Why software engineers don't get bonuses (or Rowhammer, Meltdown and Spectre)

Joseph Hallett

November 9, 2022

Abstraction, abstraction...

In computer science we like to pretend that it's all digital...

- Perfect 1s and 0s.
- Computers that work exactly how the specifications say.
- Hardware can be (largely) ignored.
- Lower level details... *that's for electrical engineers not us!*

This doesn't always work out.

Money, money, money...

It mostly works out though...

- But this whole unit is about what happens when computers start doing weird things.

Electrical engineers, and computer architects make mistakes

Cost of fixing hardware is big

- You cannot trivially fix a silicon wafer
- You cannot recall old hardware and change the circuits

Cost of fixing software is cheap

- It's just code!

When there is a bug... its the software engineers who fix it

Consequently we have to clean up after their messes

- So software is always running late
- So we don't get a bonus `:-(`

(or so said my first boss)

Plan

In this lecture we'll cover two ways hardware is broken.

- Rowhammer and DRAM
- Meltdown/Spectre and CPUs

We'll also cover how software works around it.

DRAM glorious, DRAM!

Memory! Used to store all the things the computer is thinking about that we can't fit in a register!

- Implemented using a *capacitor* and a *transistor* per bit
- *Ganged* (arranged) into long rows (~8k bits per row)
- Placed into *banks* of ganged rows

When we want to read a bit of memory:

- We find the row it is in.
- Activate the row by letting the capacitors discharge
- Which copies the row into an active memory buffer

DRAM needs to be *refreshed* so the capactiors don't lose their charge over time

- Roughly every 64ms for modern hardware

Electronic Engineering is messy

Capacitors leak charge

Current in wires induces current in other nearby wires

The 1s and 0s aren't charged or uncharged capacitors

- Its whether a capacitor is currently discharging more or less than a threshhold voltage

But this is all fine because electronic components are large!

Or they were...

- As memory capacity has *increased*...
- The physical dimensions of memory has got *smaller*.

> *"The ~~Dwarves~~ Electrical engineers tell no tale; but even as ~~mithril~~ memory density was the foundation of their wealth, so also it was their destruction: they delved too greedily and too deep, and disturbed that from which they fled, ~~Durin's Bane~~ Rowhammer."*
> — *Gandalf the Greyhat*

Flipping bits

Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors

Yoongu Kim[1] Ross Daly* Jeremie Kim[1] Chris Fallin* Ji Hye Lee[1]
Donghyuk Lee[1] Chris Wilkerson[2] Konrad Lai Onur Mutlu[1]
[1]Carnegie Mellon University [2]Intel Labs

Abstract. *Memory isolation is a key property of a reliable and secure computing system — an access to one memory address should not have unintended side effects on data stored in other addresses. However, as DRAM process technology scales down to smaller dimensions, it becomes more difficult to prevent DRAM cells from electrically interacting with each other. In this paper, we expose the vulnerability of commodity DRAM chips to disturbance errors. By reading from the same address in DRAM, we show that it is possible to corrupt data in nearby addresses. More specifically, activating the same row in DRAM corrupts data in nearby rows. We demonstrate this phenomenon on Intel and AMD systems using a malicious program that generates many DRAM accesses. We induce errors in most DRAM modules (110 out of 129) from three major DRAM manufacturers. From this we conclude that many deployed systems are likely to be at risk. We identify the root cause of disturbance errors as the repeated toggling of a DRAM row's wordline, which stresses inter-cell coupling effects that accelerate charge leakage from nearby rows. We provide an extensive characterization study of disturbance errors and their behavior using an FPGA-based testing platform. Among our key findings, we show that (i) it takes as few as 139K accesses to induce an error and (ii) up to one in every 1.7K cells is susceptible to errors. After examining various potential ways of addressing the problem, we propose a low-overhead solution to prevent the errors.*

1. Introduction

The continued scaling of DRAM process technology has enabled smaller cells to be placed closer to each other. Cramming more DRAM cells into the same area has the well-known advantage of reducing the cost-per-bit of memory. Increasing the cell density, however, also has a negative impact on memory reliability due to three reasons. First, a small cell can hold only a limited amount of charge, which reduces its noise margin and renders it more vulnerable to data loss [14, 47, 72]. Second, the close proximity of cells introduces electromagnetic coupling effects between them, causing them to interact with each other in undesirable ways [14, 42, 47, 55]. Third, higher variation in process technology increases the number of outlier cells that are exceptionally susceptible to inter-cell crosstalk, exacerbating the two effects described above.

As a result, high-density DRAM is more likely to suffer from *disturbance*, a phenomenon in which different cells interfere with each other's operation. If a cell is disturbed beyond its noise margin, it malfunctions and experiences a disturbance errors, DRAM manufacturers have been employing a two-pronged approach: *(i)* improving inter-cell isolation through circuit-level techniques [22, 32, 49, 61, 73] and *(ii)* screening for disturbance errors during post-production testing [3, 4, 64]. We demonstrate that their efforts to contain disturbance errors have not always been successful, and that erroneous DRAM chips have been slipping into the field.[1]

In this paper, we expose the existence and the widespread nature of disturbance errors in *commodity* DRAM chips sold and used today. Among 129 DRAM modules we analyzed (comprising 972 DRAM chips), we discovered disturbance errors in 110 modules (836 chips). In particular, *all* modules manufactured in the past two years (2012 and 2013) were vulnerable, which implies that the appearance of disturbance errors in the field is a relatively recent phenomenon affecting more advanced generations of process technology. We show that it takes as few as 139K reads to a DRAM address (more generally, to a DRAM row) to induce a disturbance error. As a proof of concept, we construct a user-level program that continuously accesses DRAM by issuing many loads to the same address while flushing the cache-line in between. We demonstrate that such a program induces many disturbance errors when executed on Intel or AMD machines.

We identify the root cause of DRAM disturbance errors as voltage fluctuations on an internal wire called the *wordline*. DRAM comprises a two-dimensional array of cells, where each *row* of cells has its own wordline. To access a cell within a particular row, the row's wordline must be enabled by raising its voltage — i.e., the row must be *activated*. When there are many activations to the same row, they force the wordline to toggle on and off repeatedly. According to our observations, such voltage fluctuations on a row's wordline have a disturbance effect on nearby rows, inducing some of their cells to leak charge at an accelerated rate. If such a cell loses too much charge before it is restored to its original value (i.e., *refreshed*), it experiences a disturbance error.

We comprehensively characterize DRAM disturbance errors on an FPGA-based testing platform to understand their behavior and symptoms. Based on our findings, we examine a number of potential solutions (e.g., error-correction and frequent refreshes), which all have some limitations. We propose an effective and low-overhead solution, called *PARA*, that prevents disturbance errors by probabilistically refreshing only those rows that are likely to be at risk. In contrast to other solutions, PARA does not require expensive hardware structures or incur large performance penalties. This paper makes the following contributions.

Rowhammering is a well known bug in DRAM chips since ~2010

If you repeatedly charge and discharge a row in DRAM really quickly it can cause errors in nearby rows

Manufacturers all knew about it, but didn't really bother to document it.

- Seen as a *reliability* issue, not a *security* issue
- Cached memory largely fixes it.

Several papers discuss it and explore it

- Almost all RAM is vulnerable to it (to some extent)
- *Maybe* you could do something malicious theoretically?
- Still treated as a *reliability* issue

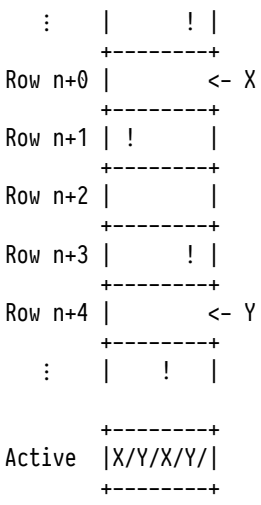
Flipping bits, in practice

```
code1a:
        mov eax, [X]
        mov ebx, [Y]
        clrflush [X]
        clrflush [Y]
        mfence
        jmp code1a
```

Find two memory addresses X and Y that are in separate rows of RAM and:

1. Load `*X` into the active buffer
2. Load `*Y` into the active buffer
3. Kick `*X` out of the cache (so next read goes directly to RAM)
4. Kick `*Y` out of the cache (so next read goes directly to RAM)
5. Ensure that the cache is really gone
6. Repeat (as fast as you can)

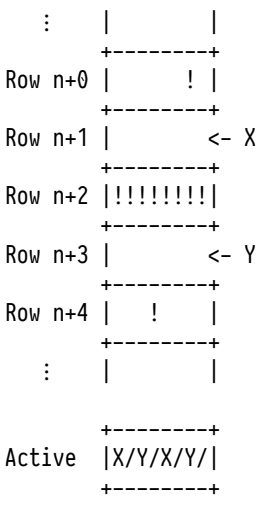
Token ASCII Art Diagram



If you perform the rowhammer with the above RAM layout

- *Eventually* you'll get errors in the adjacent rows (the !'s)
- This is called *single-sided* Row Hammering

Double Sided Rowhammering



If you select X and Y so there is exactly 1 row between them

- *Eventually* you'll get errors in the adjacent rows (the !'s)
- *Quickly* you'll get errors in the in-between row
- This is called *double-sided* Row Hammering

So what?

So we can introduce (typically) single bit errors in RAM... so what?

Mark Seaborne and Halvar Flake (and others) continue exploring

- Discover double-sided variant of Rowhammering
- Find that its not just all RAM which is susceptible to this, but that its *all rows* in *all ram* (between 30–100%... but improvements later make it 100%).

They discover the bit flips are consistent

- Same bits flip every time when you Rowhammer the same rows

And even consistent between the same RAM products

- If Alice and Bob have the same make RAM from the same manufacturer
- Then if they Rowhammer the same rows the same bits will always flip

This seems bad, but so what?

- You can violate the integrity of RAM, but is that all?
- How could you possibly use this as part of an attack to get arbitrary code execution?

NaCl Sandbox

Privileged sandbox for running *native code* from a web browser safely.

- Checks if the code is *safe* (i.e. doesn't contain any weird syscalls or violate safety properties)
- If so, it loads the chunks of instructions aligned on 32B boundaries

```
and eax, 0x000F                 ; Truncate address to 32 bits and mask to be 32-byte aligned
add rax, r15                    ; Add r15, the sandbox base address
jmp [rax]                       ; Jump to the loaded code snippet
```

Can we use Rowhammer to escape the sandbox?

(I mean obviously we can, but its more fun if you work out how to do it rather than me telling you...)

Variadic Instruction Sets

X86 is a dense instruction set

- Different instructions have different lengths
- Some have multiple length

```
20ea0: 48 b8 0f 05 eb 0c f4 f4 f4 f4    movabs rax, 0xf4f4f4ff40ceb050f
20ea2:       0f 05                      syscall
20ea4:             eb 0c                jmp 0xe
```

Last chance to guess the exploit?

Escaping NaCL

Code section is readable, so lets try and Rowhammer that `and eax, 0x000F`!

- Conveniently the code section is also readable (but not writable) by the loaded process so we can tell if it has worked

So the attack:

1. Load a sequence of safe code that happens to be *unsafe* if you were to run it with a 1-bit offset
2. Rowhammer the loading code so that NaCl checks the code with no-offset, but runs it with an offset
3. Probably the program is gonna crash 'cos the loading code isn't valid
4. Or we Rowhammer the Kernel's memory and crash the entire computer
5. ...or it works?

Luckily most unprivileged users are allowed to run crashy programs millions of times without batting an eyelid

See this course.

Whoops!

Mark Seaborn and Halvar Flake have managed to Rowhammer their way to aribtrary code execution.

- Guess it was security bug after all... `B-)`
- Also publish a similar but fiddlier Linux root privilege escallation attack using Rowhammer

Short term:

- `clflush` is banned in NaCl loaded code
- `clflush` is banned from non-root code (sometimes)

Those aren't sustainable solutions...

Buy better RAM?

- But how do you tell?

...with error correction codes (ECC)?

- Expensive though, and slower *(worth it for a server, not for a laptop...)*
- Still a potential denial of service/vulnerability if you can corrupt multiple bits at once with Rowhammer

...which refreshes faster?

- If you can't Rowhammer faster than the refresh speed the attack doesn't work
- But this slows down the *whole* computer.

...and which refreshes neigbouring rows more often?

- More recent DRAM standards do this...
- Again, slows things down.

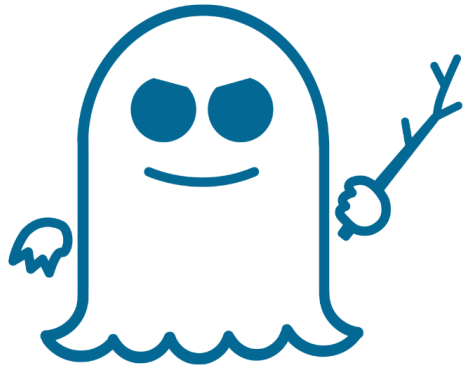
Are we depressed yet?

Have you considered taking up pottery?

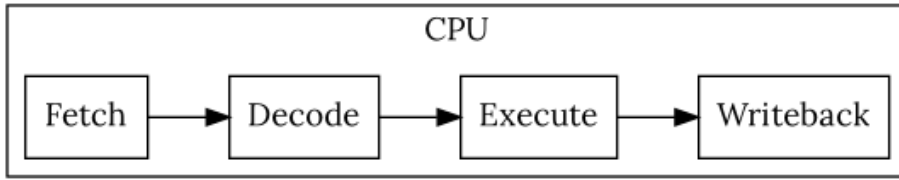
- Mud is not susceptible to Rowhammer or any of the techniques covered in this course
- Mud will not make you sad (except when your bowls collapse)
- You can make bowls and mugs and *super cute* pots!

Honestly, I cannot recommend it highly enough.

Buckle up...



CPU Pipelines



In modern CPUs instructions take different times to complete...

So we *pipeline* them

- As one instruction is *executing*...
- The next can be being *decoded*...
- And the next can be being *fetched*.

Significant performance gains!

Branch Prediction

```
unsigned long factorial(unsigned long n) {
  unsigned long result = 1;

  while (n) [[likely]]
    result *= n--;

  return result
}
```

Conditionals can cause a problem however...

- Can't load fetch the next multiply until we know if n > 0
- So pipeline stalls

Solution

Speculate that the loop is *likely* to be taken...

- CPU assumes it will be and fetches anyway
- If the assumption is wrong the CPU pipeline will have to be flushed before writeback...
- ...but that should only happen once per call
- Speedup from removing the pipeline stall is bigger than the single pipeline flush

More performance gains!

- Especially with *Symmetric Multi-Threading*.

Watch the pointer closely...

Suppose we have two arrays: `array1` and `array2`:

- What happens if we run this code?

```
if (x < array1_size) [[likely]]
  y = array2[array1[x]];
```

Which array is the pointer under?

`y` gets indexed by whatever is in `array[x]`
What about if `x > array1_size`?

```
if (x < array1_size) [[likely]]
  y = array2[array1[x]];
```

No, unfortunately that's a lemon...

The `if` statement won't succeed...
...but we *said* it was likely to succeed so the next line will be speculatively executed anyway

```
if (x < array1_size) [[likely]]
  y = array2[array1[x]];
```

And that would segfault anyway...

- And it would be mean to segfault on an instruction you never were going to execute.
- So we don't... *even if* we've speculatively executed it.

As soon as the branch misprediction is detected start the *rollback* process

- Undo changes to registers
- Reset exception flags
- Cancel any memory writes

Jobs a good 'un, am I *write* ?

Just the writes?

```
if (x < array1_size) [[likely]]
  y = array2[array1[x]];
```

See the caches are a separate subsystem and managed by the MMU.

- When the second line executes the page of memory containing `array2[array1[x]]` will be cached in preparation for the load into `y`
- And an exception signalled...
- That the CPU will tell the OS about when it hits writeback...
- ...which will never actually happen because the `if` will turn out to be a branch misprediction

Everything is still good right?

Oh dear...

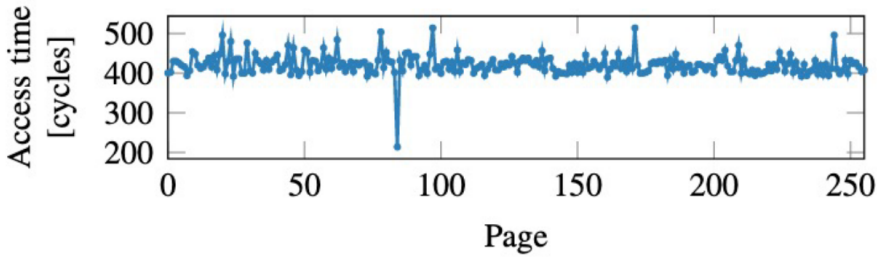
Suppose we guarantee that for every different value of `x` a different page of memory will be cached?

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

(and that the branch will ALWAYS be mispredicted by the CPU).

Oh dear, Oh dear...

And then we were to time how long it took to access every page of memory...



Anyone want to guess what the value at `array1[x]` was?

- Which reading should have caused a segfault of...

Oh dear, Oh dear, Oh dear

Suppose this attack also worked not just with C but via Javascript...

```
if (index < simpleByteArray.length) {
    index = simpleByteArray[index | 0];
    index = (((index * 4096)|0) & (32*1024*1024-1))|0;
    localJunk ^= probeTable[index|0]|0;
}
```

So you can leak a byte of memory... big deal?

- But given a few hours you could leak *all* of memory
- On any system where you can host a webpage

Good job nothing useful is ever in memory, eh?

- Keys, personal data, certificates, passwords...

The Cloud

So how are we going to fix this?

This is the *Spectre* vulnerability, and is part of the *Meltdown* family of attacks:

Meltdown (`CVE-2017-5754`) melts down security barriers

Spectre (`CVE-2017-5753` `CVE-2017-5715`) make speculative execution scary

Affects:

- All operating systems
- All CPUs with branch prediction

No, seriously please, how do we fix this?

We have a couple of ideas:

Disable branch prediction would require all new hardware, and have an enormous performance impact

Disable caches would require all new hardware and have an enormous performance impact

Disable multithreading doable is software for *most* architectures, but would halve the number of available cores. Also doesn't actually fix the issue but makes everything much harder to exploit

Which one do you think we've gone with?

Anyone's computers feeling a bit slow?

When I was growing up everytime they made new computers they always felt *lots* faster...

- Anyone not really noticed this recently?

We do have other mitigations other than turning SMT off...

- but none of them are perfect, and all have an impact
- and turning multithreading off really does make this *much*, *much* harder to exploit

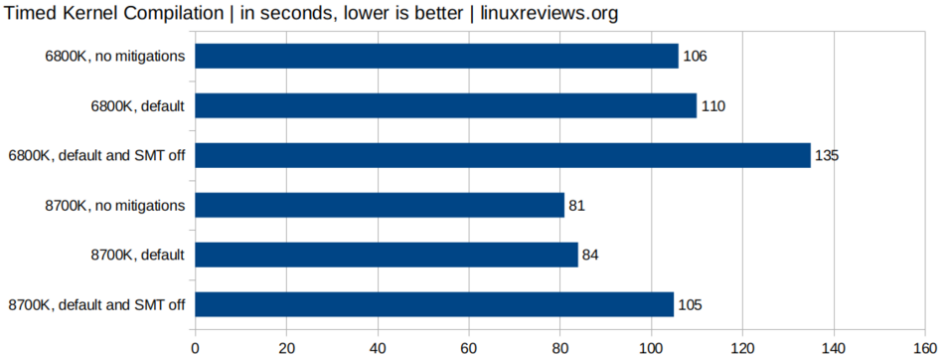
```
cat /sys/devices/system/cpu/vulnerabilities/{meltdown,spectre*}
```

```
Mitigation: usercopy/swapgs barriers and __user pointer sanitization
Mitigation: Enhanced IBRS, IBPB: conditional, RSB filling, PBRSB-eIBRS: SW sequence
```

```
curl https://make-linux-fast-again.com/
```

noibrs noibpb nopti nospectre$_{v2}$ nospectre$_{v1}$ l1tf=off nospec$_{storebypassdisable}$ no$_{stfbarrier}$ mds=off tsx=on tsx$_{asyncabort}$=off mitigations=off noibrs noibpb nopti nospectre$_{v2}$ nospectre$_{v1}$ l1tf=off nospec$_{storebypassdisable}$ no$_{stfbarrier}$ mds=off tsx=on tsx$_{asyncabort}$=off mitigations=off

It isn't just you...



About a 25–30% performance penalty in the *worst* case
About a 10% in general usage

So in conclusion...

Computer hardware fundamentally broken

- RAM doesn't work
- CPUs fundamentally broken

Software can give us a solution!

- But no one is happy about it
- More cost, slower performance
- And so no bonuses for you

My suggestion to all of you

People will always need clothes!

- Sewing is fun!
- It's about an evenings work to make a hawaiian shirt!
- Sewing machines *not* vulnerable to any attacks in this course
  - (unless they're really fancy...)