Reverse Engineering Intel DRAM Addressing and Reproduction of Blacksmith

by

Mahfuz Sobhan
20101270

Khandker Samia Rahman Pranti
21101103

Bashir Siddique
20101269

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
School of Data and Sciences
Brac University
October 2024

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Mahfuz Sobhan
20101270

Khandker Samia Rahman Pranti
21101103

Bashir Siddique
20101269

Approval

The thesis/project titled "Reverse Engineering Intel DRAM Addressing and Reproducing Blacksmith on Goldmont Plus microarchitecture" submitted by

1. Mahfuz Sobhan (20101270)
2. Khandker Samia Rahman Pranti (21101103)
3. Bashir Siddique (20101269)

As of Summer, 2024 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on October 17, 2024.

Examining Committee:

Supervisor:
(Member)

Mr. Rafeed Rahman
Lecturer
Department of Computer Science and Engineering
Brac University

Thesis Coordinator:
(Member)

Md. Golam Rabiul Alam, PhD
Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Sadia Hamid Kazi, PhD
Chairperson and Associate Professor
Department of Computer Science and Engineering
Brac University

Abstract

Rowhammer is a widely known computer hardware vulnerability in recent years which breaks the fundamental limitations of DRAM technology by repeatedly accessing in order to cause bitflips in the adjacent rows. Understanding and analyzing the effect of Rowhammer in various architectures is a primary need for enhancing system security and mitigating potential risks against RowHammer. This paper presents the development of a novel library for reverse engineering DRAM address functions, enabling efficient mapping and analysis of physical memory addresses across diverse DRAM architectures. The library accelerates the much-needed extraction of address-mapping functions, which is a pivotal part for the determination of the Rowhammer vulnerability in diverse architectures. We integrate the extracted address mappings with the Blacksmith fuzzer, a state-of-the-art Rowhammer testing tool, and deploy it on our targeted machines with three distinct architectures. Our experiments analyze and compare the Rowhammer effects across these platforms, evaluating metrics such as activation interval, refresh rates, bit flip distribution, and the potential for reliable exploitation. The results reveal architecture-specific characteristics of Rowhammer susceptibility and highlight the effectiveness of the proposed library in automating and streamlining DRAM address function extraction. Our findings offer interesting insights into the variations in Rowhammer susceptibility across architectures which contributes to the ongoing efforts of designing resilient systems and develop standardized testing methodologies for hardware vulnerabilities.

Keywords: Rowhammer, Bitflips, Blacksmith, DRAM, Address Mapping

Acknowledgement

First and foremost, all praise to Allah Subhanahu Wa Ta'ala under whose blessings our thesis has been successfully concluded without any major disruptions. Second of all, we are deeply grateful to our supervisor Mr. Rafeed Rahman for this opportunity. This thesis would have not been completed without his support and freedom he gave us throughout our journey.

Table of Contents

List of Figures

Chapter 1

Introduction

DRAM - Dynamic Random Access Memory, is the major technology behind the main memory in modern computing systems because of its advantageous cost per capacity. It depicts an important role in the overall performance and reliability of the system. In order to increase the capacity of DRAM, vendors has to scale down the technology node size which results in the increment of DRAM cell density that undermines the DRAM reliability. One of the main challenges behind RowHammer effect is related to interference which appears as a result of technology scaling. RowHammer is a serious security vulnerability that occurs by repeated activation of a single DRAM row which eventually causes disturbance in its neighboring rows. This disturbance phenomenon results to bit flips in the physically adjacent rows. The attacker can exploit the bug by accessing a single memory address to change data in another address despite being co-located physically only. Since DRAM is a critical bug across modern systems, many sophisticated RowHammer attacks had been performed in recent studies [16], [23], [27], [33], [29].

RowHammer introduces a major challenge for system designers as it is based on the fundamental DRAM circuit behavior which is hard to modify. This make the RowHammer a possible threat over various generations and designs of DRAM. Research done by Kim et al. [7] shows that RowHammer appears as the DRAM technology scales down [5], [7], [22], which means, with the increment of DRAM storage density, chips may become prone to this effect. The susceptibility of a particular DRAM chip to RowHammer is expressed in terms of the number of times a particular row has been activated (single-sided RowHammer) to produce the first flipping of the first bit. Recently, a study by Yang et al. [30] supported the theory by recognizing a specific circuit-level charge leakage mechanism that could be responsible for RowHammer. This leakage impacts adjacent circuit components, which means that as manufacturers intensify scaling techniques to enhance storage density [1]–[3], the threat posed by RowHammer is likely to intensify.

To overcome the RowHammer problem, several works suggest methods designed to shield the system from RowHammer-induced bit flips. A significant drawback while designing a RowHammer mitigation is thinking is presuming that the attackers will perform in the same manner once the defense is deployed. This is especially true

for in-DRAM Target Row Refresh (TRR) which is a protection strategy directed towards the mitigation of an increasing RowHammer phenomena in DRAM substrates. At present, proprietary and undisclosed in-DRAM TRR is the only existing protection mechanism that shield systems against RowHammer attacks in browsers, mobile gadgets, and indeed, in the cloud and over the network [10], [14], [16]. Despite of this flaw, TRR is showing its effect in the newer microarchitectures. In this paper, we demonstrated how the variations in the conventional RowHammer access patterns allow one to flip bits on our tested DDR4 DIMMs and also how the latest microarchitectures acts against this.

Target Row Refresh (TRR) is a hardware-based mitigation that is designed to address the Rowhammer vulnerability with recent variants functioning fully within the DRAM chips [33]. However, TRR seeks to sense rows that are frequently accessed (hammered) and re-write the data on the neighboring rows before the charges leak and cause data to be written inappropriately. The challenge behind TRR is to identify the frequent items that are being accessed in the DRAM efficiently. Since the frequent item count technique is pretty expensive in hardware, some TRR mechanisms struggle to track the all of them which may lead to data corruption[36]. Yet, around 70% of TRR implementations are effective in detecting all the aggressors rows if they are hammered frequently enough.

In this paper we analyzed the reproducibility of RowHammer fuzzing in three different machines from the same vendor manufactured in three different times showing that even though bitflip exists, the manufacturers are going hard against it. We focused on bitflips vs refresh interval and how the clock cycle differs in all three different machines. Our research is basically finding answer to the question, are the vendor getting harder against RowHammer bitflips over time?

For our experiment, we reproduced Blacksmith, a scalable RowHammer fuzzer in the frequency domain [41]. Blackmsith is initially built and tested on Coffee Lake architecture. In order to run it on a different architecture, we developed a library to call from the Blacksmith fuzzer that will reverse engineer the the DRAM address functions for out tested machines. The contributions of this papers are:

1. Building a library to revere engineer the DRAM address function for our tested INTEL architectures.

2. Reproducing the results of Jattke et al. [41] on three different machines with completely different architectures. The architectures are listed below:

    - Intel(R) Celeron(R) N4000 CPU - Goldmount Plus Architecture
    - Intel(R) Core(TM) i5-10500 CPU - Commet Lake Architecture
    - Intel Core i9-13900K - Raptor Lake Architecture

3. Studying how the vendors are reorganizing their architecture against the RowHammer.

Chapter 2

Literature Review

The RowHammer vulnerability is first introduced in 2014 by Kim et al. [7], highlighting a critical flaw in DRAM which creates memory disturbance errors. They did their experiment in an FPGA based DRAM and presented a mitigation named Probabilistic Adjacent Row Activation (PARA) which enables probabilistic refreshes to susceptible rows. PARA mitigation is cost effective with minimal performance impact which requires sensitive address mapping data that vendors keep private. Their paper also showed that how a RowHammer attack can violate memory protection by reading row in DRAM. Following the year, in 2015, Seaborn and Dillien [11] introduced the first RowHammer attack, utilizing it for kernel privilege escalation from user land. The attack was based on Page Table Entry (PTE) manipulation which enables attacker an unauthorized access to over the memory. Their work exposed several risk residing in the commodity DRAM chips. After that, Rowhammer researches expanded with significant progress in exploitation techniques. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector [14], introduced the first JavaScript-based RowHammer attack. The paper showed that the attackers can remotely exploit RowHammer by executing codes in Microsoft Edge. This attack utilized the memory deduplication which enhances attack reliability.
Soon after the paper was published, a group of researches from Graz University of Technology came up with a paper, Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript [10] that demontrated a similar kind of attack which uses cache eviction to remove accessed rows from the cache. Their work implemented RowHammer in both JavaScript and Native code using cache manipulation techniques to utilize software faults.
Within the same time Flip Feng Shui: Hammering a Needle in the Software Stack [19] came up with a new attack vector that allows precise control over bit-flipping. The method includes randomly flipping and the option to map from physical to virtual memory address spaces and thus breaking the Virtual Machine memory isolation. Another version of this approach is called Drammer [20], a development where Rowhammer got adapted to Android smartphones, and further from ordinary applications that were unprivileged. Drammer used brute force approach by rapid hammering to place victim data in vulnurable regions which targets contigous

physical memory It was the first RowHammer attack on android system.
Some studies started showing that RowHammer can also be exploited on virtual environments. Through a virtual machine, an atatcker can gain read and right access on memory with double sided RowHammer techniques [21].
A paper by Qiao and Seaborn [18] introduced a different kind of RowHammer variant that bypass the CLFLUSH instruction, expanding potential remote RowHammer attack avenues. Their study revealed that how the widely used function like memset and memcpy can trigger RowHammer and break application layer defenses. Additional researches like Curious case of Rowhammer: Flipping Secret Exponent Bits using Timing Analysis [13] and DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks [17] used cache eviction techniques to exploit RowHammer. These studies emphasized the threat of RowHammer over various environments. Tatar et al.'s work, Defeating Software Mitigations against Rowhammer: a Surgical Precision Hammer [24], demonstrated how to circumvent some of the more recent Rowhammer defenses such as ANVIL [12] and CATT [15], the authors claimed considerable efficacy by accurately aiming at the DRAM address space. In another study, GuardION: In Practical mitgation of DMA-Based Rowhammer Attacks on ARM [25], the authors introduced RAMpage, a series of Rowhammer attacks on android. RAMpage was capable of launching app-to-app attacks or even exploit at the root level notwithstanding all the methods of mitigation.
In 2019, researchers tested ECC memory's effectiveness against Rowhammer in Exploiting Correcting Codes: A study on the performance of ECC Memory Against Rowhammer Attacks [26]. They reproduced ECC implementations through memory error patterns and developed ECCploit which is a Rowhammer attack composite of ECC vulnerabilities. TeleHammer: Another work A Stealthy Cross-Boundary Rowhammer Technique [31] proposed a technique called TeleHammer for using indirect row access through third party processes and showed PThammer as a cross-boundary Rowhammer methodology.
In 2020, Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers [32] introduced a testing methodology for Rowhammer in DRAM in cloud environments, tackles the often discussed problem of identifying the useful CPU instruction sequences for row activation and revealing physical memory maps which is often kept secret by manufacturers. Its success was called into doubt since the researchers demonstrated real and potent Rowhammer attacks on the sort of hardware found in the TRR-DRAMs. Lastly, RAMBleed: The Rowhammer effect was originally used in Reading Bits in Memory Without Accessing Them [35] with the aim of affecting DRAM confidentiality instead of its integrity and enabled bit-reading without requiring permissions to access the memory. The authors obtained the secret keys from OpenSSH server and explained that Rowhammer could lead to catastrophic information leakage.
In Revisiting RowHammer: The Effects of Row Operation and mechanisms to address it: An Experimental Analysis on Modern DRAM Devices [34], the study

analyzed over 1500 DRAM chips across DRAM generations and from different manufacturers and concluded that Rowhammer threats have evolved with time. They noted that, newer DRAM devices get attacked relatively fewer times from aggressor rows, signifying that exploitation is easier, and asking for stronger remediation. In GhostKnight: Originally starting from attacking data integrity through speculative execution known as DataAquarius [38], GhostKnight that utilizes both Spectre and Rowhammer was introduced.

In 2021, SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript [40] extended many-sided Rowhammer approach of TRRespass [33] and the latter was performed on DRAMs with Target Row Refresh (TRR) protections. In SMASH, a variety of issues was reported in terms of memory allocation, generation of more access patterns and timing but was achieved remarkable attacks ignoring cache flushes. Another study, Half-Double: New hammering technique for DRAM Rowhammer bug [39] operates at second-level neighbor rows, which affect the victim row from a farther distance.

In 2022, the Blacksmith fuzzer from BLACKSMITH: Scalable Rowhammering in the Frequency Domain [41] was developed to induce bit flips in DRAM using various, non-uniform access patterns. SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks [42] merged Rowhammer with speculative execution attack by improving bit-flip rates which later uncovered thousands of Spectre gadgets in the Linux kernel. SpecHammer enabled attacks such as buffer overflow canary discovery and arbitrary memory reading, showing the heightened risk Rowhammer poses when paired with Spectre for speculative attacks.

Chapter 3

Background

This chapter contains the background knowledge of DRAM, RowHammer and the Blacksmith fuzzer. The first section covers an overview of DRAM, a computer's main memory, and the target of the Rowhammer attack. This part explains the anatomy of DRAM, the inner functionalities of a single cell, and the cooperation between CPU and DRAM. The following section provides explanation on Rowhammer and its connection with DRAM. The last section gives an highlevel overview of the Blacksmith [43] fuzzer which we are using to test our targeted machines and how it attacks the DRAM to exploit RowHammer.

3.1 Dynamic Random Access Memory (DRAM)

DRAM, the main memory of a computer system which is also known as a computer's physical memory contains all the current running programs and their data, including the perating system itself. The primary reason behind its widespread usage is its lower size and cheaper price. The effort behind the reduction of production cost has increased the DRAM cell density significantly without implementing any adequate memory isolation technique.

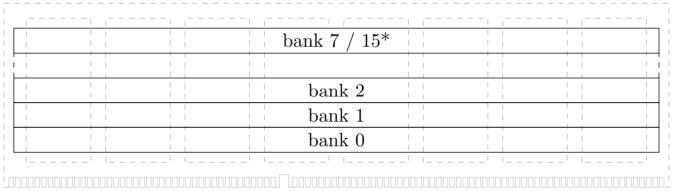

Figure 3.1: A DRAM rank is segmented into multiple banks.

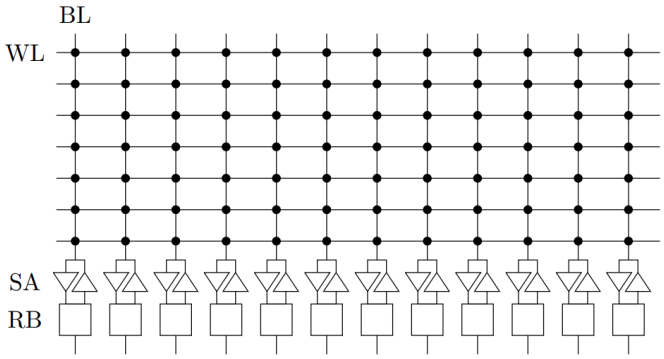

Figure 3.2: A DRAM Bank contains multiple rows and columns consisting of cells.

A DRAM consists of innumerable memory cells. To efficiently identify the read and write addresses of DRAM, it is divided into separate sections. The uppermost level of the DRAM hierarchy is the Dual Inline Memory Module (DIMM), and the bottommost level of that hierarchy is a single cell that contains a bit. Modern computers contain several DIMMs which are connected to the motherboard. Each physical side of these memory modules is known as a rank. A rank is later divided into several banks, as shown in Figure 3.1. In a DDR4 memory, the banks are separated into two or four bank units where the timing delay is reduced while activating or accessing different rows from the same bank, as these separate groups of banks can operate freely. The previous versions of DDR (e.g DDR2 or DDR3), took more time to switch between row buffers. It is also visible in the figure that the banks are divided into eight parts of a non error code correction memory.

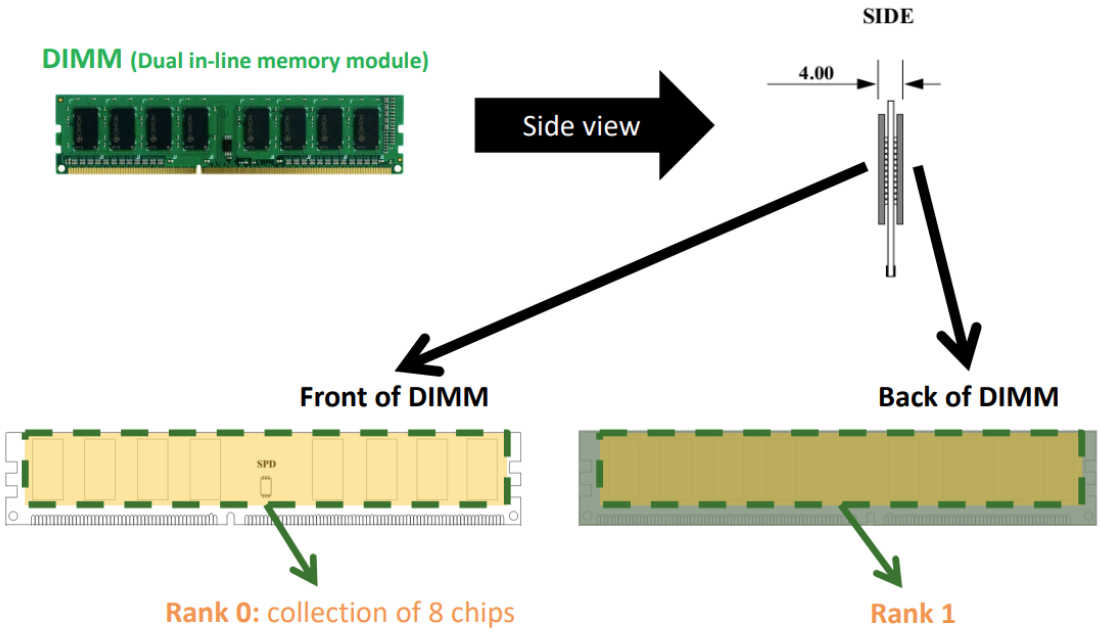

Figure 3.3: Dual in-line memory module with ranks

Cells inside a bank are arranged among rows and columns in a grid-like structure. Cells connecting side to side inside a bank are known as word lines, and the cells connecting up and down are known as bit lines. The bit lines are vertically connected with the sense amplifiers of the row buffer. The structure is shown in Figure 3.2.
Mapping functions allow converting physical address of certain memory location to the corresponding channel, rank, bank and row. Some of these functions are CPU architecture dependent and in most cases are not publicly divulged. In order to read a particular address, the word line corresponding to the particular row is enabled these connect all the cell capacitors in the row to the sense amplifiers in the row buffer. This action is named opening a row which makes the row in the buffer the active row. The coupling capacitors being smaller than 10 fF, get discharged when they are sensed of the amplifiers, which results in a destructive read of the row information. Subsequent accesses from this row are accessed from the row buffer to improve performance. Nevertheless, if data from another row is required, then, the currently active row must be closed and the data written back from the buffer before a new row can be activated and put in the buffer. Therefore, reading from different rows within the same bank also results in triggering of writes to those rows.
In most CPUs, after the selection of a row, that particular row remains open and active until another row is chosen, in order to allow many more accesses to go in the active row. This strategy is effective when several accesses are made near by memory locations, this is a typical case of large numbers of desktop computers having few cores. However, if the next accesses go to different rows, an additional time is required to close the previous row, which will negatively affect performance. The content of rows which does not access frequently can be more efficiently protected by closing the row right after the access. This method called the closed-row policy is different from the open-row policy and is implemented mainly in those CPUs that have more cores.
In order to understand the Rowhammer vulnerability in a better way, it is necessary to consider silicon structure in detail. The latter is stored in a cell having a MOSFET and a capacitor [12]. Ongoing enhancements in these cells have resulted in a design where two cells use the same active region in the silicon. The structure of this shared active region is illustrated in a schematic in Figure 2.4 left, and the corresponding silicon structure in Figure 2.4 right. The orange area (1) is the two cell capacitors although their height here is not to scale. The blue areas (2) represent the drain and source of the two transistors of the circuit. The bit line in the purple (3) is selected and connected to both transistors as the two word lines in the dark blue (4) are implemented in the gates in the gray (5).

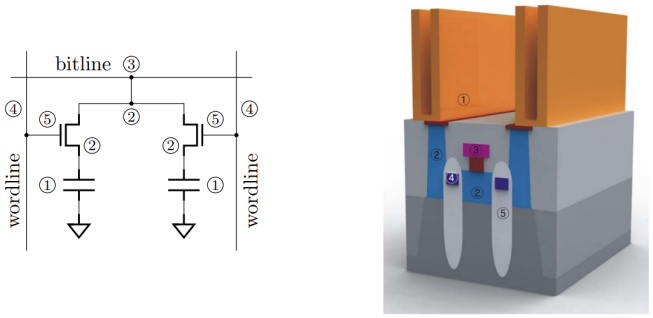

Figure 3.4: A single DRAM cell consists of a MOSFET and a capacitor. Two cells share an active region in the silicon.

The structures of this design have an elongated oval gate shape, which makes it possible to have a longer channel length than a wide structure, unlike the planar MOSFET structure. Capacitor discharge by very short channel MOSFETs increases subthreshold current that is detrimental to DRAM memory [6]. Even so, due to the very small capacitors, the stored charge can be easily affected by a small leakage current, which is not optimised in the channel structure: the channel structure is optimised for the smallest possible leakage currents in this world, below 10 fF. Therefore, capacitor's voltage in DRAM cells should be constantly refreshed to maintain the capacitor's voltage higher than the digital "1" threshold [17]. This requirement for constant updates is the reason why DRAM is called dynamic random access memory, as opposed to static RAM (SRAM), which affords no such updating. In most of today DRAM modules, there is a refresh rate of 64 ms where every cell is refreshed. Each row is refreshed in a sequential manner and every few microseconds a row is refreshed.
Memory partitioning divides memory into several levels including row and channel to enhance the parallelism of memory access hence enhancing memory bandwidth. To maximize this parallelism in the DRAM structure, it is desirable for accesses to different rows to appear random. This goal has been achieved by CPU manufacturers designing DRAM mapping functions that map physical addresses to banks, ranks, and channels seemingly randomly. While some of these functions of mapping are unique to particular CPU families, most of these are not stated clearly by the manufacturers [17].

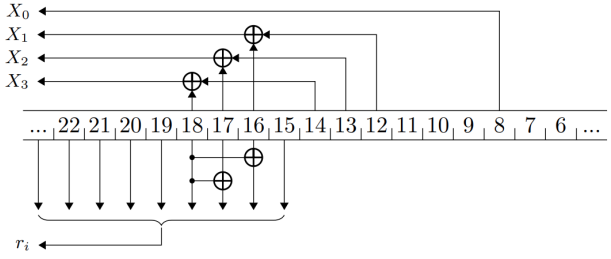

Figure 3.5: DRAM address mapping from physical memory

Knowledge of these mapping functions can make attacks such as cross-CPU row-buffer covert channels [11], and Rowhammer attacks [4, 8]. DRAMA [13] is a tool that is designed to do just the opposite of these mappings, by analyzing large contigouous memory regions and detecting timing differences that are due to row-buffer hits and conflicts. Figure 2.5 illustrates the reverse engineered mapping from physicall address to DRAM banks and ranks. In our test, this mapping will enable us to determine physical neighbors and additional physical address bits from the pages in those neighbors. These extra bits will enable us to execute row operations with a bit index r almost completely avoiding the row-shuffling Rowhammer defense present on the target device.

3.2 RowHammer

Modern DRAM devices are susceptible to disturbance errors which is caused by repeated access to a single DRAM row and that can inadvertently alter the values of cells in neighboring rows. This is a technology failure that occurs from electromagnetic crosstalk interference with the neighboring cells called RowHammer. The risk of RowHammer is increasing due to the technology nodes getting smaller, where, the neighboring DRAM cells become more compact and closer to each other. Consequently, as DRAM manufacturers progress improving storage density, the susceptibility of chips to RowHammer-caused bit flips also increases. RowHammer is a low-level system security vulnerability that has been explored widely in previous research from both offensive and defensive perspectives. Studies have shown that RowHammer can be exploited to perform various system-level attacks which includes privilege escalation [3, 5, 6, 7, 13], data leakage [21], and denial of service [16, 17]. These findings necessitate the systems to incorporate protections against RowHammer to maintain secure and reliable memory system. Previous works has proposed RowHammer defenses across both hardware [8], [9] and software [13] domains. DRAM vendors have implemented an in-DRAM mechanisms known as Target Row Refresh (TRR) [4], which utilizes proprietary operations to reduce vulnerability to RowHammer attacks, though these solutions have recently shown limitations [26]. Memory controllers and system manufacturers have introduced additional defenses, such as increasing refresh rates [22, 23] and Hardware RowHammer Protection (RHP) [28], [37].

3.3 Blacksmith

This section illustartes the design and implementation of Blacksmith. Firstly we describe the high-level overview of Blacksmith's architecture and then how Blacksmith creates RowHammer patterns, providing a formal explanation of the core concepts involved. Figure 9 shows the main components of Blacksmith. The Pattern Gen-

erator, (1), which is responsible for implementing non-uniform access patterns by randomizing the timing of aggressor accesses within each pattern. The Aggressor Mapper, (2), assigns these aggressor patterns to DRAM locations using known bank/rank address functions [17], [40].

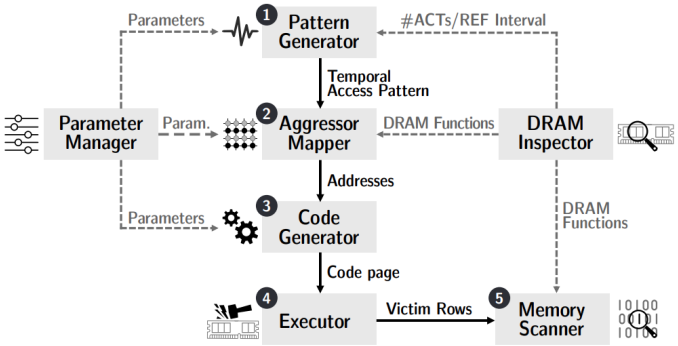

Figure 3.6: Overview of Blacksmith Fuzzer

During this process, aggressors are either evenly distributed across the same DRAM bank or randomly mapped, maintaining one row between aggressors targeting the same victim row. Mapping parameters are shuffled in the fuzzing phase. Virtual addresses generated by the mapper for the hammered rows and passes them to the Code Generator (3), which compiles the hammering instructions into formatted code. To avoid conditionals like if-else during execution (to prevent speculative execution of branches), it compiles access patterns directly, which ensures access order integrity. This compilation also lets us identify where memory reads and flushes need to be serialized using fences. We apply a "flush-early and fence-late" approach by flushing aggressors from the cache right after access and fencing immediately before re-access to reduce the performance overhead of serialization. The Executor, (4) then executes the compiled code across multiple refresh windows (64 ms intervals). To maintain the correct frequency of row accesses, it synchronizes with the DRAM REFRESH at the start of each pattern repetition, similar to. Finally, the Memory Scanner, (5), checks if any changes took place in the generated random data pattern previously written to memory. Since any aggressor pattern may cause bit flips, the scanner inspects the two rows around each aggressor for bit flips; if detected, it records the flipped bits and restores the original data pattern. Following this, Blacksmith either re-hammers the same pattern on a different DRAM location, or hammers the same pattern with a new mapping or generates a new pattern to restart the process.

Chapter 4

Research Methodology

In this chapter, we discuss the methodology undertaken to develop a comprehensive DRAM address mapping library and test it across multiple Intel architectures to validate its effectiveness in generating precise address mapping functions. This research aims to bridge the gap between DRAM address mapping extraction and fuzz testing by facilitating automated mapping function generation adaptable to different CPU architectures. Following the extraction and analysis phase, where we validated the library's functionality on Intel's Goldmont Plus (Celeron N4000), Comet Lake (Core i5-10500), and Raptor Lake (Core i9-13900K) architectures, we plan to integrate the generated mapping functions into the Blacksmith fuzzer. This application will enable methods of access to memory that will be targeted and also improved ways of creating fuzzers and this will boost the chances of finding out vulnerabilities that are related to DRAM. Thus, with an exact reproduction of the specific DRAM configuration of each architecture, the developed mapping functions will facilitate enhanced fuzzing and enable the Blacksmith fuzzer to perform the tests more efficiently and accurately according to each architecture's characteristics.

4.1 Reverse Engineering the DRAM Address function

In order to reproduce the Blacksmith fuzzer on our tested machines, we first had to reverse engineer the DRAM address functions for the respective architectures as the DRAM address functions were hard coded in Blacksmith [43]. Based on the technique used in IAIK DRAMA [44], we developed a library by calling which, the DRAM bank address functions will be reverse engineered based on the architecture. Our library with Python specially designed to automate the setup building and the execution of DRAM address mapping methods used in IAIK DRAMA [44] which is extensively used in memory research. The DRAMA tool expects users to perform a number of compilational steps and to deal with several dependencies on their own which necessitates largely manual approach. To overcome these challenges our library abstracts these tasks into simple Python interface, which makes the process

much easier for common users, who may have no experience with that makefiles or build systems.

The process by which DRAMA works is splitted into three main stages. In step 1, it detects row and column bits that gives coarse-grained results which is either most of the row and most of the column is detectable but some of them are still in grey boxes. In Step 2, physical addresses are chosen that only vary at the bits represented by grey boxes, and then categorize them into distinct sets (each set corresponds to a bank), from which we can determine bank address functions that hold for all members of the set. In the third step, we perform a more detailed examination of the determined bank address functions in order to identify new row or column bits that are also present in the bank address functions of the previous step. The following figure explains our description.

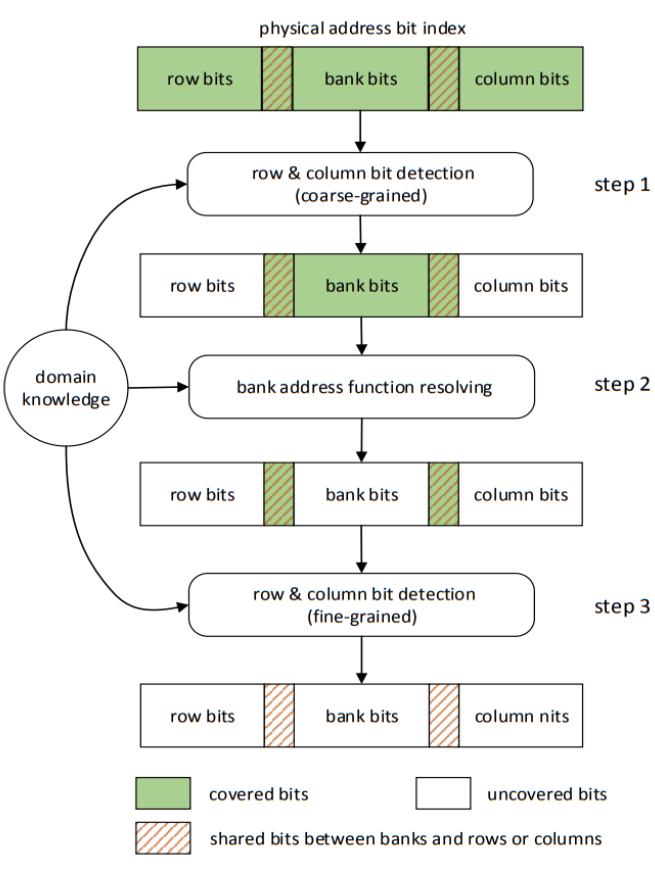

Figure 4.1: DRAM address mapping from physical memory

The design of our library involved significant effort to ensure reliability and simplicity of use. First of all, we studied the structure behind DRAMA project and its dependencies, custom build targets and potential failure points during compilation. Based on our observations, we implemented a series of Python functions for utilizing the make command through the subprocess module so that each phase, from dependency confirmation to build implementation, is seamless. For the users, it means that testing DRAM address mapping tool becomes simple which requires just calling

some functions in Python bypassing the manual setup typically required. Besides reducing the setup time, our work also enhances the reproducibility by minimizing variations in build environments.
Moreover, actively using logging and error handling, we incorporated them into our library while studying possible problems that may occur during the compilation and execution phases. In particular, we worked to parse error outputs and provide substantial feedback to the users, which is especially valuable for debugging complex build issues. Our library also upholds customization, letting users to specify different makefile targets or extra configuration options, adaptable for various research needs. This customization along with the streamlined execution, makes the library compatible for integration into any automated workflow or larger data processing pipelines, which are often essential in DRAM research.
In addition to automating the setup, building and the execution of the DRAM address mapping tool, we moved a step further. Our library analyze the results produced and generate corresponding DRAM address functions. The the algorithm behind DRAMA tool outputs only the raw data regarding the DRAM address mapping which necessitates further interpretation to be practically useful for research and applications. By widening our library to process these results, we equip the users with a more complete workflow, letting them to move data directly to functional insights without the need for additional post-processing scripts.
Once the DRAMA algorithms completes its analysis and generates output data on address mapping, our library parses this data to recognize key patterns and relationships within the DRAM address space. Following the insights, the library then automatically constructs address mapping functions that translate between physical addresses and DRAM row, column, and bank coordinates. This step is very important, especially for the researchers who want to have efficient and reliable way to analyze or to simulate memory access patterns. By creating these functions automatically, we avoid the need for users to reverse-engineer DRAM address mappings, which otherwise tasks a significant amount of time and is prone to mistakes.
The address functions produced by our library are configured in a user-friendly manner that allows straightforward integration into various environments. In addition, these functions are outlined to be flexible for different DRAM configurations and addressing schemes. By producing DRAM address functions automatically, our library not only streamlines the use of the DRAMA tool but also enables researchers to incorporate DRAM-specific address mapping directly into larger projects, whether for performance profiling, security analysis, or architectural studies. This additional functionality puts out entire library as an extensive solution for both the generation and practical application of DRAM address mappings, as a result, it enhances the accessibility and impact of DRAM research.
The final result, our complete library is a highly accessible and customizable tool that empowers researchers to focus on the analysis and implementation of DRAM address mappings without being encumbered by the complexities of tool compilation

and setup. The final result, our complete library is a highly accessible and customizable tool that empowers researchers to focus on the analysis and implementation of DRAM address mappings without being encumbered by the complexities of tool compilation and setup. Through this library, we tend to make the DRAMA tool more accessible to the research community, potentially accelerating progress in fields related to memory performance, reliability, and security. Our code has been open sourced here.

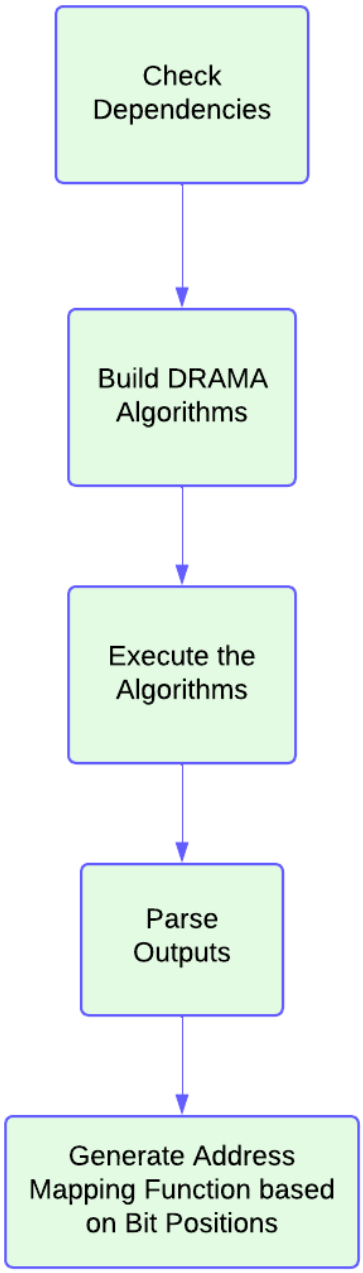

Figure 4.2: DRAM Address Mapping Library Process Flow

Once we are Ready with our DRAM address mapping library, we tested and analyzed its performance across three different INTEL architectures to observe the variations in DRAM address mappings and validate the library's effectiveness. Each of our tested architecture - Intel Celeron N4000 CPU (Goldmont Plus architecture), Intel Core i5-10500 CPU (Comet Lake architecture), and Intel Core i9-13900K (Raptor Lake architecture) shows a different generation and feature set, offering a broad spectrum of DRAM configurations.

4.2 Testing Process and Analysis

1. Running the DRAM Address Mapping Tool:

    - For each architecture, we executed the library's pipeline which involved checking dependencies, building the DRAMA algorithms, running it on the target CPU, parsing the output, and generating DRAM address mapping functions based on the extracted bit positions.
    - This programmatic process allowed consistent data acquisition across systems, ensuring that the observed mappings were directly comparable and free of user-induced variance.

2. Extracted DRAM Address Mapping:

    - The tool successfully parsed the DRAM address mappings, uncovering rows, columns, and bank configurations on our tested architectures. The bit position and length for each of these fields were analyzed, revealing architecture-specific nuances in DRAM address mappings.
    - To illustrate the N4000 (Goldmont Plus) CPU exhibited a simpler DRAM address layout with fewer row and column bits, the Comet Lake and Raptor Lake architectures presented more complex layouts due to their higher memory hierarchy sophistication and enhanced performance optimizations.

3. Generated Address Mapping Functions:

    - Our library generated address mapping functions tailored to each architecture. The differences in row, column, and bank bit positions observed across architectures necessitated unique bit-masking configurations for each function, allowing precise mapping for any given physical address.
    - The functions for the Raptor Lake architecture on CPU (i9-13900K) required handling a greater number of address bits, due to the higher DRAM bandwidth and parallelism in newer architectures. This function also accommodated more banks, supporting the increased memory access concurrency in the Raptor Lake design.

4.3 Analysis of Architectural Differences in DRAM Address Mapping

- **Goldmont Plus (N4000):** This architecture is typically used in low powered devices displpayed a minimalistic DRAM mapping structure. Relatively less address bits were dedicated to rows and columns, with limitations in bank parallelism, which aligns with its energy-efficient design goals. The address mapping function generated for this architecture was the simplest, requiring fewer bitwise operations.

- **Comet Lake (i5-10500):** As a widely used processor, the DRAM address mapping for Comet Lake demonstrated more complexity. The extracted bit positions indicated a balanced allocation between rows, columns, and banks, allowing for moderately parallel memory accesses. The generated address mapping function reflected this balanced approach, with more extensive bit extraction logic than the Goldmont Plus but simpler than the Raptor Lake.

- **Raptor Lake (i9-13900K):** As the latest high-performance architecture, Raptor Lake exhibited the most intricate DRAM address mapping scheme. This configuration allocates significant bits to banks and rows which supports high level parallelism and bandwidth in memory. The resulting address mapping function was the most complex, incorporating sophisticated bit masks and shifts to map addresses accurately within this high-concurrency memory model.

4.4 Insights and Implications

Our library's design flexibility allowed it to grasp varied DRAM address mapping requirements dynamically for each architecture. Through the automatic detection of parsing and function generation, the library delivered accurate address mapping functions for every CPU, which could be used directly to address research issues such as memory profiling or security testing. The differences that are observed in DRAM configurations across architectures emphasize the need for adaptable address mapping tools, particularly as memory architectures continue to evolve with each CPU generation.

Overall, this analysis illustrates the effectiveness of our DRAM address mapping library across diverse Intel architectures, enabling straightforward mapping function generation for different DRAM configurations. In our future analysis, we use this reverse engineered DRAM address functions in the fuzzer used for our research and studied the RowHammer effect in all three architectures.

Chapter 5

Our Analysis and Findings

In this chapter, we present the findings from our experimentation with the Blacksmith fuzzer across three different Intel architectures, implementing our DRAM address mapping library for each CPUs. After establishing accurate DRAM mappings for each architecture (Goldmont Plus, Comet Lake, and Raptor Lake), we implemented these mappings within the Blacksmith fuzzer to conduct rigorous memory access pattern testing. Our analysis focuses on various aggressor row activation patterns, refresh intervals, and activation counts to assess each architecture's susceptibility to row hammering effects.

5.1 Reproduction of Blacksmith Fuzzer

In order to reproduce and extend the capabilities of Blacksmith fuzzer over different DRAm configurations, we first had to integrate the architecture specific DRAM address functions generated for CPU model: Goldmont Plus (Celeron N4000), Comet Lake (Core i5-10500), and Raptor Lake (Core i9-13900K). By utilizing these mappings, the fuzzer acquired the accuracy required to conduct targeted RowHammering attacks based on the unique DRAM configurations of each CPU with distinguished bit allocation for rows, columns, and banks.

Our modification towards the architecture of the fuzzer included replacing its generic address translation logic with our custom mapping functions. This approach equips the fuzzer to identify and consistently target specific physical DRAM rows. This adaptation was necessary because row hammer vulnerabilities are highly architecture dependent and rely on memory layout knowledge. This changed arrangement enabled the fuzzer to target memory access patterns that trigger bit flips more efficiently.

The modified fuzzer harnessed the unique memory access sequence that derived from each architecture's DRAM configuration to maximize the chances of RowHammering adjacent rows. We incorporated aggressor row addresses that matched each architecture row, column, and bank layout to enhance the fuzzer RowHammering effect on each of the target systems. Further, in the fuzzer, we used a cycle-aware

mechanism that included the timing information of the clock speeds of each architecture and DRAM refresh time so as to arrive at a consistent architecture-tuned row activation during fuzzer testing.
In summary, the harnessing of our custom address mapping functions into the Blacksmith fuzzer allowed us to observe its RowHammering patterns to the unique DRAM layouts of each tested architecture. This adaptation provided a more accurate and effective approach to triggering row hammer vulnerabilities, setting the stage for detailed analysis on aggressor row activation counts, refresh intervals, and the efficacy of two-sided hammering across Goldmont Plus, Comet Lake, and Raptor Lake architectures.

5.2 Aggressor Row Vs Refresh Interval:

In our first fuzzer run, we will observed the number of aggressor row per refresh intervals for our tested machines. We define Goldmount Plus architecture as Module A, Comet Lake as Module B and Raptor Lake as Module C in the dataset. The number of aggressors stands along the Y asix where the x axis represents the refresh interval in milliseconds. The rows in the DRAM refreshes periodically in a specific time. The JEDEC standard for refresh interval is 64ms, but it may differ based on the temperature inside the DRAM. The increment in the aggresor rows number with prolonged refresh intervals indicates that the longer refresh intervals are more susceptible to RowHammer as they grant more time for row activations to aggregate. From the graph, we can see that Goldmount Plus architecture has the highest number of aggressor rows over the refresh intervals, stating that this architecture is the most sensitive to increased refresh intervals.
Comet Lake architecture follows a similar trend like the Goldmount Plus architecture but at a lower magnitude. The number of aggressors increases as the refresh interval extends. This tells that Comet Lake is having a moderate level of vulnerability to RowHammer effects than Goldmont Plus.

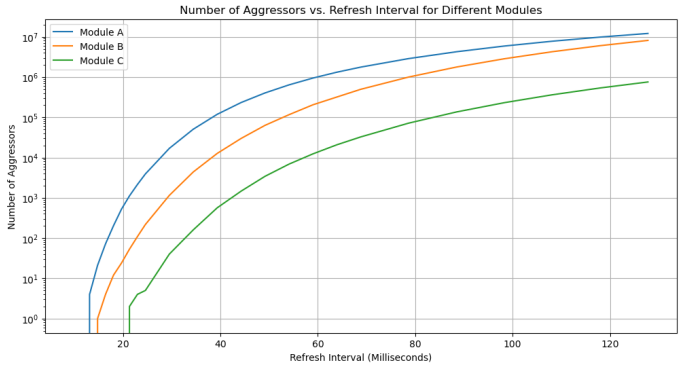

Figure 5.1: Aggressor rows per Refresh Intervals

Raptor Lake architecture has the lowest number of aggressor rows at each refresh interval which indicates that it has an improved resilience against RowHammer

attacks. Its curve shows a more steady increase where the aggressor rows remain under 10^4 even at the high refresh intervals.

5.3 Aggressor Row Vs Activation Interval:

In our second fuzzer run, we observed the number of aggressor rows per activation intervals. Once the fuzzer finds the aggressor row, it generates activation intervals against the refresh interval to make successful hammering in the targeted rows. Goldmount Plus shows the highest number of aggressor rows within the range of activation intervals with a sharp drop-off near 400 ms. Comet Lake architecture has fewer aggressor rows compared to Goldmount Plus architecture. Yet it is more that Raptor Lake which has the lowest aggressor per activation interval among all the tested architectures.

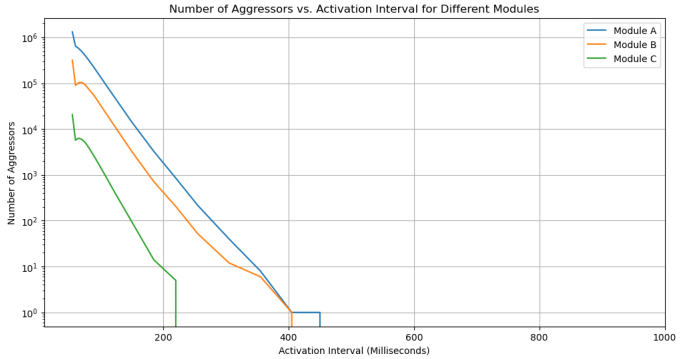

Figure 5.2: Aggressor rows per Activation Intervals

5.4 Bitflips and our Observations:

We know from our earlier discussions that RowHammer phenomenon occurs when bitflips occur in the victim cells of an aggressor row. It happens if the activations of aggressor row, also known as hammering, within the refresh intervals are successful. Our third and final fuzzer run counts the bitflips over victim cell per aggressor rows. Similarl to the previous results, the highest RowHammer susceptibility is found in the Goldmount Plus architercute as the highest number of bitflips are found in this architecture which raised up 20 victim bits per row. After reaching the peak, the number of victim cells started showing a downward tend. Comet lake shows a similar tend to Goldmount Plus but with a lower magnitude of bitflips where the bitflips peak is around 10 to 20 victim cells per aggressor row. Raptor Lake has the least number of calculated bitflips which means it is less susceptible to RowHammer impact. Its Bitflips peak was at fewer than 10 victim cells per row.
Based on our experiments, we can come to the decision that Goldmont Plus appears to be the most susceptible to RowHammer and Raptor Lake demonstrates the best

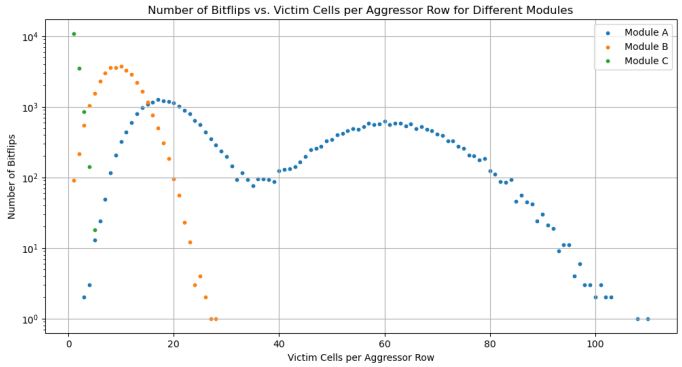

Figure 5.3: Bitflips per Aggressor rows

resilience. Our study on overall RowHammer effect in our chosen architecture give a bunch on interesting insights. First of all, out decision is not just based on the number of bitflips per aggressor rows, yet it was based on the overall RowHammer attack surface to a targeted machine. We started our experiment from studying the refresh interval, then activation interval and finally the bitflips and throughout all the layer of experiments, we Goldmount Plus architecture was easy to break into. That means, the architecture failed to protect itself from RowHammer since the beginning, where as the protection against RowHammwer for Raptor Lake remained very strong throughout all the experiments. It seems like that the architecture has it own shield against the RowHammer. However, the Coffee Lake architecture remained lenient. It was neither too weak to be a victim of RowHammer, not it had its strong position against RowHammer which signifies that a sophisticated RowHammer attack is possible by targeting a machine based on Coffe Lake architecture. Also, among all the tested architectures, Goldmount Plus is the oldest and Raptor Lake is the newest. So we can also claim that, the vendors are taking RowHammer seriously and working on their protection against it. However, despite of the strengths and weaknesses of our targeted architectures, from our last experiment, by observing the declination of bitflips once it hits the peak, we can clearly say that regardless of the manufacturing date, all the architectures has the mechanism to act against RowHammer. In our future work, we are looking forward to count the number of clock cycles it takes per each activations over across the architectures which may lead to a completely new insignts. We claimed that the most recent architecture, Raptor Lake is relatively secured than the other two, yet, if it's clock cycle rate per activation is higher than the other two, it will be equally prone to RowHammer vulnerability regardless of it's modern protection mechanisms.

Chapter 6

Conclusion

Our study demonstrates the effectiveness of a custom library built for reverse engineering DRAMM address functions which enables precise mapping of DRAM rows to physical addresses across architectures.By integrating these mappings with the Blacksmith fuzzer and running it on our tested machines, we uncovered key observations in RowHammer vulnerability. Our experiments showed that the hammering to the rows and the occurrence of bitflips are significantly differently across each of the architectures based on memory controller designs, cache structures, and DRAM row address mappings.
The library proved essential in optimizing the address extraction process which enables a streamlined workflow for RowHammer analysis across diverse hardware setups. These findings emphasizes the necessity of architecture-aware testing tools for accurately assessing Rowhammer susceptibility and provide valuable insights for designing DRAM configurations and memory controllers that are more resilient to such attacks. Ultimately, this research contributes to the development of standardized, cross-platform approaches for assessing and mitigating hardware vulnerabilities, advancing efforts to safeguard next-generation systems against emerging threats.

Bibliography

[1] J. A. Mandelman, S. H. Lee, D. H. Lu, R. R. M. L. McGuire, and C. M. N. N. Z. Y. D. Reiner, "Challenges and future directions for the scaling of dynamic random-access memory (dram)," *IBM Journal of Research and Development (IBM JRD)*, vol. 46, no. 2, pp. 1–10, 2002.

[2] S. Hong, "Memory technology trend and future challenges," in *Proceedings of the 2010 IEEE International Electron Devices Meeting (IEDM)*, IEEE, 2010, pp. 1–4.

[3] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *Proceedings of the 2010 ACM/IEEE International Symposium on Microarchitecture (MICRO)*, IEEE, 2010, pp. 1–12.

[4] JEDEC, "Double data rate 4 (ddr4) sdram standard," JEDEC, Tech Report, 2012. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd79-4.

[5] O. Mutlu, "Memory scaling: A systems architecture perspective," in *Proceedings of the 2013 IEEE International Memory Workshop (IMW)*, IEEE, 2013, pp. 1–6.

[6] S. J. Vaughan-Nichols, "The secret origins of google's chrome os," *ZDNet*, Mar. 2013. [Online]. Available: https://archive.is/TODn1.

[7] Y. Kim, O. Mutlu, D. Burger, and L. W. Smith, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2014, pp. 1–12.

[8] K. Bains *et al.*, "Row hammer refresh command," US Patent 9,117,544, Filed in 2015, 2015.

[9] K. S. Bains *et al.*, "Row hammer monitoring based on stored row hammer threshold value," US Patent 9,032,141, Filed in 2015, 2015.

[10] S. Mark and T. Dullien, *Exploiting the dram rowhammer bug to gain kernel privileges: How to cause and exploit single bit errors*, Black Hat USA, Las Vegas, NV, Aug. 2015, 2015. [Online]. Available: https://www.youtube.com/watch?v=0U7511Fb4to.

[11] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," in *Black Hat*, vol. 15, 2015, p. 71.

[12] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "Anvil: Software-based protection against next-generation rowhammer attacks," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.

[13] S. Bhattacharya and D. Mukhopadhyay, "Curious case of rowhammer: Flipping secret exponent bits using timing analysis," in *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, Springer, 2016, pp. 602–624.

[14] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA: IEEE, 2016, pp. 987–1004. [Online]. Available: http://ieeexplore.ieee.org/document/7546546/.

[15] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Can't touch this: Practical and generic software-only defenses against rowhammer attacks," *arXiv preprint arXiv:1611.08396*, 2016.

[16] D. Gruss, C. Maurice, S. Mangard, and T. Eisenbarth, "Rowhammer.js: A remote software-induced fault attack in javascript," *CoRR*, vol. abs/1605.09536, 2016. [Online]. Available: https://arxiv.org/abs/1605.09536.

[17] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks," in *Proceedings of the USENIX Security Symposium*, USENIX, 2016, pp. 565–581.

[18] R. Qiao and M. Seaborn, "A new approach for rowhammer attacks," in *Proceedings of the 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, IEEE, 2016, pp. 161–166.

[19] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *Proceedings of the 25th USENIX Security Symposium (USENIX Security 16)*, USENIX, 2016, pp. 1–18.

[20] V. V. D. Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 1675–1689.

[21] Y. Xiao, T. Eisenbarth, B. Sunar, M. Lipp, and D. Gruss, "One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation," in *Proceedings of the 2016 USENIX Security Symposium*, USENIX Association, 2016, pp. 1–16.

[22] O. Mutlu, "The rowhammer problem and other issues we may face as memory becomes denser," in *Proceedings of the 2017 Design, Automation Test in Europe Conference (DATE)*, IEEE, 2017, pp. 1–6.

[23] D. Gruss, C. Maurice, S. Mangard, and T. Eisenbarth, "Another flip in the wall of rowhammer defenses," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 1–18.

[24] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating software mitigations against rowhammer: A surgical precision hammer," in *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, Springer, 2018, pp. 47–66.

[25] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "Guardion: Practical mitigation of dma-based rowhammer attacks on arm," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Springer, 2018, pp. 92–113.

[26] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 55–71.

[27] L. Cojocar, M. Neugschwandtner, M. Schwarz, D. Gruss, and K. Razavi, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, 2019, pp. 55–71.

[28] V. Corporation, *Blackbird bios reference manual*, Accessed: 2024-11-09, 2019. [Online]. Available: https://www.versalogic.com/wp-content/themes/vsl-new/assets/pdf/manuals/MEPU44624562BRM.pdf.

[29] S. Ji, Y. Zhang, Y. Wu, X. Zhang, and Z. Zhan, "Pinpoint rowhammer: Suppressing unwanted bit flips on rowhammer attacks," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (ASIACCS)*, ACM, 2019, pp. 1–14.

[30] T. Yang, J. Li, S. K. S. S. Rajendran, M. Yu, and Y. K. M. T. D. A. Robinson, "Trap-assisted dram row hammer effect," *IEEE Electron Device Letters (EDL)*, vol. 40, no. 1, pp. 1–4, 2019.

[31] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, and Z. Wang, "Telehammer: A stealthy cross-boundary rowhammer technique," *arXiv preprint arXiv:1912.03076*, 2019.

[32] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are we susceptible to rowhammer? an end-to-end methodology for cloud providers," *arXiv preprint arXiv:2003.04498*, 2020.

[33] P. Frigo, M. Lipp, E. W. N. K. H., T. Eisenbarth, and B. Sunar, "Trrespass: Exploiting the many sides of target row refresh," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1–18.

[34] J. S. Kim, M. Patel, A. G. Yaglıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020, pp. 638–651.

[35] A. Kwong, M. Lipp, T. Eisenbarth, and B. Sunar, "Rambleed: Reading bits in memory without accessing them," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1–18.

[36] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet lightweight row hammer protection," in *Proceedings of the 2020 ACM/IEEE International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, p. 13. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9251863.

[37] TQ-Systems, *Tqmx80uc user's manual*, Accessed: 2024-11-09, 2020. [Online]. Available: https://www.tq-group.com/fileadmin/downloads/files/products/embedded/manuals/x86/embedded-modul/COM-Express-Compact/TQMx80UC/TQMx80UC.UM.0102.pdf.

[38] Z. Zhang, Y. Cheng, and S. Nepal, "Ghostknight: Breaching data integrity via speculative execution," *arXiv preprint arXiv:2002.00524*, 2020.

[39] S. Qazi, Y. Kim, N. Boichat, E. Shiu, and M. Nissler, *Introducing half-double: New hammering technique for dram rowhammer bug*, 2021.

[40] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized many-sided rowhammer attacks from javascript," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[41] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable rowhammering in the frequency domain," in *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022.

[42] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "Spechammer: Combining spectre and rowhammer for new speculative attacks," in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022, pp. 681–698.

[43] C. Group, *Blacksmith: Scalable rowhammering in the frequency domain*, Accessed: 2024-11-09, 2023. [Online]. Available: https://github.com/comsec-group/blacksmith.

[44] IAIK, *Drama: Dynamic row address mapping tool*, Accessed: 2024-11-09, 2023. [Online]. Available: https://github.com/IAIK/drama.