# Hitting the Nail on the Head - Exploring the Post-Quantum Potential of Rowhammer

Vincent Palmer, Hannes Ryberg

Department of Electrical and Information Technology
Lund University

Supervisor: Qian Guo, Alexander Nilsson

Examiner: Thomas Johansson

May 15, 2024

Printed in Sweden
Tryckeriet i E-huset, Lund

# Abstract

The dense arrangement of memory cells in modern computing systems introduces a significant concern known as read disturbance errors, stemming from the electrical properties of the capacitors within memory cells. When a memory cell is read, there exists a probability that it may inadvertently discharge its stored electrical charge to neighbouring cells, potentially altering their state from uncharged to charged.

This phenomenon poses a significant threat to the security of systems and the integrity of data in memory, as these errors can lead to unintended alteration in stored information, potentially compromising the confidentiality of sensitive data.

One attack that uses this vulnerability is called *Rowhammer*. In this attack, memory rows above and below a targeted memory row are repeatedly accessed. Through this repeated access, bit-flips can be induced in the targeted row. This discovery has led to many different applications of the attack, including one detailed in a paper by Michael Fahr Jr. et al. called *When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer*. In their work, the Rowhammer attack is utilized to compromise a cryptographic algorithm called FrodoKEM, rendering it vulnerable to exploitation.

The question arises: How does the attack perform on other platforms? Can we get a more detailed evaluation, and can it be further modified in order to target other cryptographic algorithms? That is what we aim to investigate.

We successfully implement the attack in the programming language Rust and show that the attack is indeed reproducible. Furthermore, we discuss the potential of using the attack on other algorithms, and argue that it's theoretically possible if the attack can be made faster or the victim algorithm can be made slower.

# Acknowledgements

We first and foremost want to thank our supervisors, Qian and Alexander for their support during our work with the thesis. Without their thoughts and ideas this thesis would not be possible in the first place. We also want to thank the employees at Advenica for their help and support when problems arose or when we had questions.

# Popular Science Summary

Rowhammer: The Mischievous Patron Cracking Quantum Tomes

Vincent Palmer, Hannes Ryberg

Have you ever wondered how the memory in your computer works? Think of it like a vast library, its rows of memory cells resembling shelves filled with essential pieces of data, much like books. Yet, just as accidents can occur in libraries so too can they happen in computer memory. One such mishap is known as "read disturbance errors".

When your computer accesses data from one location, there's a risk of unintentionally altering data in neighboring memory cells. Think of it as pulling out a book from a shelf in the library and inadvertently causing books on a neighbouring shelf to topple. This vulnerability poses a significant security risk to computer memory, potentially leading to altering of sensitive information. Enter "Rowhammer". In this attack, memory rows are repeatedly accessed, resulting in the alteration—or "flipping"—of data in neighbouring rows. Almost like a mischievous patron repeatedly pulling out and pushing back the same book on a shelf, eventually causing books on neighbouring shelves to fall.

Recent research has demonstrated how Rowhammer can be used to compromise FrodoKEM, a cryptographic algorithm employed for securing communication. Essentially, its like finding a way to tamper with the lock of a secure vault, putting sensitive information at risk.

Our investigation delved into implementing this attack on other platforms and its potential to target other cryptographic algorithms. Using Rust as the programming language for our implementation, we successfully managed to reproduce the attack and argued for its theoretical viability on other algorithms.

Understanding and mitigating vulnerabilities like this is paramount as our reliance on digital data grows. Like a vigilant librarian safeguarding the books on the shelves, we must ensure that our sensitive information remains secure in an ever-evolving digital landscape.

# Table of Contents

# List of Figures

Chapter 1

# Introduction

Nearly all cryptographic algorithms rely on one of three computationally hard to solve mathematical problems; factorizing an integer, the discrete logarithm problem and the elliptic curve discrete logarithm problem, these problems all have one thing in common and that is that they can all be said to be forms of "*one-way functions*".

One-way functions are mathematical functions that are easy to compute in one direction but computationally difficult to reverse. In other words, given an input, it's easy to compute the output, but given the output, it's computationally infeasible to determine the original input. Thus making these kinds of functions ideal for ensuring data security.

For an ordinary computer solving these problems is very computationally intensive, and cannot be solved by any other method than essentially brute-forcing the solution. A sufficiently powerful quantum computer, however, could disrupt this, as there is an algorithm for quantum computers called Shor's algorithm which potentially could solve any of these problems easily.

To combat this potential future disruption of known cryptography systems, efforts have been made to develop so-called *post-quantum cryptographic algorithms*. These are algorithms which instead rely on problems that—as far as we know—cannot be easily solved on a quantum computer. In order to allow for a standardization of which post-quantum algorithms to use in the future, the *National Institute of Standards and Technology* (NIST) initiated an "algorithm competition" called NIST PQC standardization. This took place 2022 and resulted in the algorithm CRYSTALS-Kyber (or just Kyber) being chosen as the proposed future standard to use when working with public-key encryption and key-establishment [19].

During roughly the same time as the competition took place, a paper was published which described a successful attack on the post-quantum algorithm *FrodoKEM*. The attack used an exploit called Rowhammer to induce flips of single bits in working memory. By flipping specifically chosen bits, an error could be induced into the generated public key, which in turn made it possible to retrieve information about the private key generated by FrodoKEM.

Even though FrodoKEM didn't win the NIST PQC standardization effort it still remained a strong contender for standardization, as it lasted until round 3 of 4 of the competition. This is further proven by the German authority BSI recommending the use of FrodoKEM [9]. Furthermore FrodoKEM as an algorithm is very similar to Kyber and even though it is a slower algorithm it is more conservative in its implementation.

Fahr et al. concludes their paper *When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer* with mentioning that they believe other NIST PQC algorithms could be broken using a similar attack as the one presented in the paper, and specifically names Kyber as one of the potential targets [8]. The aim of this master's thesis is therefore to explore that claim and hopefully shed some light on the possibility of breaking Kyber.

## 1.1 Purpose and Goal of this Thesis

In this master's thesis, we investigate the impact and usability of Rowhammer attacks on post-quantum algorithms. Our primary goal is to reproduce the attack discovered by Fahr et al. by implementing it in the programming language Rust. In addition, we aim to explore the feasibility of implementing such attacks specifically using the Rust programming language as our sole implementation language.

The claim made by Fahr et al. that a similar attack should theoretically be able to target Kyber forms the basis of the secondary goal of this thesis. We aim to modify and expand our implemented attack to be able to target Kyber, which is proposed as the future standard of post-quantum algorithms by NIST. Specifically, we seek to investigate whether Kyber is susceptible to a Rowhammer attack similar to the one described in the paper *When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer*.

## 1.2 Contributions

The primary contribution of this thesis lies in the development, implementation and exploration of a novel attack methodology targeting the FrodoKEM cryptographic algorithm, as initially developed and proposed by Fahr et al.. Through rigorous experimentation and analysis, we demonstrate the feasibility of inducing bit-flips by employing a Rowhammer attack on the FrodoKEM key generation process. Specifically, this attack targets the error matrix of the FrodoKEM public key, potentially compromising the integrity of the generated keys and leading to security vulnerabilities. Additionally, our investigation extends to exploring the potential applicability of the attack methodology to other cryptographic algorithms, such as Kyber. By highlighting the importance of protecting memory against Rowhammer attacks, our findings contribute to a broader understanding of hardware-based-attack and memory vulnerabilities.

## 1.3 Limitations

Developing and implementing an attack of this complexity is a significant undertaking. It requires an in-depth understanding of various hardware and software specifications, as well as extensive experimentation and iteration. While developing an entirely new theoretical attack from scratch would be an impressive achievement, it exceeds the practical scope of this thesis. Instead, our goal will be to implement and modify an existing attack methodology as a proof of concept.

It is important to note that the proposed attack methodology has certain technical prerequisites. For the purposes of our theoretical analysis, we'll assume that the attack process will be executed with elevated privileges, and that ASLR[1] is disabled on the target system. There exists known techniques to bypass these requirements [6, 23], but incorporating them was deemed too complex for the scope of this thesis. However, a more comprehensive attack scenario could potentially incorporate such techniques to address these limitations.

The primary objective of this thesis is therefore to provide a theoretical framework and a practical demonstration of the proposed attack methodology, highlighting its potential impact and implications. By modifying an existing attack, we aim to contribute to the understanding of these complex security vulnerabilities and foster further research in this field.

## 1.4 Disposition

Chapter 2 aims to present important background information needed to understand the problem at hand, present the scenario as well as the necessary theory that is included in the thesis. The contents of this chapter ranges from post-quantum cryptography to Dynamic Random Access Memory (DRAM). Chapter 3 covers the methodology used to be successful in the attack i.e. how is the attack carried out, what steps have been taken and what needs to work in order to extract the correct information from the targeted system. Chapter 4 is then used to present the results of our investigation and what information was extracted from our attack. Chapter 5 aims to discuss the relevance of the results and if they are meaningful enough. Lastly Chapter 6 is used as a conclusion to wrap up the thesis.

[1] ASLR stands for Adress Space Layout Randomization and is used to randomize the locations of data loaded into memory.

Chapter 2

# Background and Theory

## 2.1 Cryptography

Cryptography is a very important field in today's day and age. Since messages and data is almost always sent over the internet and through networks where it might be readable for unauthorized users we want to have some way of making data readable only to the intended recipient of the message. This problem is solved by cryptography. Say that we have two people Alice and Bob that wants to send a message between each other that they do not want a third person Eve to be able to read. To achieve this secure communication Alice can encrypt the message she wants to send to Bob with a key or secret that only they know, then Bob will be able to decrypt the message once received, but Eve who does not know the secret cannot decrypt the transmitted message and in turn can't read the actual message.

### 2.1.1 Encryption and Decryption

Encryption and decryption are processes that are used in cryptography to achieve confidentiality in messages. This confidentiality can be achieved in many different ways but the most common way is through cryptographic key algorithms.

Encryption is the process of turning readable data (plaintext) into unreadable data (ciphertext). This is achieved by using said key algorithms on the data to turn the plaintext unreadable. Decryption is the exact opposite process, turning the ciphertext into plaintext by using the decryption method provided by the employed cryptographic algorithm.

## 2.2 Key Algorithms

Modern day cryptographic algorithms can be broadly classified in two different ways, depending on the type of the used keys: Symmetric-key algorithms and Asymmetric-key algorithms (public-key algorithms).

### 2.2.1 Symmetric-Key Algorithms

Symmetric-key algorithms, also referred to as private-key or secret-key algorithms, constitute a significant component of modern cryptography. These algorithms operate by utilizing a shared secret key, known to both the message sender and receiver. This secret key serves as the basis for both the encryption process (performed by the sender) and the decryption process (performed by the receiver). Symmetric key algorithms stand out for having high efficiency and speed, rendering them well-suited for the encryption of big volumes of data. However, a big challenge accompanying the use of these algorithms relates to the secure distribution and management of these secret keys to facilitate efficient communication between parties. This is particularly challenging in situations where safeguarding the process of key exchange is of paramount concern.

One of the most pervasive symmetric-key algorithms in use today is the Advanced Encryption Standard (AES). The AES algorithm supports various key lengths, offering a robust level of security. Its versatility is evident in its widespread application across contemporary information systems.

### 2.2.2 Asymmetric-Key Algorithms

Asymmetric-key algorithms, also referred to as public-key algorithms, introduce a novel concept in cryptography. In contrast to symmetric-key algorithms, which use a single key for both encryption and decryption, asymmetric-key algorithms utilize a pair of keys: one for encryption (public key) and another for decryption (private key), collectively known as a public-private key pair. The security of asymmetric-key algorithms is rooted in the intricate mathematical relationships behind these key pairs. These relationships render it computationally unfeasible to deduce the private key from the public key, despite their inherent connection, i.e. making the algorithms act as one-way functions. This fundamental structure allows anyone to encrypt data by using the recipient's public key, but only the intended message recipient can decipher it by using their private key. Because of these mathematical relationships asymmetric algorithms are also not suitable for bulk data due to the computational overhead and slower speed when compared to symmetric algorithms.

Unlike symmetric-key algorithms, asymmetric-key algorithms offer a secure means of communication over insecure networks. This advantage arises from the ability to openly exchange public keys without directly jeopardizing the security of the corresponding private key within the pair.

## 2.3 Post-Quantum Cryptography

With the likely arrival of quantum computers in the future a new problem arises. "How will our current cryptographic systems stay secure?". The answer is; they won't. With sufficiently powerful quantum computers currently used cryptographic algorithms will most likely be broken and rendered insecure. For that reason new quantum-safe cryptographic algorithms are needed. This is where

post-quantum cryptography (PQC) comes in. Post-quantum cryptography refers to cryptographic algorithms that are thought to be secure against cryptographic attacks by a quantum computer. The problem with current cryptographic algorithms is that they mostly rely on one of three computationally hard to solve mathematical problems. Integer factorization, the discrete logarithm problem or the elliptic-curve discrete logarithm problem. Every single one of these problems could be relatively easily solved by a quantum computer running an algorithm called *Shor's algorithm* [5]. This is what post-quantum cryptography aims to prevent.

### 2.3.1 Lattice-Based Cryptography

A solution to this problem are cryptographic systems based on lattices. These lattices are in a way multidimensional grids that extends infinitely in all directions. The lattices used in cryptographic systems are defined by sets of vectors and are characterized by a lattice basis. The relationships between these vectors then give rise to computationally challenging problems which is what lattice-based cryptography is based upon.

As mentioned lattice-based cryptography relies on the difficulty of lattice problems, especially the Shortest Vector Problem (SVP) and the Learning With Errors (LWE) problem. These problems when solved efficiently would present significant security concerns to lattice-based schemes. But due to the underlying structure of lattices these problems are believed to be computationally unfeasible to crack, even for quantum computers [20].

## 2.4 Key Encapsulation Mechanisms

Key encapsulation mechanisms are a way of being able to use symmetric keys for communication but transmitting the used symmetric key with asymmetric-key algorithms. Thus giving us the security of asymmetric-key algorithms while still having the efficiency and speed of symmetric-key algorithms. The way this works is by first generating a random symmetric key and then encrypting it using a chosen public-key algorithm. The recipient of the public key encrypted message then decrypts it and in turn receives the symmetric key to be used in the communication. This is very efficient since the only real efficiency downside of this mechanism is when first key establishment is computing. After this, both the recipient and sender share the same symmetric key to use in their communication and can send long messages with much smaller computational overhead due to using symmetric encryption as opposed to asymmetric encryption [7].

## 2.5 Learning with Errors Problem

The Learning With Errors (LWE) problem is one of the mathematical problems that lattice-based cryptography is based on. It uses the idea of representing the secret information as a set of equations with errors in them, i.e. LWE is a way of hiding the value of secrets by introducing noise to them. Furthermore LWE can

be reduced to lattice problems making it fall into the lattice category of cryptographical problems. The LWE distribution is defined as:

*Let $n, q$ be positive integers, and let $\chi$ be a distribution over $\mathbb{Z}$. For an $s \in \mathbb{Z}_q^n$, the LWE distribution $A_{s,\chi}$ is the distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ obtained by choosing $a \in \mathbb{Z}_q^n$ uniformly at random and an integer error $e \in \mathbb{Z}$ from $\chi$, and outputting the pair $(a, \langle a, s \rangle + e \mod q) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ [1].*

In essence, the goal of the LWE problem is to solve a system of linear equations despite the presence of errors in the equations.

When it comes to computational LWE problems there are two kinds of problems: *search* which has the goal of recovering a secret $s \in \mathbb{Z}_q^n$ given a certain number of samples drawn from the LWE distribution $A_{s,\chi}$; and *decision* which is to distinguish a certain number of samples drawn from the LWE distribution from uniformly random samples. For both of these variants of the problem, there are two distributions to consider. The first is the uniform distribution of the secret $s \in \mathbb{Z}_q^n$ and the distribution $\chi^n \mod q$ where each coordinate is drawn from the error distribution $\chi$ and reduced modulo q. This form is called the "normal form" of LWE [24].

## 2.6 FrodoKEM

FrodoKEM is a family of IND-CCA2 secure key-encapsulation mechanisms that are built to be conservative yet practical post-quantum constructions. IND-CCA2 or "indistinguishability under chosen-ciphertext attack" is a security property in cryptography that ensures an adversary cannot distinguish encryptions of two different plaintexts, even when having access to an "decryption oracle" capable of decrypting arbitrary ciphertexts [4].

The security of the mechanisms comes from the usage of LWE on algebraically unstructured lattices. The core building block of FrodoKEM is a public-key encryption scheme called FrodoPKE whose IND-CPA security is tightly related to the hardness of the LWE problem. IND-CPA or "indistinguishability under chosen-plaintext attack" is like IND-CCA2, a security property in cryptographic algorithms, but unlike IND-CCA2, IND-CPA means that if an adversary has access to two messages of the same length and once one of the messages is encrypted, the adversary has a hard time telling which one of the two messages was encrypted [4].

FrodoKEM further enhances this security by using a so called Fujisaki-Okamoto (FO) [11] transform to the existing FrodoPKE scheme, thus making it IND-CCA2 secure.

In FrodoKEM the key pair generation process involves the utilization of three distinct matrices: the secret matrix (S), the error matrix (E) and the public key matrix (A). The secret matrix is comprised of secret values known only to the key owner and serves as the foundation for the private key while the public key matrix is a pseudorandomly generated matrix of size $n \times n$ with coefficients in

$$\begin{bmatrix} e_{1,1} & \cdots & e_{1,8} \\ \vdots & \ddots & \vdots \\ e_{R,1} & \cdots & e_{R,8} \end{bmatrix}$$

**Figure 2.1:** Error matrix of the FrodoKEM algorithm, where $R \in \{640, 976, 1344\}$ and $e$ is a pseudorandomly generated 16-bit value.

$\mathbb{Z}_q$ [1]. Finally we have the error matrix, this consists of random noise values that are added to the public key to enhance its security. This enhancing of security is due to the error matrix aiming to obscure the relationship between the public and private keys, along with making it more computationally difficult to derive the private key from the public key. Additionally, it's worth noting that the dimensions of the error matrix depends on the parameters employed in FrodoKEM. However, the number of rows in the matrix is either, 640, 976 or 1344, while the number of columns is set to 8. A visual representation of the error matrix can be found in Figure 2.1.

The error matrix plays a crucial role in ensuring the security of the mechanisms behind FrodoKEM. However, if an attacker can somehow manipulate or modify the values in the error matrix, it has the potential of compromising the security of the system. The consequences of such an attack can be severe. By modifying the error matrix in specific ways an attacker may be able to derive the private key from the public key, allowing them to decrypt messages encrypted by the public key. This should be computationally difficult to do, but by manipulating the error matrix in specific ways, the relationship between the public key and private key is made more clear.

With its careful usage of LWE and leveraging its computational hardness FrodoKEM aims to create a secure and robust cryptographical system that is resilient to both classical and quantum attacks. Furthermore the design behind FrodoKEM uses conservative parameterization and aims to be quite secure while still remaining as simple to use as possible.

## 2.7 CRYSTALS-Kyber

CRYSTALS-Kyber, or simply Kyber is quite similar to FrodoKEM since it is also a family of cryptographic algorithms designed for post-quantum security. While both Kyber and FrodoKEM share similarities in their construction and reliance on the LWE problem, there are fundamental differences in their underlying mathematical structures. Unlike FrodoKEM, which is based on the standard LWE problem on algebraically unstructured lattices, Kyber instead uses the Ring-LWE problem, operating over the algebraic ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ instead of the finite field $\mathbb{Z}_q$. This algebraic structure offers Kyber several advantages, including efficient

polynomial arithmetic and modular reduction, critical for cryptographic constructions. Consequently, Kyber demonstrates a greater efficiency over FrodoKEM. By leveraging the inherent properties of the ring to facilitate faster cryptographic operations, allowing reduced computational overhead and improving performance in resource-constrained environments [3].

## 2.8 Understanding Rowhammer Vulnerabilities

In the upcoming sections, we aim to provide the reader with insight into the foundational concepts and theory that underlie the attack explored in this thesis. By shedding light on necessary key concepts required for the successful implementation of the attack, our objective is to offer a comprehensive understanding of the mechanisms underlying these sorts of attacks on RAM memory and why they are possible in the first place.

## 2.9 Rowhammer

*Rowhammer* is a memory exploit first presented in 2014 by Kim et al. [16]. By targeting physically adjacent memory addresses in RAM with rapid memory reads it is possible to have electrical charge "leak" from one memory cell to another. The effect of this is that physically adjacent bits have the potential of "flipping", i.e. changing their value from 0 to 1, or 1 to 0. This was the foundation for the attack on FrodoKEM presented by Fahr et al. [8].

The exploit in itself poses quite a significant security risk since it can lead to unexpected changes to data stored in memory, potentially compromising the integrity and security of systems and processes on a host. By using Rowhammer to induce bit flips in memory an attacker could manipulate sensitive information, undermine cryptographic protocols or even gain unathorized access to systems.

A graphical look at how Rowhammer works in memory banks can be found in Figure 2.3.

## 2.10 Dynamic Random Access Memory (DRAM)

The working memory of a computer—so-called *random access memory*, or RAM—is currently often of the *dynamic* type, namely *dynamic RAM*, or DRAM. This memory is split up into multiple *banks*, and these banks each contain a matrix-like formation of single bits (see Figure 2.2). Each of these bits consist of a capacitor and a transistor. The capacitor can be either discharged or charged, and these two states each correspond to the two possible values of a bit; 0 and 1 respectively. The operating system of a computer interacts with the RAM to perform several tasks essential for the operation of the computer. When a program is executed, its instructions and data are loaded into the RAM from storage devices like hard drives connected to the computer. This allows for faster access to the data compared to fetching it directly from the storage device.

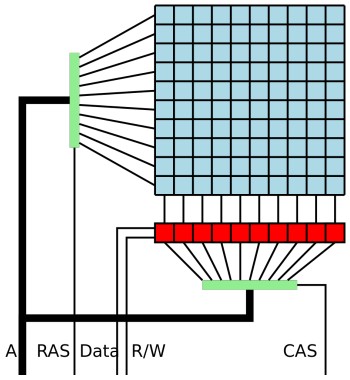


**Figure 2.2:** A model of a bank inside DRAM. Each cell corresponds to a single bit in memory [12].

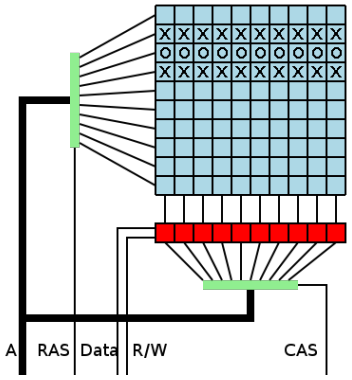


**Figure 2.3:** Rowhammer. Repeatedly accessing bits marked with X could potentially flip bits marked with O.

As the CPU processes instructions it reads and writes data to and from the RAM. This data transfer is facilitated by the RAMs random access nature hence the name RAM, allowing multiple memory cells to be accessed at the same time and in any order. Dynamic RAM needs to be periodically refreshed in order to preserve to data over time hence the term "dynamic". The refreshing of data is managed by the memory controller in the computer and is performed automatically requiring no intervention from the CPU or OS.

### 2.10.1 Memory Refresh Period

As stated, the memory in DRAM needs to be refreshed over time in order to keep data in the memory. The reason for this is that the electrical charge in the capacitors behind each memory cell quickly diminishes over time. To prevent data loss it therefore necessitates the existence of a *memory refresh circuit*, which periodically reads and rewrites the memory contained in the entire bank. By doing this, the electrical charge in each capacitor is restored to their fully charged or fully discharged state. The time between each refresh is called the *memory refresh period*. This period varies between models, but as per the JEDEC standard it's usually 64 ms or less [15]. The operation of the circuit involves accessing each row of the memory cells and reading the contents, followed by immediately writing the same data back. This effectively rejuvenates the charge stored in the capacitors of each memory cell, restoring them to their original state.

### 2.10.2 The Row Buffer

Each time a bit in a bank is read, the entire row containing that bit is put into a *row buffer*. This means that every time a cell is read the entire row is read as well, and subsequently, the entire row is refreshed. When reading other memory

close by one can access the currently checked out row instead of having to access individual bits of the RAM again.

### 2.10.3 Electrical Charge Leakage

Reading cells in a dynamic memory bank is a *destructive* action, i.e. reading a storage cell causes that cell to discharge, effectively wiping the memory stored. Therefore, each cell must be recharged after every read. When reading the cell there is also the phenomenon of electrical charge leakage to take into account. Since the charges are stored in capacitances the electrical charge needs somewhere to go once discharged. Because of the compact nature of modern RAM the memory cells are in close proximity to each other. This leads to electrical charge leakage becoming more and more of a problem and poses a challenge to the integrity of the data stored on RAM.

As stated, the stored electrical charge in a memory cell can leak into neighboring memory cells and have the potential of modifying the data contained within. This phenomenon is what lead to be the foundation of Rowhammer. One could successfully charge up neighboring cells by accessing a row multiple times until electrical charges in the accessed row dissipated into neighbouring rows. This without ever accessing the neighbouring row directly, i.e. allowing a user to modify bits in data without having direct access to them. This breaks a fundamental security rule of the modern memory model, where processes should not ever be able to read from or write to unauthorized memory.

## 2.11 Memory Massaging

For Rowhammer to be effective the attacking process needs to have allocated memory physically adjacent to memory of the victim process. This is achieved with *memory massaging*, and is done using a technique called "Frame Feng Shui", described by Kwong et al. [18].

This is achieved by manipulating how the memory is allocated in the operating system. In Linux, the memory allocator keeps a list of recently deallocated memory pages in a so-called *page cache*. This cache operates on a last-in-first-out basis, meaning, that when a program requests a memory page it will receive the most recently deallocated page [22]. By allocating a number of so-called "dummy pages" and then deallocating them right before the target process is run this behaviour can be exploited to control the position and order of the allocated memory pages.

To successfully execute this exploit, an attacker must first analyze the victim process to determine how many memory pages are allocated before the specific memory region they intend to target with Rowhammer. Once that information is obtained the attacker can utilize Frame Feng Shui to manipulate the memory allocation of the victim process. By allocating the number of needed pages before the specific memory region an attacker can force the target memory to be allocated at an arbitrarily chosen page.

## 2.12 Performance Degradation

To be able to succeed with the attack it is required that there is a large enough time window where the targeted memory is stored in memory. The amount of time needed for a successful attack is, in this case, far greater than the time available, as the attack window is about 8 ms while the attack itself needs around 530 ms to be successful close to 100% of the time. To increase the time window where the attack can be run a technique is used called *performance degradation*. This paper mainly uses the method described by Allan et al. [2].

The goal of using performance degrading techniques is to intentionally increase the execution time of the victim process. By deliberately impeding the victim processes execution one could increase the time window needed for the attack to be successful and increase the likelihood that the rowhammer attack has induced bitflips in the memory allocated to the victim. In this paper a technique is used where the instruction cache is repeatedly flushed, with the hope that the target process will spend exponentially more time fetching instructions.

# Chapter 3

# Rowhammering (Method)

This chapter presents a comprehensive methodology for implementing and executing the Rowhammer attack against the FrodoKEM cryptosystem. We outline the detailed steps involved in conducting the attack, including the selection of the target hardware and software. Our goal being to provide a detailed overview of the attack methodology, aiming to offer transparency and reproducibility of the implemented techniques, hopefully facilitating future findings and investigations into the implications and efficiency of Rowhammer as a potential attack vector.

## 3.1 Objective of the Attack

In the event of the Rowhammer attack on FrodoKEM being successful, the attacker's primary objective is to recover the private key generated by the compromised key-generation process. This process of key recovery is facilitated by the manipulation of the error matrix, which is a crucial component in obfuscating the relationship between the private and public keys.

The error matrix plays a vital role in FrodoKEM's key generation mechanism, introducing randomness to enhance security of the key pair by obfuscating the relationship between the private and public keys. However, if an attacker can manipulate the error matrix by inserting specific values through a Rowhammer attack, this security measure becomes compromised. Manipulating the error matrix effectively weakens the obfuscation process, exposing a more direct relationship between the private and public keys. By exploiting this vulnerability, an attacker can discern certain patterns and correlations that would otherwise remain obscured, making it significantly easier to derive the private key from the modified public key.

One potential technique an attacker might employ to extract the private key is a decryption failure attack. In this approach, the attacker attempts to decrypt ciphertexts using the modified public key and observes the failures in the decryption process. By carefully analyzing the patterns of successful and failed decryptions, the attacker can gradually extract information about the private key. This type of attack leverages the weakened obfuscation caused by the manipulation of the

error matrix, allowing the attacker to gain insights into the private key that would not be possible under normal circumstances.

More specific details on the key recovery process can be found in [8].

With the objective of the attack established, we now delve into the methodology used when executing the Rowhammer attack against the FrodoKEM algorithm.

## 3.2 Threat Model

Rowhammer attacks can have severe implications on the security of a system, potentially giving attackers unauthorized access to sensitive data or escalated privileges on a system. However, the consequences and impact of a Rowhammer attack depend on the threat model and the setup of the system the attack is performed on. In order to accurately assess the risks and potential consequences on our system from our implementation of the Rowhammer attack, it is crucial to define a threat model tailored to our specific system and the hardware running on it. By establishing this threat model, we can try to better understand the scope and limitations of our attack implementation, as well as the potential impact it may have on our system.

### 3.2.1 Inherent Requirements of Rowhammer Attacks

Many Rowhammer attacks share a common threat model, regardless of the specific setup of the system the attack is run on. These requirements can be defined as follows:

- The attacker is assumed to have unprivileged code execution on the target machine.
- The attacker process and the targeted process must run on the same physical hardware.
- The system must use DRAM memory (e.g., DDR3 or DDR4) susceptible to bit flips caused by repeated memory row activations.

### 3.2.2 Threat Model for Our Experimental Setup

In order to allow for an easier time when implementing the attack, time constraints, and our setup being experimental, we made the following assumptions and introduced some constraints:

- The target machine is running Ubuntu and the attacker and victim are running in two different processes.
- Address space layout randomization (ASLR) is disabled in order to simplify the implementation of the attack. While disabling ASLR may be an unrealistic assumption, it has been demonstrated that ASLR can be bypassed in certain scenarios [6], theoretically making Rowhammer attacks possible even with ASLR enabled.

- The attacker has root privileges (`sudo` access) to allow gathering of memory allocation information from `/proc/self/pagemap`. This information gathering could alternatively be accomplished using a technique like the one proposed by [23], eliminating the need for root access.

- The victim process is the FrodoKEM-640 key generation algorithm running with no compiler optimizations and using the reference code implementation. The goal being to induce a known error into each column of its error matrix (see 2.1).

We also want to bring to light that we are running the attack on DDR3 memory, which has been proven susceptible to Rowhammer attacks. However, it is also worth noting that attacks on more modern systems using DDR4 memory may yield larger effects and higher rates of bitflips. For instance, the attack proposed by Haocong Luo and Mutlu [13] uses a similar but different technique called Rowpress, and seems to be able to induce significantly more bitflips when compared to many Rowhammer attacks. This approach was tested on DDR3 memory but yielded no results at all. Nonetheless, we anticipate that using this approach instead of a regular Rowhammer attack on DDR4 could potentially give us a significant improvement to the effectiveness of our implementation. The Rowpress approach, while effective, comes with the downside of having to use `mlock`, which necessitates the usage of root privileges (`sudo`).

## 3.3 Setup

The computer the attack was performed on contained a 4GHz i7-4790K CPU and two Corsair DDR3 8GB 1333MHz non-ECC DIMMs (part number: CMZ16GX3M2A1600C10).

The OS that was running on the system was Ubuntu 22.04.2 LTS with some slight modifications. As mentioned earlier, address space layout randomization (ASLR) was disabled in order to simplify the attack. ASLR randomizes the memory locations of processes with the aim of making memory corruption attacks—like Rowhammer—harder to execute successfully. ASLR has previously been breached by, for example, Canella et al. [6], making it theoretically possible to design an attack that does not require ASLR to be disabled.

Furthermore, the attack process itself was also running as root. This was needed in order to gather information about the allocated memory from `/proc/self/pagemap`, as if the process is not running as root the relevant data in the file is zeroed [21]. There is, however, an attack by Tobah et al. [23] which instead uses the unrestricted `/proc/buddyinfo` file, theoretically eliminating the need for root access.

With our established threat model and the experimental setup of our system in mind we now present the method that was used in order to accomplish the attack.

## 3.4 Mapping Virtual to Physical Addresses

When interacting with the computer's working memory *virtual addresses* are used, which are free to be mapped to *physical addresses* in any way the operating system desires. Two adjacent virtual addresses are therefore *not* guaranteed to be physically adjacent, and because this is a requirement for Rowhammer to have any effect it presents a problem.

### 3.4.1 Memory Pages

When a program running in user space requests memory from Linux it is only ever given memory in 4KiB-sized chunks, so-called *memory pages*. These pages each have a *page frame number* (PFN), which is equal to its physical address divided by the page size in bytes, i.e. 4096.

Prior to Linux 4.0 this could be easily obtained via the virtual file `/proc/self/pagemap`, which is part of the Linux Process API. Since Linux 4.0, however, this information is—as a direct consequence of the discovery of Rowhammer—restricted to superusers by either zeroing the data out or by failing unauthorized processes attempting to access the pagemap file [21]. This is why the attack process needs to be run as root.

After obtaining the PFN, the physical address can then be easily calculated by multiplying the PFN by 4096.

### 3.4.2 Row Index

Each bank in a RAM stick has an amount of rows, and this is different for each type of RAM. For the RAM used in this paper the number of rows was $2^{16}$ ($= 65536$).

The row index was calculated by dividing the physical address by the row size in bytes, which in our case was $2^{18}$ ($= 262144$). This is practically the same as collecting the 14 most significant bits of the 32-bit physical address, i.e. bits 18-31 (inclusive, zero-indexed). This process allows us to precisely identify the row containing the targeted memory cells.

### 3.4.3 DRAM Bank Mapping

This analysis of the memory layout in the RAM is not enough to facilitate the execution of the attack in itself. Each brand of CPU has a unique way of mapping physical addresses to different memory banks in RAM. This is done to more evenly divide work between different banks, as they have a delay after reading or writing where they cannot be used. The bank index is obtained by XOR'ing different bits of the physical address together to obtain the different bits of the bank index.

This memory mapping is, unfortunately for us, proprietary and therefore not publicly available. However, thanks to Wang et al. [25] the exact sequence of XOR operations to obtain the bank index could be found. For our setup (Haswell North-

bridge with 2 DDR3 DIMMS of 16 GB total memory) the sequence was

$$\{\{14, 18\}, \{15, 19\}, \{16, 20\}, \{17, 21\}, \{7, 8, 9, 12, 13, 18, 19\}\}$$

meaning you XOR bit 14 and 18 of the physical address to obtain bit 0 of the bank index, XOR bit 15 and 19 to obtain bit 1, and so on.

## 3.5 Memory Profiling

Rowhammer is repeatable, meaning a bit that has flipped once is very likely to flip the same way again given the same circumstances [16]. This enables us to profile the computer's memory in advance in order to predict the exact locations of future bitflips with high accuracy. This is needed as the error induced must be known in order to apply a potential decryption failure attack on the cryptographic algorithm. To profile the RAM for bitflips the following approach was used.

Firstly, a large amount of memory was allocated using the syscall `mmap`. Then, the virtual address of each page mapped was collected, and—by using the data in `/proc/self/pagemap`—the physical address and row index of each page was obtained.

Once information for all pages in the mapping was collected the actual profiling was done by picking three physically adjacent rows and collecting all pairs of pages above and below the middle row which were in the same bank. By initializing the middle row to all zeroes and the outer rows to all ones, we could hammer rows in the same bank and check which bits had flipped, i.e. had a value of 1. This was used as a initial test to get a quick overview of which pages had a sufficient number of bitflips. More detailed tests were run later to determine the exact location of the flips.

## 3.6 Filtering Pages

After finding rows with sufficient amounts of bitflips we needed to find specific memory pages with bitflips in specific locations from our attack. Looking to the attack by Fahr et al. [8] they argue that the 8th bit—called a 256-bit—of each value in the matrix is a good target for the attack. According to the authors one 256-bit flip in each column of the error matrix is desired in order to achieve the highest chance of succeeding with the decryption failure attack while still remaining unnoticed. An illustration on what the error matrix might look like after a successful attack can be found in Figure 3.1

We therefore set a memory pattern in our attacking rows. This pattern consisted of a 1 in each 256-bit position for each halfword (2 bytes) in the pages on that row, with the rest of the memory zeroed (see Figure 3.2). The resulting flips in the error matrix can then be figured out, since if a flip was achieved in the matrix the 256-bit will be set to a value. Filtering the memory pages with this logic in mind resulted in 3 pages with 9 reproducible bitflips in total, all in the 256-bit spot.

| | Col 1 | Col 2 | Col 3 | Col 4 | Col 5 | Col 6 | Col 7 | Col 8 |
|---|---|---|---|---|---|---|---|---|
| $R_1$ | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $R_2$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** |
| $R_{50}$ | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 |
| $R_{170}$ | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 |
| $R_{400}$ | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 0 |
| $R_{500}$ | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 |
| $R_{630}$ | 0 | 0 | 0 | 0 | **1** | 0 | **1** | 0 |

**Figure 3.1:** Illustration of possible induced errors in the error matrix. The highlighted **1** in each column indicates a targeted 256-bit flip, with each induced bitflip ending up in a unique column.

| Index of bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Figure 3.2:** The set pattern for halfwords in the attacking rows (bit at index 8 set to 1 and all other bits set to 0).

## 3.7 Memory Massaging

To be able to modify the memory of the victim process exactly as we want, we need full control of where the process allocates its memory. More specifically, we need it to allocate the memory where the **E**-matrix is located in the pages chosen during the profiling stage. This is done via an exploitation of how the Linux page allocator works.

### 3.7.1 The Linux Page Allocator

The Linux page allocator works by keeping a list of all available memory blocks in a so-called *free list*. This list operates on a last-in-first-out basis, and so returns the most recently freed page whenever a page is requested. We can therefore place a requested page wherever we want if the victim process requests a predictable number of pages before requesting the target page.

### 3.7.2 Coercing the Location of the **E**-matrix

In order to place the **E**-matrix in our allocated victim pages the following approach was used. Let $n$ be the number of memory pages requested by the victim process before the pages used for the **E**-matrix are requested. The attack process first starts by allocating a large amount of pages. Afterwards $n$ pages from the large allocation are unmapped in order to place them at the top of the memory allocators free list, followed by the deallocation of the victim pages. Afterwards the attack process induces the victim process to run its algorithm. This will make the victim

process allocate its first $n$ memory pages in the large allocation, followed by the memory of the matrix being allocated in our chosen memory pages susceptible to bit-flips from Rowhammer. This will then allow us to run our Rowhammer implementation to induce bit-flips in the **E**-matrix.

The **E**-matrix in memory will be an array of arrays, in the form

$$\begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ \vdots \\ R_{640} \end{bmatrix}$$

where $R_i$ is an array of 8 16-bit values. The total memory footprint of the matrix is therefore $640 \times 8 \times 2 = 10240$ bytes, which is equivalent to $10240/4096 = 2.5$ memory pages. Therefore, the **E**-matrix will span 3 or 4 memory pages, depending on its offset within the first page. In our case, the **E**-matrix spanned 3 pages.

By carefully controlling the memory allocation process, the attack ensures that the error matrix is allocated in the three memory pages that was previously identified as being susceptible to bit-flips. This strategic allocation allows the attack to induce the necessary bit-flips in those specific memory pages, which will, in turn, enable the attacker to succeed by modifying the values of the error matrix in a controlled manner.

## 3.8 Performance Degradation

For the attack to be successful, a wide attack window was needed to be able to flip the targeted bits. For our implementation of the attack and on the hardware in question we needed about around 530 ms in order for the Rowhammer attack to be successful. Unfortunately, the reference implementation of the FrodoKEM algorithm and the time needed for allocation of the **E**-matrix takes about 8 ms. This timing of the attack window was without using compiler optimization and building the FrodoKEM implementation with the setting "*OPT_LEVEL=REFERENCE*". Because of this short attack window and us needing more time to achieve our bit-flips we therefore needed to run a performance degradation on the FrodoKEM process in order to make the timing window large enough for our attack to be successful.

In our case the performance degradation attack used involved finding the most frequently executed instruction in the code and flushing the cacheline associated with that instruction using the CLFLUSH instruction as presented by Allan et al. [2]. Analyzing the FrodoKEM algorithm and the key generation process found that the usage of a Shake-128 function involves a loop that frequently executes a store64 instruction. Targeting that instruction and using CLFLUSH on the offset associated with the instruction made it then possible to increase the attack window enough for our attack to be effective. When running the degradation process we ran the degradation on 4 different processor cores in total. One running on the

same core as the FrodoKEM process, one on the sibling core of the FrodoKEM process and two on other virtual cores. This gave us an attack window of around 1300ms leaving us with more than enough room for our attack to be successful and induce the bitflips in the error matrix.

## 3.9 Running the Attack

With every detail needed for the attack in place, it's time for the most crucial part: executing the attack. First we need to run the performance degradation and leave it running for the entirety of the attack. Once that's done, we also need to run the actual implementation of the attack. This is done with the command `attack` along with a sufficient amount of memory (measured in percent of total RAM) and the amount of "dummy" pages that was found in section 4.6.2. When the attack program is up and running, it first allocates the required amount of memory until it has the victim pages in its memory space. Afterwards, it collects data about the required dummy pages needed in the attack, followed by forking to spawn two different processes on the system.

The first process starts by going through all the dummy pages and setting a value at their addresses to ensure they remain in memory and are not removed by the memory controller. Afterwards, it runs the a Rowhammer attack on each of the victim pages until instructed to stop. The second process first deallocates all the dummy pages, followed by deallocating the victim pages that will be targeted for bit-flips to make sure that the memory controller gives these deallocated pages to the victim process. Once this is done, it induces the system to run the FrodoKEM key-generation algorithm. This causes the algorithm to start generating its keys and allocate the memory for its error matrix in our victim pages. Once the FrodoKEM process is generating its key-pair, the other process running the Rowhammer attack continuously tries to induce bit-flips in the victim pages where the error matrix is placed, allowing us to induce extra errors in the matrix.

After the FrodoKEM key-gen algorithm is done, we stop the processes running the attack, perform some cleanup, and the attack is complete. Hopefully, we have managed to induce bit-flips in the error matrix.

## 3.10 Checking for an Induced Error in the $\mathbf{E}$-matrix

Once the attack has finished its execution and the FrodoKEM key-gen algorithm is complete, it is important to verify the actual results. This was done by using the modified FrodoKEM code provided by Fahr et al. [8], enabling the printing of the entire error matrix of the generated key pair. Given that the victim in the attack was the FrodoKEM-640 key generation algorithm, the size of the error matrix was $640 \times 8$ of `uint16_t` values. This allowed for the utilization of a simple Python script to inspect all integer values in the error matrix for the presence of a set 8th bit, resulting in an incremented value of 256. Since our attack targeted the 8th

bit in every halfword[1] contained in the victim pages, the presence of a bit-flip was therefore confirmed if any integer value in the error matrix had its 8th bit set to 1.

The finalization of the method then primarily involves determining the locations within the error matrix where the induced bit-flips occurred. By following the steps outlined earlier, we have successfully implemented and executed an attack aimed at inducing bit-flips in the error matrix of the public key associated with the FrodoKEM-640 algorithm.

[1] Halfword = 16 bits

# Chapter 4

# Results

With everything established and running our implementation written in Rust managed to induce errors into the error matrix used in the public key for FrodoKEM. Our implementation of the attack required around 530ms to induce the desired bit-flips in our victim pages. On our system the attack window (when the **E**-matrix remains in memory and is not yet added to the public key) was found to be around 8ms. This attack window was too short for our attack to have any chance of inducing bit-flips and for this reason we also employed a performance degradation attack. By employing this attack the attack window was increased from 8ms to around 1300ms, giving us the appropriate amount of time to attack the FrodoKEM keygen algorithm and induce our bit-flips in the **E**-matrix.

For our machine, the victim pages were chosen to be 0x3b4bf1, 0x3dd31e, and 0x400b3a. These pages gave us 7 desired flips without flipping any other bits very close to 100% of the time. Each of these bits needs to be 0 at the start of the attack for the flip to have any effect, as the attack can only flip bits from 0 to 1. Assuming the probability for a bit being either 0 or 1 is the same we can conclude the expected success rate for the attack to be 1 out of every $2^N$ attempts, where $N$ is the number of targeted columns in the error matrix. This gave us a success rate of $1/2^8 = 1/128$.

Furthermore, our investigation revealed several mapped memory pages having an inherent susceptibility to bit-flips, as depicted in Figure 4.1. This finding suggests that while individual pages may not exhibit a high number of bit-flips, they are distributed relatively evenly across the entirety of the RAM. This distribution of bit-flips further underscores the importance of mitigating Rowhammer vulnerabilities in RAM in order to maintain system stability and data integrity.

When it comes to our investigation of attacking Kyber, our implementation faced challenges hindering its effectiveness against the algorithm. Several factors contributed to this outcome, with a significant obstacle being the optimized and robust implementation of Kyber compared to FrodoKEM. Kyber's optimization gives it a superior performance, rendering it significantly faster and consequently less susceptible to performance degradation attacks. This obstacle thwarted our attempts to slow down the algorithm sufficiently to enable Rowhammer to induce bit-flips

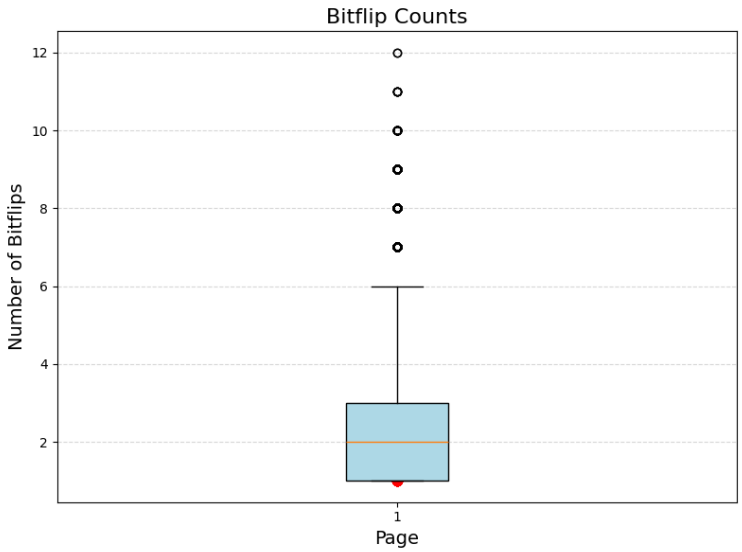


**Figure 4.1:** Box plot illustrating the distribution of bitflip occurrences in mapped memory pages, showcasing median values, interquartile ranges, and potential outliers.

effectively. Thus it was deemed not possible in the time frame of the thesis to successfully attack Kyber, this does not make it impossible however, as we will try to discuss further.

To foster deeper exploration into Rowhammer-related vulnerabilities, we are committed to transparency and collaboration. As such, we will be sharing our attack implementation on GitHub, providing a valuable resource for researchers to delve into this critical area of study. By making our work openly accessible, we aim to catalyze further investigations and contribute to the ongoing efforts to understand and mitigate Rowhammer threats.

The code is available on https://github.com/CodeyBoi/kyber-not-it.

# Chapter 5

# Discussion

With the continuous increase in RAM capacities, the possibility of read disturbance errors in RAM will probably not cease to exist anytime soon. With every new generation of RAM we see new attempts to try to mitigate the effects of these vulnerabilites on RAM, but more often than not these mitigations are circumvented by new attack techniques and methods. [10, 13]. In light of these challenges, the observed distribution of bit-flip occurrences in Figure 4.1 underscores the inherent vulnerability of memory cells to Rowhammer-induced bit-flips. The figure reveals a concerning pattern of bit-flips across various memory pages, indicating a systemic susceptibility to Rowhammer attacks. Furthermore, this inherent vulnerability poses significant risks to the reliability of systems and data integrity.

Our efforts in implementing the attack have yielded us significant progress, particularly in our ability to target FrodoKEM effectively. By aiming to replicate the results outlined in Fahr et al. [8], we successfully managed to induce 256 bit-flips across 7 out of 8 columns of the error matrix. Although we did not go as far as to actually produce failing ciphertexts and in turn completing the decryption failure attack, our replication of the induced errors in the error matrix signals the correctness of our attack implementation for FrodoKEM. This achievement underscores the potential effectiveness of our approach and furthermore demonstrates the feasibility of using Rust as the programming language for such attacks. Despite the necessity of having to employ some `"unsafe"` calls in our code, the majority of it still remains memory-safe and more comprehensible than one written in C.

However, transitioning our attack to instead target Kyber presented us with more challenges than we initially anticipated. Kyber's optimized and robust implementation posed a significant obstacle, as its efficiency minimizes the window of opportunity to induce significant bit-flips in the error matrix. Compared to FrodoKEM, Kyber is almost 85 times faster [17]. This made it considerably harder to achieve a successful performance degradation of the process. As mentioned earlier, for our implementation of a performance degradation attack we targeted the `store64` operation in the binary. Using this approach against the implementation of Kyber yielded us close to no results and only slowed down the process to a fraction of

what was actually needed. It therefore appears that employing a more efficient and drastic performance degradation attack against Kyber may be necessary. However, given that the key generation is already very fast, achieving a sufficient level of performance degradation without potentially alerting the user of the system's compromise seems implausible. This suggests that while a more efficient performance degradation may be required, it comes with the downside of potentially raising suspicion from the user of the targeted system. Balancing effectiveness with stealth in attacks like these then becomes crucial to be able to maintain the integrity of the overall strategy.

Working with our implementation, we faced numerous challenges that required meticulous attention to detail and extended periods of troubleshooting. One such challenge involved identifying memory regions susceptible to bit-flips, a process that proved to be exceedingly time-consuming and tedious. Finding the number of allocated memory pages of the victim process also proved to be a difficult task as we did not find any fool-proof way to do it. Therefore we had to go through many numbers of allocated "dummy" pages and look at the error matrix of FrodoKEM to see if our attack successfully induced any bit-flips, a process that took a long time to complete.

Looking back, some things could have been optimized and streamlined in order to efficiently develop our implementation. Developing efficient methods for identifying vulnerable memory regions and accurately mapping the allocated pages of the victim process could have improved the workflow immensely but no efficient way of doing this was found during the implementation phase. We explored various techniques to try to gather this vital information, such as analyzing memory dumps and leveraging debugging tools, but none of them provided us with a fool-proof solution, forcing us to resort to a time-consuming trial-and-error approach.

Worth noting is that performing an attack like the one described is a complicated task and not quite as feasible practically as it is theoretically. While our implementation managed to successfully induced bit-flips in the error matrix of FrodoKEM, translating these kinds of attacks into a genuine concern in realistic environments poses substantial practical challenges. The attack's success relied on specific conditions, such as disabling ASLR and requiring root privileges to facilitate gathering information on memory allocation. Additionally, the necessity for a performance degradation attack to extend the attack window highlights the difficulties in targeting optimized and high-performance cryptographic implementations like FrodoKEM.

In practical settings where ASLR is enabled and root access is restricted, executing a Rowhammer attack like this one becomes significantly more complex and potentially infeasible. Furthermore, the estimated success rate of 1 in 128 attempts in our controlled environment may be lower in real-world scenarios due to additional security measures, unpredictable system behaviors and the inherent variability of the Rowhammer phenomenon.

It's also important to acknowledge the limitations of our study, including the controlled nature of our experimental setup and the potential variability of the

results in different hardware and software configurations. By recognizing these limitations, we can better understand the scope and implications of our findings and work towards more robust and comprehensive solutions for protecting against Rowhammer attacks in practical environments.

Overcoming these practical hurdles and developing more advanced techniques would be crucial to realizing the potential vulnerability of FrodoKEM and other cryptographic implementations to Rowhammer attacks in genuine operational environments.

Executing Rowhammer attacks effectively requires a deep understanding of the low-level system mechanics involved in memory management and processor operations. Specifically, a comprehensive grasp of virtual-to-physical address translation mechanics, memory caching policies and the intricate interactions between hardware and software components is crucial. Without this knowledge, it becomes exceedingly difficult to pinpoint the exact memory regions that are susceptible to bit-flips and to accurately predict the potential impact of induced bit-flips on the target algorithm's execution.

Our choice of Rust as the implementation language also proved to be quite a viable option, offering a good balance between memory safety and performance. While our progress with FrodoKEM was promising, addressing the challenges posed by Kyber will require further refinement of our approach and an exploration of potential strategies to overcome its robust defences.

## 5.1 Future Research

In this section we will discuss the potential future discoveries, and other factors which has not been further explored in this paper.

### 5.1.1 DDR3 vs DDR4

In our system the type of RAM used was DDR3. DDR3 lacks some of the defences implemented in DDR4 to directly combat Rowhammer-related vulnerabilities – such as Target Row Refresh (TRR)—and is therefore easier to design an attack for. In a paper by Frigo et al. [10] a method for bypassing TRR was presented, called TRRespass. This was furthermore extended by Jattke et al. [14] to positive results. TRRespass makes use of *non-uniform attack patterns* which bypasses the TRR's detection of the attack. This is bad news, as the memory cells in DDR4 are much closer together (to accommodate for more memory) and therefore more vulnerable to memory corruption attacks. This make DDR4 a good candidate in future investigations into Rowhammer-related vulnerabilities.

Furthermore this also points to a more general problem. As memory cells gets smaller and more tightly packed it increases the effect of discharging the electricity to the nearby cells, which means that RowHammer likely will become even more of a threat as time goes on. Developing an efficient counter-measure for this is difficult, as any monitoring at the hardware level of, for example, the number or

sequence of recent memory accesses introduces overhead, lowering performance.

### 5.1.2 Rowpress

During the development phase of our implementation, we wanted to explore the potential of Rowpress [13] as an alternate attack method. The reason being that Rowpress seems to have a huge potential in increasing the amount of induced bit-flips in memory compared to Rowhammer. However, our experimentation revealed that it did not have a significant effect on DDR3 memory as opposed to DDR4 memory that it was created for. Furthermore, employing Rowpress necessitates the use of the `CLFLUSHOPT` instruction, unfortunately for us our system's hardware constraints makes this instruction unavailable, as it was first introduced with the Intel *Broadwell* architecture. Consequently, we were therefore unable to use Rowpress in our implementation.

Despite our inability to employ Rowpress in our attack, it's potential still remains substantial. Rowpress has demonstrated a capability to drastically reduce the minimum number of required aggressor row activations to induce at least one bit-flip ($AC_{min}$). In some instances, this requirement was observed to be as low as a single activation, albeit with the caveat of having to keep the aggressor row open for 30ms [13]. This underscores the efficiency and potential effectiveness of Rowpress in inducing bit-flips.

Furthermore with the introduction of TRR on DDR4 memory, Rowpress faces the challenge of having to circumvent these mitigations. To overcome this hurdle Rowpress employs a strategy utilizing specific memory access patterns. These patterns involve accessing a number of "dummy" aggressor rows along with the real aggressor rows. The goal being to trick the TRR mechanism to identify only the dummy rows and not the real aggressor rows.

Despite the mitigations posed by mechanisms like TRR, the impact of the Rowpress attack remains significant. As such, for systems utilizing DDR4 memory, we consider the use of Rowpress the preferred attack method to potentially achieve desired results.

### 5.1.3 Attacking Kyber

As mentioned earlier, Kyber stands out for its robustness and highly optimized implementation. This efficiency is exemplified by its key generation process, which is approximately 85 times faster when compared to FrodoKEM [17]. This makes it apparent that one might need to employ a more rigorous performance degradation attack in order to actually be able to induce bit-flips in the error matrix of Kyber. Attacking Kyber from our perspective might be fully feasible if one were to overcome this hurdle. But then one could argue about the relevance of the vulnerability for Kyber seeing that it might be fast enough to not be susceptible to Rowhammer attacks.

While Kyber's speed may offer some resistance to traditional Rowhammer attacks, it's important to acknowledge that susceptibility to other variants, such as

Rowpress attacks, still remains a possibility. Even though Kyber may not be as vulnerable to standard Rowhammer techniques, the emergence of alternative attack methods like Rowpress underscores the ongoing need for vigilance in assessing and addressing security vulnerabilities across cryptographic algorithms.

# Chapter 6

# Conclusion

In this thesis our primary objective was twofold. Firstly we aimed to implement the attack detailed in [8] using the Rust programming language. Secondly we sought to extend and adapt our implementation of the attack to target Kyber. This was further motivated by the persistent nature of Rowhammer vulnerabilities, which are likely to remain a significant concern in RAM due to the close proximity of memory cells.

While we successfully managed to implement the attack using Rust we encountered some challenges when attempting to accomplish the second goal. Despite our best efforts this objective was not fully achieved within the scope of this thesis. Nonetheless, the insights gained from our implementation provide some valuable lessons for future research in Rowhammer-related vulnerabilities.

Rowhammer, as a phenomon, is still quite new in the research world. Despite continuous efforts to mitigate these vulnerabilities, it is evident that they will persist as a significant concern in RAM, particularly as memory technologies used in computers continue to advance and memory densities increase. While a complete removal of these risks might not be achievable, it remains crucial to deepen our understanding of these vulnerabilities and to continue exploring effective ways of mitigating them.

# References

[1] Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. Frodokem, learning with errors key encapsulation, algorithm specifications and supporting documentation. `https://frodokem.org/files/FrodoKEM-specification-20210604.pdf`, 2021.

[2] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 422–435, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450347716. doi: 10.1145/2991079.2991084. URL `https://doi.org/10.1145/2991079.2991084`.

[3] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber algorithm specifications and supporting documentation (version 3.02), 2021. URL `https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf`.

[4] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography, Chapter 5: Symmetric Encryption, 2005. URL `https://web.cs.ucdavis.edu/~rogaway/classes/227/spring05/book/main.pdf`.

[5] Daniel J. Bernstein. *Introduction to post-quantum cryptography*, pages 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-540-88702-7. doi: 10.1007/978-3-540-88702-7_1. URL `https://doi.org/10.1007/978-3-540-88702-7_1`.

[6] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. Kaslr: Break it, fix it, repeat. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 481–493, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367509. doi: 10.1145/3320269.3384747. URL `https://doi.org/10.1145/3320269.3384747`.

[7] Sofía Celi and Goutam Tamvada. Deep dive into a post-quantum key encapsulation algorithm, 2022. URL https://blog.cloudflare.com/post-quantum-key-encapsulation.

[8] Michael Jr. Fahr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, and Daniel Apon. When frodo flips: End-to-end key recovery on frodokem via rowhammer. Cryptology ePrint Archive, Paper 2022/952, 2022. https://eprint.iacr.org/2022/952.

[9] Federal Office for Information Security (BSI). Migration to post quantum cryptography, recommendations for action by the bsi, 2020. URL https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Crypto/Migration_to_Post_Quantum_Cryptography.pdf?__blob=publicationFile&v=2.

[10] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Trrespass: Exploiting the many sides of target row refresh, 2020.

[11] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999. doi: 10.1007/3-540-48405-1_34.

[12] HandigeHarry. Dynamic ram, March 5 2008. URL https://commons.wikimedia.org/wiki/File:DRAM.svg. Public Domain.

[13] Abdullah Giray Yağlıkcı Yahya Can Tuğrul Steve Rhyner M. Banu Cavlak Joel Lindegger Mohammad Sadrosadati Haocong Luo, Ataberk Olgun and Onur Mutlu. RowPress: Amplifying Read Disturbance in Modern DRAM Chips. In *ISCA*, 2023.

[14] Patrick Jattke, Victor Van Der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 716–734, 2022. doi: 10.1109/SP46214.2022.9833772.

[15] JEDEC. JEDEC Manual of Organization and Procedure. https://web.archive.org/web/20060308100413/http://www.jedec.org/Home/manuals/jm21l.pdf, 2002.

[16] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014. doi: 10.1109/ISCA.2014.6853210. URL https://ieeexplore.ieee.org/document/6853210.

[17] Manish Kumar. Post-quantum cryptography algorithms standardization and

performance analysis, 2022. URL https://arxiv.org/ftp/arxiv/papers/2204/2204.02571.pdf.

[18] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711, 2020. doi: 10.1109/SP40000.2020.00020. URL https://ieeexplore.ieee.org/document/9152687.

[19] National Institute of Standards and Technology (NIST). Post-Quantum Cryptography Standardization. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization, 2022.

[20] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), sep 2009. ISSN 0004-5411. doi: 10.1145/1568318.1568324. URL https://doi.org/10.1145/1568318.1568324.

[21] The Kernel Development Community. Examining Process Page Tables. https://www.kernel.org/doc/html/v4.18/admin-guide/mm/pagemap.html, 2022.

[22] The Linux Kernel Organization. Physical Page Allocation. https://www.kernel.org/doc/gorman/html/understand/understand009.html.

[23] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. Spechammer: Combining spectre and rowhammer for new speculative attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 681–698, 2022. doi: 10.1109/SP46214.2022.9833802. URL https://ieeexplore.ieee.org/document/9833802.

[24] Vinod Vaikuntanathan. Cs 294: The learning with errors problem: Introduction and basic cryptography. Course notes for CS 294-5: Learning Theory and Algorithms, 2005. URL https://people.csail.mit.edu/vinodv/6876-Fall2018/lecture1.pdf.

[25] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. Dramdig: A knowledge-assisted tool to uncover dram address mapping. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, DAC '20. IEEE Press, 2020. ISBN 9781450367257. URL https://arxiv.org/abs/2004.02354.