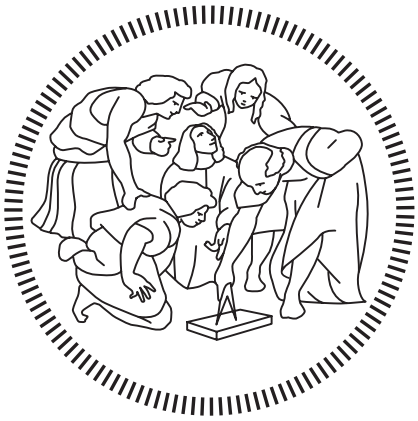
POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Dipartimento di Elettronica, Informazione e Bioingegneria



Reliably achieving and efficiently preventing Rowhammer attacks.

Relatore: Prof. Alessandro BARENGHI

Relatore: Prof. Gerardo PELOSI

Tesi di Laurea di:

Niccolò IZZO, Matr. 852226

Anno Accademico 2017-2018

To my other half.

Author's thanks

Exploring the unknown is always difficult. It is also a daily occurrence when doing scientific research. During my work, I have lost my way and motivation several times, but I have never been alone. I would like to thank some of the people which made my journey possible:

My advisor Alessandro Barenghi for helping me solve even the most difficult problems always in elegant ways.

My second advisor Gerardo Pelosi for showing me how to do serious and reliable research and always question my previous knowledge.

Davide Zoni for his helpful advices on how to implement my hardware countermeasure.

My girlfriend, my brother, friends and family who stood next to me in this long path.

Prefazione

Lo scopo di questo lavoro di tesi è un'analisi approfondita del fenomeno denominato *Rowhammer*, esso si manifesta con una **corruzione selettiva** dei dati contenuti in una memoria *SDRAM*, a seguito dell'esecuzione di un determinato ***pattern* di accessi** alla memoria stessa. *Rowhammer* si verifica anche con accessi in **sola lettura**, di conseguenza il fenomeno può causare la modifica dei dati all'interno di zone di memoria marcate come *read-only*. Questa eventualità è in grado di minare la solidità di un gran numero di meccanismi di sicurezza che si affidano a permessi di sola lettura, come la separazione *userspace-kernelspace*, le azioni di *sandboxing* o l'impiego di coppie di chiavi crittografiche asimmetriche.

Per poter riprodurre il fenomeno di *Rowhammer* su un sistema suscettibile all'attacco è necessario conoscere approfonditamente come gli **indirizzi virtuali** dei processi in esecuzione nel sistema si mappano sulle **strutture interne** dei moduli *DRAM* del sistema stesso. In questa tesi si descrive un **metodo completo e corretto** per ricavare la funzione di mapping del sistema su cui un processo è in esecuzione.

Infine si espongono due contromisure, una software e l'altra hardware per **mitigare** il fenomeno di *Rowhammer* fino a renderlo inoffensivo.

Il *Rowhammering* si verifica solo su righe di memoria fisicamente adiacenti, la **funzione di mapping** determina quali **indirizzi virtuali** sono geometricamente adiacenti in un determinato *chip* di memoria *DRAM*.

La contromisura software consiste nel modificare l'allocatore di memoria del sistema operativo, per far si che le varie entità (singoli processi e kernel) siano **fisicamente isolate** nella loro allocazione in memoria da un numero sufficiente di *row*. Di conseguenza ogni processo, se anche riuscisse a causare errori di disturbo, essi avrebbero effetto solo sulle sue strutture dati, impedendo qualsiasi tipo di *privilege escalation*.

Invece dal lato hardware si propone una modifica dell'architettura dei moduli *DRAM*, di modo che ad ogni inizializzazione del modulo, sia generata una **permutazione casuale** dei *row index*. In questo modo la **funzione di mapping** cambierà ad ogni avvio della macchina , riducendo drasticamente la fattibilità di un attacco basato su *Rowhammer*.

Preface

The purpose of this thesis work is an in-depth analysis of the phenomenon named *Rowhammer*. *Rowhammer* manifests itself as a **selective corruption** of data inside *SDRAM* memories, following the execution of a particular **access pattern** to that same memory. *Rowhammer* manifests itself even with **read-only** accesses, therefore the phenomenon may cause modification of data inside *read-only* memory regions. This eventuality can undermine the solidity of a large number of security mechanisms which rely on read-only permissions, such as *userspace-kernelspace* separation, *sandboxing* processes or the use of asymmetric cryptographic keys.

To reproduce the *Rowhammer* phenomenon on a system susceptible to this attack, it is necessary to have a deep knowledge of how processes'**virtual addresses** map onto the **internal structures** of the *DRAM* modules in use on the system.
In this thesis a **complete and correct** method for obtaining the mapping function of the system on which a process is executed is described.

Finally two countermeasures are shown: a software one and a hardware one, to **mitigate** the *Rowhammer* phenomenon until it becomes harmless.

Rowhammering happens only on physically adjacent *rows* of memory, a **mapping function** determines which **virtual addresses** are geometrically adjacent in a *DRAM chip*.

The software countermeasure consists of a modification of the OS'memory allocator to make it so that the various entities (individual processes and the kernel), be **physically isolated** by a sufficient number of *rows*. Consequently, even if a process caused disturbance errors, the latter would affect only the data structures of said process. Hence impeding any type of *privilege escalation*.

On the hardware side, we propose a *DRAM* modules architectural modification, such that everytime the module is initialized, a **random permutation** of *row index* is generated. This way, the **mapping function** will change at every machine boot, drastically reducing the feasibility of a *Rowhammer*-based attack.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

All the research efforts of memory manufacturers for the last ten years, have been focused on increasing cell density on silicon. This brought us extremely dense memory chips, and nowadays it is common to find *DRAM* chips with 2GB capacity each, soldered in groups onto memory modules.

This increasing density however amplifies all the disturbance effects that hit the *DRAM* circuits: subthreshold leakage, **gate-induced drain leakage** and **row to row coupling**. These disturbance effects have been known to the *DRAM* manufacturers for a long time [5], even so, they were always considered as a mere reliability concern, not a security one.

The ability to induce and use these disturbance effects to selectively corrupt **read-only** memory content is dangerous, because it takes away one of the cornerstones of any security mechanism: the separation of privileges.

In fact with a base knowledge of the victim system, and with some key elements i.e. a fast access to *DRAM* memory, even an unprivileged or a sandboxed process can trigger the vulnerability and exploit it.

The *Rowhammer* [6] vulnerability has been exploited even from javascript code, running inside a webpage loaded by a browser [7]. This gives an idea of the subtlety of such an attack, which can be triggered even from high-level sandboxed code and leverage a hardware vulnerability.

When an attacker is able to alter data marked as *read-only*, a whole

group of attacks becomes feasible. All the Operative Systems which feature memory virtualization keep some data structures called **page table entries** (**PTE**s) in memory. These entries keep track of the mapping between **virtual addresses** and **physical addresses**, and information about whether the page is writable or accessible from userspace [8].

By definition, *PTE*s are *read-only*, otherwise a process could change the **physical address** at which its **virtual pages** are mapped and thus gain arbitrary writes over the whole memory of the execution environment. By leveraging the *Rowhammer* vulnerability, an attacker can corrupt a *PTE*, by changing the mapping of one of its **virtual page** to another of its own *PTE*. After that, the second *PTE* can be edited freely, for example making it point to one of the **kernel data structures**, to modify them and gain **root privileges** over the machine. This attack has been implemented by Google's Project Zero [9].

Another possible use of the *Rowhammer* vulnerability is to corrupt asymmetric cryptographic keys. It has been demonstrated by Razavi et al. [10] that it is possible to use *Rowhammer* to corrupt an OpenSSH public key. Therefore the resulting corrupted key might become easily factorizable, and thus a corresponding private key can be easily derived and used to gain remote access to the system.

A vulnerability that originates directly from the *DRAM* such as *Rowhammer* is particularly critical, because nowadays there are countless devices which run user-controllable code and rely on *DDR3* or *DDR4* SDRAM. Among those devices are smartphones, that constitute particularly sensitive targets because they carry our most personal data and are equipped with microphones and cameras. As if this were not enough they are directly connected to the internet and we keep them always close to ourselves.

Researchers from VUSec have shown practical examples of *Rowhammer* attacks on Android devices [2], by using a deterministic method to

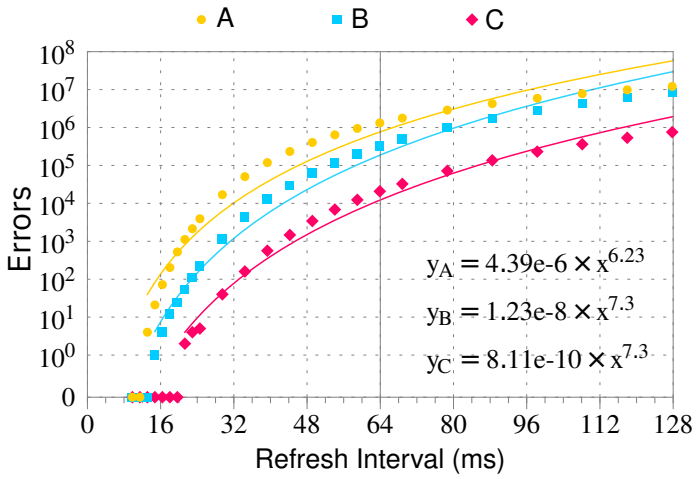


Figure 1: Disturbance Errors Number over *DRAM Refresh Interval*. Graph rebuilt from Kim, Yoongu et al. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors" [1].

control the Android **memory allocator**.

Since the problem is widespread on so many different platforms it is important to create **machine-independent** solutions which are flexible enough to be ported to the most diverse systems.

The one between security and performance has always been a tough battle, and even in this case, securing our systems does mean giving up on performance or energy consumption.

As shown in the original *Rowhammer* paper [1], incrementing the *Row Refresh Rate* under the 16ms threshold reduces the number of disturbance errors greatly, as can be seen in figure 1. Apple introduced an EFI patch to bring the *Row Refresh Period* from the standard 64ms to 32ms [11]. When using the normal 64ms *Refresh Period* the *DRAM* is unavailable 2.2% of the time; halving the period makes the *DRAM* unavailable for 4.4% of the total time thus degrading performance. Furthermore more frequent refreshes consume more power, and 32ms are not enough to reduce significatively the number of disturbance errors.

Our aim is providing a solution which greatly reduces the feasibility

of a *Rowhammer-based* attack without surrendering to heavy performance costs. Such kind of solutions will be easily portable even to less capable devices, without degrading the overall user experience, and at the same time keeping those systems reasonably secure.

Chapter 1

Origins of the phenomenon

1.1 Memory structure

Modern *DRAM* modules all share the same **organization criterion** for their internal components. For our analysis we will be considering the *DIMM* form factor, which has been the most commonly used on desktop computers, workstations and servers in the last decade.

A single *DIMM* stick is called a **module**, it is composed by a printed circuit board with chips soldered on its surface and electrical contacts on both sides of one edge of the board (figure 1.1).

Different *DIMM* sockets may be connected to different *channels*; a *channel* is an independent bus from the memory controller to a module (figure 1.2).

Every module can be either one-sided or two-sided, depending on how many sides of the board have been populated with memory chips. A module presents many chips on its surface, and always features a control logic called *chip select* which allows to interconnect all the chips to the same bus. A set of *DRAM* chips connected with the same *chip select* logic is called a *rank* (figure 1.2). Usually, in two-sided modules, there is one

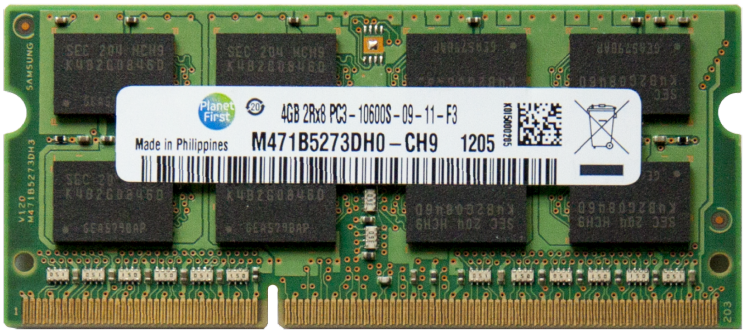
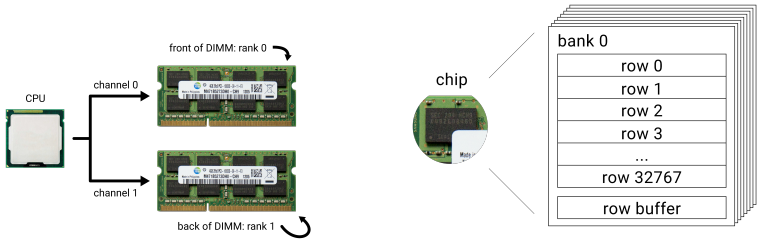


Figure 1.1: DDR3 *SO-DIMM* Module



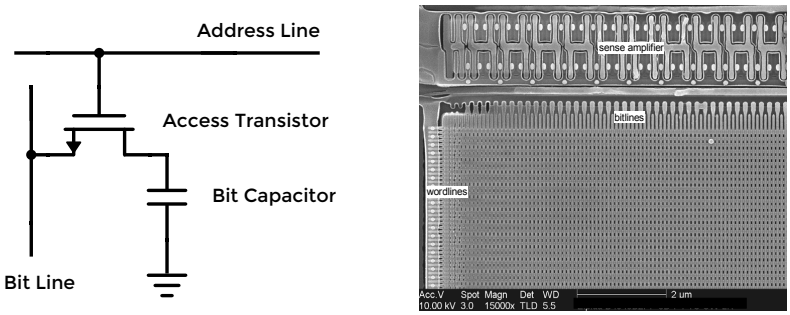
(a) Single-bit Storage Cell (b) Bank Die-Shot

Figure 1.2: SO-DIMM Modules Organization

rank per side.

Every *DRAM* chip contains one or more *banks*, each *bank* is a large collection of *rows*, with one *row-buffer* (figure 1.2).

Rows represent the smallest granularity of storage which can be read or written inside a *DRAM chip*. Each *row* is several bytes long (8192B tipically), but to read even a single byte, the whole *row* must be read. The same happens for writing. Furthermore the *DDR3* standard requires burst reads to hide the memory latency, therefore the memory controller will always issue an 8 bytes consecutive read, even to read a single byte. The same happens also for writes.



(a) Single-bit Storage Cell (b) Bank Die-Shot

Figure 1.3: Storage Cell Scheme and Die-Shot of a DRAM Bank, Photo from Chipworks. "DRAM Process Report, Sample Report." URL: `https://www.chipworks.com/TOC/DRAM_Process_Report-Sample.pdf`.

In a *DRAM* module every bit is represented by a little capacitor, connected to an *access transistor* (figure 1.3). A wire called *address line* connects all the gates of the transistors of each bit in a *row* together. Furthermore there are as many *bit lines* as memory cells in every *row*.

The *row buffer* is an array of *latch-amplifier* pairs and it has the same length of a *row*. Every amplifier (*sense amplifier*) is connected to a different *bit line* (figure 1.3).

When a memory *word* has to be read, the corresponding *address line* is activated, switching-on all the transistor of the *row* in which the *word* resides. The charges of all the capacitors of the *row* are sensed by the *sense amplifiers*, which save them in the corresponding *latches*. In this process, termed *row activation*, all the capacitors of the *row* are **discharged**. As a consequence they will have to be replenished with the same values (in case of memory read) or with updated values (in case of memory write).

If more words are read or written in the same *row*, only a single *row activation* is issued and the following reads and writes are made directly on the *row buffer*'s registers. Furthermore, all the charges in the capacitors tend to fade naturally due to *subthreshold leakage* [12] and *gate-induced drain leakage* [13].

As a consequence, *rows* have to be continuously read and rewritten, this process is called *refresh*. In all modern *DRAM* modules, a refresh is issued with a mean period of 64ms.

1.2 Why does *Rowhammer* happen

The *Rowhammer* phenomenon manifests itself as a bit corruption inside a particular *row* that happens when the adjacent *address lines* are repeatedly activated and deactivated with a frequency much higher than the *refresh frequency*.

Kim et al.'s paper [1] gives a physical explanation of the *Rowhammer* phenomenon and poses some necessary conditions to make it happen: When the voltage of an *address line* is commuted repeatedly, i.e. the cell is *hammered*, the transistors of the *bit-storage cells* of the adjacent *rows* are activated due to **gate-induced drain leakage** and **row to row coupling**. This causes the cells of the *rows* adjacent to the *hammered* one to lose charge at a faster speed. If a cell drops below the logic level threshold before the next *row refresh*, then its value changes and the next refresh will confirm the corrupted value (figure 1.4).

This example of assembler code, provided some conditions hold on the two addresses X and Y, can trigger a disturbance error on other addresses in memory.

```
rowhammer_clflush:
    mov (X), %eax
    mov (Y), %ebx
    clflush (X)
    clflush (Y)
    jmp rowhammer_clflush
```

To find disturbance errors on the neighbouring *rows*, the two addresses

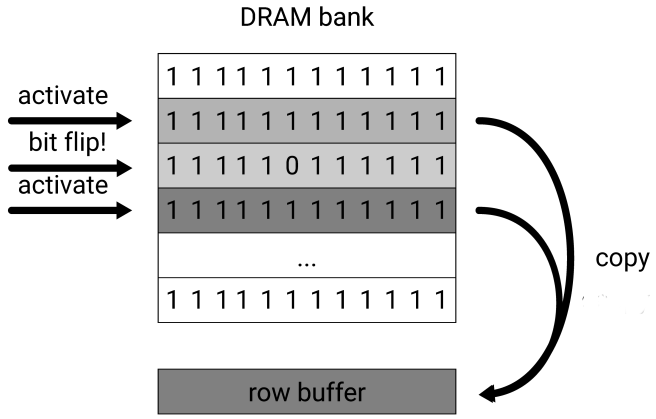


Figure 1.4: *Rowhammer* Bit-Flip Logical Scheme.

X and Y must belong to the same *bank* and to different *row*. We will later on refer to this condition as the *Same Bank Different Row Condition* or *SBDR Condition*.

This happens because every *bank* has a single *row buffer* and when a byte is requested in a *row* different from the current *open row*, the *current row* has to be closed and the *next row* has to be opened.

Differently, if the two addresses belong to different *banks*, their access can happen in parallel, without the need of repeatedly opening and closing two *rows*. Furthermore, if the two addresses belong to the same *row*, all the reads and writes will happen to the *row buffer* without causing new *address line* commutations.

1.3 Single vs Double Rowhammer

To trigger the *Rowhammer* vulnerability we need at least two addresses that satisfy the *SBDR* condition, this way we can access them repeatedly and cause the *rows* to open and close at each access. If the two *rows* are far away from each other they will trigger the faults in two different zones of

the target *bank*. This kind of attack can be defined *Single Rowhammer*, because each fault is caused by a single *row* repeatedly opening and closing. If we move the two *rows* to be at one *row* of distance from one another, like in figure 1.4, we will cause a more powerful attack that we can call *Double Rowhammer*. This attack is much more dangerous because the victim *row* (the one between the two attacking *rows*) is suffering from the cumulative disturbance effects of the two *rows* simultaneously. However triggering a *Double Rowhammer* attack is much more difficult than a *Single Rowhammer*, in fact finding two addresses in the same *bank* is relatively easy, while in order to find two adjacent *rows*, we need a deep knowledge of the **memory mapping function** that maps *virtual addresses* to geometrical parts of each *DRAM* module.

Chapter 2

State of the Art

In this chapter we will review the current state of the art for the realization and mitigation of *Rowhammer* attacks.

2.1 Attack primitives

To achieve a successfult *Rowhammer* attack, memory accesses must have three characteristics: they have to be uncached, fast and targeted.

2.1.1 Uncached accesses

If memory accesses hit one of the cache levels, they will never reach the *DRAM* and thus never cause any *address line* modification. Several techniques that generate uncached access to memory have been described in the literature. They mainly rely on four types of mechanisms: flushing caches, using non-temporal instructions, evicting caches and using uncached memory primitives. The first can be achieved in the x86-64 architecture by using the **clflush** instruction ([1]). This kind of instruction flushes all the levels of caches, and thus guarantees subsequent memory operations to be performed directly on the *DRAM*. As a consequence of the

usefulness of the **clflush** instruction for performing *Rowhammer* attacks, it has been removed from the allowed instructions in the NaCl sanbox (the sandbox used by the Chrome browser) [9]. Another way to obtain uncached accesses is to use non-temporal instructions [14], such as **movnti** or movntdq on a x86_64 architecture. These instructions define a *non-temporal hint*, so the CPU avoids caching data. However, it is to be noted that the memory used by these instructions is treated as *Write Combining* type. Consequently, non-temporal writes to the same address are always combined at WC buffer, and only the last write goes through towards the *DRAM* chips. This will make any naive implementation with these two primitives ineffective. The *Write Combining* cache can be flushed by issuing a cached memory access, thus the following assembly loop is able to trigger the vulnerability:

```
rowhammer_movnti:
  movnti %eax, (X)
  movnti %eax, (Y)
  mov %eax, (X)
  mov %eax, (Y)
  jmp rowhammer_movnti
```

A third way to issue uncached reads is to perform a cache eviction. This method was used to create *rowhammer.js* since the **clflush** instruction is not available in javascript [7]. The cache eviction is performed by reverse-engineering the cache mapping function and eviction policies, and trying to issue a set of writes targeted to fill up the caches quickly and perform uncached reads.

A fourth and last way to generate uncached reads is to use different forms of uncached memory i.e. the ION Android memory allocator [2]. From Android 4.0 on, the ION memory manager was introduced [15], allowing developers to use unified DMA Buffer Management APIs. This en-

ables even userland apps to obtain fast, uncached, access to physically contiguous memory.

2.1.2 Fast accesses

Rowhammer can be considered as a race against the *rows refresh*. In order to win this race, a fast memory primitive is needed, neither slow memory controllers nor slow access primitives could achieve a sufficiently high frequency to beat the *rows refresh*.

2.1.3 Targeted accesses

For an attack to be successful, the attacking process has to know how to reach individual *rows* to build up an attack pattern. There are two mapping layers to be crossed: *virtual address* → *physical address*, *physical address* → *memory geometry*. The *virtual* to *physical* address mapping is described in the /**proc**/self/pagemap pseudo-filesystem, but to make *Rowhammering* more difficult, from kernel 4.0 on, only the users with CAP_SYS_ADMIN capability can read *page frame number*s (PFN) from the pagemap interface [16]. and the CAP_SYS_ADMIN capability is equivalent to root permissions [17]. Another way to infer the *virtual* → *physical* mapping is to leverage kernel *Huge Pages* [18] (Super Pages on BSD, Large Pages on Windows). This functionality, if enabled on the kernel, allows userspace processes to allocate pages which have a size of 2048KB or 1024MB depending on the CPU support. Since the addresses in an allocated huge page are contiguous, the process has sufficient knowledge to obtain a relative *row* addressing [2]. Nevertheless *Huge Pages* [18] support is turned off by default in Linux, most Android smartphones and Windows.

Getting trough the *physical address* → *memory geometry* though is a harder task: the mapping function is hardwired inside the memory controller, it is undocumented but it has been reverse engineered for Intel Sandy

Bridge [19] and Ivy Bridge, Haswell and Skylake [20]. Whichever the mapping is, it changes with the amount of installed memory, and some of the mappings reported in previous works prove to be wrong on our test machines with the same Intel processor generation.

Being the *physical address → memory geometry* mapping the *condicio sine qua non* of every *Rowhammer* attack, this thesis work focuses mainly on developing a complete and correct method for inferring the mapping on every system, regardless of previous knowledge about the processor generation.

The final *physical address → memory geometry* mapping is actually the composition of two mapping functions, one applied by the memory controller and the other applied by the *DRAM* module internally. When detecting the mapping with software methods such as in DRAMA's work, the perceived function is the composition of the two.

An interesting approach for reverse engineering only the *DRAM* internal addressing bit permutation is proposed by Matthias Jung in his paper "Row Hammer with Crosshair" [21]. Since disturbance errors in *DRAM* modules are directly proportional to the temperature, applying a temperature gradient on the surface of the module will result in a disturbance error pattern on the data contained in it. In a sequence of steps with different thermal gradients, the actual *DRAM* mapping can be reconstructed.

2.2 Exploiting the vulnerability

This Project Zero blog post (Google internal security team) represents the first use of the *Rowhammer* phenomenon as a security vulnerability [9].

Google's team presents two attacks: one based on *page table entries* (*PTE*s) corruption to obtain a kernel exploit and the other performed on

Google Chrome sandbox, obtaining arbitrary code execution.

This second vulnerability can be concatenated with the first to obtain a kernel exploit directly from code running in Chrome, for example in a malevolent extension.

This Chrome sandbox (NaCl) vulnerability was mitigated by removing **clflush** from the allowed x86 instructions (CVE-2015-0565). On the other hand, as previously stated there are ways other than the **clflush** instruction to trigger a *Rowhammer* attack.

There are other interesting examples of *Rowhammer* vulnerability exploiting, in particular Flip Feng Shui [10] is a cross-vm setting attack. The attack relies on a feature present in many hypervisors, called *Memory Deduplication*. Whenever duplicate pages are detected, even if they belong to different *Virtual Machines*, this feature frees one of them and replaces it with a *Copy on Write* pointer to the other page. As a consequence, a *Rowhammer* attack, corrupting a deduplicated memory page, will corrupt other pages, potentially belonging to victim *Virtual Machines*, without triggering the *Copy on Write* and as a result keeping the two pages merged and corrupted.

The Flip Feng Shui attack aims at obtaining an OpenSSH access to a victim VM, contained in the same hypervisor environment of the attacker VM. The attack is performed by reproducing a page memory in the attacker VM, identical to the one containing the authorized_keys file of the victim VM. A Rowhammer attack to the attacker VM's own duplicated page will corrupt the public SSH keys contained in the victim virtual page. The SSH public keys corrupted this way, will be with high probability made vulnerable to a factorization attack.

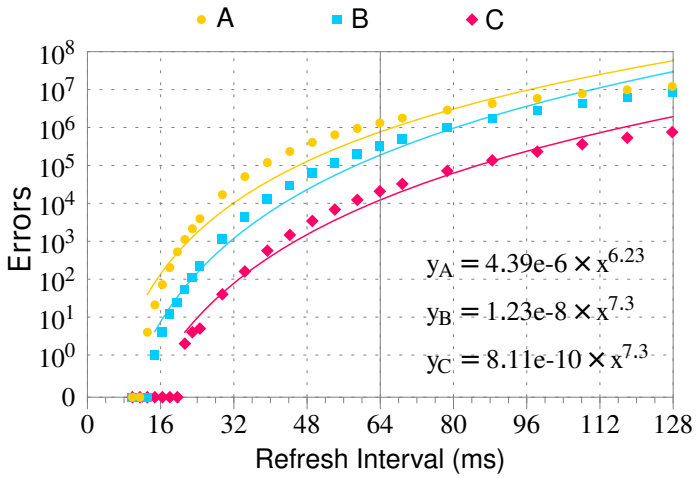


Figure 2.1: Disturbance Errors Number over *DRAM Refresh Interval*. Graph rebuilt from Kim, Yoongu et al. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors" [1].

2.3 Proposed countermeasures

Several countermeasures [22] have been proposed to eliminate the *Rowhammer* phenomenon, resulting in a more or less effective mitigation.

2.3.1 Doubling *DRAM Refresh Rate*

The most common countermeasure against the attack is the halving of the *Row Refresh Period* of *DRAM* modules. As we can see from figure 2.1, beyond causing a considerable power consumption and accessing latency penalties, it just reduces by some orders of magnitude the number of disturbance errors, without eliminating them. During ordinary use, the refresh process is responsible for 30% of the power consumption of each *DRAM* module. In particular for laptops and smartphones where the environment is power-constrained, such countermeasures are not even to be considered. To completely eliminate the phenomenon the *Row Refresh Period* should be dropped below 16ms, but this would imply a huge power and performance penalty.

2.3.2 *B-CATT* and *G-CATT*

Ferdinand Brasser, in his recent research paper [23] introduced two interesting software countermeasures, named B-CATT and G-CATT. The first works by checking the memory regions which are more vulnerable to the *Rowhammer* phenomenon and excluding them at boot time, yet this countermeasure is likely to render systems unusable. Namely, on vulnerable machines, a significant portion of the memory (even more than 90%) may be subject to this phenomenon. Furthermore there is no guarantee that vulnerable memory regions would not change over time, so this countermeasure is likely to be ineffective. G-CATT on the contrary works at runtime by tweaking the Linux memory allocator to partition the available memory into security domain. The countermeasure splits the *banks* in two sets, separated by an empty *row*, and then assigns different processes to different isolated sets. This countermeasure has a couple of weak spots: first, in a standard setup the number of *banks* is limited (in our Skylake test machine we have 16 *banks* per modules and 2 modules) and therefore the number of isolated sets can be exhausted easily. Secondly, the countermeasure is strictly dependend on the memory addressing function of the system, and if that function is incomplete, the whole countermeasure mitigates but does not eliminate the attack surface. Thirdly, G-CATT only leaves a single *row* between the various isolated regions, so it is not guaranteed that a finely tuned *Rowhammer* attack would not cross the boundary.

2.3.3 *Pseudo Target Row Refresh*

Other countermeasures are hardware based and require a replacement of existing *DRAM* modules, among those is the introduction of the *Pseudo Target Row Refresh* (pTRR) command by JEDEC, which is used to selectively refresh *rows* neighbouring to the one we just accessed. This primitive

does not introduce any drawback on either performance or power consumption.

2.3.4 *Probabilistic Adjacent Row Activation*

An interesting countermeasure proposed by Yoongu Kim in his paper [1] consists in refreshing the neighbour *rows* after every memory access, with a non-null probability. Hence, in case of multiple consecutive accesses, the victim *row* would be updated more frequently compared to the normal *Refresh Period*, thus reducing or possibly eliminating the *Rowhammer*-related disturbance errors. This method, if finely tuned on a *DRAM* module's characteristics, can achieve very low performance impact, nonetheless, as any hardware countermeasure, it requires the *DRAM* OEM to introduce it in memory modules manufactured in the future, so it does not protect already deployed systems.

2.3.5 *ECC* modules

As the number of errors can go beyond the two in a single *row*, the use of *ECC* modules is not sufficient to neutralize the *Rowhammer* phenomenon, in fact *ECC* modules are *Single Error Correction Double Error Detection* (*SECDED*). Some manufacturers suggest to use ECC memories as a countermeasure [24], but as we have seen in the past, this countermeasure is ineffective.

2.4 Finding the *virtual* → *geometrical* addresses mapping function

Most modern Operating Systems make use of *memory virtualization*. A fragmented *physical memory space* with several holes, e.g. due to peripheral address mappings, is translated into a contiguous *virtual address space*,

unique for each process. Consequently there is a mapping function that translates *virtual addresses* to *physical addresses*.

Furthermore the function which maps *physical addresses* on *channels*, *ranks*, *banks*, *rows* and *columns* is undocumented for Intel processors.

So how is it possible to find couples of addresses which satisfy the *Same Bank Different Row* condition, necessary for a *Rowhammer* attack?

2.4.1 Random approach

Project Zero's approach [9] is to extract random addresses from a large addresses pool (~1GB). If we are attacking a single *DDR3* module of 4GB size, it will contain 2 *ranks*, each composed by 8 *banks*. Therefore we will have a probability of $\frac{1}{16}$ of obtaining two addresses belonging to the same *bank*. Oppositely, the probability of choosing two addresses belonging to the same *row* is very low, since every *bank* contains a large number of *rows* (2^{15} in the aforementioned *DDR3* module).

This way though, it is only possible to obtain a *Single Rowhammer* attack, because while it is easy to pick two address in the same *bank*, it is very hard to pick three adjacent *rows*.

2.4.2 Huge pages

Another approach is based on having the *huge pages* feature active on the target system. If this is the case, we can allocate pages greater than 4KB, for example 2MB or 1GB according to the processor of the target machine. Allocated pages are mapped on *physically contiguous memory segments*. This allows to identify in a trivial way couples of addresses which satisfy the *SBDR* condition but also to obtain three adjacent *rows* suitable for perfoming the *Double Rowhammer* attack. This feature is present on most Operating Systems, though it is often turned off by default as on Linux and Android AOSP.

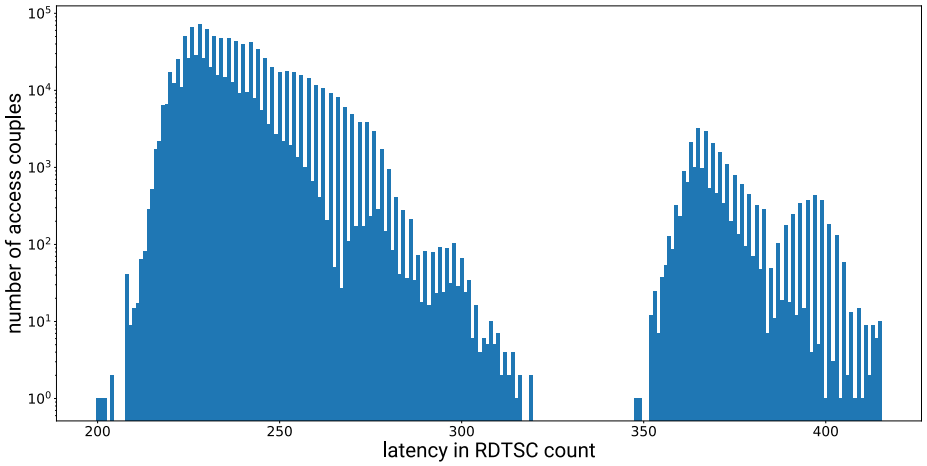


Figure 2.2: Uncached Memory Access Times to Random *DRAM* Addresses

2.4.3 Pagemap

The Linux kernel exposes a list of the mapped pages of each process with the corresponding *physical addresses* in the virtual file /**proc**/self/pagemap [16]. From kernel version 4.0 on, this information is zeroed out unless the process user is root or has the CAP_SYS_ADMIN [17] capability.

2.4.4 Timing

By measuring uncached memory access times to random sampled *DRAM* addresses we obtain a figure that can be coarsely approximated to a bi-modal normal distribution. The figure presents additional noise due to the interference of the *Row Refresh* process with the measurements.

The first mean represents access to addresses belonging to different *DIMM channels*, different *ranks* or *banks*. Conversely, the second gaussian mean with an average value greater than the first one represents addresses which satisfy the *SBDR* condition, together with address couples subject to other kinds of conflicts i.e. *DDR4* Bank Group Conflics. *SBDR* accesses are slower than the first ones due to the time necessary to close the first *row* and open the second *row*.

It is possible to identify addresses belonging to this second set, choosing them by their longer access times. This allows us to *reverse engineer* the mapping function of *physical addresses* to the geometrical characteristics of the *DRAM* modules, as explained in the DRAMA paper [20].

Chapter 3

Assessing the reliability of known *Rowhammering* methods

The road towards achieving *Rowhammer* on a machine is a long one. As a first step, Google's *extended-text* [25] has been run on the the target machines for ten hours without obtaining results from any of the available machines.

The test performs a *Single Rowhammer* attack by picking random addresses from a large mapped address space and performs the **CLFLUSH**-based *Rowhammer* loop. If by any chance the two picked addresses belong to the same *bank*, a *Rowhammer* fault could be triggered.

The outcome of a *Single Rowhammer* attack depends solely on the weakness of the target modules. As Yoongu Kim showed in his paper [1], the majority of modules produced after 2013 are vulnerable to *Rowhammer*. Even so, there are some stronger modules which are immune to *Single Rowhammer* attacks, in this case our only chance of performing an attack

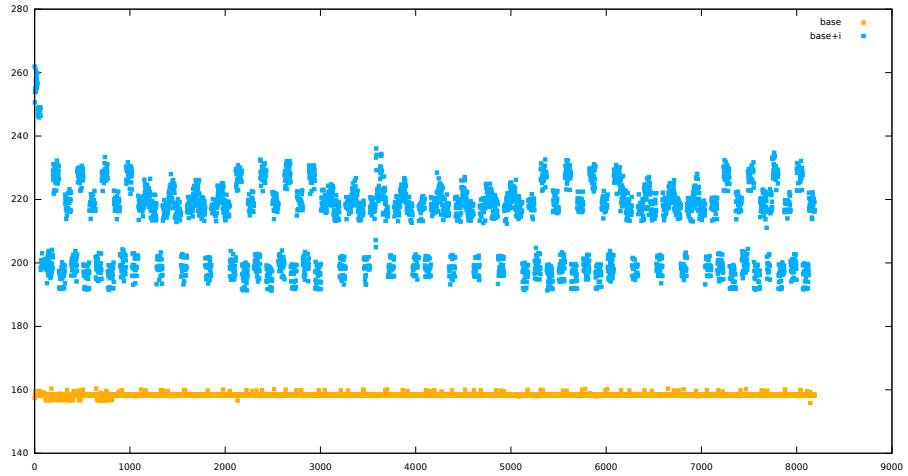


Figure 3.1: First Latency Test Output on Sandy Bridge Microarchitecture

is to build a *Double Rowhammer*.

A fundamental prerequisite of a *Double Rowhammer* attack is knowledge of *virtual addresses* mapping on *memory geometry*. From now on we will focus on finding a technique to reverse engineer the aforementioned mapping.

3.1 Memtest86 timings measurements

A first concern about collecting data in order to find the **mapping function** is to avoid the additional remapping done by the Operating Systems. To this aim a latency measurement tool has been built inside the Memtest86 utility, usually run directly as a bootloader payload without any Operating System in-between.

A first test measures the consecutive access time of uncached reads to two memory addresses, the first fixed and the second swept over all the available addressing space (figure 3.1). The first address access times are colored in orange, while the second address access times are colored in teal. Every value is the average of 200 samples.

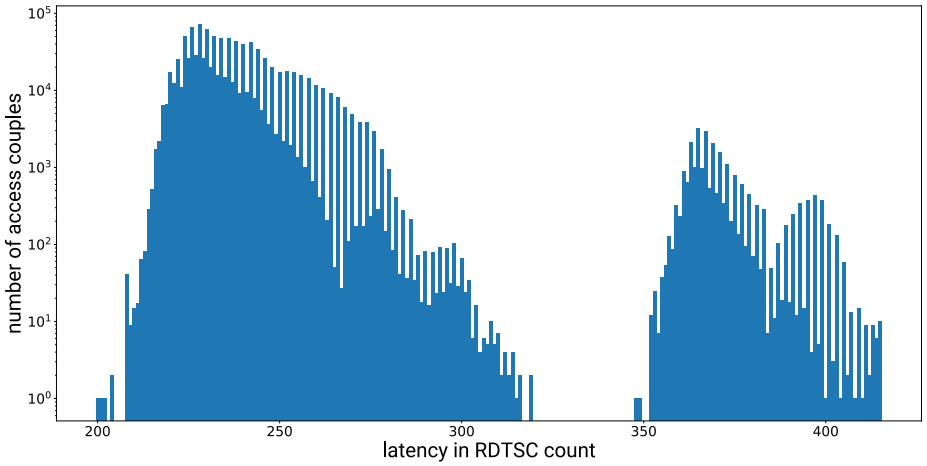


Figure 3.2: Uncached Memory Access Latencies Histogram

The graph shows clearly that while the first access latency is constant, the second access latency depends on the relative position of the two addresses. The second address latency, divides into two latency classes. The slower accesses are probably caused by the *SBDR* conflicts.

Nonetheless, this kind of data is too raw to be directly interpreted and obtain a mapping function.

3.2 Memory mapping model

Some previous work has been carried out on reverse engineering the mapping between physical addresses and memory location, in particular the DRAMA [20] framework embeds an example code to detect mapping functions in any machine.

Their code is available online in a public GitHub repository [26]. The code works on our Sandy Bridge setup, but it is extremely fragile and unable to produce any result on our Ivy Bridge test machine.

This code generates a histogram of uncached memory access times, obtaining a graph similar to the one shown in figure 3.2.

As we can see, the access latencies can be divided in two normal dis-

tributions. The addresses belonging to the second gaussian satisfy the SDBR condition and therefore belong to the same *bank* but to different *rows*.

However, this is a snippet of the DRAMA code which separates the two gaussians:

```
// find separation
        int empty = 0, found = 0;
        for (int i = max; i >= min; i--) {
            if (hist[i] <= 1)
                empty++;
            else
                empty = 0;
            if (empty >= 5) {
                found = i + empty;
                break;
            }
        }
```

This implementation leads to faulty memory mapping function identification on some machines where the two gaussians are very close to each other.

Once low-latency addresses have been separated from high-latency ones, the code authors try to partition the high-latency addresses into sets, whose members all satisfy the *SBDR* condition. In particular, their method yields a set of functions that show which bits of the addressing function are XORed to one another.

The mapping function depends on the CPU generation, on the pattern to which the *DRAM* modules are connected (how many *channels*, how many *DIMM*s) and on the particular memory modules.

For the machines from which the DRAMA code gave us a plausible re-

sult, we set up a test sandbox, with the purpose of assigning the corresponding set and *row* numbers to each address. This way, if the models prove correct, mounting a *Double Rowhammer* attack will become trivial.

3.3 First *bit flip*s

To test if the modules were vulnerable to *Single Rowhammer* attacks, and validate the mapping models obtained through the DRAMA [20] code, the Project Zero extended-test [25] was run on the testing machines.

Only two *SO-DIMM* 4GB modules, manufactured by Samsung were found vulnerable to this kind of attack.

The extended test codes were instrumented, thus yielding for each flip the *row* and set numbers according to our model for that test machine. These are the results of the tests:

- Correct sets, wrong rows
  - Attacking: 0x8a948000 belongs to set 28 and row 8869
  - Attacking: 0x74e1d000 belongs to set 28 and row 7480
  - Victim: 0x74e3fe08 belongs to set 28 and row 7480
- Wrong sets, correct rows
  - Attacking: 0x5b2c1000 belongs to set 12 and row 5835
  - Attacking: 0x78285000 belongs to set 12 and row 7690
  - Victim: 0x5b2a66f0 belongs to set 13 and row 5834
- Wrong sets correct rows
  - Attacking: 0x79e3b000 belongs to set 24 and row 7800
  - Attacking: 0x72c7f000 belongs to set 24 and row 7345
  - Victim: 0x79e5d960 belongs to set 25 and row 7801
- Correct sets and rows

  - Attacking: 0x9391b000 belongs to set 10 and row 9444
  - Attacking: 0x6d31a000 belongs to set 10 and row 6988
  - Victim: 0x9397dfb0 belongs to set 10 and row 9445
- Correct sets and rows
  - Attacking: 0x8502d000 belongs to set 14 and row 8512
  - Attacking: 0x889d3000 belongs to set 14 and row 8743
  - Victim: 0x8504be20 belongs to set 14 and row 8513

As it can be seen from the results, there is some misclassification on both sets and *row*s; this can be related to inaccuracies in the mapping model adopted. As one can imagine, such kind of inaccuracies undermine the possibility of performing a *Double Rowhammer* attack.

It follows that the model has to be corrected to implement a *Double Rowhamer* attack in a deterministic way on our test machine. On some of our test machines, the *Single Rowhammer* has proven to be ineffective, although they might still be vulnerable to the more powerful *Double Rowhammer* attack.

Finally, obtaining a mapping model for a target machine is fundamental to implement effective software countermeasures to all the *Rowhammer* attacks. As a matter of fact, for each sensitive memory address it should be enough to avoid *hammering* on the two adjacent memory *row*s. This, in the end, provides a chance to protect sensitive data structures such as Kernel *page table entries* (*PTE*s), file permissions data structures and asymmetric cryptographic keys, with a minimum space consumption penalty and no performance penalty whatsoever.

3.4 Successful attack on Sandy Bridge

By using Google's code as reference we were able to craft our own *Double Rowhammer* attack. And by exploiting the known mapping function for

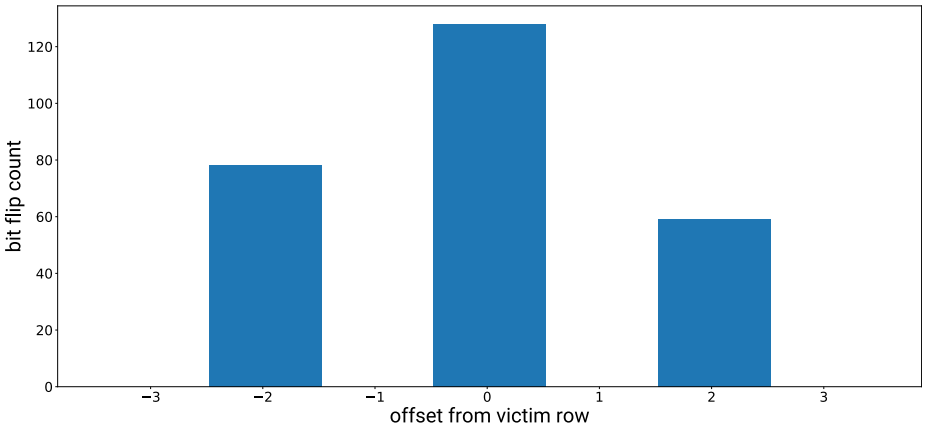


Figure 3.3: Flip Offset Histogram on Sandy Bridge

Sandy Bridge, as shown in a blog post by Mark Seaborn [19], we were able to perform double rowhammer on Sandy Bridge. Our attack yields a large number of flips, few seconds after it is launched.

As a result of the attack we computed a *flip offset histogram*, shown in figure 3.3, which shows for a single pair of *attacking rows*, the distribution of disturbance errors. The *flip offset histogram* was obtained by performing a *double Rowhammer* attack, on a single *victim row*, and then observing the count of errors of the neighbouring rows. The single *victim row* is represented as the *row* with offset 0, the two *attacking rows* are positioned at offsets 1 and -1. Due to being continuously refreshed to perform the hammering, the two *attacking rows* will never show disturbance errors. This observation is confirmed by the graph and can be used to build a useful sanity check when performing *Rowhammer* tests. As we can see from the *flip offset histogram*, in our setup, disturbance errors show up in rows which are at most 2 *rows* away from the *victim row*.

Chapter 4

Characterizing physical-to-geometrical mappings

Since available methods for performing the *Rowhammer* attack have proven to be ineffective on both our Ivy Bridge and Skylake test machines, in particular due to a low reliability of available mapping detection algorithms.

In this chapter we pose the necessary theoretical and practical foundations to build a reliable mapping detection method. We start by introducing our Bit-Diff test, which is useful to highlight the differences between the various addressing bits.

4.1 The Bit-Diff test

In our test we allocate a physically contiguous 4GB wide segment of memory. For the purpose of having a large segment of contiguous memory we activate and use huge pages [18]. By allocating hugepages with a kernel

CMDLINE parameter we obtain a non-fragmented memory segment of custom size. The test performs the following steps:

Input: Physically contiguous 4GB segment of memory
Output: For each addressing bit b, get the timing measure of the second access, when accessing consecutively a random address A_{v1} and the same address with the b bit inverted.
foreach *virtual address* A_{v1} *sequentially taken with a step of 4KB* **do**
  foreach *physical address bit* b **do**
    Get physical address A_{p1} relative to A_{v1}
    Flip b^{th} bit in A_{p1} and get A_{p2}
    Get virtual address A_{v2} relative to A_{p2}
    Read sequentially A_{v1} and A_{v2}
  end
end

Algorithm 1: Bit-Diff

A_{v2} access times correspond to different relative physical locations for the two addresses. The following tests are performed with N_SAMPLES $= 100$.

The function get_bounded_random_addr picks a safe address inside the mapped space, that is an address which, even if incremented by $2^{bitIndex}$ or decremented by $2^{bitIndex}$ will still be in the mapped space.

In figure 4.1 we can see the A_{v2} times as the changed bit varies, for the Sandy Bridge architecture.

4.2 Reliable timing measurements

Measuring uncached accesses in a reliable way is not trivial on Intel platforms because Intel CPUs perform out-of-order execution [27]. Furthermore the timer functions may cause access to RAM and thus interpose memory accesses between our two memory accesses.

A solution to both problems is to use the **rdtscp** instruction, which yields the value of the *Time Stamp Counter* [28] and being a serializing instruction, it cannot be reordered.

However, Intel CPUs older than Haswell, a similar result can be achieved by combining rdtsc non-serializing timer with the cpuid instruction [27]. This instruction fills the CPU registers with information about the processor and, as a side-effect, forces the serialization of previous instructions; consequently, no instruction can be reordered past a cpuid instruction and before the execution, the write buffers are emptied.

The following is a reliable and accurate latency measurement code:

```
sched_yield();
*f;
asm volatile("clflush␣(%0)" : : "r" (f) : "memory");
asm volatile("clflush␣(%0)" : : "r" (s) : "memory");
asm volatile("mfence" : : : "memory");
uint64_t start = rdtsc();
*f;
*s;
uint64_t end = rdtsc2();
sched_yield();
return end - start;
```

If not otherwise specified, all the following tests and algorithms will adopt this latency measuring procedure.

4.3 Mapping functions addressing fields

All the *DDR* memory controllers have a characteristic *addressing function*, which maps physical addresses to a precise geometrical position inside a *DRAM* device. Every addressing function has as image several groups of bits, each one addressing a specific geometrical dimension of all the *DRAM* modules. We call these group of bits *addressing groups* and we assign for each of them a fixed nomenclature:

- K independent *channels* of memory
- L *ranks* per *channel*
- B *banks* per *rank*
- R *rows* per *bank*
- C *columns* per *row*
- V bytes per *column*

For convenience, the last two can be replaced by:

- N cachelines per *row*
- Z bytes per cacheline

Where N can be computed as $N = \frac{C \cdot V}{Z}$.

The size of each *addressing group* can be read on the datasheet of the *DIMM* modules or the datasheet of the storage chip on the surface of the module or even in the *Serial Presence Detect* data, which is a serial storage chip on the surface of each module, used for automatic configuration [29]. The dimensions retrieved from the datasheet should satisfy the following condition, for each machine under test:

- Total installed memory size is $K \cdot L \cdot B \cdot R \cdot C \cdot V$
- Alternatively $K \cdot L \cdot B \cdot R \cdot N \cdot Z$

For every uppercase parameter, the lowercase letter denotes the exponent of the corresponding power of two, which represents the required addressing bits.

Summing up what is expressed in David Wang's PhD thesis [4], in table 4.1 we see some examples of mapping functions.

Table 4.1: Most common mapping functions as shown in David Wang's PhD thesis [4]

Description	Function
Trivial memory mapping is k:l:b:r:c:v, or equivalently	k:l:b:r:n:z
Baseline memory mapping for open-*row* policy	r:l:b:n:k:z
Baseline memory mapping for closed-*row* policy	r:n:l:b:k:z
Extensible memory mapping for open-*row* policy	k:l:r:b:n:z
Extensible memory mapping for open-*row* policy	k:l:r:n:b:z

Bank address aliasing (see section 5.3.5, "Bank Address Aliasing (stride collision)" of David Wang's thesis [4]) is a phenomenon where what the process sees as interleaved accesses over two vectors, maps into access on the same *bank* and different *rows*. This causes the target *rows* to be opened and closed for every memory read. This negative effect can be alleviated by XORing the *row* number over the *rank* and *bank* number. This way consecutive *DRAM row*s become mapped to different *banks*, so that interleaved accesses to two arrays will never end up in the same *bank*.

In the following subsections we will provide the geometrical characteristics of each platform under test, together with the results of the Bit-Diff test.

4.3.1 Intel Sandy Bridge microarchitecture mapping

We first apply our Bit-Diff test on our Intel Sandy Bridge test machine; this machine has well documented mapping function, which has already been confirmed by our tests.

As we can see from the graph in figure 4.1, the first 7 bits have the lowest latency, this is exactly what we expected since reading two addresses in the same cacheline will result in no *row* being opened or closed. All these kinds of reads are directly performed on the *row buffers* and thus are

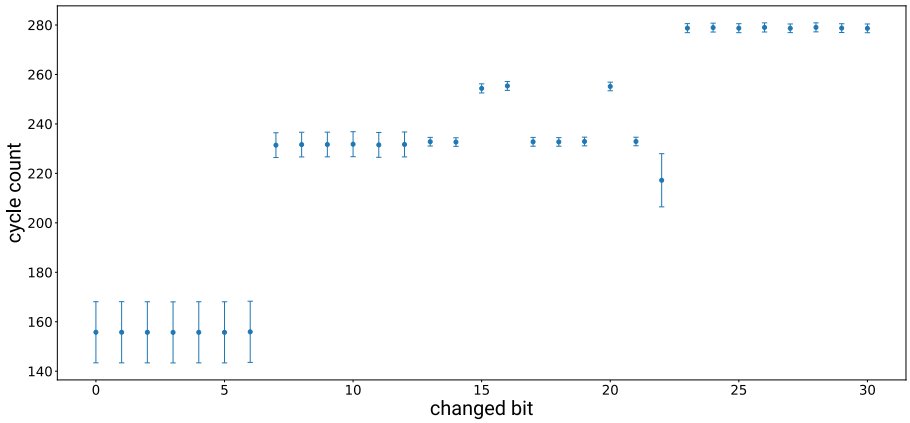


Figure 4.1: Sandy Bridge Bit-Diff Test Result

very fast. Conversely, bits from the 23rd on are always slow because they always cause one row to be closed and another to be opened. Furthermore, bits 14, 15 and 16 have the same latency profile of bits 18, 19, 20. Therefore we can confirm that those bits are XORed together, as shown in Mark Searbon's work [19] and in the DRAMA paper [20].

Every module has a size of 4GB, 2 *ranks*, 8 *banks*.

$$\text{Banks} \cdot mboxRows \cdot mboxColumn \cdot mboxBits : 8 \cdot 15 \cdot 10 \cdot 64$$

Table 4.2: Mapping parameters characterization for Sandy Bridge architecture

Parameter	Single Channel	Dual Channel
k	0	1
l	1	1
b	3	3
r	15	15
n	7	7
z	6	6
Total size	32	33

Parameter	Single Channel	Dual Channel
Mapping	r:l:b:n:k:z	r:l:b:n:k:z

The mapping found by Mark Searborn [19] is r:l:b:n:k:z, where the least significant 3 *row* bits are XORed over b.

In the performed test, the various mapping regions have different values and we clearly see that the *channel* selection bit (denoted as k, in the sixth position) disappears when we perform the test with a single *channel* memory setup.

To sum up bit fields characteristics:

- z and n bits have high variance and high value
- k, b and l bit have low variance and low value
- r values have low variance and mixed value

4.3.2 Intel Ivy Bridge microarchitecture mapping

As a first step to obtain the unknown memory mapping function for the Intel Ivy Bridge microarchitecture, we apply the Bit-Diff test to our Ivy Bridge machine, the result can be seen in figure 4.2. Here the observation about the cacheline bits does not hold anymore, probably due to a noisy sampling of the data. But still we can confirm that the *row* addressing bits are the most significant bits. But since we only see 10 of those 15 row addressing bits, 5 of them are XORed with other addressing bits, which bring their latency down. This happens because if we consider a *row* addressing bit which is XORed with a *bank* addressing bit, changing the *row* bit will also change the *bank*, thus violating the SBDR condition and causing low latencies.

Every module has a size of 4GB, 2 *ranks*, 8 *banks*.

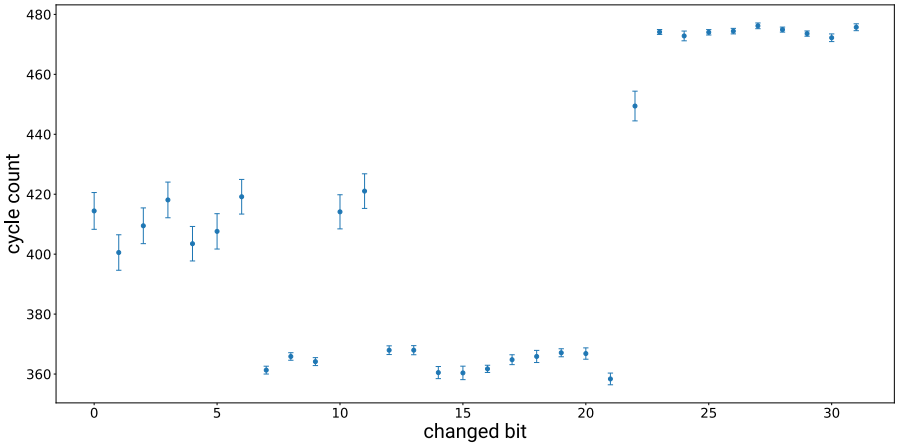


Figure 4.2: Ivy Bridge Bit-Diff Test in Dual Channel configuration

$$\text{Banks} \cdot mboxRows \cdot mboxColumn \cdot mboxBits : 8 \cdot 15 \cdot 10 \cdot 64$$

Table 4.3: Mapping parameters characterization for Ivy Bridge architecture

Parameter	Single Channel	Dual Channel	Single Ch 2x	Dual Ch 2x
k	0	1	0	1
l	1	1	2	2
b	3	3	3	3
r	15	15	15	15
n	7	7	7	7
z	6	6	6	6
Total size	32	33	33	34
Mapping	r:b(1):l:b(2):n:k:z	r :?:?:?:?: z	r :?:?:?:?: z	r :?:?:?:?: z

4.3.3 Intel Skylake microarchitecture mapping

Here in figure 4.3 we see the application of our Bit-Diff test to our Intel Skylake test machine. Here we have a setting very similar to the Ivy Bridge one, plus we see two bits (21, 22) which have a latency similar to row selection bits bit not so high. These bits, could be related with the *Bank Groups* addressing, which is necessary on DDR4 modules [30].

Every module has 8GB size, 1 *rank*, 2 *bank groups*, 8 *banks*.

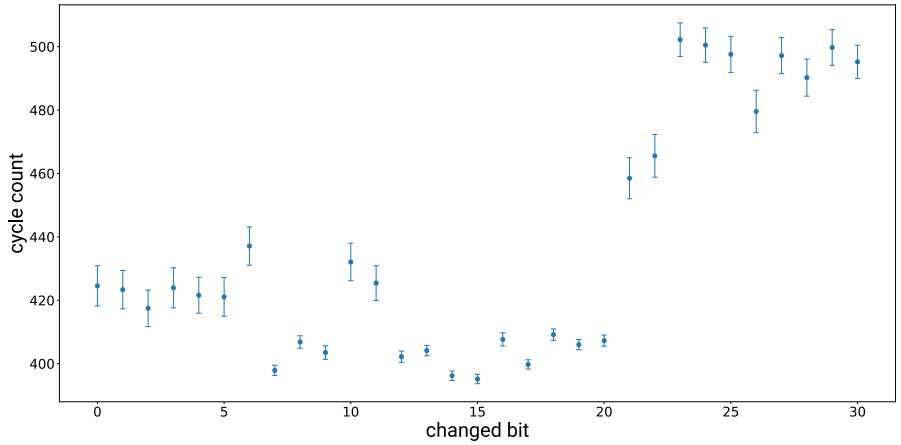


Figure 4.3: Skylake Bit-Diff Test in Dual Channel configuration

[30]. Consequently in every module we have 4 groups of 4 *banks* each.

Table 4.4: Mapping parameters characterization for Skylake architecture

Parameter	Single Channel	Dual Channel	Single Ch 2x	Dual Ch 2x
k	0	1	0	1
l	0	0	1	1
b	4	4	4	4
r	16	16	16	16
n	7	7	7	7
z	6	6	6	6
Total size	33	34	34	35
Mapping	?:?:?:?:?: z	?:?:?:?:?: z	?:?:?:?:?: z	?:?:?:?:?: z

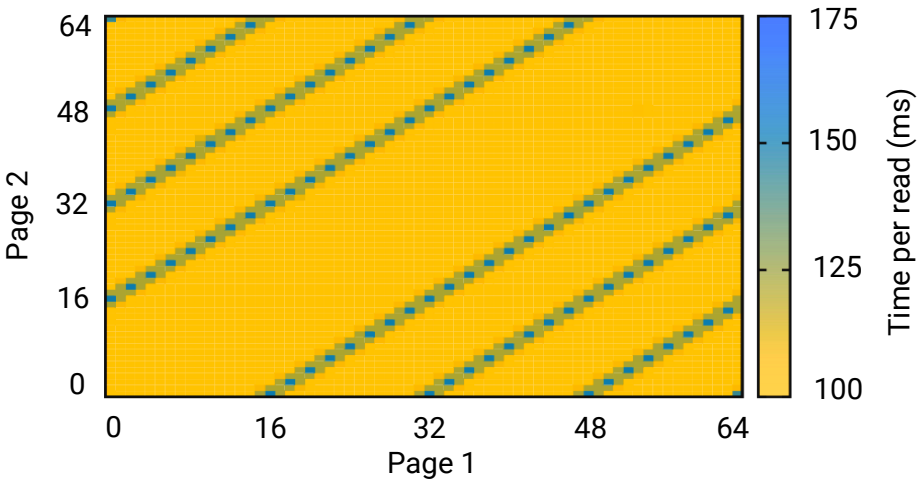


Figure 4.4: 2D Memory Latency Heatmap on a LG Nexus 5. Graph rebuilt from Van Der Veen "Drammer: Deterministic Rowhammer attacks on mobile platforms" [2].

4.4 Page Heatmap test

The Drammer [2] paper describes an interesting technique for finding the *row length*, term for which they identify the number of consecutive 4K pages which are needed to go back into the same *bank*, on the next or previous *row*.

It suggests to build bidimensional heatmap with the access times of a group of sequential pages, with increasing offsets from a base address.

In their results some dark diagonal lines are clearly visible, denoting a fixed *row length* as visible in figure 4.4.

The results for the three architectures under examination can be seen in figure 4.7. As we can see, Ivy Bridge and Skylake heatmaps are symmetrical with respect to their antidiagonal, thus we can confirm that mapping functions are deterministic, and the function that determines the latency of a couple of addresses is symmetric. In neither of the three heatmaps we can identify the row length since there are no secondary antidiagonals at fixed offsets.

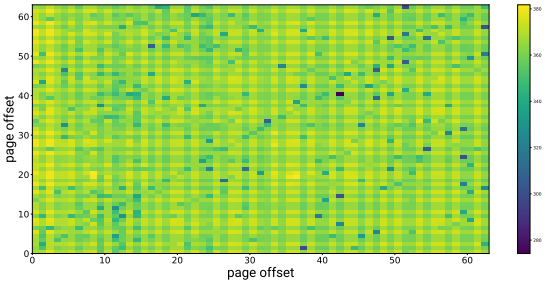


Figure 4.5: Sandy Bridge Heatmap

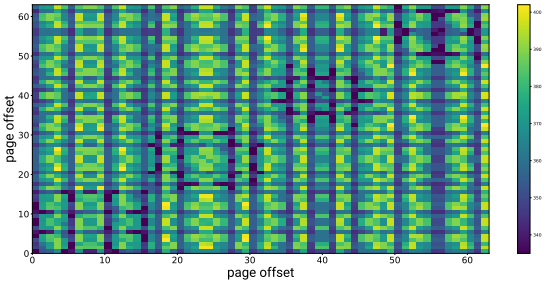


Figure 4.6: Ivy Bridge Heatmap

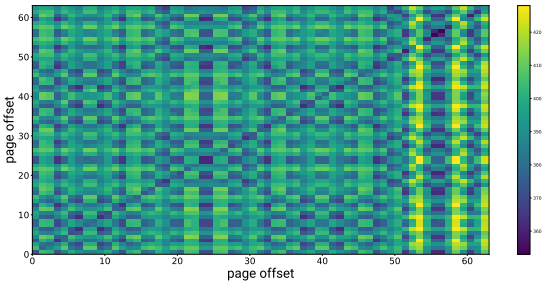


Figure 4.7: Skylake Heatmap

Chapter 5

Systematically deriving geometrical mapping

Since the methods shown in previous literature [20] for finding the mapping functions have proven to be inaccurate and fragile, we started from their work to build a more reliable and resilient algorithm, to be used as a basis for the development of new and effective software countermeasures.

In particular, we were unable to obtain any *bit-flip* on our Skylake test platform using the mapping functions provided in the DRAMA [20] paper. Therefore, as a further step, it is necessary to build a reliable method for finding mapping functions to be used in building attacks and countermeasures using the *Rowhammer* technique.

We start by obtaining the exact geometrical characteristics of every *DRAM* chip involved: *row* size, *bank* size *rank* number and *bank* groups. We can obtain this information by looking into the technical datasheets of the *DRAM* chips used in the test machines.

5.1 Scan the memory and verify geometrical characteristics

As a second step, we have to fingerprint the installed memory to retrieve and confirm some of its geometrical characteristics. Memory latency (access time) is influenced mainly by the location of previous accesses. In our study we consider only uncached accesses, and we obtain them by flushing the caches via the **CLFLUSH** instruction.

We try to allocate via huge pages the largest amount of memory possible, then we perform couples of memory accesses, measuring only the second access. The first address stays the same (first page of the entire allocated block), while the second ranges over all the allocated memory, with a span of 1 virtual page (4096B).

Then we collect the uncached accesses latency histogram of each machine under test and we try to visually separate high-latency pairs to low-latency pairs. The number of high-latency pairs should be in the same magnitude order of the number of *SBDR* derived from the geometrical characteristics. The theoretical number of *SBDR* pages in this setup is given by the formula:

$$|SDBR| = (R-1) \cdot \frac{V \cdot C}{\text{page size}}$$

Every uppercase letter represents the number of elements counted by the lowercase letter bits.

5.2 Deriving *row* addressing bits

The number of *row* addressing bits is described in the datasheets of the *DRAM* chips, we can find those bits in the set of all the bits that change when comparing physical addresses of *SBDR* couples.

We can do so by computing the or-collapse of the XORing of each physical address couple for each bin in the histogram. All the bits involved with addressing of *banks* should be set to 0 in each bin containing only *SBDR* pairs.

5.3 Bins clustering

In order to capture all the different sets we have to randomly sample both the first and the second address.

We have collected all the bins of the latency histogram, taking particular care in using a reliable measuring primitive, such as the one shown in the previous chapter.

In our setup we implemented the measuring function as a loop which captures three access times to the same pair of addresses and yields the result only when the three accesses are reasonably similar. In our case we consider acceptable sets of three measurements which are equal until the 4th least significant bit.

We have to determine a latency threshold for *SBDR* pairs. First we group up our bins into clusters i.e. contiguous sequences of non-empty bins. Then we filter the clusters by keeping only the ones with a bin width greater than an adjustable threshold.

Finally we can then estimate the approximate number of *SBDR* pairs we should have found in our random sampling. For a random first address, the number of second addresses which will end up in a *SBDR* pair is fixed and depends only on the memory geometry. This probability, that we name $SBDRratio$, is defined as follows:

$$\text{SBDR ratio} = \frac{\text{\#SBDR bytes}}{\text{mem size}} = \frac{N \cdot Z \cdot (R-1)}{N \cdot Z \cdot R \cdot B \cdot L \cdot K} \approx \frac{1}{\text{banks}}$$

Once we have partitioned the sets into clusters and obtained the num-

ber of *SBDR* pairs, we start from the highest latencies and begin adding up clusters until we reach the required number of *SBDR* pairs.

With this technique we have obtained a coarse subdivision into *SBDR* and non-*SBDR* bins.

5.4 Splitting bins into sets

Our *SBDR* cluster will contain pairs from every *bank* in the installed *DRAM* modules, but we can partition those addresses into sets corresponding to different *bank*s. If we take two couples of *SBDR* addresses, they may or may not be in the same *bank*. If they are, their cross-latency will be higher than the *SBDR* threshold, otherwise it will be lower.

We propose a new algorithm for partitioning the *SBDR* pairs into sets (algorithm 2).

5.5 Identifying the sets

In the previous steps we were able to discriminate between slow and fast access times. Using this difference we split our bins in sets. Addresses belonging to the same set should always yield slow access times, when accessed consecutively.

Some of the sets found in this way should contain mostly address pairs satisfying the SBDR condition. Once these sets have been isolated, we can leverage this information to obtain the mapping functions which best fit those

5.6 Bruteforcing functions

Since we cannot be sure that our data is free of outliers, we adopt a rating approach instead of an exclusion one. We test all the possible functions

```
Input: Histogram bins which have been classified as SBDR
Output: Subdivision of SBDR bins into high-latency sets
foreach bin in sbdr_bins do
    foreach pair in bin do
        found_set = false
        # Test the pair against the members of all the new sets
        foreach set in sets do
            foreach s_pair in set do
                # If it achieves high-latency, put the pair in that set
                if measure(pair.first, s_pair.first) > threshold or
                measure(pair.first, s_pair.second) > threshold or
                measure(pair.second, s_pair.first) > threshold or
                measure(pair.second, s_pair.second) > threshold then
                    found_set = true;
                    set.insert(bin)
                    break
                end
            end
            if found_set then
                break
            end
        end
        # If no set exists or no set achieves high latency,
        # create a new one
        if not found_set then
            sets.insert(get_new_set())
            sets.back().insert(bin)
        end
    end
end
```

Algorithm 2: SBDR Sets Subdivision

on all the sets and we assign them a score according to how well they perform on each set. At the end of the process we select only the best functions and perform a deduplication step to remove functions which are the composition of others. The whole procedure description follows (algorithm 3). The algorithm has to be executed separately for the two kinds of sets. We represent efficiently each function as a bitmask, where each bit is set if the corresponding input is to be XORed, or not set if otherwise. In this way we can easily enumerate all the functions by counting from 0 to 2^a where a are the available addressing bits. This allows us to compute the result of the application of each function in two instructions: a

popcount and a modulo (%).

Input: Sets of physical addresses which produce high access times

Output: Linear functions which perform best on all sets

```
foreach set in sets do
    for counter in range(0, last_function) do
        performance = 0;
        mask = counter ≪ sizebit;
        foreach pair in set.pairs do
            # We efficiently compute the result of each function
            # applied to each address by counting the parity of ones
            # of the AND of function mask and address
            first_result = popcount(pair.first & mask) % 2;
            second_result = popcount(pair.second & mask) % 2;
            # If the two parity values are the same, the function
            # is correctly classifying the current pair of addresses
            if first_result == second_result then
                performance += 1;
            end
            setranking.append({mask, performance});
        end
    end
    points = 101;
    # Convert fitness to points to be independent from set size
    for top 100 functions in setranking do
        # Lower the points only when fitness gets lower
        if function.fitness < savedfitness then
            points −= 1;
            savedfitness = function.fitness;
        end
        globalranking.append({mask, points});
    end
end
return globalranking;
```

Algorithm 3: RankFunctions

5.7 Functions Deduplication

Since our test highlights constant bits e.g. *bank* selection, *bank group* selection and *channel* selection bits, we cannot distinguish between functions and their compositions. So we employ a simple greedy procedure (algorithm 4) to select an equivalent smallest subset of all the functions returned by the bruteforcing algorithm. We perform this step to write our functions in a more compact way and to compare our results with other papers such as DRAMA [20].

Input: Top scoring mapping functions

Output: One of the minimal equivalent sets of functions

```
# Sort functions by Hamming weight
sort(top_functions, hamming_compare);
# Compute all the bits set by at least one function
or_collapse = fold(top_functions, binary_or);
sel_mask = 0;
sel_popcount = 0;
foreach function in top_functions do
    new_mask = sel_mask | function.mask;
    new_popcount = popcount(new_mask);
    # If adding this function covers one more bit, add it
    if new_popcount > sel_popcount then
        sel_mask = new_mask;
        sel_popcount = new_popcount;
        sel_functions.append(function);
    end
end
return sel_functions;
```

Algorithm 4: Functions Deduplication

Chapter 6

Experimental validation

In this chapter we'll show the results of the application of our detection method, to our test machines.

6.1 Intel Sandy Bridge Geometrical Characteristics

Our method started by retrieving the geometrical characteristics of the *DRAM* chips involved. Our Skylake machine has two Kingston KHX2133C14/8G *DDR4* modules installed. Those modules use Micron *DRAM* chips internally, whose datasheet is available online [3].

Secondly, we verified the obtained geometrical characteristics by trying to count the SBDR addresses.

We first performed our measurements on the well-known Intel Sandy Bridge microarchitecture, the result is visible is figure 6.1:

Our sandybridge machine has two Hynix HMT325S6CFR8C-H9 modules, in dual-*channel* setup. The modules have the following addressing characteristics:

- 1 *channel* bits (k)

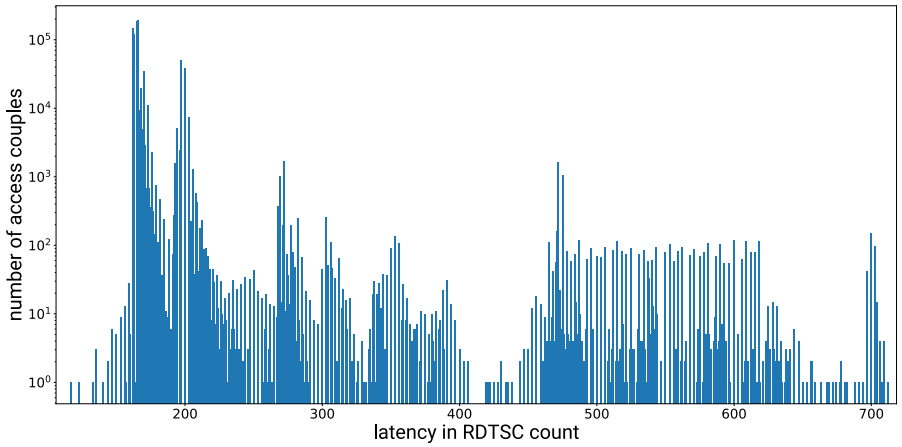


Figure 6.1: Sandy Bridge Latency Histogram for Uncached Accesses

- 0 *rank* bits (l)
- 3 *bank* bits (b)
- 15 *row* bits (r)
- 10 *column* bits (c)
- 3 byte bits (hidden, span of each burst) (v)

Its *row* size is thus 8192B, so each *row* fits two 4096B virtual pages.

In our setup, $|SBDR| = 65534$. So we can highlight on the graph the probable threshold of $SBDR$ latencies. The pages highlighted in red in figure 6.2 sum up to 62809 pages, which is pretty close to the actual number of pages. In addition to this, our test has been run in just 3.25GB of the 4GB of *DRAM*.

6.2 Intel Skylake Geometrical Characteristics

Our skylake machine is equipped with two Kingston KHX2133C14/8G modules, in a dual *channel* setup, each one with 8 *DRAM* chips. Since the bus width is 64 bits, each chip is X8. The technical datasheet [3] reports the following measures:

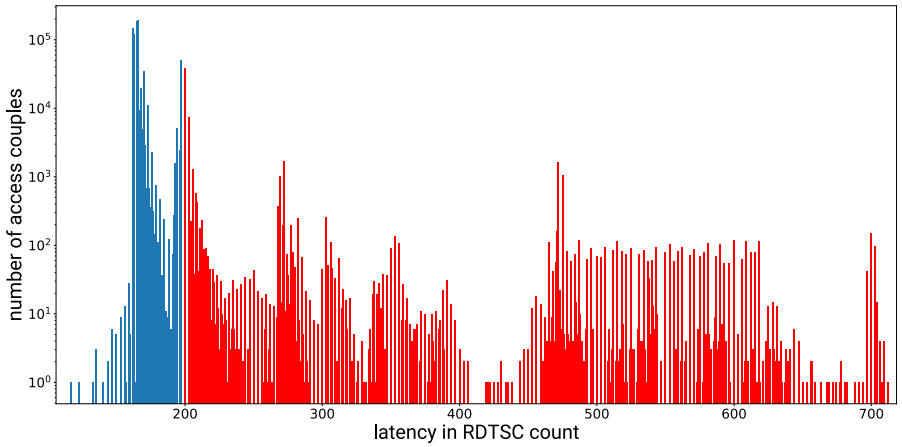


Figure 6.2: Sandybridge Latency Histogram with *SBDR* Latencies Highlighted in Red

- single *rank*
- 16 internal *bank*s
- 4 *bank* groups
- 4 *bank*s per *bank* group
- 16 bit *row* addressing
- 10 bit *column* addressing
- 128x64 byte selection in *row*
- 1KB page size

Which can also be seen in the functional diagram of the *DRAM* modules (figure 6.3).

An addressing scheme will be composed of the following bits:

- 1 bit *channel* selection (k)
- 0 bit *rank* in *channel* (r)
- 2 bit *bank* group in *rank* (g)
- 2 bit *bank* in *bank* group (b)
- 16 bit *row* in *bank* (r)
- 10 bit *column* in *row* (c)
- 3 bit byte in *column* (hidden when querying the *DRAM*) (v)

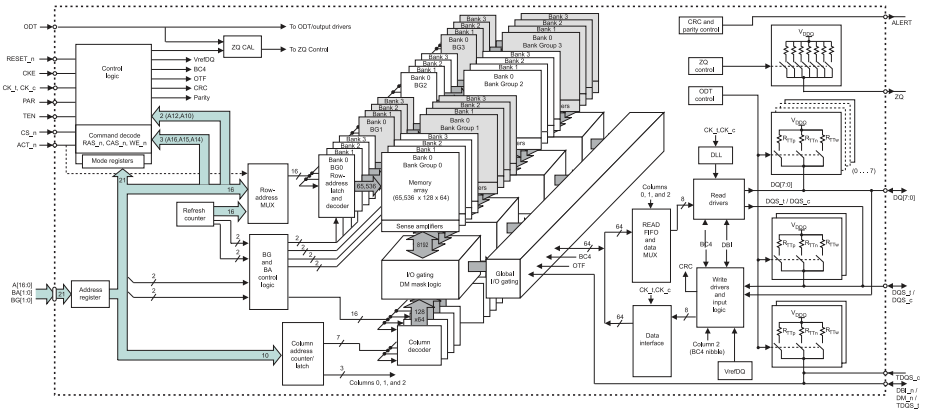


Figure 6.3: Functional Diagram of Kingston KHX2133C14/8G Modules. As shown in the chip datasheet [3].

So a total of 31 bits will identify a sequence of 8 Bytes inside the *DRAM* modules. Since DDR memory hides latency by transferring bursts of 8 Bytes at a time, every memory access will yield 64 bits. The 3 lsb will be used to identify the correct byte, as a result, these 3 bits are never sent to the memory modules.

In short, each *row* is 8192 Bytes long, each *row* contains 2 pages. Each *bank* contains 65536 *rows*. In our setup the theoretical number of SDBR pages is 131070.

We allocate the largest amount of memory possible by using huge pages, in our setting this corresponds to 15GB over 16GB of total memory. On that allocated memory we perform couples of uncached memory accesses. Figure 6.4 represents the latency histogram on skylake, where the candidate *SBDR* latencies are highlighted in red.

6.3 Deriving *Row* Addressing Bits on Intel Skylake

As found by the datasheets there are 16 *row* addressing bits. Therefore we compute the or-collapse of the XORing of each physical address couple

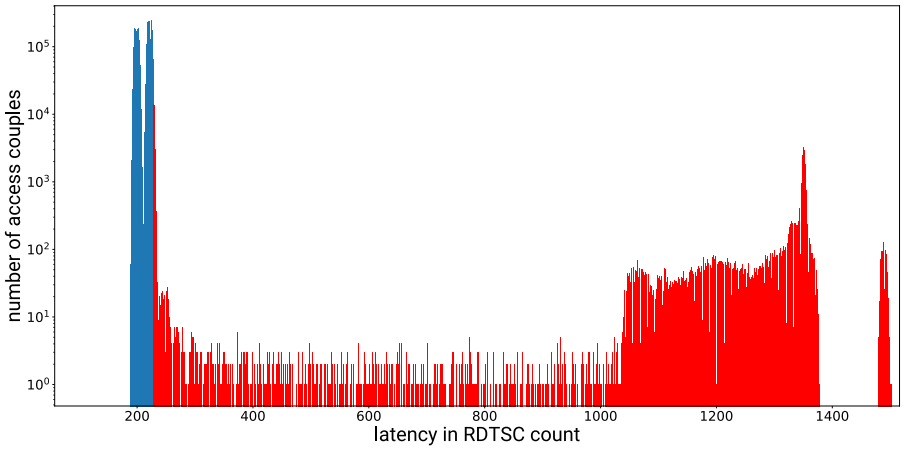


Figure 6.4: Skylake Uncached Accesses Latency Histogram with *SBDR* Latencies Highlighted in Red

for each bin in the histogram, to find bits that change in *SBDR* couples.

In the sandy bridge test we obtained all the bits to 1 except for the first 12 bytes which are the exact span of a virtual page.

What we would expect is the first 14 bits of the address, to never change in *SBDR* address couples, but since the *row* identification bits are XORed over to *bank* group, *bank* and *channel* selection bits, we see every bit to 1 except from the first 12. The first 12 bits are always set to 0 because we test with granularity of a virtual page ($2^{12} = 4096$).

6.4 Noise reduction

When measuring the latency histogram, we found an interesting form of noise. As depicted in figure 6.5, it was displayed in the form of overlapped triangles, of fixed width and fixed offset, with the same height with respect to the number of accesses.

This form of noise is probably produced by the interference of the *DRAM Row Refresh* refresh, which is the only periodic event inside the *DRAM*.

To eliminate it, we perform every measure three times and continue

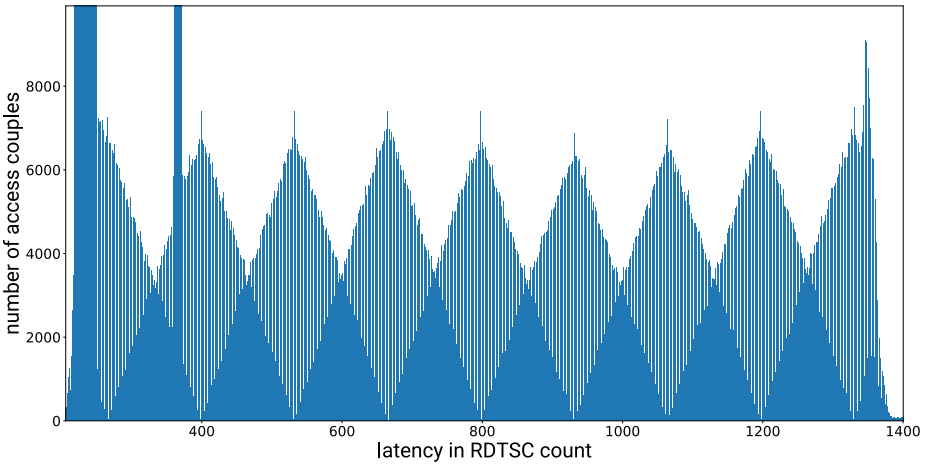


Figure 6.5: Memory Refresh Noise, Detail taken from the Uncached Accesses Latency Histogram

sampling until the three samples are identical to the fourth bit.

6.5 Bins Clustering on Intel Skylake

The result of the application of our bins clustering algorithm, applied on our Skylake setup can be seen in Figure 6.6. In that filtered histogram we can see a clear subdivision between low-latency (colored in blue) and high-latency accesses (colored in red), with no outliers in between. This clean subdivision will greatly improve the output of the subsequent steps.

6.6 Set Partitioning on Intel Skylake

The result of the application of our SBDR sets subdivision (algorithm 2) is shown in figure 6.7.

Our skylake test platform has 32 total *bank*s, 16 for each module, subdivided in 4 *bank* groups of 4 *bank*s each. Running the tests on our setup yields 16 big sets, with ~1000 pairs each and 32 small sets, with ~250 pairs each.

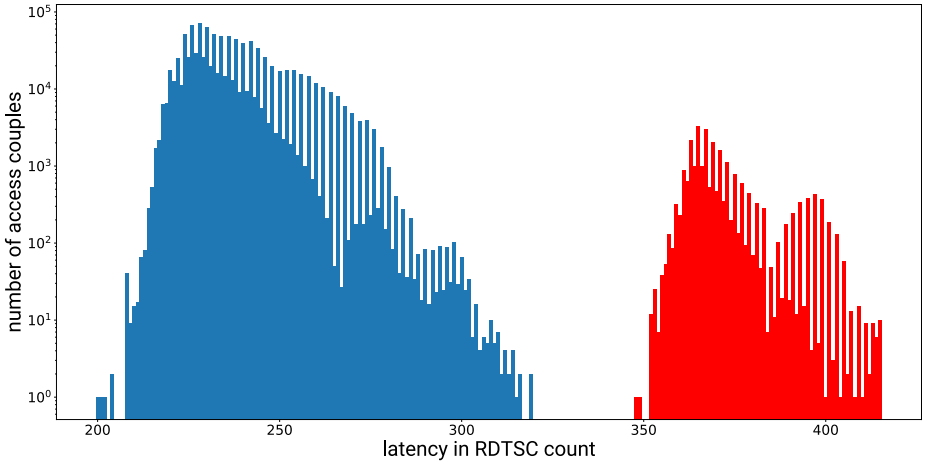


Figure 6.6: Skylake Uncached Accesses Latency Histogram, Filtered, with Highlighted Clusters Partitioning

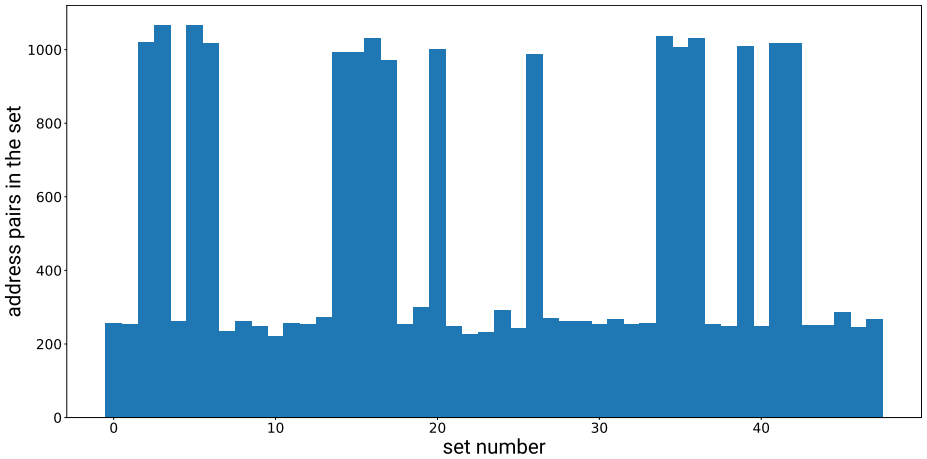


Figure 6.7: Skylake High-Latency Sets Partitioning

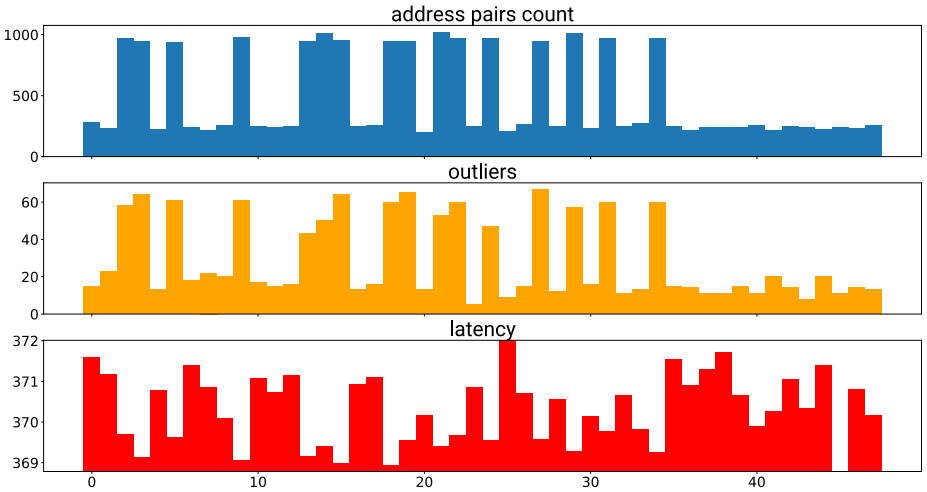


Figure 6.8: Skylake Sets Features Analysis on High-Latency Sets

We are probably seeing not only *SBDR* pairs (the 32 sets containing less addresses) but also *bank*-conflict pairs, which are 3-times as many as the *SBDR* pairs. This because if we consider a single *bank group*, composed by 4 *banks*, and we pick an address inside the first *bank*, all the address in the other *rows* of that *bank* will satisfy the *SBDR* condition, while all the address in the other 3 *banks* ($\sim 3 \cdot SBDR$ pairs) will satisfy the *bank*-conflict condition.

As we can observe from 6.7, in our Skylake testing environment, after sampling one million address pairs, and splitting the high-latency bins, we have 48 total sets, 32 with ~200 address pairs each and 16 with ~1000 address pairs each.

To further refine our subdivision we tested every address of each set against all the others of the same set, removing the addresses which lied far higher or lower than the mean latency of the set. We call these addresses outlier pairs.

6.7 Set Partitioning Interpretation

To give a reasonable meaning to the two kinds of sets we sampled and plotted two features, for each set:

- the number of outliers removed from the set
- the mean access latency for the members of each set

We can see those values plotted in figure 6.8. The number of outliers is clearly directly proportional to the size of the set, so we can assume that this feature is the consequence of a uniform noise applied on the latencies of all address pairs. The latency, on the contrary is inversely proportional to the pair count of each set. This makes large sets slightly faster than small sets. We recall that the set partitioning on Sandy Bridge (figure 6.10), apart from the added noise, showed only one kind of set.

This let us propose a possible interpretation for the origin of the two sets. As it is known from previous literature [20], the formation of this kind of slow-latency set is caused by the so-called *Same Bank Different Row* conflicts. Furthermore, the only architectural change between *DDR3* memories (used by Sandy Bridge) and *DDR4* memories (used by Skylake) is the introduction of *Bank Groups* [30].

Bank Groups introduce and interesting behaviour to the memory architecture, called *bank* conflict [3]; when two consecutive memory accesses fall into the same *bank group*, the second access is slowed down considerably (6 clock cycles instead of 4 clock cycles on our Kingston modules as ACTIVATE-to-ACTIVATE time).

We can consequently assume that one of the two kinds of set is related to the *SBDR* conflict, while the other is due to *bank group* conflicts.

To assign the right kind of conflict to the right kind of set we can leverage our knowledge on the internal structure of our *DRAM* chips. We know that our *banks* groups are composed by 4 *banks*, then for each first address

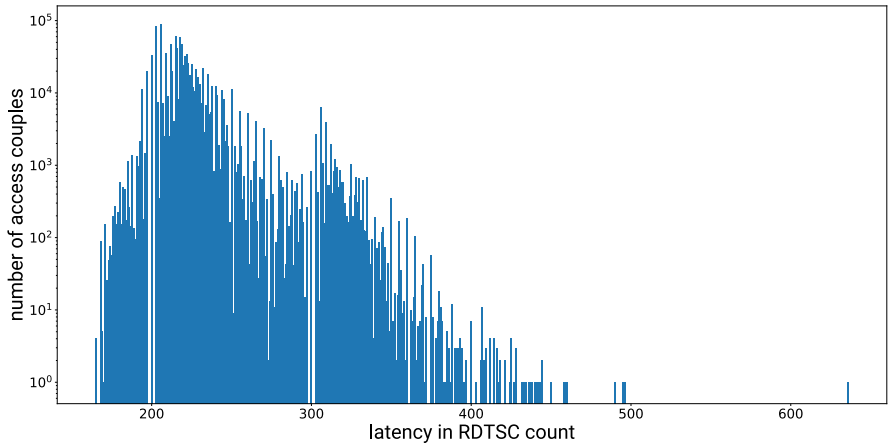


Figure 6.9: Sandy Bridge Filtered Uncached Accesses Histogram

of a consecutive two-address sequence, there will be N addresses which will cause a *SBDR*-conflict i.e. the second address will fall into the same *bank* as the first, and $3N$ addresses which will cause a *bank group* conflict i.e. the second address will fall into the same *bank group* as the first access, but not in the same *bank*.

So we expect one of the sets generated by these particular conflicts, to be three times bigger than the other. Accordingly we can assume that the large sets we found are related to *banks* conflicts, while the small sets are related to *SBDR* conflicts.

6.8 Missing Sandy Bridge Comparison

When running the same tests on Sandy Bridge architecture, we see a thin to non-existent subdivision in clusters (figure 6.9). This weak subdivision leads to a high number of outlier pairs inside the bins classified as *SBDR*.

Every outlier pair, since it is a combination of two address which do not satisfy the *SBDR* condition, will never produce a *SBDR* latency, regardless of the set used for the measure. Thereupon every outlier pair will

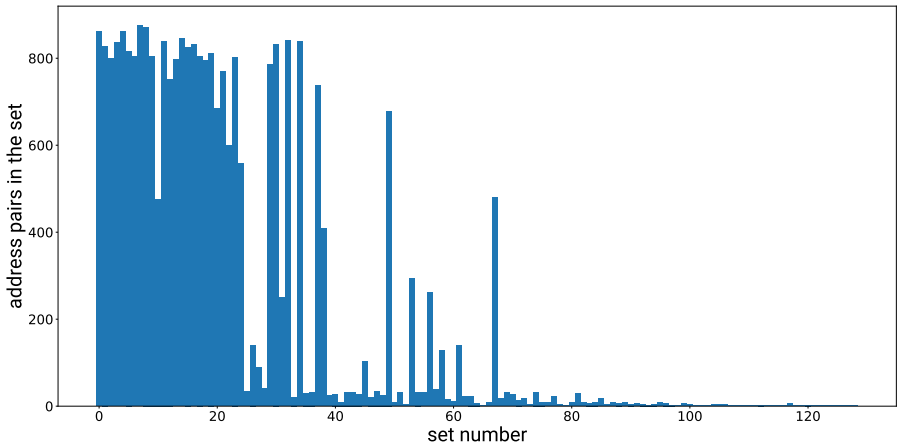


Figure 6.10: Sandy Bridge High-Latency Sets Partitioning

produce a new set, that will likely remain empty. Since bins are scanned from high latencies to low latencies, we can confirm this hypothesis by noticing that all the empty sets are right-aligned in the sets partitioning (figure 6.10), it follows that they are generated by low latency pairs, almost certainly outliers pairs that were misclassified in our noisy latency subdivision.

6.9 Bruteforcing Application on Intel Skylake

We now show the results of the application of the bruteforcing (algorithm 3) and deduplication (algorithm 4) procedures.

A simple way to represent a set of mapping functions is to put in each row the indexes of the bits XORed together to form each function. The following is the mapping function reported in the DRAMA [20] paper, to better understant the notation, the reader can refer to a graphical representation of the same function in figure 6.11.

19 18 13 12 9 8

22 18

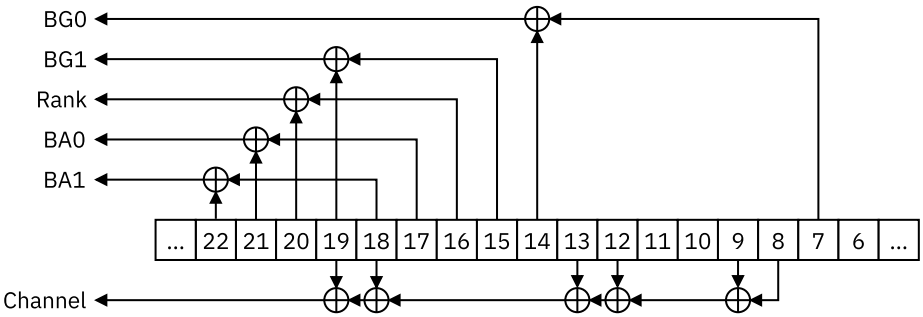


Figure 6.11: Drama Paper Function Graphical Representation

21 17

20 16

19 15

14 7

The results of our test are expressed with a slightly different notation, on the left part of each arrow we have the mapping functions, while on the right part we have the fitness score of each function. Our fitness score represent how well that function maps to our high-latency sets. The function from the DRAMA paper has the correct addressing fields associated with each function; they were able to correctly name the different functions because they directly probed the *SO-DIMM* modules connection pins with an oscilloscope.

As a result of our whole test, we obtained the following functions for the small sets (figure 6.12):

22 15 13 12 9 8 –> 3232

19 15 –> 3232

21 17 –> 3232

22 18 –> 3232

20 16 –> 3232

14 7 –> 3232

On the contrary, our mapping functions have no name and may be

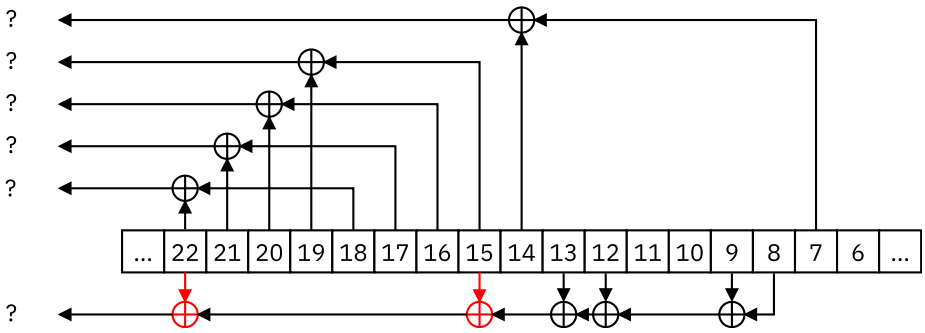


Figure 6.12: Skylake Small Sets Function Graphical Representation

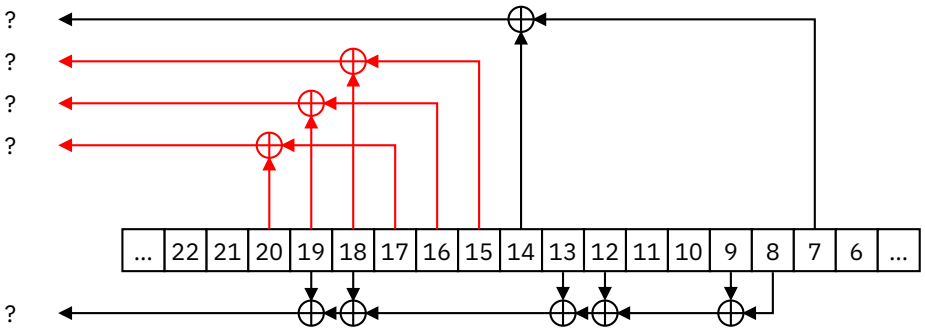


Figure 6.13: Skylake Large Sets Function Graphical Representation

the result of the composition of other functions. We picked one of the minimal sets of functions which cover all the addressing bits.

Our small sets functions (figure 6.12) are identical to what the DRAMA [20] researchers found with their physical probing setup. The *channel* function is different because we chose a combination of two mapping functions, e.g. $8\ 9\ 12\ 13\ 18\ 19 \circ 18\ 22 = 8\ 9\ 12\ 13\ 19\ 22$.

We also ran our test on the large sets found during the set partitioning and the results are as follows (figure 6.13):

19 15 13 12 9 8 -> 1616

14 7 -> 1616

18 15 -> 1616

19 16 -> 1616

20 17 -> 1616

The small sets are ascribable to *SBDR* pairs, therefore among the constant functions listed will be:

- *channel*
- *bank group*
- *bank* address.

The large sets are ascribable to *Bank Conflict Pairs*, so among their functions the following components will be constant:

- *channel*
- *bank group*.

The fact that the large set functions (figure 6.13) are not a strict subset of the small set functions is rather baffling.

There is probably more to be investigated in the meaning of the functions derived from what we named “large sets”. This is probably caused by the internal bit permutation performed by the *DRAM* modules. A permutation probably similar to what Matthias Jung found in his paper [21]. However since knowing the *SBDR* mapping is enough for our purposes, we conclude here our detection algorithm.

Chapter 7

Proposed countermeasures

With our detailed knowledge on the addressing mapping function, we are now able to introduce two countermeasures to the *Rowhammer* attack. They aim to have the lowest possible impact in terms of both performance, memory usage, as well as power consumption.

Furthermore we propose both a software and a hardware coutermeasure:

The software one is effective in the short term, with low deploying costs and also effective on already existent systems.

The hardware one is equally important for its lower performance costs, but also for the deployment of new systems which are already resistant to all the attacks which depend on the knowledge of the memory mapping function such as *Rowhammer*.

7.1 Air-Gap

The main idea of our software countermeasure, named Air-Gap, is to keep at least eight empty *rows* between the physically allocated memory of different processes.

We will consider Linux as an example kernel for applying our countermeasure, but our method has general value and can be applied also to other kernels and Operating Systems.

In Linux, memory is allocated with a minimum granularity of virtual pages of 4096B each (in x86 and ARM platforms). Each virtual page is physically contiguous with respect to physical addresses, hence addressing a virtual page uses bits from the twelfth on. As we can see from the Skylake mapping function (figure 6.11), bits up to the twelfth are XORed only with *channel* and *bank group* selection bits, the same happens even in the other mapping functions obtained by the DRAMA [20] paper. This happens for performance reasons: if no *row* selection bit is XORed on the first twelve bits, a page is stored in at most one *row* per *bank*. If this were not true, then accessing a single page would result in a *row* closing and *row* opening inside a single *bank*, causing a massive latency issue. The same happens with respect to *bank* addressing, at least on systems where *bank groups* are employed, such as our Skylake platform. Bank addressing bits are never XORed into page bits, so that one page, resides at most in one *bank* inside each *bank group*. This way we avoid *bank group conflicts* by never reading sequentially from two *banks* in the same *bank group*. Consequently a virtual page is distributed only through *channels* and *bank groups* (figure 7.1), as well as *ranks* in the case of the Samsung Exynos 7420 - *LPDDR4* function, described by DRAMA [20]. This is a minor performance issue because only a small part of a *DRAM*'s infrastructures are shared between *ranks*.

According to Yoongu Kim's paper [1], the maximum reach of disturbance errors generated by a *row* is eight *rows* as it can be seen in figure 7.2.

We introduce a policy to avoid interference between userland processes and between *userspace* and *kernelspace*. The policy is that "different

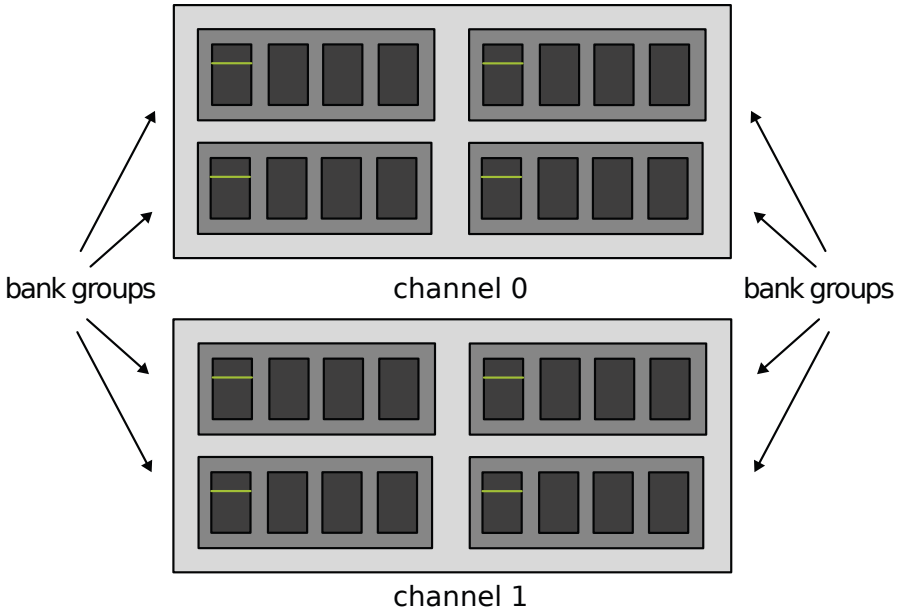


Figure 7.1: Page Distribution Across *DRAM* Structure

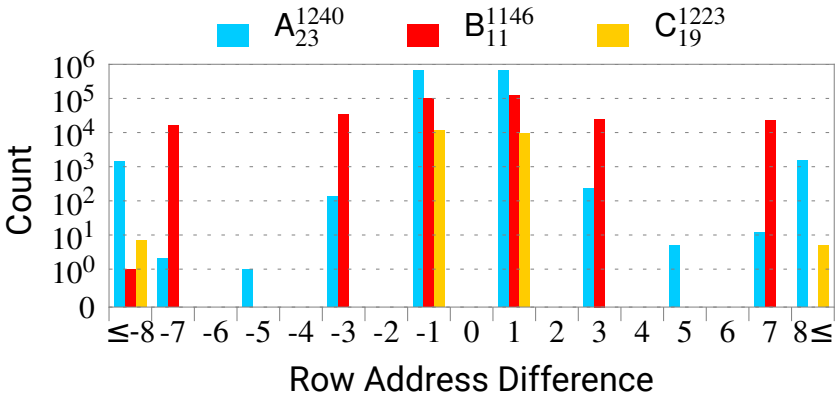


Figure 7.2: Reach of Disturbance Errors Generated by a Single Row. Graph rebuilt from Kim, Yoongu et al. “Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors” [1].

entities e.g. processes or kernel structures, may reside at **no less than eight *rows* of distance** from one another" as represented in figure 7.3. Our Air-Gap countermeasure works by enforcing this policy inside the memory allocator. For each page we can compute the *row* number, for example on Skylake architecture by picking the most significant 16 bits, and assign each process to *row* numbers whose difference is at least eight. This *row* check has to be performend only if the outcome of the *bank*, *rank* and *bank group* selection is the same on both pages, if those numbers diverge, the two pages are going to reside in different *banks* and thus there would be no *Rowhammer*-related concerns. Checking the *bank* and *bank group* numbers is entirely optional and aims at reducing the memory footprint of our countermeasure, but not applying this optimization does not weaken the countermeasure in itself. To check those addressing fields we need the knowledge of the mapping function, Even so, without this optimization Air-Gap can be applied even to machines with unknown mapping functions.

7.1.1 Performance cost

The most considerable cost of our countermeasure is met in terms of consumed space. The wasted space grows with memory fragmentation; for each new process we have to consider 8 *rows* (~65KB per *bank*) of wasted space. This is a relatively low cost but scales linearly with the number of processes. If we consider a typical usage in a Linux environment we will have around 100 entities to be isolated; in that environment our Air-Gap countermeasure would consume ~200MB of RAM in a 32 *bank* setup, which is a memory footprint similar to the one of most Linux daemons i.e. colord. Since we trust our own kernel, we can avoid to isolate kernel structures from each other, thus saving some space for higly fragmented kernel data structures such as *PTE*s.

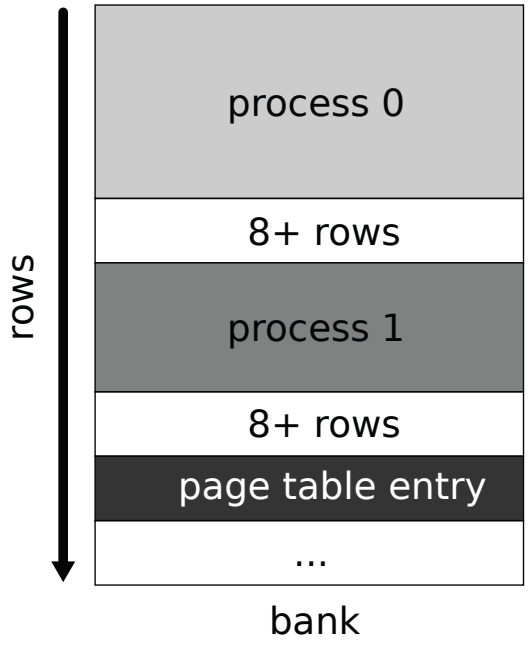


Figure 7.3: Air-Gap Policy Representation

Although memory fragmentation might increase our space consumption cost exponentially, several different solutions have been proposed in order to counteract it [31] and many of them are already merged in mainline Linux kernel. We leave a deeper analysis on memory fragmentation to future research.

7.1.2 Comparison with G-CATT

A suitable rival to compare our Air-Gap software countermeasure to, is Ferdinand Brasser's G-CATT [23]. This countermeasure acts in a very similar way to Air-Gap, by partitioning the installed *DRAM* into security domains. They form each security domain by splitting each *bank* in half, keeping an empty *row* in between and assigning different entities to different halves of the available *banks*. A first advantage of Air-Gap over G-CATT is the thickness of the boundaries between isolated domain; in G-CATT the boundaries are only one *row* wide, while our countermeasure

keeps entities at eight *rows* of distance or more. Although on our Sandy Bridge setup we were unable to observe flips at more than 1 *row* of distance from the attacking ones (figure 3.3), Yoongu Kim's paper [1] shows clearly that disturbance errors may propagate up to the eighth *row* in both directions from the attacking *rows* (figure 7.2). Furthermore, every *DRAM* setup has a limited number of *banks*. In our Skylake setup we have 32 *banks* for 16GB of installed memory; if we applied the G-CATT countermeasure on our machine, we would have at most 64 different domains. This makes it possible to separate kernel pages from user pages, but does not allow to isolate every process from the others as Air-Gap does. On the contrary, our countermeasure on the same setup would add up only 2MB of space for each process, thus allowing to have even 512 separate security domains while consuming only $\frac{1}{16}$ of installed memory. Therefore our countermeasure is at the same time more effective in contrasting the *Rowhammer* phenomenon while having a lower space consumption penalty. The Air-Gap countermeasure is particularly useful in low-medium size memory setups like smartphones and laptops, where the number of processes to be isolated is in the hundreds and the number of *banks* is low.

7.2 Row-Mix

The most important goal of the research efforts in both industry and academia is to release systems which are already compliant with security measures able to cope with known vulnerabilities. To achieve this goal with *Rowhammer* in the near future, we introduce a hardware countermeasure called Row-Mix. This countermeasure can be deployed only in collaboration with *DRAM* chip manufacturers, but once released, it will secure every system on which the new modules are installed, regardless of the Oper-

ating System used.

Effective implementations of the *Double Rowhammer* attack are possible only by knowing the exact mapping function employed on the target machine. Memory mapping functions are the result of the composition of several functions:

- Kernel *virtual addresses* to *physical addresses* mapping
- CPU memory controller *physical addresses* to addressing bits
- *DRAM* modules addressing bits permutation

The kernel mapping can be guessed by leveraging Huge Pages [18], or by exploiting the known behaviour of the Linux buddy allocator as tested in the Drammer paper [2]. The memory controller's mapping depends only on the installed memory configuration. It has been reverse engineered in the past already, by Mark Seaborn [19] and Peter Pessl [26]. In this thesis we described extensively our own methodology for reverse engineering this component of the global mapping function. The last part of mapping is performed inside each *DRAM* module for performance reasons; Matthias Jung has proposed an interesting technique for revealing this mapping in unknown memory modules, provided we have physical access to them and some equipment for having direct access to the module [21].

As a hardware countermeasure we propose to shuffle the *row* addressing inside each memory module. Our countermeasure is made of three parts:

- A permutation generation logic: that leverages a *true random number generator* to obtain a unique permutation every time the module is initialized.
- A state register, to save the selected permutation
- A combinatorial logic component, which permutes in real time the addresses as they pass from the *Row-address MUX* towards the *bank*

groups. The structure of the countermeasure is depicted in figure 7.4. During the module initialization, a random permutation is generated and pushed to the state register. Care has to be taken when sourcing the entropy necessary to generate a random permutation from a true random number generation, otherwise the permutations could become predictable. The combinatorial logic may employ a pipelined scheme if its single-stage critical path is greater that the one of a typical memory controller.

7.2.1 Performance cost

This countermeasure has two main performance drawbacks: The module initialization takes slightly longer to allow the permutation to be generated and pushed to the state register. Even so, this would probably cause little to no boot slowdowns because a considerable amount of time passes from the *DRAM* modules being powered on and them being actually used for the first time. A series of checks are usually performed during that time, such as:

- Intel ME initialization
- Authenticated Code Modules execution
- Intel Boot Guard enforcement

During all these operations the CPU runs in Cache as RAM mode and this will hide the time required for the permutation components to be initialized completely. The steps required to boot an Intel Architecture Platform are described in a white paper by Intel [32].

We have only little runtime cost since our combinatorial logic structure will have a slightly longer critical path with respect to an unprotected implementation. If the latency will be particularly high, we could always redesign our combinatorial part employing a pipelined structure; this

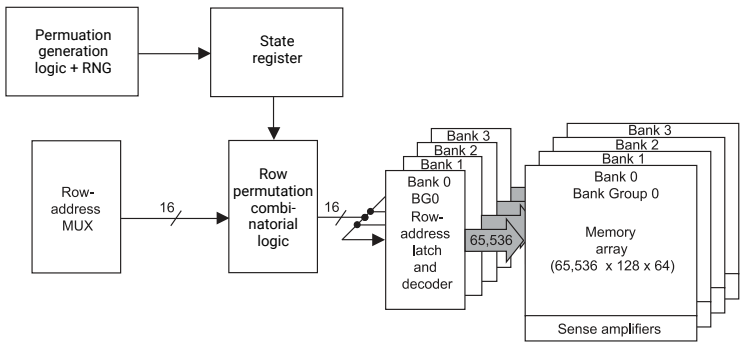


Figure 7.4: Row-Mix Countermeasure Functional Scheme

way the throughput is the same of an unprotected DRAM module, only the latency remains slightly higher. This latency will anyways be partially hidden by the memory controller's own typical latency.

7.2.2 Security

Since we generate true random permutations of *row* addressing bits, at each inizialization, the mapping scheme becomes one in $r!$ possible mappings. This means $16! = 20922789888000$ on a 16 bit *row* addressing setup, a number far too high to be simply guessed. The Row-Mix countermeasure will make precise *row* addressing a very difficult task to achieve, with little performance and silicon area consumption costs. The smallest pieces of information we are required to know about the mapping function in order to perform a *Double Rowhammer* attack are the addresses of three physically adjacent *rows* inside the same *bank*. Even if the attacker resolved all the other two mapping functions i.e. Operating System and memory controller mappings, picking three consecutive *rows* would become hard. The attacker could try to pick only the two attacking *rows* and scan all the other $2^r - 2$ *rows* to find bit flips, admitting that they have access to a sequence of pages of memory which are spread over all the available physical memory, but provided they had such access, the prob-

ability of randomly selecting two *rows* with a distance of one in between is:

$$\frac{\text{\#Rows with a distance of 1}}{\text{\#Combination of two rows}} = \frac{R-2}{\binom{R}{2}} = \frac{65534}{\binom{65536}{2}} = \frac{65534}{2147450880} = 3.05\text{e}{-5}$$

So it will take on average $\frac{1}{3.05\text{e}-5} = 32769$ attempts to find two *rows* with distance equal to 1. Since the sufficient number of iterations for the *Rowhammer* code loop to trigger the vulnerability is around one million, and this takes tens of seconds on commodity hardware, it would take little less than 4 days of continuous hammering to find a vulnerable couple of adjacent *rows*. This optimistic estimate also relies on the assumption that the attacker is able to allocate memory in a very precise way. If that assumption were not true, the probability of picking randomly three consecutive *rows* would be:

$$\frac{R-2}{\binom{R}{3}} = \frac{65534}{\binom{65536}{3}} = \frac{65534}{46910348656640} = 1.39\text{e}{-9}$$

This means an average of $\frac{1}{1.39\text{e}-9} = 715816960$ attempts to succeed. If we consider just 1 second for each hammering attempt, it will take more than 22 years of hammering to obtain three consecutive *rows*. This makes it very difficult to mount a successful *Rowhammer* attack. In addition to that, the longer the time needed to find the correct *rows*, the higher the probability that the attacker is discovered through some the detection methods as the ones described in Marco Chiappetta's paper [33].

7.2.3 Comparison with PARA

Kim Yoongu, in his first *Rowhammer* analysis paper [1], proposed an interesting hardware countermeasure called *Probabilitstic Adjacent Row Activation*. PARA is implemented in the memory controller and involves refreshing the neighbor *rows* after each memory access with a non-null

probability. This is a smart countermeasure with respect to counter-based ones; it being *stateless*, the complexity of the implementation is low. Implementing this kind of countermeasure in the memory controller means that the controller itself needs knowledge on how the *row* addressing is performed, although this assumption is highly impractical. Firstly, the controller manufacturer would have to ask every memory manufacturer to disclose their *row* mapping function for each memory module produced in the past. Secondly, if we had to apply such countermeasure on *DDR3* modules, we would have to include the mappings of 10 years of memory manufacturing, and all this without even considering forward compatibility. If we had to build a controller for *DDR4* instead, we would have to update our memory controller with new mapping functions as new memory modules are released in the future. Since manufacturers only usually provide 2 or 3 years of software updates, this would likely leave old memory controllers equipped with newer *DIMM* modules unprotected. Our countermeasure instead is implemented in the *DRAM* itself, so there is no dependence on the knowledge of the *row* mapping since we are performing our countermeasure directly on the *row* bits. As a result, our countermeasure will always be functional, even if the protected module is connected to a newly produced memory controller.

Chapter 8

Conclusion

In this work of thesis we have presented a complete overview of the *Rowhammer* phenomenon, starting from its physical causes, focusing on how to reliably obtain a mapping function to make every attack possible, finishing with the proposal of two novel countermeasures.

Our detection algorithm pushes forward the state of the art in detecting address mapping functions in unknown system, our method is more reliable than current software method and achieves results that until now were only possible trough physical probing. This will be useful in future research in the fields of *side channel attacks* and *cache attacks*.

The software countermeasure we proposed can be implemented and applied in already deployed systemd. Our implementation is focused on the Linux kernel, but the key concepts are general and can be ported easily to other Kernels and Operating Systems, protecting a great variety of systems. We offer a high level of security with less severe drawbacks with respect to the current best software countermeasures (G-CATT [23]).

Our hardware countermeasure will give manufacturers the ability to produce memory modules which will be born resistant to this vulnerability, and will be able to secure every system on which they will be in-

stalled. In addition to this, our Row-Mix countermeasure, once implemented, will work regardless of the machine on which the *DRAM* will be installed.

But there is more behind this technical overview. *Rowhammer* is a phenomenon of particular interest, differently from other kinds of attacks it spans many different layers of the target system. It has its roots in the physical hardware implementation of *DRAM* memories but its leaves are as high-level as a javascript webpage code. This is what makes this attack so dangerous and powerful. For example when we design a system e.g. a browser sandbox we will hardly think about the consequences of including an assembly instruction such as CLFLUSH to our allowed set. This is exactly what happened for the Google NaCl sandbox (see Project Zero's paper [9]).

This phenomenon represents a very important message for the whole engineering community: it is never enough to look only inside the boundaries of our models. We work every day with complex, multi-layered systems; these machines mandate us to always keep an all-round vision of the entire world.

But not only menaces lie in the boundaries between disciplines, technological breakthroughs and innovation are often achieved by contamination and mixture of knowledges.

Finally, a vulnerability like *Rowhammer* will become relevant more than ever with the advent of the next generation of non-volatile memories (NVMs), most importantly if these new kinds of memories will be used as permanent cache for volatile *DRAM*s. Some of the current countermeasures could be defeated in such a setting, for example in counter-based countermeasures, if the charge depletion will persist across reboots, then just rebooting multiple times the machine might reset the counters and elude the countermeasure.

Bibliography

[1] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 361–372. [Online]. Available: https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf

[2] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, *Drammer: Deterministic Rowhammer attacks on mobile platforms*. Association for Computing Machinery (ACM), 10 2016, vol. 24-28-October-2016, pp. 1675–1689. [Online]. Available: https://vvdveen.com/publications/drammer.pdf

[3] *8Gb: x4, x8, x16 DDR4 SDRAM*, Micron Technology, Inc., 2015, rev. K 7/17 EN. [Online]. Available: https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf

[4] D. T. Wang, "Modern dram memory systems: Performance analysis and scheduling algorithm," Ph.D. dissertation, University of Maryland, College Park, MD 20742-7011 (301)314-1328., 2005. [Online]. Available: https://www.ece.umd.edu/~blj/papers/thesis-PhD-wang--DRAM.pdf

[5] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 60–71, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2508148.2485928

[6] Wikipedia. (2017) Row hammer — wikipedia, the free encyclopedia. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Row_hammer&oldid=802794809

[7] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," *CoRR*, vol. abs/1507.06955, 2015. [Online]. Available: http://arxiv.org/abs/1507.06955

[8] T. L. Foundation. Page table entries description. [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/understand006.html

[9] T. D. Mark Seaborn. (2015) Exploiting the dram rowhammer bug to gain kernel privileges. [Online]. Available: https://googleprojectzero.blogspot.it/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

[10] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi

[11] Apple. (2017) About the security content of mac efi security update 2015-001. [Online]. Available: https://support.apple.com/en-us/HT204934

[12] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand, "Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits," *Proceedings of the IEEE*, vol. 91, no. 2, pp. 305–327, 02 2003. [Online]. Available: https://pdfs.semanticscholar.org/13a1/32025a76e35280022bff23e6fd0f0f2e9658.pdf

[13] K. Saino, S. Horiba, S. Uchiyama, Y. Takaishi, M. Takenaka, T. Uchida, Y. Takada, K. Koyama, H. Miyake, and C. Hu, "Impact of gate-induced drain leakage current on the tail distribution of dram data retention time," in *International Electron Devices Meeting 2000. Technical Digest. IEDM (Cat. No.00CH37138)*, 12 2000, pp. 837–840. [Online]. Available: https://pdfs.semanticscholar.org/8963/230958477fd760ac060d5d2eb66d310f78c6.pdf

[14] R. Qiao and M. Seaborn, "A new approach for rowhammer attacks," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 6 2016, pp. 161–166. [Online]. Available: http://www.seclab.cs.stonybrook.edu/seclab/pubs/host16.pdf

[15] LWN. (2012) The android ion memory allocator. [Online]. Available: https://lwn.net/Articles/480055/

[16] T. L. Foundation. (2016) pagemap, from the userspace perspective. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/pagemap.txt

[17] LWN. (2012) Cap_sys_admin: the new root. [Online]. Available: https://lwn.net/Articles/486306/

[18] T. L. Foundation. Hugetlbpage support summary. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt

[19] M. Seaborn. (2015) How physical addresses map to rows and banks in dram. [Online]. Available: http://lackingrhoticity.blogspot.it/2015/05/how-physical-addresses-map-to-rows-and-banks.html

[20] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 565–581. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[21] M. Jung, C. C. Rheinländer, C. Weis, and N. Wehn, "Reverse engineering of drams: Row hammer with crosshair," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: ACM, 2016, pp. 471–476. [Online]. Available: https://ems.eit.uni-kl.de/fileadmin/ems/pdf/38-final-acm.pdf

[22] M. Seaborn. (2015) Vendor responses to the rowhammer bug. [Online]. Available: https://github.com/google/Rowhammer-test/blob/master/docs/vendor_responses.md

[23] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 117–130. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser

[24] O. Santos. (2015) Mitigations available for the dram row hammer vulnerability. [Online]. Available: https://blogs.cisco.com/security/mitigations-available-for-the-dram-row-hammer-vulnerability

[25] M. Seaborn. (2017) rowhammer_test_ext: Extended version of rowhammer_test. [Online]. Available: https://github.com/google/Rowhammer-test/tree/master/extended_test

[26] P. Pessl, D. Gruss, C. Maurice, and M. Schwarz. (2017) Drama reverse-engineering tool and side-channel tools. [Online]. Available: https://github.com/IAIK/drama

[27] "How to benchmark code execution times on intel® ia-32 and ia-64 instruction set architectures," Tech. Rep.

[28] Wikipedia, "Time stamp counter — wikipedia, the free encyclopedia," 2017. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Time_Stamp_Counter&oldid=800977564

[29] ——. (2017) Serial presence detect — wikipedia, the free encyclopedia. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Serial_presence_detect&oldid=782127854

[30] G. Allan. (2013) Ddr4 bank groups in embedded applications. [Online]. Available: https://www.synopsys.com/designware-ip/technical-bulletin/ddr4-bank-groups.html

[31] ——. (2013) Internal fragmentation in slab allocators: a simple comparison. [Online]. Available: https://elinux.org/Kernel_dynamic_memory_analysis#Internal_fragmentation_in_SLAB_allocators:_a_simple_comparison

[32] J. A. P. Jenny M Pelner, "Minimum steps necessary to boot an intel® architecture platform," Intel Corporation, Tech. Rep., 2010. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/minimal-intel-architecture-boot-loader-paper.pdf

[33] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Appl. Soft Comput.*, vol. 49, no. C, pp. 1162–1174, Dec. 2016. [Online]. Available: https://doi.org/10.1016/j.asoc.2016.09.014