

Exploiting the DRAM rowhammer bug to gain kernel privileges

How to cause and exploit
single bit errors

Mark Seaborn and Thomas Dullien

Bit flips!

This talk is about single bit errors -- i.e. bit flips:

- How to cause them
- How to exploit them

Specifically: bit flips caused by the “rowhammer” bug

The rowhammer DRAM bug

Repeated row activations can cause bit flips in adjacent rows

- A fault in many DRAM modules, from 2010 onwards
- Bypasses memory protection: One process can affect others
- The three big DRAM manufacturers all shipped memory with this problem
 - A whole generation of machines

Overview of talk

- How to cause bit flips by row hammering
- Proof-of-concept exploits
- Mitigations and the industry's response

- Topics covered in our Project Zero blog post
- Plus things we've learned since the blog post
 - Rowhammer from Javascript?

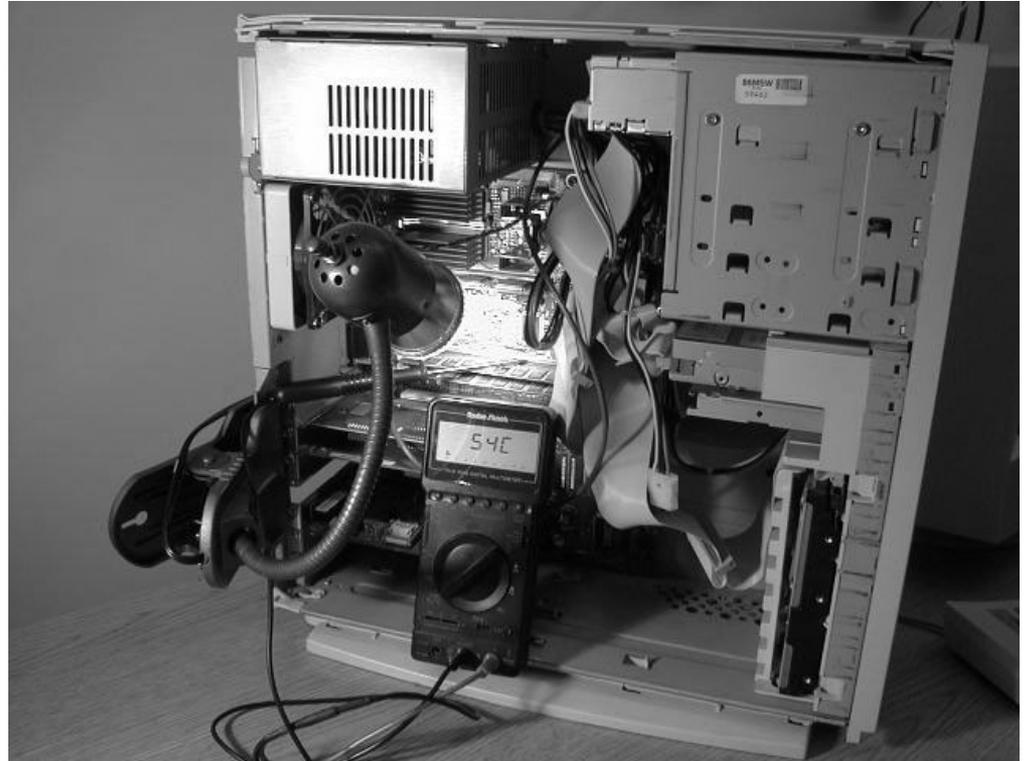
Exploiting random bit flips

How would one exploit a truly random bit flip in physical memory?

2003 paper:

“Using Memory Errors to Attack a Virtual Machine”

- by Sudhakar Govindavajhala, Andrew Appel
- Escape from Java VM



Exploiting random bit flips

How would one exploit a truly random bit flip in physical memory?

- Generic strategy:
 - Identify data structure that, if randomly bit-flipped, yields improved privileges
 - Fill as much memory as possible with this data structure
 - Wait for the bit flip to occur
- Apply this to JVM:
 - Spray memory with references
 - Bit flip causes reference to point to object of wrong type

Types of memory error

Totally random (e.g. cosmic ray) vs. repeatable

Rowhammer is inducible by software, and *often repeatable*

- Similar exploit techniques can be used in both cases
- But repeatable bit flips offer more control

Intro to DRAM

- Cells are capacitors → refresh contents every 64ms
- “Analogue” device → sense amplifiers
- Accessed by row → “currently activated row”, row buffer

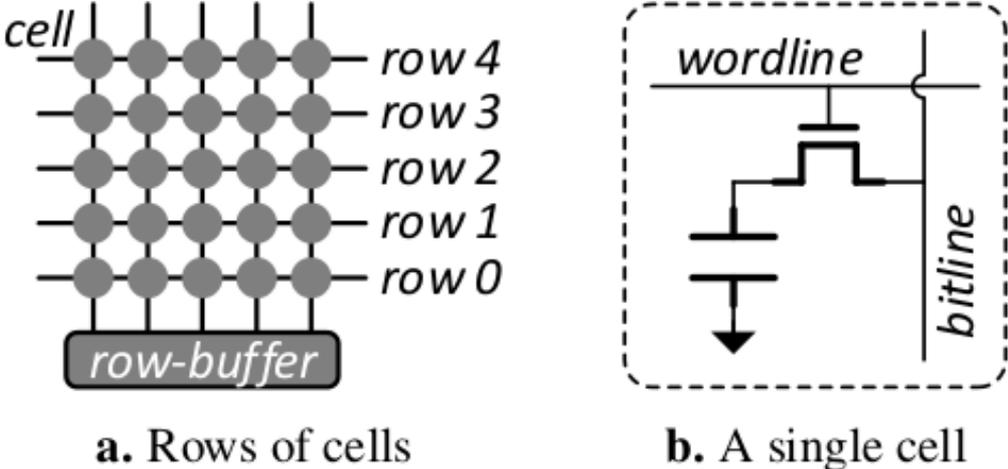
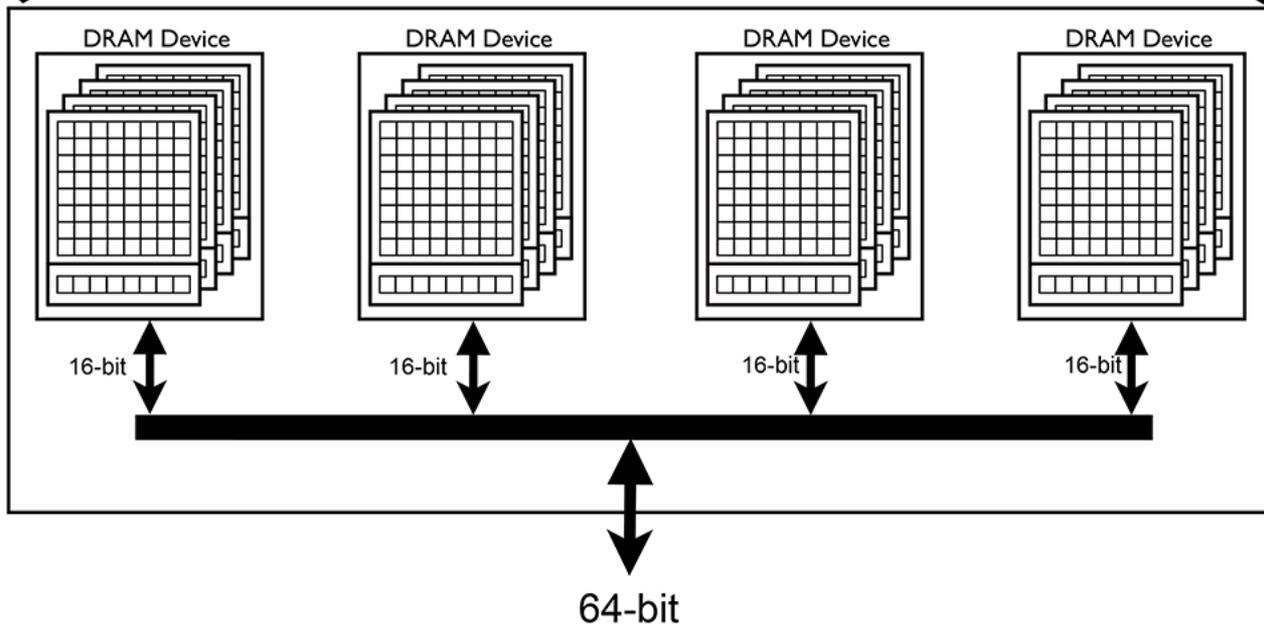


Figure 1. DRAM consists of cells

DRAM disturbance errors

- Cells smaller and closer together
 - <40nm process
- Electrical coupling between rows
 - *“Word line to word line coupling”*
 - *“Passing gate effect”*
- Activating a row too often causes “disturbance errors”
 - Can be as low as 98,000 activations (8% of spec)
 - DDR3 spec allows upto 1,300,000 activations
 - Insufficient testing by manufacturers?



(Diagram from
ARMOR
project,
University of
Manchester)

Timeline: 2014

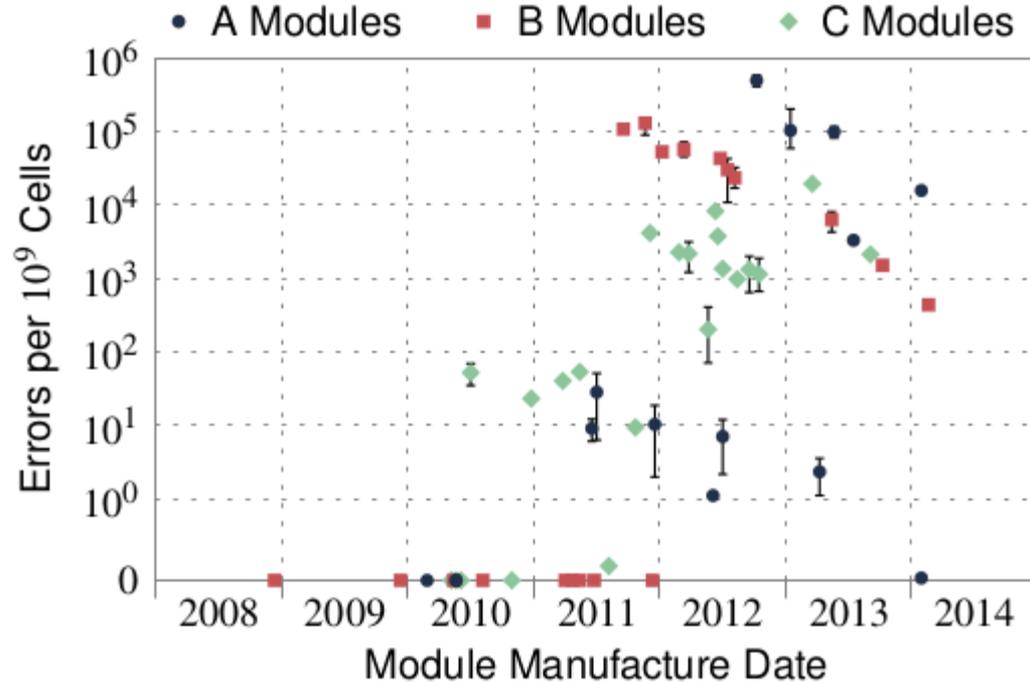
Summer:

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

-- Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, Onur Mutlu, at CMU

- 5th September: Read the paper
- 9th September: Repro'd bit flips on spare laptop using Memtest
- Also tested some desktops
 - but they had ECC -- a pretty good mitigation

DRAM badness by year



(Graph from Kim et al)

Figure 3. Normalized number of errors vs. manufacture date

How to row hammer on x86

code1a:

```
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
// mfence // In CMU paper, but not actually needed
jmp code1a
```

- Requirement #1: **Bypass the cache** → x86 CLFLUSH instruction
 - Unprivileged instruction
 - No way to disable it (unlike e.g. RDTSC)

How to row hammer on x86

code1a:

```
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
// mfence // In CMU paper, but not actually needed
jmp code1a
```

- Requirement #2: Search for bad rows
 - Some DRAM modules have more bad rows than others
 - Allocate big chunk of memory, try many addresses

How to row hammer on x86

code1a:

```
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
// mfence // In CMU paper, but not actually needed
jmp code1a
```

- DRAM is divided into **banks** → each has its own **current row**
- Requirement #3: Pick ≥ 2 addresses
 - Map to **different rows** in the **same bank**
 - “Row-conflict address pair”

Row-conflict address pairs

Could use physical addresses

- Memtest: runs in supervisor mode (bare metal)
- On Linux: could use `/proc/$PID/pagemap`

CMU paper uses: $Y = X + 8\text{MB}$

Row #	Bank 0	Bank 1	Bank 2	...	Bank 7
0	0	0x2000	0x4000	...	0xe000
1	0x10000	0x12000	0x14000	...	0x1e000
...
128...	0x800000	0x802000	0x804000	...	0x80e000

Row-conflict address pairs

Pick address pairs **randomly**

- 8 banks → 1/8 chance of getting a row-conflict pair
- Insight on 18th Sept, ~2 weeks after reading paper
 - Repro'd bit flips in userland, under Linux

Row #	Bank 0	Bank 1	Bank 2	...	Bank 7
0	0	0x2000	0x4000	...	0xe000
1	0x10000	0x12000	0x14000	...	0x1e000
2	0x20000	0x22000	0x24000	...	0x2e000
3...	0x30000	0x32000	0x34000	...	0x3e000

Address selection

Refinement: Try hammering >2 addresses, e.g. 4 or 8

- Tests more rows at a time
- Increases chances of row conflicts
- Hardware can often queue multiple accesses

Row #	Bank 0	Bank 1	Bank 2	...	Bank 7
0	0	0x2000	0x4000	...	0xe000
1	0x10000	0x12000	0x14000	...	0x1e000
2	0x20000	0x22000	0x24000	...	0x2e000
3...	0x30000	0x32000	0x34000	...	0x3e000

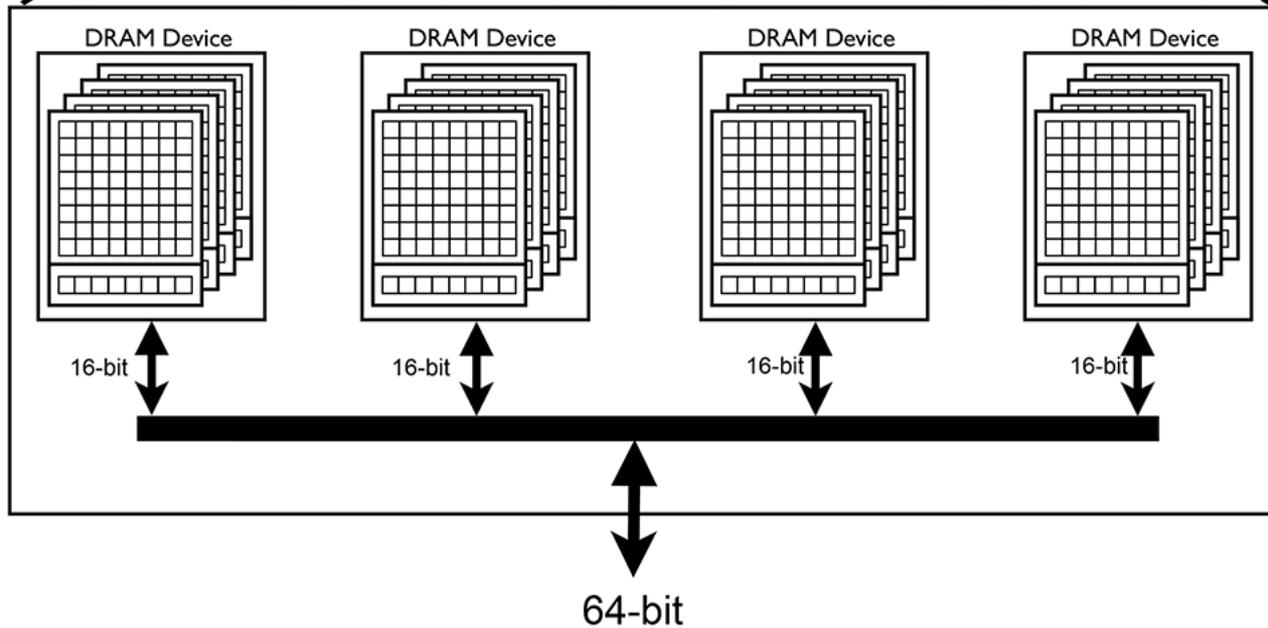
Double-sided row hammering

- Activate *both* neighbours of a row, not just one
- Less data: Existing papers haven't explored this

Row #	Bank 0	Bank 1	Bank 2	...	Bank 7
0	0	0x2000	0x4000	...	0xe000
1	0x10000	0x12000	0x14000	...	0x1e000
2	0x20000	0x22000	0x24000	...	0x2e000
3	0x30000	0x32000	0x34000	...	0x3e000
4	0x40000	0x42000	0x44000	...	0x4e000
5...	0x50000	0x52000	0x54000	...	0x5e000

Double-sided row hammering

- Figure out DRAM address mapping:
 - by bit flips observed
 - by timing
- Picking addresses:
 - Using physical addresses -- /proc/PID/pagemap, disabled
 - Huge pages (2MB) -- not disabled
 - Other chunks of contiguous physical memory



(Diagram from
ARMOR
project,
University of
Manchester)

Querying DRAM's SPD data

```
$ sudo decode-dimms
```

```
...
```

```
----== Memory Characteristics ==----
```

Fine time base	2.500 ps
Medium time base	0.125 ns
Maximum module speed	1333MHz (PC3-10666)
Size	4096 MB
Banks x Rows x Columns x Bits	8 x 15 x 10 x 64
Ranks	2

```
...
```

→ 2^{15} rows. Each contains $2^{10} * 64$ bits = 8 kbytes.

Result: rowhammer-test

- <https://github.com/google/rowhammer-test>
- Runs in userland
 - Allocates 1GB, looks for bit flips in this
- Risky: Could corrupt other processes or the kernel
 - In practice, it rarely does

Iteration 4 (after 4.42s)

Took 99.7 ms per address set

Took 0.997074 sec in total for 10 address sets

Took 23.080 nanosec per memory access (for 43200000 memory accesses)

This gives 346614 accesses per address per 64 ms refresh period

Checking for bit flips took 0.104433 sec

Testing more machines

2014 timeline:

- 7th Oct (4.5 weeks in): NaCl exploit working
- 23rd Oct (~7 weeks in): Testing more laptops → got repros

#	Laptop model	Laptop year	CPU family (microarch)	DRAM manufacturer	Saw bit flip
1	Model #1	2010	Family V	DRAM vendor E	yes
2	Model #2	2011	Family W	DRAM vendor A	yes
3	Model #2	2011	Family W	DRAM vendor A	yes
4	Model #2	2011	Family W	DRAM vendor E	no
5	Model #3	2011	Family W	DRAM vendor A	yes
...

Further refinements

- Easy: Use timing to find row-conflict pairs
 - Find bad rows quicker
- Easy: Hammer for 128ms (= 64ms refresh period * 2)
 - Maximise row activations between refreshes
 - Maximise chance of disturbing a bad row
- Harder: 2-sided row hammering
 - Requires more knowledge of physical addresses

Exploitability

- Systems rely on memory staying constant!
- Two exploits:
 - Native Client (NaCl) sandbox in Chrome
 - bit flip in validated-to-be-safe code
 - easier: can read code to see bit flips
 - Linux kernel privilege escalation
 - bit flip in page table entries (PTEs)
 - gain RW access to a page table
- Dense data structures

Intro to Native Client (NaCl)

- Sandbox for running native code (C/C++)
- Part of Chrome
- Similar to Asm.js, but code generator is not trusted
- “Safe” subset of x86 -- “Software Fault Isolation”
 - Executable (*nexe*) checked by x86 validator
 - But it allowed CLFLUSH -- “*safe in principle*”
- Two variants:
 - PNaCl, on open web. Runs *pexe* (LLVM bitcode): compiled to *nexe* by in-browser *translator*. No CLFLUSH?
 - NNaCl, in Chrome Web Store. Could use CLFLUSH.
- Disclosure: I work on NaCl :-)

NaCl exploit

Safe instruction sequence:

```
andl $~31, %eax // Truncate address to 32 bits
                // and mask to be 32-byte-aligned.
addq %r15, %rax // Add %r15, the sandbox base address.
jmp  *%rax      // Indirect jump.
```

NaCl sandbox model:

- Prevent jumping into the middle of an x86 instruction
- Indirect jumps can only target 32-byte-aligned addresses

NaCl exploit

Bit flips make instruction sequence unsafe:

```
andl $~31, %eax // Truncate address to 32 bits
                // and mask to be 32-byte-aligned.
addq %r15, %rax // Add %r15, the sandbox base address.
jmp *%rax      // Indirect jump.
```

e.g. %eax → %ecx

- Allows jumping to a non-32-byte-aligned address

NaCl exploit

Bit flips make instruction sequence unsafe:

```
andl $~31, %eax // Truncate address to 32 bits
                // and mask to be 32-byte-aligned.
addq %r15, %rax // Add %r15, the sandbox base address.
jmp  *%rax      // Indirect jump.
```

- Create many copies of this sequence -- `dyncode_create()`
 - Look for bit flips -- code is readable
- Exploit handles changes to register numbers
 - Can exploit 13% of possible bit flips
 - Test-driven development

NaCl sandbox address space

Total size: 1GB or 4GB

stack (initial thread)	read+write	
available for mmap()	anything but exec	
nexe rldata segment	read+write	variable size
nexe rodata segment	read	variable size
dynamic code area	read+exec	~256MB
nexe code segment	read+exec	variable size
NaCl syscall trampolines	read+exec	64k
zero page	no access	64k

Hiding unsafe code in NaCl

Existing technique for exploiting non-bundle-aligned jump:

```
20ea0: 48 b8 0f 05 eb 0c f4 f4 f4 f4
      movabs $0xf4f4f4f40ceb050f, %rax
```

This conceals:

```
20ea2: 0f 05      syscall
20ea4: eb 0c      jmp ...    // Jump to next hidden instr
20ea6: f4        hlt       // Padding
```

NaCl mitigations

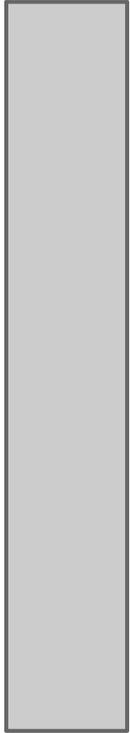
- Disallow CLFLUSH
- Hide code?
 - Might not help

Kernel exploit

- x86 page tables entries (PTEs) are **dense and trusted**
 - They control access to physical memory
 - A bit flip in a PTE's physical page number can give a process access to a different physical page
- Aim of exploit: Get access to a page table
 - Gives access to all of physical memory
- Maximise chances that a bit flip is useful:
 - Spray physical memory with page tables
 - Check for useful, repeatable bit flip first



...

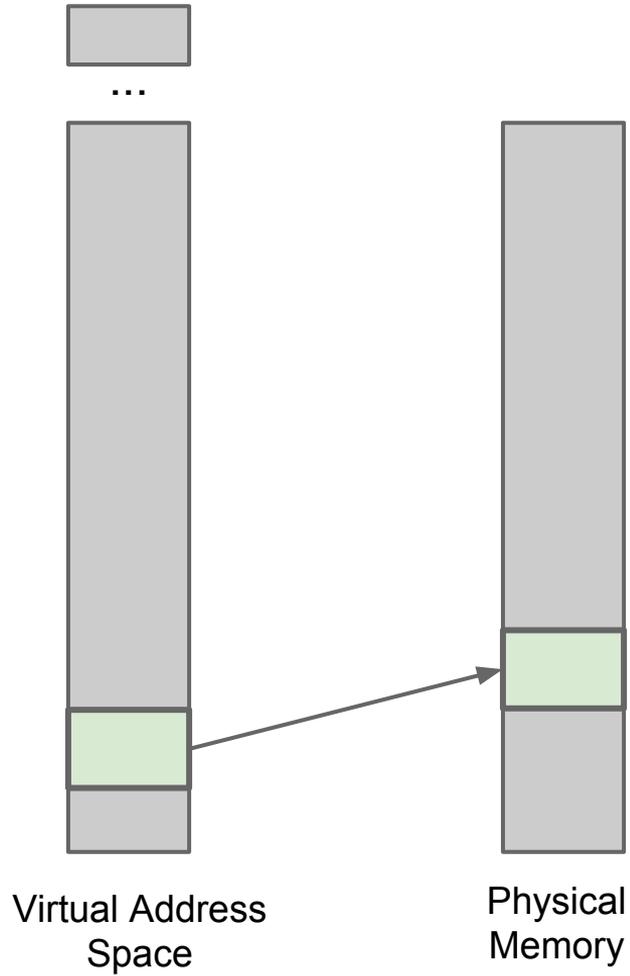


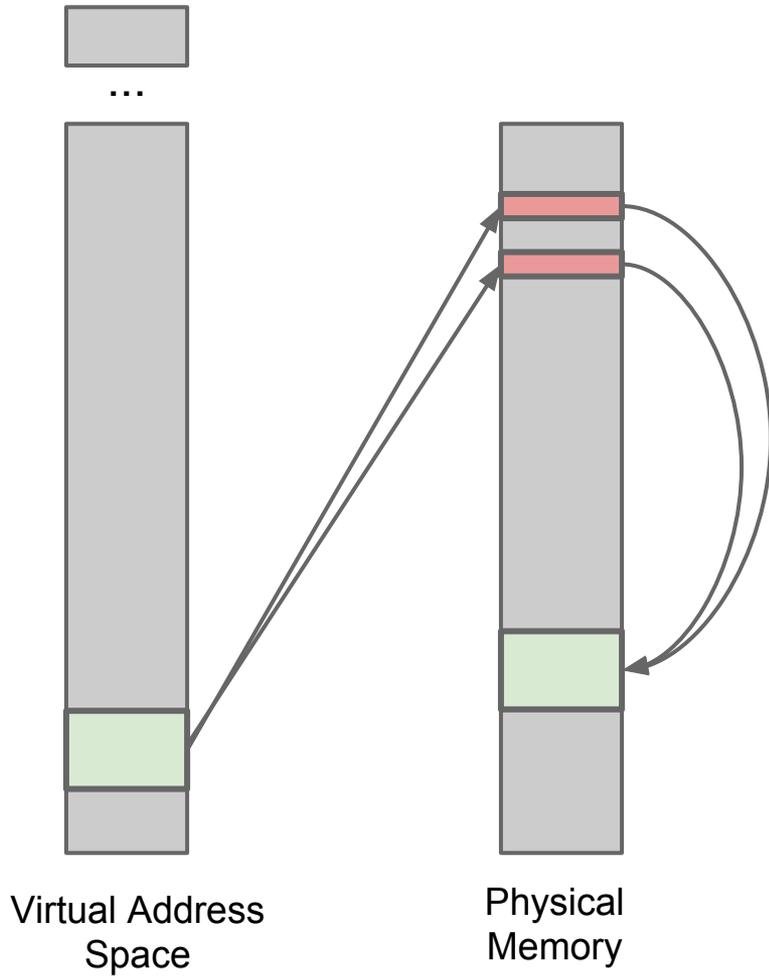
Virtual Address
Space



Physical
Memory

What happens when we map a file with read-write permissions?

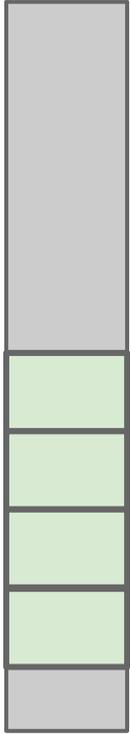




What happens when we map a file with read-write permissions? Indirection via page tables.



...



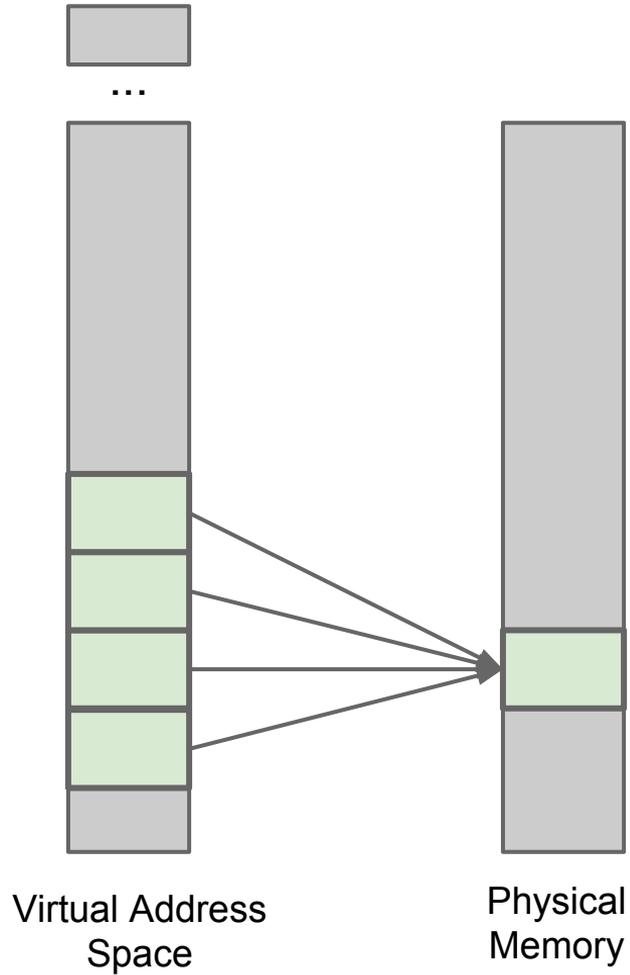
Virtual Address
Space

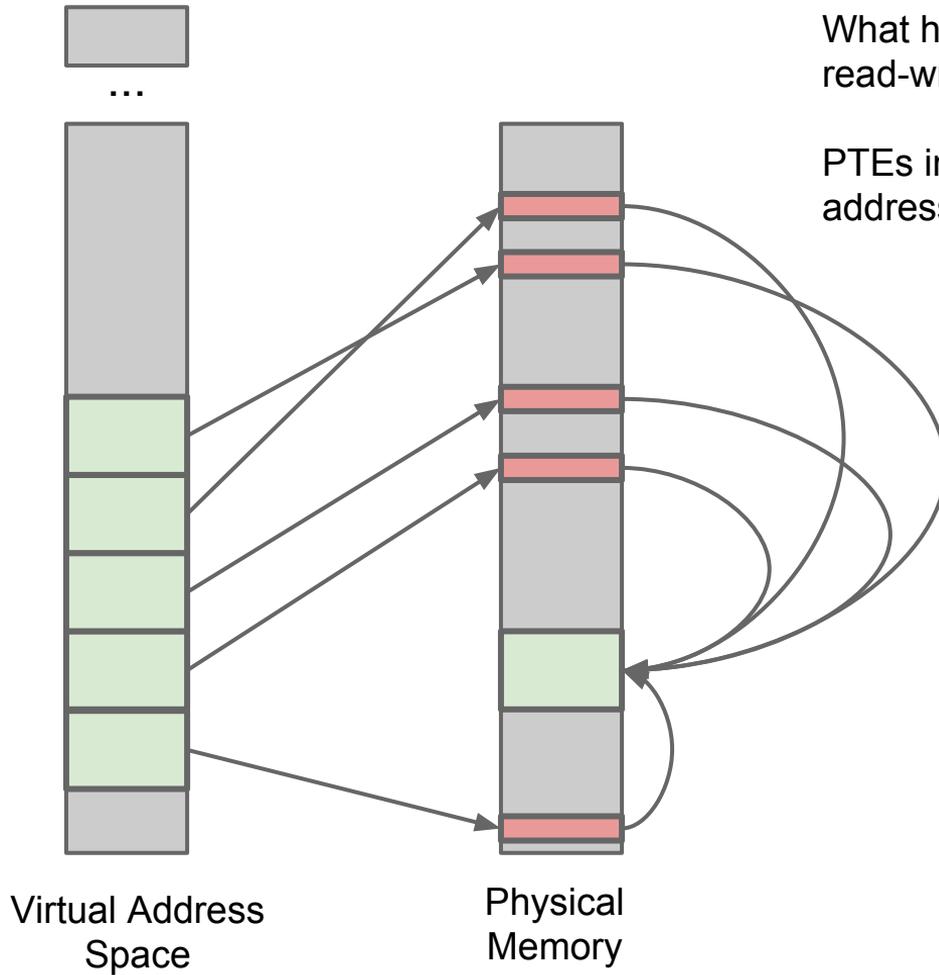
What happens when we repeatedly map a file with
read-write permissions?



Physical
Memory

What happens when we repeatedly map a file with read-write permissions?



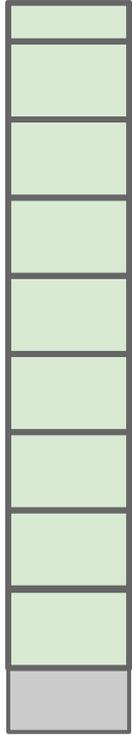


What happens when we repeatedly map a file with read-write permissions?

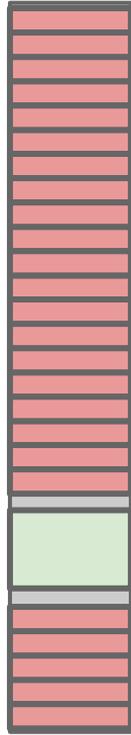
PTEs in physical memory help resolve virtual addresses to physical pages.



...



Virtual Address
Space



Physical
Memory

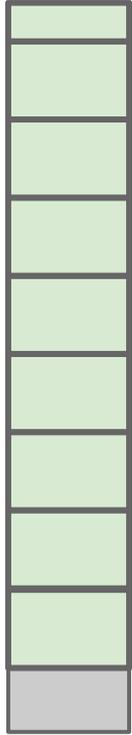
What happens when we repeatedly map a file with read-write permissions?

PTEs in physical memory help resolve virtual addresses to physical pages.

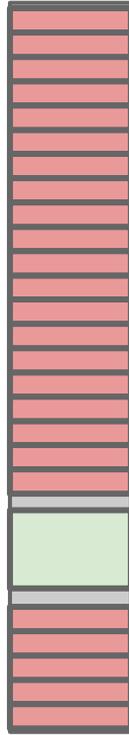
We can fill physical memory with PTEs.



...



Virtual Address
Space



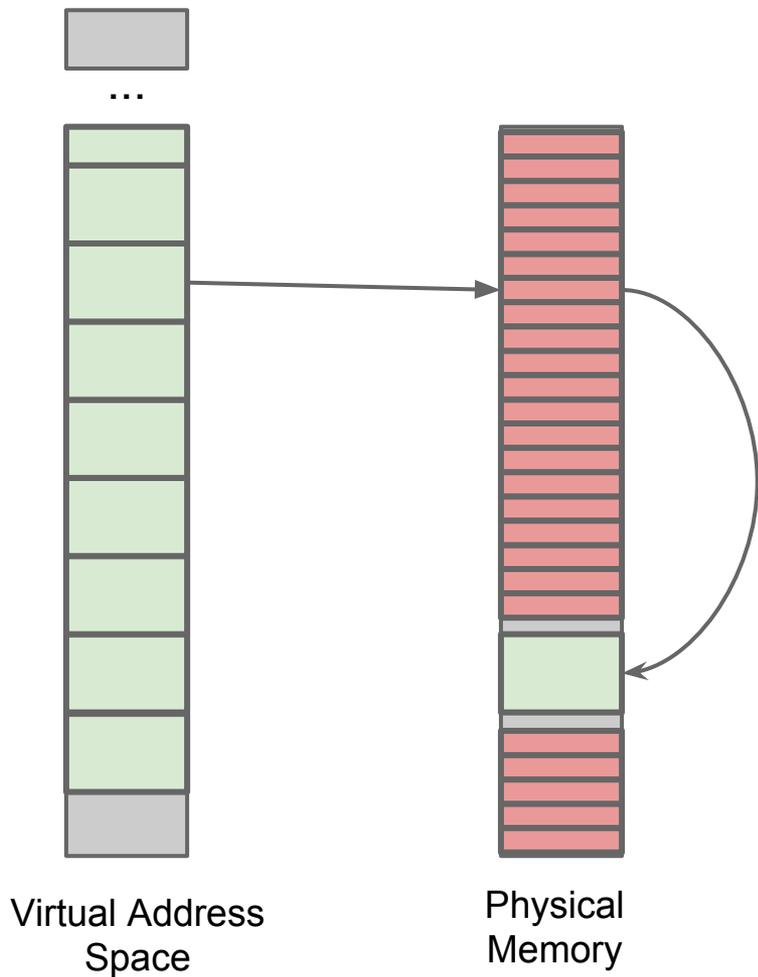
Physical
Memory

What happens when we repeatedly map a file with read-write permissions?

PTEs in physical memory help resolve virtual addresses to physical pages.

We can fill physical memory with PTEs.

Each of them points to pages in the same physical file mapping.



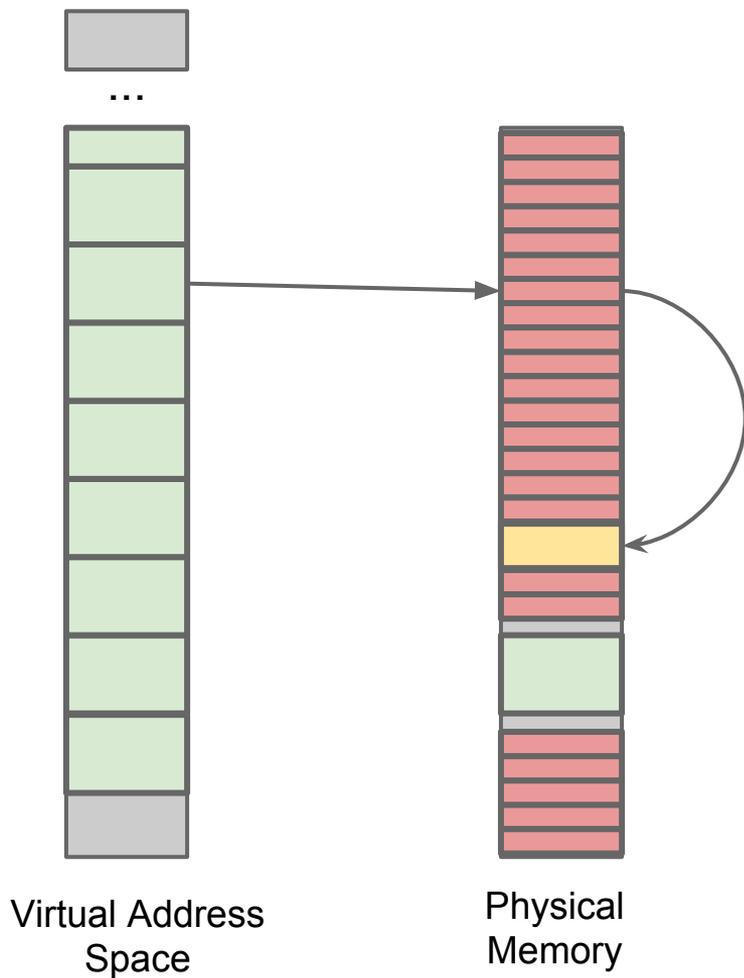
What happens when we repeatedly map a file with read-write permissions?

PTEs in physical memory help resolve virtual addresses to physical pages.

We can fill physical memory with PTEs.

Each of them points to pages in the same physical file mapping.

If a bit in the right place in the PTE flips ...



What happens when we repeatedly map a file with read-write permissions?

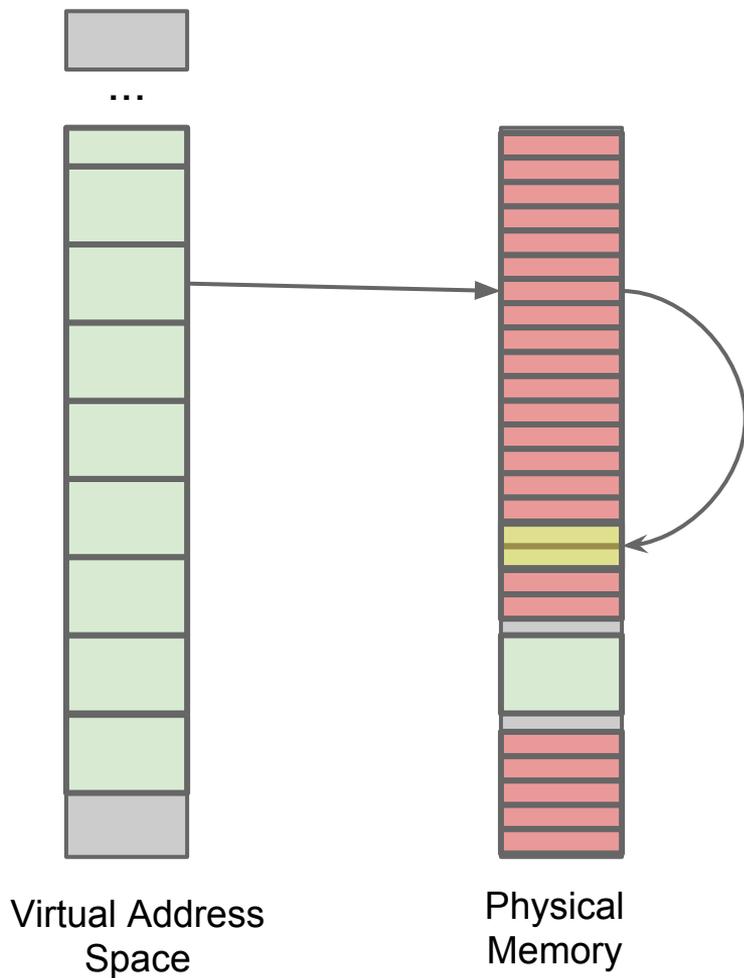
PTEs in physical memory help resolve virtual addresses to physical pages.

We can fill physical memory with PTEs.

Each of them points to pages in the same physical file mapping.

If a bit in the right place in the PTE flips ...

... the corresponding virtual address now points to a wrong physical page - with RW access.



What happens when we repeatedly map a file with read-write permissions?

PTEs in physical memory help resolve virtual addresses to physical pages.

We can fill physical memory with PTEs.

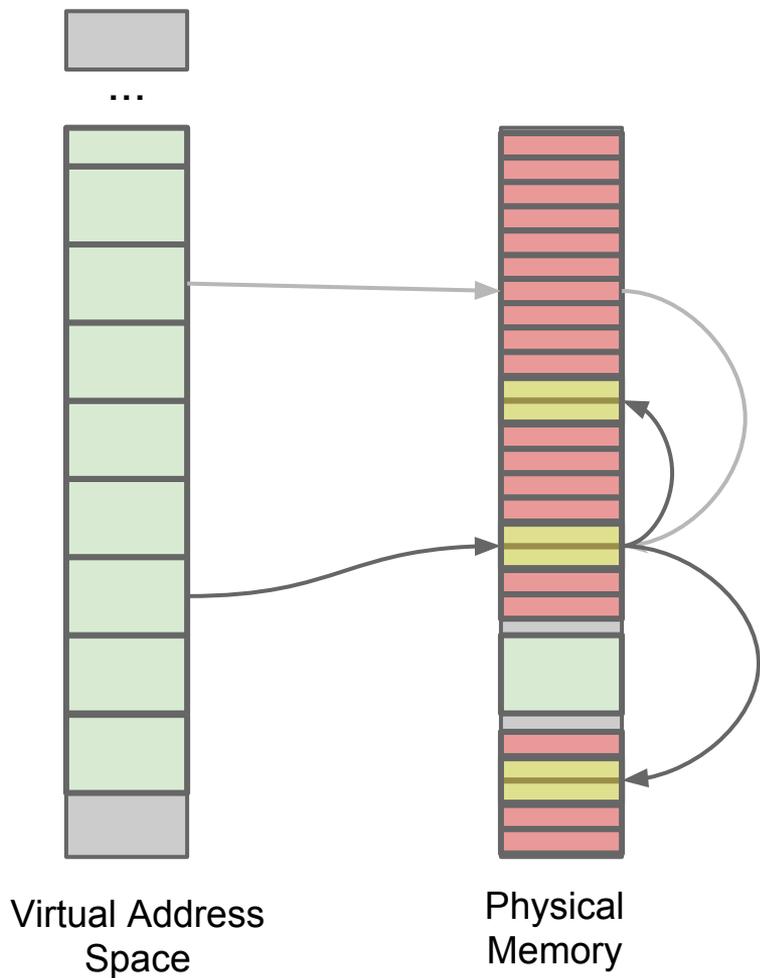
Each of them points to pages in the same physical file mapping.

If a bit in the right place in the PTE flips ...

... the corresponding virtual address now points to a wrong physical page - with RW access.

Chances are this wrong page contains a page table itself.

An attacker that can read / write page tables ...



What happens when we repeatedly map a file with read-write permissions?

PTEs in physical memory help resolve virtual addresses to physical pages.

We can fill physical memory with PTEs.

Each of them points to pages in the same physical file mapping.

If a bit in the right place in the PTE flips ...

... the corresponding virtual address now points to a wrong physical page - with RW access.

Chances are this wrong page contains a page table itself.

An attacker that can read / write page tables can use that to map **any** memory read-write.

Exploit strategy

Privilege escalation in 7 easy steps ...

1. Allocate a large chunk of memory
2. Search for locations prone to flipping
3. Check if they fall into the “right spot” in a PTE for allowing the exploit
4. Return that particular area of memory to the operating system
5. Force OS to re-use the memory for PTEs by allocating massive quantities of address space
6. Cause the bitflip - shift PTE to point into page table
7. Abuse R/W access to all of physical memory

In practice, there are many complications.

... but wait ...

In theory, theory and practice are the same.

In practice, there are many complications.

Exploit strategy

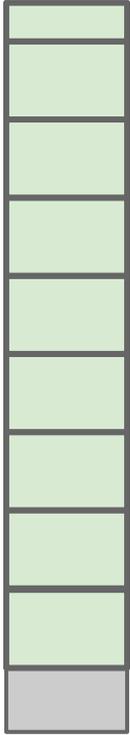
Privilege escalation in 7 easy steps ...

1. Allocate a large chunk of memory
2. Search for locations prone to flipping
3. Check if they fall into the “right spot” in a PTE for allowing the exploit
4. Return that particular area of memory to the operating system
5. Force OS to re-use the memory for PTEs by allocating massive quantities of address space
6. Cause the bitflip - **shift PTE to point into page table**
7. Abuse R/W access to all of physical memory

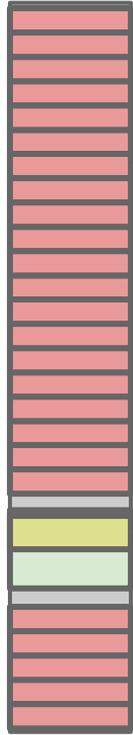
In practice, there are many complications.



...



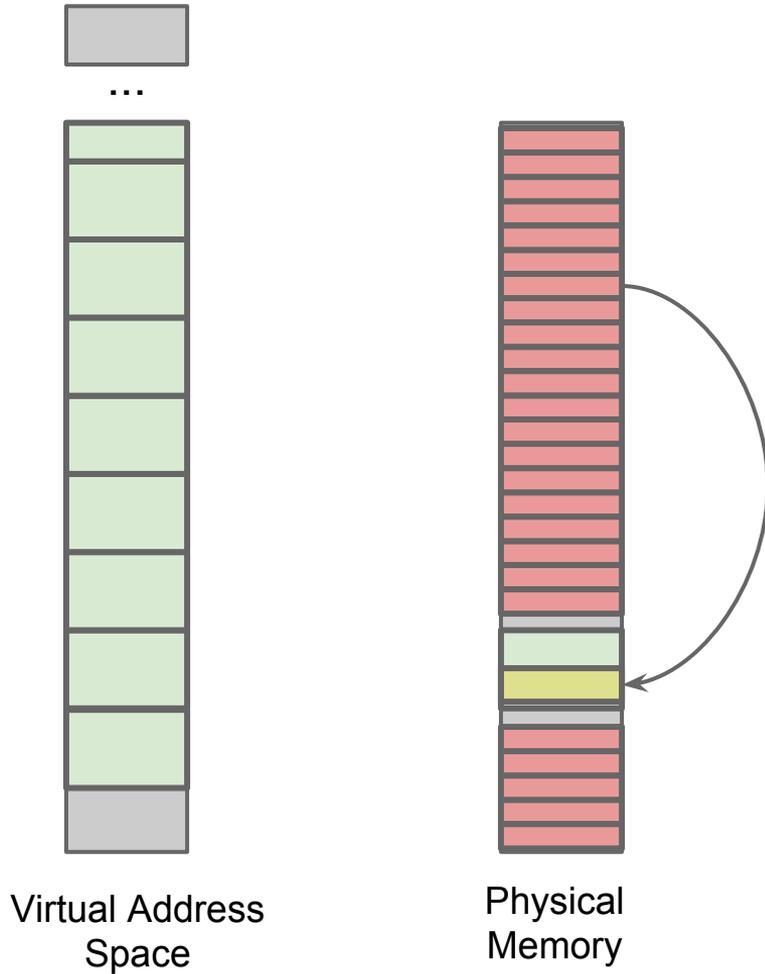
Virtual Address Space



Physical Memory

In practice there are many complications.

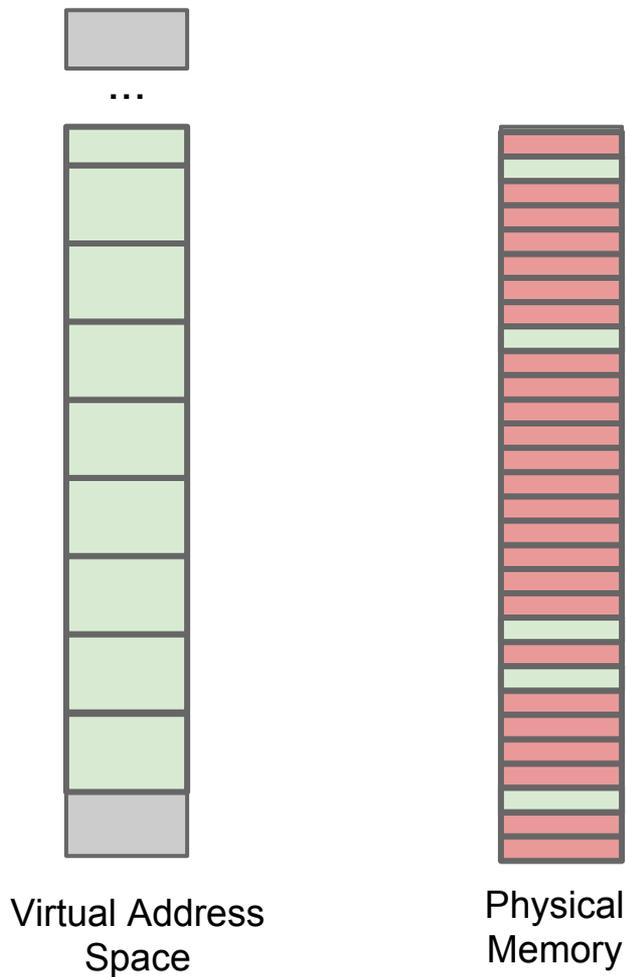
The biggest one: **If** the file is contiguous in physical memory, **and** one of the lower bits flip ...



In practice there are many complications.

The biggest one: **If** the file is contiguous in physical memory, **and** one of the lower bits flip ...

... we shift where the PTE points to, but that may still point to our mapped file - which doesn't help us.



In practice there are many complications.

The biggest one: **If** the file is contiguous in physical memory, **and** one of the lower bits flip ...

... we shift where the PTE points to, but that may still point to our mapped file - which doesn't help us. We had RW access to our mapped file beforehand.

Solution: Aggressively fragment the file data across physical memory.

100011101110001101001100100010000001001001010101.1001110110011 4 MB
10.D1001010000000010011100111001D11D001010100.00D111000111.010111 4 MB
00D11000011D0111011.D1110011101100100010101010010100D110001110011100 4 MB
1001000001011111000100010000101100D000110110101000100011011010101 4 MB
01000D11000101110101001000100011001101100100110110110110110000D10 5 MB
0100101000D0011011DD0011001101011111D110001010111D000001100D0110 5 MB
1111101011001D01010D1000001000 5 MB
00D10101101000001011011001011010000000000000000000000000000000000000 5 MB
1101D01100111101010D00110111100101110101D010101110111101010010 5 MB
0011DD010001000101010D0011100000.101D1100000111D0011001011100 6 MB
110101DD1111010100001001DD010001D110001010100D001110D000D00000000 6 MB
0.00D10100100010000010.011111000100000D01000001100D10011D00000111 6 MB
0100100D000D00101001100011011100D00100101010101010101010101010 7 MB
01000100D0000010D00100.1110D001111.010100010110011001001001DD10000 7 MB
110000010111011100000101010111001100011D1101D100101001100110100 7 MB
11D0000101D1010101001D0001D00111001101101100D001010001000.011101 7 MB
.....DD0001000D0010111010101010.00D0111000D00000101DD1010011001001 8 MB
11000010001000D00D11.D000D0010D1D1111010100D01111001000D000101 8 MB
1001D0010011001D1001111.D1111100110110010010011D00000101010010 8 MB
00111111.D00D1001000111001DD00DD1010111DD001D0110000D10100 8 MB
01D00111010000111D10111000D1000001000001111001000011011010.111 9 MB
10100111001110D011DD111011010010100D00101001D010DD11101101010100 9 MB
0001.111D00101101010111101D0001100010100D001011001111.0D1011101 9 MB
0110.010001010D0000D00D1000100100111111010001000111011101110111 9 MB
1010101DD01010111D000D001D00101D001010011D1101101001001110001 10 MB
000100001011D100000001111101000001D000101111000001DD01010001 10 MB
00110000D01100001111000011D00111000D1111100D0101D1011010191. 10 MB
011010010111011000D1111000D10.D0DD00D10010010010110.01001011D.D00 10 MB
0110D011100010D10111010101100D00111000D1101001000000001111D1 11 MB
010D11101D111010001100001001011001D1010DD001000001100000011100 11 MB
111001D11100010100101010000D0101D101D1110000000101111001010D000 11 MB
10001111DD01110001100000000111100D11D00000101011.01101101110 11 MB
.....DDDDDD101010101.101010.00101.0101.010101011.0101010101 12 MB
101001 12 MB
1.010101010101.01010101.010101010101010101010101.010101010101 12 MB
010101010101010101010101.010101010101001.0101010101010001.01010 12 MB
..DDDDDDDDDDDDDD.01 13 MB
01010101010101.010101010101.0010101010101010101010101010101.010 13 MB
10.0010101.0101.0101.01010101010101.0101010101010101010101010101 13 MB
01010101010101010101010101010101.0101.01010101.0101010101000101 13 MB
DD01110000011DD010111010101110100DD100D0011D10101D0000001010 14 MB
111111.101D1000D0100000101110110101001010000D110000D11DD0001010100 14 MB
D001000111110101100000D1D1010001100D101D001000D10011111100D00 14 MB
0101000001D001010010001100110111010100100101101111D11DD010001 14 MB
0000D0100000D1010100001D111D0010D00100D0010000111001011100011000 15 MB
1D0101000101110000D110111100110010100D0101000001100001101100000 15 MB
000100DD10110100DD011011100100010010.100011000110111000000011101 15 MB
D01000D0111001D1000011D1D10110100DD0100100010111D101100011.... 15 MB

4 MB
4 MB
4 MB
4 MB
5 MB
5 MB
5 MB
5 MB
6 MB
6 MB
6 MB
6 MB
7 MB
7 MB
7 MB
8 MB
8 MB
8 MB
8 MB
9 MB
9 MB
9 MB
10 MB
10 MB
10 MB
10 MB
11 MB
11 MB
11 MB
11 MB
11 MB
12 MB
12 MB
12 MB
12 MB
12 MB
13 MB
13 MB
13 MB
14 MB
14 MB
14 MB
15 MB
15 MB
15 MB
15 MB

..... 28 MB
..... 28 MB
..... 28 MB
.....011011.101001010100001000010D110 29 MB
1D11100100110011010101101100D0110D0001100D00110001.10001001000000D 29 MB
1111.1100010100100000101.000D01000D0010001000D001010D0111.0D11.D10 29 MB
00011010.0110010D1010101001100010000D11010001000D01111D10.00 29 MB
01D01DD01111011011D00D10001101D1001DD001001D11001D100110111000. 30 MB
101100D11100D100D101111D0101010001D0101011..... 30 MB
..... 30 MB
..... 31 MB
..... 31 MB
..... 31 MB
..... 31 MB
1111111D11.111001100111DD101111111D11110000D101DD0101111111D1 32 MB
1.11110110D01101110110D1111111D1D110011110011111101001D011 32 MB
DD1111111D010111111100D01110110.00D101DD1111D111.1011111 32 MB
1D11101111100D11111010D111DD0.D11101010111101D1101D100011 32 MB
100000111111111010DD1110DD10110111111101111D.011101DD1101.D11 33 MB
111101011111110110110100101101100111010D11000101101010 33 MB
1.01110.111011DD1101.01D1010010010110011111010D0001111111D 33 MB
.01D1D10D11011001D11.1011110D111D00D00D11D11D101011011.D1 33 MB
00D100D100000000100D1111010001D0100000111010D1010.0100D0000 34 MB
DDDDDDDDDDDDDD.11111111.0111.1101.1.11.101.011010101.1111 34 MB
111111.11111.111.01111.11011011110111101111101111101111011.10 34 MB
110101111110111111111110101010101010101010101010101010101010 34 MB
101.01 35 MB
0101010101010101..... 35 MB
..... 35 MB
D00D001110010D0D1001110100001000011DD0101000010010.00111011000 36 MB
0D100100D00D1101010D0000D.DD110111D000101010DD01011101010D00111 36 MB
11001.0011010111D10000010D110D000.10DD1011011100100001D10D10 36 MB
00D1000001000DD11101000D00100011100DD0011110000000DD101D011000 36 MB
D1111100110000D01000D1010DD000111011D0111000100000000101101 37 MB
1001100100100D0110D1011.01D01100101D10101.000000000010101.000 37 MB
D01101101101100111110101001D110D11000010DD1011011100101110 37 MB
00000D1D1001000D01101000D1.D0111000110D00110D010D111D00 37 MB
0011110111D00DD11000100011001111101D1.1100.01010111100010.01 38 MB
D00D00D10000010D110.111010001.1000010000011D001100DD00100110D0 38 MB
010011011.D001D100110.1D10D11000101D1100000.D11010001D000D1001 38 MB
001D00DD0100D.0D11111000100D00001110D100000.1D1010001100D100 38 MB
110D1110100000010011D1D000100D101DD101111D11110000100D00D00D 39 MB
10000D1000001010101000D10001D000D1001010101010DD100D0011010 39 MB
1001D0110D100D10101010011001100000110000111101010100D101D00 39 MB
D11000111010D000DD00011010000D0000001011.010D11111101D1011001 39 MB

28 MB
28 MB
28 MB
29 MB
29 MB
29 MB
29 MB
30 MB
30 MB
30 MB
30 MB
31 MB
31 MB
31 MB
31 MB
32 MB
32 MB
32 MB
32 MB
33 MB
33 MB
33 MB
34 MB
34 MB
34 MB
34 MB
35 MB
35 MB
35 MB
36 MB
36 MB
36 MB
36 MB
37 MB
37 MB
37 MB
38 MB
38 MB
38 MB
39 MB
39 MB
39 MB

Exploit strategy

Privilege escalation in 7 easy steps ...

1. Allocate a large chunk of memory
2. Search for locations prone to flipping
3. Check if they fall into the “right spot” in a PTE for allowing the exploit
4. Return that particular area of memory to the operating system
5. **Force OS to re-use the memory for PTEs by allocating massive quantities of address space**
6. Cause the bitflip - shift PTE to point into page table
7. Abuse R/W access to all of physical memory

In practice, there are many complications.

Exploit strategy

Turns out it is hard to force the OS to re-use “regular” memory for PTEs.

Possible somehow. I spent a few afternoons fumbling around in the Linux physical page allocator. Not very fun code.

Mark was more clever: He simply put the system under memory pressure - when backed into a corner, the OS behaves nicely.

Mitigations

CMU paper: “The industry has been aware of this problem since at least 2012”

- Industry preparing mitigations -- but no security advisories
- ECC (Error Correcting Codes)
- TRR (Target Row Refresh)
- Higher DRAM refresh rates

Mitigation: ECC memory

- Single-bit error correction
- Double-bit error detection
- ≥ 3 bits: not detectable
 - But not very likely?
- Reduces problem to Denial of Service

But only works if you enable proper MCE (Machine Check Exception) handling for ECC errors!

Not ideal: Expensive, and not guaranteed to work

“Ideal” fix: Target Row Refresh

- Count activations of a row
- Refresh neighbouring rows when counter reaches threshold

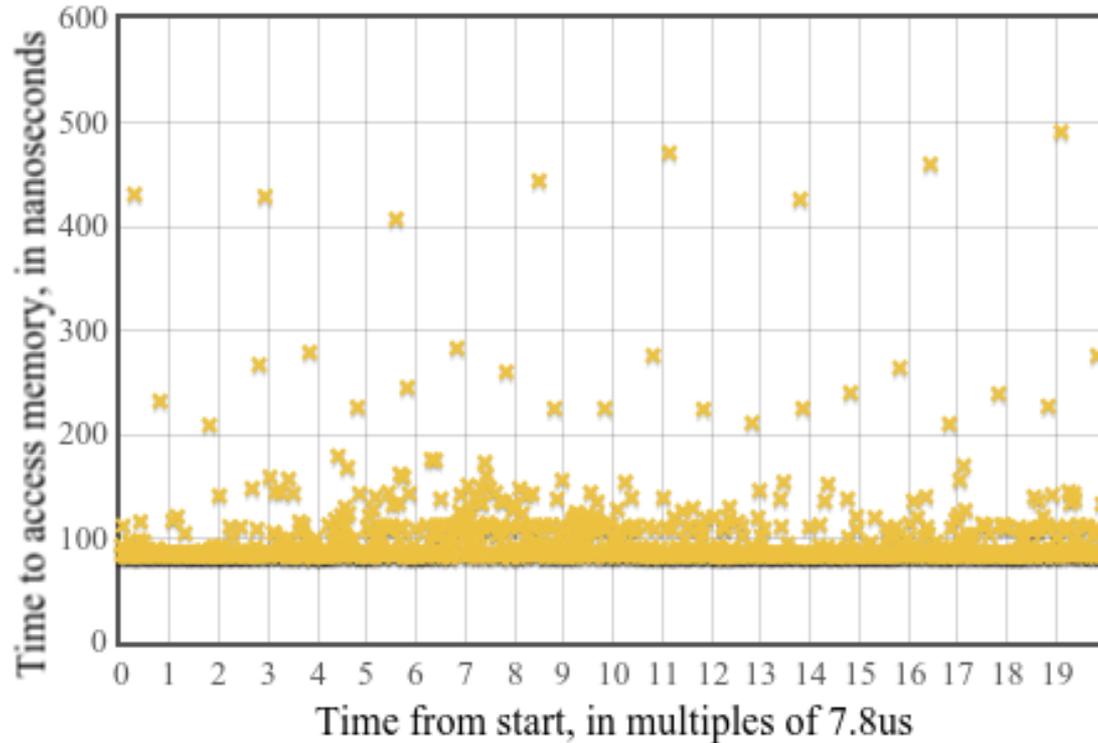
- Covered by LPDDR4
- DDR4 too?
- In DRAM: Micron data sheets
- In memory controllers:
 - pTRR (pseudo TRR)
 - One Intel presentation says Ivy Bridge supports pTRR. No further evidence of this?

Mitigation: 2x refresh rate

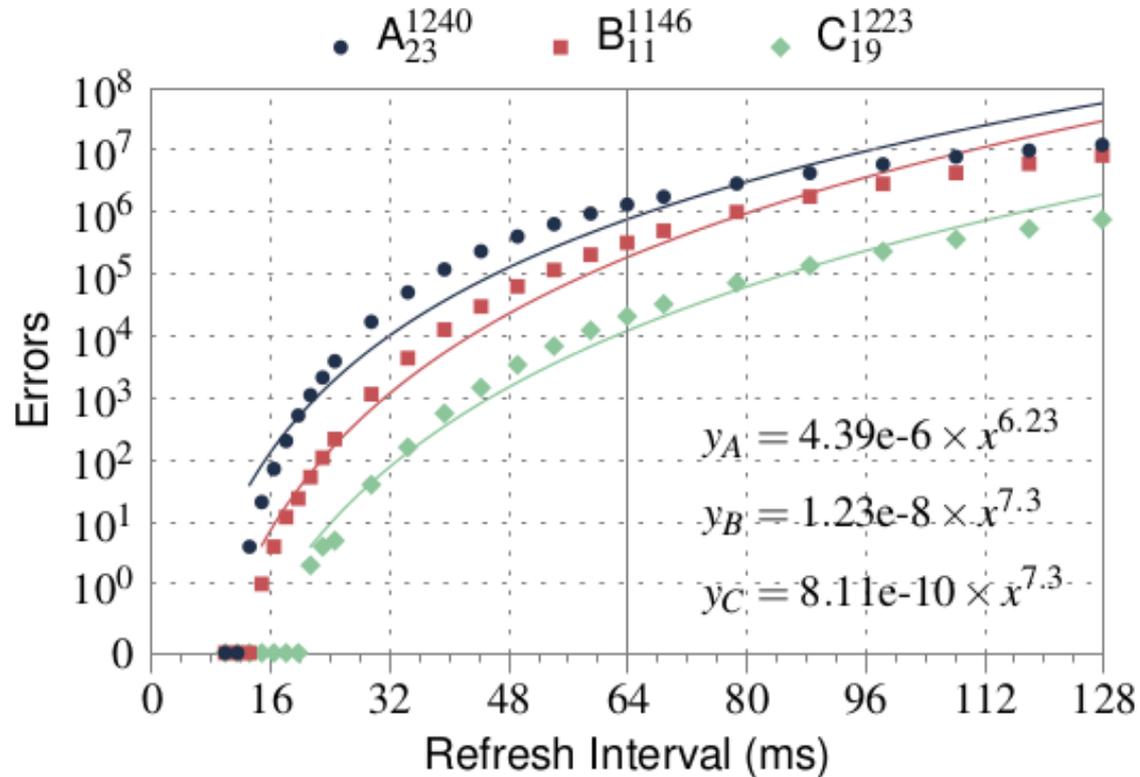
- Current CPUs support this
- tREFI parameter
 - Covered by Intel's public memory controller docs
 - Set by BIOS
 - Coreboot covers Sandy/Ivy Bridge
- Various vendor BIOS updates do this
- How to verify refresh rate?
- Is 2x refresh enough?

Timing DRAM refreshes

From https://github.com/google/rowhammer-test/tree/master/refresh_timing



Is 2x refresh enough?



Graph from
Kim et al

Rowhammer from Javascript?

- Can we do row hammering from Javascript?
 - Via normal cached memory accesses, without CLFLUSH
 - Generate many cache misses
- Javascript engine speed not a problem
 - Near-native access to typed arrays (e.g. Asm.js)
 - Cache misses are slow

- lavados reports doing this

Causing cache misses

- Have to miss at all cache levels (L1, L2, L3)
- Seems difficult?
 - Row hammering by accident in benchmarks (see paper)
 - Not with an inclusive cache!
 - Evicting cache line from L3 evicts from L1 and L2 too
 - Used by Intel CPUs

Cache profiling algorithm

- Find addresses mapping to the same L3 cache set
- e.g. For a 12-way L3 cache, find 13 addresses
 - Accessing these in turn must produce ≥ 1 cache miss

How: **“The Spy in the Sandbox -- Practical Cache Attacks in Javascript”** (Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, Angelos D. Keromytis)

- By timing memory accesses
- Original motivation: L3 cache side channel attacks

Cache eviction policy

- True LRU: would give 13 cache misses per iteration (for 12-way cache)
 - 6.5x reduction in row activations. Not ideal.
- Ideally want 2 cache misses per iteration
- Real CPUs:
 - Sandy Bridge: Bit-Pseudo-LRU, 1 bit per cache line
 - Ivy Bridge: Quad Age LRU, 2 bits per cache line
 - Plus adaptive policy: “set duelling”

Cache side channel mitigations?

- Reduce timer resolution (`performance.now()`)
 - Changes in Firefox, Chrome, Safari/WebKit
- Probably doesn't help
 - Cache profiling just takes longer?
 - Multi-threading: Build Your Own Timer
 - PNaCl
 - SharedArrayBuffers in Javascript
 - WebAssembly
- CPU performance counters?

Unknowns

- ARM and mobile devices? Depends on:
 - Cache organisation
 - Performance of CPU and memory controller
- Damage to DRAM?
 - Anecdotal observations

Conclusions

- As software-level sandboxes get better, attackers will likely target more esoteric bugs, such as hardware bugs
- Rowhammer: not just a reliability problem
- Hard to verify that hardware meets spec
 - Vendors should adopt security mindset
 - Vendors should be more transparent

For more information

Code and notes on Github:

- <https://github.com/google/rowhammer-test>

Mailing list:

- <https://groups.google.com/group/rowhammer-discuss/>