FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Mitigating Rowhammer Attacks with Software Diversity

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Manuel Wiesinger, BSc
Matrikelnummer 00825632

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Dr.techn. Edgar Weippl

Wien, 30. April 2018

Manuel Wiesinger

Edgar Weippl

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Mitigating Rowhammer Attacks with Software Diversity

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Manuel Wiesinger, BSc
Registration Number 00825632

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dr.techn. Edgar Weippl

Vienna, 30th April, 2018

Manuel Wiesinger Edgar Weippl

Erklärung zur Verfassung der Arbeit

Manuel Wiesinger, BSc
Goldschlagstraße 112/48
1150 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. April 2018

Manuel Wiesinger

Acknowledgements

I would like to thank my advisor Univ.-Prof. Dr. Stefan Brunthaler for initiating the idea of this thesis, for constantly giving helpful advise and motivation whenever I needed it. He always supported me generously and encouraged me to attempt a scientific career. Moreover, I want to thank Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr. techn. Edgar Weippl for his advice and supervision.

Furthermore, I want to express my gratitude to SBA Research, the company that has employed me for working on operation system security and is giving me the opportunity to continue my research as employee.

My gratitude goes to my parents Brigitte and Thomas, as their generosity allowed me to study without pressure. They always kept supporting me during my studies. Especially I want to thank my father who awakened my interests for natural science, technology and computer science. Furthermore, I want to thank my sister who always listened to me and made me laugh. I also want to thank my grandmother Gertrud for showing interest, supporting me and constantly believing in me.

I want to thank my girlfriend Klara for her interminable support throughout my studies, for always listening to me and giving me essential advice. I cannot imagine finishing this thesis without her.

Finally, I to thank my university friends Florian Schweikert, Simon Wolfsteiner, Kristoffer Kleine and Albert Dengg with whom I attended many classes, had lunch and coffee during my studies. Additionally, my gratitude goes to Christoph Roschger who took the part of a mentor since my first semester.

Kurzfassung

Die Rowhammer-Schwachstelle ist ein drastisches Sicherheitsrisiko für moderne Computersysteme. Bösartige Angreifer können damit gezielt Daten manipulieren bzw. Kontrolle über ganze Copmutersysteme erlangen, indem sie Daten ohne jegliche Autorisierung modifizieren. Die vorliegende Arbeit analysiert die öffentlich bekannten Rowhammer-Angriffe auf Betriebssysteme und präsentiert erstmals eine alternative, auf Softwarediversität beruhende Lösung für dieses Problem. Da die bekannten Angriffe stets auf der Vorhersagbarkeit von in modernen Betriebssystemen eingesetzten Algorithmen basieren, beabsichtigen wir, diese Vorhersagbarkeit mittels Softwarediversität zu unterbinden. In concreto stellen wir PAGE SACRIFICE vor: eine leicht umzusetzende, effiziente Modifikation bestehender Mechanismen, die Rowhammer Angriffe verhindert, indem bei jeder Speicherzuweisung physisch benachbarte Speicherblöcke zufällig freigelassen und somit Speicherzuweisungen unvorhersehbar werden. Der Speicher wird dadurch allerdings nicht dauerhaft belegt — sobald das Betriebssystem einen Speicherblock wieder frei gibt, werden auch die freigelassenen Blöcke wieder verfügbar. Um unsere Verteidigungmaßnahme zu evaluieren, wurde ein Prototyp für Linux entwickelt, der anhand weitverbreiteter Messmethoden untersucht wurde. PAGE SACRIFICE stellt eine neue Maßnahme gegen Rowhammer-Angriffe dar.

Abstract

The Rowhammer vulnerability allows the modification of arbitrary data without authorization. This is poses a dramatic security risk for modern computing systems, as it allows malicious attackers to manipulate data or even gain the control over entire systems. This thesis analyzes publicly available attacks based on Rowhammer as well as known defenses against them. Finally, it suggests a novel solution to the problem based on software diversity. As existing attacks are typically based on the predictability of algorithms used in modern operating systems, we aim to prevent these predictability using software diversity. In concreto we present PAGE SACRIFICE, an easy-to-implement, efficient modification of existing operating system mechanisms which prevents Rowhammer attacks by making the locations of memory blocks unpredictable. This is done by randomly leaving neighboring blocks free when memory is allocated. Leaving memory blocks free does not mean that they remain occupied: As soon as the operating system frees a memory block, it also frees the blocks which were left empty during the memory allocation. To evaluate our defense, we implemented a prototype for Linux which we evaluated using widely known benchmarks. PAGE SACRIFICE is a new defense against Rowhammer based attacks.

Contents

CHAPTER 1

Introduction

Modern society relies heavily on secure computing infrastructure. This infrastructure must be protected against any kind of vulnerability, because cybercriminals and intelligence agencies use vulnerabilities of digital infrastructure to acquire or manipulate information. Therefore, recent hardware vulnerabilities such as Meltdown [1] and the Spectre-family [2] [3] have attracted great media interest as they affect almost any computer. Another serious hardware vulnerability is the so called *Rowhammer* bug [4]. It affects most memory modules and consequently most computers. Rowhammer can be used to modify arbitrary data if attackers succeed to execute code on their victim's machine. Similarly, to Meltdown and the Spectre family, it may be sufficient if victims are tricked to visit vulnerable websites. Avoiding these hardware vulnerabilities is difficult though. Unlike software vulnerabilities hardware vulnerabilities cannot be easily fixed by installing software updates, instead hardware vulnerabilities require hardware replacement. If critical vulnerabilities affect widely used hardware, it is not feasible to deploy fixes in time. Instead of replacing hardware, software often can be used to circumvent hardware bugs. Circumvention makes it possible to timely deploy software-based fixes for critical hardware problems. Proposed defenses against circumventing the Meltdown, the Spectre-family and Rowhammer vulnerabilities introduce significant performance loss tough [1] [2] [5].

This thesis aims to approach the problem of Rowhammer attacks by analyzing existing attacks and defenses. Based on our analysis we propose PAGE SACRIFICE, a novel mitigation strategy inspired by the ideas of software diversity (which we discuss in section 4.4). To the best of our knowledge all publicly available attacks based Rowhammer take advantage of deterministic operating system behavior. Applying software diversity allows us to eliminate these determinisms. To evaluate our ideas, we provide a proof of concept implementation of non-deterministic memory allocation for the Linux kernel. Finally, we analyze the effects of our proposed solution on operating system performance. This analysis covers memory consumption benchmarks, CPU benchmarks and a security evaluation. We hope that this work will lead to further research questions for the applicability of software diversity in modern operating systems.

1.1 Structure of the Thesis

This thesis is structured as following: Chapter 2 gives preliminary background information on hardware architectures and fundamental operating system tasks relevant for understanding Rowhammer attacks and PAGE SACRIFICE. Chapter 3 gives a comprehensive overview of proposed software and hardware defenses against Rowhammer attacks. In chapter 4 we explain the mechanisms necessary to understand the design of our proposed defense. Chapter 5 we extensively discuss how we implemented a prototype of our defense for Linux. Finally, we evaluate our implementation in chapter 6 and draw our conclusions about our work in chapter 7.

CHAPTER 2

Background

This chapter gives the background information necessary to understand our PAGE SACRIFICE mitigation strategy against Rowhammer based attacks. First, we briefly revisit the fundamental operating system principles, as they can be found in any undergraduate course book such as Operating Systems: Internals and Design Principles by William Stallings [6] and describe the Rowhammer problem.

2.1 Hardware Architecture

In today's computers the *memory bus* connects the central processing unit (CPU) with the *Dual In-line Memory Modules* (DIMM).

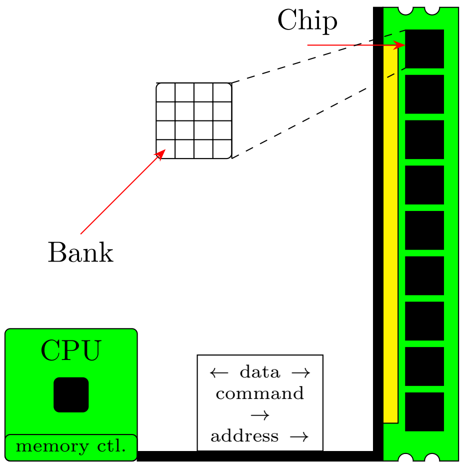


Figure 2.1: The CPU is physically connected to the primary memory via the memory bus.

Kim et al. [4] and Brasser et al. [7] summarize the low-level structure of DIMMs as following. Figure 2.1 gives a high-level view about the logical structure of the interaction of CPU and DIMMs as well as the DIMM structure. DIMMs follow a hierarchic structure, they consist of one or two *ranks*, which correspond to its front and backside. A rank consists of multiple interconnected *chips* containing one or more memory *banks*. Memory banks consist of horizontal *rows* and vertical *columns* of memory *cells*. The *wordline* connects the cells horizontally and the *bitline* connects them vertically. Memory cells are capacitors whose charged or uncharged state represents a binary data value (zero or one). The *access-transistor* manages the read and write access to the word lines and bit lines. Figure 2.2 depicts the structure of a memory bank.

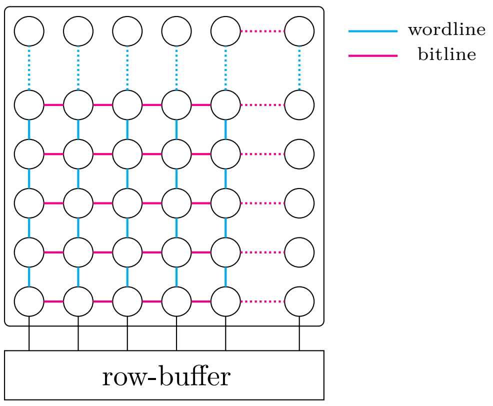


Figure 2.2: Structure of a DRAM bank.

A row of cells has typically the length of one *word*, the unit in which the CPU processes data. Today, the word size is typically 32 bit, 64 bit and for some architectures even 128 bit.

Memory Access

When the memory controller of the CPU issues a command to read a word, the reading operation triggers higher voltage on the word line, which in turn enables all access-transistors, connects the cells to the respective bit line and transfers the charge to the *row-buffer*. This is commonly referred to as *opening* a memory row. The row-buffer serves as interface to the memory bank, it carries out any operation on the data stored in a bank. Since opening rows is a destructive operation, the row-buffer immediately restores the data in the row, to preserve the data. Before another row can be accessed, the currently open row needs to be *closed*, i.e. the voltage of the word line is lowered again and the row-buffer is cleared.

2.2 CPU Cache and Main Memory Structure

Modern CPUs perform operations on integrated *registers*. Hence, data has to be loaded into registers prior to any data operation. As CPUs perform operations on registers faster than they can load blocks from primary memory, is sensible to store frequently used data blocks in faster but smaller *cache* memory, physically closer to the CPU. Cache memory allows CPUs to access frequently used data blocks faster. Commonly, CPUs employ several *cache levels*; If a block gets less frequently used, the CPU moves it to a larger but slower cache level; Typically CPUs use three cache levels, some cheaper CPUs use only two. Usually each core has its own first and second level cache, the third level is shared among all CPU cores. If a block is not accessed for a certain amount of time finally, the CPU removes it from cache. In case the data in it was modified, it is written back to main memory, otherwise it is simply removed as the data is still available in main memory. Figure 2.3 illustrates three-level caching.

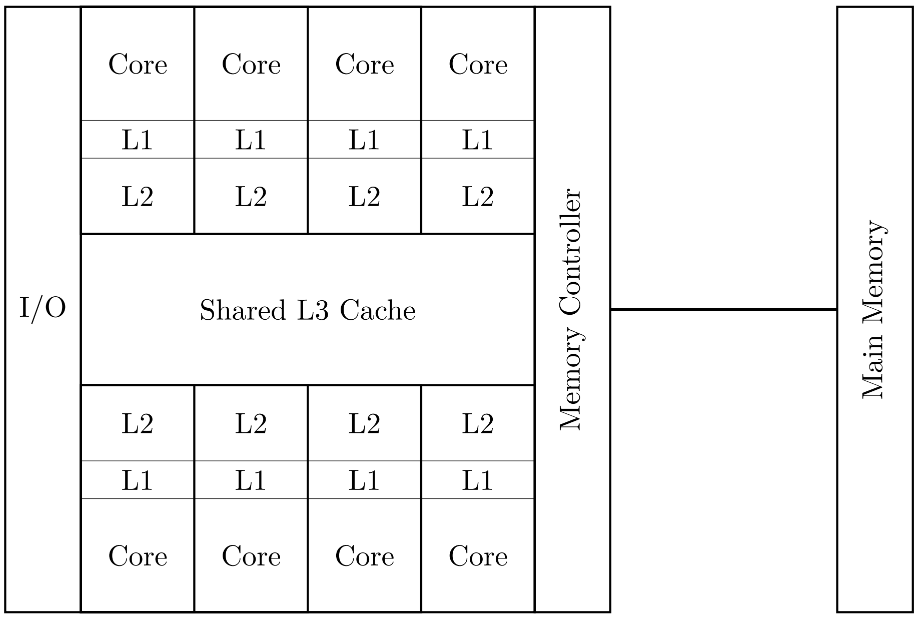


Figure 2.3: Schematic representation of an Intel Core i7-5960X (cf. Operating Systems: Internals and Design Principles [6])

The frequency the CPU loads a certain block to its registers is often described with a metaphor: frequently used blocks are called *hot* blocks and less frequently are called *cold* blocks. The *principal of locality*, states that practically memory blocks tend to reference physically close memory blocks [6]. Hence, data currently processed by the CPU is likely to reference to data that can be loaded quickly from cache.

Since CPU caches have rather limited storage capacities compared to the amount of main memory, the processor removes less frequently used pages (i.e. *cold* pages) by more frequently ones (*warm* blocks). The replacement strategies vary from CPU model to

CPU model and manufacturers generally do not publish them.

2.3 Memory Management

Multi tasking operating systems memory divide and manage memory to accommodate multiple processes. This fundamental task of modern operating system is called *memory management* [6]. In the following we discuss how memory management eases application development by providing *virtual memory* and how memory is subdivided for this purpose using *paging*.

2.3.1 Virtual Memory

Modern memory management provides *virtual memory*, such that each process can virtually address the whole system memory. The operating system instructs the CPU to translate virtual addresses used by applications to physical ones. This abstraction dramatically eases application and compiler development, as software developers do not need to take care about address translation and cannot accidentally interfere with other processes memory. If a program requires more memory than the operating system can provide, memory blocks storing data that can easily be reloaded (e.g. from secondary memory) can be removed. Additionally, less frequently used memory blocks can be temporarily *swapped-out*, i.e removed from memory and stored on secondary storage. This means that only the parts of an executable binary that are currently processed have to be in main memory, the same applies to files stored on secondary storage. The actual value of virtual addresses depends on the operating system and the processor architecture, however virtual addresses can be imagined to be enumerated from 0 to n, where n is the number of blocks the memory is subdivided into. Even memory addresses are continuous from an application's perspective, the memory blocks may be scattered across the whole memory. If virtually continuous blocks are physically stored in two or more chunks, they are called *fragmented*. The more logically continuous blocks are fragmented, the more address translation overhead is necessary to translate virtual addresses to physical ones. Consequently, fragmentation reduces system performance, to mitigate this operating systems typically attempt to minimize memory fragmentation (see section 4.1. To be able to remove parts of continuous blocks from main memory, the blocks have to be subdivided into smaller blocks. This can be achieved using the memory segmentation technique or the *paging* technique [6]. As the paging is the de facto standard method today we discuss it in the next section.

2.4 Paging

The prevailing method to implement virtual memory is *paging*. It divides physical memory into fixed-size chunks referred to as *frames* or *page frames*. A page frame stores one virtual *page*, which is a block of virtual memory of the same size. To map virtual addresses to physical addresses it is necessary to perform an address translation. The

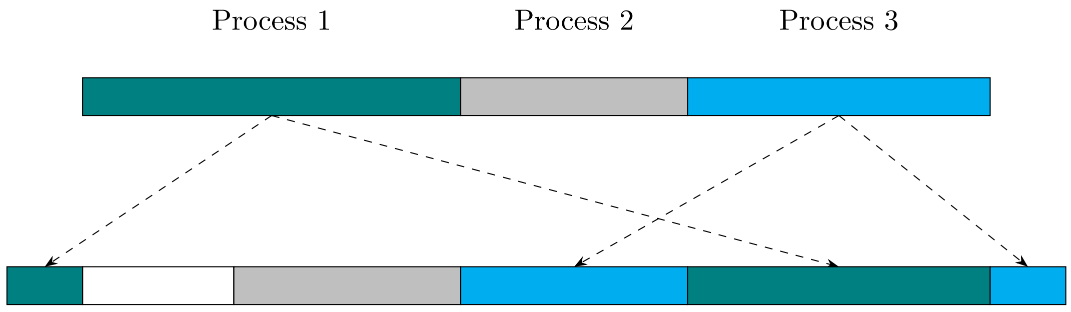


Figure 2.4: A possible mapping of virtual addresses to physical ones. Process 1 and process 3 are fragmented.

information which virtual address maps to which physical address is stored in a multilevel hierarchy of *page tables* (see section 2.4.1). The processor architecture defines their hierarchy as well as their structure so it can transparently perform address translation. [1] Each process running on the operating system has its own page table hierarchy.

Even though the address translation process is transparently done by hardware, it is the operating system's responsibility to ensure that the lower level page tables are present in memory. When a page is not in memory gets requested a *page fault* occurs and the processor raises an interrupt, and the *page fault handler* routine of the operating system gets executed. It loads the requested page and updates the translation tables accordingly. A page fault is, contrary to what the name suggests not an error but a normal operation. Often data is not loaded into memory until necessary, when a process requires a page not yet present in memory a page fault is raised to load it. When memory becomes scarce, the operating system removes (*pages out*) less frequently used pages from main memory. If the information stored in the removed page is available somewhere else (like on secondary memory), the relevant pages are cleared and marked as available, if the information is newer than on the backing storage, the *dirty* flag is set and the content is updated in the backing storage, before it is released. A typical example of a dirty pages, is a file modified and saved by the user. The file is first updated in memory, until the pages containing the file are written back to disk, they have their dirty flag set. Most operating systems also support to move pages from main memory temporarily to secondary memory, this way it is possible to remove pages containing data not available somewhere else. When memory pressure is high and the operating system therefore stores a lot of pages to secondary memory, it can come into a state, where more system resources are used for storing and loading pages than for systems' actual tasks. This state is called *trashing*.

The *transition lookaside buffer (TLB)* which is part of most modern processors, caches virtual to physical address translations. Because memory references tend to be local (i.e. to physically close memory blocks), caching is very effective (cf. section 2.2).

[1]The part of the CPU responsible for the memory translation is often referred to as *memory management unit (MMU)*. On modern CPUs the MMU is incorporated into the CPU.

2.4.1 Page Tables Hierarchies

The most simple page table hierarchy is simply a virtual address consisting of a pointer to the page table and an offset, the CPU can calculate the physical address by adding the address in the page table entry (PTE) and the offset. The page table entry typically also stores control information, such as bits indicating if the page is currently in memory or if it is dirty (i.e. modified but not yet updated on the backing storage). The page table for each process has to be permanently in memory. Figure 2.5 depicts simple paging.

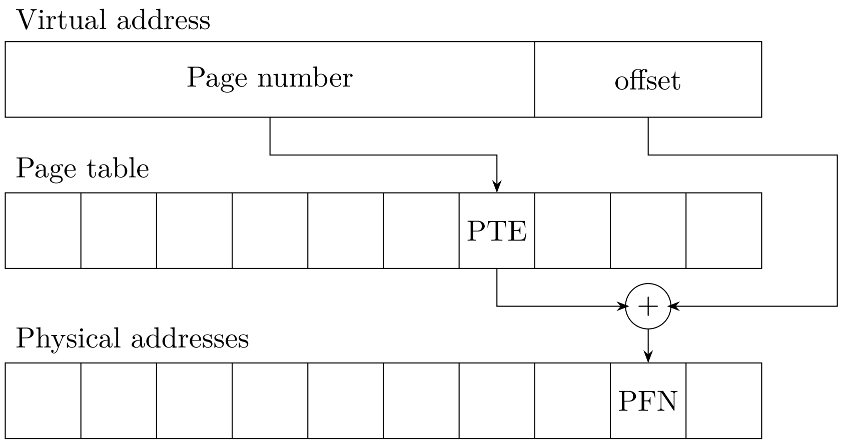


Figure 2.5: Simple paging, the physical address is calculated by adding the page table entry and the offset.

Current processors and operating systems extend the abstraction of virtual addresses, by employing several levels of page tables. Using page levels makes it possible to store page tables in virtual memory, consequently not all page tables have to be present in memory, because pages holding a page table can be paged out like any other page. Typically, operating systems lazily generate parts of page tables when they are required. Lazy generation of page table is one a core precondition of many Rowhammer attacks (see section 4.2).

Current x86_64 processors support four page levels [8]. A virtual address consists of 48 bits, of which the first nine bits (39-47) point to the top level page table, the next nine (30-38) point to the third level, bits 21-29 to the second level, bits 12-20 point to the page table entry and finally the least significant bits (0-11) are the offset. In Linux the top most level is termed *page global directory (PGD)*, the fourth level *page upper directory (PUD)*, the third *page middle directory (PMD)* and the second *page table entry (PTE)*. Figure 2.6 depicts the four-level page hierarchy of the x86_64 architecture using the Linux kernel parlance.

Other operating systems such as FreeBSD use different terms [9]. The x86 architecture only supports a three level page hierarchy, to support that the Linux kernel skips the PUD and the PMD level on these architectures. The following table summarizes the

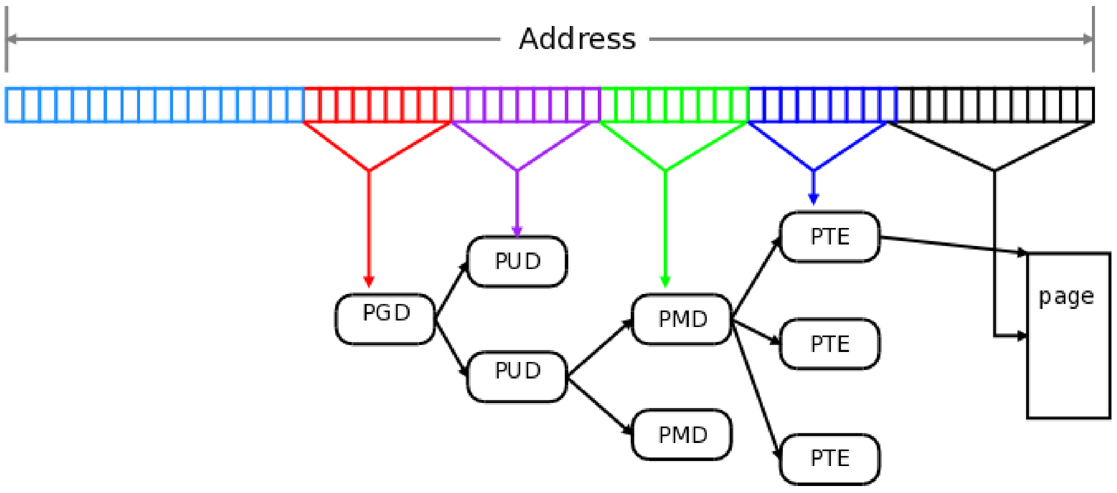


Figure 2.6: Four-level page table hierarchy of Linux for a x86_64 processor. Graphic by Jonathan Corbet `https://lwn.net/Articles/717293/`

page tables levels as for the Linux kernel. For the sake of completeness it important to mention that this summary does not mention page address extension (PAE), a feature of some x86 processors, allowing to use more than 32 bit (the processors word length) for addressing memory. Using more than 32 bit makes it possible to use more than four gigabyte of memory, however 64 bit CPUs do not have this limitation.

		Bits used				
Level		PGD	PUD	PMD	PTE	page
Architecture	x86_64	39-47	30-38	21-29	12-20	0-11
	x86	22-31			12-21	0-11

Table 2.1: Bits used for four-level paging by the Linux kernel

During a context switch the operating system stores the address of the top most table associated with a process to a special register, so the CPU can transparently can translate the addresses, on the x86_64 architecture this is the `cr3` register.

In 2017 Intel announced CPUs supporting five-level paging, using 52 address bits [10]. Even though no hardware is available yet, the Linux kernel already implemented support for five-level paging [11]. Currently, the top most level works transparently as the top most level is a simple pointer to the fourth level.

2.5 Huge Pages

At hardware level only pages of a fixed size exist. On the x86 and the ARM architecture this is typically 4 KB. Recent CPUs support larger pages (known as huge pages), which can be used to allocate large amounts (e.g. up to 1 GB on ARM) of consecutive memory, with only one page. Huge pages can be requested from the kernel by user land applications using the `mmap()` or the shared memory systems calls `shmget` and `shmat`. The purpose of huge pages is to save management information (especially in the TLB buffer) and to reduce complexity, because a huge page can be addressed with one single address. Because Rowhammer based attacks, aiming to flip bits at certain locations, such as Phys Feng Shui require a page size as small as possible to be able to control the bit to be flipped, huge pages are not discussed further in this thesis.

2.6 Page Cache

As of Linux 2.4 there is a unified cache for all block device I/O. Caching block devices is beneficial for two reasons: Pages can be ordered, so the operating system can write them back to disk more efficiently. Because memory references tend to cluster, it is very likely that pages can be served from the page cache again. Other operating systems provide similar features [6].

Because data in the page cache mostly consists of pages, that can be regenerated from secondary storage at any time, as much unused memory as reasonable is used for this cache. For this reason the Linux kernel does not count the page cache to the amount of used memory. It is important to mention that this behavior is Linux specific, as other operating systems do their I/O caches to memory statistics. For instance, FreeBSD displays much less free memory as Linux, because it counts block device caches to used memory [12].

2.7 Kernel Space And User Space

Most modern processors support at least two modes of operation: the *user mode*, in which the processor executes normal user application and the *kernel mode* in which the operating system runs. The kernel mode typically has full access to all memory locations and can use all instructions. When a system call is performed the operating systems is responsible to set and remove a control bit indicating the mode in which the processors runs. When a user mode application tries to execute an instruction or access a memory location that is not allowed in user mode, an interrupt is raised [6].

2.8 Rowhammer

At hardware level dynamic random-access memory (DRAM) is organized in two dimensional arrays of memory cells. A memory cell stores one bit of information in a capacitor, whose charge state represents a binary value. Memory banks group memory cells in rows and columns, individual rows and columns can be accessed via an access transistor (cf. section 2.1).

Because capacitors loose their charge, they have to be periodically refreshed at least every 64 milliseconds according to the DDR 3 specification [13]; this refreshing operation is done row-by-row.

The memory cells laid out so densely, that disturbance errors may occur when neighboring cells are accessed. Disturbance errors normally do not occur under normal circumstances and not every memory module is vulnerable. Susceptibility to disturbance errors depends on various properties, such as manufacturer, manufacture date, or the refresh rate of each specific memory module.

Often disturbance errors can cause interference among memory cells, in a way that a bit changes its value; this change is called a *bit flip*. Even though not every memory module is vulnerable, tests showed that about 85% of DDR3 memory modules are vulnerable [4]. It is important to mention that newer DDR4 memory modules are also vulnerable to bit flips [5].

Kim et al. [4] demonstrated that bit flips in memory cells can be triggered by performing a large amount of consecutive reading operations, on physically neighboring memory rows. This process is known as *Row hammer* or *Rowhammer*. The literature also refers to it as *rowhammering*, *row hammering* or *hammering* a memory cell. These terms were introduced by a patent, assigned to Intel [14] and are now common terms.

As not every memory cell is vulnerable to row hammering, attackers first have to search for vulnerable memory cells. Once a cell vulnerable to a bit flip has been found, it is very likely, that it can be repeated later. Repeatability is a big advantage for exploiting bit flips for privilege escalation attacks: if attackers succeed to place critical access control data (such as user credentials etc.) at memory locations where they can flip a bit of this critical data and eventually increase their privileges though.

Even memory error correction-codes (ECC) do not reliably protect from this issue. Kim et al. state:

> While most words have just a single victim, there are also some words with multiple victims. This has an important consequence for error-correction codes (ECC). For example, SECDED (single error-correction, double error-detection) can correct only a single-bit error within a 64-bit word. If a word contains two victims, however, SECDED cannot correct the resulting double-bit error. And for three or more victims, SECDED cannot even detect the multi-bit error, leading to silent data corruption. Therefore, we conclude that SECDED is not failsafe against disturbance errors.

To be able to hammer a memory row it is necessary to bypass all CPU caches, because otherwise the data form the target cells would be loaded from these caches when they are frequently accessed. There exist several techniques for bypassing CPU caches, they are discussed in section 2.9.

Hammering one row to introduce bit flips in a neighbor row is known as *single-sided row hammering.* The chances to cause bit flips increase if not only one neighboring memory row are hammered, but if both neighboring cells are hammered. Hammering from two sides is a second variant of Rowhammer known as *double-sided Rowhammer.* A third variant for some memory modules that allow to keep the connection to a memory row open was presented by Gruss et al. is *one-location hammering.* We introduce these three variants in the following.

The ability to cause reproducible bit flips itself, is not sufficient for performing a serious attack (expect data attacks aiming to corrupt Chances to cause a bit flip in critical data are relatively low, especially as many computer systems implement some sort of checksums for critical data. Attackers usually scan the whole available memory until they find one or more bit flips, with suitable alignment for their need. Then they trick the operating system to place critical data at a memory location vulnerable to a bit flip. Several techniques exist to trick operating systems to place critical data (such as user credentials) into memory rows vulnerable to a bit flip. They are discussed in section 4.2.

2.8.1 Single-sided Rowhammer

Attacking memory cells by row hammering one neighboring memory cell is known as single-sided row hammering. Contrary to what the name suggests, this technique can attack multiple cells at once; however hammering is always performed on one neighboring cell. Dullien and Seaborn successfully induced bit flips in neighboring cells by hammering four or eight locations at once [15].

Listing 2.1 shows the x86 assembly code used by Kim et al. [4] to perform single-sided Rowhammer attacks based on two memory locations, X and Y. To successfully perform bit flips the addresses X and Y must point to different memory rows in the same memory bank. The first line defines the label `code1a` marking a location to which the code can later jump (i.e. return) to; the label therefore marks the begin of a loop. Line 2 moves reads the content from address X to the `eax` register. Similarly, line 3 reads from address Y to the `ebx` register. Line 4 and 5 force the CPU to flush the data stored in X and Y from CPU caches. The last line jumps back to the label `code1a` in line 1 and the code is executed again, beginning with line 2.

Because the CPU cache is emptied by the `clflush` instruction in every iteration of the loop, the data stored in X and Y is always read from the RAM (i.e. X and Y are hammered), possibly more often than the refresh cycle of a capacitor storing a single bit.

A big advantage of singe-sided row hammering is that no knowledge about the underlying memory architecture is required, in order to successfully flip bits in neighboring memory rows, because each memory row has at least one neighbor.

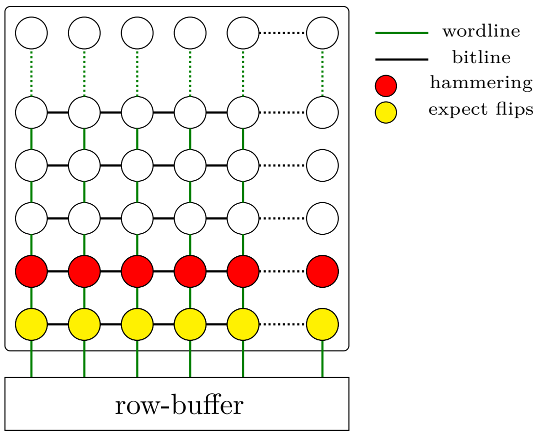


Figure 2.7: Single-sided Rowhammer is the most simple Rowhammering technique. Its advantage is that it does not require knowledge abort the inner memory structure.
.

```
code1a:
 mov (X), %eax // Read from address X
 mov (Y), %ebx // Read from address Y
 clflush (X)  // Flush cache for address X
 clflush (Y)  // Flush cache for address Y
 mfence
 jmp code1a
```

Listing 2.1: The x86 assembly used to perform single-sided row hammering

2.8.2 Double-sided Rowhammer

Double-sided hammering is the Rowhammer technique with the highest rate of successful bit-flips. Because, both neighboring memory rows are hammered the changes of successfully flipping a bit in a victim row increase. Hammering from both neighboring cells requires precise knowledge about the physical geometry of the hammered memory module. Attackers must be able to allocate exactly both neighboring memory rows of the victim row.

Seaborn and Dullien [15] found that a memory row of 256 KB is effective for several laptops of one vendor they tested. This means that for inducing bit flips in a vulnerable row an attacker must succeed to allocate 256 KB of memory physically above and below of the target cell. The authors suspect that this value is different for other hardware vendors. Figure 2.8 depicts a double-sided row hammer attack. Double-sided row hammering is the hammering technique producing the most bit flips.

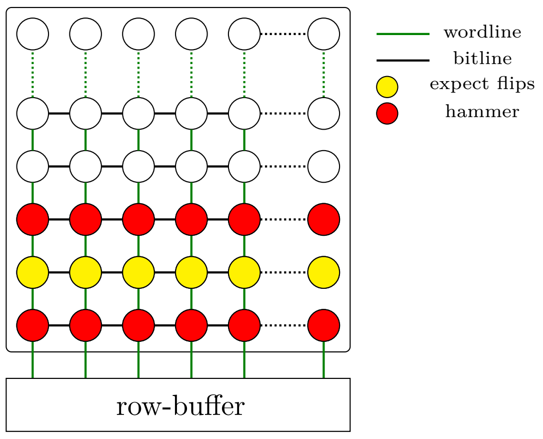


Figure 2.8: Double-sided row hammering induces bit flips by performing reading operations from both physically neighboring memory rows. This technique induces more bit flips than single-sided Rowhammering and One-location Hammering
.

2.8.3 One-location Row Hammering

One-location row hammering makes use of performance optimizations of newer memory controllers, which close the connection between row and a row buffer earlier than necessary. Repeatedly reading (i.e. hammering) a single address causes the row in which the data the address points to is stored, to be contiguously re-opened. Frequently, re-opening memory rows influences the charge of memory cells sufficiently to cause bit flips [5]. As a consequence, the one-location row hammering technique is not based on inducing interference among memory rows, like single sided- or double sided row hammering, but on effects caused by repeatedly opening a cell. The differences to the other row hamming variants are that only a single address is hammered, not one or two memory rows, and that any memory cell in the hammered bank can be subject to bit flips. Like in the other two row hammer variants, bit flips are reproducible, once they are found. Gruss et al. successfully performed one-location row hammering on DDR3 and DDR4 memory modules [5]. Similar to singe-sided row hammering this, one-location row hammering does not require knowledge about the underlying memory geometry; to cause bit flips it is sufficient to hammer a random location.

Figure 2.9 shows where bit flips can be expected when performing one-location hammering. A big advantage of one-location hammering is that Rowhammer defenses based on observing memory patterns are not able to detect the hammering, because only one location is constantly accessed, which is a common and inconspicuous operation [5]. One-location hammering is the least effective row hammering method, as is produces less bit flips than the other two methods.

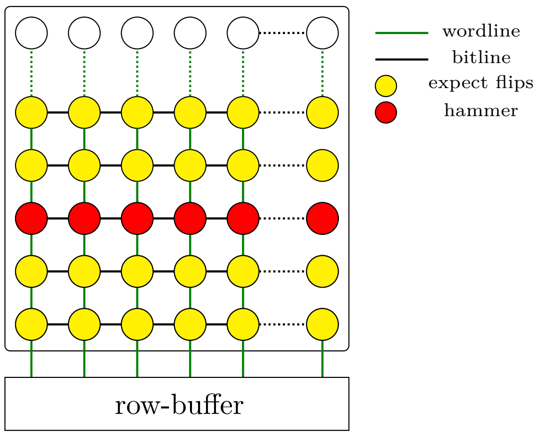


Figure 2.9: One-location hammering can induce bit flips not only in neighboring memory rows, making it a powerful row hammering technique [5]. The bit flips are not caused by row interference but by effects caused from repeatedly re-opening a memory location
.

2.8.4 Bit flip Attacks Before Rowhammer

The potential exploitation of bit flips was known long before Kim et al. [4] revealed the Rowhammer bug. In 2003 Govindavajhala et al. [16] presented an attack allowing a specially crafted Java application to take full control over the Java Virtual Machine [2] or Microsoft's .NET [3] virtual machine, once a bit flip occurs. Even though bit flips are rare, attackers can exploit them if the circumstances allow them to run their attack for a long period of time. They state that in 1996 one bit flip per month is to be expected on a end-user PC. 70% of the flips, occurring in rare events such as hardware defect or influence of cosmic rays are suitable for their attack.

2.9 CPU Cache Eviction Strategies

Modern processor architectures cache frequently accessed data in smaller but faster cache memory (see section 2.2). Because Rowhammer attacks need to repeatedly perform reading operations directly from memory, attackers need to circumvent processor caches to cause bit flips in a memory module. Circumvention can either be done by *flushing* or *evicting* data from the cache. Flushing means to force removal of the data block from cache by using a CPU instruction. Eviction means loading other data into the cache until the CPU replaces the target data block in course of normal operation. Both methods make it necessary, that the CPU loads the data from main memory again if it is accessed subsequently. Loading data from memory triggers a read operation. By repeatedly evicting and reloading data bit flips can be triggered, as explained in section 2.8. On

[2] https://www.java.com
[3] https://www.microsoft.com/net

some platforms the operating systems memory management provides direct uncached memory access, in such cases the cache does not need to be circumvented for flipping a bit in memory. In the following we discuss various techniques for flushing data from CPU caches as well as tricks for evicting data backed by victim memory rows from caches.

2.9.1 Flushing CPU Caches using CPU Instructions

The easiest way to circumvent CPU caches is using special instructions that empty it. x86 processors supporting the SSE2 [4] (that is virtually any modern x86 compatible CPU) instruction set extension, provide the `clflush` instruction, allowing user-level code to flush variables from CPU cache. Repeatedly reading from a memory location and flushing the cache afterwards can cause a bit flip (cf. section 2.1).

2.9.2 CPU Cache Eviction using Memory Access Patterns

On platforms where no instructions to flush CPU caches exist, special methods to circumvent the CPU caches are necessary to cause repeated read operations directly from memory. All cache access pattern based eviction strategies aim to find a pattern that when loaded by the CPU, reliably replaces the data backed by the target cell (i.e. a memory row that is row hammered) in memory. Therefore, subsequent instructions using the data in the target cell need to reloaded it from main memory. Repeatedly eviction and reloading often causes many fast, consecutive reading operations and can therefore be used for row hammering [18] [19].

Even though generic eviction strategies exist (see section 2.9.5, most of them require exact knowledge about the replacement algorithm the victim CPU uses to replace data in cache slots. The replacement algorithms usually replace lease frequently used slots, however they do not exactly employ a least recently used strategy [18]. CPU manufacturers usually do not publish documentation on the cache replacement strategies they employ. Nevertheless, it is possible to observe the CPU cache's timing behavior and draw conclusions about the employed eviction strategy. Based on these observations an *eviction set* can be created. An eviction set is a set of memory locations that, when loaded in into CPU cache, in the right order, replace specific data from its cache slot. The order an eviction set has to be accessed, in order to remove a page is called *access pattern.*

An eviction set in combination with a suitable access pattern can remove data from the CPU cache reliably. Once attackers have crafted a suitable eviction set and a suitable access pattern, they can remove data of their choice from CPU cache. Repeatedly evicting and reloading the target data by performing read operations, causes repeated read operations from main memory. This is fast enough to row hammer the target a memory row and consequently induce bit flips.

Gruss et al. define four types of eviction strategies [19]:

[4]SSE 2 stands for Streaming SIMD Extensions 2, it is an extension to the standard x86 instruction set. [17]

- *Static eviction set and static access pattern:* Eviction set and access pattern are pre-computed prior to performing the actual attack. Pre-computing requires precise knowledge about the microarchitecture used on the target system, but no searching for necessary parameters on the victim machine. This approach is similar to the eviction presented by Awke et al. (see section 2.9.3.

- *Dynamic eviction set and static access pattern:* Based on knowledge about the replacement strategy of the victim system attackers pre-compute an access pattern and generate eviction sets at run-time. This method is very efficient for attacks as pre-computation of access patterns, needs to be done only once for every target architecture. Various algorithms exist for finding static access patterns [20] [21] [19].

- *Dynamic eviction set and dynamic access pattern:* Brute-force search for eviction set and access pattern at run time. This method comes at the cost of a long run time but has the advantage, attackers do not need to know the target system and consequently allows wide ranging, fully automated attacks.

- *Static eviction set and dynamic access pattern:* Using a pre-defined evicting set and a randomly generated access pattern. According to the authors this has no advantage over randomly testing static access patterns.

2.9.3 Cache Eviction based on Knowledge about the target Microarchitecture

Aweke et al. [18] created an eviction strategy for Intel Sandy Bridge processors, by loading arbitrary addresses and subsequently searching for other addresses the CPU maps to the same cache slot. Reverse engineered documentation about the Intel Sandy Bridge microarchitecture provided them with information about the exact replacement strategies and allowed them to craft effective precisely access patterns. The exact functioning requires a detailed discussion of the microarchitecture and is therefore out of scope of this thesis. However, the basic idea is similar to the generic eviction strategy presented by Gruss et al. (see section 2.9.5).

2.9.4 Direct Memory Access on Android Devices

The ARM architecture does not provide an instruction to evict the CPU caches, like `clflush` does on x86_64. To evict page from CPU caches Van der Veen et al. constructed cache eviction for ARMv7 and ARMv8 CPUs, similar to the method presented by Aweke et al. [18] (see 2.9.3), however it turned out that this technique is too slow for practical use [22]. Consequently, page cache eviction on ARM is not practical for Rowhammer attacks. However, it is possible to gain direct, uncached memory access on this architecture. With Android 4.0 ICS (Ice Cream Sandwich) Google introduced the ION memory manager [23], a memory pool manager for the Android kernel (which is basically a modified Linux kernel), allowing unprivileged applications direct memory access without caching, via the `/dev/ion` device. Consequently, ION can be used to hammer memory cells and thus to induce bit flips [22].

2.9.5 Generic Cache Eviction Strategies

Gruss et al. developed a platform independent generic way to find eviction strategies based on memory access patterns. The key idea is to constantly access physically neighboring memory cells (e.g. by iterating over an arrays). Finding an optimal eviction strategy is not feasible in reasonable time, therefore Gruss et al. reduced the search space using heuristics [19]. As this strategy allows to search for eviction strategies independent form the execution environment it enables attackers to evict data from cache slots from high-level run time environments such as JavaScript [19]. The authors presented two ways to determine eviction sets and access patterns suitable to hammer memory rows from JavaScript run time environments of web browsers.

CHAPTER 3

State of the Art

This chapter summarizes defense strategies for attacks based on Rowhammer induces bit flips. First we introduce B-CATT [24] and G-CATT [25] two mechanisms preventing bit flips by searching for vulnerable memory locations in advance and subsequently prevent access to them. In section 3.3 we explain Kernel Page-Table Isolation [26] (formerly known as *KAISER*), a method to separate kernel page tables from user land page tables. This separation makes it impossible for attackers to row hammer kernel memory. Furthermore, section 3.5 introduces *ANVIL*, a strategy to detect Rowhammer attacks and consequently stop the attackers' process. Section 3.4 presents *MASCAT* an approach detecting patterns of Rowhammer attacks in executable binaries. Finally in section 3.5.1, we provide a brief overview about proposed hardware-based defenses.

3.1 B-CATT

The principal idea of *B-CATT*[1] is fairly simple, before the boot loader loads the operating system, it scans the whole memory for locations vulnerable to bit flips, that means it row hammers the whole memory. Brasser et al. implemented B-CATT by adapting the well known GRUB2 [2] boot loader. During the boot process the firmware (typically BIOS or UEFI [3] on newer machines) reports the areas of available memory to the boot loader, which reports it to the operating system. Before GRUB reports the operating system about the available memory blocks, B-CATT scans the entire memory for bit flips and reports only non-vulnerable memory blocks to the operating system. Therefore, this defense is fully transparent to the operating system and consequently does not require any modifications on operating systems. Being an operating system independent defense

[1]Brasser et al. explain the name as following: "The name B-CATT is composed of two parts: B refers to our **B**ootloader based solution, CATT abbreviates **CA**n't **T**ouch **T**his."

[2]`https://www.gnu.org/software/grub/`

[3]`https://www.uefi.org/`

is the biggest advantage of B-CATT. The authors reported a minor memory overhead of less than one megabyte on their testing systems. Furthermore, they did extensive benchmarks using various benchmarks suites and could not detect any measurable impact on the systems performance.

B-CATT is considered an effective, but unpractical defense for Row hammer attacks. Kim et al. [4] observed that 95% of the main memory would be blocked by B-CATT in realistic scenarios, Gruss et al. verified this result [5] Moreover, B-CATT requires row hammering the entire memory each time the system boots. Frequent row hammering is problematic as it may cause permanent physical damage of memory modules.

3.2 G-CATT

G-CATT (*Generic CAn't Touch This*) is a further development of B-CATT (see section 3.1) by the same authors [25]. The defense is similar to its predecessor, a low-level defense against Rowhammer attacks, however unlike its predecessor G-CATT does not require row hammering during boot-time and is implemented in the operating systems, physical memory allocator rather than the boot loader. The defense does not prevent row hammering, but aims to limit the negative consequences of Rowhammer based attacks.

G-CATT divides the physical memory in two security domains: One for kernel memory and one for user space memory. Hence, memory storing kernel memory (such as page tables) can only be allocated in predefined physical memory areas. Using pre-defined memory areas thwarts Row hammer attacks such as Phys Feng Shui (see section 4.3 for an explanation of the attack).

Brasser et al. implemented a prototype for Linux, due to the generic architecture of the Linux kernel, the prototype is architecture independent, its authors successfully thwarted Rowhammer based exploits against ARM and x86_64 platforms. To study the performance impact the authors did a huge amount of benchmarks and concluded that G-CATT has no significant impact on the system's computational power or memory usage. Even though the defense has no performance impact, it can only mitigate Row hammer attacks targeting the kernel memory. Gruss et al. presented a Rowhammer based privilege escalation attack targeting user space programs [5] Furthermore, in early 2018 Cheng et al. [27] exploited the Rowhammer bug in the presence of G-CATT using special purpose memory (e.g. video buffers) that is shared between user space and kernel space.

3.3 Kernel Page-Table Isolation

The idea of Kernel Page-Table Isolation (KPTI) [4] is similar to G-CATT (see section 3.2): kernel space and user space memory is separated. The difference is KPTI does not physically separate kernel memory and user space memory; it rather uses separate page tables for kernel space and user space. This separation makes hammering kernel memory impossible, as page tables of user space applications are stored in separate physical locations; hammering attacks requires them to be physically neighboring (see section 2.8. Before KPTI operating systems used the same top level page table for the kernel and user space applications. Using only one top level page table is problematic; there are at least three published attacks not related to the Rowhammer vulnerability [29] [26] based on that.

The basic principle of KPTI is quite simple, each time the context is switched (i.e. the processor changes the process it currently executes) from a user space process to a kernel process (or vice versa), the kernel changes the pointer to the highest page table (cf. section 2.4), to another independent page table hierarchy. However, the x86 architecture requires some parts of the kernel (e.g. interrupt handler routines and system calls etc.) to be mapped in the user space address space. KPTI patch minimizes these parts to the absolute minimum required by the hardware architecture. The Linux kernel developers included KPTI in kernel version [28] and backported it to the long-term support releases.

The authors of KAISER report a performance overhead of only 0.28% [26].

3.4 Static Binary Analysis

MASCAT [30] is a binary analysis tool to scan software binaries for unusual instruction patterns. As many low-level attacks use typical instruction sequences, harmless code can be distinguished from attack code. The tool is capable to detect a wide range of attack patterns, including side-channel attacks and Rowhammer attacks.

Single-sided and double-sided row hamming (see section 2.8 typically uses many cache flush instructions (e.g. `clflush` on the x86 architecture) in loops. Another indicator MASCAT uses to detect potential Rowhammer attacks is are instructions which trigger direct (i.e. uncached) memory access in loops (e.g. `monvnti` and `movntdq`). MASCAT does not only rely on these patterns to detect row hammering: it scans for unusual instruction patterns containing timing instructions such as `rdtscp` and `rdtsc`. These instructions are often used by timing side-channel attacks, which attackers may use to craft an eviction strategies, which can be later used to hammer a memory row (see section 2.9).

[4]KPTI is also known under the name KAISER (Kernel Address Isolation to have Side-channels Efficiently Removed), however the Linux kernel developers decided to name their implementation KPTI [28].

Gruss et al. [5] demonstrated that any static binary analysis can be circumvented by hiding attack code secure memory areas, such as SGX enclaves[5]. SGX enclaves allow user space applications to protect memory areas [31] by encrypting them. The CPU does the encryption completely independent of the operating system, therefore, even code running with maximum privileges [31] or in kernel mode, cannot access an enclaves' content. Currently, SGX enclaves are supported exclusively by recent Intel CPUs [31], however some AMD processors provide similar features [32].

MASCAT supports only scanning for instruction patterns and does not perform semantic analysis or code execution, hence it cannot guarantee that a certain pattern means an attack. Similarly, they cannot detect all kinds of attacks; skilled attackers knowing the target microarchitecture likely can circumvent static analysis tools, the authors of MASCAT state: "if an expert attacker knows the approach taken by our tool, he can always find a way to bypass it". For these two reasons, pure static analysis is a valuable tool but no ultimate solution for Rowhammer based attacks.

3.5 Rowhammer Defenses based on CPU Performance Counters

ANVIL is a Rowhammer mitigation technique aiming to detect and prevent single-sided and double-sided Rowhammer attacks. A kernel module monitors the behavior of all operating system processes; if the module detects potential row hammering it thwarts the attack by refreshing the relevant memory cells.

ANVIL counts the number of last-level cache misses, if the number exceeds a predefined limit, the defense kernel module instructs the processor to monitor which virtual memory addresses contain the code responsible to the increased amount of cache misses. Counting cache misses and evaluating the virtual memory addresses causing them requires special CPU support, typically this can be achieved by using performance monitoring features. Even though the proof of concept implementation of ANVIL relies on Intel performance counters [6], the authors point out that their defense can be implemented on any processor architecture providing such features; they state that this includes some AMD processors.

ANVIL makes use of the fact Intel performance monitor counters do not only report the number of cache misses, but also the source location from where a page was served. Consequently, the kernel modules can distinguish if a page was loaded from CPU cache or from main memory. If a page is repeatedly loaded from memory, even tough it is frequently used, ANVIL suspects that the page is used for row hammering (cf. section 2.8). As the mitigation is implemented as kernel modules it always runs in kernel mode and can therefore legally access the entire memory, thus it can access the process description data structures of the kernel (`struct task_struct` in Linux) and can

[5]`https://software.intel.com/en-us/sgx`
[6]`https://software.intel.com/en-us/articles/intel-performance-counter-monitor`

consequently figure out which operating system process causes the observed suspicious behavior.

ANVIL refreshes the suspected victim cell performing a reading operation on it. To do so it relies on information about how the RAM maps physical memory addresses to memory cells. In general RAM manufacturers do not publish documentation on this, therefore the authors of ANVIL relied on reverse engineered mapping information.

The defense has generally little overhead and no significant impact on normal applications, the authors report a peak overhead (measured by SPEC CPU2006), of 3.18% and 1.7% on average.

ANVIL is a reliable defense against Rowhammer induced bit flips, however it requires special CPU features, relies on internal memory module information which is generally not available and does not protect against one-location Rowhammer attacks [5].

The Linux Kernel developers discussed if a Rowhammer defense based on performance counters should be included in the kernel [33]

Run time detection of single-sided and double-sided row hammering using performance counters is also supported by intrusion detection systems such as HexPADS [34] and CloudRadar [35].

3.5.1 Hardware-based Defenses

Various researches proposed hardware-based defenses against Rowhammer based attacks. Most hardware defenses provide reliable protection against row hammering, however the enormous amount of vulnerable devices employed globally makes hardware replacement impossible.

- In the first paper describing the Rowhammer bug, Kim et al. [4] suggested *PARA*, a state-less, probabilistic Rowhammer defense for low-level memory controllers; Each time a row is opened or closed, one neighboring cell is also opened and therefore refreshed. During Rowhammer attacks the neighboring cells are opened and closed repeatedly, which finally can cause memory disturbances leading to bit flips (cf. section 2.8). When hammering memory row are opened frequently, PARA randomly refreshes neighboring memory rows, each time a row is opened. Because frequent accesses cause many random refreshes, the authors conclude that it is statistically guaranteed, that row hammering is mitigated, because the neighboring cells of the victim cell will be refreshed in time. To the best of our knowledge, PARA was not adapted by the industry.
- *ARMOR (A Run-time Memory Hot-Row DetectOR)* [36] is a further hardware solution eliminating the possibility to hammer memory rows. The authors suggest to add an additional cache to memory controllers. As a consequence hammering neighboring cells is prevented, since the data stored inf frequently accessed memory

cells is read from the hardware-cache rather than from memory modules, vulnerable to row hammering. Because, there is not software control about the hardware cache introduced by ARMOR the possibility to perform Rowhammer attacks is eliminated. At the time of writing this thesis, to the best of our knowledge no memory module on the market employs ARMOR.

- The LPDDR4 specification [37] defines two strategies against Rowhammer attacks.
  - *Target Row Refresh (TRR)* which introduces a counter for each memory row. Each time a memory row is accessed the counter of the neighboring cells is increased; Once the number of accesses exceeds a threshold the memory modules refreshed all neighboring memory cells.
  - *Maximum Activation Count (MAC)* sets an upper limit how often a cell can be accesses before it has to be refreshed.

Both methods prevent row hammering, as memory cells are refreshed before memory disturbances suitable for causing bit flips can occur.

3.5.2 Rowhammer related Firmware Updates

Several hardware manufacturers (e.g. HP [38], Lenovo [39]) and Apple [40] provided firmware updates doubling the refresh rates of memory cells. The DDR3 specification [13] requires memory modules to refresh memory rows at least every 64 milliseconds. Doubling the refresh rate makes (i.e. using refresh intervals of 32 milliseconds) makes bit flips less likely when attackers hammer memory cells, however it does not make them impossible. Aweke et al. [18] were able to perform double-sided row hammering, even with memory modules configured with a refresh interval of 15ms (i.e. more than four times of the required refresh rate). Furthermore, the authors state that increasing the refresh interval comes at the cost of increased power usage and reduced data throughput. For these reasons firmware updates are neither a satisfying nor a effective mitigation against row hammering. According to Corbet a refresh rate of 8ms is required to reliably prevent Rowhammer attacks [33].

CHAPTER 4

Design

While bit flips at random locations may cause serious malfunctions of computing systems and the data it processes, attacks become much more powerful if attackers can precisely control in which data they flip.

This chapter discusses the attacks based on the Rowhammer vulnerability, publicly known by the time of writing this thesis. First, we discuss the buddy system for physical memory management, as it is essential for understanding privilege escalation attacks as well as our proposed solution (see chapter 5). Then we discuss *Flip Feng Shui* and *Phys Feng Shui*, two attacks based on the deterministic nature of the buddy system. Furthermore, we discuss Linux kernel internals relevant for the prototype, we implemented. Finally, we provide a discussion of software diversity, as it is the fundamental concept used for our Rowhammer defense (see section 4.4).

4.1 The Buddy System

A fundamental task of operating systems is allocating physical memory to processes. Typically, operating systems do physical memory allocation in various different sizes. The buddy system is an algorithm to efficiently manage allocations of different sizes. Basically the algorithm splits memory blocks into halves until it finds a block of optimal size; similarly, it merges previously split blocks to larger ones. Because block sizes have to be divided by 2, their size is usually a power of two. Empirical experiments by Donald Knuth showed that the buddy system works very efficiently [41], he also states that the buddy system has a good memory balance as during his experiments no memory overflows occurred until 95% of the memory blocks were reserved.

A typical size for the smallest addressable memory block that can be addressed by hardware (i.e. a page frame) is 4096 (4 KB), older architectures may use different values [6]. Some architectures even support page sizes of different size 2.5. During memory initialization the memory initialization routine of the operating system divides the whole usable memory into blocks of size $2^n \times page\_size$. On Linux n currently defaults to 11, which results in memory blocks of size $2^{11} \times 4096$ bytes (8 MB), for $page\_size = 4096$. When the memory allocation subsystem processes a memory request, smaller than the maximum block size, it divides memory blocks in two parts until further division would result in a memory block too small for the memory request. The remaining half of a split block is marked as *buddy* of the block used to fulfill the request. When a block is freed it is *merged* (or *coalesced*) to a block of double size. After merging, the resulting block has the size of both buddies (i.e. $buddy\_size \times 2$). If the buddy of the resulting free block is also free, they are merged again. Merging continues until one buddy is reserved of the maximal block size is reached. If no free block of a certain size is available the next largest block is split.

Figure 4.1 depicts the functioning of the buddy system, using a maximum block size of 2048 KB $= 2^{11}$ in nine steps.

(1) Initially a free memory block of maximum size is available.

(2) The buddy system receives a request for a 512 block. It splits the 2048 block into two 1024 blocks; As a 1024 block is large enough for two 512 block, they are split into two 512 blocks. Without the last split operation one 512 block would remain unused.

(3) The system gets a request for a 256 block, as the smallest available block is the buddy of the 512 allocated in step 2, this block gets split into two 256 blocks.

(4) A process requested a 128 block, consequently the 256 buddy is split into two 128 buddies.

(5) The 256 block is freed. As the 128 block is still in use, no merging is done.

(6) Another 128 block gets allocated. As a suitable block is already available the buddy system immediately reserves it to fulfill the request.

(7) Both 128 blocks are freed. This triggers a several merge operations.

(8) Because both 128 blocks are free, they are merged to one 256 block.

(9) Since the buddy of the block resulting from the merging operation in step 8 is also free, they are merged to a 512 block.

Due to its high efficiency the buddy system is used in many general purpose operating systems [42][43].

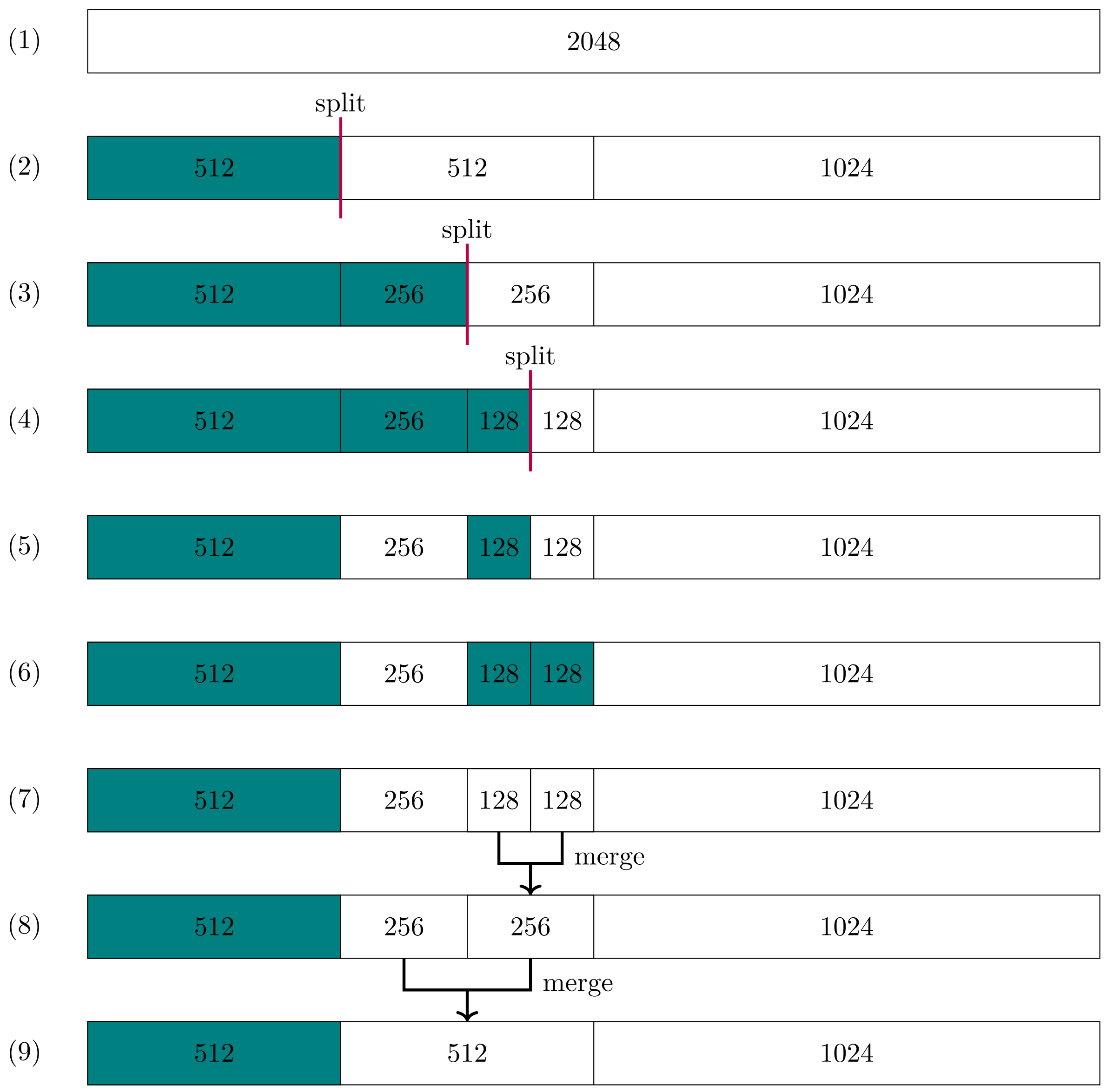


Figure 4.1: Example of the buddy allocation principle.

4.2 Rowhammer-based Attacks

This section gives an overview about state of the art attacks based on Rowhammer induced bit flips.

On a high-level all attacks based on bit flips can be divided into three steps.

- *Memory templating:* The attackers scan the entire memory for blocks vulnerable to row hammering.

- *Memory massaging:* The attackers force that the operating system stores critical data in a location where they can induce bit flips.
- *Exploitation:* The neighboring cells (which are always in the control of the attacker) are hammered and a bit flip in the critical data is induced).

4.2.1 Escaping the Sandbox of Google Chrome using Rowhammer

In 2015 Seaborn et al. published the first privilege escalation exploit based on the Rowhammer vulnerability. The exploit allows malicious web site operators to gain root privileges by escaping from the Native Client (NaCl)[1] sandboxing environment of the Google Chrome browser [2]. NaCl allows execution of native machine code instructions directly form the web browser. By using native machine code instructions attackers were able to hammer memory cells by using the `clflush` instruction (see section 2.8 for a detailed explanation).

Once attackers found a memory location vulnerable to row hammering, a second step is necessary to flip a bit at a critical location. They contiguously map the same file to memory using the `mmap` system call. This repeated mapping causes many second-level page table allocations (cf. section 2.4), therefore the whole memory is *sprayed* with second level page tables. This makes it likely that a page table is placed in a memory location vulnerable to memory bit flips. Flipping the right bit in a second level page table (i.e. the page middle directory (PMD)), changes the pointer of the page table entry (PTE, see section 2.4) such that it points to memory of the attackers process, figure 4.2 depicts this effect. As the PMD now points to memory under the control of the attackers they can control the memory addresses the PTE points to. Therefore, they can scan the whole memory for critical data. Seaborn et al. suggest to overwrite parts of executable binaries with can be executed by unprivileged users, but require system calls with root privileges [3] code to open a root shell (i.e. shell code).

To prevent row hammering from the NaCl environment Google disallowed the use of the `clflush` instruction [45]. However, Gruss et al. showed that memory attacks from web browsers are possible, even without using the `clflush` instruction [19] (cf. section 2.9.5.

4.2.2 Flip Feng Shui

Flip Feng Shui (FFS) [46] is an attack technique against virtualization servers hosting several virtual machines. Attackers can force data to physical locations where they can induce a bit flip. The attack exploits the fact that virtualization software typically store data that is equivalent on one or more virtual machines only once in physical memory.

[1] `https://developer.chrome.com/native-client`

[2] `https://www.google.com/chrome/`

[3] On Linux and UNIX operating systems, these are applications with the setuid bit set. setuid stands for set-user-ID and allows the executable to run with privileges different from the user who started it [44]

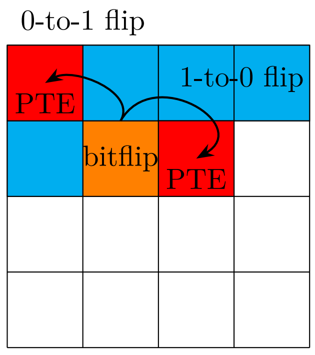


Figure 4.2: No matter in which direction a bit in the page table is flipped, it always points to attacker controlled memory.

Typically, virtualization software regularly checks for duplicate memory blocks, frees all but one of them and updates all reference pointers to the removed blocks, such that they point to the remaining block with identical data. This is a common situation, since operating system kernels, application software and libraries need to be mapped in the memory of each virtual machine. The more identical operating systems instances run on one physical host the more blocks are deduplicated. Naturally, attackers can use various techniques (such as operating system fingerprinting, or header information of services, etc.) to learn which operating system configurations run on virtual machines on the same physical host and install them on a virtual machine under their control. Once the attackers have setup a virtual machine identical one they aim to attack, they can assume that the virtualization software will deduplicate their memory blocks. As next step the attackers craft memory blocks identical to those known to be mapped in the victim virtual machines' memory. Finally, they hammer physically neighboring memory cells to cause bit flips. Because the memory block is only stored once, the bit is also flipped for the victim virtual machine.

Razavi et al. [46], presented a proof of concept implementation for manipulating SSH keys and domain names used to obtain software updates. When the attackers know about public SSH keys, a single bit flip is sufficient to make calculation of a private key feasible in many cases [46]. In particular several domains exists which differ by just one bit from `ubuntu.com`.

4.3 Phys Feng Shui

Phys Feng Shui [22] is a method for placing second-level page tables at a vulnerable memory location, similar to Seaborn et al. (see section 4.2.1). The difference is that attackers make use of the buddy allocator to force allocation of a second-level page table at a location vulnerable for bit flips.

Van der Veen et al. [22] implemented a proof of concept exploit for Android devices.

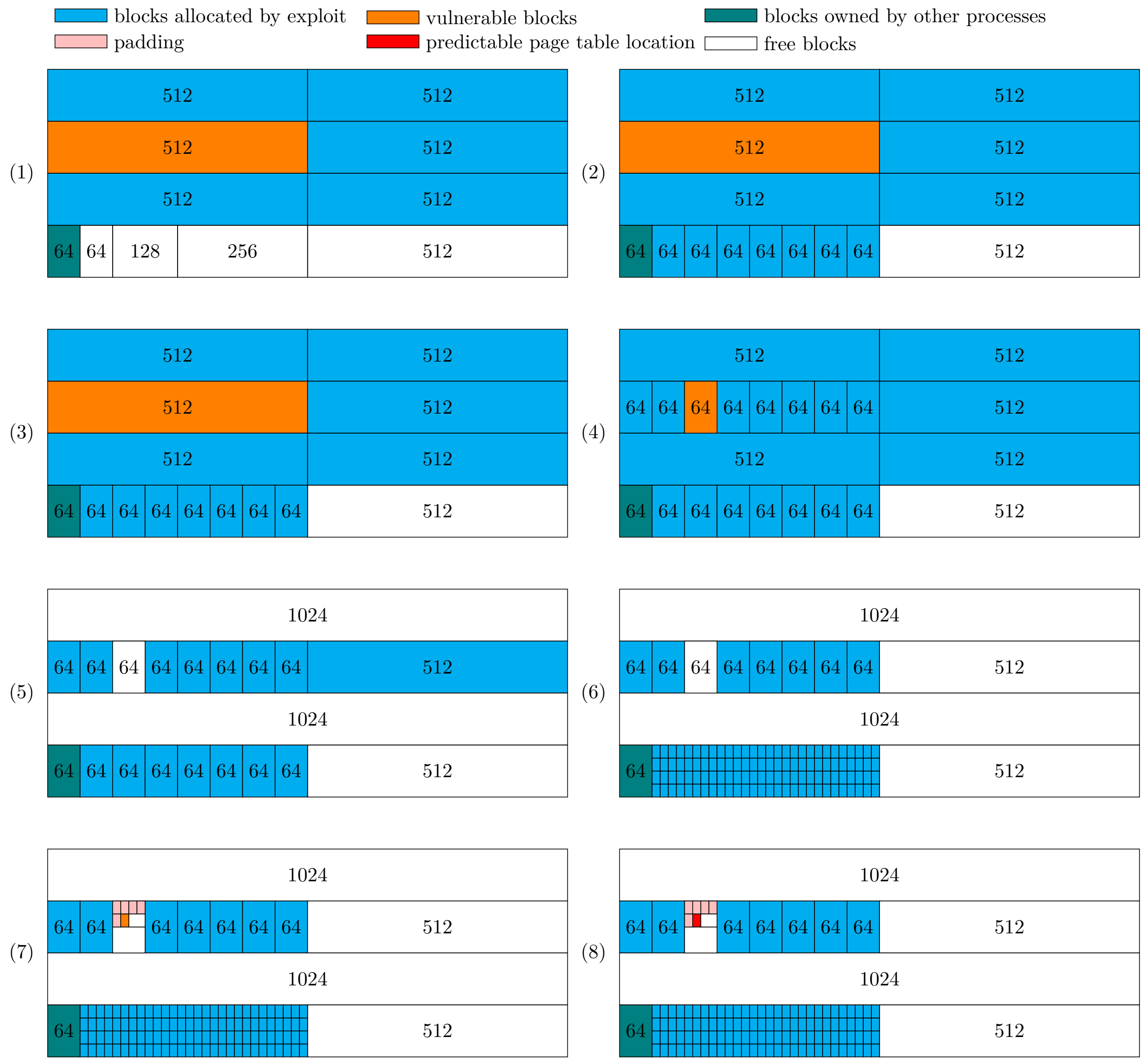

blocks allocated by exploit
vulnerable blocks
blocks owned by other processes
padding
predictable page table location
free blocks
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
512
1024
64
128
256

Figure 4.3: Example of Phys Feng Shui

Once their exploit has control over a PTE, they used it to scan the entire memory for the memory block containing the structure containing the credentials of the attackers process (`struct cred` on Linux). As they can read and write to the page (cf. section 4.2.1) they simply can increase their privileges and become root. The CPU's and operating system's access controls cannot prevent the attackers' process from raising its own privileges, because it just changes a page in memory allocated for it, which is a perfectly valid and common operation. The whole process of Phys Feng Shui is illustrated in figure 4.3.

(1) Exhauste the whole memory by allocating $512KB$ blocks. Probe each allocated

block for suitable, vulnerable bit flips.

(2) Allocate all remaining blocks (i.e. those smaller than $512KB$), by allocating 64 KB blocks. This step ensures, that the next $64KB$ allocation will be done in the vulnerable block.

(3) Free the vulnerable $512KB$ block, so it is the only free memory block of the system.

(4) Allocate eight $64KB$ blocks. It is guaranteed that they will be mapped to the vulnerable $512KB$ block.

(5) To avoid an out of memory state, immediately free all $512KB$ blocks. the buddy system coalesces them to $1024KB$ blocks. Then free the $64KB$ block, so it can be allocated in the next steps.

(6) Allocate $4KB$ (page size) blocks until the vulnerable $64KB$ block is used. The attacks can easily determine if the vulnerable block is used by reading `/proc/zoneinfo` and `/proc/pagetypeinfo`, both are world readable.

(7) Allocate $4KB$ until the $4KB$ block, containing the vulnerable bit flip is used.

(8) Trigger a page table allocation by calling `mmap(MAP_FIXED)`

4.3.1 Memory Waylaying

Memory waylaying is a reliably alternative method for placing attacker chosen memory blocks at vulnerable memory locations [5]. The method consists of two steps, the *Prefetch-based Prediction Oracle* telling attackers the location of data they want to row hammer, and repeated *Page Cache Evictions* to place the place the data at a location vulnerable to row hammering.

Prefetch-based Prediction Oracle

The first step tells attackers if two distinct virtual memory addresses map to the same physical address (i.e. page frame address). They can gather this information by a timing side channel-attack. To be able to exploit this side-channel, attackers preliminary need to measure the time *prefetch* instructions need to load memory blocks from each CPU cache level and the main memory. Prefetch instructions allow programmers and compilers to fetch to CPU caches prior to performing operations on them; Proper prefetching can drastically reduce cache misses and therefore increase performance. According to the authors Intel CPUs as well as ARM CPUs support prefetch instructions [5].

To learn the timing behavior of prefetch instructions for different cache levels, the attackers perform three steps. The trick is that the most processors load prefetched data blocks, *without* any access control.

The attackers first need to load interesting data to the processor caches. For instance, if attackers want to find out the address of an executable file, it is sufficient to map it to memory (e.g. by using the `mmap` system call on Linux or UNIX) and flush or evict a block p containing the binary from the CPU cache (cf. section 2.9). Then the attackers prefetch a random address $\bar{p}$ and reload the previously flushed block, by reading it. If the reload operation is fast (i.e corresponds with the load time for a cache level, learned in the first step) the attackers can conclude that p is loaded from cache with high probability. This is happens only if the page addresses by p is mapped by another application. The following list summarizes the three steps (cf. section [47].

1. Flush a address p (e.g. by mmapgin a file to memory)
2. Prefetch a *inaccessible* address $\bar{p}$
3. Reload p

On a typical system 10 million such measurements can be done. Therefore, attackers are able to probe the whole memory, i.e. then can sequentially use all memory addresses for $\bar{p}$ [47].

Gruss et al. call named step *prefetch-based prediction oracle.*

Alternatively, attackers can learn if two memory addresses map to the same physical one by performing a *Evict+Prefetch* side-channel attack. This attack is similar to the prefetch-based prediction oracle, the difference is that the attackers do not know $\bar{p}$, they learn that p is used by a system call or library function [47].

The first of the attack requires more effort, as the blocks are evicted from cache, rather than flushed; This means the attackers fill the processor cache until they can assume the CPU has removed the target block from cache (see section 2.9 for details). In the second step the attackers perform steps to load a target address $\bar{p}$ (e.g. by calling a library function or a system call). In the third they prefetch p and consequently measure the timing differences. The following steps summarize the Evict+Prefetch method (cf. section [47].

1. Evict a known address address p
2. Execute function or system call, that accesses an inaccessible address $\bar{p}$
3. prefetch p

Initially, Prefetch-based Prediction Oracle and Evict+Prefetch were invented to side-channel attack were developed to bypass ASLR. However, as they can be used to tell attackers if two memory addresses are physically neighboring, they help attackers aiming to a Rowhammer attacker [47] (see section 2.8).

Page Cache Eviction

Evicting contents of the page cache (see section 2.6) is especially interesting for attackers intending to perform a Rowhammer attack, as the page cache stores binary executable, and evicting them may allow attackers to place them memory locations vulnerable to bit flips. Page cache eviction must not be confused with CPU cache eviction (section 2.9), which removes data from CPU caches, page cache eviction evicts data cached in primary memory that was previously loaded from secondary memory.

The *Page Cache Eviction* procedure [5] fills the page cache (see section 2.6 and consequently evicts an executable file (e.g. the `sudo` binary). Because on Linux the page cache uses only memory which would remain unused otherwise (cf. section 2.6), this procedure is likely not to arise attention. For instance, the commonly known `free` tool telling users the amount of free memory available on the system does not count the page cache to the used memory.

Mapping very large files (several gigabyte) to memory and iterating over its content requires the kernel to remove other data from the page cache.

The `mincore` system call allows unprivileged to test if a page is currently cached in the page cache or not. This system call is also available on other UNIX-like operating systems such as FreeBSD and OpenBSD.

Memory Waylaying

The Prefetch-based Prediction Oracle and Page Cache Eviction combined can now be combined. This combination is known as *Memory Waylaying*Attackers need to repeatedly evict pages and check if a page in which they intend to flip a bit gets loaded to a location they can hammer. Because memory waylaying performs most operations on the page cache, it does not raise attention by increasing memory usage.

A disadvantage of the method is that it may take several hours or even days [5] until a page interesting for attackers gets loaded to a location where they can induce a bit flip.

Memory Chasing

An alternative variant of memory waylaying is *memory chasing*. To enforce frequent relocation of pages containing the victim binary, memory chasing exploits the copy-on-write nature of the Linux kernel's process management. Gruss et al. describe the procedure as following [5].

1. Map the whole victim binary using the `mmap` system call.
2. Invoke the `fork`[4] system call.

[4]The `fork` system call creates a child process with an identical process image, which the kernel copies once the child process performs write operations (copy-on-write). The system call is available on most UNIX-like operating systems.

3. Overwrite parts of the binary in the child process. Due to the copy-on-write nature of t ensures that the binary gets copied to a new physical page

4. Kill the parent process so the original unmodified pages are released

5. Repeat this until the victim page is placed at a vulnerable memory location (i.e. a location where a bit flip can be induced by row hammering). This can be checked by the prefetch-based prediction oracle described above.

Attackers now face the problem that with the procedure explained above the modified page (i.e. the page that child process wrote to) is placed at the vulnerable location, consequently attackers need to trick the kernel to place the original (i.e. unmodified) page from the binary at this location. This can be done by evicting the page from page cache using the page cache strategy explained above and immediately mapping the original binary again. Then the target binary is immediately released, the authors claim that this ensures that the same physical pages are used. However, they did not state a reason for this.

This approach is considerably faster than memory waylaying, however it has higher CPU usage and uses a lot of `fork` system calls, hence it can easier be easier detected by intrusion detection software.

Gruss et al. implement a proof of concept attack which flips a bit of the `sudo` binary. The flip turns `je` x86 assembly instruction turn into a `jne` instruction. Consequently another program branch is selected and the attacker can gain root access. In their analysis they figured out that the `sudo` binary has 29 locations where a bit flip can lead to a privilege escalation exploit [5].

This attack is an example for an user space to user space Rowhammer attack, it does not rely on to tricking page tables (which belong to kernel memory space) to vulnerable locations.

4.3.2 Page Types

The Linux kernel distinguishes different *mobility types* of pages, this distinction allows the kernel to group pages by their ability to be moved to other locations. This helps the kernel to store page belonging together in physically grouped locations. Storing memory pages grouped is beneficial due to the principal of locality (see section 2.2). Furthermore, grouped storage helps to minimize fragmentation effects [42].

As of Linux 4.13 three different page types exist:

- `MIGRATE_UNMOVABLE` pages of core kernel components, their fixed location is essential for the operation of the kernel.

- `MIGRATE_RECLAIMABLE` pages containing content that can easily be regenerated. A typically example for pages of this type are pages containing data stored on secondary memory, which can be loaded again at any time.
- `MIGRATE_MOVABLE` pages without special position requirements, they can be moved to any physically memory location at any time. For example, pages containing data of user land applications are always movable. When pages are moved, the kernel updates the entry in the page tables referencing them. This happens completely transparently for user land applications.

4.3.3 Page Lists

The buddy system (cf. section 4.1) implemented in the Linux kernel keeps lists of free pages for each combination of block order and migrate type. For example, there is a list of movable pages of order n, one for reclaimable pages of order n and one for non-movable pages of order n, and so on.

Often systems with more than one processor have special memory areas, which are faster accessible for specific processors. These systems are called *Non-uniform memory access (NUMA)* systems [42]. When an application is executed on a specific processor, the kernel can move them to a memory area faster accessible by the processor

Pages of order 0 are treated specially by the kernel, it organizes them in page lists for each order. The kernel users these lists for serving all memory allocations of order 0 (i.e. typically 4 KB). On non-NUMA systems (i.e systems with only one CPU) the kernel uses the same special lists only for one CPU.

The `rmqueue()` function (in the file `mm/page_alloc.c` is responsible for taking an appropriate page from a page list. If the function finds a suitable block, it removes a page from the respective list. Depending on how often the page was loaded into the CPU registers, `rmqueue()` takes the first or the last list element from the list. This is because the kernel guesses that a frequently used page (even if it is free), is still cached and consequently does not need updating the TLB (see section 2.4).

Allocations work similarly for (physically continuous) allocations of more than one page. As the kernel removes the an entry in the respective list, which represents the first page of the new allocation.

4.4 Software Diversity

The success of any Rowhammer attack depends on the attacker's ability to place a memory block at a location vulnerable to a bit flip (cf. section 2.8). If the memory placement strategy of the operating system places memory blocks in a manner, not predictable for user land applications the attacker cannot craft a reliable strategy to trick the operating system to place a memory block at a vulnerable location.

Software diversity is the idea to make the inner functioning of each copy of a software product as unique as possible, while the functionality remains equivalent. An example is a web browser, distributed over a website, each time an end user downloads a copy, the web server changes the control flow and variable locations of the web browser's binary, so each end user receives a different version of the binary. Therefore, an attacker cannot download the web browser and study its vulnerabilities and craft a large-scale automated attack by exploiting e.g. buffer overflow vulnerability, because the attack cannot predict where the memory area containing the buffer overflow is located. Naturally, a willing attacker who can access the binary version of the target system is — given enough time and persistence — will eventually be successful, however the crafted exploit will most likely not work on a similar system using the same web browser.

In his 1992 paper, Cohen [48] explained the motivation behind software diversity:

> The ultimate attack against any system begins with physical access, and proceeds to disassembly and reverse engineering of whatever programmed defenses are in place. Even with a cryptographic key provided by the user, an attacker can modify the mechanism to examine and exploit the key, given ample physical access. Eventually, the attacker can remove the defenses by finding decision points and altering them to yield altered decisions.
>
> Without physical protection, nobody has ever found a defense against this attack, and it is unlikely that anyone ever will. The reason is that any protection scheme other than a physical one depends on the operation of a finite state machine, and ultimately, any finite state machine can be examined and modified at will, given enough time and effort. The best we can ever do is delay attack by increasing the complexity of making desired alterations.

Software diversity is also an effective method against side-channel attacks [49], code reuse attacks [50].

As Rowhammer attacks are typically not done by an attacker with decent knowledge of the target system software diversity is an effective method against this kind of attacks.

The main idea of this thesis is to apply the ideas from software diversity to the memory allocator of the Linux kernel, in order to make it as hard as possible for an attacker to place a memory block at a location vulnerable to a bit flip.

CHAPTER 5

Design and Implementation of Page Sacrifice

In this chapter we describe how we implemented PAGE SACRIFICE, our proposed defense against attacks based on deterministic memory allocation. First, we give an overview about the core concepts, then we explain the functioning and effectiveness of our defense on the basis of *Phys Feng Shui* (explained in section 4.3). Finally, we discuss our prototype implementation for the Linux kernel in depth.

5.1 Overview

Basically PAGE SACRIFICE adds a random length padding both, before and after page frames and besides that it skips merging and splitting operations of the buddy system (see section 4.1). In detail padding and skipping operations work as following.

- **Adding padding:** Before the memory allocator reserves a page frame, it does a random decision; Based on this decision a page frame is either reserved or skipped. Skipping means that the kernel ignores this page frame, to fulfill the request for a free page frame it continues to either skip or reserve page frames until the decision is made to reserve a particular page frame or a maximum is reached. The kernel keeps track of which page frames were sacrificed while reserving each physical block. When the kernel frees page frames, it also frees all pages that were skipped while allocating it. Subsequently, we refer to skipped blocks are referred to as *sacrificed page frames* or simply *sacrificed pages*. Figure 5.1 illustrates a rather extreme case of PAGE SACRIFICE: A 128 KB block consisting of 32 $4KB$ pages without PAGE SACRIFICE all pages are used, with PAGE SACRIFICE employed only 11 are used and 21 are *sacrificed* pages, a sacrifice rate of ca. 66%

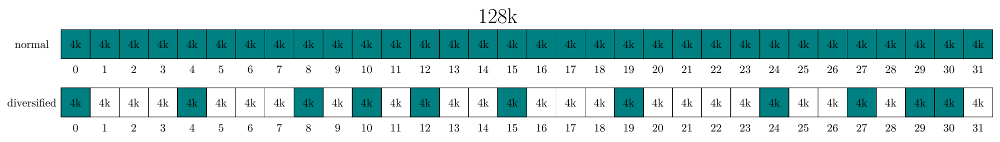


Figure 5.1: Witout PAGE SACRIFICE all blocks are deterministically allocated and used. With PAGE SACRIFICE not all pages are used, but their positioning is unpredictable in advance.

- **Skipping splitting and merging operations:** Whenever the buddy system splits a block into two buddies or merges them to a larger block, PAGE SACRIFICE makes a random decision if the operating is skipped.
  - **Skipping splitting operations:** For splitting operations this means, that the block the buddy system wants to split into smaller parts, keeps its physical size but subsequently uses it as if the skipping operation was done. This means that buddy system uses only parts of a physical memory block.
  - **Skipping merging operations:** In case of merging operations this means that previously split blocks are not merged and thus remain at their current size. The buddy system subsequently continues to works like if the unused pages do not exist. As both buddies keep buddies, the buddy blocks may be merges later, when one of the buddies gets freed.

All changes PAGE SACRIFICE does to the buddy system are depicted in figure 5.4.

5.2 Implementation

To prove the functioning of our idea, we adapted the buddy system of the Linux kernel. The choice for Linux was natural as it is the base of the Android operating system and the source code is available under a free software license.

5.3 Preliminaries

The Linux kernel needs to store information about each page frame (cf. section 2.4), the data structure for this is `struct page`. It is important to understand, that the order and alignment of `struct page` hardware defined. Therefore it cannot be modified without adapting large parts of the kernel, however it is possible to extended the structure by adding further data at its end. PAGE SACRIFICE makes use of that and extends `struct page` by a linked list used to store page frame that were sacrificed during allocation of a page in use. This list is later used to free sacrificed pages, when the page gets freed.

The random decisions, if a split or merge operation shall be skipped or how many pages shall be sacrificed is done with the help of the cryptographically secure kernel function `get_random_bytes()`. A buffer of N bytes is filled, to increase performance. Only the random bits necessary for a decision are used. For example, the kernel is configured to do skip a merge operation with probability $P = 0.5$, only one bit is used. The remaining bits of the buffer are right shifted. If less bits are available for the current decision, the buffer is refilled.

5.3.1 Implementation of Skipping Page Frames

The source code of the Linux kernels' physical memory allocation and the buddy system is implemented in several functions the `mm/page_alloc.c` file. The `rmqueue_pcplist` function is responsible for removing pages from the CPU specific list of free pages (cf. section 4.3.3.

The following code listing shows the function including our modifications.

```
static struct page *rmqueue_pcplist
(
struct zone *preferred_zone, struct zone *zone,
unsigned int order, gfp_t gfp_flags, int migratetype) {
...
local_irq_save(flags);
pcp = &this_cpu_ptr(zone->pageset)->pcp;
list = &pcp->lists[migratetype];

sacrifice_stats_total_alloc++;

sacrifice_pages(list, &(pcp->count),
&preceding_sacrifice);

page = __rmqueue_pcplist(zone,  migratetype, cold, pcp,
 list);

sacrifice_pages(list, &(pcp->count),
 &succeeding_sacrifice);

add_sacrifice_to_page(page, &preceding_sacrifice,
 &succeeding_sacrifice);
...
local_irq_restore(flags);
return page;
}
```

Listing 5.1: The modified kernel function for allocation of physical page frames. Note that variable declaration are skipped for simplicity.

- Line 8-12: Structures to store the beginning and the end of sacrificed pages are initialized.
- Line 13-15: Unmodified kernel code, `local_irq_save` disables interrupts for the current CPU. The following other two lines get a variable the list of free pages, associated with the current CPU.
- Line 19: Sacrifice page frames physically preceding the page that actually gets reserved for the request. The `sacrifice_pages` function does a random decision how many pages are sacrificed and removes them form the list of free pages.
- Line 22: Remove the page used to fulfill the memory request.
- Line 25: Similar to Line 19, we sacrifice physically succeeding pages.
- Line 28: Associates the sacrificed pages with the page that actually gets used.
- Line 33: Finally, interrupts are enabled again on the current CPU.

5.3.2 Implementation of Skipping Split Operations

The buddy system splits blocks into smaller blocks until it reaches a suitable block size (see section 4.1). Splitting is done by the `expand` function[1]

```
static inline void expand(
struct zone *zone, struct page *page,
int low, int high,
struct free_area *area, int migratetype) {
unsigned long size = 1 << high;
while (high > low) {
  area--;
  high--;
  size >>= 1;
  ...
  sacrifice_stats_total_splits++;
   if (high == 0 && make_random_sacrifice_decision()) {
        sacrifice_stats_splits++;
        return;
}
list_add(&page[size].lru, &area->free_list[migratetype]);
area->nr_free++;
set_page_order(&page[size], high);
} }
```

Listing 5.2: Skipping split operations happens in the expand function

[1]Note that the function name `expand` might be misleading, expand means to expand the amount of free blocks of a certain order.

The expand function uses a while loop to iterate to the lowest block order suitable for the allocation. If the variable high has value 0, the function splits two blocks of order 1 into two of order 0, we make a random decision if the skipping operation should be skipped or not. The rest of the function is kernel internal bookkeeping and not part of PAGE SACRIFICE.

5.3.3 Implementation of Skipping Merge Operations

When the kernel frees a physical page, it calls the __free_one_page function an checks, if the page has a buddy, it not it immediately continues freeing without merging, by jumping to the done_merging label. PAGE SACRIFICE reuses this label, according to a random decision the buddy blocks get merge or not. It is important to mention, that skipping merging operations, does not mean sacrificed memory, since the blocks can still be used.

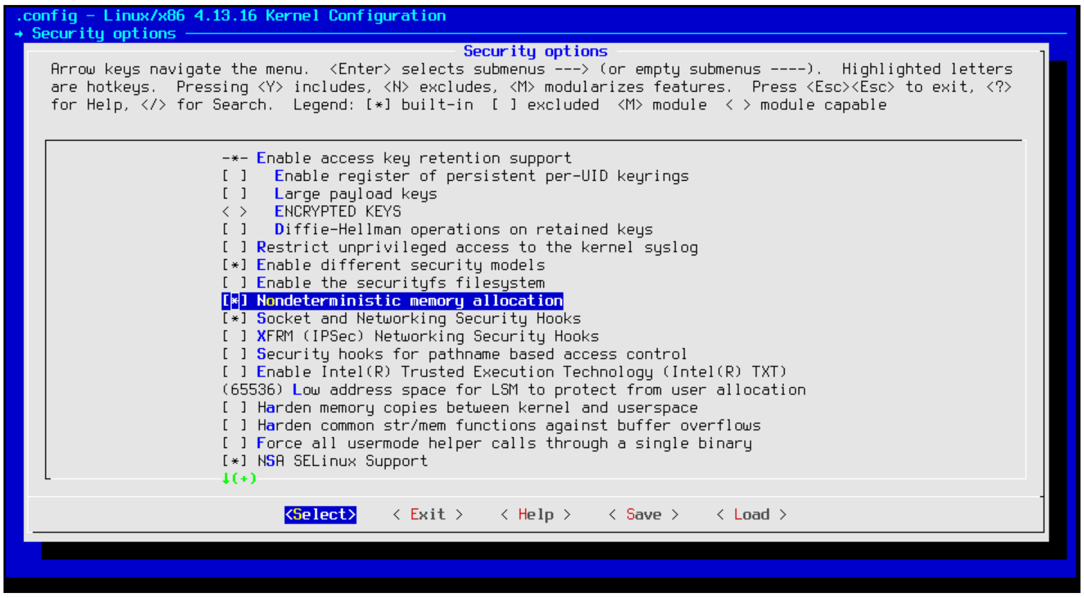
```
static inline void __free_one_page(struct page *page,
unsigned long pfn,
struct zone *zone, unsigned int order,
int migratetype)

...
if (!page_is_buddy(page, buddy, order))
  goto done_merging;


if (order == 0 && make_random_sacrifice_decision()) {
  sacrifice_stats_merges++;
  goto done_merging;
}
...
```

Listing 5.3: PAGE SACRIFICE skips merge operations in the ___free_one_page function

5.3.4 Configuration

Because the Linux kernel has a broad user base and sacrificing pages for higher security may not be an option for every use case, the feature and default parameters can be configured in the kernel configuration. Fine grained control at run time is also possible via the sysctl interface. Users can configure the following parameters:

- vm.page_sacrifice if set to a nonzero value PAGE SACRIFICE is enabled.



```
.config - Linux/x86 4.13.16 Kernel Configuration
→ Security options
                              Security options
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters
  are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?>
  for Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                    -*- Enable access key retention support
                    [ ]   Enable register of persistent per-UID keyrings
                    [ ]   Large payload keys
                    < >   ENCRYPTED KEYS
                    [ ]   Diffie-Hellman operations on retained keys
                    [ ] Restrict unprivileged access to the kernel syslog
                    [*] Enable different security models
                    [ ] Enable the securityfs filesystem
                    [*] Nondeterministic memory allocation
                    [*] Socket and Networking Security Hooks
                    [ ] XFRM (IPSec) Networking Security Hooks
                    [ ] Security hooks for pathname based access control
                    [ ] Enable Intel(R) Trusted Execution Technology (Intel(R) TXT)
                    (65536) Low address space for LSM to protect from user allocation
                    [ ] Harden memory copies between kernel and userspace
                    [ ] Harden common str/mem functions against buffer overflows
                    [ ] Force all usermode helper calls through a single binary
                    [*] NSA SELinux Support
                    ↓(+)

                 <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 5.2: page sacrifice can be enabled at compile time using standard Linux tools. Here the menuconfig is shown.

- `vm.page_sacrifice_max_order` highest order to sacrifice blocks. This is in particular useful, because sacrificing blocks of high order can cause a serious memory overhead which easily leads to out of memory situations.

- `vm.page_sacrifice_max_at_once` integer defining the maximum number of pages (or blocks if the order is greater than zero) sacrificed in one allocation. This defines the maximum size of a gap in memory.

- `vm.page_sacrifice_prob_bits` integer defining the number p of bits used for the random decision if a block shall be sacrificed or not. A bit is sacrificed if all p bits are set to zero. This means that each time a block is allocated it is sacrificed with probability of $\frac{1}{2^p}$. Accordingly, setting this parameter to 1 means: scarify with probability of $\frac{1}{2}$, to 2 means: $\frac{1}{4}$ and so on. Note that setting this parameter to 0 makes no sense, since the first page allocation is sacrificed until the whole memory is sacrificed.

5.3.5 Effectiveness of Page Sacrifice

When page sacrifice is deployed exploits such using Phys Feng Shui (cf. section 4.3 cannot assume predictable locations of page frames. Figure 5.4 depicts an attempted Phys Feng Shui attack prevented by page sacrifice. Additionally, determining if a coalescing or split operation is performed becomes impossible. Another advantage of page sacrifice is that pages are physically placed at a location far away of the usual. This is because the buddy systems does not know about the sacrificed pages, when a

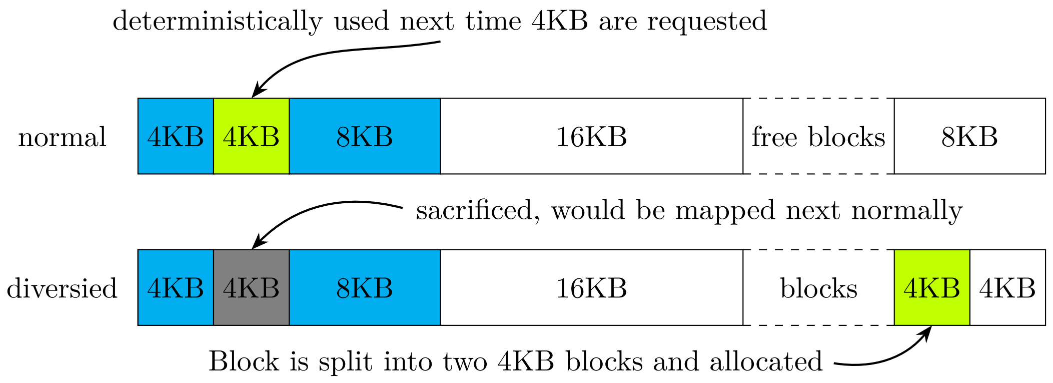

deterministically used next time 4KB are requested
normal
4KB
4KB
8KB
16KB
free blocks
8KB
sacrificed, would be mapped next normally
diversied
4KB
4KB
8KB
16KB
blocks
4KB
4KB
Block is split into two 4KB blocks and allocated

Figure 5.3: Often pages are stored at a physical location very different from where the original kernel would place it.

block is requested the buddy allocator assumes that pages are stored contiguously in memory. As this is not the case with PAGE SACRIFICE pages may be stored at memory locations hard to predict. Figure 5.3 illustrates this effect.

(1) Like in figure 4.3, the whole memory is filled by allocating $512KB$ blocks, here the split operation of the buddy system was skipped. So the physical $1024KB$ block is seen as $512KB$ block, by the buddy system.

(2) All remaining $64KB$ blocks are filled. During these allocation operations two $64KB$ blocks are sacrificed.

(3) This step is equivalent to what would happen without PAGE SACRIFICE. The $512KB$ block is freed.

(4) The exploit fills $64KB$ blocks until the previously freed $512KB$ block, containing the vulnerable page is filled. In this example two $64KB$ blocks are sacrificed.

(5) This step is very similar to what would happen, without our defense deployed: The vulnerable $64KB$ block is freed, and all remaining $512KB$ blocks are freed, to avoid an out of memory situation. The only difference is that in this example two $512KB$ blocks (depicted in the third row) are not merged to a $1024KB$ block.

(6) Like without the PAGE SACRIFICE technique, $4KB$ blocks are allocated, until the vulnerable $64KB$ block is used. While allocating a a large number of blocks of different size is sacrificed.

(7) The vulnerable $64KB$ block has been reached. The Phys Feng Shui inserts padding blocks until the vulnerable block is reached. Because, in this example two blocks are sacrificed, two more padding blocks are inserted, physically right behind the vulnerable block.

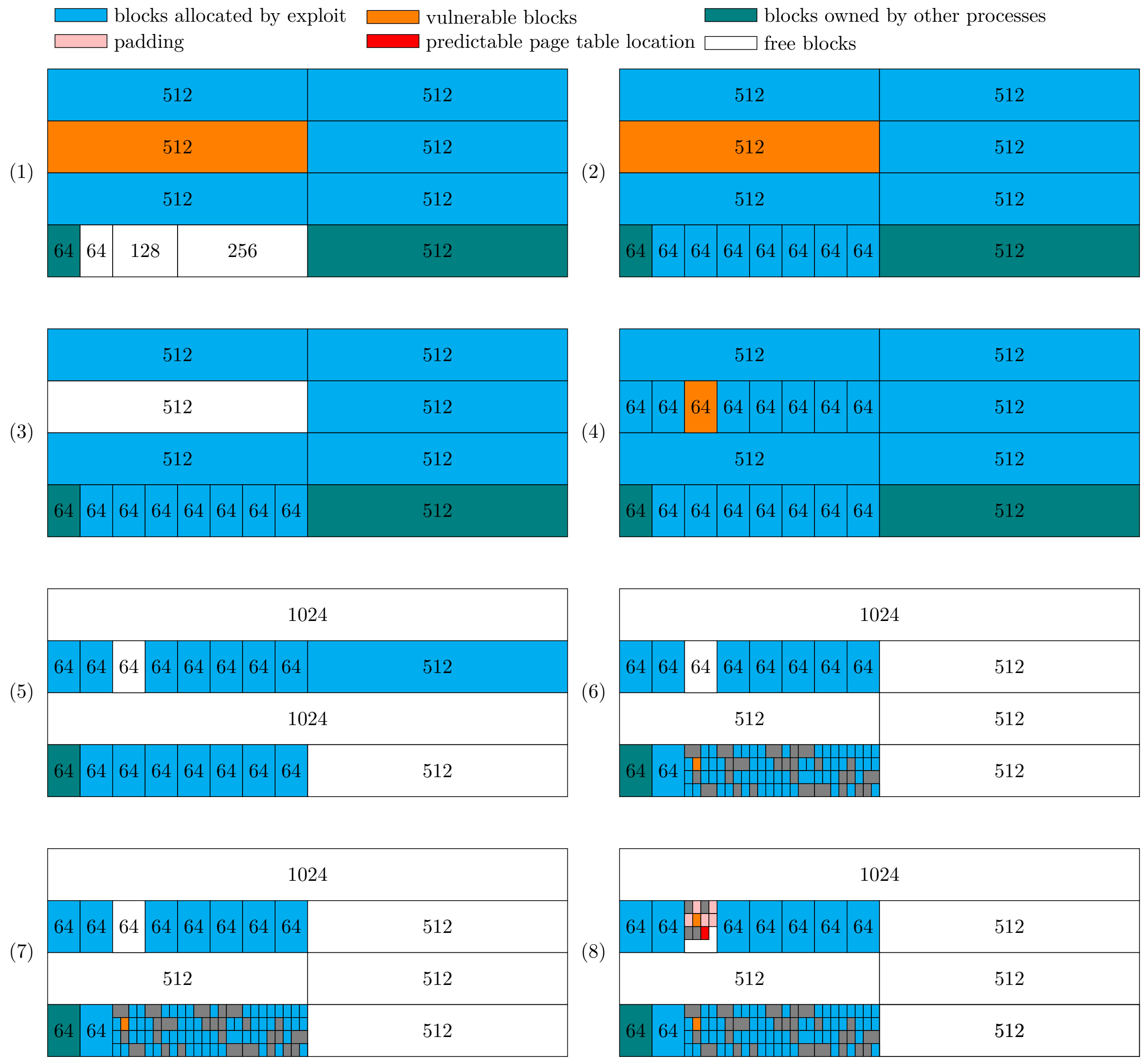

blocks allocated by exploit
vulnerable blocks
blocks owned by other processes
padding
predictable page table location
free blocks
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
512
64
128
256
1024

Figure 5.4: A Phys Feng Shui attack is repelled by page sacrifice.

(8) The attacker forces a page table allocation, in this case two pages are sacrificed and the page table is not allocated at the block, vulnerable to a bit flip induced by a Rowhammer attack.

In summary there are three reasons why Phys Feng Shui fails. Each reason alone would prevent a successful privilege escalation exploit:

- **Unpredictable padding:** The number required padding blocks cannot be determined, because pages can be sacrificed when they are allocated.
- **Unpredictable page table location:** The page table allocation enforced by the attacker is not allocated at the vulnerable physical block, because a random number of sacrificed pages is inserted.
- **Vulnerable block sacrificed** The block containing the vulnerable page is sacrificed and is therefore not used by the operating system.

CHAPTER 6

Evaluation

This chapter evaluates and discusses the PAGE SACRIFICE technique we presented in the last chapter. We analyze the effects on memory usage and performance and evaluate the improved system security.

6.1 Memory Usage

It is in the nature of things that PAGE SACRIFICE increases memory usage. The increase depends on the probability parameter p, which controls the probability of a page sacrifice. Table 6.1 shows memory usage for the Chromium and Mozilla Firefox web browsers with different parameters. To perform memory tests we set up a virtual machine using the Kernel Virtual Machine (KVM) [1]. We chose the Debian 9 [2] Linux distribution as platform for our tests, with a PAGE SACRIFICE-enabled Linux kernel version 4.13. The test environment ran only the minimum amount of processes, necessary to start our tests. Notably the X Window System[3] as graphical user interface, the LightDM [4] display manager and the xterm [5] terminal emulator. After ten seconds our evaluation process killed the web browser's process and determined the amount of used memory and the total number of sacrificed pages. The total memory usage was determined by observing the total amount of physically available free memory, before and after the start of the application. It is important to note that the total memory usage is an approximate value, since the Linux does not keep track of the exact physical memory usage per process.

[1]https://www.linux-kvm.org
[2]https://www.debian.org/
[3]https://www.x.org/
[4]https://freedesktop.org/wiki/Software/LightDM/
[5]http://invisible-island.net/xterm/xterm.html

Browser	p	Sacrificed pages	Mem.sacrificed	Total mem. usage	Overhead rate
Chromium	0.125	20771	81 MB	158 MB	51.27 %
	0.25	37706	147 MB	192 MB	76.56 %
	0.5	99472	389 MB	326 MB	119.33 %
	1	346016	1352 MB	875 MB	154.51 %
Firefox	0.125	12636	49 MB	245 MB	20.00 %
	0.25	34644	135 MB	308 MB	43.83 %
	0.5	94393	369 MB	478 MB	77.20 %
	1	263025	1027 MB	931 MB	110.31 %

Table 6.1: Memory overhead of web browsers

6.2 CPU Benchmarks

To quantify the impact of our solution on the computing power of the system, we ran the SPEC CPU2006[6] benchmark suite on a Intel Core i5-8350U processor on a notebook with 16 GB memory. The test system used Debian 9[7], with the GNU C compiler (GCC)[8] version 6.3.0 using Linux 4.13 with PAGE SACRIFICE enable as kernel. Figure 6.1 shows the performance degradation. A Detailed of our benchmarks can be found in appendix A.

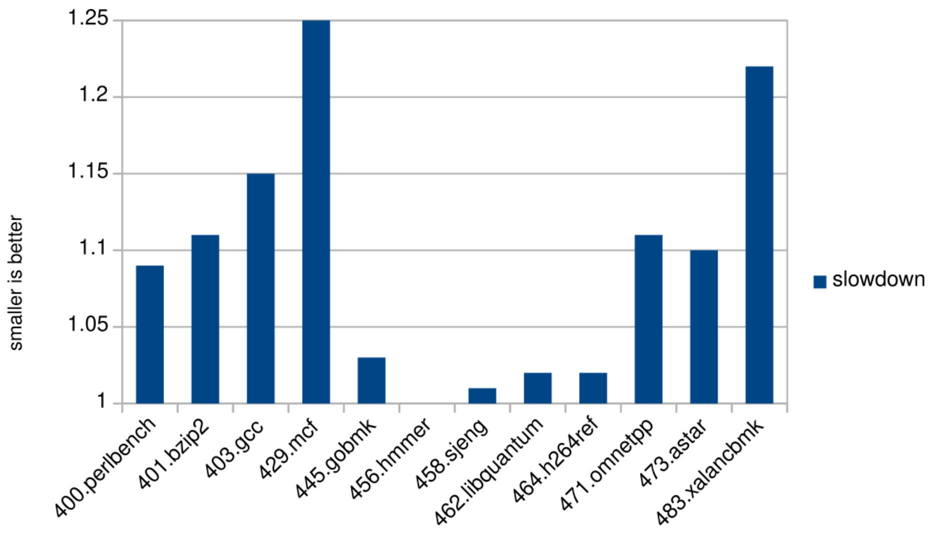


Figure 6.1: The performance degradation of PAGE SACRIFICE is negligible for most uses cases.

[6]https://www.spec.org/cpu2006/
[7]https://www.debian.org/
[8]https://gcc.gnu.org/

6.3 Security evaluation

A key idea of PAGE SACRIFICE is that processes cannot enforce to get the same physical page again. To evaluate the security of PAGE SACRIFICE, we examined how many attempts are requires until Linux reserves the same physical page frame for a process, which constantly allocates and frees memory. The Linux kernel provides information on the allocated physical pages of a process in the `/proc/self/pagemap` file. This file is only accessible by the root user executing binaries with the `cap_sys_admin` captive set.

In our test environment, this allowed us to reliably access page mapping information, however in a real world attack scenario, attackers aim to escalate their privileges, so when their process has already root access and the `cap_sys_admin` captive set, there is no reason to perform further attacks, since the maximum privileges are already reached and the attacker has full control over the system.

We wrote a test application simulating the Phys Feng Shui attack (see section 4.3). The tool probes how many attempts it takes until the Linux kernel assigns the same page frame, it does so by constantly allocating, filling and freeing heap memory. Table 6.2 shows the results of our experiments. If at least one page is sacrificed the operating system never assigned the same page frame to the process. However, if we artificially increased the memory pressure, by running another process constantly allocating and freeing memory, we were able to force the operating system to assign the same page to the process simulating an attack. Due to lacking memory, the operating system kernel was forced to assign the same page frame again after less than 10000 iterations.

Hence, if attackers succeed to fill the entire memory and free a page frame vulnerable to a bit flip at the right moment, they will eventually succeed to place a page table entry at a vulnerable location and consequently hammer it gain write-access to the whole memory, despite the PAGE SACRIFICE defense. However, in this case it is very likely that the attack fails because the attacking process runs of memory. Additionally, it is likely the attacker's process abnormal behavior get detected by system monitoring services or system administrators.

p	same page after n runs	run time
0.125	1	< 1 s
0.25	1	< 1 s
0.5	< 8	< 1 s
1	never	1.5 h

Table 6.2: Attempts needed to return the same physical page.

CHAPTER 7

Conclusion

7.1 Contribution

In this thesis we discussed the fundamental techniques necessary to exploit a producible bit flip in main memory (known as *Rowhammer*) to a working privilege escalation exploit. We introduced the underlying software and hardware architectures and presented an in depth discussion of techniques to tamper operating system design principles to successfully run privilege escalation attacks.

We presented PAGE SACRIFICE (see chapter 5), a modification of the well known buddy allocation mechanism, based on the principles of software diversity. Our defense dramatically increases the efforts for an attacker trying to perform a *Phys Feng Shui* attack (see section 4.3.

We implemented our idea for the Linux kernel version 4.13 and ran tests for real world applications. Moreover, we evaluated the impact of our solution on computational power using the SPEC CPU2006 benchmark suite. To evaluate the effect on memory usage, we measured the memory usage of current web browsers, running on a Linux system equipped with PAGE SACRIFICE. While the impact on the computing power is rather moderate and might be acceptable for many use cases, we found out that the memory consumption of our current solution is extremely high.

7.2 Limitations

It turned out that only sacrificing pages is not satisfying in terms of memory usage. Our experiments revealed that the memory pressure increases drastically for real world applications such as web browsers. Therefore, we consider that the current implementation of only PAGE SACRIFICE will not gain broad acceptance. However, we consider PAGE SACRIFICE as a first part of an effective solution the Rowhammer bug.

7.2.1 When page sacrifice is not suitable

Hardware supporting direct memory access (DMA), i.e. hardware components (such as secondary storage) can access memory directly, without the need to load the data to CPU registers, expects physically contiguous data. Therefore PAGE SACRIFICE is not suitable for memory locations used for DMA. Furthermore, no pages are sacrificed when the allocator is called with the `__GFP_NOFAIL` bit set. When called with this flag the system is in a critical state and must at all cost allocate usable memory. For instance this happens when memory is scarce and the kernel needs memory for critical operations. For performance reasons requests that are made with the `GFP_ATOMIC` flag set are also not sacrificed. It is important to note that only the kernel can request these flags, so disabling memory in these cases, does not mean that attackers can use them to weaken our defense.

7.3 Future Work

We intend to develop PAGE SACRIFCE further to achieve only a constant memory overhead. This can either be done by keeping lists of pages that are returned to requesting processes, when memory is scarce or by constantly returning sacrificed memory. Furthermore, we aim to tackle the Rowhammer problem and apply the ideas of software diversity to other core components of modern operating systems and compiler toolchains. Another motivation for our future research is that our work showed, that software diversity is a suitable, effective and promising method to invalidate the assumption of current and future side-channel attacks.

APPENDIX A

SPEC CPU2006 Benchmarks

Benchmark	Run 1	Run 2	Run 3	avg.
400.perlbench	216 sec	217 sec	216 sec	216 sec
401.bzip2	370 sec	376 sec	375 sec	374 sec
403.gcc	237 sec	235 sec	236 sec	236 sec
429.mcf	314 sec	318 sec	359 sec	330 sec
445.gobmk	369 sec	377 sec	373 sec	373 sec
456.hmmer	285 sec	285 sec	285 sec	285 sec
458.sjeng	394 sec	406 sec	405 sec	402 sec
462.libquantum	249 sec	249 sec	245 sec	248 sec
464.h264ref	383 sec	384 sec	384 sec	384 sec
471.omnetpp	329 sec	339 sec	339 sec	336 sec
473.astar	349 sec	355 sec	361 sec	355 sec
483.xalancbmk	188 sec	195 sec	203 sec	195 sec

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[2] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[3] US-CERThttps://www.us-cert.gov/ncas/alerts/TA18-141A. *Side-Channel Vulnerability Variants 3a and 4*. May 2018. URL: `https://www.us-cert.gov/ncas/alerts/TA18-141A`.

[4] Yoongu Kim et al. "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*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 361–372. ISBN: 978-1-4799-4394-4. URL: `http://dl.acm.org/citation.cfm?id=2665671.2665726`.

[5] Daniel Gruss et al. "Another Flip in the Wall of Rowhammer Defenses". In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 2018, pp. 245–261. DOI: `10.1109/SP.2018.00031`. URL: `https://doi.org/10.1109/SP.2018.00031`.

[6] William Stallings. *Operating Systems: Internals and Design Principles*. 9th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2017. ISBN: 978-0134670959.

[7] Ferdinand Brasser et al. "CAn'T Touch This: Software-only Mitigation Against Rowhammer Attacks Targeting Kernel Memory". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, pp. 117–130. ISBN: 978-1-931971-40-9. URL: `http://dl.acm.org/citation.cfm?id=3241189.3241200`.

[8] *Four-level page tables merged*. Jan. 2005. URL: `https://lwn.net/Articles/117749/`.

[9] *FreeBSD Architecture Handbook*. URL: `https://www.freebsd.org/doc/en/books/arch-handbook/vm-pagetables.html`.

[10] Intel. *5-Level Paging and 5-Level EPT White Paper*. May 2017. URL: `https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf`.

[11] *Five-level page tables.* Mar. 2017. URL: `https://lwn.net/Articles/717293/`.

[12] The FreeBSD Documentation Project. *Frequently Asked Questions for FreeBSD 10.X and 11.X.* URL: `https://www.freebsd.org/doc/en/books/faq/misc.html`.

[13] Jedec Solid State Technology Association. *DDR3 SDRAM Standard.* July 2012. URL: `https://www.jedec.org/standards-documents/docs/jesd-79-3d`.

[14] Kuljit S Bains et al. *Row hammer refresh command.* US Patent 9,236,110. 2016.

[15] Mark Seaborn and Thomas Dullien. *Exploiting the DRAM rowhammer bug to gain kernel privileges.* Mar. 2015. URL: `https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`.

[16] S. Govindavajhala and A. W. Appel. "Using memory errors to attack a virtual machine". In: *2003 Symposium on Security and Privacy, 2003.* May 2003, pp. 154–165. DOI: `10.1109/SECPRI.2003.1199334`.

[17] *Intel® Streaming SIMD Extensions Technology.* July 2017. URL: `https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html`.

[18] Zelalem Birhanu Aweke et al. "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 743–755. ISBN: 978-1-4503-4091-5. DOI: `10.1145/2872362.2872390`. URL: `http://doi.acm.org/10.1145/2872362.2872390`.

[19] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.Js: A Remote Software-Induced Fault Attack in JavaScript". In: *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721.* DIMVA 2016. San Sebastián, Spain: Springer-Verlag New York, Inc., 2016, pp. 300–321. ISBN: 978-3-319-40666-4.

[20] Yossef Oren et al. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications". In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security.* CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 1406–1418. ISBN: 978-1-4503-3832-5. DOI: `10.1145/2810103.2813708`. URL: `http://doi.acm.org/10.1145/2810103.2813708`.

[21] Fangfei Liu et al. "Last-Level Cache Side-Channel Attacks Are Practical". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy.* SP '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 605–622. ISBN: 978-1-4673-6949-7. DOI: `10.1109/SP.2015.43`. URL: `https://doi.org/10.1109/SP.2015.43`.

[22] Victor van der Veen et al. "Drammer: Deterministic Rowhammer Attacks on Mobile Platform". In: *Proceedings of the 23rd Conference on Computer and Communications Security (CCS 2016).* Oct. 2016.

[23] Thomas M. Zeng. *The Android ION memory allocator*. Feb. 2012. URL: https://lwn.net/Articles/480055/.

[24] Franz Ferdinand Brasser et al. "CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks". In: *CoRR* abs/1611.08396 (2016). arXiv: 1611.08396. URL: http://arxiv.org/abs/1611.08396.

[25] Franz Ferdinand Brasser et al. "CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks". In: *CoRR* abs/1611.08396 (2016). arXiv: 1611.08396. URL: http://arxiv.org/abs/1611.08396.

[26] Daniel Gruss et al. "KASLR is Dead: Long Live KASLR". In: *Engineering Secure Software and Systems*. Ed. by Eric Bodden, Mathias Payer, and Elias Athanasopoulos. Cham: Springer International Publishing, 2017, pp. 161–176. ISBN: 978-3-319-62105-0.

[27] Yueqiang Cheng, Zhi Zhang, and Surya Nepal. "Still Hammerable and Exploitable: on the Effectiveness of Software-only Physical Kernel Isolation". In: *CoRR* abs/1802.07060 (2018). arXiv: 1802.07060. URL: http://arxiv.org/abs/1802.07060.

[28] Jonathan Corbet. *The current state of kernel page-table isolation*. Dec. 2017. URL: https://lwn.net/Articles/741878/.

[29] Dave Hansen. *KAISER: unmap most of the kernel from userspace page tables*. Oct. 2017. URL: https://lwn.net/Articles/737940/.

[30] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "MASCAT: Stopping Microarchitectural Attacks Before Execution". In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 1196. URL: http://eprint.iacr.org/2016/1196.

[31] Intel Corporation. *Intel® Software Guard Extensions (Intel® SGX)*. URL: https://software.intel.com/en-us/sgx.

[32] David Kaplan, Jeremy Powell, and Tom Woller. *AMD Memory Encryption White Paper*. Mar. 2016. URL: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[33] Jonathan Corbet. *Defending against Rowhammer in the kernel*. Oct. 2016. URL: https://lwn.net/Articles/704920/.

[34] Mathias Payer. "HexPADS: A Platform to Detect "Stealth" Attacks". In: *Engineering Secure Software and Systems*. Ed. by Juan Caballero, Eric Bodden, and Elias Athanasopoulos. Cham: Springer International Publishing, 2016, pp. 138–154. ISBN: 978-3-319-30806-7.

[35] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. "CloudRadar: A Real-time Side-channel Attack Detection System in Clouds". In: Sept. 2016.

[36] M.Ghasempour, M. Lujan, and J.Garside. *ARMOR: A Run-time Memory Hot-Row Detector*. 2015. URL: http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/armor.html.

[37] Jedec Solid State Technology Association. *Low Power Double Data Rate 4*. Mar. 2017. URL: `http://www.jedec.org/standards-documents/docs/jesd209-4b`.

[38] Hewlett-Packard Inc. *HP Servers - "Rowhammer" Security Vulnerability*. Mar. 2015. URL: `https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c04639675`.

[39] Lenovo Inc. *Row Hammer Privilege Escalation Lenovo Security Advisory: LEN-2015-009*. July 2016. URL: `https://support.lenovo.com/at/en/product_security/row_hammer`.

[40] Apple Inc. *About the security content of Mac EFI Security Update 2015-001*. Jan. 2017. URL: `https://support.apple.com/en-us/HT204934`.

[41] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89683-4.

[42] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Birmingham, UK, UK: Wrox Press Ltd., 2008. ISBN: 9780470343432.

[43] Jason Evans. *A Scalable Concurrent malloc(3)Implementation for FreeBSD*. Apr. 2006. URL: `https://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf`.

[44] *setuid(2) Linux Programmer's Manual*. Sept. 2017.

[45] JF Bastien. *Disallow the x86 "clflush" instruction due to DRAM "rowhammer" problem*. Mar. 2015. URL: `https://bugs.chromium.org/p/nativeclient/issues/detail?id=3944#c15`.

[46] Kaveh Razavi et al. "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. ISBN: 978-1-931971-32-4. URL: `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi`.

[47] Daniel Gruss et al. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 2016, pp. 368–379. DOI: `10.1145/2976749.2978356`. URL: `http://doi.acm.org/10.1145/2976749.2978356`.

[48] Frederick B. Cohen. "Operating System Protection Through Program Evolution". In: *Comput. Secur.* 12.6 (Oct. 1993), pp. 565–584. ISSN: 0167-4048. DOI: `10.1016/0167-4048(93)90054-9`. URL: `http://dx.doi.org/10.1016/0167-4048(93)90054-9`.

[49] Stephen Crane et al. "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity." In: *NDSS*. 2015, pp. 8–11.

[50] Andrei Homescu et al. “Profile-guided Automated Software Diversity”. In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. CGO ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–11. ISBN: 978-1-4673-5524-7. DOI: 10.1109/CGO.2013.6494997. URL: http://dx.doi.org/10.1109/CGO.2013.6494997.