Development and evaluation of solutions for the protection of DRAM and MRAM memories against Rowhammer attacks

Loïc France

▶ To cite this version:

Loïc France. Development and evaluation of solutions for the protection of DRAM and MRAM memories against Rowhammer attacks. Cryptography and Security [cs.CR]. Université de Montpellier, 2022. English. NNT : 2022UMONS086 . tel-04117848

HAL Id: tel-04117848

https://theses.hal.science/tel-04117848

Submitted on 5 Jun 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESIS TO OBTAIN THE DEGREE OF DOCTOR OF THE UNIVERSITY OF MONTPELLIER

In SyAM - Automatic and Microelectronic Systems

Doctoral school: Information, Structures and Systems sciences

Research Unit: LIRMM

Development and evaluation of solutions for the protection of DRAM and MRAM memories against Rowhammer attacks

Presented by Loïc France

December 16, 2022

Under the supervision of Pascal Benoit and Florent Bruguier

Thesis Committee:

Pascal BENOIT , Associate Professor , University of Montpellier	Thesis Director
Florent BRUGUIER , Associate Professor , University of Montpellier	Thesis Co-supervisor
Jean-Luc DANGER , Professor , Telecom-ParisTech	Examiner
Giorgio DI NATALE , Research Director , University of Grenoble Alpes	Reporter
Guy GOGNIAT , Professor , University of Bretagne Sud	Reporter
Maria MUSHTAQ , Associate Professor , Telecom-ParisTech	Examiner
David NOVO , Research Scientist , CNRS	Examiner
Lionel TORRES , Professor , University of Montpellier	Examiner

UNIVERSITÉ DE MONTPELLIER

Abstract

Modern computer memories have shown reliability issues. The main memory is the target of a security threat called Rowhammer, which takes advantage of cell-to-cell disturbance between DRAM rows to cause bit-flips in adjacent victim cells of repeatedly activated aggressor rows. Moreover, as DRAM manufacturers keep increasing the memory density to improve efficiency and reduce cost, the disturbance between cells gets more important over the years, worsening the threat. The abundant research on this subject led to the development of numerous countermeasures. Each proposal comes with pros and cons in terms of modularity, performance cost, silicon area, and energy overheads, with implementation in software, hardware or both. The development of hardware-based mitigation techniques can be made easier with a computer architecture simulator such as gem5, which facilitates the development of computer architectures that integrate new hardware components with existing and future memories or other elements. However, existing simulators are not suitable for Rowhammer mitigation development, as they cannot simulate memory corruption from Rowhammer attacks, which makes verifying mitigation techniques more difficult.

In this work, we first improve the open-source simulator gem5 to make it a complete tool for Rowhammer mitigation development. We add a memory corruption module that is able to simulate the bit-flips caused by Rowhammer attacks, with various parameters to adapt it to mature and future memories, and utility functions to facilitate the integration and evaluation of mitigation techniques into the architecture. Then, we study how changing the counting granularity of counter-based Rowhammer mitigation proposals could reduce their storage requirements. Some of the most efficient proposals rely on row activation counters, using for example Counting Bloom Filters or the Misra-Gries algorithm. We demonstrate that those proposals can have their storage requirements reduced by 40% to 50% without impacting the protection level, by changing their counting granularity from bank-level to rank-level. Additionally, we propose two new Rowhammer detection mechanisms. We show that by including hardware event counters inside the architecture, a machine-learning algorithm implemented in the hardware can classify traces from these counters to detect Rowhammer attacks. We also propose a new detection mechanism that evaluates the

activation frequency of every DRAM row to accurately detect aggressor rows and prevent the corruption, with an adaptive energy consumption. As a line of research, we explore the vulnerabilities of emerging non-volatile memories to variations of the Rowhammer attack. Through power consumption analysis, we retro-engineer the internal architecture of commercial Toggle-MRAM and STT-MRAM to design attacks that are susceptible to produce bit-flips in the memory, and execute these attacks to check if they can corrupt the memory.

Résumé de la thèse

Les mémoires des ordinateurs modernes sont sujettes à des problèmes de fiabilité. La mémoire principale est la cible d'une attaque appelée Rowhammer, qui exploite les perturbations électriques entre les lignes de la DRAM pour corrompre la donnée stockée dans les cellules voisines de cellules activées fréquemment. De plus, la densité des mémoires augmentant avec l'amélioration des technologies de fabrication pour optimiser l'efficacité et réduire les coûts, les effets des perturbations entre les lignes sont de plus en plus importants, empirant la menace. La recherche abondante sur ce sujet a conduit au développement de nombreuses contremesures. Chaque proposition a ses propres avantages et inconvénients en termes de modularité, d'impact sur les performances, sur la surface de silicium ou la consommation énergétique, avec des implémentations purement logicielles, matérielles ou un mélange des deux. Le développement de contremesures peut être facilité par l'utilisation d'un simulateur d'architecture tel que gem5, qui simplifie l'intégration de nouveaux composants matériels avec différentes technologies de mémoires et de processeurs. Cependant, les simulateurs existants étant incapables de simuler la corruption de la mémoire produite par une attaque Rowhammer, ils ne sont pas appropriés pour le développement de protections contre Rowhammer car ils ne permettent pas de vérifier l'absence de corruption.

Dans un premier temps, nous proposons une amélioration du simulateur ouvert gem5 pour en faire un outil de développement de contremesure contre Rowhammer. Nous y incorporons un simulateur de corruption de mémoire capable de simuler des bit-flips causés par les attaques Rowhammer, configurable en plusieurs points pour

l'adapter aux mémoires existantes et futures, et fournissant les fonctions nécessaires à l'intégration et l'évaluation de contremesures dans l'architecture. Dans un deuxième temps, nous étudions comment le changement de granularité de comptage peut réduire les besoins en mémoire de certaines contremesures contre Rowhammer. Parmi les propositions de contremesures les plus efficaces à ce jour, certaines utilisent des compteurs d'activations des lignes de la DRAM, en utilisant par exemple les compteurs à filtre de Bloom ou l'algorithm de Misra-Gries pour réduire les besoins en mémoire. Nous démontrons que ces propositions peuvent avoir leurs besoins en mémoire réduits de 40% à 50% sans impacter le niveau de protection, en modifiant la granularité de comptage du niveau *bank* au niveau *rank* de l'architecture de la DRAM. Dans un troisième temps, nous proposons deux nouveaux mécanismes de détection des attaques Rowhammer. La première proposition utilise un algorithme de machine-learning ainsi que des compteurs d'événements introduits dans l'architecture pour détecter les attaques Rowhammer. La seconde proposition combine le comptage des activations et l'évaluation de la fréquence d'activation de toutes les lignes de la DRAM pour détecter les attaques. Dans un quatrième et dernier temps, nous explorons les vulnérabilités des mémoires émergentes non-volatiles aux variations de l'attaque Rowhammer. En analysant la consommation d'une mémoire externe de technologies Toggle-MRAM et STT-MRAM lors de différentes opérations de lecture et d'écriture, nous reconstituons une partie de l'architecture interne de ces mémoires, afin de concevoir des attaques susceptibles de corrompre les données stockées, puis exécutons ces attaques pour évaluer la vulérabilité de ces mémoires.

Acknowledgements

This thesis was made possible with the support of many people, which I would like to thank.

Foremost, I want to express my gratitude to my supervisors Pascal Benoit, Florent Bruguier, Maria Mushtaq and David Novo, who all offered assistance in various fields. Their support during those three years has been a major contribution to the success of my thesis. First, my thesis director Pascal Benoit, for his mentorship, scientific support and guidance. The regular meetings we had maintained a productive work throughout the duration of the thesis Then, Florent Bruguier, who brought this thesis subject to me, and was my day-to-day contact to discuss about all aspects of my contributions and publications; Maria Mushtaq, who offered her helped at the beginning of my thesis, participated in the development of my first contribution and continued to follow me to the end of my thesis; and last but not least David Novo, for his listening, scientific knowledge and help on the publications to bring the best out of them.

I am grateful for my PhD student colleagues Theo Soriano and Paul Delestrac with whom I shared the office during this three years and regularly brainstormed on research subjects, and Quentin Huppert for accompanying us through our theses; I also thank my other colleagues Jonathan Miquel, Geoffrey Chancel and Julien Toulemont who shared lunch and coffee breaks with us.

I extend my appreciation to the teachers of my former engineering school Polytech Montpellier, especially Laurent Latorre, Eric Dubreuil and Guy Cathébras, who offered me the possibility to give programming lessons to students, which played an important part of the last three years.

I want to thank former Engineer student Constantin Gaboury for his contribution in the experiments on STT-MRAM memories for his end-of-study project.

I am truly thankful to the reviewers of this manuscript Guy Gogniat and Giorgio Di Natale for the time they took to review my thesis manuscript and their helpful feedback, and all jury members presided by Lionel Torres for their interest in my work, their interesting questions and suggestions.

Finally, I express my gratitude to my family and friends who have supported me up to this day, and have without a doubt contributed to this achievement.

The authors acknowledge the support of the French Agence Nationale de la Recherche (ANR), under grant ANR-19-CE39-0008 (project ARCHI-SEC). They also acknowledge the French Ministère des Armées – Agence de l'innovation de défense (AID) under grant ID-UM-2019 65 0036.

Contents

List of Figures **xiii**

List of Tables **xv**

List of Algorithms **xvii**

List of Acronyms **xix**

1 Introduction **1**

1.1 Context 2

1.2 Contribution 3

2 Background **5**

2.1 Memory Architecture and operation 6

2.1.1 Memory architecture, Core to main memory read operation . . . 6

2.1.2 DRAM architecture and operation 7

2.2 Rowhammer attack 13

2.2.1 Cell-to-cell disturbance 13

2.2.2 Basic exploit 15

2.2.3 Exploits in literature 17

2.3 Rowhammer countermeasures 19

2.3.1 Basic Principles 19

2.3.2 Software-based protection 20

2.3.3 Hardware probabilistic protection 20

2.3.4 Hardware counter-based protection 21

2.3.5 Conclusion 23

2.4 Conclusion 23

3 Rowhammer Simulation **25**
3.1 Motivation 26
3.2 Rowhammer simulation requirements 28
3.3 gem5 and Ramulator 31
3.4 Memory Corruption simulation 33
3.4.1 Integration of the memory-corruption module in gem5 and Ramulator 33
3.4.2 Disturbance and corruption simulation 34
3.5 Mitigation integration in gem5 37
3.6 Usage, limitations and evaluation 38
3.6.1 Limitations 40
3.6.2 Evaluation 41
3.7 Conclusion 42

4 Counter-based Rowhammer mitigations improvement **43**
4.1 Motivation 44
4.2 Bank-level and rank-level counting granularity 44
4.3 Implication in State-of-the-art mitigation proposals 46
4.4 Considerations for technology and timings 49
4.4.1 DDR generation parameters 49
4.4.2 Feasibility - timing considerations 50
4.5 Conclusion 51

5 Mitigation proposals **53**
5.1 Motivation 54
5.2 Hardware counters and machine learning for Rowhammer detection . . 54
5.2.1 Methodology 55
5.2.2 Experiments and results 57
5.2.3 Conclusion 60
5.3 F-CorD: Forgetful Counters for Rowhammer Detection 61
5.3.1 Introduction: Unsynchronised refresh issue for counter-based Rowhammer mitigation 61
5.3.2 Tracking frequently-activated rows 62

5.3.3 Detecting attacks 64
5.3.4 Discretisation 68
5.3.5 Periodic maintenance 69
5.3.6 Implementation details 71
5.3.7 Number of entries 73
5.3.8 Example 74
5.3.9 Conclusion 75

6 Experiments on MRAM **77**

6.1 Motivation 78
6.2 Attack on a Toggle-MRAM chip 82
6.2.1 Platform Requirements 82
6.2.2 Test Platform 83
6.2.3 Reverse-engineering of the memory module architecture 86
6.2.4 Designing the attack 88
6.2.5 Results 90
6.3 Attack on an STT-MRAM chip 91
6.3.1 Test Platform 91
6.3.2 Reverse-engineering of the memory module architecture 92
6.3.3 Designing the attack 94
6.3.4 Results 94
6.4 Conclusion 95

7 Conclusion and Perspectives **97**

7.1 Contributions 98
7.1.1 Rowhammer simulation in gem5 98
7.1.2 Improvement of Rowhammer mitigations 99
7.1.3 Mitigation proposals 99
7.1.4 Experiments on MRAM 99
7.2 Future work 100
7.3 Concluding remarks 101

Bibliography **103**

List of Figures

2.1 Memory architecture for run-time data in modern computers 7
2.2 Read operation flowchart (simplified) from CPU Core to main memory . 8
2.3 Address bit layouts 8
2.4 Processor to DRAM bank architecture 9
2.5 Bank-level states and data-access-related commands 10
2.6 ACTIVATE-PRECHARGE cycle 11
2.7 Illustration of DDR timings 13
2.8 Charge trapping and migration to victim memory cells 14
2.9 Memory cell capacitor voltage evolution under normal behaviour and under disturbance 14
2.10 Memory bank under attack 16

3.1 gem5 and Ramulator memory architecture 32
3.2 gem5 and Ramulator memory architecture, with the Memory-Corruption module 34
3.3 ACT and REF callback functions 35
3.4 disturbance simulation 36
3.5 Bit-flip simulation with polynomial equation 37
3.6 gem5 and Ramulator memory architecture, with Memory-Corruption module and Rowhammer mitigation 38

4.1 Alternating CBFs of BlockHammer 48

5.1 Machine Learning detection mechanism integration in computer architecture 57
5.2 Counters reset not synchronised with row refresh 62
5.3 Evolution of the remaining time for one row in the table 64
5.4 Evolution of the counter for one row in the table 65

5.5 Evolution of the value for one row in the table 66
5.6 Split attack on F-CoRD 67
5.7 Slow attack with discrete timings 69

6.1 Bit-cells of Emerging NVMs 79
6.2 STT-MRAM structure and write process 80
6.3 Toggle-MRAM structure and write process 80
6.4 Cell-to-cell magnetic disturbance on MRAM 81
6.5 Rowhammer effect on STT-MRAM 82
6.6 Simplified architecture of the Toggle-MRAM module 83
6.7 FPGA-based platform peripherals architecture 84
6.8 Attack pattern on Toggle-MRAM 89
6.9 Microcontroller-based platform architecture 92
6.10 STT-MRAM energy levels 93
6.11 Hypotyhesis on the order of row and column bits in the address vector of the stt-mram module 94

List of Tables

2.1 DRAM timing parameters 12

3.1 Architecture Simulators comparison 30
3.2 DRAM layout configuration example. 40
3.3 Impact of the Memory-Corruption module (M-C) on simulation performance. 42

4.1 Values of W_B and W_R, number of banks per rank, and theoretical reduction of total number of counters, for DDR3, DDR4 and DDR5 (c.f., Table 2.1 page 12). 46
4.2 Values of W_B and W_R, number of banks per rank, and theoretical reduction of total number of counters, for DDR3, DDR4 and DDR5 (c.f., Table 2.1 page 12). 50

5.1 ML models categorisation accuracy 60
5.2 minimum table size for the F-CoRD implementationon DDR3, DDR4 and DDR5, for a bank-level implementation counting granularity. 75

6.1 Current consumption of Toggle-MRAM memory module for read operations 87
6.2 Toggle-MRAM deducted address layout 87
6.3 Current consumption of Toggle-MRAM memory module for write operations 88
6.4 current measurements during read and write operations on the STT-MRAM memory module 93

List of Algorithms

1 F-CoRD global algorithm 72

2 Calculate the required number of entries of F-CoRD 73

3 Attack algorithm on Toggle-MRAM 89

List of Acronyms

ACT	Activate command
ALU	Arithmetic Logic Unit
BL	Bit Line
CAM	Content-addressable Memory
CBF	Counting Bloom Filter
CNN	Convolutional Neural Network
CSL	Column Selection Logic
CPU	Central Processing Unit
DDR	Double Data Rate (memory)
DIMM	Dual In-line Memory Module
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
EMC	External Memory Controller
FN	False Negative
FP	False Positive
FPGA	Field-Programmable Gate Array
HPC	Hardware Performance Counter
LLC	Last-Level Cache
LPDDR	Low Power DDR memory
LSTM	Long Short-Term Memory
MAT	Memory Array Tile
ML	Machine Learning
MLP	Multi-Layer Perceptron
MRAM	Magnetic Random Access Memory
MTJ	Magnetic Tunnel Junction

NVM	Non-Volatile Memory
OS	Operating System
PRE	Precharge command
PRNG	Pseudo-Random Number Generator
RAM	Random Access Memory
RB	Row Buffer
REF	Refresh command
RFM	Refresh Management command
RTL	Register-Transfer Level
SA	Sense Amplifier
STT-MRAM	Spin-Transfer Torque MRAM
WL	Word Line

I

Introduction

Contents

1.1 Context **2**

1.2 Contribution **3**

1.1 Context

Memory is a key component of computing systems. It is used to store the input values, intermediate variables and end results of any program. Over the years, following the Moore's Law, computing systems have become exponentially more powerful, allowing computers to run faster, with an increasing number of more and more complex programs running in parallel. Handling this complexity and speed requires the memory to become faster, with an ever-increasing capacity to store the large quantity of data programs need during run-time. The most popular technology nowadays for the main memory is the Dynamic Random Access Memory (DRAM). This technology uses capacitors to store the data: the charge of each capacitor determines the value of the bit it stores. Again, following the Moore's law, DRAM manufacturers have been able to increase the density of the memory, reducing the production cost per bit and allowing more data to be stored in the same silicon area.

However, as the memories became denser, storage capacitors became smaller, and their maximum capacity reduced. Additionally, physical phenomenon that originally had no consequence on the behaviour of the devices started to cause reliability issues. In 2014, Kim et al. [1] published the first public paper on cell-to-cell disturbance errors caused by DRAM row activations. When a DRAM row is activated, undesired electric communication between adjacent DRAM rows disturb the storage nodes of adjacent rows. Capacitors of victim storage nodes, which by nature already leak charge over time, see their charge reduced by a small amount when disturbed. This additional leakage is not important and do not compromise the data by itself, as capacitors are periodically refreshed to compensate their natural leakage. However, repeated activations of the neighbouring rows can cause this disturbance to progressively empty the victim capacitors, effectively flipping the bits.

The authors showed that this phenomenon can be easily leveraged by a malicious program to mount the Rowhammer attack. Over the next years, several contributions showed that this attack can be improved to cause major issues in modern computers, ranging from discovering encryption keys [2] to remotely disabling systems [3] or performing privilege escalation from web browsers [4]. In the same time, various mit-

igation techniques were proposed by the community or the manufacturers to prevent those attacks from harming the systems. Some proposals are implemented in software, and try to detect the attack processes from their abnormal memory activity; some are implemented in hardware, in the memory controller or in the memory modules, and refresh the potential victim rows before they experience bit-flips; finally, some mitigation proposals are implemented in both hardware and software, detecting potential aggressor rows using hardware components and asking the software to stop or slow down the suspicious processes. In addition to the variety of implementation levels, we see a variety of detection methods. Some proposals exploit the already-implemented Hardware Performance Counters (HPC) to detect suspicious memory activity; some implement row activation counters to detect rows that are activated too many times; some use probabilistic mechanism to prevent aggressor rows to corrupt neighbour rows; finally, some exploit specific mechanisms such as empty "barrier" DRAM rows, or randomly swapped rows to prevent aggressor from corrupting critical data.

However, as DRAM manufacturers continue to increase the storage density, attacks are made easier. The minimum amount of activations to produce a bit-flip has drastically reduced, and it became possible to corrupt rows that are not immediately adjacent to the aggressor rows. Hence, most mitigation proposals become useless against recent attack. Manufacturer's proprietary mitigation techniques are proved to be ineffective, and some proposals incur increasing performance and silicon area overhead to stay effective on more and more vulnerable memories. Therefore, Rowhammer mitigation is still to this day an important issue that requires extensive research.

1.2 Contribution

During the development of a Rowhammer mitigation technique, testing on a working system is an important step, that helps optimising the solution and permits the validation against state-of-the-art attacks. However, most proposals require modifications of hardware components, sometimes in the DRAM, and therefore cannot be implemented on existing systems. For our first contribution, we present a simulation tool to develop and validate Rowhammer attacks on a configurable computer architecture. The tool

simulates the memory corruption caused by Rowhammer attacks, and provides tools to implement various mitigation techniques.

Counter-based hardware mitigation techniques are among the most efficient Rowhammer mitigation techniques. Most of them are implemented with a bank-level counting granularity, which means that each bank has a dedicated set of counters. However, due to limitation at the rank level, all banks of the memory cannot be accessed at their maximum frequency. For our second contribution, we propose to change the counting granularity of some mitigation proposals to rank level to reduce the required number of counters, ultimately reducing the dedicated memory size by 40% to 50%.

Rowhammer mitigation is still an open research. Numerous mitigation techniques are proposed every year, using different algorithms to detect the attacks while trying to reduce the performance and area overhead to the minimum. For our third contribution, we propose two new detection mechanisms. Machine learning has not been extensively studied as a detection mechanism. In our first proposal, we demonstrate that machine learning can be exploited as a hardware component to detect Rowhammer attacks using counters of specific micro-architecture events. For our second proposal, we describe a new counter-based mitigation technique that uses the activation frequency of each row to detect aggressors.

Recent memory technologies such as the various Magnetic Random Access Memory (MRAM) variants show promising characteristics as DRAM replacements. However, the vulnerability of this new technology has not been extensively studied. Previous studies have shown that these technologies may be vulnerable to cell-to-cell disturbance error. In our last contribution, we try to attack two commercially-available MRAM external memory modules using custom platforms. We show that even if the attacks we try are not able to produce bit-flips in the memory, we can successfully retro-engineer some information about the internal structure and behaviour of the memory modules, that could be leveraged to perform side-channel attacks.

II

Background

Contents

2.1 Memory Architecture and operation **6**

2.1.1 Memory architecture, Core to main memory read operation . 6

2.1.2 DRAM architecture and operation 7

2.2 Rowhammer attack **13**

2.2.1 Cell-to-cell disturbance 13

2.2.2 Basic exploit 15

2.2.3 Exploits in literature 17

2.3 Rowhammer countermeasures **19**

2.3.1 Basic Principles 19

2.3.2 Software-based protection 20

2.3.3 Hardware probabilistic protection 20

2.3.4 Hardware counter-based protection 21

2.3.5 Conclusion 23

2.4 Conclusion **23**

2.1 Memory Architecture and operation

2.1.1 Memory architecture, Core to main memory read operation

In modern computing systems such as consumer computers, memory is a key component. During run-time, the largest portion of the data used by running programs is stored in a Random-Access Memory (RAM). Data transit between the main memory and the processor through multiple cache memory levels, integrated in the processor. These cache memories are faster to access than the main memory, but have much lower capacity. They are organised in levels, typically from L1 to L4. Higher cache levels have more storage capacity, but are slower to access. The cache memories are used to store the most frequently used data. Ideally, the more frequently used are the data, the lower they are stored in the cache levels. While the last levels are global to the processor, each core has its own sets of first level caches (typically L1 and L2). Moreover, the first cache level is split into instructions cache and data cache. Finally, each core has its own register file, which it uses to store data when performing operations on it. Register files have a very low capacity, but are directly connected to the Arithmetic Logic Units (ALU), and therefore provide very low access time. When the registers are needed to store other data, this data is stored back in the address space mapped to the RAM which is external to the processor. When a requested data is not stored in a cache memory, it is fetched from the RAM. When a data is evicted from the cache, it is stored back in the RAM.

If the RAM (or main memory) capacity is not sufficient, a SWAP partition can be used in the hard drive to temporarily store run-time data that is more rarely used. The memory architecture is illustrated in Figure 2.1.

When the core requests some data from the memory, it usually follows the process illustrated in Figure 2.2. The request first goes to the first cache level (L1-I for instructions, or L1-D for other data). If the data is stored in this cache, it is returned. Otherwise, the request goes to the next cache level, on which the same test is performed. If the data is not stored at any level of the cache memories, the request is passed on to the main memory. After fetching from a higher cache level or the main memory, the new

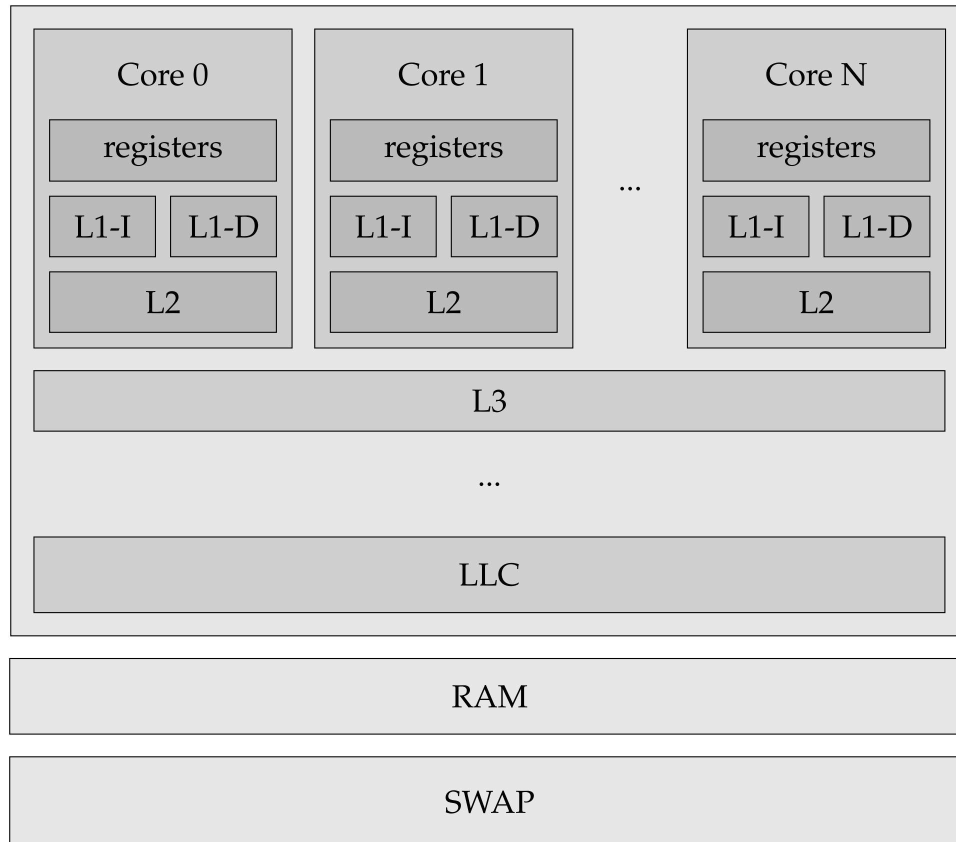


Figure 2.1: *Memory architecture for run-time data in modern computers*

data can be stored in the current cache, replacing older data following some eviction rules. We call the event of a data found in a cache memory a cache hit, and the data missing in a cache memory a cache miss.

2.1.2 DRAM architecture and operation

In consumer computers, the technology used for the main memory is the Dynamic Random Access Memory (DRAM) technology.

In this memory, the data is addressed using the following architecture levels, from higher to lower: channel, rank, bank group, bank, row and column. Each level has its own bits in the address word, depending on the chosen layout. Two common address layouts, ChRaBaRoCo and RoBaRaCoCh (**Ch**: Channel, **Ra**: Rank, **Ba**: Bank, **Ro**: Row, **Co**: Column, Bank group is omitted), are illustrated in Figure 2.3.

The architecture of a DRAM main memory is illustrated in Figure 2.4. The proces-

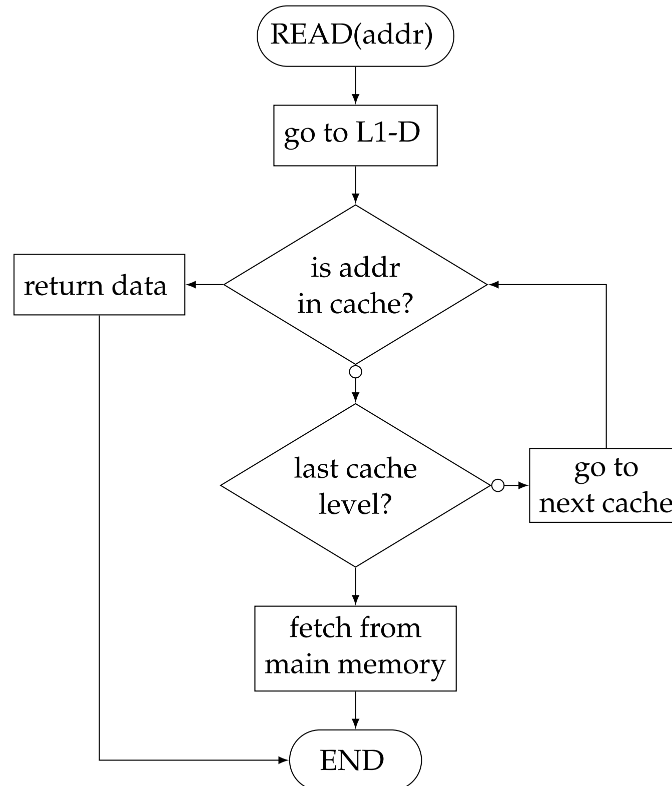
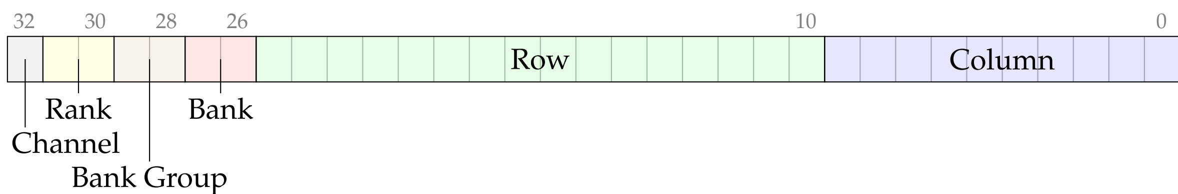
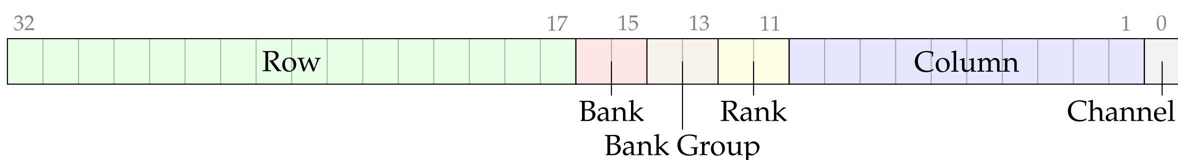


Figure 2.2: *Read operation flowchart (simplified) from CPU Core to main memory*



(a) *ChRaBaRoCo Layout*



(b) *RoBaRaCoCh Layout*

Figure 2.3: *Address bit layouts for a main memory of 2 channels, 4 ranks of DDR4 8Gb x8*

sor communicates with the main memory through distinct memory channels ①, each having its own set of command, data and address buses. Multiple memory modules ② (usually Dual In-line Memory Modules, DIMM) can be connected to a single channel. A module includes multiple memory chips ③, grouped in ranks. The command and address buses are common to all chips, but the data bus is split among all the chips of a rank. A memory chip contains several banks ④, grouped in bank groups.

The bank is the smallest structure visible to the memory controller. It is the target

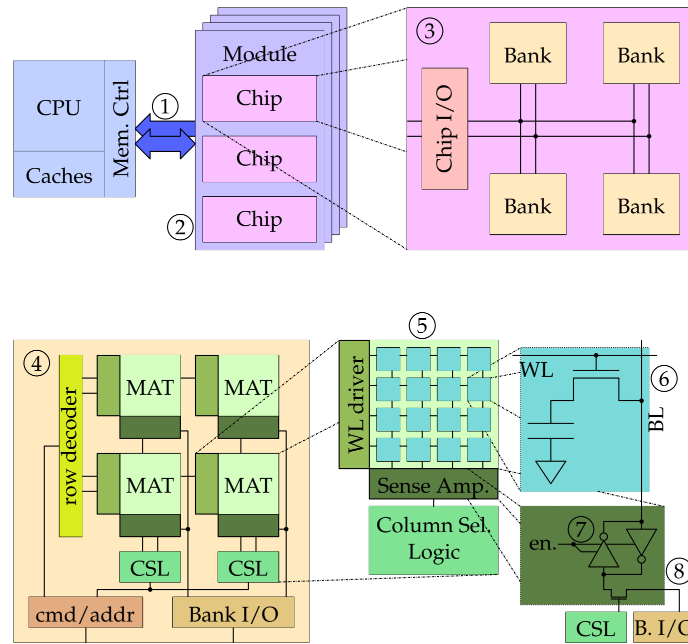


Figure 2.4: *Processor to DRAM bank architecture*

of all data-access requests. A bank is made of interface ports, drivers to select the row to activate (the row decoder) and the column to read or write on (the column selection logic, CSL), and a set of Memory Array Tiles (MAT) ⑤.

In a MAT, the bit cells ⑥ are organised in a matrix, where all cells of a row share a unique wordline (WL), and all cells of a column share a unique bitline (BL). The MAT also contains a WL driver and a set of Sense Amplifiers, which are used to read and write values in cells. A memory cell is the combination of one capacitor and one transistor. The transistor, called access transistor, is driven by the WL of the row. When activated, it connects one end of the capacitor to the BL of the column. The other end of the capacitor is permanently connected to the ground. The charge of this storage capacitor defines the value of the bit stored by the cell.

At one end of the BL is the Sense Amplifier (SA) ⑦, which is made of two inverters in a cross-coupled configuration. One side is connected to the BL, and when the bit is accessed, the CSL connects the other end of the SA to the bank data I/O ⑧. The Sense amplifiers are also used to store the last accessed row, therefore the set of Sense amplifiers is also called Row Buffer (RB).

To read or write data in the memory, the target row must be activated to load its content in the RB, before accessing the data in the RB. The PRECHARGED state, AC-

TIVATE, READ and WRITE commands are illustrated in Figure 2.5.

- When no row is loaded in the RB, the bank is in the idle state.
- To load a DRAM row in the RB, the memory controller issues an ACTIVATE command (ACT) to the bank.
- when a row is loaded in the RB, the memory controller can issue READ commands to request data from the RB,
- or WRITE commands to change data in the RB. As the row is still activated, any change on the RB is reflected in the DRAM row.
- When an other row is accessed, the connection between the currently-activated row and the RB must first be closed using the PRECHARGE command (PRE), resetting the bank to the PRECHARGE state.

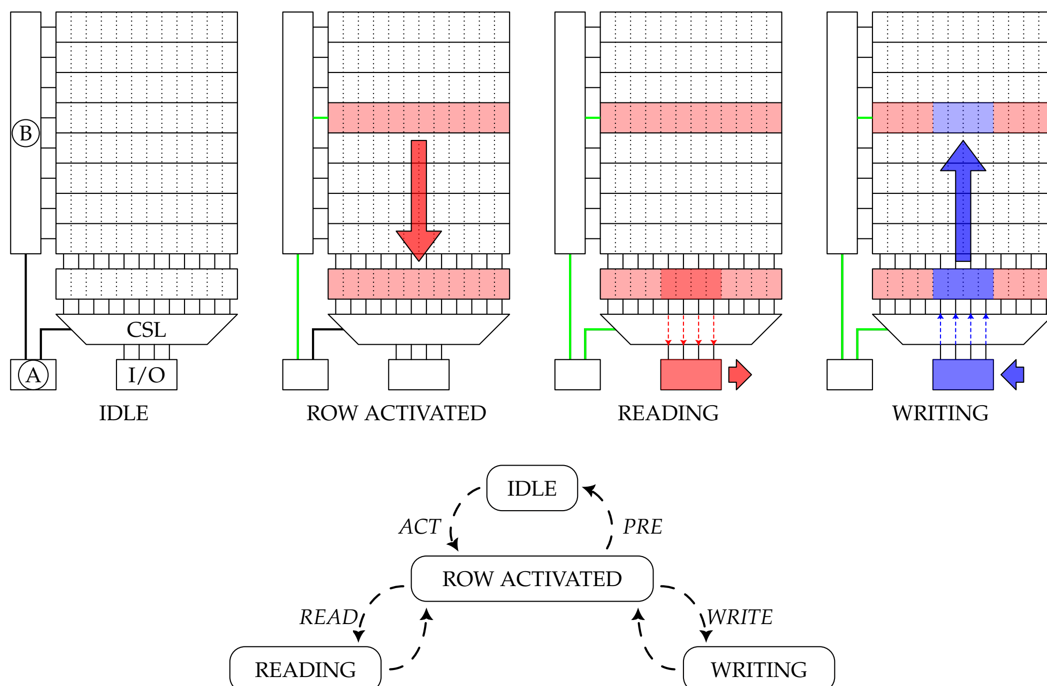


Figure 2.5: *Bank-level states and data-access-related commands. (A): cmd/addr decoder, (B): row decoder and WL driver. Changing from IDLE state to ROW ACTIVATED state and back is done using the commands ACT and PRE. commands READ and WRITE temporarily change the state to READING and WRITING, but automatically it changes back to ROW ACTIVATED once the operation is completed.*

Loading rows in the RB is done using the ACTIVATE-PRECHARGE cycle. On an electrical level, the ACTIVATE-PRECHARGE cycle is illustrated in Figure 2.6.

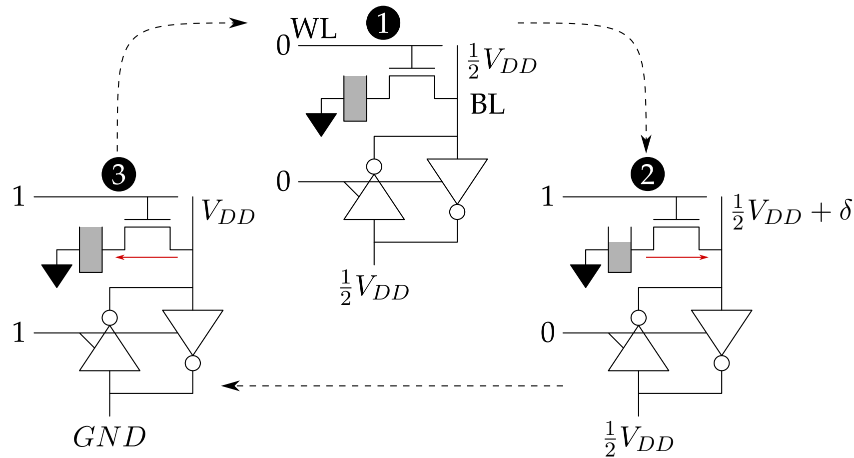


Figure 2.6: *ACTIVATE-PRECHARGE cycle. the grey rectangle inside the capacitor represents its charge.*

1. By default, every bit-cell is in the precharged state ❶: the capacitor is fully charged (resp. fully depleted), the WL is lowered, and both ends of the SA, including the BL, are maintained at a voltage of $\frac{1}{2}V_{DD}$.

2. In order to load the content of the bit in the row buffer, the memory controller issues an ACT to the bank. When the bank receives this command, the appropriate WL is raised, connecting the capacitor of the memory cells in the target row to their BL. Each capacitor and its BL share their charge, raising the voltage of the BL to $V_{DD} + \delta$ (resp. lowering it to $V_{DD} - \delta$) ❷. Then, the SA is enabled, detects the voltage difference between its ends, and amplifies it until the BL is at V_{DD} (resp. GND) and the other end of the SA is at GND (resp. V_{DD}) ❸. As the capacitor is still connected to the BL, its charge is restored (resp. depleted) by the voltage change in the BL.

3. When writing data in the RB, changing the value in the RB changes the voltage of the BL and therefore the charge of the capacitor, keeping the bit-cell updated with the proper value.

4. When the RB is needed to store an other row, the currently-activated row must first be closed. When the bank receives a PRE, the WL of the active row is lowered, isolating it from the BL. The voltage of both ends of the SA returns to $\frac{1}{2}V_{DD}$, and the system returns to the precharged state ❶, ready for the next ACT.

When accessing data, if the targeted row is already loaded in the RB (we call this event a **row hit**), the operation is performed directly on the RB. If an other row is

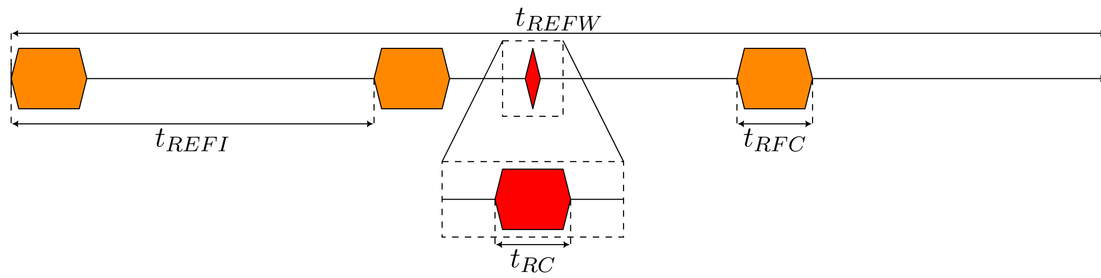
activated (we call this event a **row conflict**), the bank must precharge the previous row, and activate the target row before returning the data. If no row is currently active in the bank (we call this event a **row miss**), the bank only needs to activate the target row before returning the data.

Capacitors are not perfect charge holders. They leak charge over time. If a memory cell is not accessed (*i.e.*, loaded into the RB) frequently enough to refill the capacitor, the charge will reduce until it goes below the threshold at which it can raise the voltage of the BL when the row is activated. At ambient temperature, most memory cells will lose their data after a few seconds of inactivity. This effect varies from cell to cell, and is inversely proportional to the temperature, so it is lowered in cold environments [5]. To counter this issue and to avoid losing data while the system is running, the memory controller periodically issues REFRESH commands (REF) to the memory. To process this command, the memory banks activate a few rows to refill their storage capacitors.

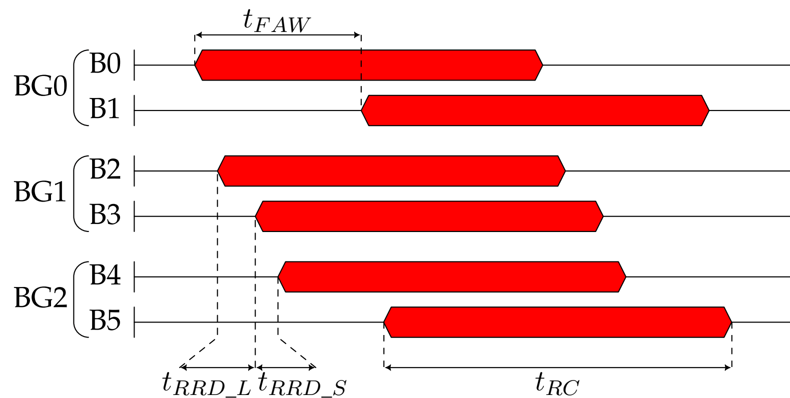
The relevant DRAM timing parameters are illustrated in Figure 2.7 and listed in Table 2.1 for three models of DDR3, DDR4 and DDR5. The minimum interval between two ACTs is defined by t_{RC}, t_{RRD_L} or t_{RRD_S} if the rows to activate are respectively located in the same bank, in different banks of the same bank group, or in different bank groups. Additionally, a maximum of four ACTs can be issued to a rank every t_{FAW}. REF commands are issued by the memory controller every t_{REFI} and last t_{RFC}. In a cycle of t_{REFW}, all the rows of the memory are refreshed at least once.

Table 2.1: *Relevant timing parameters for a DDR3 1600 8Gb x8 [6], DDR4 2400 8Gb x8 [7], and DDR5 4000 8Gb x8 [8].*

Name	Description	DDR3	DDR4	DDR5
t_{RC}	Same-bank minimum ACT interval	48.75ns	45.8ns	46ns
t_{RRD_S}	Different-Bank-Group minimum ACT interval	6.25ns	3.3ns	4ns
t_{RRD_L}	Same-Bank-Group minimum ACT interval	-	4.9ns	5ns
t_{FAW}	Four activate window	30ns	21.67ns	16ns
t_{REFW}	REF window (ms)	64ms	64ms	32ms
t_{REFI}	REF interval (µs)	7.8µs	7.8µs	3.9µs
t_{RFC}	REF command duration	350ns	350ns	195ns



(a) Bank-level timings



(b) rank-level timings

Figure 2.7: *Illustration of DDR timings. Horizontal axis is time. Coloured sections represent periods when the bank is busy. An orange section represents a bank busy from a REF command. A red section represents the period after an ACT command when the bank cannot be issued an other ACT command. for the rank-level timings (b), 3 bank groups are represented, with 2 banks for each bank group.*

2.2 Rowhammer attack

2.2.1 Cell-to-cell disturbance

As manufacturing processes become more efficient over the years, manufacturers are able to increase the memory density of DRAM chips, resulting in lower production cost, lower energy consumption for the same storage capacity, and higher storage capacity for the same silicon area. However, increasing the density of DRAM banks also results in rows being closer to each other, lower noise margin for bit cells, leading to increased parasitic electrical interactions between nearby bit cells.

Kim et al. [1] discovered that when a row is activated, the capacitors of adjacent rows experience a small charge leakage, more important than their normal leakage. Multiple causes were considered by the author: electromagnetic coupling had already

been proved to cause undesirable interactions between nearby wordlines [9]; bridges could be formed between adjacent wires and/or capacitances to accelerate the charge loss when toggling the WL [10, 11]; toggling a WL repeatedly for long periods of time could permanently damage the row ant its neighbours by hot carrier injection [12], altering the property of access transistors or injecting charges in nearby capacitors. At that time, the author could not determine precisely the main cause of this issue. More recently, Yang et al. [13] demonstrated that the cause of this issue is that when a WL is switched up and down, charge traps located below the WL capture some negative charges, which are emitted in the substrate, then migrate to the nearby capacitors, causing the charge leakage.

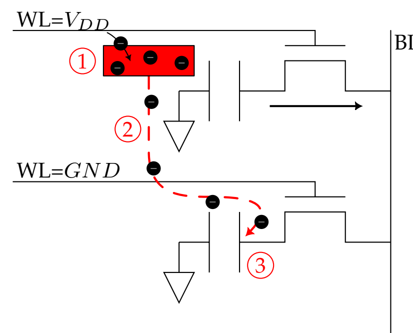


Figure 2.8: *Charge trapping and migration to victim memory cells. When a WL is toggled, charge traps located below the WL ① capture some negative charges ⊖, which are emitted into the substrate ② and migrate to nearby capacitor ③, reducing the stored charge.*

Alone, this issue is not dangerous, as the charge leakage is not important. The next REF or ACT on this row will refill the capacitor. However, if this operation is repeated enough times so that the charge of a capacitor goes below the threshold at which it can raise the BL voltage during an ACT, the bit will be misinterpreted on the next ACT, the charge depleted, resulting in the corruption of the data. The evolution of the charge of the capacitor under normal operation and under disturbance from repeated ACTs on adjacent rows is illustrated Figure 2.9

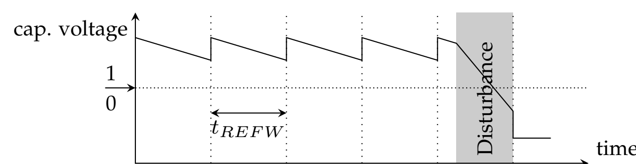


Figure 2.9: *Memory cell capacitor voltage evolution under normal behaviour and under disturbance. Stored value starts at 1, and changes to 0 after corruption.*

We define the **disturbance level** of a row as the number of ACTS that have been

issued to its neighbours since the last ACT or REF on it. Bit-flips only appear after the disturbance level reaches a certain value. We call this value the Rowhammer threshold (T_{RH}). It is primarily defined by the technology: because of smaller capacitors and closer wordlines, denser memories will have a lower T_{RH} [1, 14]. The value of T_{RH} has been greatly reduced over the years, getting from 138k ACTs to produce a bit-flip on older DDR3 to 9.6k for recent LPDDR4 [15].

Additionally, multiple factors can influence the effect of cell-to-cell disturbance on bit-flips. While the capacitors can only discharge under the influence of cell-to-cell disturbance, the bit-flip can happen in both directions ($1 \rightarrow 0$ or $0 \rightarrow 1$) depending on how the voltage is interpreted by the logic circuits. However, this direction being determined by the logic circuits, it cannot vary in time for one bit. According to the experiments conducted by Kim et al. in 2020 [15], the data pattern in the memory (*i.e.*, the positions of 1s and 0s in the MAT, *e.g.* row stripes, column stripes, checkerboard) seems to have a very important influence on the presence of bit-flips after an attack. This study shows that the pattern that produces the most bit-flips varies largely across DRAM generations, but does not vary as much across the tested manufacturers.

2.2.2 Basic exploit

Cell-to-cell disturbance can be triggered intentionally and exploited to perform what is called a **RowHammer** (RH) attack. The goal of the aggressor is to ACT the neighbours of a victim rows enough times between two ACTs on the victim in order to flip some bits. Two obstacles arise to perform this attack:

- the row buffer: it contains the last activated row, accessing the row stored in it will trigger a row hit. Row hits must be avoided, as they don't lead to ACTs on the row, and therefore don't disturb the neighbours. The aggressor must use at least two aggressor rows of the same bank to avoid row hits;
- the cache memories: they contain the most frequently-accessed data. Repeatedly accessing the same rows will generate cache hits, and no request will reach the main memory. The aggressor must either bypass the cache [16], or use cache

eviction mechanisms to remove the aggressor rows from the cache memories, using for example cache flush instructions when they are available [1, 17], or cache eviction techniques [4, 18, 19].

Memory corruption using the RowHammer attack can be performed using the simple x86 assembly loop presented in Listing 2.1 [1]. The positions of the aggressor rows and the victim cells in the DRAM bank attacked by this program is illustrated in Figure 2.10.

```
loop:
    mov (X), %eax ; read value from address X
    mov (Y), %eax ; read value from address Y
    clflush (X)   ; evict address X from cache
    clflush (Y)   ; evict address Y from cache
    mfence        ; wait for previous instructions to complete
    jmp loop      ; restart the loop
```

Listing 2.1: *Rowhammer loop on x86 system*

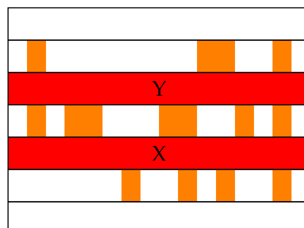


Figure 2.10: *Memory bank under attack on rows X and Y. Aggressor rows are colored in red, and victim cells in orange.*

However, the cache cannot be flushed manually by user-level programs on all systems. ARMv7-A processors restricted the cache-managing instructions to privileged programs, and ARMv8-A processors can be configured to restrict their usage by unprivileged programs. Cache-managing instructions are also not available to web pages, as the available instructions are limited by the web browser. In these cases, the aggressor can rely on cache-eviction strategies that exploit the replacement policy of the cache memory to replace the used address with an other data in the caches [4, 18, 20, 3, 21], on the DMA to bypass the cache [16], or on the integrated GPU which has a very simple cache from which it is easy to evict data [19].

Recently, it has been demonstrated that the Rowhammer attack could be escalated beyond immediate neighbours of aggressor rows [15, 22]. The increasing density made

the disturbance reach beyond the immediate neighbours on recent LPDDR4 [15]. But more importantly, a study demonstrated that it was possible to *propagate* the disturbance from the immediate neighbour of an aggressor row to its next neighbour, two rows from the main aggressor [22]. This attack combines many accesses to the main aggressor rows with a few accesses to the secondary aggressors, adjacent of the firsts. It manages to induce bit-flips in the victim rows, immediate neighbours of the secondary aggressor rows, at a distance of two rows from the main aggressors.

2.2.3 Exploits in literature

Rowhammer attacks are used to corrupt some bits in the memory without accessing them. However, the goal of aggressors is rarely to simply corrupt the memory. Aggressors can use this attack to perform privilege escalation, even from web browser sandboxes [4, 21, 23]; to retrieve sensitive information, especially encryption keys [24, 25, 2]; or to remotely crash a system [3], without malicious code running on it, through network requests alone. The main goal of an aggressor is often to perform privilege escalation: aggressors try to produce bit-flips in Page Table Entries (PTEs), which defines the memory sections a program has access to. A bit-flip in this table can grant the prgram access to restricted sections [23]. Flipping the bits of a specific value in the memory can be summarised in three steps:

1. **Locate a potential bit-flip location**. All bits of the memory are not equally vulnerable to cell-to-cell disturbance, the victim location must be prone to experience bit-flips and have accessible neighbour rows. To search for potential bit-flip locations, the aggressor can allocate a large chunk of memory, perform RowHammer attacks on it and check of bit-flips in the allocated memory to find a suitable location to attack.

2. **Place the target value at this location**. The aggressor forces the system to allocate this location for the target value. To do that, the memory is allocated to the maximum allowed space, and the target location is freed in order to force the Operating System (OS) to reallocate it for the target value.

3. **Flip the bits of the target value**. This last step simply consists in hammering the neighbours of the victim row until the bit-flip happens.

Searching for potential bit-flip locations is often the longest task for an aggressor. A prerequisite is the knowledge of the DRAM addressing function: to select the neighbour rows of potential victims, the attack program has to know how the virtual addresses are translated to physical address by the operating system, then into rank, bank and row numbers on the DRAM bus by the memory controller. The addressing function is not an information that the attacker can acquire directly. Even if this is a public information for some manufacturers [26] it still varies from processor to processor. Attacker programs might not know, which processor it is running on. To get the addressing function, attackers have to retro-engineer it. The reverse-engineering of the addressing function has been studied over the years [27, 28, 29, 30], and multiple methods were proposed to do it. Some use hardware probes on the memory bus [27] when a physical access to it is possible, or exploit access timing differences between row hits and row conflicts [27, 28, 29, 30]. Once the addressing function is known by the aggressor, it still must understand the physical layout of the memory, especially the rows adjacency. Indeed, consecutive row numbers do not always translate to adjacent rows. DRAM manufacturers can choose to change the physical location of logical rows, *e.g.* to use a backup row if a row does not work as intended after the fabrication of the memory [31]. The only way for an aggressor to determine which rows are adjacent to each other is to hammer some rows and search for bit-flips in other rows. Once the program has determined what rows can be attacked from accessible rows, it can move on to place the target victim value, and proceed to hammering it to produce a bit-flip.

In addition to the main memory of the computer, Zhang et al. [32] demonstrated that this attack can be used on Solid-State Drive (SSD) devices. These devices use a DRAM memory to map logical addresses to physical block. Bit-flips can be triggered in this memory to redirect the mapping of a victim logical block to a different physical block.

Over the years, the countermeasures against Rowhammer attacks implemented by manufacturers in consumer products have been proved to be inefficient by attacks that are aware of their presence [33, 34], once their mechanism is retro-engineered, even

when said mechanism is not documented [34].

2.3 Rowhammer countermeasures

2.3.1 Basic Principles

As the threat of Rowhammer attacks has worsen over the years, the community has conducted extensive research on Rowhammer mitigation.

We consider an attack successful when the disturbance level of a victim row reaches T_{RH}, as bit-flips can only happen when this level is above or equal to T_{RH}. Hence, mitigating RowHammer attacks consists in preventing the disturbance level from reaching T_{RH}, *i.e.*, preventing the aggressor from performing T_{RH} ACTs on the neighbours of a victim row between two ACTs or REFs on it. This can be done by issuing additional ACTs to the victim or preventing ACTs on neighbours of the victim when its disturbance level is getting too high.

Rowhammer mitigation mechanisms can be implemented in software, modifying the behaviour of the system or communicating with existing components to prevent the attack; or in hardware, with the addition of specifically-designed components to detect the attack or to prevent the corruption. Mitigation mechanisms can also rely on detection mechanisms, to either detect the aggressor process or the aggressor rows to prevent the attack from completing.

The implementation of a Rowhammer Mitigation mechanism can have a negative impact on the system, on multiple points:

- CPU performance, if the mitigation is implemented in software and requires regular observation;
- Memory performance, when issuing additional commands to the memory to prevent the corruption;
- Silicon Area, if the mitigation requires additional hardware components;

- Energy consumption.

2.3.2 Software-based protection

Some mitigation techniques are implemented primarily in software. Among them, some use hardware performance counters [18, 35, 36, 37] to detect patterns that result from an attack. They analyse the events and either stop the attack processes, or refresh the victim rows. Some use other mechanisms, such as ZebRAM [38] that isolates sensitive data in the memory with empty DRAM rows. These rows serve as barriers against Rowhammer attacks, preventing aggressors from hammering the neighbours of potential victim rows.

Software-based protections are easy to implement, as they don't require hardware modifications. However, they come with either a processing performance cost for mitigations that periodically analyse events, or a memory cost for ZebRAM [38] which uses DRAM rows as barriers.

To avoid such performance issues, most mitigation proposals are implemented in hardware.

2.3.3 Hardware probabilistic protection

Some solutions implemented in the hardware are mostly probability-based. PARA [1] randomly refreshes the neighbours of activated rows. Frequently-activated rows have more chance to trigger the random refresh of their neighbours, and an attack that activates a few rows tens of thousands of times has a very high probability to trigger the refresh on the potential victims. Discreet-PARA [39] aims at improving the previous solution by adding counters, where each counter counts the ACTs in a section of the bank, to only trigger the original PARA when an ACT is issued in a section that is frequently activated, in order to reduce the performance impact of PARA. ProHit [40] uses a priority table in which neighbours of activated rows are randomly inserted and promoted to a higher priority. During the periodic refreshes (every t_{REFI}), the highest priority

row is refreshed to prevent the corruption on the most likely victim. MRLoc [41] stores the neighbours of the last activated rows in a queue, and uses the frequency at which it is inserted in the queue to determine a refresh probability, then use this probability to randomly activate an additional refresh.

Some proposals use the probabilistic structure of a Counting Bloom Filter (CBF) [42] to evaluate the activation count of every row in the memory to detect rows that are activated too many times [43, 44].

These probabilistic solutions offer a low performance cost and a low silicon area overhead. The rows activations only have a low probability of triggering additional refreshes, and improvements on the original PARA proposal aim at reducing the chance of triggering a refresh on rows that are not used as aggressors. They don't require a lot of memory to store the queue or tables, hence the silicon area overhead of these solutions is relatively low. However, due to their probabilistic nature, these solutions cannot offer a guaranteed protection.

2.3.4 Hardware counter-based protection

Counter-based hardware protection against RowHammer attacks rely on counters to monitor the number of ACTs issued to each row during a refresh window. When the count of one row reaches a threshold, the row is considered as an aggressor row and the mechanism acts accordingly, either refreshing the neighbours of the aggressor which are the potential victims of the attack, or using a mean to prevent further ACT on the aggressor row. Contrary to probabilistic solutions which offer no guarantee of protection, counter-based mitigation techniques are designed to detect all attacks, at the additional expense of having to store and manage more counters.

A naive solution for counter-based Rowhammer protection would be to have one counter per row, increment them when the associated row is activated, and react when the count reaches a specified threshold. However, a typical DDR4 bank has $2^{16} = 65536$ rows per bank, and $2^4 = 16$ banks per rank. Hence, this naive solution would require $2^{20} \approx 10^6$ counters per rank, with 12 to 16 bits per counter depending on the corruption

threshold, for a total of 12Mib to 16Mib bits of additional memory per rank, which is not reasonable [1].

Therefore, every counter-based solution uses a different counter structure to reduce the number of counters to a minimum, while maintaining the guarantee of protection and minimal false positive detection.

Multiple mitigation proposals use a counter tree structure, which divides row groups into progressively smaller groups as they get activated in order to detect the rows that get activated the most [45, 46, 47].

BlockHammer [48] uses two CBFs to evaluate the activation count of every row, and compensates the probabilistic nature of the CBF by never decreasing the values (except for periodic reset every refresh cycle t_{REFW}) and preventing further accesses to detected aggressors instead of refreshing the neighbours.

Some proposals use variations of the Misra-Gries [49] or Counter-based summary [50] algorithms to determine the most activated rows within a time period [51, 52, 53]. A table associates a row to a counter and increments the counters when the row is activated. When a new row is activated, it eventually replaces the row of the entry with the minimum counter value by the new row, without reseting the counter. At any point in time, the actual activation count of a row is lower or equal to the counter value, and higher or equal to the minimum counter value in the table. When a specific threshold is reached, the mechanism either issues an additional ACT to the victim rows [51], uses the DDR5 Refresh Management (RFM) command to refresh the victims [53], or swaps the row with another to prevent the corruption [52].

PanOpticon [54] implements counters in the DRAM chips, associates every row to a unique counter stored in an additional MAT inside the DRAM bank. A counter is incremented when its associated row is activated. When it reaches a specific threshold, the mechanism refreshes the potential victims, potentially delaying other accesses by faking a missed access using the ALERTn signal to leave enough time for the refresh to take place.

TWiCe [55] uses the lossy-counting algorithm to count the activations on every

row but periodically removes the table entries for rows that have not been activated frequenlty-enough.

2.3.5 Conclusion

Numerous mitigation techniques have been proposed, each with their pros and cons. However, many require modifications of the protocol to allow the countermeasures to precisely refresh the neighbours of the aggressor rows without knowing the internal layout of the memory.

While the community has proposed numerous algorithms to detect attacks with great accuracy and low overhead, some proposals have been made to facilitate the integration of these mitigations with the DRAM. Multiple propositions were made to add RowHammer-mitigation-specific commands to the DDR standard to allow the memory controller to request refreshes on the victim rows [56, 57], and to allow the memory controller to get information on the DRAM [58]. In the mean time, DRAM manufacturers have also implemented their own mitigations in the DRAM modules. However, while the details of these mitigations were not made public, the community proved that they can be partially retro-engineered by a malicious program and bypassed to produce bit-flips [59, 34]. Consequently, researchers are asking DRAM vendors to provide precise information about the integrated defences and limitations [57].

From an other perspective, while most mitigation proposals aim at correcting the symptoms of the Rowhammer attack (*i.e.*, preventing the disturbance level from reaching the point when bit-flips happen), some discuss the possibility to correct the problem at its root by changing the manufacturing process to reduce, if not prevent, cell-to-cell disturbance that cause the memory corruption on DRAM [60, 61, 13].

2.4 Conclusion

Since the discovery of the cell-to-cell disturbance error in 2014 [1], the Rowhammer attack which exploits this issue has been an important subject of research in the ar-

chitecture design and security communities. Numerous mitigations were proposed in the last decade and continue to be published to this day, but as the density of DRAM increases, the technology becomes more and more vulnerable to the attack, rendering some of the proposed countermeasures less effective, drastically increasing their silicon area or performance overhead, and sometimes even making them ineffective against modern attacks.

The study of Rowhammer attacks and countermeasures is to this day an important research subject, as modern systems cannot be considered protected against this threat.

III

Rowhammer Simulation

Contents

3.1 Motivation **26**

3.2 Rowhammer simulation requirements **28**

3.3 gem5 and Ramulator **31**

3.4 Memory Corruption simulation **33**

3.4.1 Integration of the memory-corruption module in gem5 and Ramulator 33

3.4.2 Disturbance and corruption simulation 34

3.5 Mitigation integration in gem5 . . . . **37**

3.6 Usage, limitations and evaluation **38**

3.6.1 Limitations 40

3.6.2 Evaluation 41

3.7 Conclusion **42**

3.1 Motivation

When designing a Rowhammer mitigation technique, it is important to properly evaluate it on multiple points. This allows the designers to compare it against existing proposals, improve or correct it if necessary, and configure some parameters to optimise it. A mitigation technique proposal can be evaluated on the following points:

- Its capability to protect a system against RowHammer attacks;
- Its impact on the performance of the system it is implemented in;
- Its energy consumption and silicon area overhead.

The silicon area and energy models of a mitigation proposal can be measured with electrical or register-transfer level(RTL) simulations.

To evaluate the capability to protect the system and the performance cost, mitigation proposals must use evaluation platforms which generate metrics such as False Positive (FP) and False Negative (FN) detection rates, and some mitigation-specific metrics like the number of additional refreshes, the number of row swaps or the number of detected aggressors.

Even if software-based mitigation proposals do not need any modification on the hardware to protect the system, generating these metrics still requires the development of an evaluation platform that embeds the necessary tools to monitor them. Some information about the memory are not known by the processor, and if an attack succeeds to corrupt one bit in the memory, there is no guarantee that the bit-flip will be acknowledged when generating the metrics. Hardware-based mitigation proposals on the other hand will always require to be implemented in the architecture to be evaluated.

To properly evaluate the mitigation proposals, an evaluation platform needs to be able to execute modern attacks that require a working system (*e.g.*, attacks that escape web browser sandboxes) while running the mitigation technique, provide the mitigation with all the information it needs, in an environment close to consumer systems,

and configurable to tweak memory vulnerability and mitigation parameters.

Designing and fabricating new integrated circuits for each iteration during the development of RowHammer mitigation techniques would have a huge financial and time cost. Researchers can rely on FPGA [1] to emulate a memory controller, implementing their proposals to test them against existing attacks on modern systems and DRAM memories, while varying the temperature of the memory to increase its vulnerability to bit-flips. This has the advantage of being very faithful to real systems, while offering a high modularity and low development cycle to create mitigation techniques. However, both software implementation on real systems and hardware implementations on FPGA face the same limitations. First, the internal layout of the DRAM banks is not known to the memory controller. Mitigation proposals which use victim row refreshes cannot protect the memory as they cannot know what are the neighbour rows to refresh. Second, having to test on existing memory modules is an issue when considering future memory technologies like DDR5 or emerging non-volatile memories which are potentially susceptible to variations of the Rowhammer attack [62, 63]. As future technologies are by definition not available as memory replacements for modern computers, testing on these technologies is hard, if not impossible.

The last solution for this is to use simulators, where researchers can simulate modern or future architectures, with recent memories or future technologies. However, existing simulators do not provide any tool to simulate the corruption of the memory from Rowhammer attacks. Most Rowhammer attack need to witness the corruption in the memory in order to work properly. A simulator that is not capable of simulating the memory corruption will not be able to simulate such attacks.

State-of-the-art mitigation proposals use various simulation tools to evaluate the performance. For example,

- Graphene [51], TWiCe [55], ProHIT [40] use the x86 microarchitecture simulator McSimA+ [64];
- BlockHammer [48] and PARA [15] use the DRAM timing simulator Ramulator [65];

- and MRLoc [41] uses the modular architecture simulator gem5 [66].

However, the authors had to modify the tools to generate the necessary metrics. Furthermore, these tools were used to simulate the performance cost of the integration of mitigations in the system. The simulator would not have simulated the corruption even if there were nothing to prevent it. Finally, the first two simulators work by re-executing a trace of memory accesses on x86-only systems. Evaluating the performance cost on an ARM-based system would require to move to a ARM-compatible simulator.

In this chapter, we propose to integrate the memory corruption inside a simulator, tools to include various mitigations and generate various metrics, in order to facilitate the development and evaluation of Rowhammer mitigation techniques.

3.2 Rowhammer simulation requirements

A simulator for Rowhammer mitigation development needs to have the following features:

- It needs to be able to simulate modern complex systems with a running operating systems and programs. The latest Rowhammer attacks are designed to be executed on web browser [21, 4]. The simulator must be able to run an operating system with a running web browser to execute such attacks, with a simulated memory large enough to fit this system.
- It must be modular enough to implement memory corruption without disturbing the simulated memory bandwidth: the corruption should not use the intended ways to perform memory accesses from the simulated process as to not disturb the simulated timings of the system. Corruption should therefore not be seen by the simulated memory as a standard memory request, even if the processor is not aware of it, but should directly hit the stored value.
- The simulation must be accurate regarding the timings of the memory. The memory timings is critical when running an attack. Cache hits, cache misses, DRAM

row hits, row conflicts and row misses must have realistic timings so that the attacks will behave exactly as they would on a real system. Furthermore, some attacks rely on timing differences between row hits and row conflicts to retro-engineer the layout of the memory [27, 28]. Some countermeasures also use the timings of memory accesses to detect attacks, *e.g.* through the classification of hardware event traces by a machine learning model [44, 36, 67].

- The corruption needs to happen as the system is running. Some attacks need to witness the corruption happening in order to work properly [23].
- It must be able to execute programs compiled for at least the two most popular Instruction Set Architectures (ISA) for consumer computers, *i.e.* x86 and ARM.

We do not need to simulate electrical details of the system architecture. Electrical simulators such as SPICE are very accurate, but are too heavy and slow for a system simulation. Furthermore, the system will rapidly become very complex, and the memory very large as the architecture will run an OS executing web browsers. Integrating the system and the mitigations in an electrical simulation would be very time consuming. Additionally, the electrical cause of the disturbance [13] would be very complex to configure depending on the technology, and add a significant overhead in the simulation. Therefore, detailed electrical simulation is not suitable to simulate microarchitectural attacks such as Rowhammer. Electrical simulators, however, are a good fit to evaluate the energy overhead and processing time of single operations on hardware-based mitigations.

The appropriate abstraction level for this kind of simulation is the system-level architecture simulation. Architecture simulators reproduce the functional behaviour of a individual components of a computing device (*i.e.* the processor, the cache memories and the main memory) to generate metrics while a program is running on the simulated system. They can be divided into two categories: functional simulators and timing simulators. Functional simulators, sometimes called Instruction Set Simulators, are meant to reproduce the architecture from a program point of view. They reproduce the functionality of the device without considering the timings of internal components. These can be used as a fast way to verify the functionality or evaluate the performance

of some algorithm or applications on some specific architecture. On the contrary, timing simulators implement the behaviour of internal components of the architecture more precisely. They consider the communication between all the components and the time needed for their operation. To simulate the corruption from Rowhammer attacks, the timing of the components, in particular that of the memory, is important. Hence, the simulator will be a timing simulator.

Timing Architecture simulators can run two types of simulations: execution-driven simulation, and trace-driven simulation. A Trace-driven simulation works by reading a trace of instructions captured by a previous execution on a simulator of any type or a real device. This has the advantage of being a fast solution to compare multiple architectures for the same program execution. The simulated system can be much simpler, sometimes only simulating the memory components. However, programs that need to interact with the user or with the system to collect data to take decisions will not behave normally. Alternatively, an execution-driven simulation works by making the simulated system execute the program. This type of simulation is slower than trace-driven simulation, but it is able to simulate the programs that could not be simulated on trace-driven simulation for the reason mentioned above. In order to work properly, most RowHammer attacks need to witness the corruption, for example by searching for bit-flip locations to place the desired value in before performing the hammering loop. Therefore, the simulator needs to be able to run execution-driven simulation.

From all the simulators presented by Akram et al. in 2019 [68], only six timing simulators can simulate both x86 and ARM ISA. Among those simulators, the only simu-

Simulator	Timing simulation	Execution driven	Modular
gem5 [66]	Yes	Yes	Yes
McPAT [69]	Yes	Yes	No
Multi2Sim [70]	Yes	No	Yes
SIMICS [71]	Yes	Yes	No
SimpleScalar [72]	Yes	Yes	No
TEM^2P^2EST [73]	Yes	Yes	No

Table 3.1: *Architecture Simulators comparison.*

lator that is modular and capable of doing timing and execution-driven simulation is gem5.

3.3 gem5 and Ramulator

gem5 is an open-source modular computer architecture simulator, widely used in academia and industry. Users can create custom computer architectures by configuring and connecting CPUs, cache memories, memory buses and a main memory. The source code can be modified to create new architecture components for specific purposes, or to improve existing ones to add functionalities. For the particular case of the main memory simulation, gem5 integrates multiple memory simulators. Users can select the simulator that fits their needs. Among the integrated simulators, DRAMSIM2 [74] and and its recently added successor DRAMSIM3 [75] are the most precise simulators regarding the timings of the DRAM. However, gem5 can be configured to use other memory simulators if they are compatible. In the case of the memory corruption simulation, we chose to use Ramulator [65] as our main memory simulator. As DRAMSim3 was not integrated in gem5 at the time we worked on the corruption simulation, Ramulator was a satisfying solution for its cycle-accurate characteristics, as well as its sources that could be easily modifiable.Anyway, the modifications made on Ramulator to simulate the memory corruption can easily be implemented in an other DRAM simulator.

Ramulator is a fast and extensible cycle-accurate DRAM simulator. It is used to precisely simulate the timings of DRAM memories, taking periodic REF, row hits, row misses and row conflicts into account. The simulation of the DRAM is only used to simulate the timings of the accesses to have a realistic delay. When simulating a memory access, the data is accessed (*i.e.*, read or written) by gem5.

When focusing on the memory, the behaviour of gem5 and Ramulator is illustrated in Figure 3.1.

1. For the processor to communicate with the memory module, gem5 simulates the memory bus.
2. Upon reception of a packet ①, the memory module of gem5 extracts the address and the type of access (read or write). It then generates a request to Ramulator ② for it to simulate the timings.

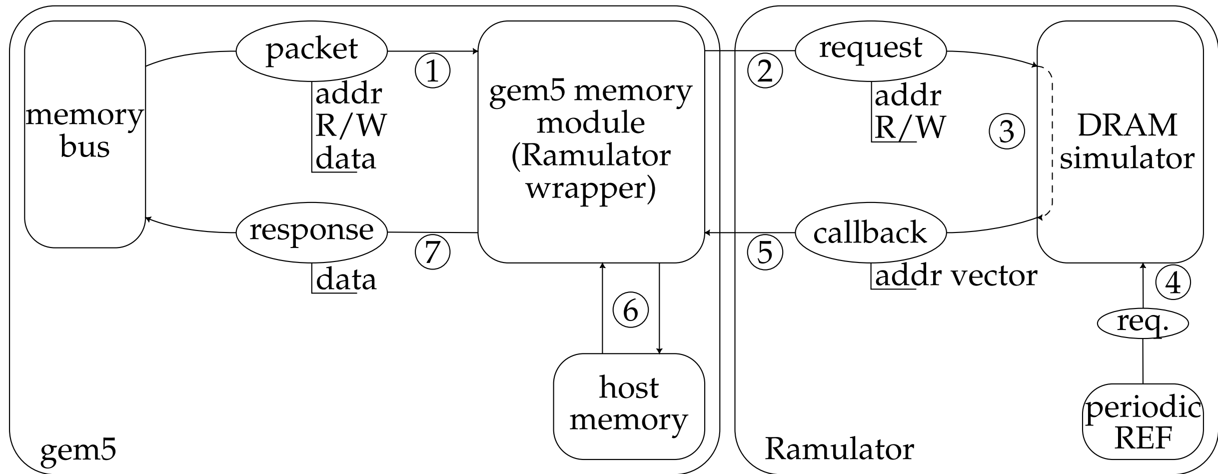


Figure 3.1: *gem5 and Ramulator memory architecture*

3. The DRAM simulator of Ramulator receives the request, and generates the address vector by extracting the target channel, rank, bank group, bank, row and column from the address. The address vector is stored in the request fields.

4. Using this information, it checks for row hits, row conflicts or row misses, and considers all the timing parameters of the DRAM ③. Ramulator autonomously takes the periodic refreshes into account, sometimes delaying incoming requests ④. The simulation of REFs only serves the purpose of making the memory busy for a short period. Ramulator does not communicate the Refresh events to gem5.

5. Once the timing of the request is respected, Ramulator uses the callback of the request to notify gem5 that the access is finished ⑤.

6. When the memory module is notified of a completed memory request, it performs the actual access to the stored value (either writing or reading it) ⑥.

7. When the memory access is finally complete, the memory modules sends the response of the original packet to the memory bus with the data ⑦.

The memory module of gem5 is the main component that handles memory accesses. It receives memory access packets from other components of the architecture, orders Ramulator to simulate the timings, and once the timing has been simulated, performs the actual access to the storage before replying to the packet.

3.4 Memory Corruption simulation

3.4.1 Integration of the memory-corruption module in gem5 and Ramulator

To simulate the memory corruption in gem5, we can either modify existing modules to integrate the corruption, or create a new component and connect it to the architecture. To follow the modularity of gem5 and for compatibility with other forks and future versions of gem5, we chose to create a separate memory-corruption module that will handle the memory corruption, and connect it to existing modules by performing only a few changes in the existing modules. This module is responsible for measuring the disturbance of every row in the memory from ACTs on neighbouring rows, and modifying the stored data to simulate the corruption.

The memory-corruption module will be connected directly to gem5's memory module. It will communicate with it to get notified when memory accesses are completed, and will ask it the location of the stored data to perform the actual modifications on the memory to simulate the corruption.

The memory-corruption module needs to know when rows are activated. However, whether the request led to a row hit or not is not part of the information provided in the request callback from Ramulator. Additionally, REF commands, which periodically reset the disturbance of the rows, are not notified to gem5. Therefore, the behaviour of Ramulator must be modified to notify gem5 when REF and ACT commands are issued. ACT commands are always a result of a memory access; this information can be added to the request callback. REF commands, on the contrary, are handled internally by Ramulator. They use the same request format as memory accesses, but the callback function of the request is not set. To notify REF commands to gem5, a `set_refresh_callback(cb)` method is implemented in Ramulator that, when used, sets the callback function for all future REF commands to the specified function. We make sure to call this function once at the beginning of the simulation in the wrapper of Ramulator in gem5. However, as simulating the timing of REF commands does not imply to specify which rows are being refreshed, Ramulator does not

specify which rows are refreshed for each REF. Hence, even with the callback function implemented, there is no viable way to know which rows are being refreshed each time a REF command is issued.

The modified behaviour of gem5 and Ramulator is illustrated in Figure 3.2.

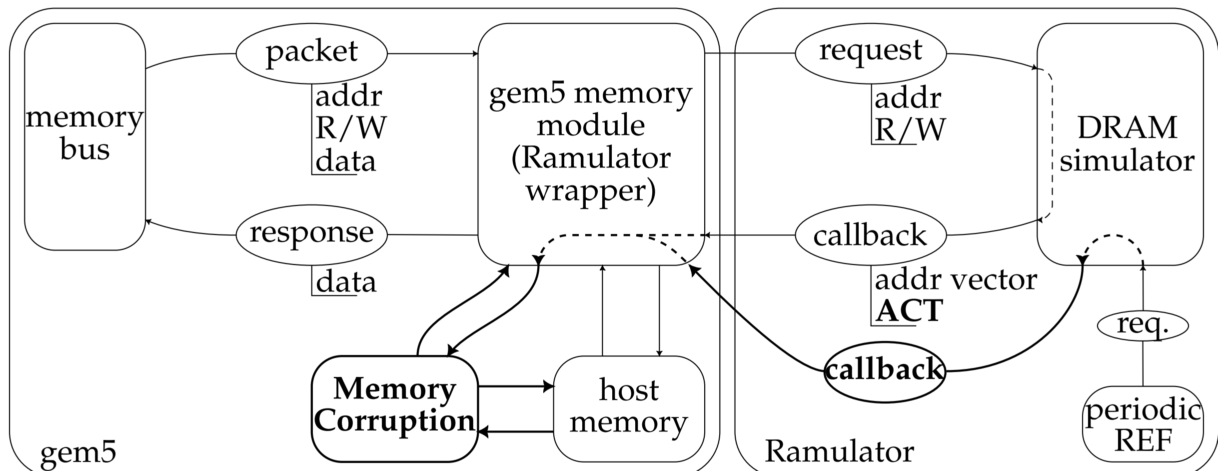


Figure 3.2: *gem5 and Ramulator memory architecture, with the Memory-Corruption module*

3.4.2 Disturbance and corruption simulation

The algorithm used by the memory-corruption module to handle row activations and refreshes is illustrated in Figure 3.3. In this figure, C is a table associating row positions to a counter of how many times they were disturbed by ACTs on adjacent rows. When the module is notified of an ACT being issued, it transforms the address into the position of the row. All addresses that target the same channel, bank group, bank and row but a different column have the same position, and adjacent rows have consecutive positions. If the table C has an entry for this position, it removes it, simulating the reset of the disturbance on this row. The memory-corruption module then simulates the disturbance on the neighbours of these rows, if they exist. As the REF simulation by Ramulator does not specify which rows are refreshed, the memory corruption module considers that all the rows are refreshed at once. REFs are issued once every t_{REFI}. Every t_{REFW}, all rows must have been refreshed. Therefore, we can consider that all rows are refreshed every $\frac{t_{REFW}}{t_{REFI}}$ REFs. According to Table 2.1 page 12, this ratio is equal to 8192 for all 3 considered DRAM generations.

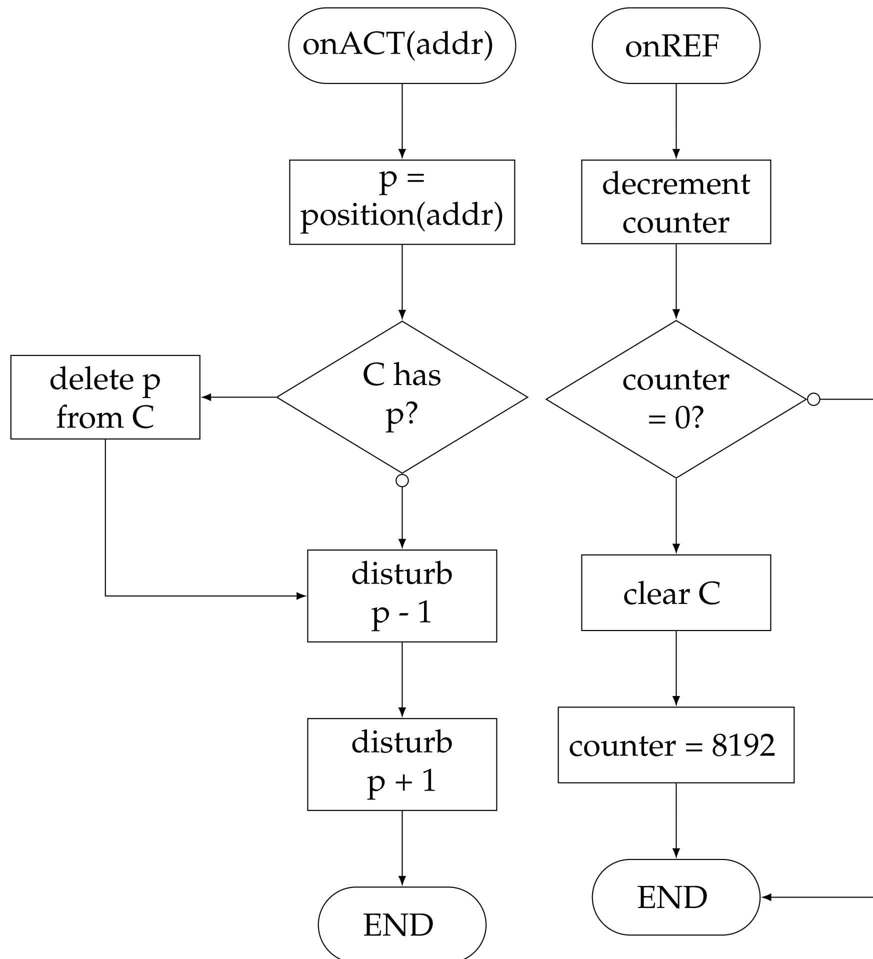


Figure 3.3: *ACT and REF callback functions*

The `disturb` function used in this algorithm is illustrated in Figure 3.4. Simulating the disturbance on a row consists in increasing its counter in the table C. If C already has an entry for the row, it is incremented; otherwise the entry is created and initialised at 1. When the count is above the corruption threshold T_{RH}, the program uses the `corrupt(p,n)` function to flip some bits in the row.

The corruption of the memory happens every time the neighbour of a row is issued an ACT from the moment the disturbance level of the victim row reaches T_{RH} and until it is either refreshed or activated.

All bits are not flipped at once, some bits are more vulnerable than others. The corruption simulation must be able to simulate progressive random corruption of the memory, illustrated in Figure 3.5.

The progressive nature of the corruption can be viewed as a function that takes the

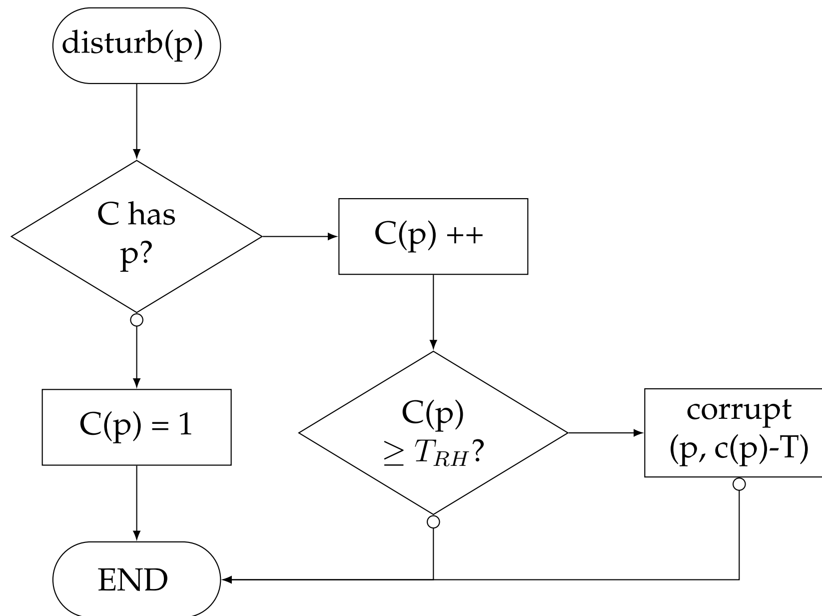


Figure 3.4: *disturbance simulation*

disturbance level minus the corruption threshold as input, and outputs the approximate portion (from 0 to 1) of bits that should be flipped at this point of the attack. When this value reaches 1, all bits of the row are flipped to 0. According to the approximation theory of mathematics, any function can be approximated to a polynomial function. Hence, to allow any function to be used for this while keeping the implementation simple, the configured function must be a polynomial function ①. The polynomial given in the example in Figure 3.5 is $-2 \times 10^{-9}x^3 + 3 \times 10^{-6}x^2$. Using this polynomial results in an ease-in-out function from 0 to 1000 ACTs after the threshold. As this function outputs the portion of bits that are flipped since the beginning of the attack, its derivative is the approximate number of bits that flip every time the disturbance level is increased. The derivative of an ease-in-out function is a gaussian function. Using this type of polynomial, we simulate an gaussian distribution of the corruption of all bits. Other polynomials can be used, to simulate for example ease-in or ease-out functions of any range. The sensitivity of bit cells for disturbance can be considered random and determined by process variations. To simulate this random sensitivity, a Pseudo-Random Number Generator (PRNG) ② is used to associate a random number between 0 and 1 to every bit cell of the row ③. This PRNG being seeded with the row position when entering the `corrupt` function, the random number associated with a bit is constant. A bit will only flip to 0 when the output of the polynomial function is greater than its associated random value. To flip a bit, the simulator simply sets the

appropriate bit to 0 in the memory.

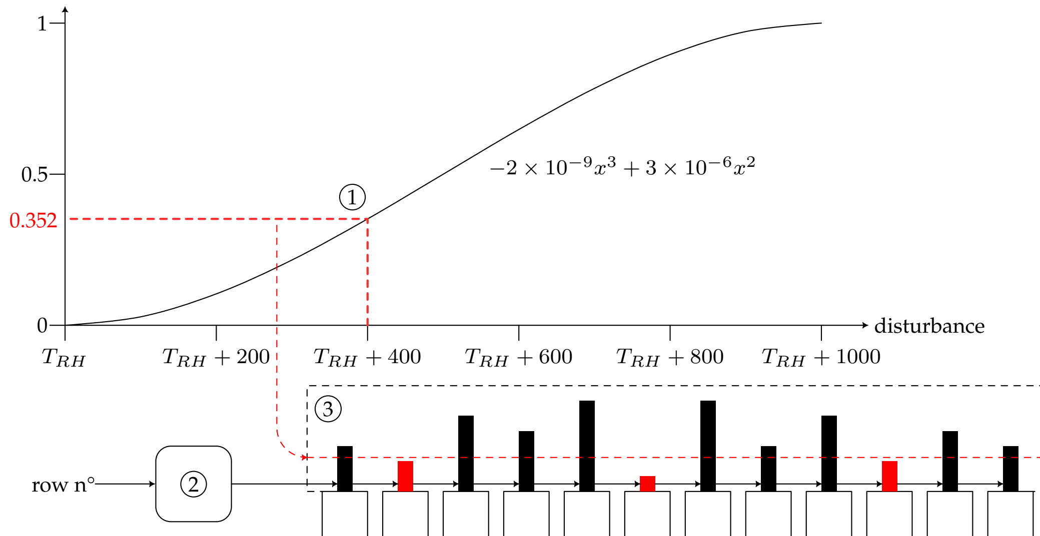


Figure 3.5: *Bit-flip simulation with polynomial equation. Squares on the bottom represent individual bit cells, each having its own unique threshold specified by the PRNG ②. After* $T_{RH} + 400$ *ACTs, all cells with an individual threshold below 0.352 will be corrupted.*

3.5 Mitigation integration in gem5

The main goal of simulating Rowhammer attacks is to design, evaluate and improve countermeasures. For this purpose, the simulator provides all the necessary information, events and functions to design hardware-based mitigation techniques. As illustrated in Figure 3.6, the memory-corruption module can be connected to a mitigation module, written in C++ and that inherits the provided `Mitigation` class. The mitigation module is notified of ACTs and REFs, and can use provided functions to issue additional ACTs to refresh victim rows, log events and generate statistics that will be saved at the end of the simulation.

Rowhammer countermeasure designers can integrate mitigation proposals that react when rows are activated, and refresh the neighbours of the aggressor rows to prevent the corruption. Countermeasures such as PARA [1] which randomly refreshes neighbours of activated rows, and Graphene [51] which tracks the most activated rows using the Misra-Gries algorithm, are good examples of this type of Rowhammer countermeasures. The simplified code for the integration of PARA is presented in Listing 3.1. In this code, functions `onACT` (resp. `onREF`) are called when an ACT (resp.

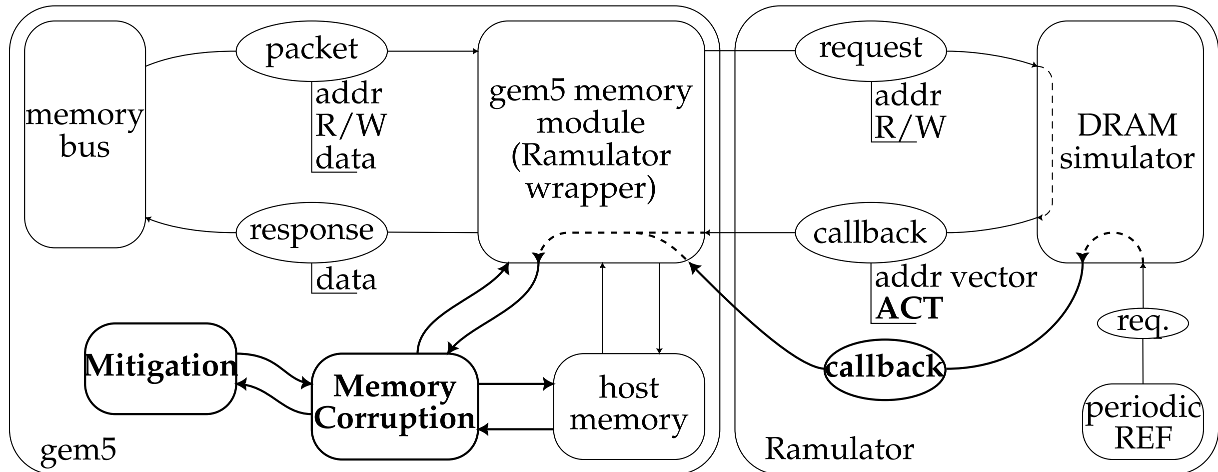


Figure 3.6: *gem5 and Ramulator memory architecture, with Memory-Corruption module and Rowhammer mitigation*

a REF) command is issued. The `addr` argument is the address vector, containing the addressed channel, rank, bank, bank group, row and column. The `neighbor` function stores in the third argument the address vector for the neighbour row, with the offset specified in the second argument relative to the address vector in first argument. This function returns `true` if the neighbour exists, `false` otherwise. Finally, the function `refresh` performs an additional dummy read operation on the target row.

3.6 Usage, limitations and evaluation

3.6.0.1 Usage, configuration

gem5 modules can generate statistics, which are written in a file at the end of the simulation. Both the memory-corruption module and the mitigation module can generate statistics. The memory-corruption module generate statistics for the number of bit-flips, and the maximum disturbance level registered by the module. This can help to evaluate the efficiency of attacks, and the protection offered by the countermeasure. The mitigation module generates statistics for the number of additional refreshes issued, and designers can add statistics of their own.

This module is intended to be used to simulate Rowhammer attacks and mitigations. The DRAM technology keeps evolving, changing the vulnerability of the mem-

```
class PARA : public Mitigation
{
    float p; //!< refresh probability
public:
    PARA(float p) : Mitigation(), p(p)
    { }
    void onACT(const std::vector<int>& addr)
    {
        float r = ranged_random(0,1);
        std::vector<int> adj;
        if (r < p/2)
        {
            if (neighbor(addr, -1, adj))
            {
                refresh(adj);
            }
        }
        else if (r < p && neighbor(addr, 1, adj))
        {
            refresh(adj);
        }
    }
    void onREF()
    {
        // do nothing
    }
}
```

Listing 3.1: *PARA implementation in gem5 with memory-corruption module*

ory to Rowhammer attacks. To make this module as versatile as possible, it needs to to be adaptable to various cases. In addition to the mitigation integration, multiple parameters can be used to configure the memory corruption module to adapt it to various conditions.

The **internal mapping of rows** in the memory chip can change between two DRAM modules. Two adjacent rows in one chip may not be adjacent in another chip [76]. The memory-corruption module can take a DRAM layout file as an optional parameter, containing the physical position of each logical row in a bank. Table 3.2 illustrates an example where rows 0, 1, 3, 6 and above are at their proper positions, whereas rows 2, 4 and 5 are respectively at positions 4, 5 and 2. This means that when using this configuration, the row 2 is adjacent to rows 3 and 4 in all the banks of the memory.

The **corruption threshold** T_{RH} depends on the technology, the manufacturing pro-

Table 3.2: *DRAM layout configuration example.*

logical row	0	1	2	3	4	5	6
physical row	0	1	4	3	5	2	6

cess and its variations. It ranges from around 10,000 for recent LPDDR4, to 50,000 for early DDR4 and 139,000 for some DDR3 memory modules [15]. Its value can be configured in the memory-corruption module to fit the target DRAM modules.

The **bit-flip probability polynomial** used to progressively flip the bits in the memory can also be configured, to either flip all the bit at once using the constant polynomial 1, to change the speed at which the memory corrupts itself past the threshold, or to keep some random bits not flipped. The polynomial used to illustrate the progressive corruption in Figure 3.5 page 37 uses the polynomial $-2 \times 10^{-9}x^3 + 3 \times 10^{-6}x^2$, that produces an ease-in-out curve between 0 at T_{RH} to 1 at $T_{RH} + 1000$. By default, the constant 1 is used, which means that all bits are flipped when the disturbance level reaches T_{RH}.

Finally, the corruption of the memory can be deactivated. In this case, the bit-flips are not simulated, but the event of the disturbance level reaching the threshold is still logged.

3.6.1 Limitations

The simulation of Rowhammer attacks and countermeasures by this simulator is limited in some points. First, the simulation of disturbance and corruption is not a perfect reproduction of the effect on physical systems. At the time of writing, the physical phenomenon behind the bit-flips caused by Rowhammer attacks is not entirely understood. Even among the known parameters that affect the corruption of the memory in real systems, some are not integrated in the memory-corruption module.

First, the corruption threshold must be fixed before the beginning of the simulation. The temperature at which the memory is working has an effect on the corruption, but the evolution of temperature and its effects on the corruption cannot be integrated in the simulator.

Second, the data pattern stored in the memory was proved to have some effect on the bit-flips [15]. Row stripes of 1s and 0s, or a checker board alternating 1 and 0 for every bit seems to have a significant effect on the bit-flips. However, the cause of this effect is not entirely understood and varies between memory generations. Consequently, we did not integrate this effect in the memory-corruption module.

Third, the internal layout of the banks is not only unknown to the memory controller, it can also be different for every bank. But for the sake of simplicity, it was decided that the simulated memory will have a common layout in all banks.

Finally, it has been recently demonstrated that the victim row is not necessarily adjacent to its aggressors [15, 22]. However, simulating the mechanisms behind this effect would involve a complicated configuration from the user, and the simulation would most probably not be very accurate. Therefore, this effect is not yet integrated in the simulator.

3.6.2 Evaluation

Adding another module to a simulator inevitably adds an overhead in processing time and resources usage.

Multiple simulations were made, using both ARM and x86 architecture, to compare the performance of the simulation with the memory-corruption module enabled and disabled. Three benchmark programs were used: a simple Rowhammer attack, the STREAM benchmark, and a Linux OS boot and shutdown. These programs were chosen for their memory usage and representation of a typical usage of a Rowhammer simulator. For this evaluation, gem5 was configured to use one TimingSimpleCPU running at 1GHz; two 32KB L1 caches (one L1-D and one L1-I); one 512KB L2 cache; and finally a DDR4 DRAM at 2400MHz as the main memory, with a storage limit of 4GB, handled by Ramulator. The results of this evaluation are displayed in Table 3.3.

We used Valgrind [77, 78] when launching the simulation to measure the peak memory usage. As doing so greatly lengthen the duration of the simulation, it could not be done for the simulation of the boot and shutdown of Linux. The measurements

show that there is no noticeable difference in timings for all tested benchmarks, and almost no difference in peak memory usage.

Table 3.3: *Impact of the Memory-Corruption module (M-C) on simulation performance.*

Bench-mark	**time** (avg $\pm\sigma$) M-C enabled	**time** (avg $\pm\sigma$) M-C disabled	peak **memory** usage (M-C en.)	peak **memory** usage (M-C dis.)
STREAM	6m32s $\pm$ 2.3% (8 samples)	6m30s $\pm$ 1.3% (8 samples)	x86: 885.2MiB	x86: 885.2MiB
RowH. attack	11.47s $\pm$ 3.5% (10 samples)	11.63s $\pm$ 3.5% (10 samples)	x86: 890.44MiB ARM: 941.13MiB	x86: 890.44MiB ARM: 938.14MiB
Linux boot	31m10s $\pm$ 7.7% (5 samples)	30m58s $\pm$ 5.3% (5 samples)	not measured	not measured

3.7 Conclusion

In this chapter, we presented an improvement of the gem5 architecture simulator to simulate the memory corruption caused by Rowhammer attacks. The new module we created is attached to the memory controller of gem5, intercepts row activations and periodic refreshes to simulate the disturbance between adjacent rows, and performs the bit-flips in the memory to simulate the bit-flips. Various Rowhammer mitigation mechanisms can be connected to the memory-corruption module to validate their efficiency in protecting a system against Rowhammer attacks. However, this module has several limitations: complex corruption mechanisms such as the influence of the data pattern in the memory are not integrated, and only mitigation mechanisms that can be integrated are hardware-based mitigation and that produce additional REFs to victim rows can be implemented. Finally, This corruption simulation is configurable on multiple key parameters such as the corruption threshold and the adjacency of the rows in the memory, and outputs statistics and logs using the dedicated functions of gem5.

This work resulted in two publications in conferences [79, 80] and communications in national symposiums [81] and in an international conference [82].

IV

Counter-based Rowhammer mitigations improvement

Contents

4.1 Motivation **44**

4.2 Bank-level and rank-level counting granularity **44**

4.3 Implication in State-of-the-art mitigation proposals **46**

4.4 Considerations for technology and timings **49**

4.4.1 DDR generation parameters 49

4.4.2 Feasibility - timing considerations 50

4.5 Conclusion **51**

4.1 Motivation

Rowhammer mitigation development is a very active domain of research. Over the past years, numerous countermeasures have been proposed. Counter-based mitigation techniques, implemented as hardware components in the micro-architecture, are among the most performant proposals. However, they come at the cost of requiring an additional memory to store the counters. Most counter-based mitigation proposals come with a bank-level counting granularity, with one set of counters per bank. The total silicon area overhead of implementing this kind of countermeasure is approximately the silicon area overhead of the bank-level protection multiplied by the number of banks. In this work, we evaluate how the storage requirement could be reduced when changing the counting granularity of counter-based Rowhammer mitigation techniques from bank level to rank level, without affecting their protection level.

4.2 Bank-level and rank-level counting granularity

The objective of Rowhammer attacks is to send enough ACTs to neighbours of a victim DRAM row without directly accessing it, to corrupt some bits in it. Because of limitations such as the unknown layout of the memory, counter-based algorithms cannot measure the disturbance of victim rows because they don't know what are the adjacent rows of activated ones. Instead, they try to detect when rows are used as aggressors by counting their activations. A DRAM row must be considered as an aggressor row by the mitigation mechanisms before it is issued enough ACTs to corrupt a victim neighbour row. As the two neighbours of a row can be used to disturb it T_{RH} times, an attack is considered successful if an aggressor is issued $HC_{first} = T_{RH} \div 2$ ACTs.

In general, the required memory overhead to guarantee the detection of a Rowhammer attack depends on (1) the bitwidth of each counter and (2) the total number of counters. The number of bits per counter S directly depends on the detection threshold, which is calculated from the corruption threshold T_{RH}. In principle, we would need to keep count of each row separately. However, having one counter per row is excessively expensive (*e.g.*, a typical DDR4 bank has $2^{16} = 64K$ rows, which would require

approximately 1MiB of counter storage per bank for a T_{RH} of 32768 [15]). Thus, most existing counter-based detection mechanisms include methods to minimise the number of counters while still guaranteeing detection. The parameters of these methods are the detection threshold, and W the maximum number of ACTs that can be issued during t_{REFW}.

The detection threshold determines the vulnerability of the memory to corruption, and is fixed after fabrication. W is determined by the DRAM timings, and the counting granularity. At bank level, row activations are limited by the interval between two ACTs t_{RC}, and the periodic refreshes defined by t_{REFI} and t_{RFC}. Hence, W at bank level W_B can be calculated using the formula

$$W_B = \left\lceil \frac{t_{REFW} \times \left(1 - \frac{t_{RFC}}{t_{REFI}}\right)}{t_{RC}} \right\rceil. \tag{4.1}$$

Alternatively, considering the rank-level counting granularity, row activations is not limited by the individual W_B of the banks. Row ACTs in a rank are limited to four times during t_{FAW}. As a consequence, not all banks can be accessed at their maximum frequency. DDR3 and DDR4 technologies use an all-bank refresh mechanism, which refreshes all banks at the same time when issuing a REF command, keeping the rank busy for t_{RFC} every t_{REFI}. DDR5 standard introduced the single-bank REF command [83] which allows the banks of a rank to be refreshed individually to keep the other banks available, as $t_{RFC} \ll t_{REFI}$. Hence, the value of W at rank level W_R can be calculated using the formula

$$W_R = \left\lceil \frac{t_{REFW}(1 - \frac{t_{RFC}}{t_{REFI}})}{t_{FAW} \div 4} \right\rceil \tag{4.2}$$

for DDR3 and DDR4 memories, and the formula

$$W_R = \left\lceil \frac{t_{REFW}}{t_{FAW} \div 4} \right\rceil \tag{4.3}$$

for DDR5 memories.

Considering N_{bank} as the number of banks per rank, the total number of counters in a rank for bank-level detection is N_{bank} times the number of counters per bank. If

the number of counters is proportional to W, the effective value of W for the sum of all bank-level detection mechanisms in a rank is $N_{bank} \times W_B$. Hence, when $N_{bank} \times W_B > W_R$, a rank-level detection mechanism has a lower effective W than the sum of all bank-level detection mechanisms. As a consequence, moving to rank-level counting granularity could reduce the number of counters needed to protect the memory against Rowhammer attacks.

For the memory modules whose timings are detailed in Table 2.1 page 12, the values of W_B, W_R are listed in Table 4.1. In this table, the reduction of W is calculated using the formula

$$W_{red} = 1 - \frac{W_R}{W_B \times N_{bank}}. \tag{4.4}$$

This table shows that as the technology evolves and the standard allows more banks per rank, the reduction of W when moving from bank-level to rank-level counting granularity increases, from 19% for DDR3 to 62% for DDR5.

Table 4.1: *Values of W_B and W_R, number of banks per rank, and theoretical reduction of total number of counters, for DDR3, DDR4 and DDR5 (c.f., Table 2.1 page 12).*

Memory	W_B	W_R	N_{bank}	W reduction
DDR3	1.25×10^6	8.15×10^6	8	19%
DDR4	1.33×10^6	11.3×10^6	16	47%
DDR5	6.61×10^5	8.00×10^6	32	62%

4.3 Implication in State-of-the-art mitigation proposals

To quantify the memory reduction resulting from the reported W reduction, we consider two recently-published countermeasures, namely Graphene [51] and BlockHammer [48]. Since these countermeasures were only dimensioned for DDR4 memories, the results in this section are restricted to DDR4.

Graphene. Graphene stores row addresses and counters in a Content-Addressable Memory (CAM) and uses the Misra-Gries algorithm [49] to only count the ACTs on the N_{entry} most activated rows of the bank. Following the original publication, the

minimum number of entries in the CAM is calculated using the formula

$$N_{entry} = \left\lfloor \frac{W}{T_{RH} \div 4} \right\rfloor . \tag{4.5}$$

An entry in this CAM includes a key-value pair, where the key is the row address and the value is a counter, plus one overflow bit for the counter. This overflow bit is used as a trigger to detect when the victim has been activated too many times, and its neighbours need to be refreshed. The counter must be capable of holding a value up to $HC_{first} \div 2 - 1$. For a typical corruption threshold $T_{RH} = 32768$ [15], this means a maximum value per counter of $HC_{first} \div 2 - 1 = 8191 = 2^{13} - 1$, hence a counter of 13 bits. The address part of the entry must be able to identify every rows. Considering the bank-level counting granularity, as proposed in the original publication, and DDR4 banks with 65536 rows, the address part of the entry would be 16 bits wide, for a total entry size of $S = 16 + 13 + 1 = 30$ bits. According to Equation 4.5 with $W = W_B$, a prototypical DDR4 memory bank needs 162 CAM entries. Thus, the total CAM size is 16 banks $\times$ 162 entries $\times$ 30 bits $= 9.49$KiB per rank (5.06KiB for the keys, 4.43KiB for the rest).

Alternately, considering the rank-level counting granularity, the entries would require more space to fit the row unique address. As a DDR4 rank can contain up to 16 banks, the row address must be 4 bits bigger than its bank-level equivalent to uniquely identify each row. As a result, the total entry size becomes $S' = 20 + 13 + 1 = 34$ bits. According to Equation 4.5 with $W = W_R$, a prototypical DDR4 memory rank needs 1394 CAM entries. Thus, the total CAM size is 1394 entries $\times$ 34 bits $= 5.71$KiB per rank (3.36KiB for the keys, 2.35KiB for the rest).

For one rank, the additional storage required by Graphene with rank-level counting granularity is 40% lower than the sum of 16 additional independent storages required by graphene with bank-level counting granularity.

BlockHammer. BlockHammer uses Counting Bloom Filters (CBF) [84] to count the number of ACTs for every rows. A CBF works by associating each possible input value to a unique set of k counters out of a total of m counters, using a hash function. When a row is activated, the CBF uses the hash function to determine the associated

k counters, and increments them. When a row is activated and all its associated counters are above a pre-defined threshold N_{BL}, the mechanism considers the row as an aggressor. It communicates with the processor to slow down the process that activated the aggressor row, until the row is refreshed. Every t_{REFW}, all the counters of of the CBF are reset to take the periodic refresh into account. To avoid missing an attack that would not target a row synchronously refreshed with the reset of all counters, BlockHammer uses two CBF. They are alternately refreshed every $t_{REFW} \div 2$, and both are incremented on an ACT, but only one is considered at a time. The principle of this mechanism is illustrated in Figure 4.1.

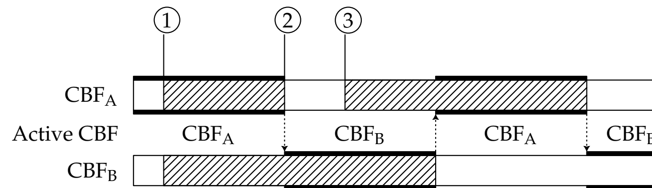


Figure 4.1: *Alternating CBFs on BlockHammer. The Two CBFs count all the accesses, but only one of them is taken into account at a time. When The active CBF detects an aggressor (1), it blacklists the row. Every $t_{REFW} \div 2$, the active CBF is cleared, deactivated and the other CBF is activated (2). If the inactive CBF detects an aggressor (3), it is not taken into account until it gets activated.*

This mechanism uses a reduced set of counters to count the accesses to all the rows. As counters are shared between multiple rows the detection mechanism must be designed to be able to distinguish aggressor rows from other accessed rows. As a comprehensible and representative way to evaluate the capability of BlockHammer to discriminate aggressor rows from benign rows, we introduce the noise level n. The noise level is the mean value reached by counters after W ACTs evenly-distributed across all rows of the considered memory. It must be significantly lower than the detection threshold N_{BL} to make the counters associated to aggressor rows stand out from counters incremented by benign memory accesses. The value of n is calculated using the formula

$$n = \frac{W \times k}{m}. \tag{4.6}$$

The bank-level implementation, proposed by the authors of BlockHammer, includes $m = 1024$ counters per CBF, $k = 3$, and $N_{BL} = 8192$. For the considered DDR4 with $W_B = 1.33 \times 10^6$, we have a noise level of $n \approx 3896$. This means that during normal but memory-intensive operation, most counters will likely reach this value. The mech-

anism can easily distinguish aggressor rows, for which the counters will reach N_{BL}, approximately twice the value of n.

When considering the rank-level implementation, the noise level must be kept at a similar value to maintain the protection level. As $W_R = 11.3 \times 10^6 \approx 8.5 \times W_B$, we selected $m = 8192$ and $k = 3$. For simplicity, we chose to keep k constant, as it is already a very small number. We selected m as the closest power of two to keep a similar noise value. With these values, we have a noise level of $n \approx 4138$.

The number of bits per counter is only defined from N_{BL}, and therefore does not change between bank-level and rank-level counting granularities. As a standard DDR4 has 16 banks per rank, and the rank-level CBF has 8 times the number of counters compared to bank-level CBF, the total size occupied by the counters for the rank-level counting granularity is half of what it is for the bank-level counting granularity.

For both Graphene and BlockHammer, the additional memory required by the mechanism can be reduced by 40% to 50% when changing the counting granularity from bank-level to rank-level.

4.4 Considerations for technology and timings

4.4.1 DDR generation parameters

Even though the proposed mitigation techniques were originally designed for DDR4 memory modules, they can be easily adapted to other generations of DDR memories. When changing the counting granularity from bank level to rank level, the storage reduction should follow the reduction of W calculated in Table 4.1 page 46. Table 4.2 lists the storage reduction that can be obtained when moving Graphene and BlockHammer to rank-level counting granularity, for DDR3, DDR4 and DDR5.

Table 4.2: *Values of W_B and W_R, number of banks per rank, and theoretical reduction of total number of counters, for DDR3, DDR4 and DDR5 (c.f., Table 2.1 page 12).*

		Bank level	Rank level	reduction
DDR3	W total ($8 \times W_B$, W_R)	10×10^6	8.15×10^6	19%
	Graphene CAM	$4.45KiB$	$4.00KiB$	11.2%
	BlockHammer CBFs	$26KiB$	$26KiB$	0%
DDR4	W total ($16 \times W_B$, W_R)	21.28×10^6	11.3×10^6	47%
	Graphene CAM	$9.61KiB$	$5.79KiB$	40%
	BlockHammer CBFs	$52KiB$	$26KiB$	50%
DDR5	W total ($32 \times W_B$, W_R)	21.15×10^6	8×10^6	62%
	Graphene CAM	$9.38KiB$	$4.05KiB$	57%
	BlockHammer CBFs	$52KiB$	$19.5KiB$	62.5%

4.4.2 Feasibility - timing considerations

An important issue that arises when changing the counting granularity from bank-level to rank-level is the capability of the countermeasure to withstand the shorter delay between two ACTs. for a prototypical DDR4 memory, at bank level, the memory controller can issue one ACT every $45ns$. Most counters can perform all the operations within this period. However, at rank level, the memory controller can issue one ACT every $t_{FAW} \div 4 = 5.42ns$ on average. BlockHammer only introduces a $1ns$ latency before issuing ACTs to check the safety of the operation. It can therefore withstand the shorter delay between two ACTs at rank level. However, due to a high constraint on the power line, CAM used by Graphene and some other mitigation proposals have a tendency to be slower the bigger they get [85]. Consequently, current CAM designs might not be able to withstand the shorter ACT-to-ACT period of rank-level counting granularity. This issue has been pointed by a previous study for the mitigation proposal CAT-TWO [47], for which the authors pointed that the reduced frequency of the CAM would render their proposal unusable at rank-level counting granularity. Nonetheless, we expect that future CAM designs will be able to perform the searchs at a fast-enough rate to allow CAM-based detection mechanisms to move to rank-level counting granularity. Pipelined CAM [86, 87] could be a potential solution for this. CAM-based detection mechanisms do not suffer from the latency of the CAM as the ACTs are not delayed while being processed by the mechanism. Only the throughput matters. The increased latency introduced by the use of pipeline circuitry does not

bring a decrease of throughput. On the contrary, as the number of active entries drastically decreases as the search progresses through the pipeline, using this method would greatly reduce the constraint on the power lines, and therefore increase the throughput sufficiently to make it usable for Rowhammer detection.

4.5 Conclusion

In this chapter, we have calculated the potential reduction of the required memory that could be obtained by changing the counting granularity of counter-based Rowhammer mitigation techniques from bank-level to rank-level. For the DDR4 technology, the minimum delay between two ACTs at rank level is approximately 8 times lower than the minimum delay at bank level, while there is up to 16 banks per rank. Hence, during one refresh window t_{REFW}, one can send 8 times more ACTs to a rank than it can to a bank. Additionally, as for counter-based Rowhammer mitigation techniques, the number of counters is proportional to the number of ACTs that can be issued to the memory during t_{REFW}, a mitigation technique with rank-level counting granularity requires approximately 8 times the number of counters that is required at bank level, despite the rank having up to 16 banks. Therefore, a mitigation mechanism with rank level counting granularity would have approximately half of the counters compared to the sum of all 16 mitigation mechanisms with bank level counting granularity. For the two state-of-the-art mechanisms tested, Graphene and BlockHammer, the total memory size can be decreased by 40% to 50% for typical DDR4 memory modules.

However, the shorter timings between two consecutive ACTs at rank level must be taken into account. Some mitigation techniques such as Graphene might not be able to withstand the increased activation rate at rank level.

Finally, with the introduction of the recent DDR5 standard that allows up to 32 banks per rank, this reduction will get more important.

This work resulted in one communications in a workshop [88].

V

Mitigation proposals

Contents

5.1 Motivation **54**

5.2 Hardware counters and machine learning for Rowhammer detection . **54**

5.2.1 Methodology 55

5.2.2 Experiments and results 57

5.2.3 Conclusion 60

5.3 F-CorD: Forgetful Counters for Rowhammer Detection **61**

5.3.1 Introduction: Unsynchronised refresh issue for counter-based Rowhammer mitigation 61

5.3.2 Tracking frequently-activated rows 62

5.3.3 Detecting attacks 64

5.3.4 Discretisation 68

5.3.5 Periodic maintenance 69

5.3.6 Implementation details 71

5.3.7 Number of entries 73

5.3.8 Example 74

5.3.9 Conclusion 75

5.1 Motivation

To this day, Rowhammer attacks are still important threats. Many mitigation techniques were proposed over the past decade, based on various concepts, with different advantages and drawbacks. However, manufacturers have only implemented proprietary mechanisms that were shown to still be vulnerable to (sometimes specifically-designed) Rowhammer attacks [34]. Consequently, Rowhammer mitigation is still an active research field. The main issue with Rowhammer mitigation is the detection of aggressor rows or attack processes. Once an attack is detected, various mitigation mechanisms can be used to prevent the corruption. In this chapter, we propose two new Rowhammer detection mechanisms. The first one uses microarchitecture event counters embeddeded in the hardware to generate traces, and based on a machine-learning algorithm to classify traces from these counters as depicting an attack or a normal behaviour of the system. The second one uses row activation counters and evaluates the activation frequency of every row to detect attacks.

5.2 Hardware counters and machine learning for Rowhammer detection

In this section, we propose a method to create a machine-learning-based detection mechanism to be implemented in hardware, that is able to detect attacks in a few hundreds of microseconds. This Rowhammer detection mechanism inserts probes in the microarchitecture to count microarchitecture events, generate traces from these counters, and feed these traces to an artificial neural network to classify said traces as depicting a normal behaviour or an attack.

To our knowledge, this is the first ML-based Rowhammer detection mechanism proposal that targets a hardware implementation. A prior publication by Chakraborty et al. [36] proposed a software-based mitigation mechanism, that monitors the LLC miss rate to detect suspicious processes. It then records DRAM bank and row accesses from this process, and uses a Convolutional Neural Network (CNN) to categorise the

access pattern as being from an attack process or not. While this solution does not require any hardware modification, it takes a lot of time (1.5s on average) to detect an attack process after it has begun. However, Rowhammer attacks must perform a bit-flip in less than t_{REFW} =64ms, and a process that already has access to the aggressors close to the desired victim can produce a bit-flip in less than 10ms [1]. Therefore, requiring 1.5s to detect attack processes from their DRAM access pattern seems insufficient.

5.2.1 Methodology

The creation of the mechanism is divided into four steps:

1. *Features selection*: selecting the list of hardware events to trace;
2. *Simulation*: generating traces by simulating and monitoring the system this mechanism will be implemented in;
3. *Training and testing*: the generated traces are divided into a training set and a testing set, and used to train the ML model and evaluate it.
4. *Hardware integration*: implementing the detection algorithm into the system.

Features Selection. When a Rowhammer attack is running, it has some influence on microarchitectural events. The list of events that are traced for the detection should allow the mechanism to distinguish attacks and benign behaviour of the system. While variety in the traced events is important to properly detect attacks and reduce the false positive rate, selecting redundant events could make the implementation more expensive in terms of silicon area, energy consumption and/or detection time without significant benefits for the precision of the detection.

The goal of the Rowhammer attack is to flip bits in the main memory by rapidly and repeatedly activating DRAM rows. The attack will therefore generate a lot of row conflicts in the DRAM banks, and very few row hits, which are the first events that the mechanism will trace. To access the main memory, the attack must bypass all the cache levels, flushing the aggressor rows in it. The first-level cache (L1) is split into L1-I for

instructions, and L1-D for data. The hammering loop of a Rowhammer attack is a very short loop. Once its instructions are stored in the L1-I, it is not likely to generate a lot of cache misses on this cache. Hence, the L1-I will experience more cache hits than cache misses, where the L1-D will experience more cache misses. Therefore, cache hits and cache misses of L1-I and L1-D must be traced. The activity of other cache levels can be partially deducted from the cache misses in L1 and the activity of the main memory. Therefore, tracing cache hits and cache misses on other levels is redundant. The selected features are the following ones: L1-I hits, L1-I misses, L1-D hits, L1-D misses, row hits and row misses.

Simulation. Using gem5 [66] (*cf* Chapter 3), we configure the architecture on which the mechanism works. The simulator must be configured to log the selected features when simulating the programs. While cache-related events can be logged by gem5 directly, row buffer-related events are not logged by Ramulator, which is used to simulate the main memory. The Memory-Corruption module introduced in chapter 3 can be used to log row hits and row conflicts. Various programs to run are chosen, including attack programs and memory performance benchmarks to stress the components that would also be targeted by the attack. These programs are run sequentially on a single simulated processor, or in parallel using multiple simulated processors.

Training and testing. The feature traces are extracted from the output files of gem5. Logged events are transformed into fixed-duration samples, and grouped into windows (fixed-length buffers) labelled as attack or no attack. Two consecutive windows are partially overlapping: the second half of the nth window is repeated for the first half of the $(n+1)^{th}$ window. The overlap ratio between two consecutive windows can be changed at will. The windows are then randomly selected for either the training dataset or the testing dataset. Multiple ML models are trained and tested with the datasets. To select the model to use for the final implementation, the accuracy, memory usage and inference time when running in software must be taken into account.

Hardware integration. Once the ML model is configured and able to classify event windows as coming from an attack or not, it can be implemented as an online detection mechanism in the system architecture. The integrated mechanism is illustrated in Figure 5.1. The implemented mechanism can be viewed as three main components:

hardware counters ①, sample buffers ②, and finally the neural network ③.

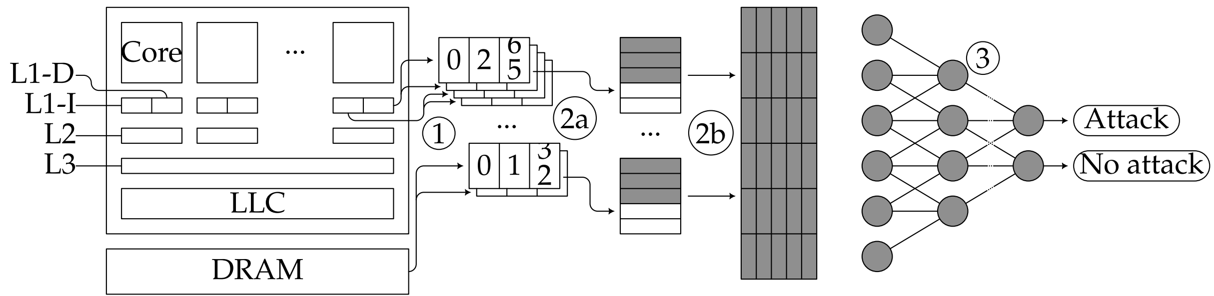


Figure 5.1: *Machine Learning detection mechanism integration in computer architecture*

① Hardware counters are directly integrated in the architecture, and count the selected events. Each core has a separate set of counters to count the hit and miss events on L1-I and L1-D.

② (a) Counters are regularly read, copied in buffers and cleared. (b) Once the buffers are full, they are copied in the input buffer of the neural network and partially cleared, with respect to the overlap ratio between consecutive windows.

③ The implemented neural network processes its input buffer, and classifies the window as depicting a normal behaviour or an abnormal one.

5.2.2 Experiments and results

For simplicity, the machine learning model training and testing are performed directly on a laptop computer with an Intel® Core™ i7-8565U CPU @ 1.80GHz, 16GB RAM running Windows 10, version 19042. The system simulation using gem5 is running on a server. Multiple system architectures are used to generate the datasets to increase the variety in the datasets. The configurations include one or two CPUs running at 1 GHz, using the included CPU classes TimingSimpleCPU which considers the timings of each instructions, or DerivO3CPU which in addition is capable of out-of-order execution. The system uses two 32KiB L1 caches per CPU (one L1-D and one L1-I), one global 512KiB L2 cache, and finally a DDR4 DRAM at 2400MHz handled by Ramulator [65] as the main memory, with a storage limit of 4 GiB.

To generate datasets, we selected two different programs to run on the simulated

system. The first program is the memory-intensive STEAM benchmark [89], which is made to test the performance of the memory. The second program is a combination of Rowhammer attack with random memory accesses. The Rowhammer attack is the assembly loop of Listing 2.1 page 16, and the random memory accesses is written in Listing 5.1. In this code, the function `rand()` selects a random address within a 2^{18}-bytes array. Both loops are executed alternately multiple time, with a random duration (where the maximum duration is 5 times the minimum duration), with approximately 50% of the total execution time each. Both the STEAM benchmark and the attack/random-accesses program are memory-heavy programs, so the machine learning model will have to recognise attack patterns and not only memory-heavy programs.

```
rand_loop:
    mov rand(), %eax ; put a random address into %eax
    mov (%eax), %ebx ; read value at address %eax
    jmp rand_loop    ; restart the loop
```

Listing 5.1: *Random memory access x86 assembly loop*

From the simulation log file, we generate 100ns samples, which we group by 100 into 10µs windows with a 50% overlapping between consecutive windows: the last 50 samples of a window is reused as the first 50 samples of the next window. The program counter is used to categorise every window as containing an attack or not.

Three different ML models are tested: Long Short-Term Memory (LSTM) [90], Multi-Layer perceptron (MLP) [91] and Convolutional Neural Network (CNN) [92, 93].

The machine learning models are build using Keras [94], with the python codes presented in Listings 5.2, 5.3 and 5.4. In these listings, `n_timesteps = 100` is the number of samples per window, and `n_features = 6` is the number of different features logged by gem5 and used as inputs. All three models take the `n_timesteps` $\times$ `n_features` samples window buffer as input, and have 2 outputs ("attack" and "no attack"). All three models are modified models originally created by Jason Brownlee on Machine Learning Mastery [95]. The LSTM and CNN models were originally intended for human activity recognition [96, 97] with a 9×128 input dimension, and the MLP model for time series forecast [98].

```
model = Sequential()
model.add(LSTM(100, input_shape=(n_timesteps,n_features)))
model.add(Dropout(0.5))
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs, activation='softmax'))
model.compile(loss='categorical_crossentropy',
        optimizer='adam', metrics=['accuracy'])
```

Listing 5.2: *Python code to build the LSTM model*

```
model = Sequential()
model.add(Permute((2,1), input_shape=(n_timesteps, n_features)))
model.add(Dense(n_timesteps // 2))
model.add(Flatten())
model.add(Dense(128, activation="relu"))
model.add(Dense(n_outputs, activation="softmax"))
model.compile(loss='categorical_crossentropy',
        optimizer='adam', metrics=['accuracy'])
```

Listing 5.3: *Python code to build the MLP model*

```
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
        input_shape=(n_timesteps,n_features)))
model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
model.add(Dropout(0.5))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs, activation='softmax'))
model.compile(loss='categorical_crossentropy',
        optimizer='adam', metrics=['accuracy'])
```

Listing 5.4: *Python code to build the CNN model*

They are trained and tested with datasets generated with the simulation of the architecture in multiple load conditions:

- Isolated execution: only one program is run at a time;
- Concurrent execution: two programs are run in parallel on two simulated cores.

To extend the datasets, both configurations are run on in-order CPUs and out-of-order CPUs. The accuracy of the different models are displayed in Table 5.1. The processing times are listed for a software execution of the ML models. While it does not represent the processing time with a hardware implementation, the ratio between the different models will likely be similar.

ML model	Load	Accuracy (%)	FP (%)	FN (%)	Processing overhead (running in software)
LSTM	Isolated	99.9447	0.0527	0.0026	236µs
	Low load	99.8861	0.0902	0.0237	246µs
MLP	Isolated	99.9824	0.0167	0.0009	7.5µs
	Low load	99.7675	0.2183	0.0142	6.9µs
CNN	Isolated	99.9851	0.0140	0.0009	40µs
	Low load	99.9715	0.0285	0	53µs

Table 5.1: *ML models categorisation accuracy*

This table shows promising results, as all three tested models are able to detect with great accuracy when the system is under attack or not. The CNN seems to offer the best accuracy for the tested datasets, with more than 99.97% accuracy in any case, and with less than 0.001% FN. However, with a software execution, the processing time is higher with the CNN model than with the MLP model.

5.2.3 Conclusion

This proof of concept demonstrates that machine learning can be used to detect Rowhammer attacks from hardware event traces with good accuracy. Nonetheless, this solution requires an important silicon area overhead to store the input buffer and machine learning model, and does not guarantee the detection of all attacks. To make it competitive against state-of-the-art algorithmic Rowhammer detection mechanisms, the machine learning models and traces must be optimised further to reduce the silicon area overhead required and to improve the detection accuracy.

Interestingly, ML-based solutions may become more attractive than counter-based solutions in the future. Indeed, counter-based algorithmic countermeasures use a number of counters that is inversely proportional to the Rowhammer threshold. As cell-to-cell disturbance worsen over the years [15], this number of counter will continue to rise proportionally. ML-based countermeasures, on the contrary, will not scale in an inversely proportional way with the Rowhammer threshold. It must be dimensioned and trained according to the memory technology it has to protect, but even if the dimensions of the ML models will certainly grow with the reduction of the thresh-

old, this evolution will certainly not be proportional.

5.3 F-CorD: Forgetful Counters for Rowhammer Detection

5.3.1 Introduction: Unsynchronised refresh issue for counter-based Rowhammer mitigation

To this day, counter-based Rowhammer mitigation proposals offer the best protection guarantee. These solutions use row activations counters to detect aggressor rows. Contrary to probabilistic solutions, counter-based proposals can offer a guaranteed protection against bit-flips, where an aggressor will always be detected before the attack succeeds.

To account for periodic refreshes of the DRAM rows, the counters in a counter-based mitigation technique must be periodically reset. However, the periodic refresh of all DRAM rows does not happen at the same time in the cycle. Refreshes are spread out in the short chunks of t_{RFC} every t_{REFI}. But as memory controller is not aware of which row is refreshed every t_{REFI}, the reset of a counter cannot be synchronised with the refresh of the row it is watching. As a result, most existing counter-based mitigation techniques reset all the counters at the same time, every t_{REFW}. This has some direct consequence for the design of countermeasures.

As most counters are not reset synchronously with the refresh of the row, aggressor could use this information to target rows whose counters will be reset in the middle of the attack while the row has not been refreshed, resulting in an attack not witnessed by the countermeasure. Figure 5.2 illustrates an attack on a row that is not refreshed synchronously with the counters reset. In this figure, the refreshes on the considered row ② are not synchronised with the reset of the counters ①. Therefore, when the counters are reset, the actual ACT count of the row is higher than the value of the counter. If an attack uses this row as an aggressor, when the actual ACT count of the

row exceeds HC_{first} ③, the value of the counter is below the actual count ④, as it was reset during the attack. In this case, the attack will not be detected.

Existing mechanisms use multiple methods to take this fact into account. For example, BlockHammer [48] chooses to double the whole mechanism and alternately reset them, and Graphene [51] divides the detection threshold by 2, consequently using twice the initial number of counters.

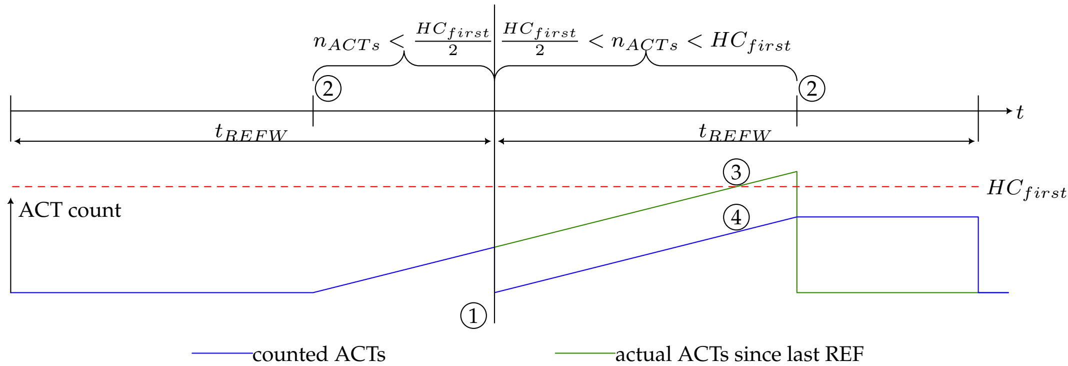


Figure 5.2: *Counters reset not synchronised with row refresh. Every t_{REFW}, all counters are reset ①. However, most rows are not refreshed synchronously with the reset of the counters ②. If an unsynchronised row is attacked, its counter will be reset in the middle of the attack. When the attack will have performed enough ACTs on the aggressor rows to produce a bit-flip ③, the counted ACTs will not reach the detection threshold ④.*

Modern mitigation proposals are greatly affected by this unsynchronised refreshes issue, which forces them to double the number of counters.

We propose here a new counter-based detection mechanism that does not need a periodic reset of all its counters, and therefore is not affected by the unsynchronised refreshes.

5.3.2 Tracking frequently-activated rows

The counter-based detection mechanism proposed here is based on the combination of ACT counters and ACT frequency evaluation. To simplify the demonstration, we assume that 2 ACTs will always take the same time to be executed, even if a periodic REF command is issued in-between, *i.e.*, we will only consider the time during which the memory can be issued ACTs. The time will be considered *paused* when the memory is not accessible because of, *e.g.*, periodic REF commands (that are issued every t_{REFI}).

In a bank, a total of W ACTs can be issued during t_{REFW}. Considering a corruption threshold T_{RH}, as the two neighbours of a victim rows can be used to corrupt it, the required number of ACTs per row to induce a bit-flip is

$$HC_{first} = \frac{T_{RH}}{2}. \tag{5.1}$$

We can deduce that a total of

$$N_{agg} = \frac{W}{HC_{first}} \tag{5.2}$$

aggressors can be used at the same time. Therefore, the mean period between two ACTs on an aggressor cannot exceed $P = N_{agg} \times t_{RC}$ for a successful attack. In other words, if the mean ACT period of a row is greater than P, it is not an aggressor as it cannot complete the attack within a refresh window (t_{REFW}).

In order to track only the potential aggressor, we can track only the rows that are activated frequently enough to be aggressors. The tracked rows will be stored in a table. The table entries are constituted of a key-value pair, where the key is the row id, and the value is the expiration time. To track only frequently-activated rows, we can do the following steps:

1. When a row is issued an ACT for the first time, allocate an entry in the table for it. The expiration time is set to $t + P$, where t is the current time.

2. If the row is activated again before the expiration time is reached, increase the expiration time by P. The expiration time will be set to $t_0 + n \times P$, where t_0 is the time of the first ACT, and n is the number of times it has been activated since the step 1.

3. When the expiration time is reached, the entry can be removed from the table. Subsequent ACTs on this row will start again on step 1.

Using this method, only the rows that are activated at least once every P on average will be kept in the table. However, monitoring all expiration dates of the table to check for expired values would be too time-consuming. Instead of having a constant monitoring of expiration dates, new entries will first try to replace expired entries. Ad-

ditionally, a periodic maintenance is put in place to periodically remove expired entries that were not replaced by new entries. This maintenance is discussed in Section 5.3.5.

This technique can track rows which are activated frequently-enough to be part of an attack. The evolution of one entry in the table is illustrated in Figure 5.3. In this figure, the black line represents the remaining time before the entry is forgotten by the table. ACTs are represented with red circles, and periods when the row is not tracked by the mechanism are indicated by grey rectangles.

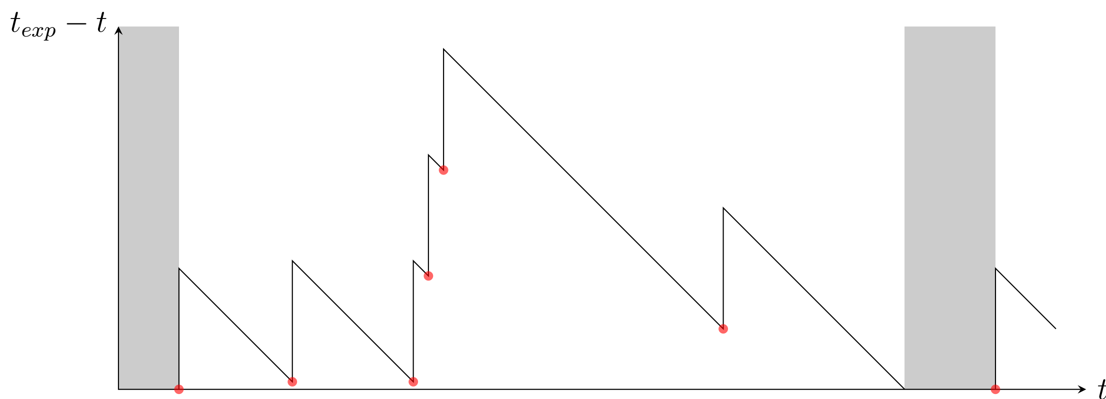


Figure 5.3: *Evolution of the entry for one row in the table. The vertical axis is the remaining time before the entry is removed. Red circles indicate ACTs. Grey rectangles represent moments when the row is not stored in the table.*

5.3.3 Detecting attacks

The mechanism is able to keep track of all the rows which are activated frequently-enough to be potential aggressors.

In addition to the expiration time, a counter is added to the table entry to keep count of the number of activations that have been issued to the row since it was added to the table. This counter is initialised at 1 when the entry is allocated, and is incremented when the row is issued an ACT. When the entry is forgotten then re-used, the counter restarts at 1. The evolution of the counter's value along with the time until expiration is illustrated in Figure 5.4. Simple attacks could be detected using only this counter: when the counter approaches the detection threshold ($\approx HC_{first}$) we can consider that the row is an aggressor and prevent the corruption by any mean, such as refreshing its neighbours, or communicating with the processor to stop the incriminated process.

Once a row has been considered an aggressor, its entry can be cleared to free it. This works for Rowhammer attacks which issue ACTs to their aggressor rows regularly, keeping the entry in the table, incrementing the counter for every ACT until it reaches the detection threshold.

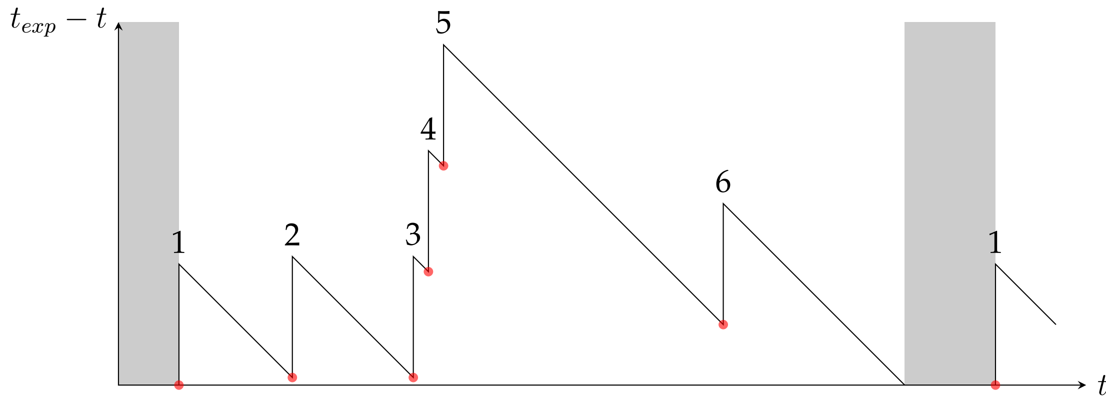


Figure 5.4: *Evolution of the entry for one row in the table. The vertical axis is the remaining time before the entry is removed. Red circles indicate ACTs. Grey rectangles represent moments when the row is not stored in the table. Displayed numbers are the values of the counter, incremented for every ACT*

However, two issues appear.

First, an aggressor can block an entry (*i.e.* keeping it used, therefore not available for new entries) for a long time by issuing a large number of ACTs to it, without reaching the detection threshold. The aggressor can then take advantage of the time the first entry is blocked to block a second entry, and so on, ultimately requiring a very high number of entries to avoid having all entries blocked by the aggressor.

To limit the consequences of this issue, we can change how we evaluate the rows as aggressors. Instead of relying only on the counter value, we can integrate the activation frequency into the formula. Considering c the counter value, t_{exp} the expiration time of the entry, t the current time and P the maximum delay between ACTs of aggressor rows, the potential of the row as an aggressor for a Rowhammer attack v_{RH} is calculated using the formula

$$v_{RH} = c \times \frac{t_{exp} - t}{P}. \tag{5.3}$$

The evolution of this value is illustrated in Figure 5.5. The threshold for v_{RH} at which we consider the row as an aggressor is $v_{RH} = HC_{first}$. Therefore, its value is only relevant when calculated at the moment an ACT is issued, at its highest. If an aggres-

sor row is issued ACTs at a regular interval $\delta t \in [2t_{RC}; P[$, the value of v_{RH} can be calculated, after an ACT, with the formula

$$v_{RH} = c\frac{t_0 + cP - (t_0 + (c-1)\,\delta t)}{P} = c\frac{c\,(P - \delta t) + \delta t}{P}, \tag{5.4}$$

where t_0 is the time of the first ACT issued to the row. If the attack issues ACTs to the row as slow as possible ($\delta t \rightarrow P$), v_{RH} reaches HC_{first} for $c = c_{max} \approx HC_{first}$. In contrast, if the attack issues ACTs as quickly as possible ($\delta t \rightarrow 0$), v_{RH} reaches the threshold HC_{first} for $c = c_{min} \approx \sqrt{HC_{first}}$. The exact value of c_{min} can be calculated as the lowest integer value of c that satisfies the inequality $v_{RH} \geq HC_{first}$, with $\delta t = \delta t_{min} = 2 \times t_{RC}$.

Using this activation frequency evaluation limits the capability of aggressors to block table entries by rapidly activating rows without attacking them, as these rows will be rapidly detected as aggressor and removed from the table if they reach c_{min} ACTs. The number of entries required for a proper behaviour of this mechanism is calculated in Section 5.3.7.

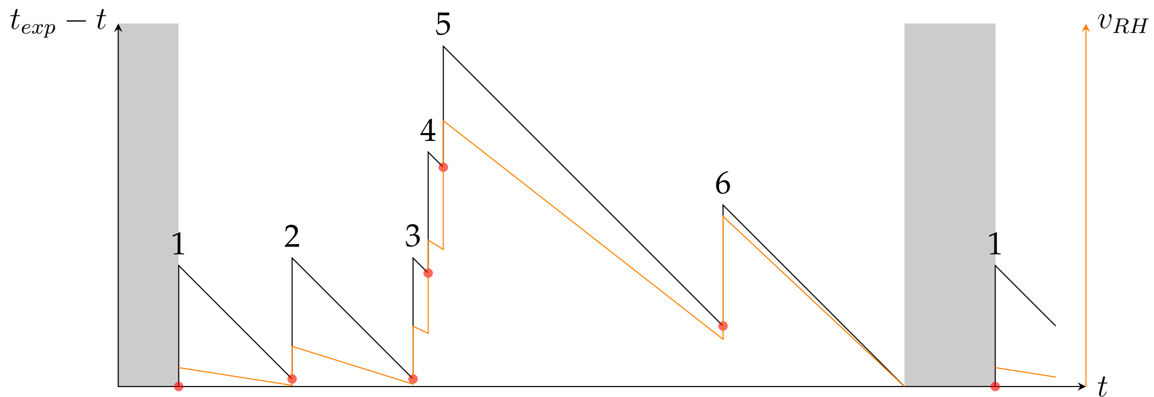


Figure 5.5: *Evolution of the entry for one row in the table. The vertical axis for the black line is the remaining time before the entry is removed. The vertical axis of the orange line is the evaluation of v_{RH}. Red circles indicate ACTs. Grey rectangles represent moments when the row is not stored in the table. Displayed numbers are the values of the counter, incremented for every ACT.*

The second issue that can appear is split attacks, illustrated in Figure 5.6. As depicted, the aggressor would typically perform slow activations until the value is close to the limit, wait for the entry to disappear from the table, and then issue the remaining ACTs for the attack within the remaining time of the refresh cycle t_{REFW} to complete the attack, without being noticed. The aggressor must ensure that there is enough time

between the disappearance of the entry from the table and the end of the refresh cycle to issue all the remaining ACTs to complete the attack.

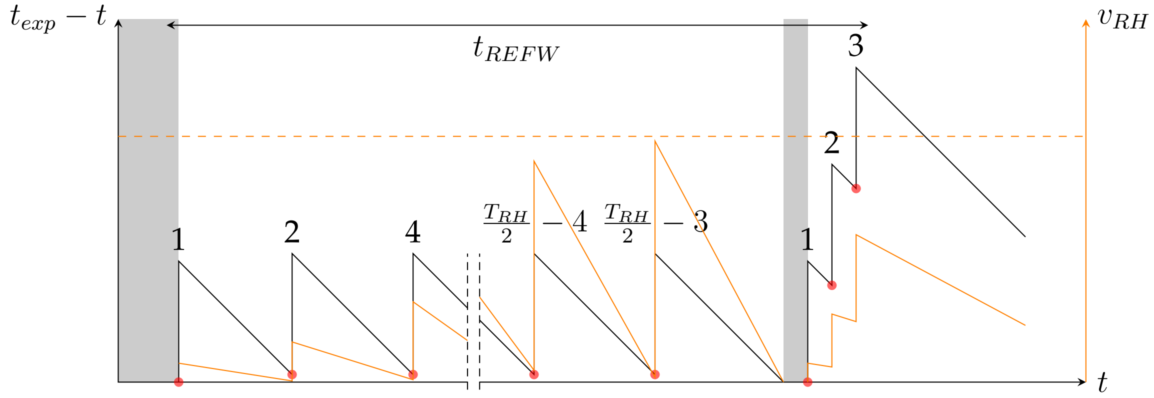


Figure 5.6: *Split attack on F-CoRD. The attack issues $T_{RH} \div 2$ ACTs within t_{REFW}, but v_{RH} never reaches the threshold (orange dashed line). Therefore, the attack is never acknowledged by the mechanism.*

To counteract this issue, P can be increased and take advantage of the way v_{RH} is calculated. The more ACTs are performed in the first part of the attack, the less time the aggressor has in the second part. Considering that making one more ACT in the first part will take the time of multiple ACTs in the second part, the first part must not issue too many ACTs to let the second part enough time to perform the remaining ACTs. To avoid detection on the second part of the attack, it must consists in less than c_{min} ACTs. To protect against split attacks, the value of P must be set so that after completing $HC_{first} - c_{min} + 1$ ACTs for the first part and waiting for the entry to be removed, the remaining time is not sufficient to issue the remaining $c_{min} - 1$ ACTs. The new value of P must satisfy the inequality

$$P\left(HC_{first} - c_{min} + 1\right) + \delta t_{min}\left(c_{min} - 2\right) \geq t_{REFW}. \tag{5.5}$$

Consequently, it can be calculated using the formula

$$P = \frac{t_{REFW} - \delta t_{min} \times \left(c_{min} - 2\right)}{HC_{first} - c_{min} + 1}. \tag{5.6}$$

As a reminder, the value of c_{min} is the lowest counter value for which being calculated as the lowest value for which $v_{RH} \geq HC_{first}$. The values of c_{min} and P being dependent

on each other, they can be determined iteratively, first calculating P with

$$P_0 = N_{agg} \times t_{RC} = \frac{W \times t_{RC}}{HC_{first}} \tag{5.7}$$

, and then iteratively calculating c_{min} and P until the value of c_{min} stabilises.

5.3.4 Discretisation

For the implementation into counters, the expiration time and therefore the period must be turned into discrete values. To discretise the time-related variables P and δt, we introduce t_{ACT} the time (in ticks) per ACT. For example, a t_{ACT} of 0.25 means that the tick is configured so that the memory can be issued 4 ACTs on the same tick. δt_{min} is now written $\delta_{min} \times t_{ACT}$, where δ_{min} is the maximum number of rows that can be used simultaneously without slowing down a fast attack. The value of t_{ACT} will determine the precision of the timer. When decreasing t_{ACT}, the timer precision is lowered, resulting in a lower entry size (less bits needed to store t_{exp}) and an easier timing management. However, it can also results in more entries. The value of t_{ACT} can be optimised to select the best compromise between entry size and number of entries, depending on how the table will be implemented.

For a bank-level counting granularity, $\delta_{min} = 2$: 2 rows are used alternately to avoid row hits. At rank level, multiple banks can be used at the same time. When one bank processes an ACT, rows in other banks can be issued ACTs in parallel, leading to a bigger value for δ_{min}. As P must satisfy the inequality 5.5, the formula to calculate the first value and the iterative formula become

$$P_0 = \frac{W \times t_{ACT}}{HC_{first}}, P = \left\lceil t_{ACT} \times \frac{W - \delta_{min} \times (c_{min} - 2)}{HC_{first} - c_{min} + 1} \right\rceil. \tag{5.8}$$

When using a discrete value for P, a new issue appears. As the timer may not be synchronised with the ACTs, new entries may get forgotten before they actually reach the initial (non-discrete) P time. To avoid this issue, new entries will have their t_{exp} set to $t + P + 1$ instead of $t + P$ (where t is the current tick) to ensure that all entries stay at least the correct amount of time in the table. As a consequence, if t_0 is the tick of the

first ACT of the row, then $t_{exp} = t_0 + c \times P + 1$.

c_{min} is calculated by taking the smallest integer that satisfies the inequality

$$c_{min} \times \frac{c_{min} \times (P - \delta_{min} \times t_{ACT}) + \delta_{min} \times t_{ACT} + 1}{P} \geq HC_{first}. \tag{5.9}$$

An illustration of a slow attack with discrete timings is presented in Figure 5.7. In this

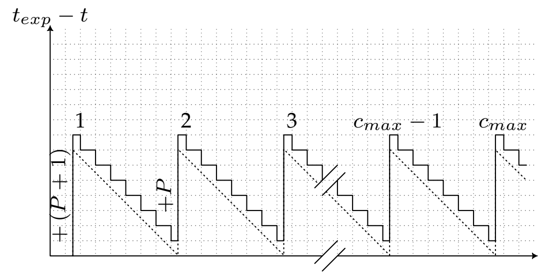


Figure 5.7: *Slow attack, with discrete timings. the dashed line represents the value if the timings were not discrete. The threshold HC_{first} for v_{RH} is reached for a maximum of c_{max} ACTs.*

attack, the attacker issues ACTs to the row as slowly as possible to trigger a bit-flip. Starting from the second one, an ACT on the aggressor row ideally happens as late as possible, *i.e.* when $t_{exp} - t = 1$. This means that after the ACT, we have $v_{RH} = c\frac{P+1}{P}$. Consequently, slow attacks are detected when

$$v_{RH} \geq HC_{first} \iff c\frac{P+1}{P} \geq HC_{first} \iff c = c_{max} = \left\lceil \frac{HC_{first}P}{P+1} \right\rceil. \tag{5.10}$$

5.3.5 Periodic maintenance

When an entry expires, it is not immediately deleted from the table. Doing so would require constant surveillance of all active entries to check when they expire. This would at least add a significant energy overhead to compare the t_{exp} of each entry with the current time. Therefore, the mechanism does not delete expired entries immediately but wait for them to be replaced by newer entries. However, due to the implementation of the timer as a fixed-size counter, it resets itself to 0 after reaching its maximum value $t_{max} - 1$, periodically. If an expired entry is not replaced when the timer is reset, it will be falsely considered active. It will not be replaced, potentially permanently blocking the entry. When the row is activated again, the evaluation of v_{RH} may consider this

row as an aggressor, creating false positives.

To avoid such issues, expired entries must be regularly deleted from the table. To periodically delete expired entries, the mechanism must be able to tell if the expiration time of an entry is plausible at the current time. That is, if $t_{exp} < t$, can t_{exp} have passed the capacity of its bits, or has it expired and it needs to be deleted ? On the contrary, if $t_{exp} > t$, has the row been recently been activated and still needs to be watched, or has t been reset since the last activation of the row, meaning that the entry has expired before t was reset ? To determine if an entry needs to be deleted, the mechanism must be able to distinguish if an expiration time is plausible for an active entry at the current time. The maximum delay until the expiration of an entry $(t_{exp} - t)_{max}$ occurs in the case of a fast attack, when a row is activated repeatedly and as fast as possible. Its value can be calculated using formula

$$(t_{exp} - t)_{max} = (c_{min} - 1) \times (P - \delta_{min} t_{ACT}) + \delta_{min} t_{ACT} + 1. \tag{5.11}$$

Telling if at t, a value t_{exp} is plausible for an active entry simply translates to verifying $t_{exp} - t \mod t_{max} < (t_{exp} - t)_{max}$. If this condition is not verified, the entry is expired and can be removed. Allowing such verification to be performed means that the timer must be able to hold values above $(t_{exp} - t)_{max}$. To make the implementation simpler, this verification is done for fixed values of t, every $t_{cycle} = t_{max} \div n_{cycles}$. As the timer is dimensioned to allow this verification to be made, the value of t_{max} can be calculated as the lowest power of 2 that satisfies the inequality

$$t_{max} \geq (t_{exp} - t)_{max} \times \frac{n_{cycles}}{n_{cycles} - 1}. \tag{5.12}$$

As there is no way for an active entry to have an expiration time n_{cycles} cycles in the future, all entries which have an expiration time in the previous cycle are expired entries. Hence, this verification can be done simply by deleting entries that have expired during the previous cycle. If n_{cycles} is a power of two, only a few bits ($log_2\left(n_{cycles}\right)$) of the expiration time need to be checked to determine if t_{exp} happened in the previous cycle. Having a higher n_{cycles} can result in a lower entry size, at the extra cost of having to check the entries more frequently.

If this mechanism is implemented for DDR3 or DDR4 memory, or for DDR5 memory with bank-level counting granularity, the periodic refresh periods can be used to delete expired entries. In DDR5, all the banks are not necessarily refreshed at the same time. In this case, the mechanism does not have a pre-defined period that allows the mechanism to perform maintenance operations, and will have to perform them while running.

5.3.6 Implementation details

The mechanism implements the table with two content-addressable memories and one array. The first CAM CAM_R holds row IDs, the second one CAM_T expiration times, and the array CNT holds counter values. The reason to use CAMs for row IDs and expiration times is because both need to be searched when an ACT is issued. CAM_R is used to search if the activated row already has an entry, and CAM_T is used to search for entries to replace or delete. The counter values in CNT, however, are only used to calculate v_{RH}. An entry in the detection mechanism shares the same index across all memories.

From the algorithm point of view, The following functions are implemented in the table :

- CAM_R implements the functions `search`(*row*), `set`(*index*, *row*).
- CAM_T implements the functions `searchExpired`(*t*), `searchAvailable`(*t*), `get`(*index*), `set`(*index*, *t*).
- An additional function `delete`(*index*) that removes an entry from all three components of the table.

Algorithm 1 presents how the mechanism works. In this algorithm, t_{max} is the maximum value attainable by t, $t_{cycle} = t_{max} \div n_{cycles}$ is the duration of maintenance cycles, *i.e.*, the number of ticks between every check of the memory to delete expired entries. The function `init` sets the initial value of t at startup. Function `onACT` is

called whenever an ACT is issued to the considered memory, and function `timer` is called every tick to increment t.

Algorithm 1: F-CoRD global algorithm

```
1  Function init:
2      t ← 0
3  Function isRowhammer(c, t_exp):
4      v_RH ← c × (t_exp − t) / P
5      return v_RH > T_RH
6  Function onACT(row):
7      index ← CAM_R.search(row)
8      if index ≥ 0 then
9          t_exp ← CAM_T.get(index)
10         c ← CNT[index]
11         if t_exp < t then
12             t_exp ← t + P + 1
13             c ← 1
14         else
15             t_exp ← t_exp + P
16             c ← c + 1
17             if c ≥ c_min and isRowhammer(c, t_exp) then
18                 detected(row)
19                 delete(index)
20             else
21                 CAM_T.set(index, t_exp)
22                 CNT[index] ← c
23     else
24         index ← CAM_T.searchAvailable(t)
25         CAM_R.set(index, row)
26         CAM_T.set(index, t + P + 1)
27         CNT[index] ← 1
28 Function timer:
29     t ← t + 1 mod t_max
30     if t mod t_cycle = 0 then
31         foreach index in CAM_T.searchExpired(t) do
32             delete(index)
```

5.3.7 Number of entries

There is a maximum of $\frac{1}{t_{ACT}}$ ACT(s) at every tick, increasing the t_{exp} by P for the concerned rows. In the same time, the value of $t_{exp} - t$ of every entry decreases by 1. Therefore, at every tick, the sum of all $t_{exp} - t$ is reduced by $n_{entries}$ and can increase by $\frac{P}{t_{ACT}}$. When $n_{entries} < \frac{P}{t_{ACT}}$, the sum of all $t_{exp} - t$ can increase faster than it decreases. Inversely, when $n_{entries} > \frac{P}{t_{ACT}}$, the sum of all $t_{exp} - t$ can only decrease. Consequently, for a regular memory-intensive application that uses the cache to avoid repeated ACTs on a few rows, the number of entries will regulate itself around $\frac{P}{t_{ACT}}$. However, an attacker could force the mechanism to use more and more entries by temporarily blocking some rows. Calculating the minimum number of entries can be done by simulating an attack that will try to use as many entries as possible. The algorithm to simulate such attack is written in Algorithm 2.

Algorithm 2: Calculate the required number of entries of F-CoRD

```
1  read t_ACT
2  read W
3  read δ_min
4  read HC_first
5  P ← (W × t_ACT) / HC_first
6  repeat
7      c_min ← findCmin(P, δ_min)
8      P ← ⌈t_ACT × (W − δ_min × (c_min − 2)) / (HC_first − c_min + 1)⌉
9  until c_min is fixed
10 n_entries ← δ_min
11 t_stop = (c_min − 1) × (P − δ_min × t_ACT) + 1
12 while t_stop > P + 1 do
13     n_ACT ← ⌈(t_stop − δ_min × t_ACT − 1) / P⌉
14     t_stop ← min(n_ACT × (P − δ_min × t_ACT) + 1, t_stop − n_ACT × δ_min × t_ACT)
15     n_entries ← n_entries + δ_min
16 n_entries ← n_entries + ⌈t_stop / t_ACT⌉
```

The algorithm itself is divided into multiple segments. The first segment, from line 5 to line 9, is used to iteratively determine the values of P and c_{min}. The function `findCmin` finds the value of c_{min} using Equation 5.9. The iteration stops when the values stabilise. This should happen very quickly, as both values only slightly vary with the value of the other, and they are both rounded up to integer values. The next

segment, on lines 10 and 11, simulates the first δ_{min} ACTs, establishing the first value for the time until earliest expiration t_{stop}, and initialising the number of entries $n_{entries}$. The third segment, from line 12 to line 16, is the core of the algorithm. Rows are selected by groups of δ_{min}, and issued enough ACTs to have the minimum influence t_{stop}. After that, t_{stop} is recalculated as the time until earliest expiration. If rows were activated more than what is needed to not influence t_{stop}, earliest rows would get forgotten by the mechanism before this one and less time will be available to activate other rows, leading to less entries being used. When $t_{stop} \leq P + 1$, the loop is not needed anymore, as $t_{stop} \div t_{ACT}$ rows can be issued one ACT each without influencing t_{stop}.

Once the earliest expiration time t_{stop} is met, any more ACT will only replace entries which get gradually forgotten. As explained earlier, if there is more than P entries in the table, the sum of all $t_{exp} - t$ of the entries in the table can only decrease, leading to the forgetting of entries.

The final value of $n_{entries}$ in this algorithm determines the number of entries of the table that are required to avoid the case of a row not having a room in the table when issued an ACT.

However, as previously stated, the number of entries used will rarely go over $P \div t_{ACT}$ during normal operation. To reduce the energy consumption, the mechanism could implement the necessary circuitry to disable large portions of the table when they are not used.

5.3.8 Example

Considering a typical DDR4 memory, using the timing parameters listed in Table 2.1 page 12, and $HC_{first} = 16384$. For a bank-level implementation, the relevant timing parameters are as follows: $W = 1.33 \times 10^6$, $\delta_{min} = 2$.

For simplicity, we will select $t_{ACT} = 1$, *i.e.*, one tick per ACT. Executing Algorithm 2 results in $P = 83$, $c_{min} = 130$ and $n_{entries} = 447$. Note that in this configuration, the values of P and c_{min} are small enough to be determined with only one iteration. The maximum value that can be reached by the counter is $c_{max} = 16189$.

DDR gen.	HC_{first}	t_{REFW}	W	optimal t_{ACT}	number of counters	CAM_R + CAM_T size	CNT array size
DDR3	69.2K	64ms	1.25M	1	114	3.34Kib	1.89Kib
DDR4	16K	64ms	1.33M	1/12	465	12.3Kib	6.36Kib
DDR5	4.8K	32ms	661K	1/28	701	17.8Kib	8.21Kib

Table 5.2: *minimum table size for the F-CoRD implementationon DDR3, DDR4 and DDR5, for a bank-level implementation counting granularity.*

With the values of P and c_{min} , we can calculate $(t_{exp} - t)_{max} = 10452$, and therefore the size of the timer for different values of n_{cycles}. For $n_{cycles} = 2$, the timer must be able to hold $2 \times (t_{exp} - t)_{max} = 20904$, which requires at least 15 bits. For $n_{cycles} = 3$, the timer would only have to hold up to $1.5 \times (t_{exp} - t)_{max} = 15678$, which requires 14 bits.

Considering a typical DDR4 with $2^{16} = 65536$ rows, for $n_{cycles} = 2$, each table entry will consists in 16(row id)$+15$(expiry time)$+14$(ACT count) $= 45$bits. The table having $n_{entries} = 447$ entries, the total size of the table will be $447 \times 45 = 19.6Kib$ (including both the CAMs and the CNT array).

Table 5.2 lists the size of F-CoRD for DDR3, DDR4 and DDR5. The timing parameters used to generate this table are as listed in Table 2.1 page 12. The value of HC_{first} for the DDR3 is the first considered value for this DDR generation [15]. As the value of HC_{first} for DDR5 has not been determined yet, we selected for this generation the lowest registered value for the LPDDR4 [15].

For comparison, the bank-level implementation of Graphene for the considered DDR4 would use respectively a CAM of 4.8Kib, the BlockHammer for the same configuration would use a 26Kib array of counters.

5.3.9 Conclusion

This Rowhammer detection mechanism provides a solution to detect aggressors with great accuracy. Contrary to other countermeasure proposals, this solution does not suffer from the unsynchronised refresh of all rows in the memory. However, the important additional memory it requires makes it less interesting compared to existing stat-of-the-art detection mechanisms. Nonetheless, as the quantity of entries it uses

at any time depends on the access pattern and frequency of the memory, the number of entries used for a regular application is far below the entry usage for other counter-based proposals. As such, deactivating the unused entries could lead to a lower energy consumption than existing State-of-the-art detection mechanisms.

VI

Experiments on MRAM

Contents

6.1 Motivation **78**

6.2 Attack on a Toggle-MRAM chip **82**

6.2.1 Platform Requirements 82

6.2.2 Test Platform 83

6.2.3 Reverse-engineering of the memory module architecture . . . 86

6.2.4 Designing the attack 88

6.2.5 Results 90

6.3 Attack on an STT-MRAM chip **91**

6.3.1 Test Platform 91

6.3.2 Reverse-engineering of the memory module architecture . . . 92

6.3.3 Designing the attack 94

6.3.4 Results 94

6.4 Conclusion **95**

6.1 Motivation

To this day, the Rowhammer attack is still an important issue for modern systems that use DRAM memories to store run-time memories. Numerous countermeasures have been proposed, to prevent aggressors from taking advantage of cell-to-cell disturbance to perform privilege escalation, retrieve sensitive information or crash victim systems. However, these countermeasures only work on DRAM-based memory. In the past few years, multiple non-volatile memory (NVM) technologies have been proposed as replacements for the DRAM technology. These memories use the physical properties of materials to store the data. Among current NVM propositions, 4 technologies stand out.

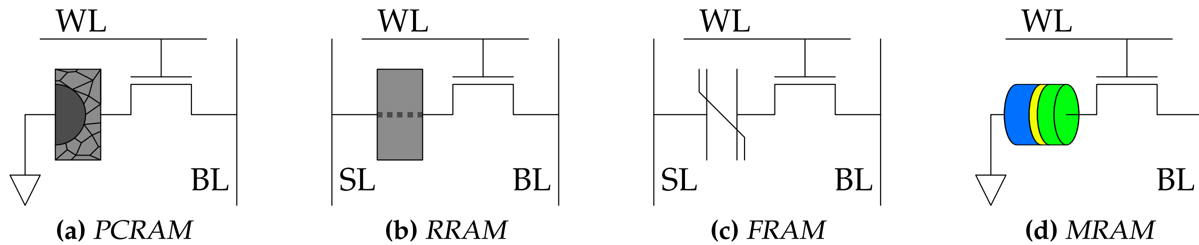
Phase-Change RAM (**PCRAM**, Figure 6.1a) uses the difference of resistance between the crystalline and amorphous states of a material to store the value of the bit. The state can be changed by controlling the cooling speed of the material after heating it.

Resistive RAM (**RRAM**, Figure 6.1b) creates or breakes a conductive filament in a dielectric material by applying a current on it. The presence of a filament greatly lowers the resistance of the material, which determines the value of the bit.

In **Ferroelectric RAM** (**FRAM**, Figure 6.1c), a ferroelectric crystal replaces the dielectric material in the storage capacitor to create a ferroelectric capacitor. The polarity of this capacitor determines the value of the bit, and can be changed by applying a current on it.

Magnetoresistive RAM (**MRAM**, Figure 6.1d) uses a Magnetic Tunnel Junction (MTJ) to store data. An MTJ is composed of two ferromagnetic layers: a pinned layer (PL, or reference layer) with a fixed magnetic orientation, and a free layer (FL) whose magnetic orientation can change. These two layers are separated by a thin insulating layer. Depending on the orientation of the free layer relative to the reference layer (either parallel or anti-parallel), the resistance of the material changes, resulting in two differentiable states to code one bit of data. There is multiple methods to change the magnetic orientation for the free layer, that lead to as many different technologies.

This last technology, especially the STT-MRAM variant, is considered the most promising non-volatile replacement for the DRAM technology [99].



(a) *PCRAM* **(b)** *RRAM* **(c)** *FRAM* **(d)** *MRAM*

Figure 6.1: *Bit-cells of Emerging Non-Volatile-Memories (NVM). A transistor controlled by the wordline (WL) connects the bitline (BL) to the storage element when activated. On the other end of the storage element is either the ground or a sourceline (SL).*

These technologies offer speed and endurance comparable to that of the DRAM technology, but are non volatile. We will focus here on the MRAM technology, and more specifically on the Spin-Transfer-Torque MRAM (STT-MRAM) [100] and Toggle-MRAM [101] variants, which have interesting characteristics among the NVM memories, and are commercially available as stand-alone chips.

The STT-MRAM switching mechanism is illustrated in Figure 6.2. It uses the magnetic torque induced by the spin of electrons that are let through or blocked by the tunnel junction to change the polarity of the FL. Electrons whose spin are not compatible with the PL cannot pass through it. If the electron flow passes first through the PL, only electrons with the same orientation as the PL are let through the FL. On the contrary, if the electron flow passes first through the FL, the rejected electrons will stay in the FL. In the FL, the electrons that do not match the orientation of the material generate a torque capable of inverting it.

The Toggle-MRAM switching mechanism is illustrated in Figure 6.3. The Free Layer of the Toggle-MRAM is made of two sub-layers (FL1 and FL2) with opposite magnetic orientation. Above and below the MTJ are two perpendicular Write Lines (WL1 and WL2) used to generate perpendicular magnetic fields intersecting at the MTJ. Activating one write line after the other generates a rotating magnetic field that inverts the magnetic orientations of FL1 and FL2.

MRAM are known to be very resistant to external disturbance, as each memory cell is s bi-stable mechanism that requires a significant energy input to switch state because

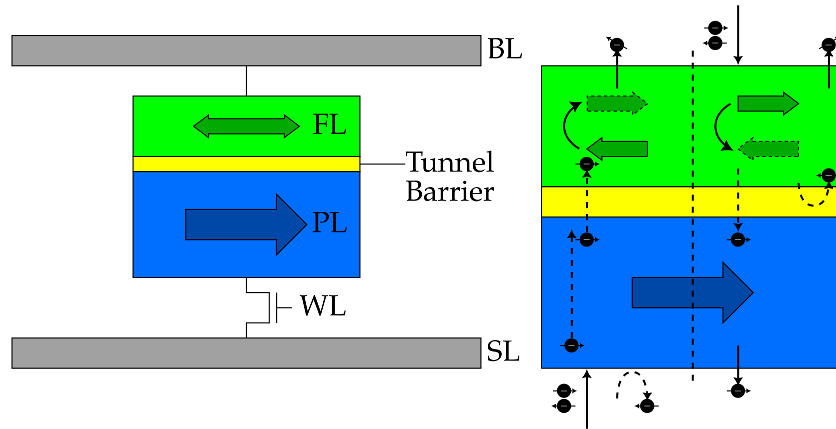


Figure 6.2: *STT-MRAM structure (left) and write process (right). Electrons inside the FL generate a magnetic torque that changes the orientation of the FL.*

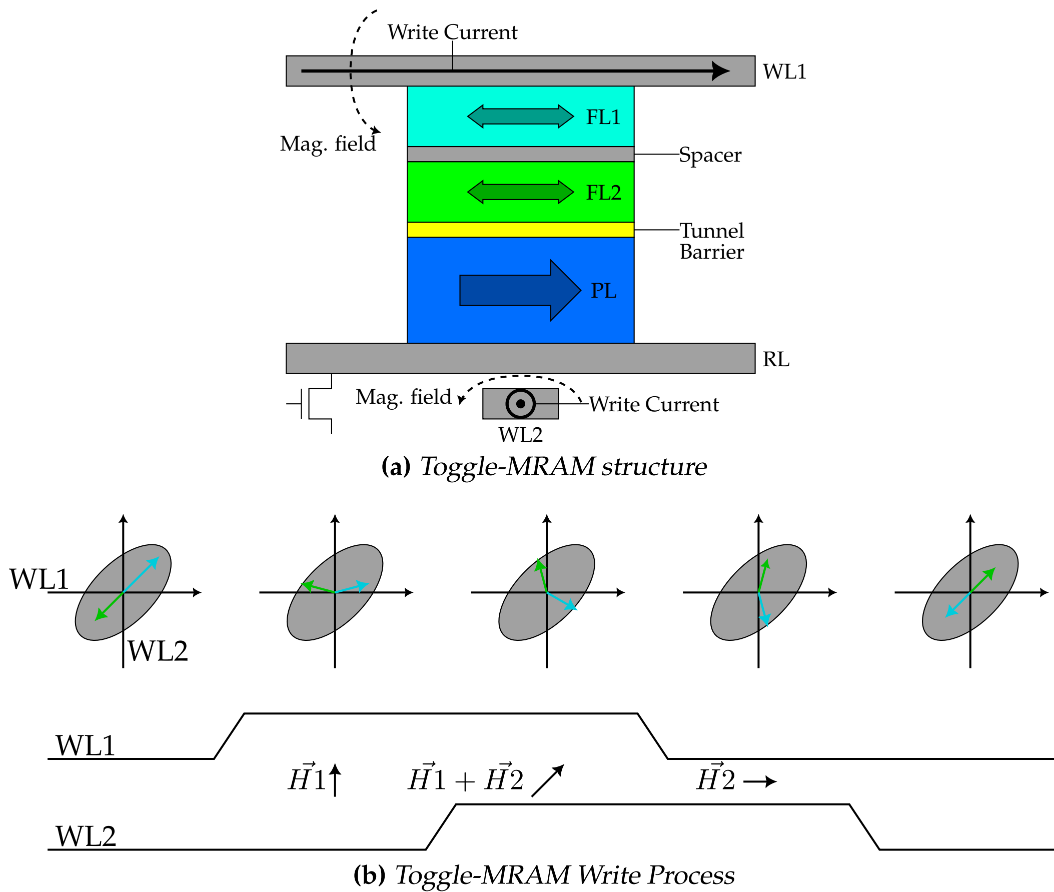


Figure 6.3: *Toggle-MRAM structure (a) and write process (b). Two perpendicular Write Lines (WL1 and WL2) are used to generate a rotating magnetic field that inverts the orientation of FL1 and FL2. The rotating magnetic field generated by WL1 and WL2 is used to rotate the polarity of FL1 and FL2.*

of a thermal barrier between the two stable states.

A recent study [62] has shown that the magnetic field generated by MTJs can disturb nearby bit-cells if the distance between them is small enough. This principle is

illustrated in Figure 6.4. According to this study, if the nearest neighbours ① (A) and next-nearest neighbours ② (B) of a bit-cell all have their MTJ in the Parallel State, the induced magnetic field experienced by the central MTJ (O) can lower the energy required to switch it from Anti-Parallel to Parallel. Reducing the spacing between cells increases the induced magnetic field, and a higher magnetic field results in a reduced energy to switch the state. Considering MTJ of 55nm diameter, a bit spacing (BS) of 200nm incurs a negligible increase of the Bit Error Rate (BER) compared to an ilosated MTJ; a BS of 150nm incurs an increase of the BER by less than one order or magnitude; and a BS of 100nm incurs an increase of the BER by 3 orders of magnitude.

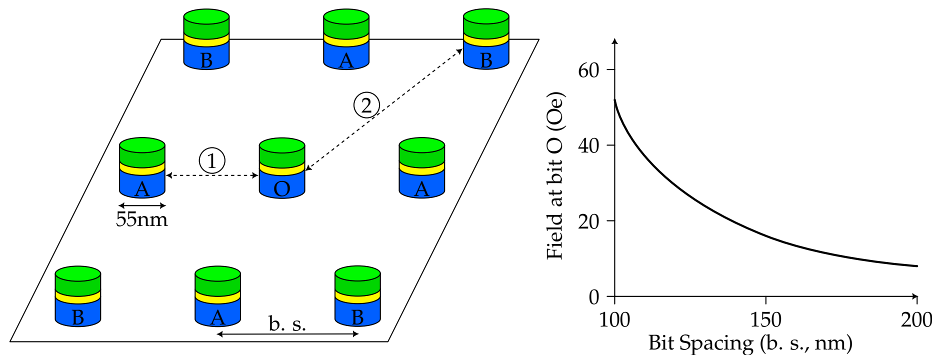


Figure 6.4: *Cell-to-cell magnetic disturbance on MRAM [62]. the nearest neighbours ① A, and next-nearest neighbours ② B have their MTJ in the Parallel State, and the MTJ of O is in the Anti-Parallel state to maximise the magnetic field experienced by O.*

In addition, an other study [63] demonstrated that due to the high current necessary to change the state of MTJs and to how STT-MRAM banks are designed, if an adversary keeps writing to a particular address, it would generate a ground bounce that propagates to the wordlines drivers. The access transistors of the unselected bits that share the same bit-line and source-line with the accessed cells will partially activate and incur a disturb current. The affected cells will experience retention failures. The principle described by the author is illustrated in Figure 6.5.

However, these studies are purely theoretical. To the best of our knowledge, no experiment has been done on a physical memory to confirm the flaws described by these studies.

In this chapter, we characterise the vulnerability of two commercial MRAM memory modules against variants of the Rowhammer attack, using an FPGA-based platform and a microcontroller-based platform.

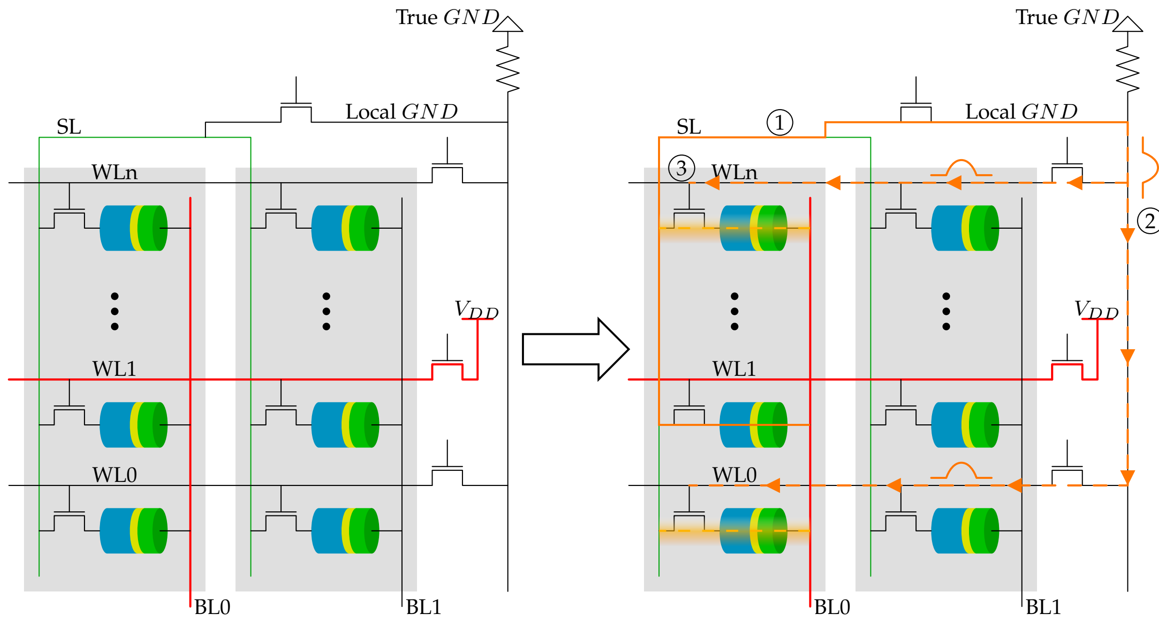


Figure 6.5: *Rowhammer effect on STT-MRAM [63]. When switching a bit-cell to Anti-Parallel state, the WL and BL of the selected cell are put to V_{DD} (left). The current used to change the state of the MTJ then navigates to the true ground of the memory through the local ground of the bank ① (right). Due to the resistance of the material layers between the local GND and the true GND, the current on the local ground generates a ground bounce, that navigates to unselected WL ②. This bounce slightly opens unused access transistors which lets a small current flow from the active BL ③ through the MTJ, inducing retention loss.*

6.2 Attack on a Toggle-MRAM chip

6.2.1 Platform Requirements

The first tested memory module is a Toggle-MRAM module from Everspin [102]. This module is a 16Mib memory, with 2^{20} words of 16 bits. It communicates using a parallel interface, with 20 pins for the address and 16 pins for the data in addition to control pins (*e.g.*, Write Enable, Output enable). Its architecture is illustrated in Figure 6.6.

Therefore, the platform must be compatible with the parallel interface of the memory, and must allow the program running on it to access the memory at its maximum allowed frequency. Additionally, MRAMs tend to be less reliable at higher temperature, accelerating the retention loss. In order to characterise the impact of the high temperature on memory-corruption attacks, the platform must be able to either control the temperature of the memory module, or work in a high temperature environment when put inside a temperature chamber. Measuring the current consumption while

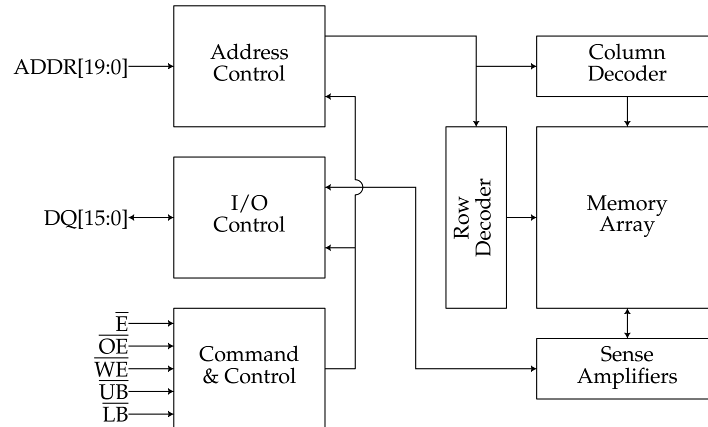


Figure 6.6: *Simplified architecture of the Toggle-MRAM memory (**ADDR**: address, **DQ**: data I/O, **E**: global Enable, **OE**: Output Enable, **WE**: Write Enable, **UB**: Upper Byte enable, **LB**: Lower Byte enable).*

performing some operations on the memory is also important, as variations of the current can be used to understand how the memory works. Finally, the platform must communicate bidirectionally with a computer to configure the tests and display the results using *e.g.* a serial port.

6.2.2 Test Platform

The first platform we worked on is based on an ARTY Z7 development platform [103]. This board includes a Zynq-7000 System-on-Chip (SoC) from Xilinx [104] which embeds a dual-core 650MHz ARM Cortex-A9 processor, together with a Xilinx 7-series Field-Programmable Gate array (FPGA).

The combination of a processor and an FPGA allows the creation of dedicated peripherals, and custom logic circuits to optimise memory accesses. The system architecture with the processor and the peripherals integrated in the FPGA is illustrated in Figure 6.7.

The processor integrates a dedicated port to communicate with the RAM, and communicates to its peripherals through the AXI bus ①. When integrating more peripherals into the FPGA, we use an AXI crossbar to connect one master to multiple slaves.

Accessing an external memory from the processor requires the usage of an External Memory Controller (EMC) peripheral to manage the address, data and control pins of

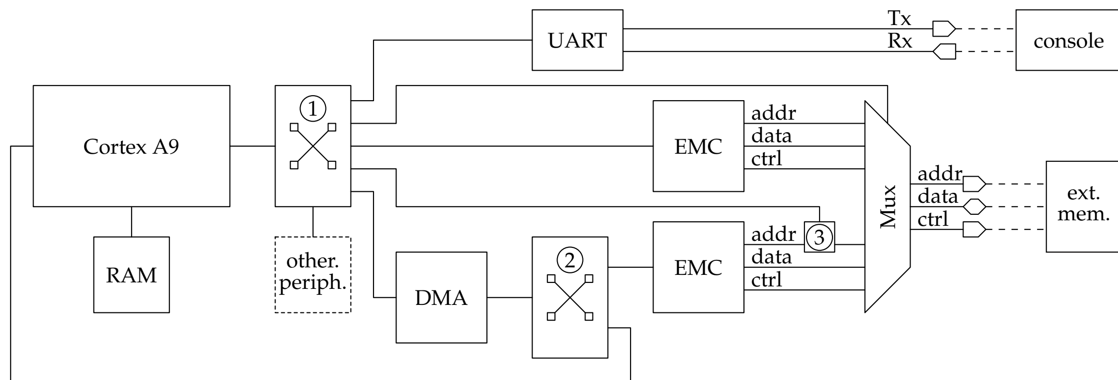


Figure 6.7: *FPGA-based platform peripherals architecture. Each AXI crossbar ① ② is used to connect multiple slaves to a single master. The memory address matrix ③ modifies the address at the output of the EMC connected to the DMA to create repeated patterns from linearly increasing addresses.*

the external memory. This EMC must be configured when designing the logic circuit with the timings of the memory to access it at its maximum speed. After experimenting, we notices that the minimum delay the EMC can achieve between two consecutive accesses to the external memory is 50ns. While it is higher than the minimum 45ns listed in the datasheet of the memory, the difference is small enough to consider that this as a viable configuration.

Memory accesses from the processor can be slowed down by two factors. First, the processor is limited in the number of memory accesses it can perform with one instruction. After completing the series of memory accesses for one instruction, the processor must fetch and decode the next instruction from the memory. This operation takes some time during which the processor does not access the tested memory. Additionally, the program will use loop-related instructions (decrementing a counter and jumping to the beginning of the loop) to repeat the memory accesses a large number of times. These instructions will take some additional time for the processor to fetch and decode, during which it will not access the tested memory.

For these reasons, the platform integrates a Direct Memory Access (DMA) peripheral in the FPGA, that will perform a large number of memory accesses without the intervention of the processor. The processor will only intervene once at the end of each loop to restart it if necessary, limiting the impact of instructions fetching and decoding on the process. The DMA acts as a slave on the AXI bus connected to the processor ①, and as a master to the bus that connects it to the memories it works on. Therefore, it

has a dedicated AXI crossbar ② that connects it to a dedicated EMC and to the Cortex to communicate with the RAM.

However, the DMA is limited in the address patterns it can use when accessing the memory. The simple DMA that is integrated in the FPGA is only capable of performing read or write accesses on consecutive addresses of the memory. To characterise the sensitivity of emerging memories to corruption attacks, the platform must be able to perform repeated accesses on a single address or on a few addresses. To make the development of the platform easier, instead of modifying the DMA to allow such memory access patterns, we implement a matrix that multiples the address vector at the output of the EMC after the DMA and generates a new address vector for the address pins ③. This final vector can be calculated using the formula

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}_{pins} = \begin{bmatrix} m_{0,0} & m_{0,1} & \dots & m_{0,n} \\ m_{1,0} & m_{1,1} & \dots & m_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n-1,0} & m_{n-1,1} & \dots & m_{n-1,n} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \\ 1 \end{bmatrix}_{EMC} . \tag{6.1}$$

The matrix is only made of 1s and 0s, and the addition step of matrix multiplication is replaced by a bit-by-bit OR. An additional 1 is appended at the end of the input address vector for more control of the address modification. This address matrix is implemented as an independent peripheral editable by the processor.

Finally, connecting the memory only to the DMA would prevent the processor from accessing it directly. A direct access can be used to perform simple tasks such as looking for corruption or initialising the memory. Hence, a multiplexer peripheral is inserted to connect the EMC of the Processor and the EMC of the DMA (followed by the address matrix) to the MRAM pins. This allows the processor to freely switch between accesses from the DMA and accesses from the processor, and to configure the two EMCs with different timing parameters. The EMC of the DMA can be configured to optimise the attack speed of the DMA at the detriment of the viability of the written or read values. On the contrary, the EMC accessed by the processor can be configured with slower timing parameters to ensure the viability of the values written or read in

this configuration.

6.2.3 Reverse-engineering of the memory module architecture

The datasheet of the memory specifies a 20-bit address space, with 10 bit for the column address and 10 bits for the row address. However, they do not specify if the module uses an internal cache or row buffer, and the order of the row and column bits in the 20-bit address.

To efficiently perform attacks on a memory module, this knowledge is important to determine the attack pattern. The attack will try to avoid cache hits and row buffer hits to maximise bit-cell accesses and, if multiple rows must be used, avoid using adjacent rows that would reset the disturbance on potential victims. To determine the presence of buffers and the layout of the memory, we can measure the power consumption of the memory module when performing various operations. First, the power consumption may slightly vary from row to row because of process variations. This will allow us to partially determine the order of the bits in the 20-bits address. Then, if a row buffer or a cache exists, the power consumption when reading multiple times to the same address should be lower than when changing row between each read.

We used the platform together with a current probe on the power pins of the memory module to measure its current consumption while performing various accesses on the memory. First, we performed one read operation repeatedly on a single address. The address was set to either $\texttt{00000}_h$ or $\texttt{FFFFF}_h$ (respectively all bits at 0 and all bits at 1), the value of the read word to either $\texttt{0000}_h$ or $\texttt{FFFF}_h$, and the value of the remaining cells of the memory to either $\texttt{0000}_h$ of $\texttt{FFFF}_h$. The results of this experiment is depicted in Table 6.1.

Multiple information can be extracted from this table. First, the read data seems to have a high impact on the power consumption. Reading $\texttt{0000}_h$ consumes around 25mA, while reading $\texttt{FFFF}_h$ consumes from 40mA to 80mA. This table shows that reading $\texttt{FFFF}_h$ from $\texttt{00000}_h$ consumes almost twice the current of reading $\texttt{FFFF}_h$ from $\texttt{FFFFF}_h$. Finally, the data stored in the remaining of the memory seems to have a small

Read val.	Read addr.	remaining mem.	current
$\texttt{0000}_h$	$\texttt{00000}_h$	$\texttt{0000}_h$	22.9mA
$\texttt{0000}_h$	$\texttt{00000}_h$	$\mathbf{FFFF}_h$	27.0mA
$\texttt{0000}_h$	$\mathbf{FFFFF}_h$	$\texttt{0000}_h$	23.2mA
$\texttt{0000}_h$	$\mathbf{FFFFF}_h$	$\mathbf{FFFF}_h$	27.3mA
$\mathbf{FFFF}_h$	$\texttt{00000}_h$	$\texttt{0000}_h$	76.0mA
$\mathbf{FFFF}_h$	$\texttt{00000}_h$	$\mathbf{FFFF}_h$	79.2mA
$\mathbf{FFFF}_h$	$\mathbf{FFFFF}_h$	$\texttt{0000}_h$	39.9mA
$\mathbf{FFFF}_h$	$\mathbf{FFFFF}_h$	$\mathbf{FFFF}_h$	42.5mA

Table 6.1: *Current consumption of Toggle-MRAM memory module for read operations*

effect on the power consumption. Additionally, when the remaining of the the memory contains only $\texttt{FFFF}_h$, the current consumption is slightly higher than when it is filled with $\texttt{0000}_h$. From this experiment, we can also deduce the absence of cache and row buffer. Indeed, while doing the measurements, we have not noticed any difference in power consumption between the first operation and the subsequent ones.

Further experimentation allowed us to understand the internal architecture further: First, the difference of current consumption that comes from the address is primarily due to the bit 4 of the address. When the read value is $\texttt{FFFF}_h$, the bit 4 of the address adds around 30mA to the power consumption of the read operation when it is set to 0. This result was consistent when using multiple modules of the same model. We can make the hypothesis that this difference is due to the use of multiple banks, where half of them would have a longer path to or from the I/Os of the module, requiring more power or additional logic circuits to maintain the data.

Finally, by progressively reducing the size of the remaining memory that we set to a particular value before the measured read operations, we deducted that only the values stored at addresses which have the same most significant 10 bits actually influence the power consumption. Consequently, we can deduce that the 20 address bits are laid out as displayed in Table 6.2.

row address										column address									
MSB															❶				LSB

Table 6.2: *Toggle-MRAM deducted address layout. The most significant 10 bits are used for the row address. The least significant 10 bits are used for the column address, except for the bit 4* ❶. *This bit is hypothetically used as a bank selection bit.*

Write data	address	remaining mem.	current
$\texttt{0000}_h$	$\texttt{00000}_h$	$\texttt{0000}_h$	61mA
$\texttt{0000}_h$	$\texttt{00000}_h$	$\mathbf{FFFF_h}$	70mA
$\texttt{0000}_h$	$\mathbf{FFFFF_h}$	$\texttt{0000}_h$	60mA
$\texttt{0000}_h$	$\mathbf{FFFFF_h}$	$\mathbf{FFFF_h}$	69mA
$\mathbf{FFFF_h}$	$\texttt{00000}_h$	$\texttt{0000}_h$	70mA
$\mathbf{FFFF_h}$	$\texttt{00000}_h$	$\mathbf{FFFF_h}$	77mA
$\mathbf{FFFF_h}$	$\mathbf{FFFFF_h}$	$\texttt{0000}_h$	70mA
$\mathbf{FFFF_h}$	$\mathbf{FFFFF_h}$	$\mathbf{FFFF_h}$	77mA
$\texttt{0000}_h$ / $\mathbf{FFFF_h}$	$\texttt{00000}_h$	$\texttt{0000}_h$	115mA
$\texttt{0000}_h$ / $\mathbf{FFFF_h}$	$\texttt{00000}_h$	$\mathbf{FFFF_h}$	108mA
$\texttt{0000}_h$ / $\mathbf{FFFF_h}$	$\mathbf{FFFFF_h}$	$\texttt{0000}_h$	114mA
$\texttt{0000}_h$ / $\mathbf{FFFF_h}$	$\mathbf{FFFFF_h}$	$\mathbf{FFFF_h}$	108mA

Table 6.3: *Current consumption of Toggle-MRAM memory module for write operations. the last four measurements are done with alternately writing* $\texttt{0000}_h$ *and* $\texttt{FFFF}_h$*.*

To complete our study, we measured the current consumption when writing data into the memory. We repeated a write operation on one address, with the address set to either $\texttt{00000}_h$ or $\texttt{FFFFF}_h$, and the remaining data in the row set to either $\texttt{0000}_h$ or $\texttt{FFFF}_h$. The results of this experiment is displayed in Table 6.3.

From these measurements, we can see that changing the address, including the bit 4, does not have any significant effect on the current consumption. This corroborates our hypothesis of additional sense amplifiers for some banks, as sense amplifiers are not needed for write operations and therefore only affect the consumption of read operations. This table also shows that switching the state of the bit consumes more power than setting it to the value it already has. As switching a Toggle-MRAM bit cell from 0 to 1 uses the same process as switching it from 1 to 0, the controller must only use this process for bits that must be changed.

6.2.4 Designing the attack

When designing an attack that aims at flipping bits in the memory, we want to maximise the potential disturbance between bit-cells. Algorithm 3 describes the access pattern used to test the memory against attacks. In this algorithm, the constants N_{rows} and N_{cols} are configured with the number of rows and the number of columns, respectively. the parameter n_{rep} determines the number of times it repeats the accesses on the

same pair of addresses. The `write` function takes a row, a column and a 16-bit value as parameters and writes the value at the specified address made from the row and the column indices. As the direction of the flip is not known, the memory is initialised with alternating 0s and 1s ($\texttt{5555}_\mathrm{h}$ or $\texttt{AAAA}_\mathrm{h}$), so that both flip directions can be witnessed.

Algorithm 3: Attack algorithm on Toggle-MRAM. The `write` function takes a row, a column and a 16-bit value and writes it at the specified address.

```
1  read n_loop
2  read n_rep
3  for r ← 0 to N_rows − 1 do
4      for c ← 0 to N_cols − 1 do
5          write(r, c, AAAA_h)
6  for r ← 0 to N_rows − 2 by 2 do
7      for c ← 0 to N_cols − 1 do
8          for i ← 0 to n_rep do
9              write(r, c, 0000_h)
10             write(r + 2, c, 0000_h)
11             write(r, c, FFFF_h)
12             write(r + 2, c, FFFF_h)
```

The attack pattern is illustrated in Figure 6.8. This attack uses even rows as aggressor rows and odd rows as victim rows.

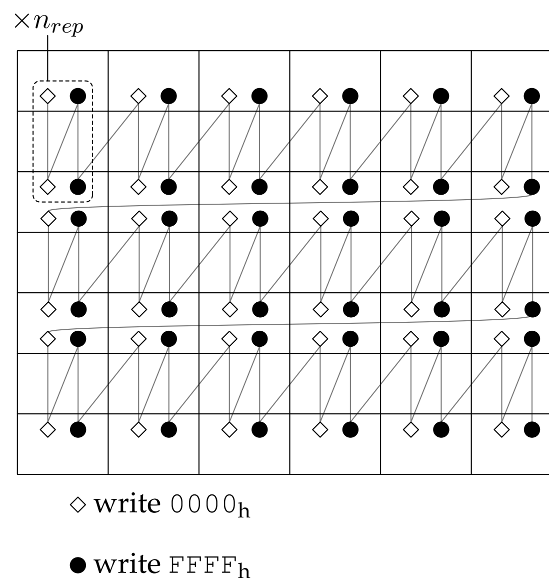


Figure 6.8: *Attack pattern on Toggle-MRAM. Memory cells are accessed by pairs, the two cells being exactly two rows from each other. The pattern is repeated n_{rep} times for each cell.*

6.2.5 Results

To increase our chances of corrupting the memory, we run all the experiments in a temperature chamber set to 80°C. We run the attack three times for approximately 100h each time, with $n_{rep} = 1$, $n_{rep} = 1 \times 10^3$ and $n_{rep} = 1 \times 10^6$. However, we have not been able to notice any bit-flip in the memory.

The lack of corruption can be due to multiple factors.

- The technology node is too big. On DRAM, the bit-flips started to appear when the memory density increased. This memory module having a very low capacity compared to modern DRAM modules, it is possible that the attacks failed because its density is too low. At this technology size, the lines may not be close enough, or the MTJ too big, for this attack to have any significant effect. However, as the technology matures and its density increases, it may become vulnerable to this type of attack.
- The memory uses protection mechanisms. While not explicitly written on the documents, this module may use simple protection mechanisms such as Error-Correcting Codes (ECC) integrated inside the chip to prevent reading corrupted values. The presence of an error-correcting mechanism could be verified by producing errors in the memory through other means, using *e.g.* laser fault injection.
- The Rowhammer attack may not be designed correctly. We tried to port the attack pattern of the Rowhammer attack that originally works on DRAM onto the Toggle-MRAM. The technology may not be vulnerable to disturbances that come from the same mechanism that cause bit-flip in DRAMs. In this case, attacking the Toggle-MRAM with this attack will not cause bit-flips.

According to the executed tests, the Toggle-MRAM technology does not seem to be currently vulnerable to the Rowhammer attack. However, future memory modules may be vulnerable to variants of this attack. Nonetheless, the experiments showed that a simple current analysis during accesses to the memory can provide information on the data that is written and its location in the address space.

6.3 Attack on an STT-MRAM chip

While the Toggle-MRAM technology uses the *classical* MRAM writing mechanism, its manufacturing complexity and characteristics do not make it a good replacement for the DRAM technology. The STT-MRAM technology however, is less mature but has lower manufacturing complexity, with better characteristics to make it a good replacement for the DRAM. Therefore, our next experiments will concern this technology.

The second memory module is an STT-MRAM external memory from Avalanche Technology [105]. This is a 2Mib memory, with 2^{18} words of 16 bits.

6.3.1 Test Platform

Its architecture is very similar to the architecture of the Toggle-MRAM module, illustrated in Figure 6.6 page 83, with the difference of having less address bits. This module has a minimum delay between two accesses set to 35ns, which is very low compared to the minimum delay of 50ns that the EMC of the FPGA can achieve. To access this memory at its maximum frequency, we need to change the test platform.

This second platform will have the same requirements as the first FPGA-based platform: a parallel interface compatible with the memory module, that can reach the 35ns minimum delay between consecutive accesses, an serial interface to communicate with a computer, and a compatibility with high temperature.

We chose the STM32F746ZG Nucleo144 board from ST Microelectronics [106]. The microcontroller of this board includes an EMC compatible with the parallel interface of the tested memories. All the pins necessary to connect the external memory to the EMC are available on the selected development board.

The architecture of the platform is illustrated in Figure 6.9. The microcontroller integrates a DMA with the same capability as the one integrated in the FPGA-based platform regarding the address patterns when accessing the external memory. Contrary to the AXI bus of the FPGA-based platform, the memory bus of this microcontroller is used by both the processor and the DMA, allowing a single EMC to be used by both.

Furthermore, the EMC can be reconfigured at run time, allowing to switch between a fast mode for the attacks and a slow mode to read or prepare the external memory.

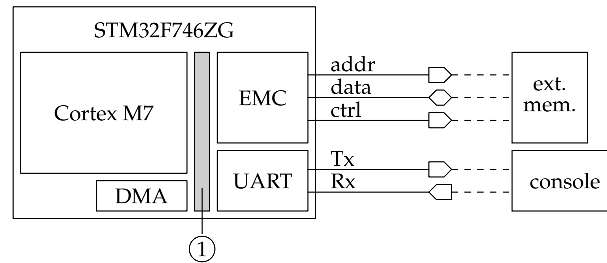


Figure 6.9: *Microcontroller-based platform architecture. The memory bus ① is used by both the processor and the DMA to communicate with the peripherals.*

6.3.2 Reverse-engineering of the memory module architecture

Like for the Toggle-MRAM memory, it is important to understand the internal architecture of the memory in order to design an attack. The datasheet of the STT-MRAM module does not specify the column and row address bits, nor if the memory uses a row buffer. In order to design an attack program, we need to know if multiple rows must be used, and what bits must be changed in the address to avoid hitting the same row As we did with the FPGA platform and the Toggle-MRAM module, we measured the current consumption of the STT-MRAM when performing read and write operations using the microcontroller platform. The results of this experiment are listed in Table 6.4. The current consumption for read and write operations did not vary as much as it did for the Toggle-MRAM. However, the variations can still be exploited to extract important information. First, the measurements show that writing $\texttt{FFFF}_h$ takes takes approximately 18% more current than reading $\texttt{0000}_h$. As illustrated in Figure 6.10, a Magnetic Tunnel Junction (MTJ) has two stable states: the parallel configuration (P state), where the Free Layer (FL) has the same magnetic orientation as the Pinned Layer (PL); and the anti-parallel configuration (AP state), where the magnetic orientation of the FL is opposite to the magnetic orientation of the PL. The energy level of the P state being lower than the energy level of the AP state, switching from the P state to the AP state requires more power than the opposite. Therefore, as the current of writing $\texttt{FFFF}_h$ on $\texttt{0000}_h$ is more important than the current when writing $\texttt{0000}_h$ on $\texttt{FFFF}_h$, we can deduce that the AP state represents a 1, and the P state represents a 0.

Read/Write	Data	current
Write	$\texttt{0000}_\text{h}$	8.5mA
Write	$\texttt{FFFF}_\text{h}$	10mA
Read	$\texttt{0000}_\text{h}$	2.2mA
Read	$\texttt{FFFF}_\text{h}$	2.3mA

Table 6.4: *current measurements during read and write operations on the STT-MRAM memory module*

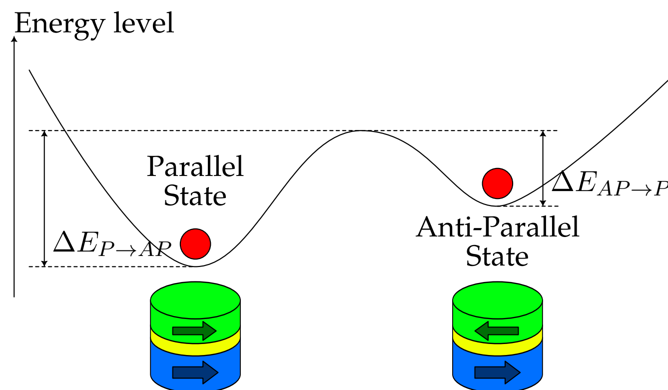


Figure 6.10: *STT-MRAM energy levels [107]. A thermal barrier separates the two stable states of the MTJ. Switching from the anti-parallel state to the parallel state requires less energy than the opposite.*

The current consumption when reading $\texttt{0000}_\text{h}$ is not significantly different than the current consumption when reading $\texttt{FFFF}_\text{h}$. When measuring the current consumption during read operations, we noticed that when changing least significant 4 bits of the address, the current is much lower than when changing bits beyond the fourth bit. The existence of a page buffer of 16 values can easily explain this, as accessing a buffer requires much less energy than reading the values from the bitcells and using sense amplifiers. Additionally, the current overhead when changing address bits beyond the fourth bit is approximately constant for all bits.

The datasheet of the memory module do not specify the number of address bits allocated to the row and the number of bits allocated to the column. However, we can safely assume that column bits are consecutive within the address bits, as well as row bits. The page buffer concerns the least four significant bits. It is safe to assume that a page buffer only concerns one row. Hence, the column bits certainly concern the least significant bits of the address, and the row bits concern the most significant bits of the address. This hypothesis is illustrated in Figure 6.11

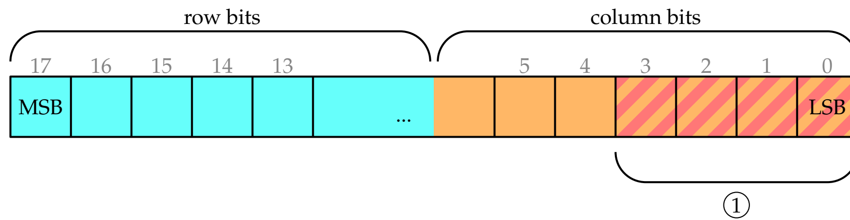


Figure 6.11: *Hypotyhesis on the order of row and column bits in the address vector of the stt-mram module. The low current when reading on values with different address bits only in the least four significant bits indicates the presence of a page buffer ①.*

6.3.3 Designing the attack

According to a recent study [63], a variation of the Rowhammer attack that could be effective on STT-MRAM memories use the high current of write operations to produce a ground bounce that propagates to unselected rows of the same bank. Hence, to perform this attack on our memories with the information at our disposal, we only need to maximise the current when doing repeated write operations. According to our experiments, changing any address bit beyond the fourth one results in the maximum current for write operations. Therefore, the attacks we conduct consist in alternately writing $\texttt{0000}_h$ and $\texttt{FFFF}_h$ at two addresses which had one bit different beyond the fourth one.

To perform this attack, we use the CPU which accesses directly the EMC peripheral that interfaces with the STT-MRAM module. Fortunately, both the CPU and the EMC are fast enough to respect the minimum delay of 35ns between two accesses.

6.3.4 Results

We run this attack for several hours, using multiple pairs of addresses. Unfortunately, we never noticed any bit-flip in the memory. As with the Toggle-MRAM module, this could be due to several factors, such as a misunderstanding of the internal architecture and behaviour, a technology node too big to be vulnerable to this attack, or too short attacks. However, this technology could still be vulnerable to better-designed attacks, and future modules might become vulnerable to this attack as their density increases.

Nonetheless, we showed that a simple current analysis during the operation of the

memory is a source of information for an aggressor. The current shows partially the locality of memory accesses (*i.e.*, whether accesses hit the page buffer or not) and the hamming weight of the data being written.

6.4 Conclusion

In this chapter, we created two platforms to evaluate the vulnerability of existing Toggle-MRAM and STT-MRAM external memory modules against the Rowhammer attack. Despite our effort, we were not able to produce any bit-flip. However, our experiments showed that while these memories have shown no vulnerability to our attacks, a simple current analysis during the operation of these memories can be used to uncover various details about the internal layout of the memory, and about the memory accesses that are performed on them.

VII

Conclusion and Perspectives

Contents

7.1 Contributions **98**

7.1.1 Rowhammer simulation in gem5 98

7.1.2 Improvement of Rowhammer mitigations 99

7.1.3 Mitigation proposals 99

7.1.4 Experiments on MRAM 99

7.2 Future work **100**

7.3 Concluding remarks **101**

The memory has always been a major component of computing systems. The main memory, implemented using the DRAM technology, is used to store run-time data, including open files, temporary variables, and sensitive information such as encryption keys. To keep up with the ever-increasing capability of modern computers, DRAM manufacturers have increased the memory density to improve the capacity and speed while maintaining a relatively low manufacturing cost. However, as its density increased, the memory became vulnerable to Rowhammer attacks, which exploits cell-to-cell disturbance to cause bit-flips. When properly exploited, this attack can be used to crash systems, reveal sensitive information or take control of the system.

Numerous mitigation techniques were proposed, using various methods to detect aggressors or to prevent corruption on vulnerable data, using mechanisms implemented in software, in the memory controller or in the memory devices themselves, using probabilistic or counter-based algorithms. Despite nearly a decade of research to stop the attacks from harming the systems, the issue has only gotten worse over the years, with an increasing vulnerability as the technology scaled down.

7.1 Contributions

7.1.1 Rowhammer simulation in gem5

To make the development of Rowhammer countermeasures easier and faster, we improved the architecture simulator gem5 with a memory-corruption module, capable of simulating the memory corruption from Rowhammer attacks. This tool allows countermeasure designers to easily evaluate their countermeasure in a realistic computer architecture, using existing components or future technologies with different vulnerabilities to the attack.

7.1.2 Improvement of Rowhammer mitigations

The most efficient Rowhammer mitigation techniques use activation counters to detect the aggressors, with a bank-level counting granularity. The most important drawback of counter-based mitigation techniques is the addition memory required to store all the counters As the memory bandwidth at rank level is not limited by the memory bandwidth at bank level, we pointed out that moving from bank-level counting granularity to rank-level counting granularity would greatly reduce the size of this additional memory. However, the reduced delay between ACT commands at rank-level could prevent some mitigations from working properly.

7.1.3 Mitigation proposals

We proposed two Rowhammer detection mechanisms. First, we used a machine learning algorithm to categorise hardware event traces as depicting an attack or a normal behaviour. While this solution shows encouraging results, it does not guarantee protection, and its hardware implementation is heavy compared to state-of-the-art counter-based detection mechanisms. Our second proposal is F-CoRD, a counter-based detection mechanism that evaluates the activation frequency of DRAM rows to keep track of only frequently-activated rows and their activation count to detect aggressor rows, with an adaptive usage of the counters.

7.1.4 Experiments on MRAM

Finally, as a line of research, we evaluated the vulnerability of recent Toggle-MRAM and STT-MRAM memory module against Rowhammer attacks. A few publications claimed that this technology might be vulnerable to variations of the Rowhammer attack, but did not verify it using physical memory modules. For this purpose, we developed two platforms to attack commercial memory modules following various access patterns. We were unable to witness any corruption of the memory, but this does not prove that the memory is not vulnerable. Future memory modules might become vul-

nerable to attacks as the density of the memory increases. However, our experiments showed that a current analysis during the operation of the memory can be used to discover various information about the internal layout of the memory, and the nature, location and value of an operation on the memory. An aggressor might be able to use it to discover sensitive information.

7.2 Future work

Multiple points can be improved in future work.

Regarding the Rowhammer simulation, the modifications we made on Ramulator can be ported on the integrated memory simulator of gem5. We initially chose Ramulator as the main memory simulator. Recently, gem5 integrated the cycle-accurate DRAM simulator DRAMSim3. Porting the modifications on the integrated memory simulator of gem5 could prove useful to extend the usage of our memory-corruption module. Additionally, the simulation can be made more accurate: we can add the disturbance of not-immediate neighbours of aggressor rows, that is not integrated yet because of its complexity; the corruption models of new memory technologies like MRAM can be integrated to the simulator to allow the development of specific countermeasures.

For our second contribution, we proposed to reduce the memory overhead of counter-based Rowhammer mitigations by changing their counting granularity to rank level. This improvement faced the issue of the increased ACT frequency that the mechanism may not be able to withstand: at rank level, the delay between two consecutive ACTs can be one order of magnitude lower than at bank level. Specifically, mitigation techniques based on Content-Addressable Memories (CAM) may not be able to process the ACTs fast enough. To fix this issue, specialised CAM could be developed, based on pipelined CAM which can more easily withstand the higher ACT frequency.

Concerning the two mitigation proposals, while the first ML-based proposal calls for several improvements, recent publications have already improved this solution with better-designed ML models and better hardware integration [44]. This solution could be explored further by evaluating other machine-learning models, and by op-

timising the event counters in the architecture. The second proposal, however, is to this day still in development. It necessitates the creation of specialised CAMs to implement the necessary operations, and we believe it can be further improved to reduce the amount of memory it needs, to make it comparable to other state-of-the-art mitigation proposals.

Finally, the experimentation on MRAM memories is far from complete. Failing to witness corruption does not mean that the technology is not vulnerable to Rowhammer attacks. Our platforms can be improved on multiple points, including the automatic control of the temperature of the memory to put more constraints on the memory. Moreover, the power supply voltage of the memory can be lowered and the access timings can be shorten in order to reduce the retention time and therefore increase the vulnerability of these memories to disturbance. Future publications on the subject of Rowhammer attacks on MRAM could be used to improve the attack pattern and try to trigger bit-flips in the memory. A collaboration with MRAM manufacturers could further improve the platform, with a precisely-known memory architecture, and evaluation of state-of-the-art memory technologies.

7.3 Concluding remarks

In this thesis, we presented a tool to facilitate the development and evaluation of Rowhammer countermeasures, proposed some improvement on existing countermeasures and built new detection mechanisms that could be used as a base for mitigation techniques. We hope that our work will help DRAM manufacturers and countermeasure designers be able to develop new mechanisms, and integrate appropriate functionalities in their products to protect future DRAM modules against Rowhammer attacks. However, with the introduction of new memory technologies as non-volatile replacements for the DRAM technology, other vulnerabilities will certainly be discovered on these new technologies, which will require appropriate countermeasures, and tools to develop to develop them.

Bibliography

[1] Yoongu Kim et al. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014. 2, 13, 15, 16, 20, 22, 23, 27, 37, 55

[2] Andrew Kwong et al. Rambleed: Reading bits in memory without accessing them. In *SP*, 2020. 2, 17

[3] Moritz Lipp et al. Nethammer: Inducing rowhammer faults through network requests. In *EuroS&PW*, 2020. 2, 16, 17

[4] Daniel Gruss et al. Rowhammer.js: A remote software-induced fault attack in javascript. In *DIMVA*, 2016. 2, 16, 17, 28

[5] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Usenix Security Symposium*, 2008. 12

[6] JEDEC. JESD79-3 DDR3 SDRAM, 2013. 12

[7] JEDEC. JESD79-4 DDR4 SDRAM, 2021. 12

[8] Micron. DDR5 SDRAM product core datasheet, 2021. 12

[9] Michael Redeker, Bruce F Cockburn, and Duncan G Elliott. An investigation into crosstalk noise in DRAM structures. In *Proceedings of the 2002 IEEE International Workshop on Memory Technology, Design and Testing (MTDT2002)*, pages 123–129. IEEE, 2002. 14

[10] Zaid Al-Ars, Said Hamdioui, Ad Van De Goor, Georgi Gaydadjiev, and Joerg Vollrath. DRAM-specific space of memory tests. In *2006 IEEE International Test Conference*, pages 1–10. IEEE, 2006. 14

[11] Rei-Fu Huang, Hao-Yu Yang, Mango C-T Chao, and Shih-Chin Lin. Alternate hammering test for application-specific DRAMs and an industrial case study. In *DAC Design Automation Conference 2012*, pages 1012–1017. IEEE, 2012. 14

[12] Pierre Chor-Fung Chia, Shi-Jie Wen, and Sang H Baeg. New DRAM HCI qualification method emphasizing on repeated memory access. In *2010 IEEE International Integrated Reliability Workshop Final Report*, pages 142–144. IEEE, 2010. 14

[13] Thomas Yang and Xi-Wei Lin. Trap-assisted dram row hammer effect. *IEEE Electron Device Letters*, 40(3):391–394, 2019. 14, 23, 29

[14] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2019. 15

[15] Jeremie S Kim, Minesh Patel, A Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur 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)*, pages 638–651. IEEE, 2020. 15, 16, 17, 27, 40, 41, 45, 47, 60, 75

[16] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1675–1689, 2016. 15, 16

[17] Zhenkai Zhang, Zihao Zhan, Daniel Balasubramanian, Xenofon Koutsoukos, and Gabor Karsai. Triggering Rowhammer hardware faults on ARM: A revisit. In *Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security*, pages 24–33, 2018. 16

[18] Zelalem Birhanu Aweke et al. ANVIL: Software-based protection against next-generation rowhammer attacks. *SIGPLAN Notices*, 2016. 16, 20

[19] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the GPU. In *2018 ieee symposium on security and privacy (sp)*, pages 195–210. IEEE, 2018. 16

[20] Moritz Lipp. *Cache attacks and rowhammer on arm*. PhD thesis, Graz University of Technology, 2016. 16

[21] Finn de Ridder et al. SMASH: Synchronized many-sided rowhammer attacks from javascript. In *USENIX Security*, 2021. 16, 17, 28

[22] Salman Qazi et al. "Half-Double": Next-row-over assisted rowhammer. https://github.com/google/hammer-kit/blob/main/20210525_half_double.pdf, 2021. 16, 17, 41

[23] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat*, 2015. 17, 29

[24] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious case of rowhammer: flipping secret exponent bits using timing analysis. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 602–624. Springer, 2016. 17

[25] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1–18, 2016. 17

[26] "BIOS and kernel developer's guide (BKDG) for AMD family 10h processors, 2013. 18

[27] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *25th USENIX security symposium (USENIX security 16)*, pages 565–581, 2016. 18, 29

[28] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops:{Cross-VM} row hammer attacks and privilege escalation.

In *25th USENIX security symposium (USENIX Security 16)*, pages 19–35, 2016. 18, 29

[29] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Software-only reverse engineering of physical dram mappings for rowhammer attacks. In *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*, pages 19–24. IEEE, 2018. 18

[30] Mohamed Hassan, Anirudh M Kaushik, and Hiren Patel. Reverse-engineering embedded memory controllers through latency-based analysis. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 297–306. IEEE, 2015. 18

[31] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (salp) in dram. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 368–379. IEEE, 2012. 18

[32] Tao Zhang, Boris Pismenny, Donald E Porter, Dan Tsafrir, and Aviad Zuck. Rowhammering storage devices. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, pages 77–85, 2021. 18

[33] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 47–66. Springer, 2018. 18

[34] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762. IEEE, 2020. 18, 19, 23, 54

[35] Manaar Alam et al. Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks. *IACR Cryptology ePrint Archive*, 2017. 20

[36] Anirban Chakraborty et al. Deep learning based diagnostics for rowhammer protection of DRAM chips. In *ATS*, 2019. 20, 29, 54

[37] Anirban Chakraborty, Manaar Alam, and Debdeep Mukhopadhyay. A good anvil fears no hammer: Automated rowhammer detection using unsupervised deep learning. In *International Conference on Applied Cryptography and Network Security*, pages 59–77. Springer, 2021. 20

[38] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: Comprehensive and compatible software protection against rowhammer attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 697–710, 2018. 20

[39] Yicheng Wang, Yang Liu, Peiyun Wu, and Zhao Zhang. Discreet-PARA: Rowhammer defense with low cost and high efficiency. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 433–441. IEEE, 2021. 20

[40] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. Making DRAM stronger against row hammering. In *DAC*, 2017. 20, 27

[41] Jung Min You and Joon-Sung Yang. MRLoc: Mitigating row-hammering based on memory locality. In *DAC*, 2019. 21, 28

[42] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking*, 8(3):281–293, 2000. 21

[43] Kwangrae Kim, Jeonghyun Woo, Junsu Kim, and Ki-Seok Chung. Hammerfilter: Robust protection and low hardware overhead method for rowhammer. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 212–219. IEEE, 2021. 21

[44] Biresh Kumar Joardar, Tyler K Bletsch, and Krishnendu Chakrabarty. Learning to mitigate rowhammer attacks. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 564–567. IEEE, 2022. 21, 29, 100

[45] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. Counter-based tree structure for row hammering mitigation in dram. *IEEE Computer Architecture Letters*, 16(1):18–21, 2016. 22

[46] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. Mitigating wordline crosstalk using adaptive trees of counters. In *ISCA*, 2018. 22

[47] Ingab Kang, Eojin Lee, and Jung Ho Ahn. Cat-two: Counter-based adaptive tree, time window optimized for dram row-hammer prevention. *IEEE Access*, 8:17366–17377, 2020. 22, 50

[48] A Giray Yağlikçi et al. Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed DRAM rows. In *HPCA*, 2021. 22, 27, 46, 62

[49] Jayadev Misra and David Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982. 22, 46

[50] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International conference on database theory*, pages 398–412. Springer, 2005. 22

[51] Yeonhong Park et al. Graphene: Strong yet lightweight row hammer protection. In *MICRO*, 2020. 22, 27, 37, 46, 62

[52] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1056–1069, 2022. 22

[53] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W Lee, and Jung Ho Ahn. Mithril: Cooperative row hammer protection on commodity dram leveraging managed refresh. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1156–1169. IEEE, 2022. 22

[54] Tanj Bennett, Stefan Saroiu, Alec Wolman, and Lucian Cojocar. Panopticon: A complete in-dram rowhammer mitigation. In *Workshop on DRAM Security (DRAMSec)*, 2021. 22

[55] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. TWiCe: Preventing row-hammering by exploiting time window counters. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 385–396, 2019. 22, 27

[56] Kuljit Bains, John Halbert, Christopher Mozak, Theodore Schoenborn, and Zvika Greenfield. Row hammer refresh command, August 25 2015. US Patent 9,117,544. 23

[57] Kevin Loughlin, Stefan Saroiu, Alec Wolman, and Baris Kasikci. Stop! hammer time: rethinking our approach to rowhammer mitigations. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 88–95, 2021. 23

[58] Kuljit S Bains and John B Halbert. Row hammer monitoring based on stored row hammer threshold value, August 1 2017. US Patent 9,721,643. 23

[59] Hasan Hassan, Yahya Can Tugrul, Jeremie S Kim, Victor Van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1198–1213, 2021. 23

[60] Chia-Ming Yang, Chen-Kang Wei, Yu Jing Chang, Tieh-Chiang Wu, Hsiu-Pin Chen, and Chao-Sung Lai. Suppression of row hammer effect by doping profile modification in saddle-fin array devices for sub-30-nm dram technology. *IEEE Transactions on Device and Materials Reliability*, 16(4):685–687, 2016. 23

[61] Seong-Wan Ryu, Kyungkyu Min, Jungho Shin, Heimi Kwon, Donghoon Nam, Taekyung Oh, Tae-Su Jang, Minsoo Yoo, Yongtaik Kim, and Sungjoo Hong. Overcoming the reliability limitation in the ultimately scaled dram using silicon migration technique by hydrogen annealing. In *2017 IEEE International Electron Devices Meeting (IEDM)*, pages 21–6. IEEE, 2017. 23

[62] S Agarwal, H Dixit, D Datta, M Tran, D Houssameddine, D Shum, and F Benistant. Rowhammer for spin torque based memory: Problem or not? In *2018 IEEE International Magnetics Conference (INTERMAG)*, pages 1–1. IEEE, 2018. 27, 80, 81

[63] Mohammad Nasim Imtiaz Khan and Swaroop Ghosh. Analysis of row hammer attack on STTRAM. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 75–82. IEEE, 2018. 27, 81, 82, 94

[64] Jung Ho Ahn, Sheng Li, O Seongil, and Norman P Jouppi. Mcsima+: A many-core simulator with application-level+ simulation and detailed microarchitecture modeling. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 74–85. IEEE, 2013. 27

[65] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2015. 27, 31, 57

[66] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 2011. 28, 30, 56

[67] Loïc France et al. Vulnerability assessment of the rowhammer attack using machine learning and the gem5 simulator - work in progress. In *SaT-CPS*, 2021. 29

[68] Ayaz Akram and Lina Sawalha. A survey of computer architecture simulation techniques and tools. *Ieee Access*, 7:78120–78145, 2019. 30

[69] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, pages 469–480, 2009. 30

[70] Rafael Ubal, Julio Sahuquillo, Salvador Petit, and Pedro Lopez. Multi2Sim: A simulation framework to evaluate multicore-multithreaded processors. In *19th*

International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07), pages 62–68. IEEE, 2007. 30

[71] Daniel Aarno and Jakob Engblom. *Software and system development using virtual platforms: full-system simulation with wind river simics*. Morgan Kaufmann, 2014. 30

[72] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002. 30

[73] Ashutosh Dhodapkar, Chee How Lim, George Cai, and W Robert Daasch. Tem²p²est: A thermal enabled multi-model power/performance estimator. In *International Workshop on Power-Aware Computer Systems*, pages 112–125. Springer, 2000. 30

[74] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE computer architecture letters*, 10(1):16–19, 2011. 31

[75] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. DRAMsim3: a cycle-accurate, thermal-capable DRAM simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020. 31

[76] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are we susceptible to rowhammer? an end-to-end methodology for cloud providers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 712–728. IEEE, 2020. 39

[77] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007. 41

[78] Valgrind. https://valgrind.org/. 41

[79] Loïc France, Florent Bruguier, Maria Mushtaq, David Novo, and Pascal Benoit. Implementing rowhammer memory corruption in the gem5 simulator. In *32nd International Workshop on Rapid System Prototyping (RSP)*. IEEE, 2021. 42

[80] Quentin Forcioli, Jean-Luc Danger, Clémentine Maurice, Lilian Bossuet, Florent Bruguier, Maria Mushtaq, David Novo, Loïc France, Pascal Benoit, Sylvain Guilley, et al. Virtual platform to analyze the security of a system on chip at microarchitectural level. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 96–102. IEEE, 2021. 42

[81] Loïc France, Florent Bruguier, Maria Mushtaq, David Novo, and Pascal Benoit. Implementation of rowhammer effect in gem5. In *15ème Colloque National du GDR SoC*^{2}, 2021. 42

[82] Loïc France, Florent Bruguier, Maria Mushtaq, David Novo, and Pascal Benoit. Modeling Rowhammer in the gem5 simulator. CHES 2022 - Conference on Cryptographic Hardware and Embedded Systems, September 2022. Poster. 42

[83] Micron. Micron DDR5 SDRAM: New features, 2021. 45

[84] Deke Guo, Yunhao Liu, Xiangyang Li, and Panlong Yang. False negative problem of counting bloom filter. *IEEE transactions on knowledge and data engineering*, 22(5):651–664, 2010. 47

[85] Supreet Jeloka et al. A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6t bit cell enabling logic-in-memory. *JSSC*, 2016. 50

[86] Kostas Pagiamtzis and Ali Sheikholeslami. A low-power content-addressable memory (CAM) using pipelined hierarchical search scheme. *IEEE Journal of Solid-State Circuits*, 39(9):1512–1519, 2004. 50

[87] Swapan Kumar Ray. Large-capacity high-throughput low-cost pipelined cam using pipelined ctam. *IEEE Transactions on Computers*, 55(5):575–587, 2006. 50

[88] Loïc France, Florent Bruguier, David Novo, Maria Mushtaq, and Pascal Benoit. Reducing the silicon area overhead of counter-based rowhammer mitigations. In *18th CryptArchi Workshop*, 2022. 51

[89] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995. 58

[90] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 58

[91] Paul Werbos. Beyond regression: new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974. 58

[92] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989. 58

[93] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 58

[94] François Chollet. Keras. https://keras.io/, 2015. 58

[95] Machine learning mastery, 2013. 58

[96] Jason Brownlee. LSTMs for human activity recognition time series classification, 2018. 58

[97] Jason Brownlee. 1D convolutional neural network models for human activity recognition, 2018. 58

[98] Jason Brownlee. Deep learning for time series, 2018. 58

[99] Tetsuo Endoh, Hiroki Koike, Shoji Ikeda, Takahiro Hanyu, and Hideo Ohno. An overview of nonvolatile emerging memories—spintronics for working memories. *IEEE journal on emerging and selected topics in circuits and systems*, 6(2):109–119, 2016. 79

[100] E Chen, D Apalkov, Z Diao, A Driskill-Smith, D Druist, D Lottis, V Nikitin, X Tang, S Watts, S Wang, et al. Advances and future prospects of spin-transfer torque random access memory. *IEEE Transactions on Magnetics*, 46(6):1873–1878, 2010. 79

[101] BN Engel, J Akerman, B Butcher, RW Dave, M DeHerrera, M Durlam, G Grynkewich, J Janesky, SV Pietambaram, ND Rizzo, et al. A 4-mb toggle

mram based on a novel bit and switching method. *IEEE Transactions on Magnetics*, 41(1):132–136, 2005. 79

[102] Everspin. MR4A16BUYS45 toggle-mram documentation. https://www.everspin.com/supportdocs/MR4A16BUYS45. 82

[103] Xilinx. Arty z7-20: SoC zynq®-7000 development board for makers and hobbyists. https://www.xilinx.com/products/boards-and-kits/1-pdb0q2.html. 83

[104] Xilinx. Zynq 7000 product page. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. 83

[105] Avalanche Technology. AS3004316-035nX0ITAY stt-mram datasheet. https://www.avalanche-technology.com/wp-content/uploads/1Mb-64Mb-Parallel-x16-MRAM.pdf. 91

[106] ST Microelectronics. NUCLEO-F746ZG development board. https://www.st.com/en/evaluation-tools/nucleo-f746zg.html. 91

[107] Luc Tillie, E Nowak, RC Sousa, M-C Cyrille, B Delaet, T Magis, A Persico, J Langer, B Ocker, IL Prejbeanu, et al. Data retention extraction methodology for perpendicular stt-mram. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 27–3. IEEE, 2016. 93