

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK

PROJECT ZERO

News and updates from the Project Zero team at Google

Monday, March 9, 2015

Exploiting the DRAM rowhammer bug to gain kernel privileges

Posted by Mark Seaborn, sandbox builder and breaker, with contributions by Thomas Dullien, reverse engineer

[This guest post continues Project Zero's practice of promoting excellence in security research on the Project Zero blog]

Overview

"Rowhammer" is a problem with some recent DRAM devices in which repeatedly accessing a row of memory can cause bit flips in adjacent rows. We tested a selection of laptops and found that a subset of them exhibited the problem. We built two working privilege escalation exploits that use this effect. One exploit uses rowhammer-induced bit flips to gain kernel privileges on x86-64 Linux when run as an unprivileged userland process. When run on a machine vulnerable to the rowhammer problem, the process was able to induce bit flips in page table entries (PTEs). It was able to use this to gain write access to its own page table, and hence gain read-write access to all of physical memory.

We don't know for sure how many machines are vulnerable to this attack, or how many existing vulnerable machines are fixable. Our exploit uses the x86 CLFLUSH instruction to generate many accesses to the underlying DRAM, but other techniques might work on non-x86 systems too.

We expect our PTE-based exploit could be made to work on other operating systems; it is not inherently Linux-specific. Causing bit flips in PTEs is just one avenue of exploitation; other avenues for exploiting bit flips can be practical too. Our other exploit demonstrates this by escaping from the Native Client sandbox.

Introduction to the rowhammer problem

We learned about the rowhammer problem from Yoongu Kim et al's paper, "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).

They demonstrate that, by repeatedly accessing two "aggressor" memory locations within the process's virtual address space, they can cause bit flips in a third, "victim" location. The victim location is potentially outside the virtual address space of the process — it is in a different DRAM row from the aggressor locations, and hence in a different 4k page (since rows are larger than 4k in modern systems).

This works because DRAM cells have been getting smaller and closer together. As DRAM manufacturing scales down chip features to smaller physical dimensions, to fit more memory capacity onto a chip, it has become harder to prevent DRAM cells from interacting electrically with each other. As a result, accessing one location in memory can disturb neighbouring locations, causing charge to leak into or out of neighbouring cells. With enough accesses, this can change a cell's value from 1 to 0 or vice versa.

The paper explains that this tiny snippet of code can cause bit flips:

```
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
    jmp code1a
```

Two ingredients are required for this routine to cause bit flips:

- **Address selection:** For code1a to cause bit flips, addresses X and Y must map to different rows of DRAM in the same bank.

Some background: Each DRAM chip contains many rows of cells. Accessing a byte in memory involves transferring data from the row into the chip's "row buffer" (discharging the row's cells in the process), reading or writing the row buffer's contents, and then copying the row buffer's contents back to the original row's cells (recharging the cells).

It is this process of "activating" a row (discharging and recharging it) that can disturb adjacent rows. If this is done enough times, in between automatic refreshes of the adjacent rows (which usually occur every 64ms), this can cause bit flips in the adjacent rows.

The row buffer acts as a cache, so if addresses X and Y point to the same row, then code1a will just read from the row buffer without activating the row repeatedly.

Search This Blog

Pages

- [Working at Project Zero](#)

Archives

2019

- [Windows Exploitation Tricks: Abusing the User-Mode...](#) (Apr)
- [Virtually Unlimited Memory: Escaping the Chrome Sa...](#) (Apr)
- [Splitting atoms in XNU](#) (Apr)
- [Windows Kernel Logic Bug Class: Access Mode Mismat...](#) (Mar)
- [Android Messaging: A Few Bugs Short of a Chain](#) (Mar)
- [The Curious Case of Convexity Confusion](#) (Feb)
- [Examining Pointer Authentication on the iPhone XS](#) (Feb)
- [voucher_swap: Exploiting MIG reference counting in...](#) (Jan)
- [Taking a page from the kernel's book: A TLB issue ...](#) (Jan)

2018

- [On VBScript](#) (Dec)
- [Searching statically-linked vulnerable library fun...](#) (Dec)
- [Adventures in Video Conferencing Part 5: Where Do ...](#) (Dec)
- [Adventures in Video Conferencing Part 4: What Didn...](#) (Dec)
- [Adventures in Video Conferencing Part 3: The Even ...](#) (Dec)
- [Adventures in Video Conferencing Part 2: Fun with ...](#) (Dec)
- [Adventures in Video Conferencing Part 1: The Wild ...](#) (Dec)
- [Injecting Code into Windows Protected Processes us...](#) (Nov)
- [Heap Feng Shader: Exploiting SwiftShader in Chrome...](#) (Oct)
- [Deja-XNU](#) (Oct)
- [Injecting Code into Windows Protected Processes us...](#) (Oct)
- [365 Days Later: Finding and Exploiting Safari Bugs...](#) (Oct)
- [A cache invalidation bug in Linux memory managemen...](#) (Sep)
- [OATmeal on the Universal Cereal Bus: Exploiting An...](#) (Sep)
- [The Problems and Promise of WebAssembly](#) (Aug)
- [Windows Exploitation Tricks: Exploiting Arbitrary ...](#) (Aug)
- [Adventures in vulnerability reporting](#) (Aug)
- [Drawing Outside the Box: Precision Issues in Graph...](#) (Jul)

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK

repeatedly activated. This is termed “row hammering”.

- **Bypassing the cache:** Without code1a's CLFLUSH instructions, the memory reads (MOV's) will be served from the CPU's cache. Flushing the cache using CLFLUSH forces the memory accesses to be sent to the underlying DRAM, which is necessary to cause the rows to be repeatedly activated.

Note that the paper's version of code1a also includes an MFENCE instruction. However, we found that using MFENCE was unnecessary and actually reduced the number of bit flips we saw. Yoongu Kim's modified memtest also omits the MFENCE from its row hammering code.

Refining the selection of addresses to hammer

Using the physical address mapping

How can we pick pairs of addresses that satisfy the “different row, same bank” requirements?

One possibility is to use knowledge of how the CPU's memory controller maps physical addresses to DRAM's row, column and bank numbers, along with knowledge of either:

- The absolute physical addresses of memory we have access to. Linux allows this via `/proc/PID/pagemap`.
- The relative physical addresses of memory we have access to. Linux can allow this via its support for “huge pages”, which cover 2MB of contiguous physical address space per page. Whereas a normal 4k page is smaller than a typical DRAM row, a 2MB page will typically cover multiple rows, some of which will be in the same bank.

Yoongu Kim et al take this approach. They pick $Y = X + 8\text{MByte}$ based on knowledge of the physical address mapping used by the memory controllers in Intel and AMD's CPUs.

Random address selection

The CPU's physical address mapping can be difficult to determine, though, and features such as `/proc/PID/pagemap` and huge pages are not available everywhere. Furthermore, if our guesses about the address mapping are wrong, we might pick an offset that pessimises our chances of successful row hammering. (For example, $Y = X + 8\text{kByte}$ might always give addresses in different banks.)

A simpler approach is to pick address pairs at random. We allocate a large block of memory (e.g. 1GB) and then pick random virtual addresses within that block. On a machine with 16 DRAM banks (as one of our test machines has: 2 DIMMs with 8 banks per DIMM), this gives us a 1/16 chance that the chosen addresses are in the same bank, which is quite high. (The chance of picking two addresses in the same row is negligible.)

Furthermore, we can increase our chances of successful row hammering by modifying code1a to hammer more addresses per loop iteration. We find we can hammer 4 or 8 addresses without slowing down the time per iteration.

Selecting addresses using timing

Another way to determine whether a pair of addresses has the “different row, same bank” property would be to time uncached accesses to those addresses using a fine-grained timer such as the RDTSC instruction. The access time will be slower for pairs that satisfy this property than those that don't.

Double-sided hammering

We have found that we can increase the chances of getting bit flips in row N by row-hammering both of its neighbours (rows N-1 and N+1), rather than by hammering one neighbour and a more-distant row. We dub this “double-sided hammering”.

For many machines, double-sided hammering is the only way of producing bit flips in reasonable time. For machines where random selection is already sufficient to cause bit flips, double-sided hammering can lead to a vastly increased number of bits flipped. We have observed 25+ bits flipped in one row on one particularly fragile machine.

Performing double-sided hammering is made more complicated by the underlying memory geometry. It requires the attacker to know or guess what the offset will be, in physical address space, between two rows that are in the same bank *and* are adjacent. Let's call this the “row offset”.

From our testing, we were able to naively extrapolate that the row offset for laptop Model #4 (see the table below) is 256k. We did this by observing the likelihood of bit flips relative to the distance the selected physical memory pages had from the victim page. This likelihood was maximized when we hammered the locations 256k below and above a given target row.

This “256k target memory area, 256k victim memory area, 256k target memory area” setup has shown itself to be quite effective on other laptops by the same vendor. It is likely that this setup needs to be tweaked for other vendors.

This 256k row offset could probably be explained as being a product of the row size (number of columns), number of banks, number of channels, etc., of the DRAM in this machine, though this requires further knowledge of how the hardware maps physical addresses to row and bank numbers.

Doing double-sided hammering does require that we can pick physically-contiguous pages (e.g. via `/proc/PID/pagemap` or huge pages).

Arbitrary ... (Apr)

- Reading privileged memory with a side-channel (Jan)

2017

- aPAColypse now: Exploiting Windows 10 in a Local N... (Dec)
- Over The Air - Vol. 2, Pt. 3: Exploiting The Wi-Fi... (Oct)
- Using Binary Diffing to Discover Windows Kernel Me... (Oct)
- Over The Air - Vol. 2, Pt. 2: Exploiting The Wi-Fi... (Oct)
- Over The Air - Vol. 2, Pt. 1: Exploiting The Wi-Fi... (Sep)
- The Great DOM Fuzz-off of 2017 (Sep)
- Bypassing VirtualBox Process Hardening on Windows (Aug)
- Windows Exploitation Tricks: Arbitrary Directory C... (Aug)
- Trust Issues: Exploiting TrustZone TEEs (Jul)
- Exploiting the Linux kernel via packet sockets (May)
- Exploiting .NET Managed DCOM (Apr)
- Exception-oriented exploitation on iOS (Apr)
- Over The Air: Exploiting Broadcom's Wi-Fi Stack (P... (Apr)
- Notes on Windows Uniscribe Fuzzing (Apr)
- Pandavirtualization: Exploiting the Xen hypervisor... (Apr)
- Over The Air: Exploiting Broadcom's Wi-Fi Stack (P... (Apr)
- Project Zero Prize Conclusion (Mar)
- Attacking the Windows NVIDIA Driver (Feb)
- Lifting the (Hyper) Visor: Bypassing Samsung's Rea... (Feb)

2016

- Chrome OS exploit: one byte overflow and symlinks (Dec)
- BitUnmap: Attacking Android Ashmem (Dec)
- Breaking the Chain (Nov)
- task_t considered harmful (Oct)
- Announcing the Project Zero Prize (Sep)
- Return to libstagefright: exploiting libutils on A... (Sep)
- A Shadow of our Former Self (Aug)
- A year of Windows kernel font fuzzing #2: the tech... (Jul)
- How to Compromise the Enterprise Endpoint (Jun)
- A year of Windows kernel font fuzzing #1: the resu... (Jun)
- Exploiting Recursion in the Linux Kernel (Jun)
- Life After the Isolated Heap (Mar)
- Race you to the kernel! (Mar)
- Exploiting a Leaked Thread Handle (Mar)
- The Definitive Guide on Win32 to NT Path Conversio... (Feb)
- Racing MIDI messages in Chrome (Feb)
- Raising the Dead (Jan)

2015

- FireEye Exploitation: Project Zero's Vulnerability... (Dec)

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK

this task!

We found various machines that exhibit bit flips (see the experimental results below). Having done that, we wrote two exploits:

- [The first](#) runs as a [Native Client](#) (NaCl) program and escalates privilege to escape from NaCl's x86-64 sandbox, acquiring the ability to call the host OS's syscalls directly. We have mitigated this by changing NaCl to disallow the CLFLUSH instruction. (I picked NaCl as the first exploit target because I work on NaCl and have written proof-of-concept NaCl sandbox escapes before.)
- [The second](#) runs as a normal x86-64 process on Linux and escalates privilege to gain access to all of physical memory. This is harder to mitigate on existing machines.

NaCl sandbox escape

Native Client is a sandboxing system that allows running a subset of x86-64 machine code (among other architectures) inside a sandbox. Before running an x86-64 executable, NaCl uses a validator to check that its code conforms to [a subset of x86 instructions that NaCl deems to be safe](#).

However, NaCl assumes that the hardware behaves correctly. It assumes that memory locations don't change without being written to! NaCl's approach of validating machine code is particularly vulnerable to bit flips, because:

- A bit flip in validated code can turn a safe instruction sequence into an unsafe one.
- Under NaCl, the sandboxed program's code segment is readable by the program. This means the program can check whether a bit flip has occurred and determine whether or how it can exploit the change.

Our exploit targets NaCl's instruction sequence for sandboxed indirect jumps, which looks like this:

```
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.
```

The exploit works by triggering bit flips in that code sequence. It knows how to exploit 13% of the possible bit flips. Currently it only handles bit flips that modify register numbers. (With more work, it could handle more exploitable cases, such as opcode changes.) For example, if a bit flip occurs in bit 0 of the register number in “`jmp *%rax`”, this morphs to “`jmp *%rcx`”, which is easily exploitable — since `%rcx` is unconstrained, this allows jumping to any address. Normally NaCl only allows indirect jumps to 32-byte-aligned addresses (and it ensures that instructions do not cross 32-byte bundle boundaries). Once a program can jump to an unaligned address, it can escape the sandbox, because it is possible to hide unsafe x86 instructions inside safe ones. For example:

```
20ea0:      48 b8 0f 05 eb 0c f4 f4 f4 f4    movabs $0xf4f4f4f4f40ceb05f,%rax
```

This hides a SYSCALL instruction (`0f 05`) at address `0x20ea2`.

Our NaCl exploit does the following:

- It fills the sandbox's dynamic code area with 250MB of NaClized indirect jump instruction sequences using NaCl's `dyncode_create()` API.
- In a loop:
 - It row-hammers the dynamic code area using CLFLUSH, picking random pairs of addresses.
 - It searches the dynamic code area for bit flips. If it sees an exploitable bit flip, it uses it to jump to shell code hidden inside NaCl-validated instructions. Otherwise, if the bit flip isn't exploitable, it continues.

We have mitigated this by changing NaCl's x86 validator to disallow the CLFLUSH instruction (tracked by CVE-2015-0565). However, there might be other ways to cause row hammering besides CLFLUSH (see below).

Prior to disallowing CLFLUSH in NaCl, it may have been possible to chain this NaCl exploit together with the kernel privilege escalation below so that a NaCl app in the Chrome Web Store app could gain kernel privileges, using just one underlying hardware bug for the whole chain. To our knowledge there was no such app in the Chrome Web Store. PNaCl — which is available on the open web — has an extra layer of protection because an attacker would have had to find an exploit in the PNaCl translator before being able to emit a CLFLUSH instruction.

Kernel privilege escalation

Our kernel privilege escalation works by using row hammering to induce a bit flip in a page table entry (PTE) that causes the PTE to point to a physical page containing a page table of the attacking process. This gives the attacking process read-write access to one of its own page tables, and hence to all of physical memory.

There are two things that help ensure that the bit flip has a high probability of being exploitable:

1. Rowhammer-induced bit flips tend to be repeatable. This means we can tell in advance if a DRAM cell tends to flip and whether this bit location will be useful for the exploit.

For example, bit 51 in a 64-bit word is the top bit of the physical page number in a PTE on x86-64. If this changes from 0 to 1, that will produce a page number that's bigger than the system's physical memory, which isn't useful for our exploit, so we can skip trying to use this bit flip. However, bit 12 is the bottom bit of the PTE's physical page number. If that changes from 0 to 1 or from 1 to 0, the PTE will still point to a valid physical page.

Other interesting links

- [Windows Drivers are True'y Tricky](#) (Oct)
- [Revisiting Apple IPC: \(1\) Distributed Objects](#) (Sep)
- [Kaspersky: Mo Unpackers, Mo Problems.](#) (Sep)
- [Stagefrightened?](#) (Sep)
- [Enabling QR codes in Internet Explorer, or a story...](#) (Sep)
- [Windows 10^H^H Symbolic Link Mitigations](#) (Aug)
- [One font vulnerability to rule them all #4: Window...](#) (Aug)
- [Three bypasses and a fix for one of Flash's Vector...](#) (Aug)
- [Attacking ECMAScript Engines with Redefinition](#) (Aug)
- [One font vulnerability to rule them all #3: Window...](#) (Aug)
- [One font vulnerability to rule them all #2: Adobe ...](#) (Aug)
- [One font vulnerability to rule them all #1: Introd...](#) (Jul)
- [One Perfect Bug: Exploiting Type Confusion in Flas...](#) (Jul)
- [Significant Flash exploit mitigations are live in ...](#) (Jul)
- [From inter to intra: gaining reliability](#) (Jul)
- [When 'int' is the new 'short'](#) (Jul)
- [What is a "good" memory corruption vulnerability?](#) (Jun)
- [Analysis and Exploitation of an ESET Vulnerability...](#) (Jun)
- [Owning Internet Printing - A Case Study in Modern ...](#) (Jun)
- [Dude, where's my heap?](#) (Jun)
- [In-Console-Able](#) (May)
- [A Tale of Two Exploits \(Apr\)](#)
- [Taming the wild copy: Parallel Thread Corruption](#) (Mar)
- [Exploiting the DRAM rowhammer bug to gain kernel p...](#) (Mar)
- [Feedback and data-driven updates to Google's discl...](#) (Feb)
- [\(^Exploiting\)s*\(CVE-2015-0318\)s*\(in\)s*\(Flash\\$\)](#) (Feb)
- [A Token's Tale](#) (Feb)
- [Exploiting NVMAP to escape the Chrome sandbox - CV...](#) (Jan)
- [Finding and exploiting ntpd vulnerabilities](#) (Jan)

2014

- [Internet Explorer EPM Sandbox Escape CVE-2014-6350...](#) (Dec)
- [pwn4fun Spring 2014 - Safari - Part II](#) (Nov)
- [Project Zero Patch Tuesday roundup, November 2014](#) (Nov)
- [Did the "Man With No Name" Feel Insecure?](#) (Oct)
- [More Mac OS X and iPhone sandbox escapes and kerne...](#) (Oct)
- [Exploiting CVE-2014-0556 in Flash](#) (Sep)
- [The poisoned NUL byte, 2014 edition](#) (Aug)
- [What does a pointer look like, anyway?](#) (Aug)
- [Mac OS X and iPhone sandbox escapes](#) (Jul)
- [pwn4fun Spring 2014 - Safari - Part I](#) (Jul)
- [Announcing Project Zero](#) (Jul)

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK

There are two caveats:

- Our exploit runs in a normal Linux process. More work may be required for this to work inside a sandboxed Linux process (such as a Chromium renderer process).
- We tested on a machine with low memory pressure. Making this work on a heavily-loaded machine may involve further work.

Break it down: exploit steps

The first step is to search for aggressor/victim addresses that produce useful bit flips:

- `mmap()` a large block of memory.
- Search this block for aggressor/victim addresses by row-hammering random address pairs. Alternatively, we can use aggressor/victim physical addresses that were discovered and recorded on a previous run; we use `/proc/self/pagemap` to search for these in memory.
- If we find aggressor/victim addresses where the bit flipped within the 64-bit word isn't useful for the exploit, just skip that address set.
- Otherwise, `munmap()` all but the aggressor and victim pages and begin the exploit attempt.

In preparation for spraying page tables, we create a file in `/dev/shm` (a shared memory segment) that we will `mmap()` repeatedly. (See later for how we determine its size.) We write a marker value at the start of each 4k page in the file so that we can easily identify these pages later, when checking for PTE changes.

Note that we don't want these data pages to be allocated from sequential physical addresses, because then flips in the lower bits of physical page numbers would tend to be unexploitable: A PTE pointing to one data page would likely change to pointing to another data page.

To avoid that problem, we first deliberately fragment physical memory so that the kernel's allocations from physical memory are randomised:

- `mmap()` (with `MAP_POPULATE`) a block of memory that's a large fraction of the machine's physical memory size.
- Later, whenever we do something that will cause the kernel to allocate a 4k page (such as a page table), we release a page from this block using `madvise() + MADV_DONTNEED`.

We are now ready to spray memory with page tables. To do this, we `mmap()` the data file repeatedly:

- We want each mapping to be at a 2MB-aligned virtual address, since each 4k page table covers a 2MB region of virtual address space. We use `MAP_FIXED` for this.
- We cause the kernel to populate some of the PTEs by accessing their corresponding pages. We only need to populate one PTE per page table: We know our bit flip hits the Nth PTE in a page table, so, for speed, we only fault in the Nth 4k page in each 2MB chunk.
- Linux imposes a limit of about 2^{16} on the number of VMAs (`mmap()`'d regions) a process can have. This means that our `/dev/shm` data file must be large enough such that, when mapped 2^{16} times, the mappings create enough page tables to fill most of physical memory. At the same time, we want to keep the data file as small as possible so as not to waste memory that could instead be filled with page tables. We pick its size accordingly.
- In the middle of this, we `munmap()` the victim page. With a high probability, the kernel will reuse this physical page as a page table. We can't touch this page directly any more, but we can potentially modify it via row hammering.

Having finished spraying, it's hammer time. We hammer the aggressor addresses. Hopefully this induces the bit flip in the victim page. We can't observe the bit flip directly (unlike in the NaCl exploit).

Now we can check whether PTEs changed exploitably. We scan the large region we mapped to see whether any of the PTEs now point to pages other than our data file. Again, for speed, we only need to check the Nth page within each 2MB chunk. We can check for the marker value we wrote earlier. If we find no marker mismatches, our attempt failed (and we could retry).

If we find a marker mismatch, then we have gained illicit access to a physical page. Hopefully this is one of the page tables for our address space. If we want to be careful, we can verify whether this page looks like one of our page tables. The Nth 64-bit field should look like a PTE (certain bits will be set or unset) and the rest should be zero. If not, our attempt failed (and we could retry).

At this point, we have write access to a page table, probably our own. However, we don't yet know which virtual address this is the page table for. We can determine that as follows:

- Write a PTE to the page (e.g. pointing to physical page 0).
- Do a second scan of address space to find a second virtual page that now points to somewhere other than our data file. If we don't find it, our attempt failed (and we could retry).

Exploiting write access to page tables

We now have write access to one of our process's page tables. By modifying the page table, we can get access to any page in physical memory. We now have many options for how to exploit that, varying in portability, convenience and speed. The portable options work without requiring knowledge of kernel data structures. Faster options work in $O(1)$ time, whereas slower options might require scanning all of physical memory to locate a data structure.

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK

`mmap()` + `MAP_POPULATE`) and query which physical pages it was loaded into using `/proc/self/pagemap`.

- A similar approach is to modify a library that a SUID executable uses, such as `/lib64/ld-linux-x86-64.so.2`. (On some systems, SUID executables such as `/bin/ping` can't be `open()`'d because their permissions have been locked down.)
- Other, less portable approaches are to modify kernel code or kernel data structures.
 - We could modify our process's UID field. This would require locating the "struct cred" for the current process and knowing its layout.
 - We could modify the kernel's syscall handling code. We can quickly determine its physical address using the `SIDT` instruction, which is exposed to unprivileged code.

Routes for causing row hammering

Our proof-of-concept exploits use the x86 `CLFLUSH` instruction, because it's the easiest way to force memory accesses to be sent to the underlying DRAM and thus cause row hammering.

The fact that `CLFLUSH` is usable from unprivileged code is surprising, because the number of legitimate uses for it outside of a kernel or device driver is probably very small. For comparison, ARM doesn't have an unprivileged cache-flush instruction. (ARM Linux does have a `cacheflush()` syscall, used by JITs, for synchronising instruction and data caches. On x86, the i-cache and d-cache are synchronised automatically, so `CLFLUSH` isn't needed for this purpose.)

We have changed NaCl's x86 validator to disallow `CLFLUSH`. Unfortunately, kernels can't disable `CLFLUSH` for normal userland code. Currently, `CLFLUSH` can't be intercepted or disabled, even using VMX (x86 virtualisation). (For example, `RDTSC` can be intercepted without VMX support. VMX allows intercepting more instructions, including `WBINVD` and `CPUID`, but not `CLFLUSH`.) There might be a case for changing the x86 architecture to allow `CLFLUSH` to be intercepted. From a security engineering point of view, removing unnecessary attack surface is good practice.

However, there might be ways of causing row hammering without `CLFLUSH`, which might work on non-x86 architectures too:

- **Normal memory accesses:** Is it possible that normal memory accesses, in sufficient quantity or in the right pattern, can trigger enough cache misses to cause rowhammer-induced bit flips? This would require generating cache misses at every cache level (L1, L2, L3, etc.). Whether this is feasible could depend on the associativity of these caches.

If this is possible, it would be a serious problem, because it might be possible to generate bit flips from JavaScript code on the open web, perhaps via JavaScript typed arrays.

- **Non-temporal memory accesses:** On x86, these include non-temporal stores (`MOVNTI`, `MOVNTQ`, `MOVNTDQ(A)`, `MOVNTPD`, `MOVNTSD` and `MOVNTSS`) and non-temporals reads (via prefetches — `PREFETCHNTA`).
- **Atomic memory accesses:** Some reports claim that non-malicious use of spinlocks can cause row hammering, although the reports have insufficient detail and we've not been able to verify this. (See ["The Known Failure Mechanism in DDR3 memory called 'Row Hammer'"](#), Barbara Aichinger.) This seems unlikely on a multi-core system where cores share the highest-level cache. However, it might be possible on multi-socket systems where some pairs of cores don't share any cache.
- **Misaligned atomic memory accesses:** x86 CPUs guarantee that instructions with a `LOCK` prefix access memory atomically, even if the address being accessed is misaligned, and even if it crosses a cache line boundary. (See section 8.1.2.2, "Software Controlled Bus Locking", in [Intel's architecture reference](#), which says "The integrity of a bus lock is not affected by the alignment of the memory field".) This is done for backwards compatibility. In this case, the CPU doesn't use modern cache coherency protocols for atomicity. Instead, the CPU falls back to the older mechanism of locking the bus, and we believe it might use uncached memory accesses. (On some multi-CPU-socket NUMA machines, this locking is [implemented via the QPI protocol](#) rather than via a physical `#LOCK` pin.)

If misaligned atomic ops generate uncached DRAM accesses, they might be usable for row hammering.

Initial investigation suggests that these atomic ops do bypass the cache, but that they are too slow for this to generate enough memory accesses, within a 64ms refresh period, to generate bit flips.

- **Uncached pages:** For example, Windows' `CreateFileMapping()` API has a `SEC_NOCACHE` flag for requesting a non-cacheable page mapping.
- **Other OS interfaces:** There might be cases in which kernels or device drivers, such as GPU drivers, do uncached memory accesses on behalf of userland code.

Experimental results

We tested a selection of x86 laptops that were readily available to us (all with non-ECC memory) using `CLFLUSH` with the "random address selection" approach above. We found that a large subset of these machines exhibited rowhammer-induced bit flips. The results are shown in the table below.

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK

- Our sample size was not large enough that it can be considered representative.
- A negative result (an absence of bit flips) on a given machine does not definitively mean that it is not possible for rowhammer to cause bit flips on that machine. We have not performed enough testing to determine that a given machine is not vulnerable.

As a result, we have decided to anonymize our results below.

All of the machines tested used DDR3 DRAM. It was not possible to identify the age of the DRAM in all cases.

	Laptop model	Laptop year	CPU family (microarchitecture)	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
6	Model #4	2012	Family W	DRAM vendor A	yes
7	Model #5	2012	Family X	DRAM vendor C	no
8	Model #5	2012	Family X	DRAM vendor C	no
9	Model #5	2013	Family X	DRAM vendor B	yes
10	Model #5	2013	Family X	DRAM vendor B	yes
11	Model #5	2013	Family X	DRAM vendor B	yes
12	Model #5	2013	Family X	DRAM vendor B	yes
13	Model #5	2013	Family X	DRAM vendor B	yes
14	Model #5	2013	Family X	DRAM vendor B	yes
15	Model #5	2013	Family X	DRAM vendor B	yes
16	Model #5	2013	Family X	DRAM vendor B	yes
17	Model #5	2013	Family X	DRAM vendor C	no
18	Model #5	2013	Family X	DRAM vendor C	no
19	Model #5	2013	Family X	DRAM vendor C	no
20	Model #5	2013	Family X	DRAM vendor C	no
21	Model #5	2013	Family X	DRAM vendor C	yes
22	Model #5	2013	Family X	DRAM vendor C	yes
23	Model #6	2013	Family Y	DRAM vendor A	no
24	Model #6	2013	Family Y	DRAM vendor B	no
25	Model #6	2013	Family Y	DRAM vendor B	no
26	Model #6	2013	Family Y	DRAM vendor B	no
27	Model #6	2013	Family Y	DRAM vendor B	no
28	Model #7	2012	Family W	DRAM vendor D	no
29	Model #8	2014	Family Z	DRAM vendor A	no

We also tested some desktop machines, but did not see any bit flips on those. That could be because they were all relatively high-end machines with ECC memory. The ECC could be hiding bit flips.

Testing your own machine

Users may wish to test their own machines using the rowhammer - test tool above. If a machine produces bit flips during testing, users may wish to adjust security and trust decisions regarding the machine accordingly.

While an absence of bit flips during testing on a given machine does not automatically imply safety, it does provide some baseline assurance that causing bit flips is at least difficult on that machine.

Mitigations

Targeted refreshes of adjacent rows

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK

refer to proposals “to maintain an array of counters” (either in the memory controller or in DRAM) for counting activations. The paper proposes an alternative, probabilistic scheme called “PARA”, which is stateless and thus does not require maintaining counters.

There are signs that some newer hardware implements mitigations:

- JEDEC’s recently-published LPDDR4 standard for DRAM (where “LP” = “Low Power”) specifies two rowhammer mitigation features that a memory controller would be expected to use. (See [JEDEC document JESD209-4](#) — registration is required to download specs from the JEDEC site, but it’s free.)
 - “Targeted Row Refresh” (TRR) mode, which allows the memory controller to ask the DRAM device to refresh a row’s neighbours.
 - A “Maximum Activate Count” (MAC) metadata field, which specifies how many activations a row can safely endure before its neighbours need refreshing.

(The LPDDR4 spec does not mention “rowhammer” by name, but it does use the term “victim row”).

- We found that at least one DRAM vendor indicates, in their public data sheets, that they implement rowhammer mitigations internally within a DRAM device, requiring no special memory controller support.

Some of the newer models of laptops that we tested did not exhibit bit flips. A possible explanation is that these laptops implement some rowhammer mitigations.

BIOS updates and increasing refresh rates

Have hardware vendors silently rolled out any BIOS updates to mitigate the rowhammer problem by changing how the BIOS configures the CPU’s memory controller?

As an experiment, we measured the time required to cause a bit flip via double-sided hammering on one Model #4 laptop. This ran in the “less than 5 minutes” range. Then we updated the laptop’s BIOS to the latest version and re-ran the hammering test.

We initially thought this BIOS update had fixed the issue. However, after almost 40 minutes of sequentially hammering memory, some locations exhibited bit flips.

We conjecture that the BIOS update increased the DRAM refresh rate, making it harder — but not impossible — to cause enough disturbance between DRAM refresh cycles. This fits with data from Yoongu Kim et al’s paper (see Figure 4) which shows that, for some DRAM modules, a refresh period of 32ms is not short enough to reduce the error rate to zero.

We have not done a wider test of BIOS updates on other laptops.

Monitoring for row hammering using perf counters

It might be possible to detect row hammering attempts using CPUs’ performance counters. In order to hammer an area of DRAM effectively, an attacker must generate a large number of accesses to the underlying DRAM in a short amount of time. Whether this is done using CLFLUSH or using only normal memory accesses, it will generate a large number of cache misses.

Modern CPUs provide mechanisms that allow monitoring of cache misses for purposes of performance analysis. These mechanisms can be repurposed by a defender to monitor the system for sudden bursts of cache misses, as truly cache-pessimal access patterns appear to be rare in typical laptop and desktop workloads. By measuring “time elapsed per N cache misses” and monitoring for abnormal changes, we have been able to detect aggressive hammering even on systems that were running under a heavy load (a multi-core Linux kernel compile) during the attack. Unfortunately, while detection seems possible for aggressive hammering, it is unclear what to do in response, and unclear how common false positives will be.

While it is likely that attackers can adapt their attacks to evade such monitoring, this would increase the required engineering effort, making this monitoring somewhat comparable to an intrusion detection system.

On disclosures

The computing industry (of which Google is a part) is accustomed to security bugs in software. It has developed an understanding of the importance of public discussion and disclosure of security issues. Through these public discussions, it has developed a better understanding of when bugs have security implications. Though the industry is less accustomed to hardware bugs, hardware security can benefit from the same processes of public discussion and disclosure.

With this in mind, we can draw two lessons:

- **Exploitability of the bug:** Looking backward, had there been more public disclosures about the rowhammer problem, it might have been identified as an exploitable security issue sooner. It appears that vendors have known about rowhammer for a while, as shown by the presence of rowhammer mitigations in LPDDR4. It may be that vendors only considered rowhammer to be a reliability problem.
- **Evaluating machines:** Looking forward, the release of more technical information about rowhammer would aid evaluation of which machines are vulnerable and which are not. At the time of writing, it is difficult to tell which machines are definitely safe from rowhammer. Testing can show that a machine is vulnerable, but not that it is invulnerable.

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK

exploit. None of the public material we have seen on rowhammer (except for the paper by Yoongu Kim et al) discusses security implications.

However, many bugs that appear to be difficult to exploit have turned out to be exploitable. These bugs might initially appear to be “only” reliability issues, but are really security issues.

An extreme example of a hard-to-exploit bug is described in a recent Project Zero blog post (see [“The poisoned NUL byte, 2014 edition”](#)). This shows how an off-by-one NUL byte overwrite could be exploited to gain root privileges from a normal user account.

To many security researchers, especially those who practice writing proof-of-concept exploits, it is well known that bit flips can be exploitable. For example, a 2003 paper explains how to use random bit flips to escape from a Java VM. (See [“Using Memory Errors to Attack a Virtual Machine”](#) by Sudhakar Govindavajhala and Andrew W. Appel.)

Furthermore, as we have shown, rowhammer-induced bit flips are sometimes more easily exploitable than random bit flips, because they are repeatable.

On vulnerability of machines

We encourage vendors to publicly release information about past, current and future devices so that security researchers, and the public at large, can evaluate them with reference to the rowhammer problem.

The following information would be helpful:

- For each model of DRAM device:
 - Is the DRAM device susceptible to rowhammer-induced bit flips at the physical level?
 - What rowhammer mitigations does the DRAM device implement? Does it implement TRR and MAC? Does it implement mitigations that require support from the memory controller, or internal mitigations that don't require this?
- For each model of CPU:
 - What mitigations does the CPU's memory controller implement? Do these mitigations require support from the DRAM devices?
 - Is there public documentation for how to program the memory controller on machine startup?
 - Is it possible to read or write the memory controller's settings after startup, to verify mitigations or enable mitigations?
 - What scheme does the memory controller use for mapping physical addresses to DRAM row, bank and column numbers? This is useful for determining which memory access patterns can cause row hammering.
- For each BIOS: What rowhammer mitigations does the BIOS enable in the CPU's memory controller settings? For example, does the BIOS enable a double refresh rate, or enable use of TRR? Is it possible to review this?

At the time of writing, we weren't able to find publicly available information on the above in most cases.

If more of this information were available, it would be easier to assess which machines are vulnerable. It would be easier to evaluate a negative test result, i.e. the absence of bit flips during testing. We could explain that a negative result for a machine is because (for example) its DRAM implements mitigations internally, or because its DRAM isn't susceptible at the physical level (because it was manufactured using an older process), or because its BIOS enables 2x refresh. Such an explanation would give us more confidence that the negative test result occurred not because our end-to-end testing was insufficient in some way, but because the machine is genuinely not vulnerable to rowhammer.

We expect researchers will be interested in evaluating the details of rowhammer mitigation algorithms. For example, does a device count row activations (as the MAC scheme suggests they should), or does it use probabilistic methods like PARA? Will the mitigations be effective against double-sided row hammering as well as single-sided hammering? Could there be any problems if both the DRAM device and memory controller independently implement their own rowhammer mitigations?

Conclusion

We have shown two ways in which the DRAM rowhammer problem can be exploited to escalate privileges. History has shown that issues that are thought to be “only” reliability issues often have significant security implications, and the rowhammer problem is a good example of this. Many layers of software security rest on the assumption the contents of memory locations don't change unless the locations are written to.

The public discussion of software flaws and their exploitation has greatly expanded our industry's understanding of computer security in past decades, and responsible software vendors advise users when their software is vulnerable and provide updates. Though the industry is less accustomed to hardware bugs than to software bugs, we would like to encourage hardware vendors to take the same approach: thoroughly analyse the security impact of “reliability” issues, provide explanations of impact, offer mitigation strategies and — when possible — supply firmware or BIOS updates. Such discussion will lead to more secure hardware, which will benefit all users.

Credits

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

[FLERE OPLYSNINGER](#) [OK](#)

Posted by Unknown at [8:59 AM](#)

40 comments:



Zerith [March 9, 2015 at 1:18 PM](#)

Very impressive. What is scarier is that this problem will remain with us for years to come since it might not be possible to "patch" it

[Reply](#)



WANG Gai [March 9, 2015 at 1:28 PM](#)

Does the test results above make a distinction between DDR3 and DDR3L memory? It seems like that family Y and Z are completely immune to rowhammering, X is a coin-flip, and W is mostly vulnerable. It should be noted that Intel started supporting the lower voltage DDR3L as an option in Ivy Bridge machines, and then made it mandatory in Haswells and above. Could this be a factor as well?

[Reply](#)

[Replies](#)



PhilHibbs [March 10, 2015 at 3:57 AM](#)

There's only one entry for family Z so it's probably not safe to say that it's completely immune.



rasorfishsl [March 10, 2015 at 4:20 PM](#)

It would depend far more on the Silicon' construction techniques used by various memory builders.

What might be an 'interesting' test would be to see if it only applies to the complete CRAP they sell into the consumer markets or if it applies to the high level quality memory used in top end servers.

This might also be useable in 'fake' Static' rams where the actual internal array is Dram but then it attempts to look like a Static ram externally :-), by having internal refresh circuitry

If i was running research I would run tests in this area Static ram area as well, I would suspect that it might be more susceptible.



mgs [March 17, 2015 at 5:49 PM](#)

Rowhammer is a design defect in the memory array. It will show up in all devices in a manufacturers' process node regardless of 2/3/4/3L or LP. It cannot be screened out.

Pseudo-SRAM could also be susceptible if they are built with a defective bitcell design.

[Reply](#)



Aaron Miller [March 9, 2015 at 2:47 PM](#)

This comment has been removed by the author.

[Reply](#)

Anonymous [March 9, 2015 at 2:58 PM](#)

aww you ruined my crowdfunding campaign on Darkleaks

[Reply](#)



David Wren [March 9, 2015 at 5:19 PM](#)

For testing the full memory range, outside of an operating system, there is also MemTest86 V6.0, which also implements a row hammer test.

[Reply](#)

Anonymous [March 9, 2015 at 7:08 PM](#)

Glad desktops have better chances of being safe (although all of my credit cards are in the cloud) :) I turned this off after 500 iterations on desktop with not result while executing on a popular hosting provider exited on the first one!

[Reply](#)

Anababa [March 9, 2015 at 11:09 PM](#)

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK



Jolly Good March 10, 2015 at 12:34 AM

AMD makes available the information you've requested but were unable to find. See the aptly named BIOS and Kernel Developer's Guides, which have been publicly available for years.

An example of from their most recent "Steamroller" cores:
http://support.amd.com/TechDocs/49125_15h_Models_30h-3Fh_BKDG.pdf

Of your requests:

* "What mitigations does the CPU's memory controller implement? Do these mitigations require support from the DRAM devices?"

Do you want to know which mitigation schemes from the research paper are available? Likely few! It may not surprise you, but hardware takes a long time to change (assuredly longer than software).

If you want to know current DRAM refresh time, see D18F2x8C on page 365 for each DRAM controller. Bits 17:16 define the hardware refresh rate. A value of 3 in this field means the refresh rate is double normal. Section 2.9.14 "DRAM On DIMM Thermal Management and Power Capping" explains that the controller may speed refresh rate because of temperature.

* "Is there public documentation for how to program the memory controller on machine startup?"

Section 2.9 "DRAM Controllers (DCTs)" contains the subsection 2.9.9, "DCT/DRAM Initialization and Resume".

* "What scheme does the memory controller use for mapping physical addresses to DRAM row, bank and column numbers?"

This is set per DRAM controller. See the definition for D18F2x80 on page 363. Also Section 2.9.11 for the Channel and Chip Select interleaving settings. You may also want to read the definition for D18F2x[5C:40] on pages 358-360 to see how chip select is done.

* "Is it possible to read or write the memory controller's settings after startup, to verify mitigations or enable mitigations?"

Reading: Yes, MSRs and PCI configuration space values make this available to kernel mode software as shown in BKDG.

Writing: This is harder. Many DRAM controller settings require BIOS intervention and may break in protected mode. Worth trying!

[Reply](#)



Bostjan March 10, 2015 at 12:40 AM

This comment has been removed by the author.

[Reply](#)



Bostjan March 10, 2015 at 12:40 AM

If I understand correctly now everybody knows about the problem and the bad guys can build the special exploit software and exploit the vulnerability and we cannot do anything about it? The DRAM cannot be fixed/patched. Really?

[Reply](#)

[Replies](#)



mgs March 17, 2015 at 5:51 PM

This comment has been removed by the author.

[Reply](#)



Albert March 11, 2015 at 7:33 AM

Hi! Is this also an issue for PPC? Look at section 11.5 here:
http://cache.freescale.com/files/32bit/doc/ref_manual/E500C0RERM.pdf
 Could cflflush be replaced in the rowhammer-test program by one of those instructions? Which one in that case?

[Reply](#)



Guido Aulisi March 11, 2015 at 11:27 AM

For me this is simply a NOT WORKING memory and it should be retired from the market.

[Reply](#)

[Replies](#)



mgs March 17, 2015 at 5:51 PM

DRAM suppliers agree. Everyone should buy new DRAM. I wonder if they planned this.

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK



This comment has been removed by the author.

[Reply](#)

Anonymous [March 12, 2015 at 1:25 AM](#)

The problem seems not solvable by DRAM-testing at the memory-manufacturers. Even if the testing gets intensified, DRAMs degrade and might show the issues later on.

DRAM memory is not like a simple switch that is either on or off. The databits are stored in leaky capacitors having a size of a few nanometers, holding a charge of less than 10 to the power of -15 coulombs (that is "femtocoulombs"). If the charge is above a certain level, the bit is a 1 and otherwise a 0. The charge leaks out within a few milliseconds and must be refreshed by the CPU. At the same time we expect the memory cells to be able to change their content 1.6 billion times per second or more, we want billions of bits to be in a tiny little chip and it should please consume practically no power.

DRAM memory is extremely sensitive. All the cells are connected to each other by bitlines, wordlines, address lines. If you store a pattern of bits into one corner, some of the power leaks out to other memory cells and affects them. Some cells have better isolation and can hold the data a few milliseconds longer than others. Heat, antennas, radiation, electromagnetic fields stress the memory. DRAM memory also degrades over time. That means: New DRAM memory often is failure-free for a while, but at some point of time sometimes shows a single bit-flip here or there.

Bit-flips in DRAM do happen quite often, but they are not 'defects'. They are transient effects having many possible root-causes. Overwriting the flipped bit works and the issue is not directly reproducible. This explains why every crashed application works fine again after a Reset, simply because there is no 'permanent defect', but just a 'bit-flip effect'.

I recommend to read this document: <http://users.ece.cmu.edu/~yoonguk/papers/liu-isca13.pdf>

It shows further examples of why bit-flips occur unexpectedly.

The only good news is: ECC error correction helps to solve or at least reduce the problem.

ECC does not only protect from hacker-attacks by row-hammering, it also improves the overall stability.

While a PC tends to fail sometimes with freezing mousepointers, crashing programs, erratic menu-bars or even blue-screens, you won't find such issues on systems equipped with ECC. Most servers use ECC and run for years without ever having to be rebooted. Some of the High-End Cisco routers use ECC-memory and never have issues.

For PCs and laptops, there are new memory-modules coming to the market which are built with Intelligent Memory ECC DRAMs. These are Non-ECC DIMMs and SO-DIMMs made of DRAMs that have an on-chip ECC correction (called ECC DRAM, read more here: http://www.intelligentmemory.com/fileadmin/download/PB_IM_ECC_DRAM.pdf).

The modules made of those ECC DRAMs will be called intECC-modules and are about to hit the market next week. Unfortunately the maximum memory capacity is limited to 2GB per module for the time being.

The Intelligent Memory ECC DRAM chips which are used on the upcoming intECC memory modules, could also be used on many other electronic products of our daily life to make them more reliable.

How often does your WiFi-Router, smartphone, settop box, DVR or other electronics need to be rebooted? I bet the first thing we all learn about a new device is how to restart it. Many or maybe most of these hiccups are bit-errors in the memory, which ECC protects from.

Even HDD drives use a DRAM-memory chip for the write-buffer/cache. A bit flip in the DRAM on the HDD can cause those ugly effects of "file is corrupted and can not be opened".

If the manufacturers would replace the conventional unprotected memory chips with those new ECC DRAMs, I am sure we'd see a new level of reliability with much less failures.

[Reply](#)

[Replies](#)



randomusername9 [March 12, 2015 at 7:26 AM](#)

Alas, most such issues as you describe are caused by bugs in the software - in the kernel, the applications, or the libraries used. The errors caused by ram errors are comparatively infrequent. I have definitely found strong correlation of bugginess with particular applications or device driver versions, no matter what hardware it is running on. ECC ram is nice, but the stability you are seeing is simply due to a lot of work by the engineers working on the higher price systems.

Anonymous [March 12, 2015 at 11:10 AM](#)

Well, if this was the case, then why does a server with ECC - running Windows server and many complex software 24/7 for years - never fail? ECC does not help when the software is buggy, but the PC also runs Windows and less complex software, but you can hardly keep it switched on for longer than a day and still randomly have unexpected erratic behaviour sometimes.



mgs [March 17, 2015 at 5:56 PM](#)

Rowhammer is a defect of the memory array design, not a degradation of the device, although degradation DOES happen it is not related to row hammer.

Rowhammer is a problem because it causes corruption of multiple bits in a row. This cannot be corrected by ECC.



ruckb [March 24, 2015 at 5:30 AM](#)

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPPLYSNINGER OK

that single bit flips on multiple DRAMs will happen.
if you use DRAM with integrated ECC as the Intelligent memory chips in combination with controller based ECC you would have double protection: one for a fail inside a DRAM and one for single bit fails over multiple DRAMs.
==> on DRAMs it will always come down to statistics and propability when talking about fails .. but with ECC (controller and/or DRAM based) the propability to get fails is reduced by magnitudes ..
Hermann



mgs March 24, 2015 at 9:42 AM

Hermann, I'm not sure why you mention the statistical implications. We are not discussing typical system operation where statistical behavior applies, rather a pathological worst case exploit in which you need just one key to open the door. Yes, ECC covers SECEDED, but can you explain why once you go beyond 2 bit errors you claim this is not a risk?

As I understand it RH has the potential in the worst case to corrupt all the bits in the physically adjacent row. Under this scenario what mechanisms does the system have to detect or protect against this? Once that is determined we can then delve into the statistics of the implications of partial row corruption, all the way down to 3 bits. Thanks,



ruckb March 24, 2015 at 10:23 AM

Hi mgs,
in DRAM everything is about statistics .. Even DRAM spec up to DDR3 assumes a BER=0 (on interface and array) this is just a nice assumption but just not valid .. just DDR4 starts to take this into account. .. All fail mechanisms (including RowHammer) are just a matter of probability if it fails and if the fail can be corrected or detected with any means. Getting a 3bit error is haveing a low probability and once you got it you will have a statistical probability that you are going to catch it even with SECEDED. I did not mention that it is no risk, I'm just saying this is question of statistics.

Yes, row hammer can have the potential to corrupt bits in adjacent rows (if the DRAM vendor did not a good job in either designing or testing the array).

You don't know at which point in time a single bit error occurs, and once you do a readout with ECC you will correct it. if you wait longer you might get a double bit error and an ECC based system will stop and tell you that it run into an uncorrectable error. If you get more than 2 errors there are still chances that that the error is detected and the system stops. if you run rowhammer too long you have so many errors that you can not attack the system any more, as the code crashes completely .. so we are back in probabilities if you want to use this behavior for attacking a system. And ECC will lower the probability that you will get a fail allows the attack.

If you combine now Chipset ECC (correcting one bit fail over 4, 8 or 16DRAM DQ width) with DRAM devices with built in ECC (correcting one fail for each DRAM burst) you are going to decrease the probability for a Fail that allows the attack even further.

And even if a single bit fails it need to be the right bit to allow the attack ..

Thats the reason why I think especially with ECC it's all about statistics for such a scenario ...

The question for me is: Why this discussion on RowHammer ?

You can get you fails in the DRAM much simpler, but maybe with less accuracy where the fail happens. Is this really less critical for security ?

- There is no protection to change the controller registers during runtime. So adjusting the Refresh times is very simple. Now you can either try rowhammer on with this weekend array or just access all other WL in order to refresh these by read accesses ..

- Maybe you want to check your SPD on the DIMMs ... is the Software Write protect really set ?

Change write recovery times for AP and do a lot of reads to one Row. Nobody says that the critical addressspace does not share one row. So you can introduce errors along the row ..

There are so many options to get a Fail on a DRAM .. it is just a question of statistics and probability which one leads fastest to a Fail that allows to attack a system. With ECC (on controller or on DRAM) this probability is just lowered a lot .. but still not 0.

Hermann



mgs March 24, 2015 at 10:53 AM

It is just a given that the DRAM vendors as a whole did do a poor job designing the array. Prior to this it was assumed the DRAM were not susceptible to RH and on secure systems I'd assume the alternate attacks you mention are not possible. So in the past there was a zero risk from this. Now by introducing susceptible DRAM into a secure system the risk is non-zero and needs to be mitigated. It seems that every DRAM produced in the last 5 years has this susceptibility. That's a huge installed base at risk.



ruckb March 24, 2015 at 11:43 AM

Hi mgs, sorry to destroy your secure world ;-) ..

RowHammer is an old issue and DRAM has never been save (at least in my understanding).. in order to get save DRAM systems people implement things like chipkill, DIMM redundancy and other fancy features .. but only to lower the risk of a fail (or attack), not to have zero risk.

And I don't agree on DRAM vendors as a whole did a pure job. It is really a difficult task to Design a DRAM array and test it properly. How good it is done (especially the testing) is economy driven.

If you spend enough money you can buy DRAM and Systems that lower the probability of a fail (that can be used for an attack) to very low numbers .. but industry is not willing to spend this money (and I have to admit, that also my system is having no ECC for the 100GB of Memory installed.)

Seems that probability of fail is still with conventional methods low enough ..

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK

Hermann



mgs March 24, 2015 at 12:37 PM

This comment has been removed by the author.



mgs March 24, 2015 at 12:44 PM

I suppose you are right. The systems which are capable of being secured against the attacks you describe are built by companies who are also capable of and have already mitigated against RH.

I used to program ATE memory test a long time ago and row hammer never failed. I think this problem emerged when the DRAM makers made a conscious decision to design an array that had less margin to this failure mode. They were assuming that statistically, during normal system operation, that RH would never occur, and they were right. Until one day a certain server CPU maker decided to allow customers to turn off caching to speed up database performance. That's when row hammer re-appeared having been dormant for nearly 30 years, and caused system crashes on ECC protected, chipkill protected, high end enterprise servers. That was several years ago. The patents didn't start to appear until after that.



mgs March 24, 2015 at 12:46 PM

and even so, row hammer defective DRAMs are still being produced.

[Reply](#)



Asterix March 13, 2015 at 1:09 PM

So consumer-grade gear lacking ECC protection doesn't use anything at all to detect even single-bit errors? Heck, the original IBM PC used DRAM parity.

[Reply](#)

[Replies](#)



mgs March 17, 2015 at 5:59 PM

The IBM PC was a business machine and was built with DRAM that inherently had soft errors. Eventually soft errors were bought down to a tolerable level and parity was dropped. This row-hammer problem is a really big oops. Fortunately DRAM makers have designed it out (again) in their newer array designs.

[Reply](#)



FuturePlusSystems March 16, 2015 at 9:30 AM

We have been writing about this topic for almost 9 months now. Glad Google gave us a mention. We created a feature in our DDR Detective tool to detect excessive ACTIVATE (ACT) commands to a single row address on a DIMM or SODIMM module. We have posted videos on our YouTube channel giving more information (FuturePlus Systems). If you have critical applications I highly recommend you test to see if your application creates these excessive ACT commands. Our tool is available for short term rental. Please contact us barb.aichinger@futureplus.com if you would like more information. Also check out www.DDRDetective.com/row-hammer for more information

[Reply](#)

crem March 25, 2015 at 6:25 AM

Amused me somewhat reading this that one of the first computers I built, back in 1979, used 4K x1 DRAM chips that ICL had rejected (and essentially thrown away) because they suffered from what they called pattern sensitivity failures... essentially the same problem as this, writing certain patterns into memory could induce changes in nearby bits if the memory timing was near the limit.

[Reply](#)



Chaojian Hu April 16, 2015 at 9:42 PM

```
> In https://github.com/google/rowhammer-test
>
> How to run the test:
>
> ./make.sh
> ./rowhammer_test
> The test should work on Linux or Mac OS X, on x86 only.
```

Why on x86 only, is x86-64 is also ok?

x86 only.

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK

I get some results from our own row-hammer tests performed on non-ECC modules on different platforms using Memtest 6.0. All platforms failed the row-hammer test with different memories sooner or later. Every single fail was a single bit flip and there were no double-bit flips at all. The amount of single-bit-flips per each pass of the test was between 3 and 40 bit-flips and spread over multiple different addresses and different DRAMs on the modules.

Let's say a memory module uses 8 Chips of 2Gbit capacity each = 2GByte (256Mx64).

With 40 bit-flips in a row-hammer-test run by Memtest, this is averaging to 5 bit-flips per DRAM chip throughout a complete row-hammer test-pass. Unfortunately I do not know how often Memtest hammers the rows in one pass.

The modules output data in 64 bit-words. There are 256 Million of such 64 Bit words in a 2GB module. By having ECC, each of these 256 million 64 bit words could have one bit-flip and ECC would still be able to correct them.

40 bit-flips within 256 million 64 bit-words is 'peanuts' for ECC! With such small amount of bit-flips it is very unlikely that TWO of those bit-flips occur in the same 64 bit-word causing an uncorrectable double-bit error.

This said, ECC is a very effective row-hammer protection-method.

I doubt that TRR, MAC or increasing the refresh-rate can solve the problem. The theory is that adjacent rows survive a certain amount of row-hammers before they better should be refreshed. "Testing" if the DRAMs can take 300k hammer-cycles is not a guarantee that it will always survive 300k hammers. It could also sometimes fail after just a handful of accesses when you are unlucky. It is not the "amount of cycles" that causes the bit-flips.

Also: Don't forget that DRAMs do degredate and get more sensitive to disturbances after some time of use. A brandnew DRAM might be surviving 300k hammers, but what if it gets a bit older?

Besides row-hammering, DRAMs are also sensitive to heat and to radiation (from antennas or radioactivity, even natural ground-radiation). VRT-effects (variable retention time) and data-pattern-dependence (the retention time of each DRAM cell is significantly affected by the data stored in other DRAM cells) also cause bit-flips. TRR and MAC does not help against this at all.

The only solution is to detect and correct the bit-flips -> ECC Error Correction

Row-Hammer is a typical hacker-attack (=intentional), while all the other issues are non-intentional bit-flips. If you are unlucky, it does not even need a row-hammer software, but a simple multi-read of a row by your own software-code can cause a bit-flip in an adjacent row.

ANY bit-flip - no matter if intentionally triggered or not - can cause a malfunction of the software/the system, or result in a crash or data corruption. Please do not only look at PCs, laptops or servers, but also think of all the other intelligent electronics out there, like networking systems, access controls, telecommunication, medical devices, automotive, industrial computers of all kinds, etc. These will typically not be row-hammered, but they still sometimes fail and need to be reset, although they are expected to always run stable.

I think everybody had the experience that the smartphone, the settop box, the router or other electronics "needed a reset" -> this was most-likely caused by a bit-flip. While home-electronics are less critical, most industrial electronics are expected not to have such issues. Still they do.

All bit-flips have the same root-cause -> weaknesses of memory-cells

The best solution for ALL above mentioned issues is to have ECC

With ECC, all applications could become a lot more stable. They would run for years without ever having to be reset/rebooted. And at the same time they will also be row-hammer-safer.

The DRAM makers should follow the trend that Intelligent Memory started: Integrate ECC into the DRAM-chips instead of trying to find insecure workarounds only against row-hammering by MAC, TRR, faster refreshing, etc.

[Reply](#)



GuruPrasad Reddy February 17, 2016 at 12:29 AM

Can anyone suggest the equivalent instruction of CLFLUSH for ARM Processor. The CLFLUSH is specific to x86 Processor

[Reply](#)

[Replies](#)



heliopoopter June 18, 2016 at 4:03 PM

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0092b/DDI0092B_ARM940T_TRM.pdf
Page 25

[Reply](#)



Mrrr J January 26, 2017 at 5:42 AM

How long exactly did you leave those test running to find those bit flips?

Are we talking about in 5 minutes, hours, days?

[Reply](#)



Unknown September 20, 2018 at 11:51 PM

I am confused as to why this problem is not treated as a memory defect due to which under certain conditions when a row is accessed many times the charge is leaks between cells in different rows? this sounds like something that the dram vendor could easily screen for with a memory test algorithm during production like they already do for other

Dette website anvender cookies fra Google til at levere sine tjenester og analysere trafik. Din IP-adresse og brugeragent deles med Google sammen med effektivitets- og sikkerhedsmetrics. Dette sker for at sikre tjenestekvaliteten, generere brugsstatistik samt registrere og håndtere misbrug.

FLERE OPLYSNINGER OK

Enter your comment...

Comment as:

Google Accoun ▾

Publish

Preview

[Newer Post](#)

[Home](#)

[Older Post](#)

Subscribe to: [Post Comments \(Atom\)](#)