pathways taken to conduct efficient speculative execution

```
for (i = 0; i < 500; i++){
    array[i] = 'h'
}
array2[0] = 0;
```

- Branch predictor would be pretty damn sure `array[i]` is going to be used; speculatively execute that line instead of the next one

Spectre — it gets worse

- Combines cache side channels + branch prediction for arbitrary memory reading

- Can break the **process invariant —** e.g., malicious processes can trick the CPU into loading *other process memory* into the cache, and leaks that memory voluntarily

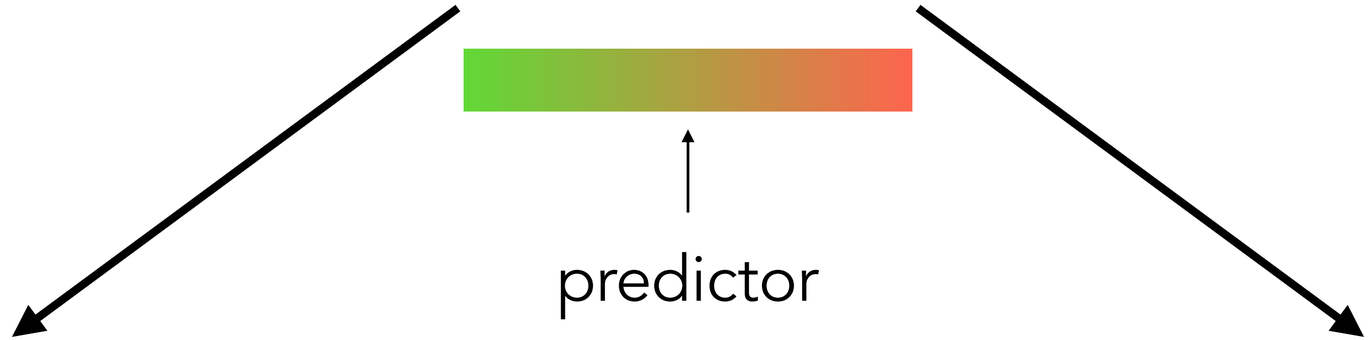
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 0;

char* data = "textKEY";

if (index < 4)
```




```
arr[data[index] * 4096]                    0
```

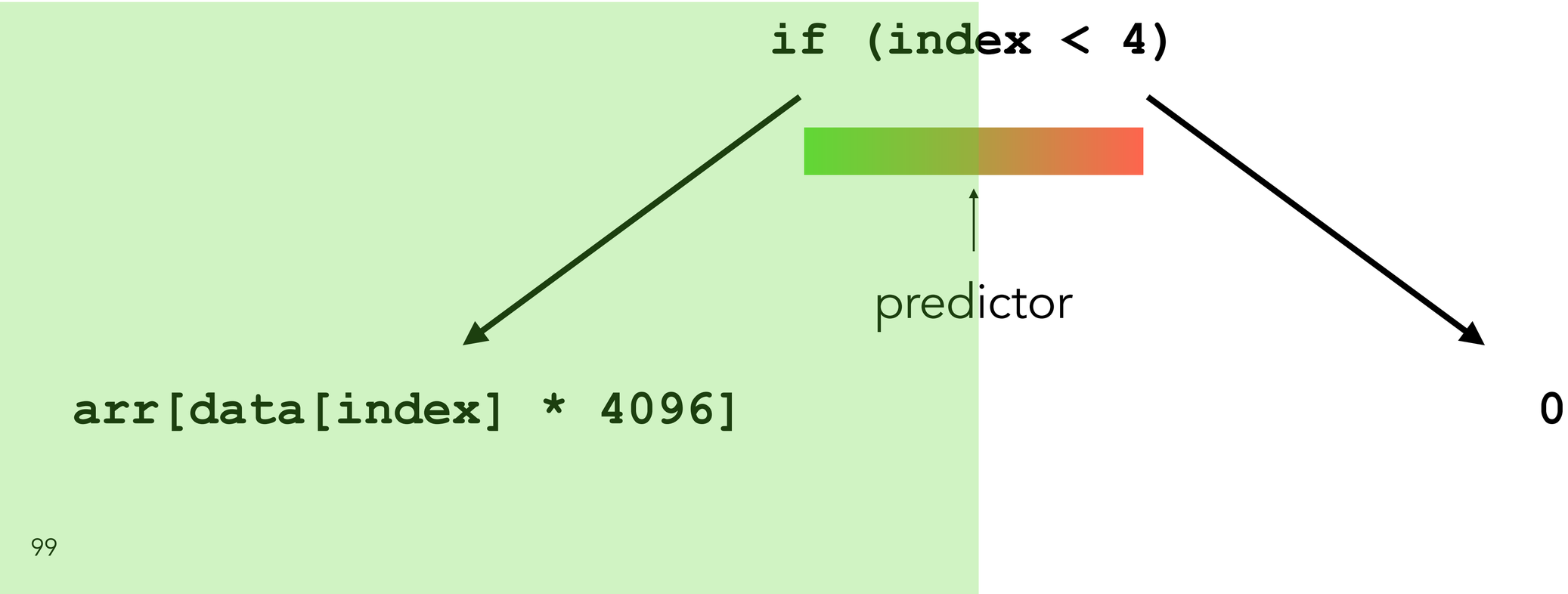
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 0;

char* data = "textKEY";

if (index < 4)
```



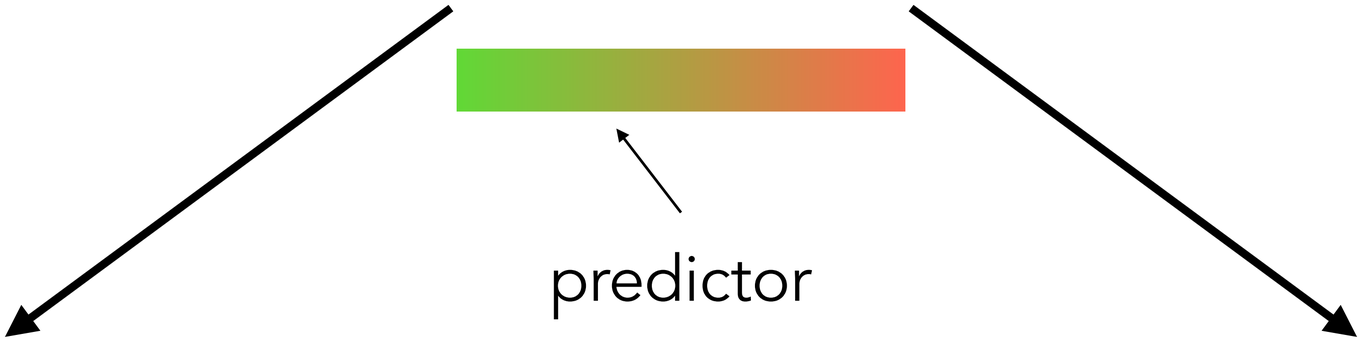
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 0;

char* data = "textKEY";

if (index < 4)
```




```
arr[data[index] * 4096]                    0
```

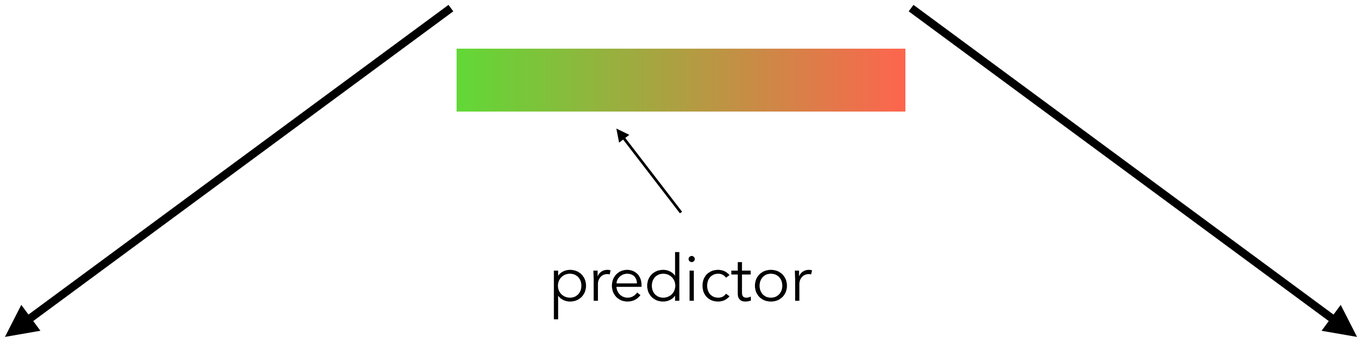
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 1;

char* data = "textKEY";

if (index < 4)
```




```
arr[data[index] * 4096]                    0
```

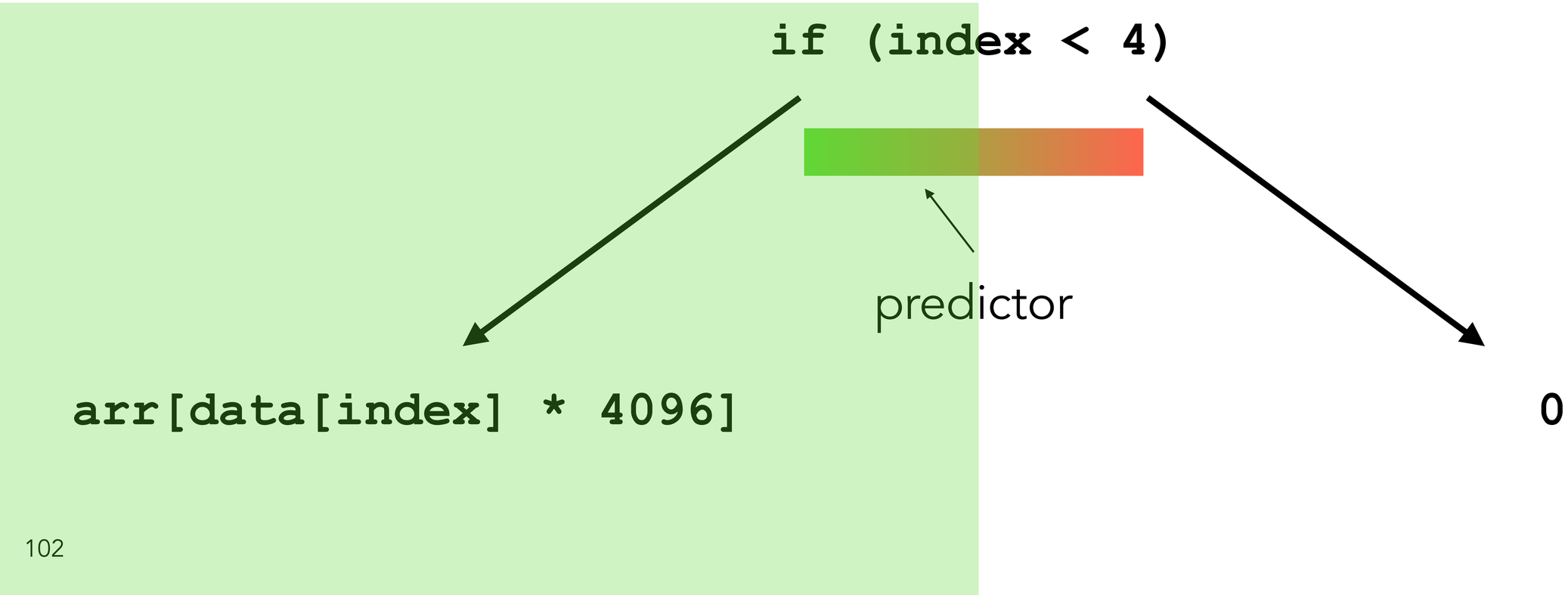
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 1;

char* data = "textKEY";

if (index < 4)
```



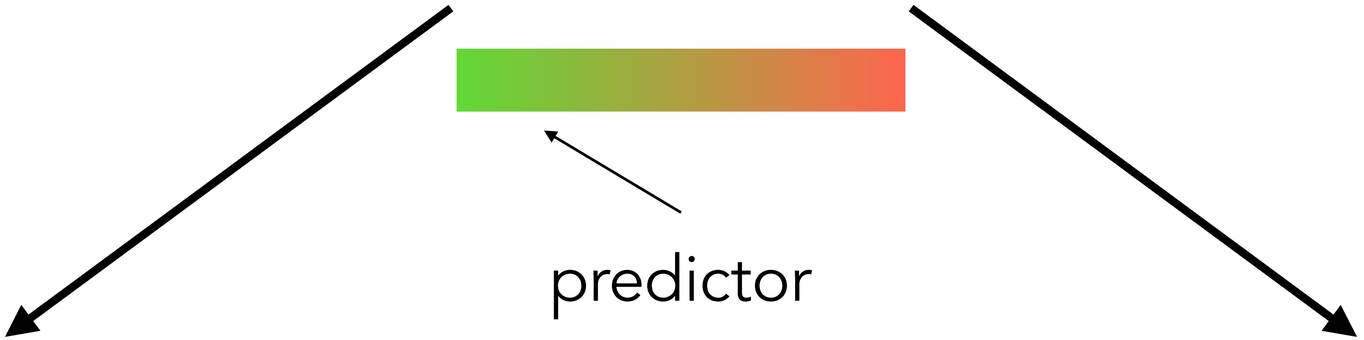
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 2;

char* data = "textKEY";

if (index < 4)
```




```
arr[data[index] * 4096]                    0
```

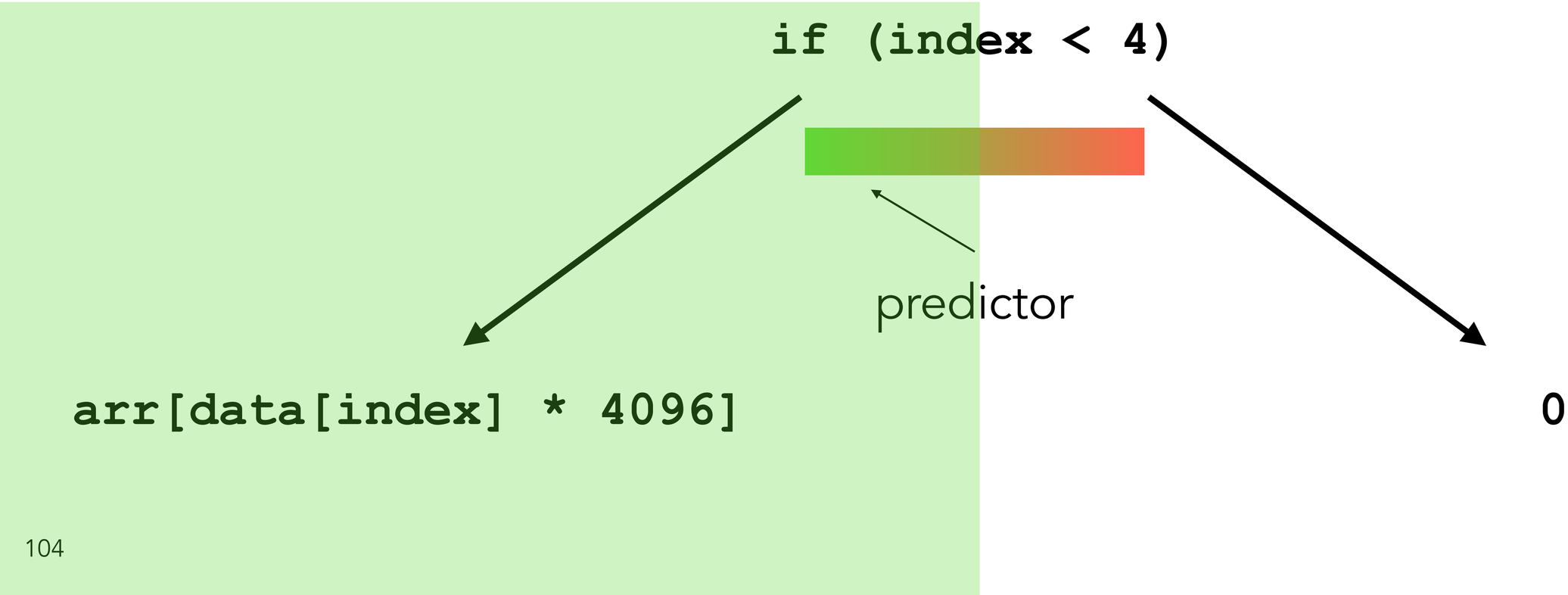
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 2;

char* data = "textKEY";

if (index < 4)
```



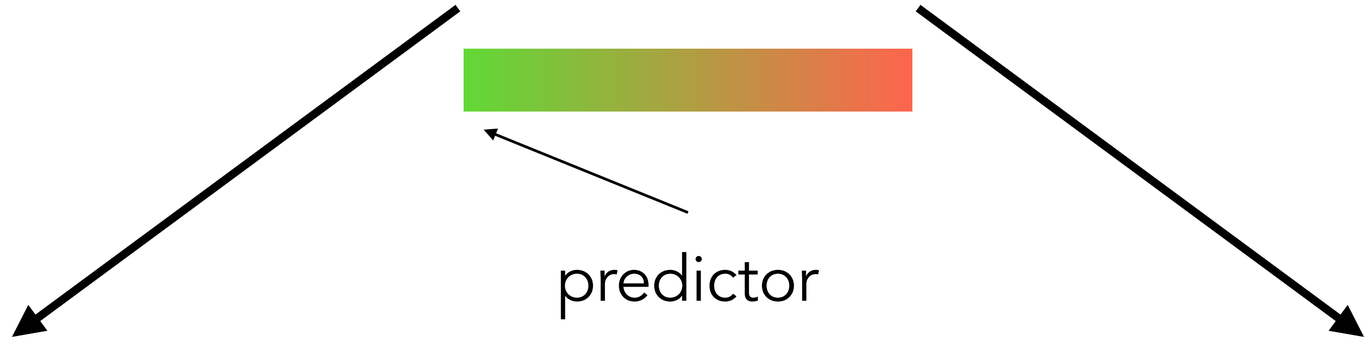
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 3;

char* data = "textKEY";

if (index < 4)
```




```
arr[data[index] * 4096]                    0
```

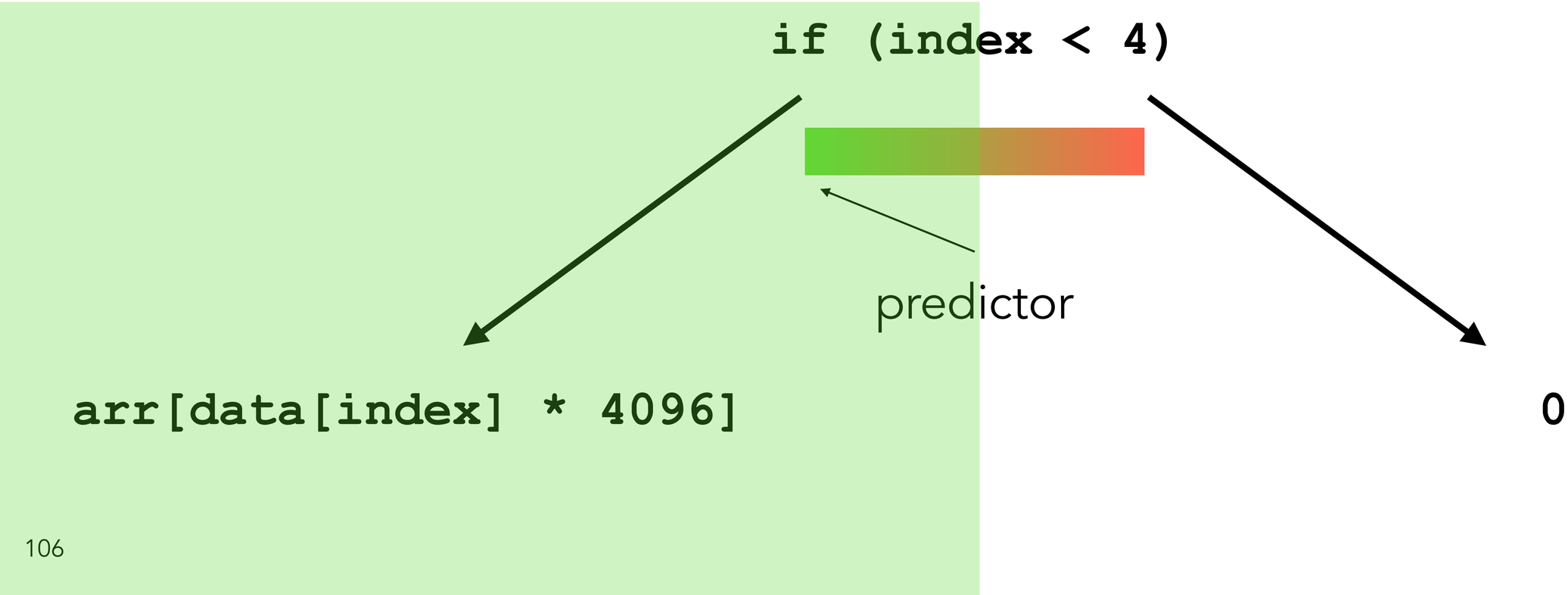
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 3;

char* data = "textKEY";

if (index < 4)
```



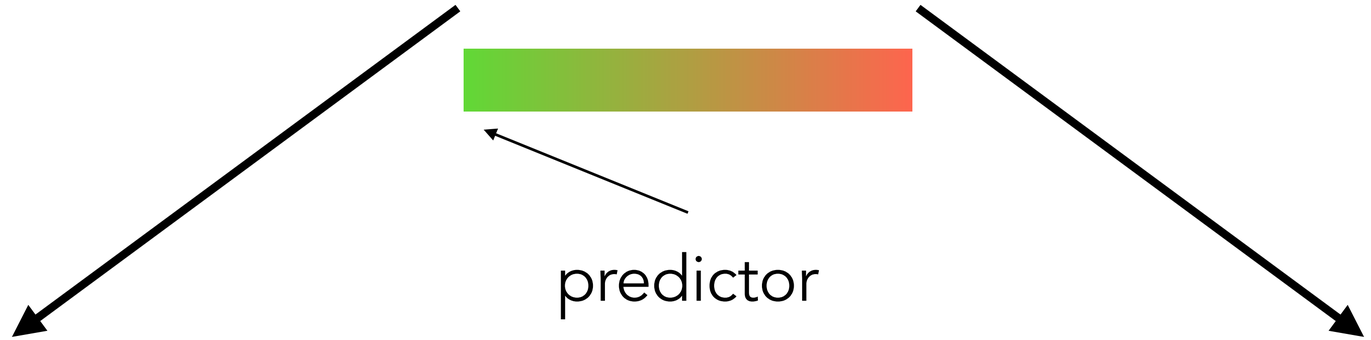
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 4;

char* data = "textKEY";

if (index < 4)
```




```
arr[data[index] * 4096]                    0
```

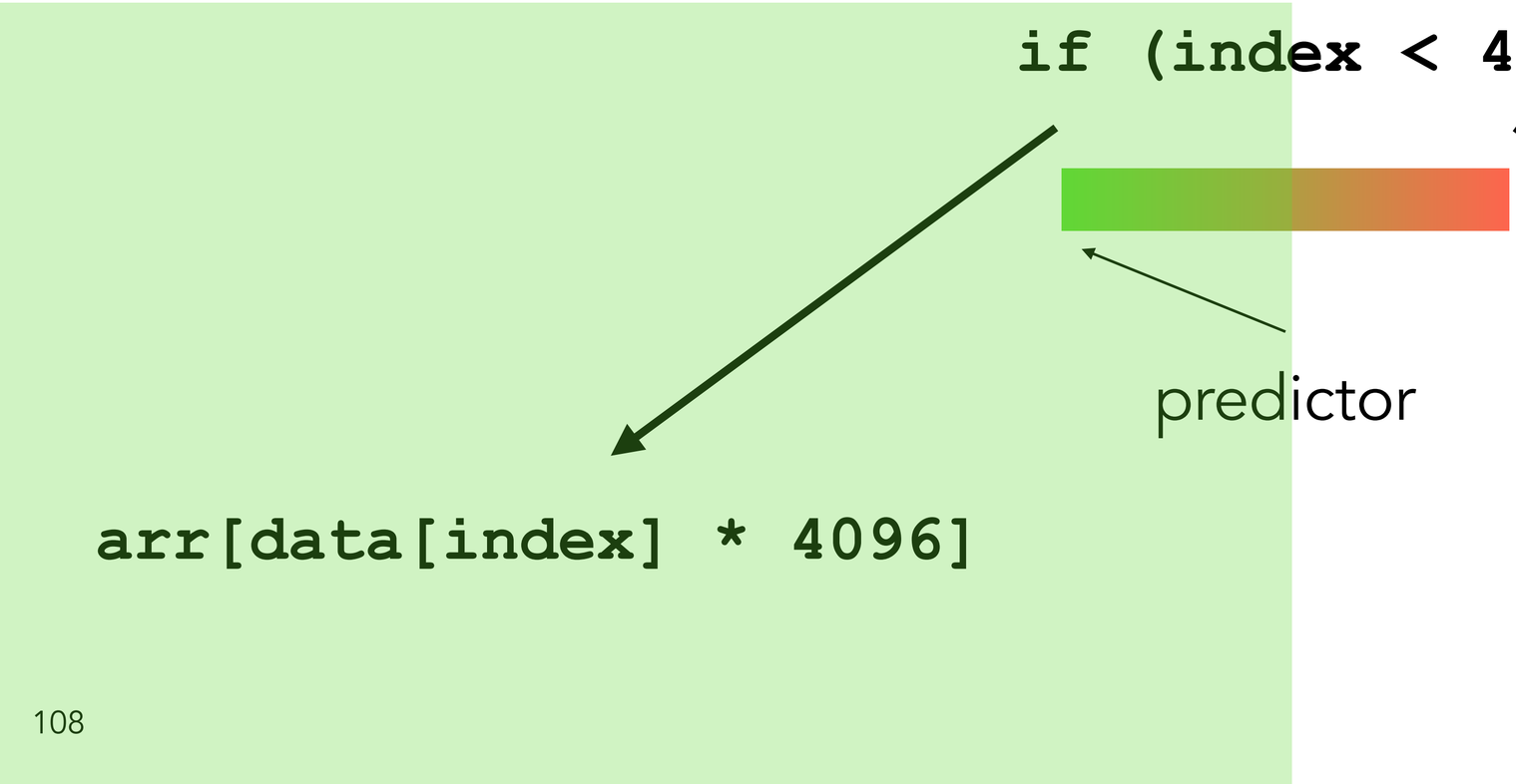
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 4;

char* data = "textKEY";

if (index < 4)
```



Branch prediction will erroneously place "K" in the cache…

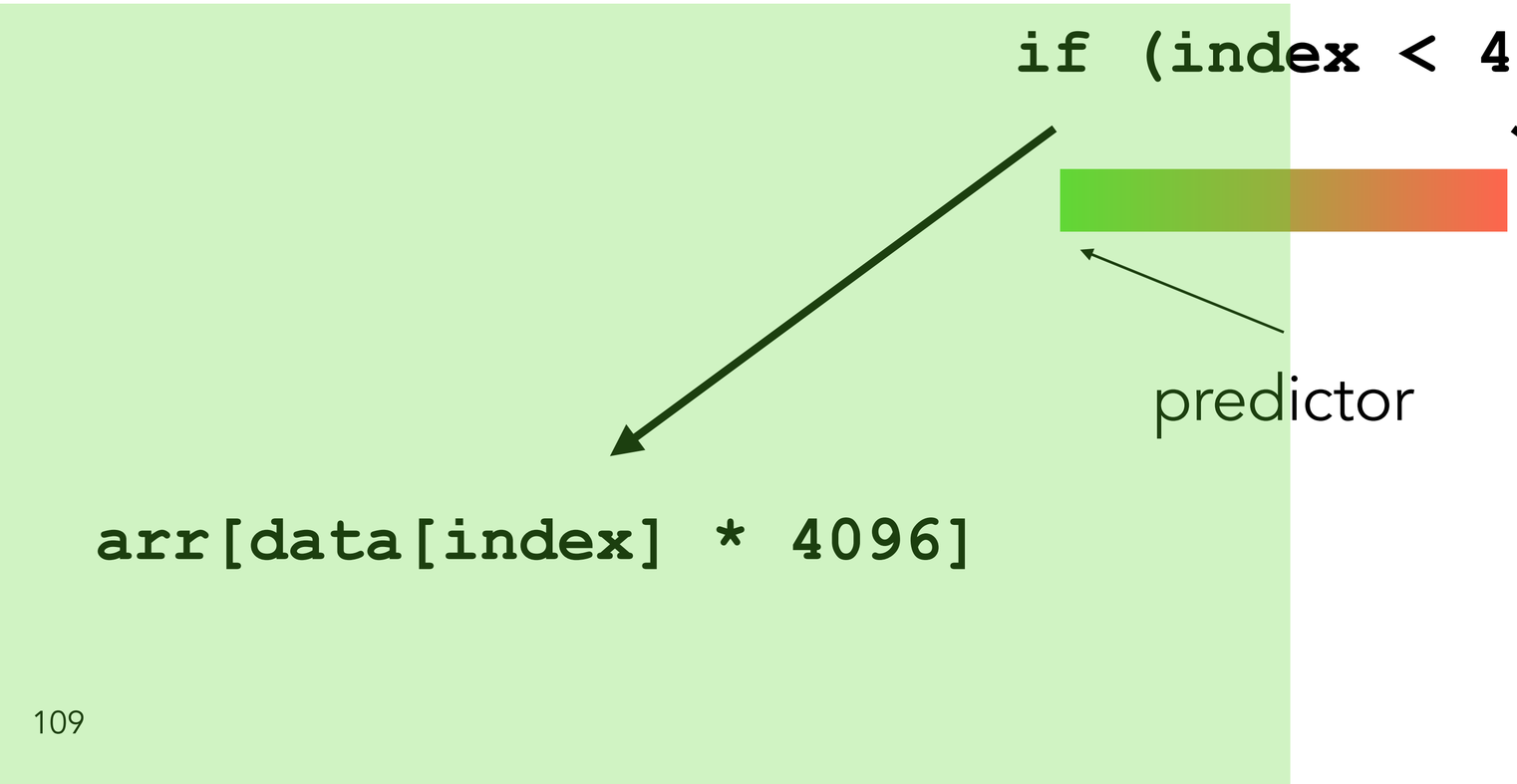
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 4;

char* data = "textKEY";

if (index < 4)
```



Branch prediction will erroneously place "K" in the cache…

Extract just like you would in Meltdown (flush + reload)

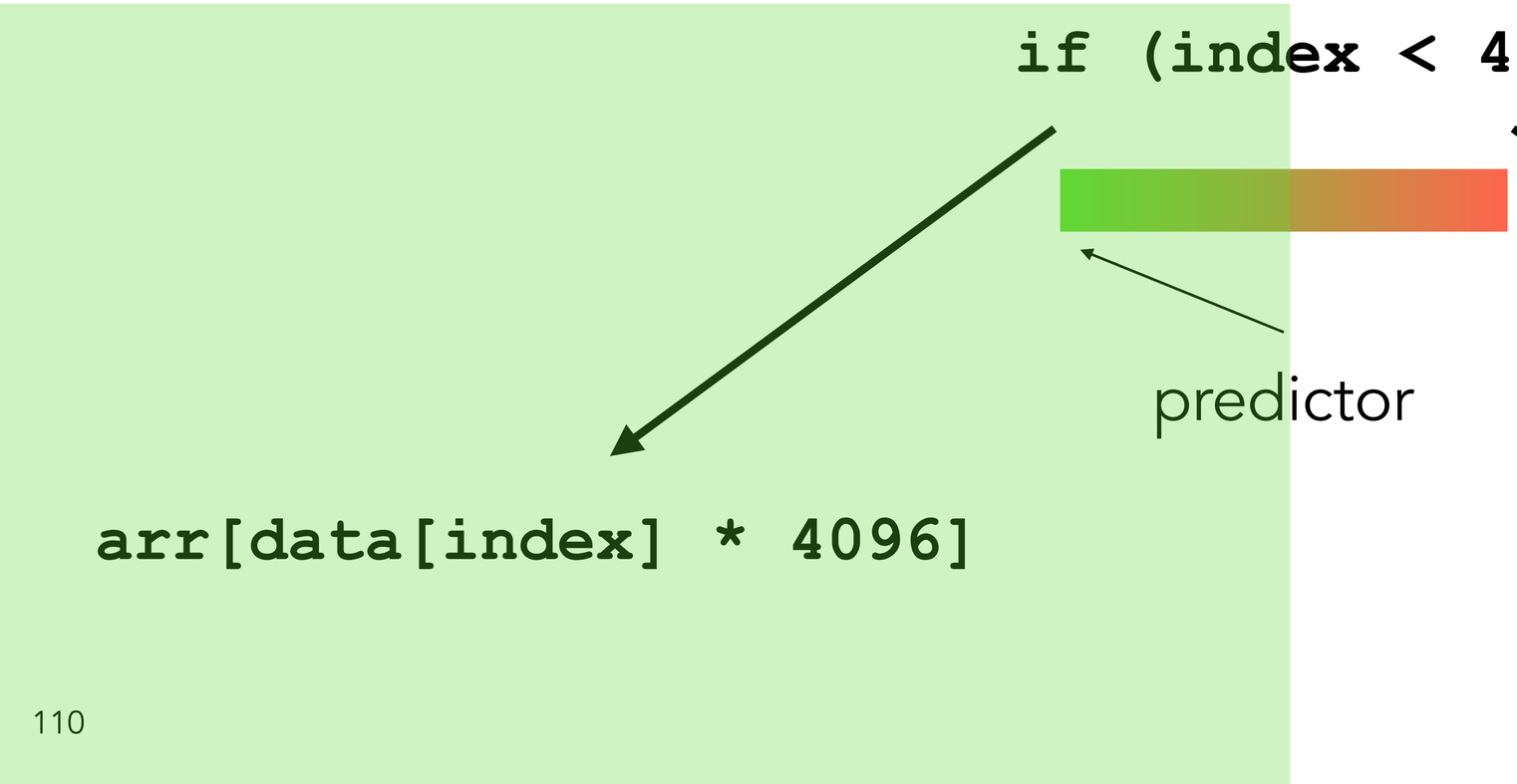
Spectre attacks (simplified)

- Goal: Train branch predictor to leak memory values

```
index = 4;

char* data = "textKEY";

if (index < 4)
```



Branch prediction will erroneously place "K" in the cache…

Extract just like you would in Meltdown (flush + reload)

game over gg

What do we do about Spectre?

- Ideas?

What do we do about Spectre?

- Ideas?
- Admittedly very rough; but good news is it's hard to exploit
  - Broad issue about speculation and branch prediction
- Could disable speculation on branches… but htere's a huge performance impact as a result
  - Also can only be done by chip manufacturer
- Selectively insert instructions to stop speculation at sensitive branches
  - LFENCE
- All of these are bandaids, not solutions

Sea-change shift in last 8 years

- Computer architects spent the last 20 years optimizing for common use case
  - Assumption was that if optimization doesn't change output then we're all good
- Sadly, every optimization is now under intense, immense scrutiny
  - New microarchitectual side-channel papers are published almost every conference…
- Hardware vendors and computer architects are retooling to figure out how to still offer optimizations without huge security hit
  - ¯_(ツ)_/¯ it's an ongoing field

My unsolicited 2c on side channel attacks

- Side channels are epicly cool and fun
  - They entertain the part of your brain that likes puzzles
- Are they the most important harm / security threat facing people today?
  - No. They’re niche, hard to execute, and often probabilistic in practice
- But they’re worth study… even if to teach us all the ways in which our assumptions might be flawed :)

Next time…

- We change gears!
  - Done with the “low level” parts of the course material
  - Moving up the OSI stack into the Web; web attacks, web defenses, etc.
  - Web will be the focus of PA3
- Thinking ahead to the midterm…
  - Application security, systems security, and web security are the main units we’ll cover