Hardware issues (and why I disable js by default...)

Rowhammer

- applies to many DRAM chips.

⊗ - by repeatedly writing a row of DRAM, you can cause random changes in <u>nearby</u> memory cells...

DRAM: array of switched capacitors...

- memory requires frequent refreshing!

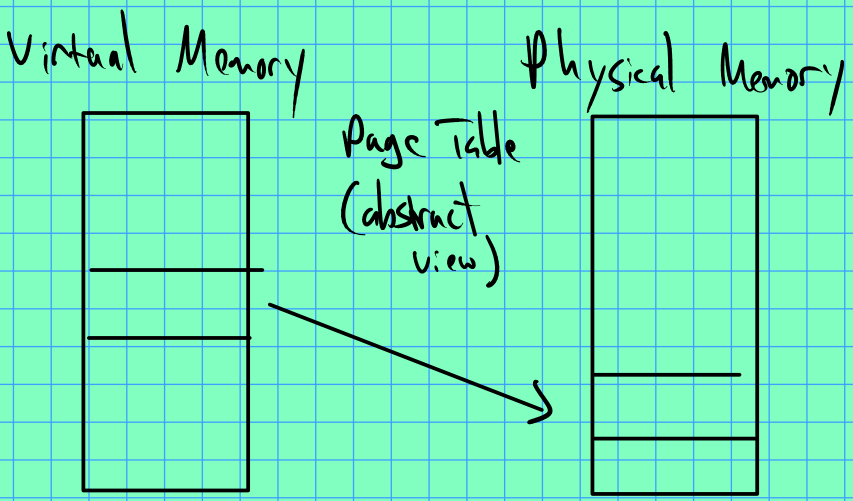
Note: forcing rewrites will require cache eviction...

Why does this matter?

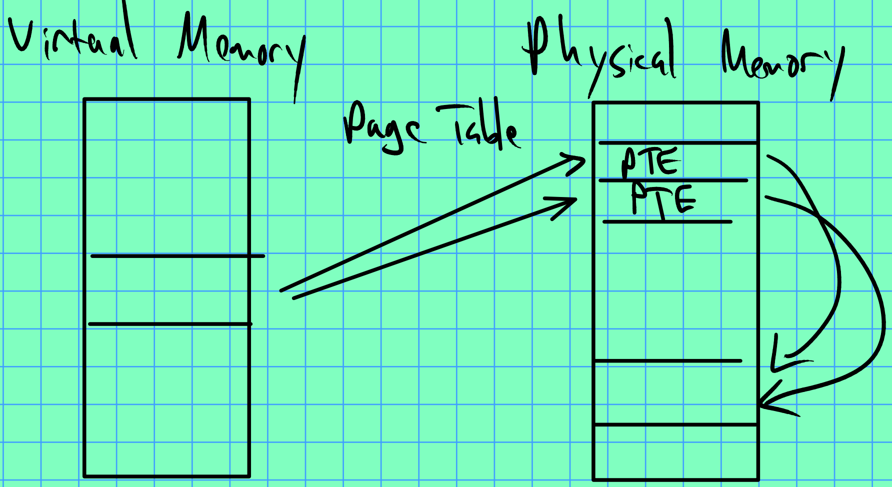
Already unsettling - random js in browser could corrupt my memory...

<u>Worse</u>: you can actually use an unprivileged process to read/write all kernel memory!

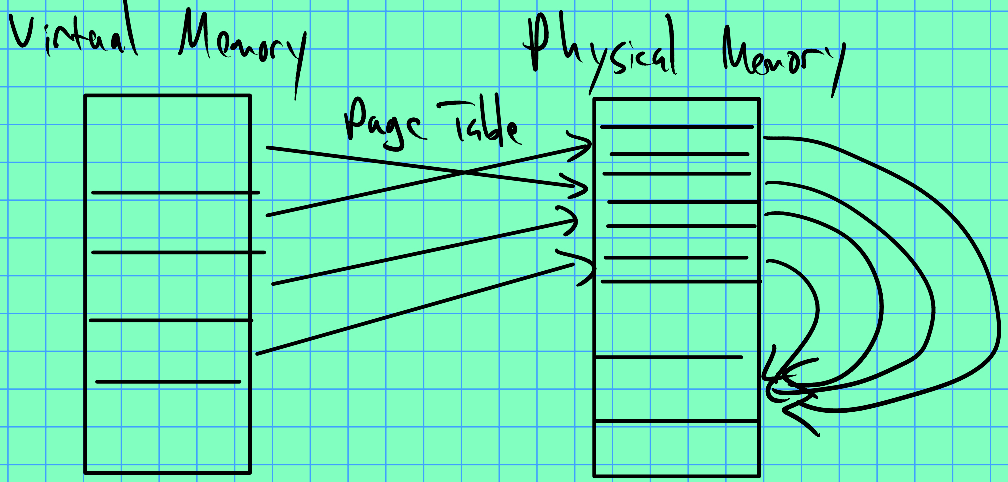
Background: Virtual Memory + Page Tables



(PTE ≡ page table entry)



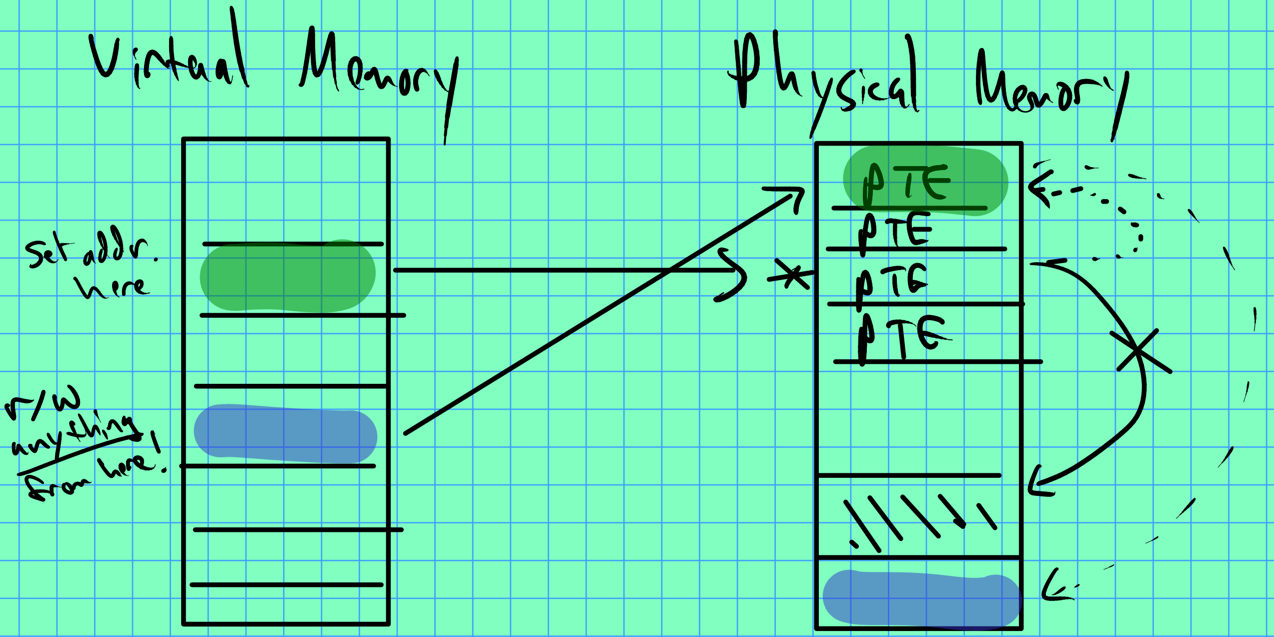
What happens if a file is repeatedly mapped by the same process?



The point: we can fill physical memory with page table entries... ⊗

So we can construct conditions by which a random bit changing is likely to be in one of our page table entries.

If the embedded pointer in the PTE changes, then that mapping points elsewhere... probably to one of our PTEs! ⊗



$*$ = location of bit flip(s)

🟩 = block of virtual memory now mapped to PTE used for 🟦

Hence, one could set desired addr. in 🟩 and write to it via 🟦

Technical obstacles : different regions of a DRAM chip may be more/less prone to the attack.
Helps to fragment data in physical memory

What can be done?

- ECC memory? Not helpful when ≥ 3 bits modified
- Increase refresh rate for DRAM ?
  also increases power consumption...
- Include (in the DRAM module) a counter for updates to a row. When this

exceeds some threshold, refresh any nearby rows that might be affected.
(search "TRR" = target row refresh)

Spectre/Meltdown

Outline/prerequisites:

- cpu cache
- speculative execution
- Out of order execution.

CPU cache Memory storage on the cpu itself is used to speed up memory access for recently used items.
Why? Waiting for memory bus could be pretty slow (for my computer, $\approx$ 8-10 clock cycles)

Possible downside: exposes a side channel via timing. By measuring how long it takes to read from some address a, you can know if that has been recently used.

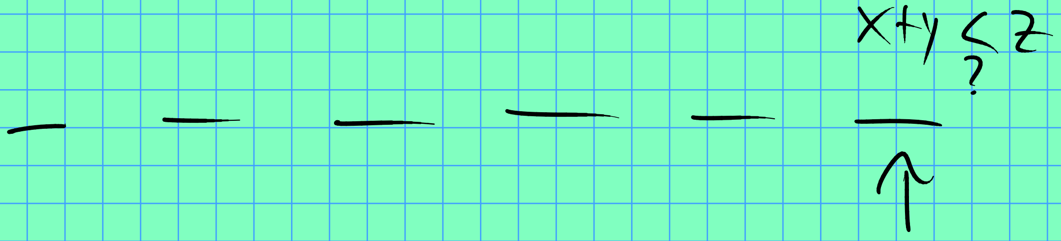
⊗ **Trick**: make address a depend on data you should not have acccss to.

Speculative execution:

Each machine code instruction is broken into μOps. Executed in a pipeline, perhaps by different execution ports which can operate in parallel. (E.g. ALU, vector, load/memory, AGU, AES)

```
while (i < 1000000) {
    if (x+y < z)
        a = b;
    else
        a = c;
}
```

branches can sabotage performance of pipelined architecture.



Basics of Spectre

Setup / Goal: want to read memory at location s

Say s is aliased by $A[x]$ ($\equiv *s$)

where A an array of size n_A.

(So $x \geq n_A$ (out of bounds))

Suppose attacker can control x for code like this:

```
if (x < n_A)            // bounds check
    t = B[A[x] * 4096];
```

Note: address in buffer B to be accessed could depend on secret data.

Suppose we have

Not cached	Cached
n_A	A, B (the pointers)
$B[0], B[1], \ldots B[n_B - 1]$	$*s \equiv A[x]$

Suppose further that branch prediction has been trained that usually $x < n_A$.

Then the fetch of $B[A[x] * 4096]$ may be issued while CPU is still evaluating $x < n_A$. Eventually CPU sees $x \nless n_A$ and rewinds.

So what? The memory fetch $B[A[x] * 4096]$ has a measurable side-effect: $B[-]$ is now cached. Those cache side effects are not rewound.

Now we can learn secret $*s$ as follows:

Time access to $B[0]$, $B[4096]$, $B[2*4096]$, ...
and see if one access was faster than the rest. This tells you $*s$!

Note: we multiply by 4096 assuming this is the size of a cache line (reading address a will also pull 4096 neighbors into the cache)

Spectre V2 — manipulate registers used in an indirect jump. Use this to execute a "spectre gadget".

Meltdown Basic attack code:

Setup: Say %rcx is addr. of secret byte you want (maybe in kernel memory!)
Say %rbx points to a buffer we can read.

In assembly:

```
① clflush                          // flush cache
② xor %rax, %rax
③ movb (%rcx), %rax                // (%rcx) ≡ *rcx
④ shl $12, %rax                    // multiply rax by 2^12 = 4096
⑤ mov (%rbx + %rax), %rdx          // read from address that
                                      depends on secret value!
```

Believe it or not, ⑤ might happen out of order before the exception raised by ③ is processed!

With spectre, it was just bad branch prediction.
Here, the code will raise an exception! (SegV)

Want to scan (%rbx), (%rbx + 4096), ...
(%rbx + 255 * 4096)
to see which entry has a faster access time.
But our program hits a segmentation violation.
What to do?

Clunky solution: fork() first.
Better solution: write a signal handler that finishes the attack.
Other solutions: make use of transactional memory (RTX)

<u>Mitigations?</u>

It was essential that kernel memory was mapped into our process!
Solution: stop making kernel page tables available at all via user space virtual addresses.

See "Kernel Page Table Isolation".
Works, but hurts performance.

What about for Spectre?

- Compiler options to produce harder to exploit binaries. (see -mindirect-branch in gcc docs)
- prevent register manipulation