

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?

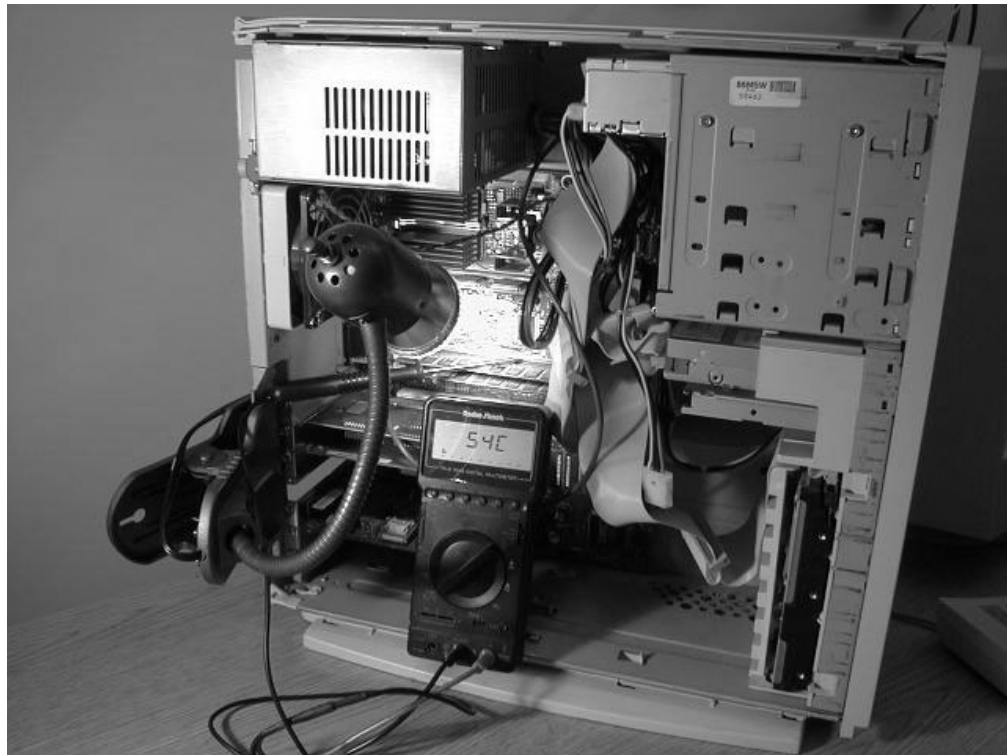
Exploiting random bit flips

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

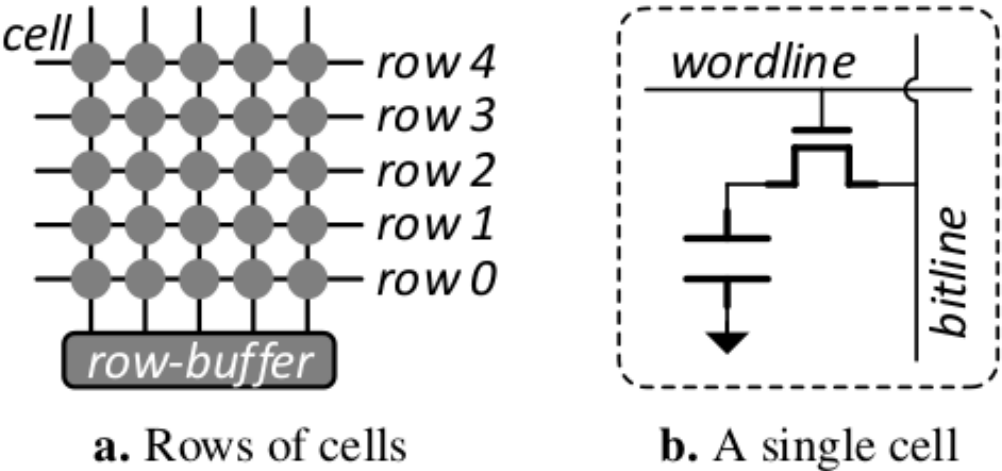
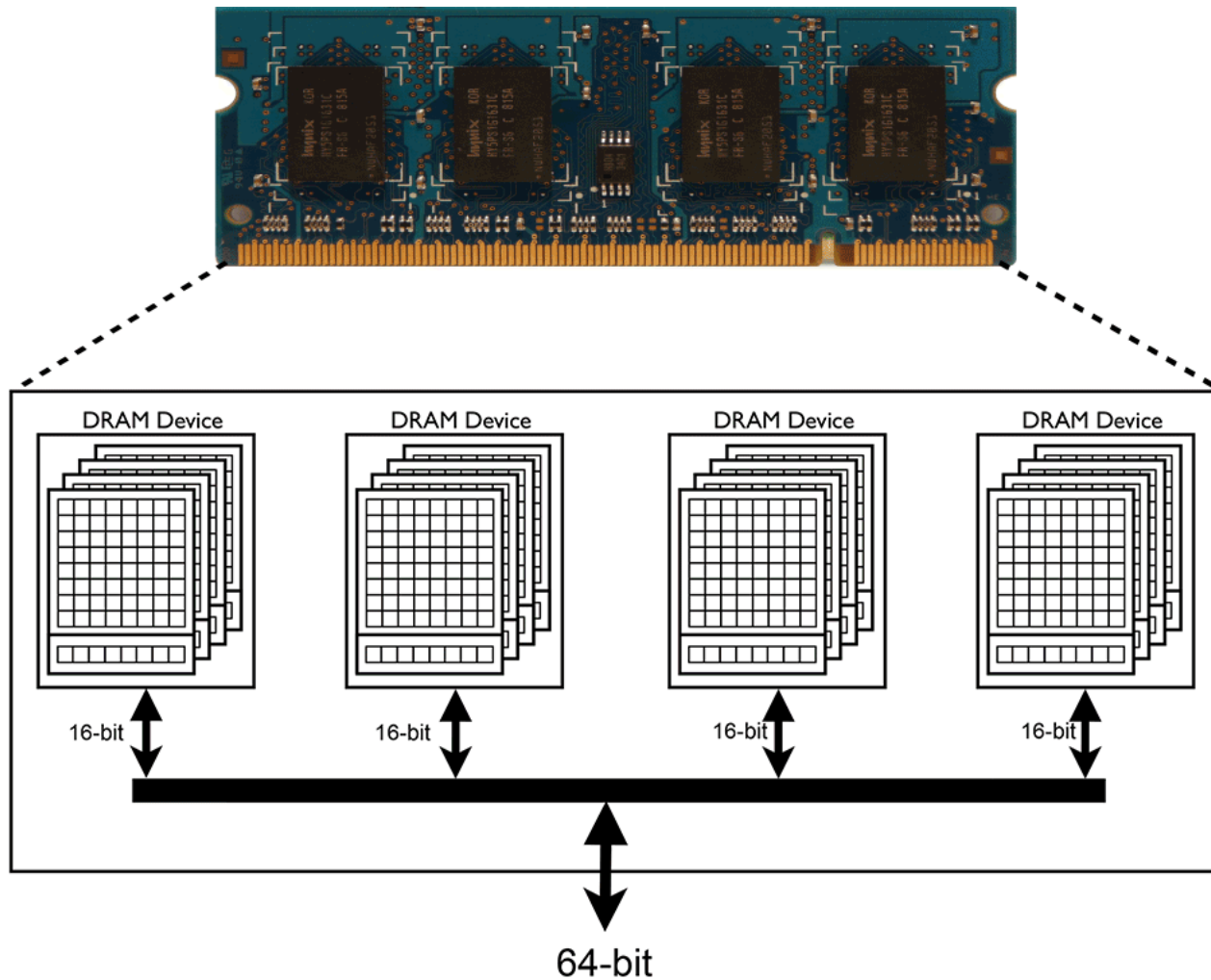


Figure 1. DRAM consists of cells

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

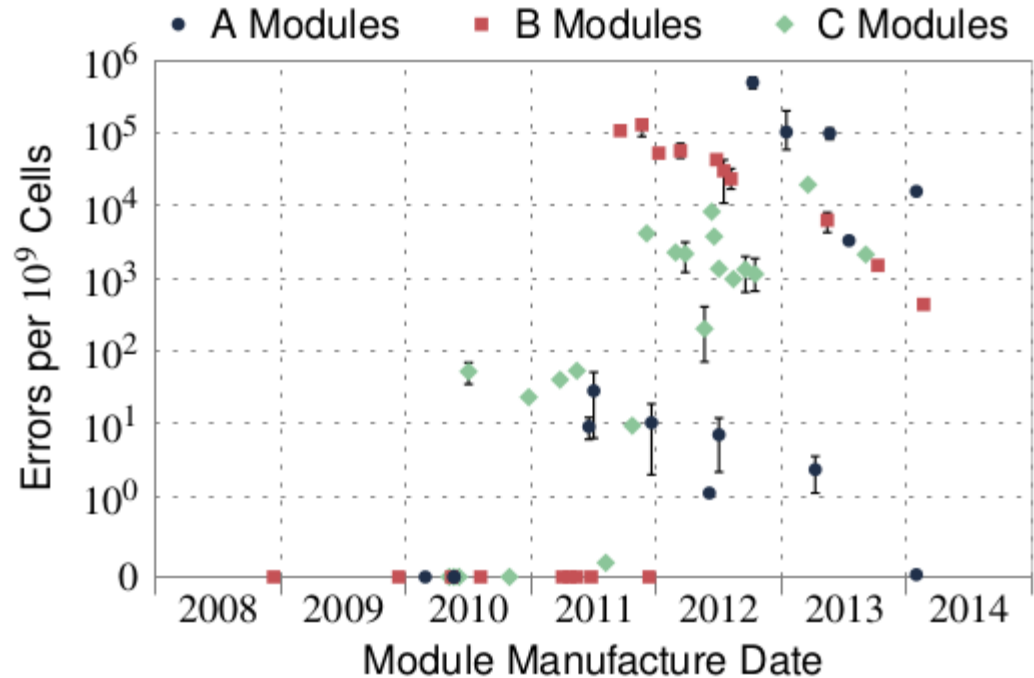
Summer:

“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 $\rightarrow$ 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

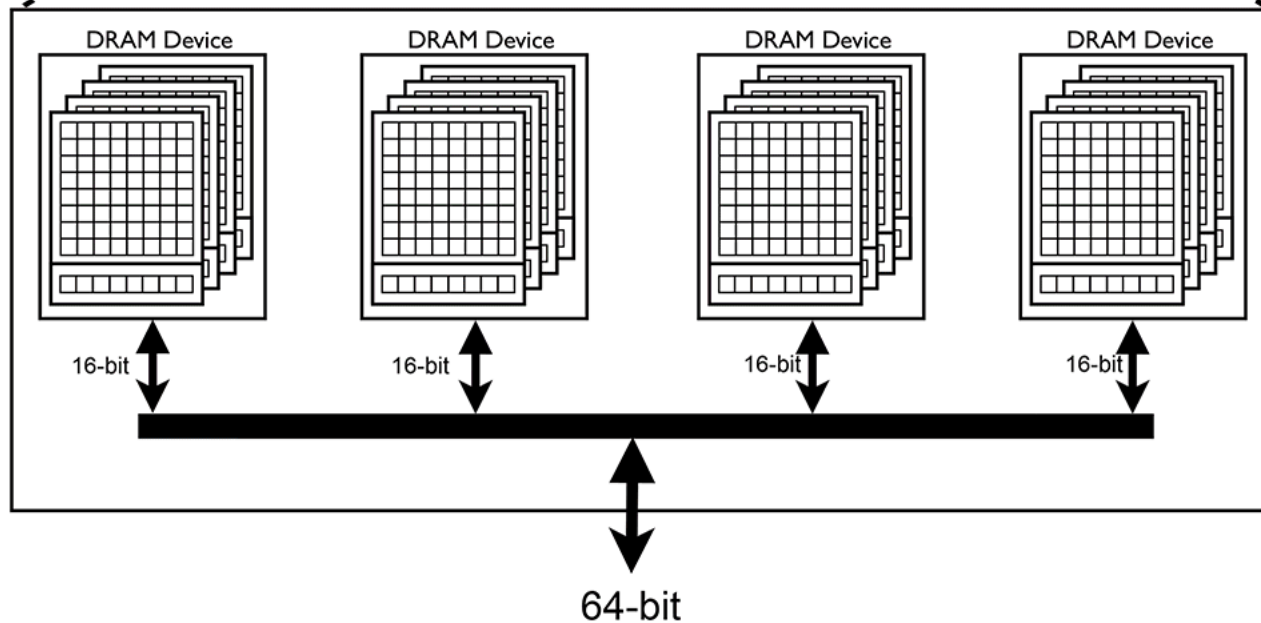
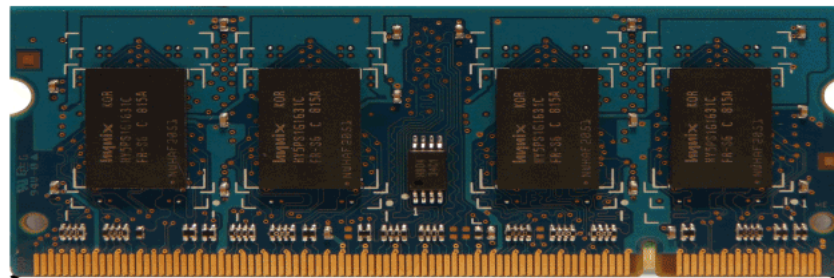
Double-sided row hammering

- 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

x86-64 Page Table Entries (PTEs)

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

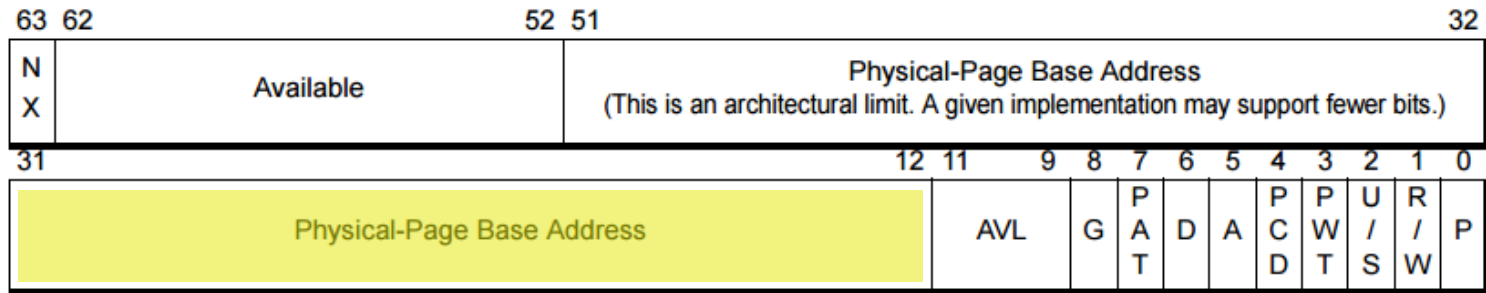
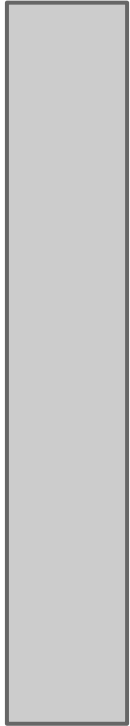


Figure 5-21. 4-Kbyte PTE—Long Mode

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



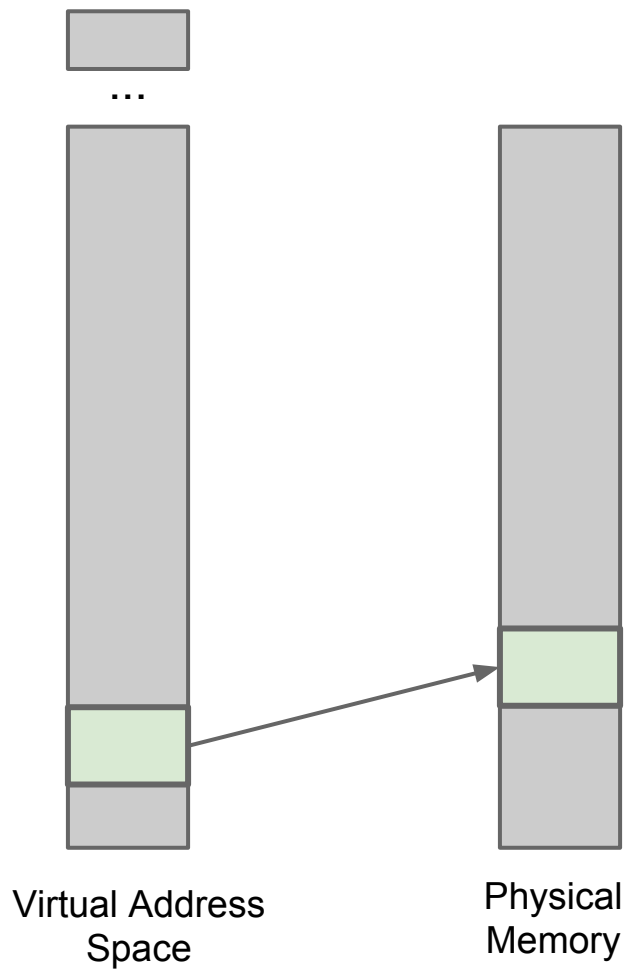
...



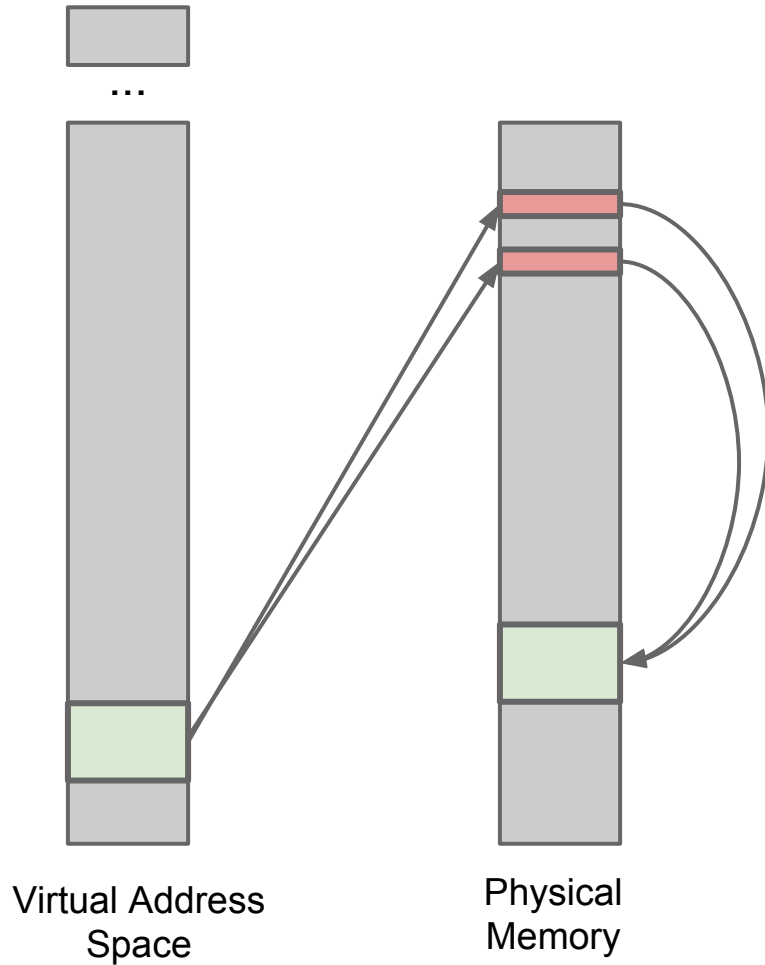
Virtual Address
Space



Physical
Memory



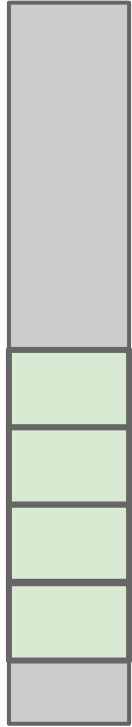
What happens when we map a file with read-write permissions?



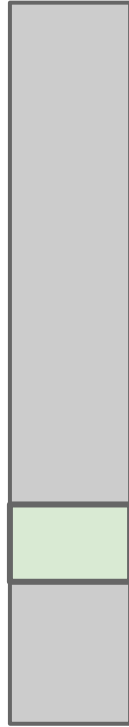
What happens when we map a file with read-write permissions? Indirection via page tables.



...

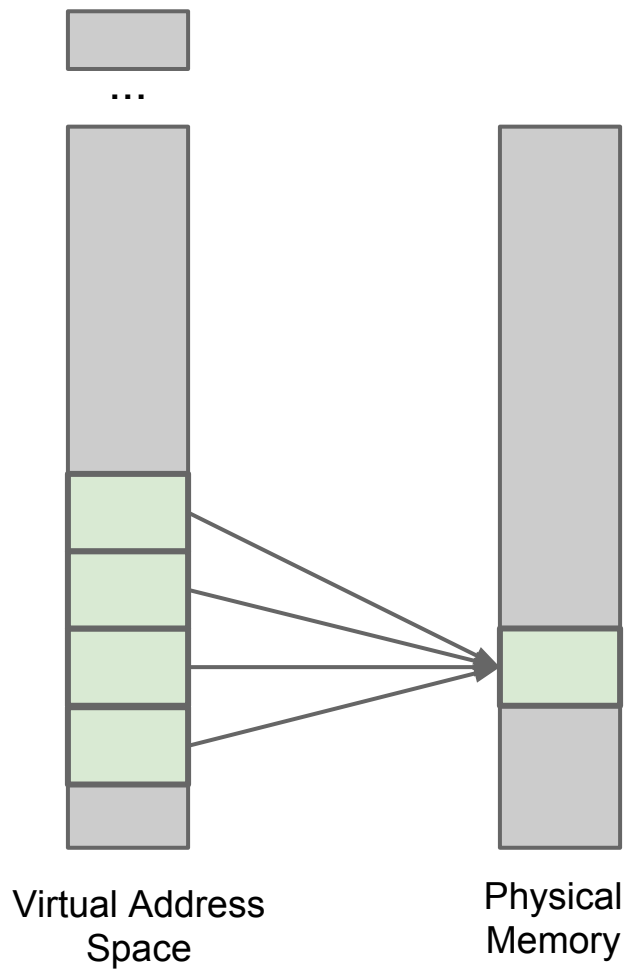


Virtual Address
Space

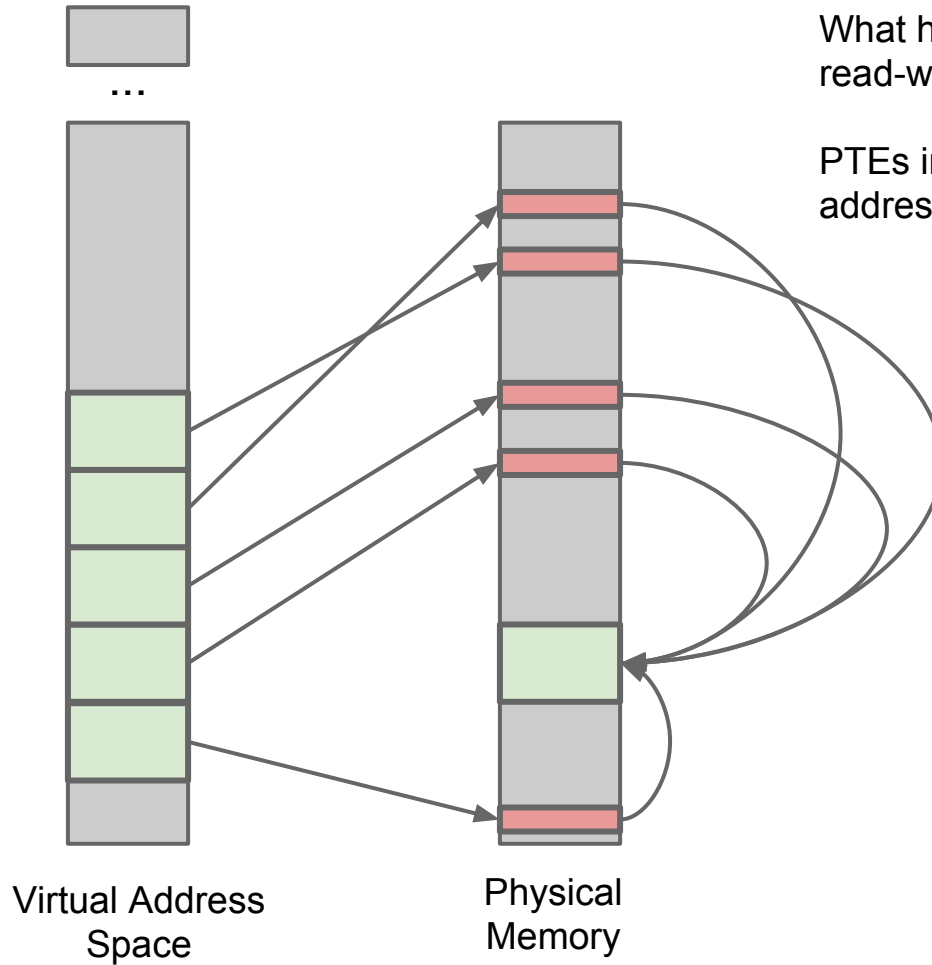


Physical
Memory

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

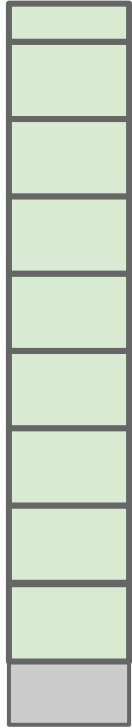


What happens when we repeatedly map a file with read-write permissions?

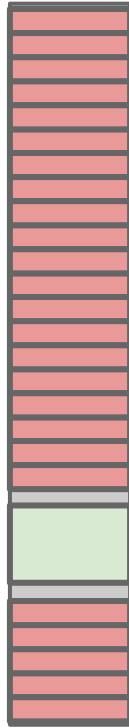




...



Virtual Address
Space



Physical
Memory

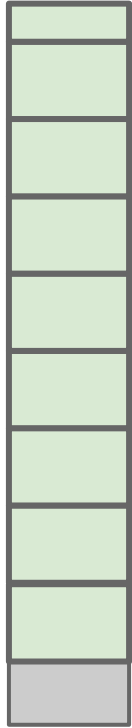
What happens when we repeatedly map a file with read-write permissions?

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

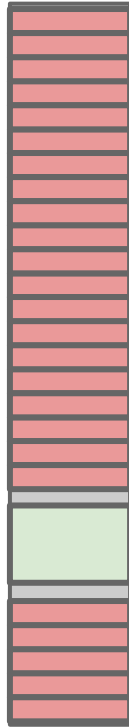
We can fill physical memory with PTEs.



...



Virtual Address
Space



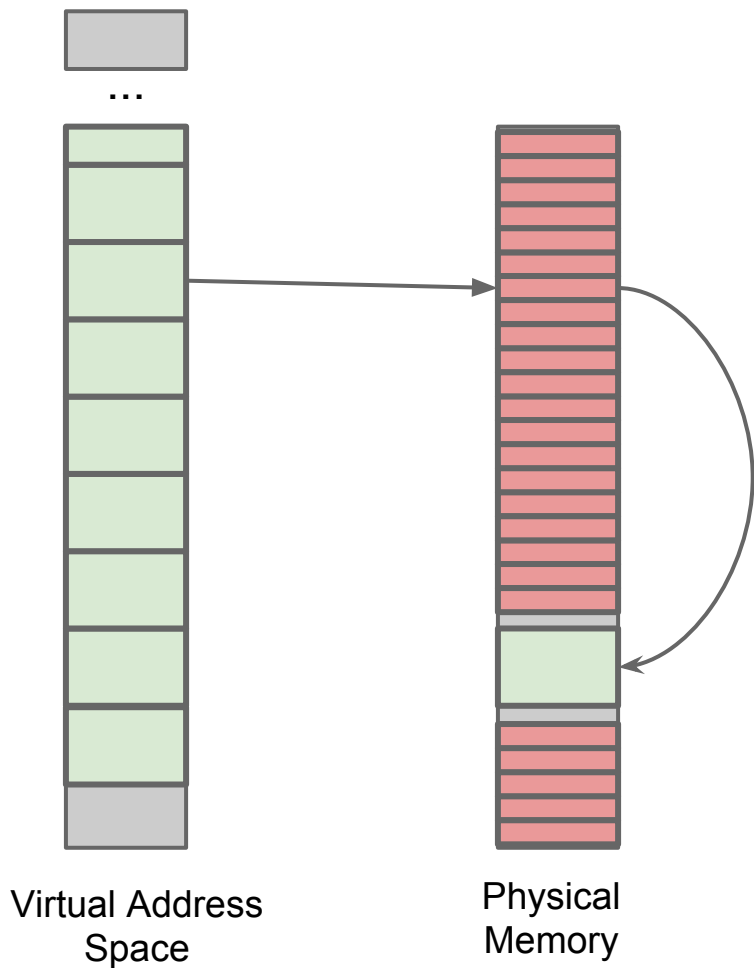
Physical
Memory

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.



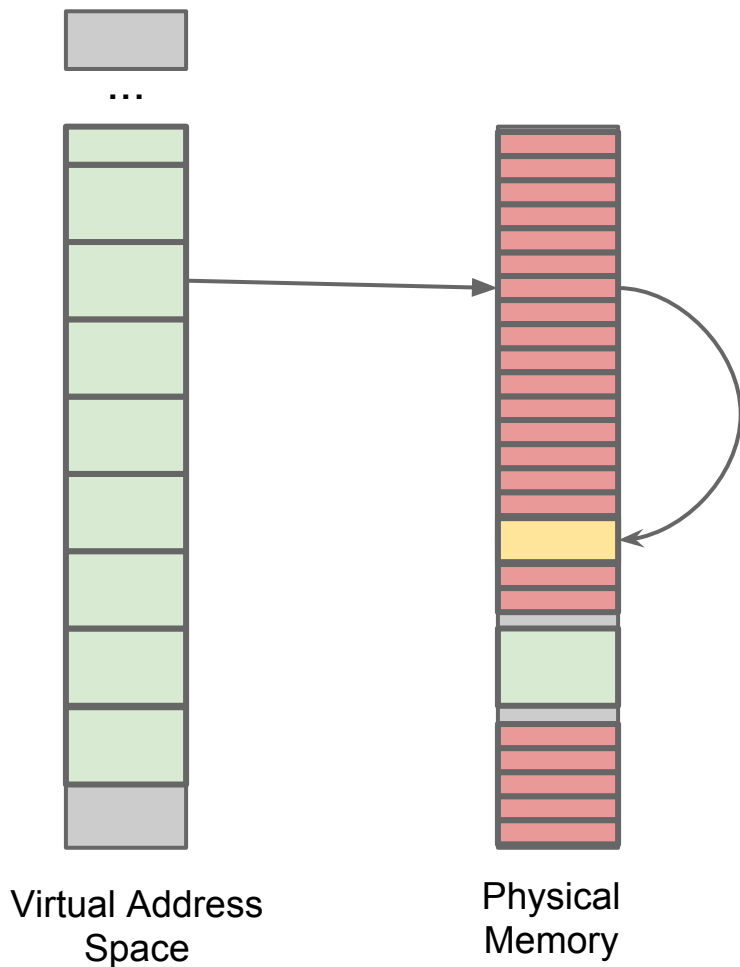
What happens when we repeatedly map a file with read-write permissions?

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?

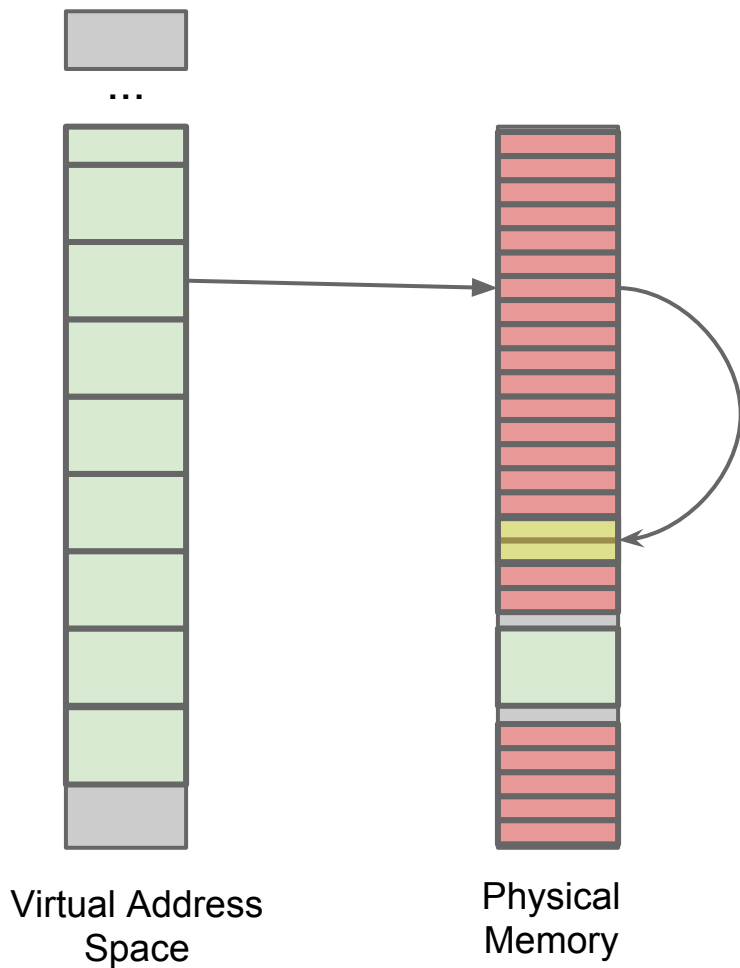
PTEs in physical memory help resolve virtual addresses to physical pages.

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.

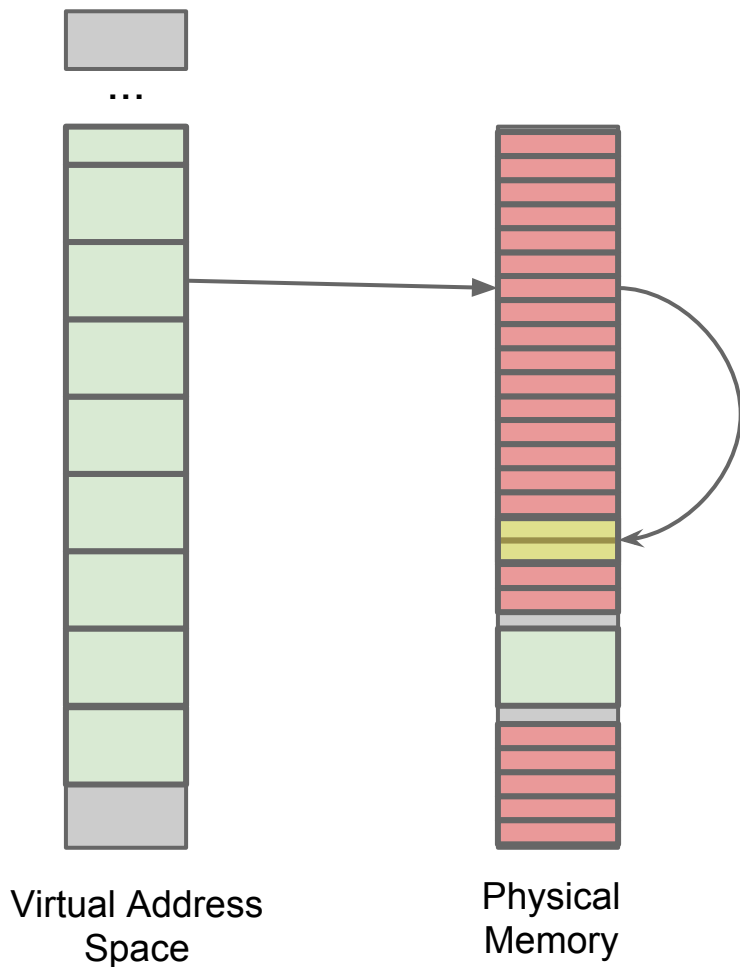
We can fill physical memory with PTEs.

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.



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.

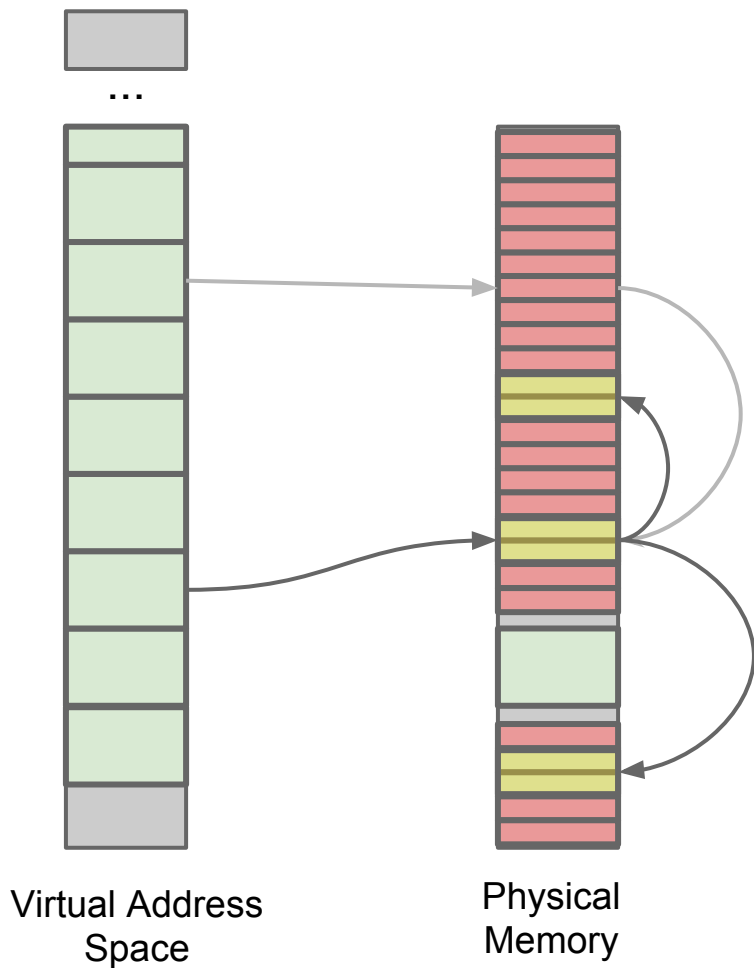
Each of them points to pages in the same physical file mapping.

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

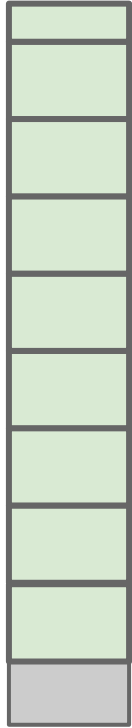
Privilege escalation in 7 easy steps ...

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.



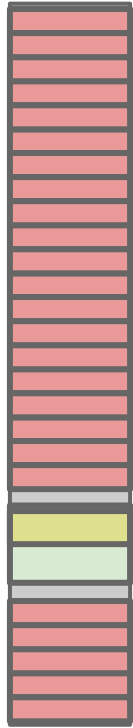
...



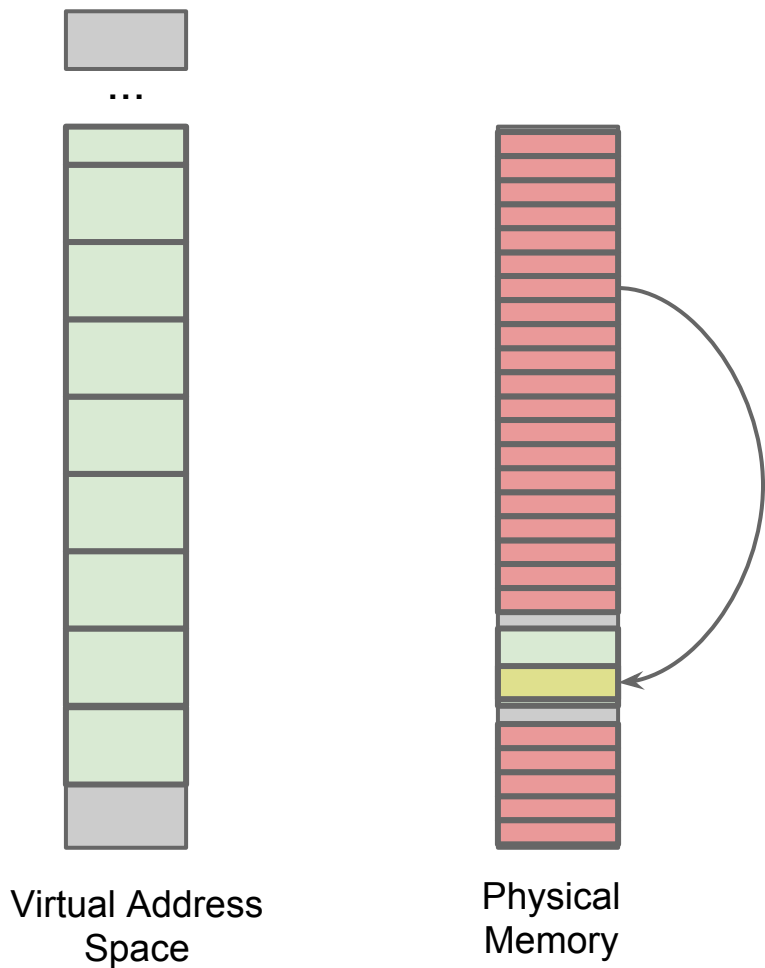
Virtual Address
Space

In practice there are many complications.

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



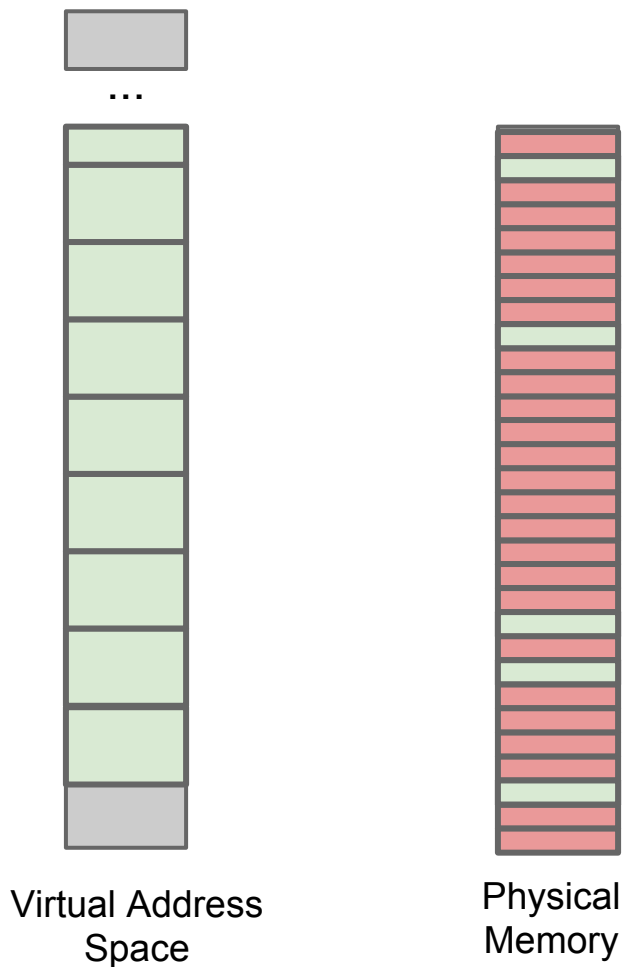
Physical
Memory



In practice there are many complications.

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

... 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.

100011101110001101001001000001001001010101.1001110110011 4 MB
10.1000101000000001001110011101011001010100.000111000111.010111 4 MB
00001100001100111011.10111001110100010001010101000010001100011100 4 MB
100100010010111110001000100001011000001011010000100110110101 4 MB
01000011000101110100010001000100011010100100110101010100000010 5 MB
01001010001001101100001100011001010111101000101100000011001110 5 MB
111101011000100110100001001000001000000111000010101010100010 5 MB
00010101010100000101010001010101000000010100110111011100010001 5 MB
11011001111010100101010111000101110101011010111100100010 6 MB
001100010001000011001000011100000.10101000000111000110011011100 6 MB
110101001111010100001000100011000010101000001100111000000000 6 MB
0.000101000100010000010.011111000100000010000011000100011000000111 6 MB
01001000000001010001100011011100000100101010101010101010100 7 MB
010001000000010000010.1110.0001111.010100010100010001000100000 7 MB
110000010111011100000010101011000100011010100100100010001000 7 MB
11000001010101010001000011001110011010101000001010001000.011101 7 MB
.....0000010001001011010101010.000011000000000101001000110010001 8 MB
11000001000100000001.1.0000000100.0111101010001111100100000000101 8 MB
1000100010001000101111.0111110011011001000100000010100010 8 MB
00111111.0000100010001110001000100111000100010001000101000 8 MB
010001111010000111.01110001.010000100000111101000000110110.111 9 MB
1010001100011001100110010001000100010010011101101010101000 9 MB
0001.111000101101010111101000110001001000001010100111.01011101 9 MB
0110.01000101000000010100010001011111010001000110110110111 9 MB
10101010001010111000000010101010001000101101101001001110001 10 MB
000010001011100100000011111010000011000011110000011000100001 10 MB
0010100001010000011100001110001110001000101011101010101. 10 MB
0110100101110110001111000010.000000100010010010110.010010110.000 10 MB
0110111000100101101010110000001110000011000000000000000000000000 11 MB
01001110101100001000010001010100010101000100010000000000000000 11 MB
11100010110001010010101000010010101011000000001011100101010000 11 MB
100001111000111000110000000011110001101000000101011.010110110 11 MB
.....0000000101010101.101010.00101.010101011.0101010101 12 MB
1010001 12 MB
1.010101010101.01010101.010101010101010101010101.01010101010101 12 MB
010101010101010101010101.0101010101010001.0101010101010001.01010 12 MB
..00000000000000.01 13 MB
.01010101010101.01010101010101.00101010101010101010101010101.010 13 MB
10.0010101.0101.0101.01010101010101.01010101010101010101010101 13 MB
01010101010101010101010101010101.0101.01010101.0101010101000101 13 MB
00011100000111000101101010111010000010001001010101000000010110 14 MB
111111.101100010100000101110110101001010000011100000101010100 14 MB
0001000111110101010000010101000110001010000010001111100000 14 MB
01010000100010001000100110011101010000010101111011000100001 14 MB
000010010000010101000010111000100100000111000101110001000100 15 MB
1001010011011100001011110001000100001000010000010111000000 15 MB
00010000010101010000010010.100011000101110000010000011101 15 MB
01000001010101000001 15 MB
0100000111000100000101010101000010001000100010001000100010001 15 MB

..... 28 MB
..... 28 MB
..... 28 MB
..... 29 MB
.....011011.101001010100001000000100110 29 MB
10111001001100110101010101010001100000100010001.1000100100000010 29 MB
1111.11000101001000000101.000001000001000010000100001010111.0011.010 29 MB
00011010.011000101010101001100010000100110100010001001111010.00 29 MB
01010001111101010100001000110010000010100010001010011000100110000. 30 MB
10110001110001000101111.001010100001001010111..... 30 MB
..... 30 MB
..... 30 MB
..... 31 MB
..... 31 MB
..... 31 MB
..... 31 MB
1111111011.111001110011100101111110111000010100010011111101 32 MB
1.1111011000110110110101111111101100111100011111010010101 32 MB
00111111100101111111100.00110110.0001001000111110111.1011111 32 MB
10111011111000101110010111000.011101010111010111010100011 32 MB
1000001111111110100011100010110111111011110.011101001101.011 33 MB
11110101111111011011010101011001101101110101010101010101010 33 MB
1.01110.1110111001101.0110100010010110011111000100011111110 33 MB
.01011010111011001011.1011110011100010001101101010111011.01 33 MB
000100011000000010001111101000110100000111010011010.010000000 34 MB
000000000000000.11111111.0111.1101.1.11.101.0110101011.1111 34 MB
111111.11111.11.01111.110111011111011111011111101111101111011.10 34 MB
11010111111101111111111101010101010101010101010101010101010 34 MB
101.01 35 MB
0101010101010101. 35 MB
..... 35 MB
..... 35 MB
000000111001000100011010000100000111000101000010010.00111011000 36 MB
001001000001101010000000.001101110001010100001011110101000111 36 MB
11001.001101011101000001001100000.10001011011100100001010010 36 MB
0010000010000011010000001000011000000111000000001001011000 36 MB
101111000100001001000101000000110110110001000000000010101 37 MB
1001010001000011001011.0110010010101101.0000010000010101.000 37 MB
001101101101100111110101000110101100001010010101100101110 37 MB
00000101000100000110010000010011000011000100101010111000 37 MB
001111011110000110001000010001111110101.1100.0101011100010.01 38 MB
00000010000010110.111010001.1000010000011100110000001001100100 38 MB
10001101.1000101100110.1101010000101110100110001000100010001 38 MB
0010000000.0011011000100010000111010100000.01101000110001000 38 MB
110111010000001000110100010001000100111110111101000100000000 39 MB
100000100000101010101000100010001000100010001000100010001000 39 MB
1001011001000001000101010100010001100001111010101000101000 39 MB
01100111010100 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

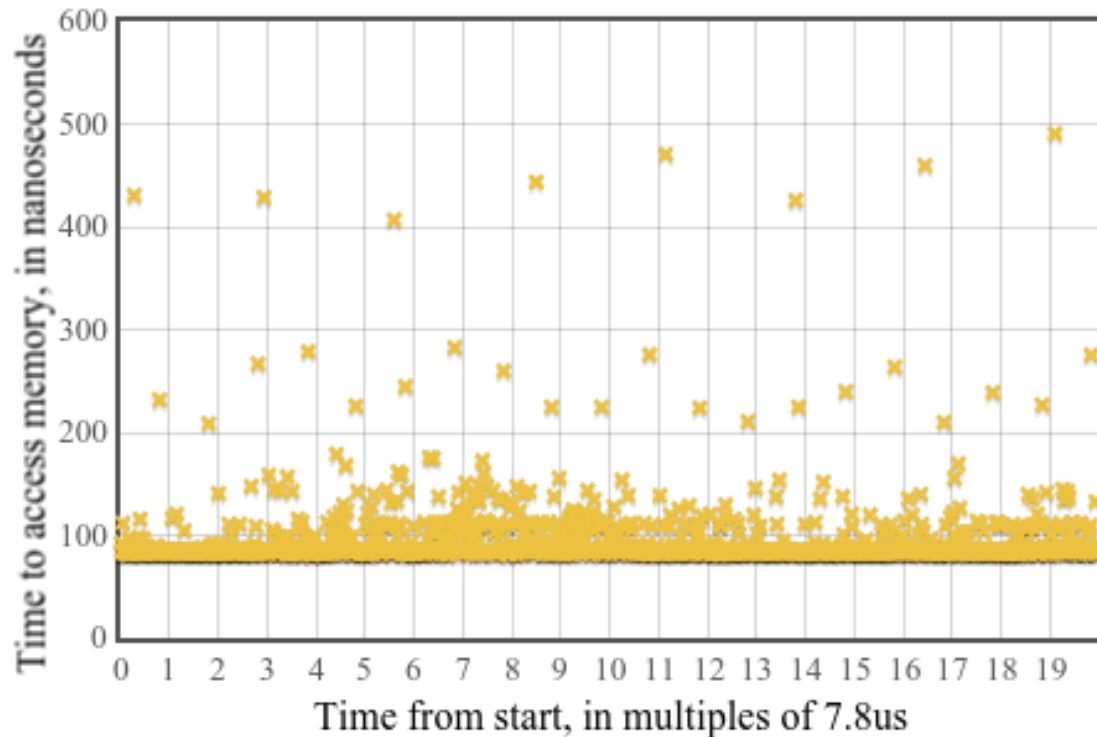
- Count activations of a row
- Refresh neighbouring rows when counter reaches threshold
- Covered by LPDDR4
- DDR4 too?
- In DRAM: Micron data sheets
- In memory controllers:
 - pTRR (pseudo TRR)
 - One Intel presentation says Ivy Bridge supports pTRR. No further evidence of this?

Mitigation: 2x refresh rate

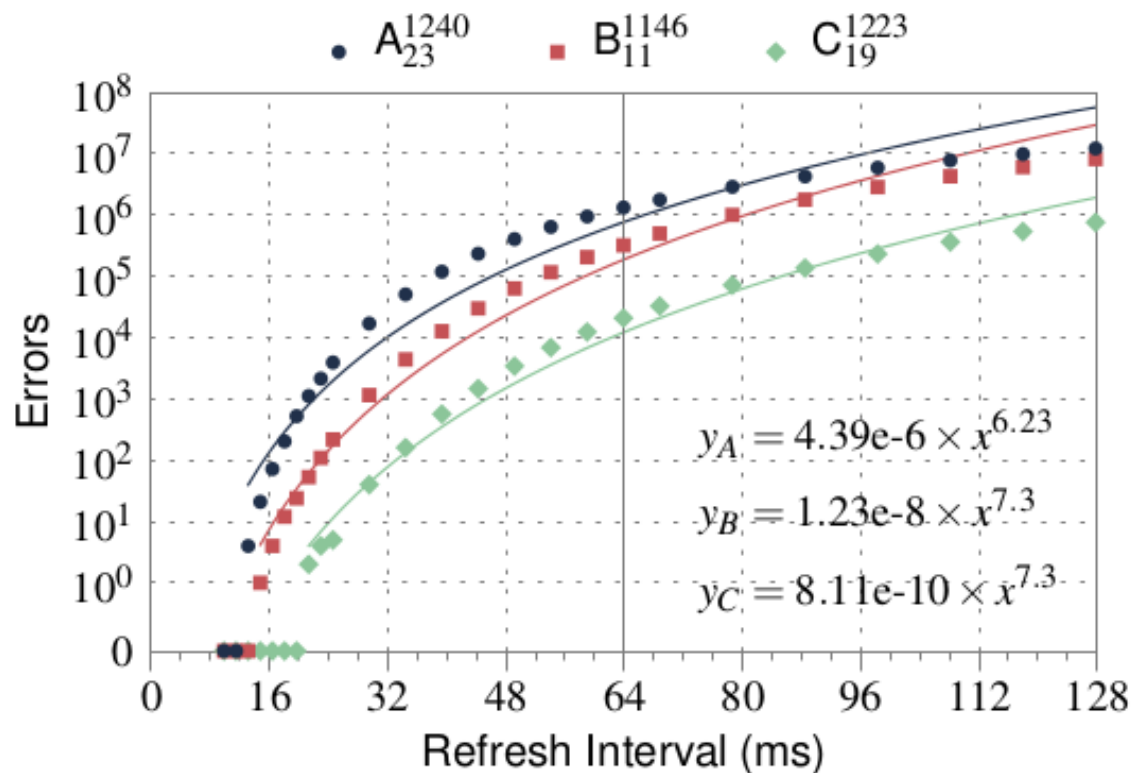
- Current CPUs support this
- tREFI parameter
 - Covered by Intel's public memory controller docs
 - Set by BIOS
 - Coreboot covers Sandy/Ivy Bridge
- Various vendor BIOS updates do this
- How to verify refresh rate?
- Is 2x refresh enough?

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/>