UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Built on Sand: Exploiting Rowhammer for a Universal Forgery Attack Against the Post-Quantum Signature Scheme SPHINCS$^+$

Auf Sand gebaut: Ausnutzung von Rowhammer für einen Universal-Forgery-Angriff auf das Post-Quantum-Signaturschema SPHINCS$^+$

Masterarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Jeremy Boy

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Luca Wilke, M. Sc.

Lübeck, den 20. Juni 2025

Abstract

Modern computing systems increasingly rely on layered defenses – employing cryptographic algorithms, operating system isolation, and hardware hardening – to guarantee authenticity, integrity, and confidentiality in the presence of sophisticated adversaries. However, as the densities of integrated circuits increase, new ways to subvert these defenses by exploiting physical characteristics of the hardware itself are made possible. One particular example is the Rowhammer bug: by repeatedly accessing specific rows in a DRAM chip, an attacker can induce bit flips in adjacent rows. Exploiting this bug is not straightforward, as many layers of the target system need to be considered, including the operating system, the memory controller, and the DRAM chip itself.

Besides hardware bugs, the security of cryptographic systems is also under scrutiny by quantum computing. Many cryptographic algorithms are based on the hardness of mathematical problems that can be solved efficiently by quantum computers, such as factoring large integers or computing discrete logarithms. Therefore, the cryptographic community is shifting towards developing post-quantum cryptographic algorithms that provide security guarantees against known quantum attacks.

In this thesis, we investigate the security of hash-based post-quantum signature scheme SPHINCS$^+$ against Rowhammer-based fault attacks. We introduce SWAGE, a novel end-to-end framework for Rowhammer attacks, and use it to demonstrate that the Rowhammer bug can be exploited to conduct a universal forgery attack against SPHINCS$^+$. To the best of our knowledge, this is the first work that demonstrates a practical Rowhammer attack against SPHINCS$^+$.

Zusammenfassung

Moderne Computersysteme verlassen sich zunehmend auf mehrschichtige Verteidigungsmechanismen, welche kryptographische Algorithmen, Betriebssystemisolierung und Hardwaresicherung einsetzen, um Authentizität, Integrität und Vertraulichkeit in Anwesenheit hochentwickelter Angreifer zu gewährleisten. Mit zunehmender Dichte integrierter Schaltkreise werden jedoch immer neue Wege gefunden, diese Verteidigungsmaßnahmen durch Ausnutzung physikalischer Eigenschaften der Hardware zu unterlaufen. Ein besonderes Beispiel ist der Rowhammer-Bug: Durch wiederholten Zugriff auf bestimmte Zeilen in einem DRAM-Chip kann ein Angreifer Werte in benachbarten Zeilen verändern. Das Ausnutzen dieses Fehlers ist allerdings nicht einfach, da viele Schichten des Zielsystems berücksichtigt werden müssen, einschließlich des Betriebssystems, des Speichercontrollers und des DRAM-Chips selbst.
Neben Hardware-Fehlern wird die Sicherheit kryptographischer Systeme auch durch Quantencomputing auf den Prüfstand gestellt. Viele kryptographische Verfahren beruhen auf der Schwere mathematischer Probleme, die jedoch von Quantencomputern effizient gelöst werden können, wie zum Beispiel die Faktorisierung großer Zahlen oder die Berechnung diskreter Logarithmen. Daher konzentriert sich die kryptographische Gemeinschaft zunehmend auf die Entwicklung kryptographischer Post-Quantum-Algorithmen, welche Sicherheitsgarantien gegen bekannte Quantenangriffe bieten.
In dieser Arbeit untersuchen wir die Sicherheit von SPHINCS$^+$, einem Hash-basierten Post-Quantum-Signaturverfahren, gegen Rowhammer-basierte Fehlerinjektionsangriffe. Wir stellen SWAGE vor, ein neuartiges End-to-End-Framework für Rowhammer-Angriffe. Mithilfe von SWAGE zeigen wir die praktische Durchführbakeit von Rowhammer-basierten Angriffen gegen SPHINCS$^+$ und demonstrieren einen Universal-Forgery-Angriff. Nach unserem besten Wissen ist dies die erste Arbeit, die einen praktischen Rowhammer-Angriff gegen SPHINCS$^+$ demonstriert.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 20. Juni 2025

Acknowledgements

Firstly, I would like to thank my advisor, Thomas Eisenbarth, for his support and guidance throughout this project. His expertise in cryptography and hardware security was invaluable, and his ability to connect me with the right people significantly contributed to the progress of this work. I am also deeply grateful to Luca Wilke, who, despite the demands of concluding his dissertation and work-related travel, generously took the time to assist me and offer thoughtful advice. Special thanks go to Antoon Purnal for his vital assistance and clear explanations, particularly regarding fault attacks against SPHINCS$^+$. I would also like to express my gratitude to Paula Arnold and Anna Pätschke for their careful proofreading of this thesis and their extensive feedback, which greatly improved the quality of this work. Finally, I want to thank Pajam Pauls and Tim Gellersen for the insightful late night discussions, Jan Wichelmann for taking work off my shoulders during the final stages of this project, and all my colleagues and friends at the Institute for IT Security for providing a productive and stimulating work environment.

Contents

1 Introduction

Sophisticated attacks against computing systems have become increasingly prevalent, with adversaries leveraging both software vulnerabilities and hardware characteristics to compromise security. These attacks can range from exploiting bugs in operating systems and applications to manipulating the physical properties of hardware components. To tackle these threats, modern computing systems employ layered defenses, which include cryptographic algorithms, operating system isolation, and hardware hardening techniques. These defenses are designed to guarantee authenticity, integrity, and confidentiality in the presence of sophisticated adversaries. At the same time, manufacturers pack more and more transistors into a single chip, leading to increased density of integrated circuits. This trend, while beneficial for performance, efficiency, and cost, also opens up new avenues for attacks that exploit the physical characteristics of hardware itself. One particular example is the Rowhammer bug: by repeatedly accessing (*hammering*) specific rows in *Dynamic Random Access Memory* (DRAM), an unprivileged attacker can induce bit flips in adjacent (inaccessible) memory rows, effectively bypassing software and hardware protections and compromising security assumptions. Since its discovery in 2014 [Kim et al., 2014], the Rowhammer bug has grown from a reliability issue to a lasting practical threat against widely deployed systems – undermining everything from privilege separation in operating systems to cryptographic key material stored in hardware enclaves.

At the same time that the Rowhammer bug became a growing interest for security researchers, the cryptographic community has also shifted towards designing cryptographic primitives secure against quantum attackers. Hash-based signature schemes have emerged as a leading candidate for post-quantum secure signatures. Most notably, the SPHINCS$^+$ signature scheme [Bernstein et al., 2019, Aumasson et al., 2022] has been standardized by NIST in 2024 as *Stateless Hash-Based Digital Signature Algorithm* (SLH-DSA) [NIST, 2024] as part of the post-quantum cryptography standardization process. However, while SPHINCS$^+$ is designed to withstand quantum-based attackers, it is also highly susceptible to *fault attacks*, where an adversary injects faults to manipulate the execution flow of cryptographic operations. To date, there has been no publicly documented, end-to-end demonstration of Rowhammer attacks against SPHINCS$^+$. Closing this gap is critical: if post-quantum schemes can be broken by practical fault attacks, they might be rendered ineffective in practice.

A primary barrier in mounting a realistic Rowhammer-based attack is the lack of robust, publicly available tooling that automates the low-level steps required in the pipeline to a successful attack. Many existing prototypes are academic proof-of-concepts or only solve a particular step in this pipeline. Without a systematic framework to orchestrate real-world fault attacks, researchers and defenders alike are in the dark about the feasibility of new attacks or countermeasures. In this thesis, we address that tooling gap by introducing SWAGE, a novel, modular end-to-end framework that automates reverse-engineering the physical DRAM address mapping, employs techniques to allocate physically contiguous memory required for an attack, and aids in finding susceptible memory regions in a victim program. By packaging every step needed for a real-world Rowhammer attack, we not only demonstrate the first Rowhammer-based fault attack against the SPHINCS$^+$ signature scheme, but also provide a foundation for future research to design countermeasures and novel attacks.

1.1 Background

The Rowhammer effect was first discussed by [Kim et al., 2014]. The authors showed that by repeatedly and quickly accessing rows in a DRAM module, they could induce bit flips in adjacent rows. This new class of hardware bug was coined as *memory disturbance errors*. Rowhammer attacks were first believed to be mostly a reliability issue, as a real world exploit appeared to be hard to achieve. Just a few months later, [Seaborn and Dullien, 2015] showed that the Rowhammer effect can be practically exploited to gain kernel privileges. Many researchers have shown that the Rowhammer effect can be exploited in real-world scenarios, and many attack vectors have been discovered. For example, [Gruss et al., 2016] show that the Rowhammer effect can be exploited in JavaScript to break out of the browser sandbox. This is a significant discovery, as previous Rowhammer attacks relied on native code execution and access to low-level memory interfaces.

Since then, manufacturers implemented several mechanisms coined under the umbrella term *Target Row Refresh* (TRR) to mitigate Rowhammer attacks in DDR4 modules. TRR is a black box mechanism that heuristically refreshes the charge in suspected target rows. For some time, it was believed that TRR would conclusively mitigate Rowhammer attacks, even though researchers discovered that bit flips are still present in some DDR4 modules [Gruss et al., 2018, Lipp et al., 2020]. However, [Frigo et al., 2020] demystifies TRR and present TRRESPASS, a fuzzing-based tool to find TRR-aware access patterns that enables Rowhammer attacks against 13 of 42 tested DRAM modules by all major manufacturers. In [Jattke et al., 2022], BLACKSMITH is introduced, filling the gaps left behind by TRRESPASS, presenting a scalable fuzzer for *non-uniform* Rowhammer access patterns.

BLACKSMITH fuzzes memory access patterns in the frequency domain, enabling the use of non-uniform hammering patterns to circumvent sophisticated TRR mechanism. This approach enables rowhammer-induced bit flips in all the 40 tested DRAM modules.
Regarding the security of post-quantum signature scheme SPHINCS$^+$, the susceptibility to fault attacks has been well studied. In [Castelnovi et al., 2018], the *grafting tree attack* is introduced, a fault attack against SPHINCS (the predecessor of SPHINCS$^+$) that allows an attacker a universal forgery attack against the scheme. In the same year, [Genêt et al., 2018] presented a practical fault injection attack against SPHINCS, demonstrating that the scheme is vulnerable to fault attacks. While both works discuss the feasibility of the grafting tree attack against SPHINCS$^+$, only [Genêt, 2023] presents a practical attack against SPHINCS$^+$.

1.2 Contributions

This thesis makes the following contributions to the fields of fault attacks and post-quantum cryptography:

SWAGE, a Modular End-to-End Framework For Rowhammer Attacks. SWAGE unifies the steps required to carry out end-to-end Rowhammer attacks under real world threat models. It provides an easy and well-documented API such that new attack vectors can be easily integrated into the code base.

First Rowhammer Attack Against SPHINCS$^+$. While the susceptibility of SPHINCS$^+$ to fault attacks is well studied, a discussion of the feasibility of Rowhammer-based attacks against the scheme was still missing. We close this gap by using SWAGE to orchestrate a Rowhammer attack against SPHINCS$^+$ on a modern system employing Rowhammer countermeasures, such as TRR and OS-based access restrictions.

Taken together, this thesis underlines the necessity of robust, extensible tooling to close the gap between theory and practice for sophisticated Rowhammer attacks, as well as the need for robust countermeasures against hardware-based attacks. We employ SWAGE as a high-quality research artifact in the study of hardware-based attacks.

1.3 Structure of this Thesis

This document is structured as follows:

Chapter 2: Preliminaries We start the discussion by giving a background on the topics covered in this thesis. This chapter establishes the technical foundation required

for our end-to-end attack. After introducing required notation, the necessary background on DRAM architecture, virtual memory management, and fault attacks, with a focus on Rowhammer attacks is given (Section 2.2). In Section 2.3, the components of the hash-based post-quantum signature scheme SPHINCS$^+$– *Winternitz One-Time Signature Scheme*$^+$ (WOTS$^+$), *eXtended Merkle Signature Scheme* (XMSS), *Forest of Random Subsets* (FORS), and the SPHINCS$^+$ hypertree – are introduced. The preliminaries are concluded by Section 2.4 with a discussion of the *grafting tree attack*, a well-studied fault attack against SPHINCS$^+$.

Chapter 3: SWAGE: An End-to-End Framework for Rowhammer Attacks We present SWAGE, a modular toolkit integrating the building blocks required to conduct an end-to-end Rowhammer attack. This chapter introduces the components of SWAGE and how they interact with each other. Each section discusses strategies to address different challenges of conducting Rowhammer attacks in practice. Section 3.1 introduces the DRAMINSPECTOR module and details our methodology for reverse-engineering undocumented physical row mapping functions. In Section 3.2, techniques for allocating contiguous memory regions suitable for Rowhammer attacks are presented. Section 3.3 describes how SWAGE employs fuzzing techniques from related work to craft and profile access patterns to induce reliable bit flips. The chapter concludes with Section 3.4, where we present the VICTIM module, which provides an API to interact with a victim process and inject faults into it. The VICTIM module also provides a set of tools to analyze the target program and a page injector to deterministically place attacker-controlled pages into the victim's address space.

Chapter 4: A Rowhammer-Based Universal Forgery Attack Against SPHINCS$^+$ We evaluate SWAGE by conducting a Rowhammer-based universal forgery attack against SPHINCS$^+$. The attack is split into three phases. During the offline phase (Section 4.2), the attacker uses a replicated target system to prepare for the online phase of the attack, which is presented in Section 4.3. After successfully conducting the attack, the evaluation is concluded in Section 4.4 with a grafting phase, where the collected faulted signatures are post-processed and the universal forgery attack is performed. The evaluation shows that the Rowhammer attack is effective in practice, allowing the attacker to forge signatures against SPHINCS$^+$.

Chapter 5: Conclusions The thesis concludes with a summary of the discoveries and contributions made throughout this work, an overview of related work (Section 5.1), and an outlook on open problems for future research (Section 5.2).

2 Preliminaries

In this chapter, a background on the topics relevant to this thesis is provided. We first introduce some notation used throughout this thesis. In Section 2.2, an overview of the physical memory architecture of DRAM and how it is managed by CPUs is given. We discuss how DRAM is organized into banks, rows, and columns, and how the memory controller manages these components in Section 2.2.1. We then introduce virtual memory, and compare different strategies for managing memory in a multitasking operating system in Section 2.2.2. In Section 2.2.3, we discuss fault attacks, a class of practical attacks that exploit physical properties of a target system. We then introduce the Rowhammer bug in Section 2.2.4, a software-based fault attack that exploits disturbance errors in DRAM. Finally, we introduce the components of SPHINCS$^+$ in Section 2.3 and discuss the *grafting tree* fault attack against SPHINCS$^+$.

2.1 Notation

We denote bytes to be elements of $\mathbb{B} = \{0, 1, \ldots, 255\}$. For $k \in \mathbb{N}$, the set of all byte strings of length k is written as $\mathbb{B}^k$, and the set of all finite byte strings is denoted by $\mathbb{B}^* = \bigcup_{i \in \mathbb{N}_0} \mathbb{B}^i$. We denote hash functions by $\mathsf{H} : \mathbb{B}^* \to \mathbb{B}^\ell$, mapping byte strings of arbitrary length to fixed-length outputs of ℓ bytes. In cryptographic contexts, we denote the security parameter by n. Sampling a value v from a set V uniformly at random is denoted by $v \leftarrow\!\!\$\, V$.

If X is a byte string of length n, we denote the i-th byte of X by $X[i]$ for $i \in \{0, 1, \ldots, n-1\}$. If X is an array of m n byte strings, we denote the i-th n byte element of X by $X[i]$ for $i \in \{0, 1, \ldots, m-1\}$, and X is represented as a byte string $X[0] \| X[1] \| \ldots \| X[m-1]$ of length $m \cdot n$. A slice of a byte string X from index i (inclusive) to index j (exclusive) is denoted by $X[i:j] = X[i] \| X[i+1] \| \ldots \| X[j-1]$, where $i < j$.

2.2 Dynamic Random Access Memory and the Rowhammer Bug

In this section, we provide an overview of DRAM's architecture and explore how its inherent design characteristics can be exploited through fault attacks. We begin by outlining the fundamental principles of DRAM operation, including its organization into banks, rows, and columns, and discuss how timing and electrical interference can lead to faults. This

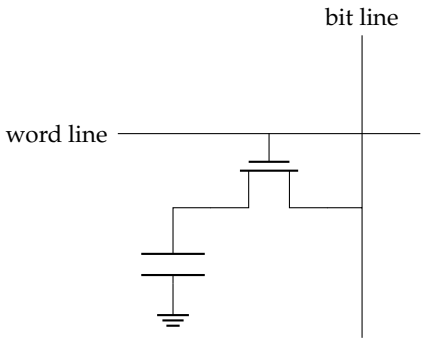


Figure 2.1: Structure of a DRAM cell. It consists of a transistor (top) and a capacitor (bottom left), connected to a *word line* and a *bit line*.

background sets the stage for a deeper examination of fault attacks, where the Rowhammer bug is used to induce errors deliberately, potentially compromising system integrity and security.

2.2.1 Dynamic Random Access Memory

Computers need memory to store data. Nowadays (and for the past several decades), memory is usually implemented using *Dynamic Random Access Memory* (DRAM). It consists of an array of DRAM cells, each containing a capacitor holding an electric charge representing the logical value of a bit (Figure 2.1).

Each storage cell consists of a transistor and a capacitor. The transistor controls the flow of current into the capacitor. The transistor's *gate* is connected to the *word line*, while the *source* and *drain* are connected to the capacitor and the *bit line*, respectively. The word line is used to select the row to read or store a value in, while the bit line is used to read or write the value stored in the capacitor. The capacitor is charged with the current on the bit line (if the bit line is high) or discharged onto the bit line (if the bit line is low) once the capacitor's connection to the bit line is closed by the transistor.

Since computers require gigabytes (or even terabytes) of memory, it is necessary to organize memory cells in a smart way. Therefore, DRAM modules organize memory in *banks*, which are arrays of *rows* and *columns* (Figure 2.2). Each row contains 65 536 columns, which amount to 8 KiB. To read a requested row, the DRAM array performs the following sequence of operations. First, the sense amplifiers are disconnected. Then, the bit lines are precharged to exactly equal voltage between low and high states. The precharge circuit is disabled, and the word line for the selected row is set to high. This connects the cells' storage capacitors to their corresponding bit lines. The capacitors then transfer their charge from their bit line (if the stored value is 1) or to the bit line (if the stored value is 0). This

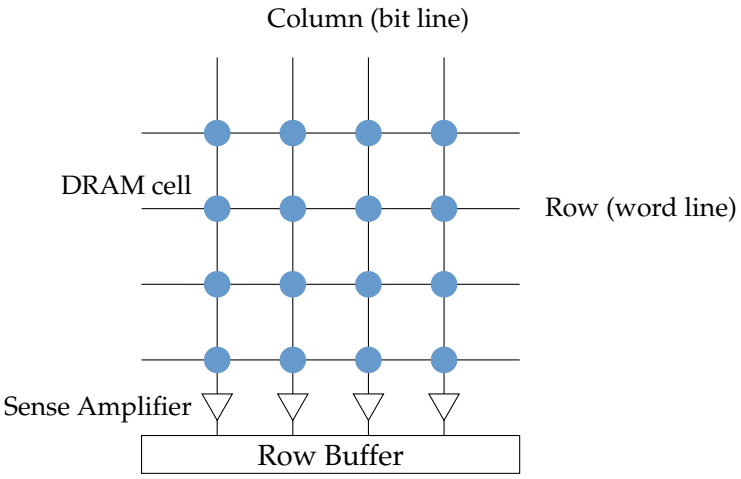


Figure 2.2: Structure of a 4x4 DRAM array.

causes the voltage on the bit line to drop or rise, respectively. The change in voltage is detected by the sense amplifiers, which amplify the signal and store it in the row buffer.
Memory should be stable under read operations, i.e., reading a bit should not change its value. For a DRAM cell, this does not intuitively hold. "Reading" the logical value of a capacitor consumes its charge. Therefore, and to increase the performance of repeated read operations, each memory bank contains a *row buffer*, holding the value of the most recently accessed row. The row buffer is realized using *Static Random Access Memory* (SRAM). In contrast to DRAM, where the logical value of a bit is represented by the charge of a capacitor, SRAM consists of only transistors in a circuit resembling a flip-flop circuit. SRAM is much faster than DRAM, but also more expensive.
In theory, a *perfect* capacitor in a DRAM cell should never lose its charge. In practice, however, this is not the case. Electronic components are not perfect, and capacitors leak their charge over time. Therefore, the charge in a DRAM cell has to be *refreshed* periodically. The DDR4 JEDEC standard [JEDEC, 2012] specifies that a DRAM cell has to be refreshed every $t_{REF} = 64\,\mathrm{ms}$ (the *refresh window*) to ensure that the charge in the capacitor is not lost during normal operation temperatures ($0\,^{\circ}\mathrm{C}$ to $85\,^{\circ}\mathrm{C}$). The refresh operation is issued automatically by the memory controller with an *Auto-Refresh* (AREF) command. An AREF command refreshes a number r of rows in every bank at once. The interval between two refresh operations is called t_{REFI}. Nominally, it can be calculated as

$$t_{REFI} = \frac{t_{REF} \cdot r}{R},$$

where R is the number of rows per bank. The JEDEC standard for DDR4 SDRAM also specifies that a refresh command has to be issued at least every $t_{REFI} = 7.8\,\mu\mathrm{s}$ (the *re-*

fresh cycle) for all memory densities under normal operation temperatures [JEDEC, 2012]. Therefore, the number of rows refreshed per bank by a single AREF command depends on the number of rows per bank and can be calculated as

$$r = \left\lceil \frac{t_{REFI}}{t_{REF}} R \right\rceil = \left\lceil \frac{7.8\,\mu s}{64\,ms} R \right\rceil = \left\lceil \frac{1}{8192} R \right\rceil.$$

The Rowhammer bug, introduced in Section 2.2.4, exploits the fact that DRAM cells can be disturbed by repeated access to neighboring rows. As refresh operations reset this effort, it is important for the attacker to understand the timing of refresh operations. When refreshing a row, the memory controller locks the row buffer and waits for the refresh operation to complete. Therefore, the refresh window t_{REF} of a DRAM module can be experimentally determined by repeatedly reading a fixed row and measuring the time it takes for the memory controller to serve the read request. This causes the read request to be delayed by the time it takes to refresh the row.
In the next section, we discuss how memory is managed in multitasking operating systems. The concept of virtual memory is introduced, which allows processes to have their own address space and enables the operating system to manage memory more flexibly.

2.2.2 Virtual Memory

In multitasking operating systems, processes have to share all the available resources including memory. But how should the operating system manage the physical memory between processes? Following the *keep it simple* approach, the operating system might choose to let processes access memory directly. But this approach has several significant problems. For one, programs have to be aware of other programs' memory layouts. Therefore, programs have to be recompiled for every system they run on, and for every combination of programs they are co-located with. Additionally, processes can access other processes' memory, which poses a severe security risk. As we can see, this approach is not feasible.
Another idea is to split physical memory in sections, where each process has its own section of memory. This approach enables the operating system to isolate processes from each other by crashing a process when it tries to access memory outside its section. But this approach comes its own problems. Firstly, this does not consider that programs have different (and changing) memory requirements. This approach also limits the number of processes that can be run simultaneously, as the operating system has to reserve memory for each process. Finally, this approach does not allow processes to be larger than the physical memory.
A much more successful approach, and the one used in modern operating systems, is to provide an abstraction of memory to programs. This abstraction is called *virtual memory*

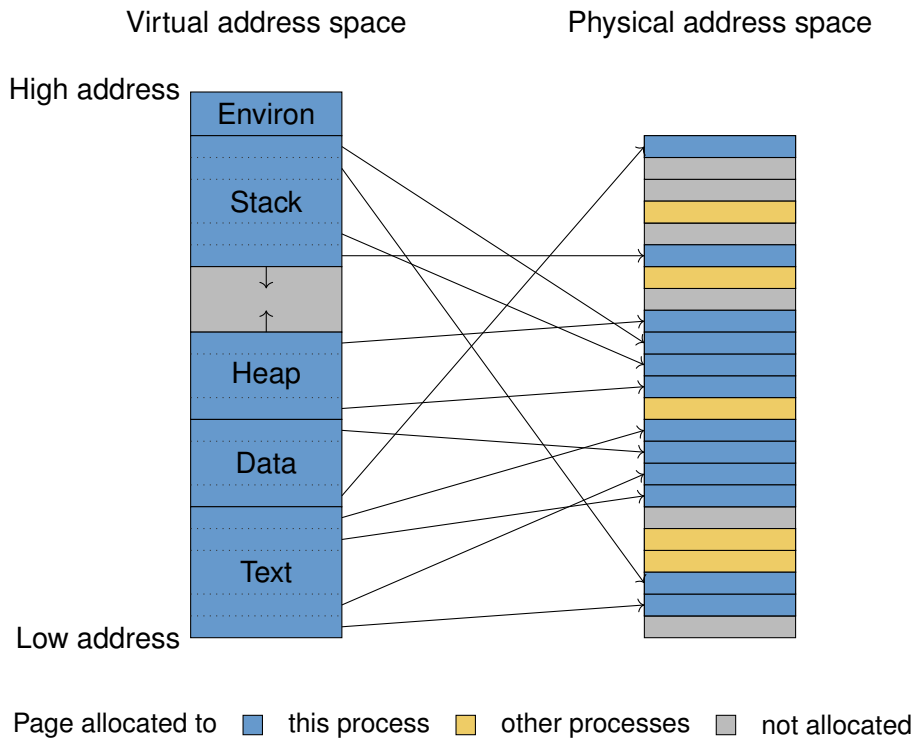


Figure 2.3: Virtual memory and virtual-to-physical address mapping of a C program.

(Figure 2.3). With virtual memory, each program has its own full address space. The operating system works in tandem with the *Memory Management Unit* (MMU) to map virtual addresses to physical addresses needed to interact with the memory subsystem. This approach has several advantages: Virtual memory is fully transparent, and programs do not have to be aware of other programs' memory layouts. Because of that, programs do not have to be adapted for different configurations. Virtual memory also allows process isolation, i.e., a process cannot access another process' memory. Additionally, memory contents can be moved from one memory location to another, updating the virtual-to-physical mapping accordingly, or even swapped to disk. Finally, processes can request more memory than physically available, and the operating system can swap out parts of the memory to disk.

Virtual memory is implemented with hardware support. The MMU is part of the CPU and translates virtual to physical addresses. This translation is performed with page-granularity where virtual memory is split into pages and the MMU translates each page individually. Therefore, the lower bits of the virtual and physical addresses are the same. The most common page size is $4\,\mathrm{KiB}$, hence $12\,\mathrm{bit}$ of equal address suffixes, but other sizes are also possible. For example, the x86-64 architecture supports $2\,\mathrm{MiB}$ and $1\,\mathrm{GiB}$ pages, so-

called *huge pages*. The MMU maintains a page table for each process, which contains the mapping between virtual and physical addresses. It is the operating system's responsibility to manage the page tables in the MMU and change page tables on context switches. Consider as an example the following simple program:

Listing 2.1: An example program demonstrating virtual memory

```
int main(void) {
    int *x = malloc(sizeof(int));
    *x = 42;
    printf("x=%p\n", x);
    printf("*x=%d\n", *x);
    return 0;
}
```

This program allocates memory for an integer `x`, and writes the value 42 to it. Then, it prints the *virtual address* and the value of `x`. The output of this program is as follows:

Listing 2.2: Output of the program from Listing 2.1

```
$ ./test
x=0x5555555592a0
*x=42
```

Hence, `0x5555555592a0` is the virtual address of `x`. To find the physical address backing this virtual address, we have to consult its corresponding page table entry. The page table entry of `x` can be found in the `/proc/<pid>/pagemap` file provided by the Linux kernel's `procfs` interface. This file contains the mapping for each page in the process' address space. The following function reads the physical address of a virtual address using the `pagemap`:

Listing 2.3: Function to read the physical address of a virtual address using `pagemap`

```
#define PAGEMAP_ENTRY 8  // Size of each entry in /proc/self/pagemap
#define PAGE_SHIFT 12    // 2**12 = 4096 (page size)
#define PAGE_SIZE (1UL << PAGE_SHIFT)
uint64_t virt_to_phys(uintptr_t va) {
    int fd = open("/proc/self/pagemap", O_RDONLY);
    uintptr_t page_idx = va / PAGE_SIZE;
    off_t offset = page_idx * PAGEMAP_ENTRY;
    lseek(fd, offset, SEEK_SET);
    uint64_t entry;
    read(fd, &entry, PAGEMAP_ENTRY);
    close(fd);
    uint64_t pfn = entry & ((1ULL << 55) - 1);
    return (pfn * PAGE_SIZE) + (va % PAGE_SIZE);
}
```

The `pagemap` follows the UNIX paradigm of *everything is a file*. First, the `pagemap` file is opened, and the index of the page containing the virtual address is calculated. Then, the file is sought to the correct position, and the entry is read. The entry contains, together with some meta information, the *Page Frame Number* (PFN) of the page, which makes up the lowest 55 bits of the entry.

After a virtual address has been translated to a physical address, it is then mapped to a physical location in DRAM. This mapping is done by a CPU-specific function subject to change between CPU generations. While the mapping function used to be publicly documented for Intel CPUs, it became undocumented in recent years, as it is considered an implementation detail.

Previous works introduced tools to reverse-engineer this now undocumented mapping function [Pessl et al., 2016, Xiao et al., 2016]. The timing side channel primitive in Algorithm 1 can be used to reverse-engineer the physical memory layout. The algorithm measures the time it takes to sequentially access a pair of memory addresses. If the timing is above a given threshold, the addresses were resolved to the same bank. This timing side channel is due to the row buffer conflict, which occurs when row r_2 is requested while another row r_1 is open in the same bank. To load r_2 into the row buffer, r_1 has to be closed and written back to the DRAM array, which takes longer than a non-conflicting access.

Algorithm 1: Row buffer conflict timing side channel

Input: Memory addresses r_1 and r_2, timing threshold τ
Output: True, if r_1 and r_2 are in the same bank, false otherwise

1 Load r_1
2 $t_0 \leftarrow$ now()
3 Load r_2
4 $\Delta_t \leftarrow$ now() - t_0
5 **return** $\Delta_t > \tau$

In the next section, we introduce fault attacks, a class of practical attacks that exploit physical properties of a system. While fault attacks cover a broad range of attacks, we focus on the Rowhammer bug in Section 2.2.4. The Rowhammer bug is a software-based fault attack that exploits disturbance errors in DRAM.

2.2.3 Fault Attacks

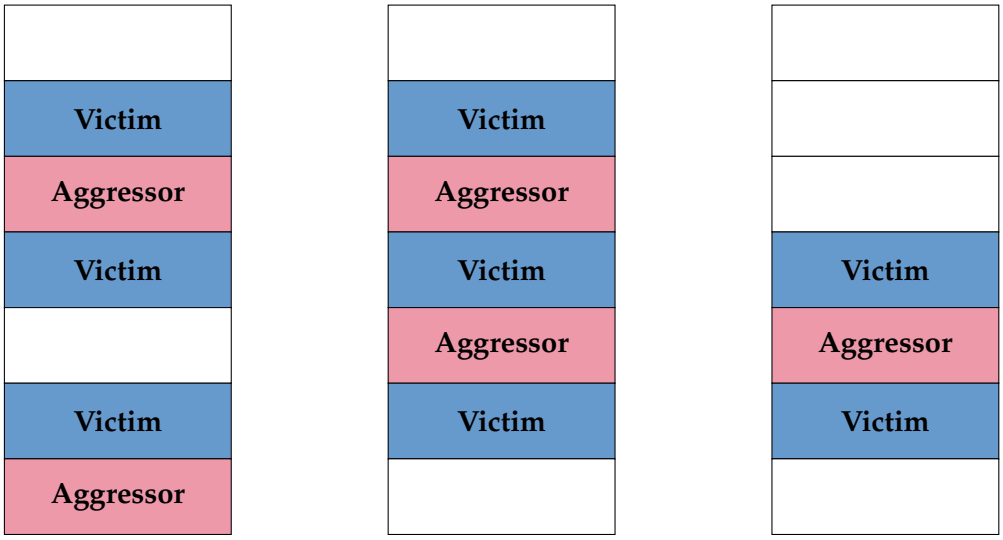
Fault attacks are a class of practical attacks that exploit physical properties of a system. This family of attacks usually assumes a stronger attacker model than the standard cryptographic model. In the standard model, the attacker either has access to the input and output of a cryptographic algorithm (a so-called *Chosen Plaintext Attack*) or has access to

a decryption oracle (a *Chosen Ciphertext Attack*). In fault attack threat models, the attacker can also influence the execution of the algorithm by stressing the device the algorithm is executed on. This stress can be applied in various ways, such as by applying electromagnetic interference, laser light, voltage glitches, or introducing disturbance in memory. Combined fault and leakage attacks strengthen the threat model by allowing the attacker to also observe the system for side-channel information such as power consumption or electromagnetic radiation while injecting faults into the system.
Different kinds of fault attacks have been used to attack different aspects of a system, a survey on relevant fault attacks against cryptographic schemes is given in [Baksi et al., 2022]. For example, fault injection can be used to bypass authentication mechanisms or to extract secret keys from cryptographic algorithms. As an introductory example, consider the simple authentication function in Listing 2.4. The function `is_authenticated` checks whether the system is in an authenticated state, for example by reaching out to an authentication server or checking the validity of a login session. It returns 1 if the current session is authenticated, and 0 otherwise. The function `show_confidential_data` stores the return value of `is_authenticated` in `auth` and grants access to confidential data if `auth` is non-zero. If an attacker can change the value of `auth` from a zero to a non-zero value between the call to `is_authenticated` and the following check, they can gain access to the confidential data without being authenticated.

Listing 2.4: Example of a simple authentication function

```
1  extern int is_authenticated(void);
2  int show_confidential_data(void) {
3      int auth = is_authenticated();
4      // ...more code that might take some time
5      if (auth) {
6          // grant access to confidential data
7      } else {
8          // deny access
9      }
10 }
```

This simple example not only illustrates the concept of fault attacks, but is also taken from real life. Constructions like the one in Listing 2.4 are often found in real-world systems, such as older versions of `sudo` or other system tools executed with elevated privileges. Those tools are regular targets for fault attacks, as they can be exploited to gain root privileges on a target system [Gruss et al., 2018, Adiletta et al., 2024], opening the door to further attacks on the system.
A simple code-based workaround for this problem is to compare `auth` to the value 1 instead of checking its truthiness. This makes it harder for an attacker to exploit the function,



(a) Single-sided (b) Double-sided (c) One-location

Figure 2.4: Standard Rowhammer access patterns. Aggressor rows are colored in red, victim rows are colored in blue. The aggressor rows are in physical proximity to the target rows. For double-sided patterns, victim rows between two aggressors are more likely to flip due to additional disturbance.

as they would have to change `auth` from zero to exactly one instead of any non-zero value. However, this is not a general solution to the problem, as it only hardens the attack surface, but does not eliminate it. Instead, countermeasures against fault attacks usually involve redundancy, such as using multiple independent implementations of the same algorithm and comparing their results. Only recently, countermeasures against combined fault and leakage attacks have been considered (e.g. [Berndt et al., 2023, Arnold et al., 2024]). They generally rely on *masking schemes* that split the secret data into multiple shares, making it harder for an attacker to extract the secret data by injecting faults or observing side-channel information.

In the following, we focus on a specific type of fault attack exploiting the Rowhammer bug, a software-induced fault attack that exploits disturbance errors in DRAM by repeatedly accessing so-called *aggressor rows*. The electromagnetic interference caused by the repeated accesses can cause nearby transistors to lose their charge, which leads to bit flips in the so-called *victim rows*.

2.2.4 The Rowhammer Bug

In the last decades, the capacity of DRAM modules has increased significantly, primarily by reducing the size of the individual memory cells and increasing the density of memory

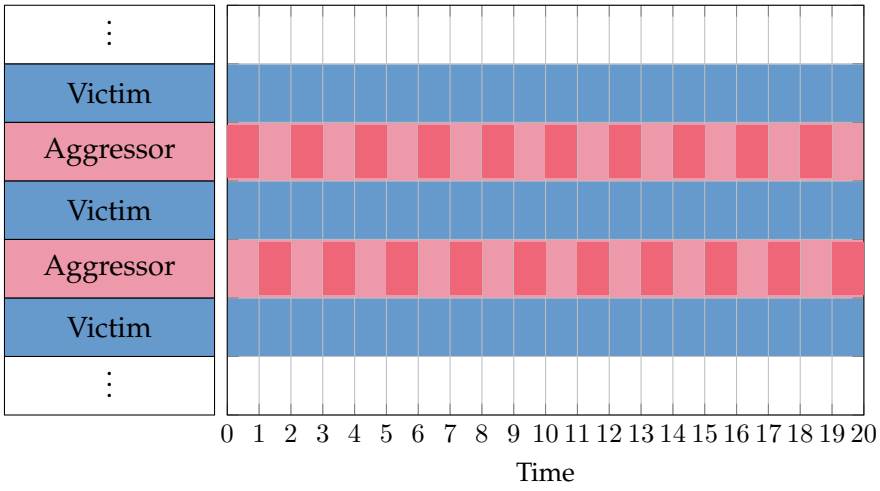


Figure 2.5: Double-sided access pattern

layouts. This trend, while essential for performance and cost-efficiency, led to a new class of errors, called *disturbance errors*. Disturbance errors occur when repeated or prolonged activation of DRAM rows causes capacitors to leak charge into adjacent memory rows, leading to unintended bit flips [Kim et al., 2014].

The *Rowhammer* bug, the first and most discussed memory disturbance error, works by repeatedly reading one or multiple *aggressor rows*. If repeated access to aggressors causes sufficient disturbance during a given *refresh cycle*, bit flips in *victim rows* in proximity to an aggressor row can be induced.

The Rowhammer bug was first disclosed to the scientific community by demonstrating that repeatedly and quickly accessing (or *hammering*) rows in a DRAM module could induce bit flips in adjacent rows [Kim et al., 2014]. Although the Rowhammer bug was initially dismissed as a reliability issue, it was quickly recognized as a potential security threat. In 2015, researchers showed that the Rowhammer effect can be practically exploited to gain kernel privileges [Seaborn and Dullien, 2015]. Since then, the Rowhammer bug has been used in various attacks. Noteworthy examples are ROWHAMMER.JS [Gruss et al., 2016], which uses the Rowhammer bug to break out of the browser sandbox, NETHAMMER [Lipp et al., 2020], a remote Rowhammer attack over the network, the FPGA-supported JACKHAMMER [Weissman et al., 2020] attack, and RAMBLEED [Kwong et al., 2020], which uses the Rowhammer bug to leak sensitive information from memory.

At a technical level, the Rowhammer bug exploits the *row buffer management* of DRAM modules. When a row is accessed, its contents are transferred into the row buffer, a small SRAM cache that holds the most recently accessed row in a bank. Subsequent access to a

different row in the same bank forces the memory controller to write back the contents of the row buffer to the DRAM array before loading the new row. This flushing and reloading of the row buffer causes a lot of electrical activity in the memory module, which can lead to disturbance errors in adjacent rows.
Early Rowhammer attacks used simple *single-sided*, *double-sided*, or even *one-location* access patterns, i.e., accessing one (or both) rows physically adjacent to the target row in quick succession (Figure 2.4). While double-sided access patterns are usually more effective due to the increased electrical activity, one-location hammering patterns – where only a single row is hammered – can under some circumstances be highly effective, because they allow a higher number of row activations per refresh cycle. Following the initial presentation of practical Rowhammer attacks, manufacturers implemented various countermeasures to mitigate Rowhammer attacks. Those countermeasures against Rowhammer attacks include:

- *Target Row Refresh* (TRR), a vendor-specific black box mechanism that tracks memory accesses and issues refresh commands for suspected *target rows* by identifying unusual access patterns. This preemptive refresh causes simple Rowhammer attacks to fail, as the charge in the target row is restored before sufficient disturbance for a bit flip can be induced.

- Increasing refresh rate t_{REFI} of DRAM modules [Kim et al., 2014]. This approach is effective by limiting the number of row actions per refresh cycle, but leads to increased power consumption and reduced performance due to the increased number of refresh commands.

- Implementing hardware-supported *Error-Correcting Codes* (ECC) to detect and correct bit flips [Seaborn and Dullien, 2015, Gruss et al., 2018]. However, this significantly increases the cost of DRAM modules. Additionally, ECC is not always effective against Rowhammer attacks [Cojocar et al., 2019] or might enable different attack vectors, e.g., timing-based attacks [Kwong et al., 2020].

- OS-level mitigations, such as *guard rows* enclosing memory pages holding sensitive data [van der Veen et al., 2018, Konoth et al., 2018]. While preventing Rowhammer attacks where aggressor and victim rows are not physically adjacent, this approach triples memory overhead and leads to significant performance degradation. Additionally, some Rowhammer attacks bypass guard rows by using aggressor rows that are not directly adjacent to the victim row [Jattke et al., 2022, Kogler et al., 2022].

However, all of these countermeasures have individual limitations, are not universally applicable, or are not effective against all Rowhammer attacks. For example, TRRES-

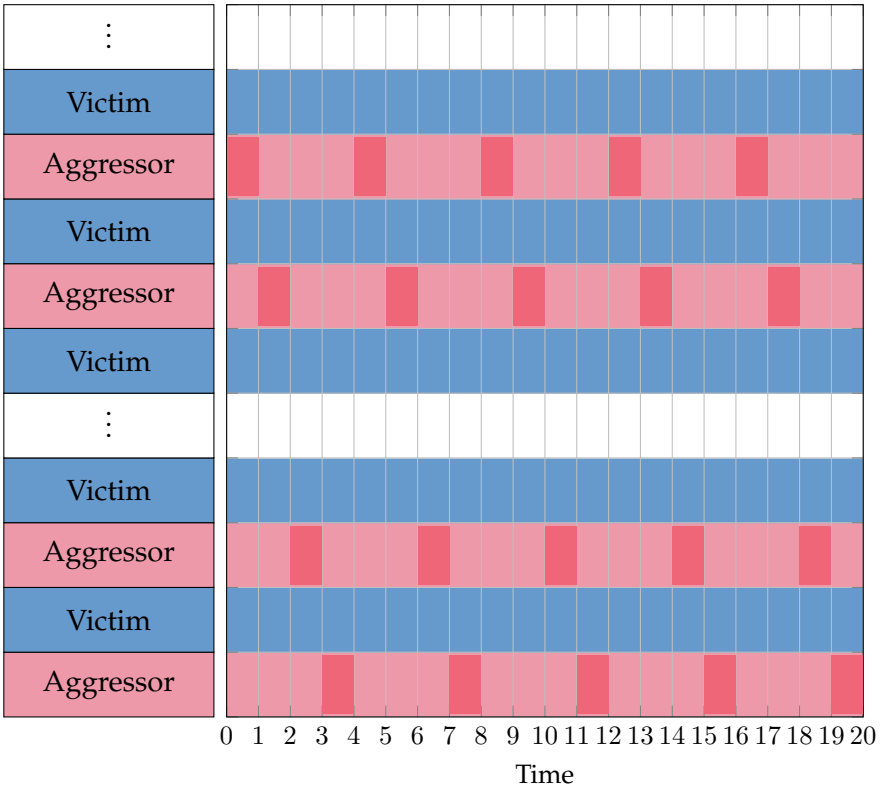


Figure 2.6: Many-sided access pattern in TRRESPASS. The figure shows a time series of memory accesses to aggressors rows (red). Victim rows are marked in blue . This memory access pattern overwhelms some TRR mechanism by accessing two pairs of aggressor rows, serving as a *dummy reads* for each other.

PASS [Frigo et al., 2020] circumvents TRR by augmenting many-sided access patterns with *dummy reads* (Figure 2.6). The dummy accesses are used to overwhelm the TRR mechanism, which is only able to track a limited number of rows and activations at once. In the practical evaluation, TRRESPASS successfully finds access patterns inducing bit flips in 13 of the 40 tested DDR4 memory modules, demonstrating that TRR is not effective against all Rowhammer attacks.
The Rowhammer bug is also not limited to a specific type of CPU architecture. In 2024, a Rowhammer attack against AMD Zen-based CPUs was presented, reverse-engineering non-linear address mapping functions and presenting the first Rowhammer attack against DDR5 modules [Jattke et al., 2024]. Together, these developments demonstrate that despite industry efforts, Rowhammer remains an active and evolving threat to memory integrity and system security.

2.3 The SPHINCS⁺ Digital Signature Scheme

Digital Signatures are an important tool to guarantee authenticity and integrity of messages. In general, a digital signature algorithm consists of the following three algorithms:

- Key generation algorithm KGen: Generates a key pair $(\mathsf{sk}, \mathsf{pk})$, where sk is the secret key used by the signing party and pk is the public key used to verify a given message.
- Signing algorithm Sign: Takes a message m and the secret key sk and outputs a signature σ.
- Verification algorithm Vf: Given a signature σ, a message m, and a public key pk, outputs "1" (or `true`) if the signature matches the message and "0" (`false`) otherwise.

We distinguish the security of a digital signature scheme against the following basic attacker models:

- *Key-Only Attack*: The attacker only knows the public key pk.
- *Known Signature Attack*: The attacker knows the public key pk and has a seen a set of signature/message pairs.
- *Chosen Message Attack*: The attacker knows the public key pk and can choose messages to be signed.

While a chosen message attack appears to be a very strong model, it is very common in practice. For example, a *Trusted Platform Module* (TPM) exposes a signing interface

`tpm2_sign` that allows the user to sign arbitrary messages using a key pair stored in the TPM. In this scenario, the TPM acts as a signing oracle for the user, identical to the chosen message attack.
We also distinguish several levels of success for an adversary against a digital signature scheme:

- *Existential Forgery*: The attacker is able to forge a valid signature of one message, not necessarily of their choice.
- *Universal Forgery*: The attacker is able to forge signatures for any message of their choice.
- *Total Break*: The attacker is able to recover the secret key from the public key and the algorithm's public parameters.

Clearly, different applications require different levels of security. Sometimes, it may be sufficient to ensure that the attacker can only forge signatures for *unimportant* messages, e.g., in scenarios where the format of *valid* messages is enforced by another protocol. In other cases, e.g., when the signing party is a notary or a software distributor, it may be necessary to ensure that the attacker cannot forge signatures for any message. In the former case, existential forgery might still be acceptable while universal forgery is not, while for the latter case, existential forgery poses a serious threat.
Many digital signature algorithms have been proposed. Some of the most prominent examples are the *Rivest-Shamir-Adleman Algorithm* (RSA), the *Digital Signature Algorithm* (DSA), and the *Elliptic Curve Digital Signature Algorithm* (ECDSA). While the security of RSA is based on the *RSA problem*, which is at most as hard as the integer factorization problem, DSA and ECDSA are based on the discrete logarithm problem. All of these problems are assumed to be *hard*, i.e., there is no known efficient non-quantum algorithm to solve them. With quantum computers, however, these problems can be solved efficiently using Shor's algorithm [Shor, 1994]. This is an example of a *total break* of the signature scheme in a *key-only attack* as the attacker can recover the secret key sk knowing only the public key pk and the algorithm's public parameters.
To ensure cryptographic security in a post-quantum world, the *National Institute of Standards and Technology* (NIST) has started the *Post-Quantum Cryptography* (PQC) standardization process to find cryptographic algorithms that are secure against quantum attackers. Since 2017, many algorithms have been proposed, several have been selected as finalists, and few have been standardized for different purposes. The algorithms proposed are based on different mathematical problems, such as lattice-based cryptography, code-based cryptography, multivariate polynomial cryptography, and hash-based cryptography. Hash-based signature algorithms are a particularly interesting field as their security

is based solely on the hardness of hash functions. Since hash functions are used in many cryptographic protocols, and are presumably not breakable by quantum attackers, hash-based signatures are a promising candidate for post-quantum cryptography.
In the following sections, we introduce SPHINCS⁺, a stateless hash-based signature scheme standardized in 2024 as SLH-DSA. We then introduce SPHINCS⁺ by first giving an overview of the scheme (Section 2.3.2), followed by a detailed description of the scheme's components (Sections 2.3.3 to 2.3.8). To conclude the introduction of SPHINCS⁺, an overview of the proposed parameter sets is given in Section 2.3.9 and differences between SPHINCS⁺ and the standardized SLH-DSA are discussed in Section 2.3.10. Since SPHINCS⁺ is a complex scheme, we first introduce a simple hash-based signature scheme in Section 2.3.1 to build up the necessary knowledge.

2.3.1 Motivating Examples for Hash-Based Signature Schemes

To motivate the concept of hash-based signature schemes, we take a closer look at a list of example schemes. The schemes introduced in this section build onto each other, starting with a simple scheme that signs an empty message and ending with a signature scheme that can sign arbitrary messages. The schemes introduced in this section were used as introductory examples in [Lange, 2021].
Let us first consider a simple hash-based signature scheme where a participant wants to sign an *empty* message. The scheme assumes the existence of *secure* hash functions. A secure hash function is defined as follows:

Definition 2.1 (Secure hash function [Easttom, 2022]). Let $\mathsf{H}\colon \mathbb{B}^* \to \mathbb{B}^n$ be a hash function. We call H a secure hash function if it satisfies the following properties:

- *Preimage resistance*: Given $h \in \mathbb{B}^n$, it is computationally infeasible to find $x \in \mathbb{B}^*$ such that $\mathsf{H}(x) = h$.
- *Weak collision resistance*: Given $x \in \mathbb{B}^*$, it is computationally infeasible to find $x' \in \mathbb{B}^*$ such that $x' \neq x$ and $\mathsf{H}(x) = \mathsf{H}(x')$.
- *Collision resistance*: It is computationally infeasible to find two distinct messages $x, y \in \mathbb{B}^*$ such that $\mathsf{H}(x) = \mathsf{H}(y)$.

Given a secure hash function H, the scheme SIGN-EMPTY consists of three algorithms KGen, Sign, and Vf.
The key generator in Algorithm 2 generates a key pair $(\mathsf{sk}, \mathsf{pk})$. It does so by first sampling a random value sk as the secret key. The public key pk is then obtained by applying H to sk.

Algorithm 2: Key generator KGen for SIGN-EMPTY

Output: Key pair $(\mathsf{sk}, \mathsf{pk})$

$sk \leftarrow\$ \{0,1\}^n$

$\mathsf{pk} \leftarrow \mathsf{H}(\mathsf{sk})$

return $(\mathsf{sk}, \mathsf{pk})$

Algorithm 3: Signing algorithm Sign for SIGN-EMPTY

Input: Message m, secret key sk

Output: Signed message σ

assert $m = \varepsilon$

return $\sigma \leftarrow \mathsf{sk}$

The signing algorithm in Algorithm 3 takes as input a message m and the secret key sk. After checking that m is indeed the empty string, it reveals sk as the signature σ.

The verification algorithm in Algorithm 4 takes as input a signature σ and the public key pk. It checks that pk is indeed the hash of σ. If this is the case, the algorithm returns the empty string, which is the message that was originally signed.

This signature scheme is secure for signing a single empty message if H is a secure hash function: In order for an attacker to sign an (empty) message, they have to find a collision for pk under H. This is hard, as H satisfies preimage resistance by definition of secure hash functions.

In the big picture, this signature scheme can be seen as a commitment pk to the secret key sk. When signing the message, the signing party reveals their secret sk, and the verifier can check that the public key is indeed the hash of the secret key. This is a one-time scheme: the secret key sk can only be used to sign one message, and after a message was signed, the secret key has to be discarded, i.e., once the commitment to pk has been revealed, it cannot be hidden again.

While the utility of the SIGN-EMPTY signature scheme might be disputable, it shows how a hash-based signature scheme can also be

Algorithm 4: Verification algorithm Vf for SIGN-EMPTY

Input: Signed message σ, public key pk

Output: True if signature is valid, false otherwise

if $\mathsf{pk} = \mathsf{H}(\sigma)$ **then**

  return true

else return false

used as a subroutine to sign a one-bit message. For the one-bit signature scheme SIGN-BIT, we represent zero and one with respective commitments to keys sk_0 and sk_1.

Algorithm 5: Key generator KGen for SIGN-BIT

Output: Key pair $(\mathsf{sk}, \mathsf{pk})$
$(\mathsf{sk}_0, \mathsf{pk}_0) \leftarrow \mathsf{KGen}_{\text{SIGN-EMPTY}}(n)$
$(\mathsf{sk}_1, \mathsf{pk}_1) \leftarrow \mathsf{KGen}_{\text{SIGN-EMPTY}}(n)$
$\mathsf{sk} \leftarrow (\mathsf{sk}_0, \mathsf{sk}_1)$
$\mathsf{pk} \leftarrow (\mathsf{pk}_0, \mathsf{pk}_1)$
return $(\mathsf{sk}, \mathsf{pk})$

Algorithm 6: Signing algorithm Sign for SIGN-BIT

Input: Message m, secret key $\mathsf{sk} = (\mathsf{sk}_0, \mathsf{sk}_1)$
Output: Signed message $s_m = (\sigma, m)$
assert $0 \leq m \leq 1$
if $m = 0$ **then**
  return $(0, \mathsf{Sign}_{\text{SIGN-EMPTY}}(\mathsf{sk}_0))$
else return $(1, \mathsf{Sign}_{\text{SIGN-EMPTY}}(\mathsf{sk}_1))$

Algorithm 7: Verification algorithm Vf for SIGN-BIT

Input: Signed message $s_m = (\sigma, m)$, public key $\mathsf{pk} = (\mathsf{pk}_0, \mathsf{pk}_1)$
Output: True if signature is valid, false otherwise
assert $0 \leq m \leq 1$
if $m = 0$ **then**
  return $\mathsf{Vf}_{\text{SIGN-EMPTY}}((0, \mathsf{pk}_0))$
else return $\mathsf{Vf}_{\text{SIGN-EMPTY}}((1, \mathsf{pk}_1))$

The key generator in Algorithm 5 generates two key pairs $(\mathsf{sk}_0, \mathsf{pk}_0)$ and $(\mathsf{sk}_1, \mathsf{pk}_1)$ using the SIGN-EMPTY scheme. The keys sk and pk are concatenations of the respective keys for zero and one. To sign a message, the signing algorithm in Algorithm 6 takes as input a message m and the secret key sk. It then delegates the signing to the SIGN-EMPTY scheme, choosing the appropriate signing key depending on the value of m. Similarly, to verify a signed message, the verification algorithm in Algorithm 7 delegates the verification to the SIGN-EMPTY scheme.

It can be seen that the SIGN-BIT scheme is secure for signing a single message due to the security of the SIGN-EMPTY scheme. It can even be used twice without compromising authenticity: once for message 0 and once for message 1.

We can now use the SIGN-BIT scheme as a building block to sign longer messages of length

k in a SIGN-k-BIT scheme. For that, the key generator has to generate $2k$ key pairs, two for each bit of the message. The signing algorithm then signs each bit of the message using the SIGN-BIT scheme. With this construction, we can sign messages of arbitrary but fixed length. The SIGN-k-BIT scheme is a one-time signature scheme. It is secure for one-shot signing if the SIGN-BIT scheme is secure.
If the signature scheme was used multiple times, the following chosen signature attack on the SIGN-k-BIT scheme would be possible. For a key $(\mathsf{sk}, \mathsf{pk})$, we denote its key pairs corresponding to bit $i \in [1, k]$ of the message by $(\mathsf{sk}_i, \mathsf{pk}_i)$. We further denote the signing key corresponding to zero or one with sk_i^0 and sk_i^1, respectively. In the following existential forgery attack, we see that the attacker can execute an existential forgery attack after sending two queries to the signing oracle.

Example 2.2 (Existential Forgery). When signing messages $m_0 = 010$ and $m_1 = 100$ using the SIGN-3-BIT scheme, the signing party reveals secret keys $\mathsf{sk}_3^0, \mathsf{sk}_3^1, \mathsf{sk}_2^0, \mathsf{sk}_2^1,$ and sk_1^0. An attacker can now combine $\mathsf{sk}_3^1, \mathsf{sk}_2^1$ and sk_1^0 to forge a valid signature for $m' = 110$.

In practice, many hash-based signature schemes are also one-time or few-time signature schemes, i.e., they can only be used to sign one or few messages without revealing the secret key. Therefore, these hash-based signature schemes have to generate many key pairs and maintain a state about the already used keys or find some other mechanism to evade attacks without the need to keep track of used keys. In the next sections, we introduce the SPHINCS$^+$ signature scheme and see how it uses a hypertree structure to maintain exponentially many signing keys in a stateless fashion.

2.3.2 Overview of the SPHINCS$^+$ Signature Scheme

The SPHINCS$^+$ signature scheme, first submitted to the NIST post-quantum standardization process in [Bernstein et al., 2017] and published in [Bernstein et al., 2019], is a stateless hash-based signature scheme standardized in 2024 as SLH-DSA [NIST, 2024]. It is constructed using other hash-based signature schemes as components: (1) the few-time signature scheme FORS, and (2) a hypertree structure XMSS acting as a many-time signature scheme. In an XMSS tree, a number of hash-based one-time signature scheme WOTS$^+$ instances are managed.
In principle, a SPHINCS$^+$ key pair consists of exponentially many FORS key pairs. The FORS scheme is used to sign messages, gradually losing security until a key pair has to be discarded. However, as managing exponentially many keys is infeasible, SPHINCS$^+$ uses a hypertree structure to maintain signing keys in a stateless fashion. By using a randomized addressing scheme, SPHINCS$^+$ locates the FORS key pair and the *authentication path*

to use for a given message. A SPHINCS⁺ signature then consists of a FORS signature and a hypertree authentication path.

Figure 2.7 shows the signing process of SPHINCS⁺. After a path randomization value R is generated, the message m is hashed to a message digest md and split into k chunks md_i signed using randomized FORS instances, producing the FORS signatures σ_i^F. Those signatures are then authenticated using their associated FORS trees, resulting in a FORS signature $\sigma^F = (\sigma_0^F, \ldots, \sigma_{k-1}^F, auth(\sigma_0^F), \ldots, auth(\sigma_{k-1}^F)$. The SPHINCS⁺ scheme then derives the FORS root key pk^F from σ^F and md. After signing the message, the SPHINCS⁺ enters the hypertree structure at layer $l = 0$, signing the FORS root key pk^F, generating a WOTS⁺ signature σ_0^W. The WOTS⁺ instance used for signing is then authenticated using its corresponding XMSS tree, emitting the XMSS authentication path $auth(\sigma_0^W)$. Subsequently, the XMSS root pk_1^X is signed using a WOTS⁺ instance at the next layer to produce a WOTS⁺ signature σ_1^W. This process is repeated d times until it reaches the root of the hypertree, resulting in the full signature

$$\sigma = (R, \mathsf{md}, \sigma^F, \sigma_0^W, auth(\sigma_0^W), \ldots, \sigma_{d-1}^W, auth(\sigma_{d-1}^W)).$$

In the next sections, the components of SPHINCS⁺ are introduced in more detail. We first introduce needed functions and definitions in Section 2.3.3, will then introduce the WOTS⁺ (Section 2.3.4) and XMSS (Section 2.3.5) schemes, and finally bring them together by introducing the SPHINCS⁺ hypertree in Section 2.3.6. Subsequently, after a brief introduction of the FORS scheme in Section 2.3.7, the SPHINCS⁺ interface is introduced in Section 2.3.8. Finally, we discuss the proposed parameter sets in Section 2.3.9 and the differences between SPHINCS⁺ and SLH-DSA in Section 2.3.10. Most algorithms and definitions introduced below are taken from the SLH-DSA standardization document [NIST, 2024], except for the **`treehash`** algorithm (Algorithm 15), which is taken from the SPHINCS⁺ specification document [Aumasson et al., 2022].

2.3.3 Functions and Definitions

The SPHINCS⁺ signature scheme comes with a set of functions, definitions, and an address structure **ADRS** used throughout the scheme. This section introduces the core cryptographic primitives and definitions that are integral to the scheme's construction. We start by introducing the hash functions and pseudorandom functions used in SPHINCS⁺ and summarizing their parameters and outputs. We then introduce helper functions used in the algorithms. Concluding, the address structure **ADRS** is discussed, which is used to uniquely key the hash functions and pseudorandom functions.

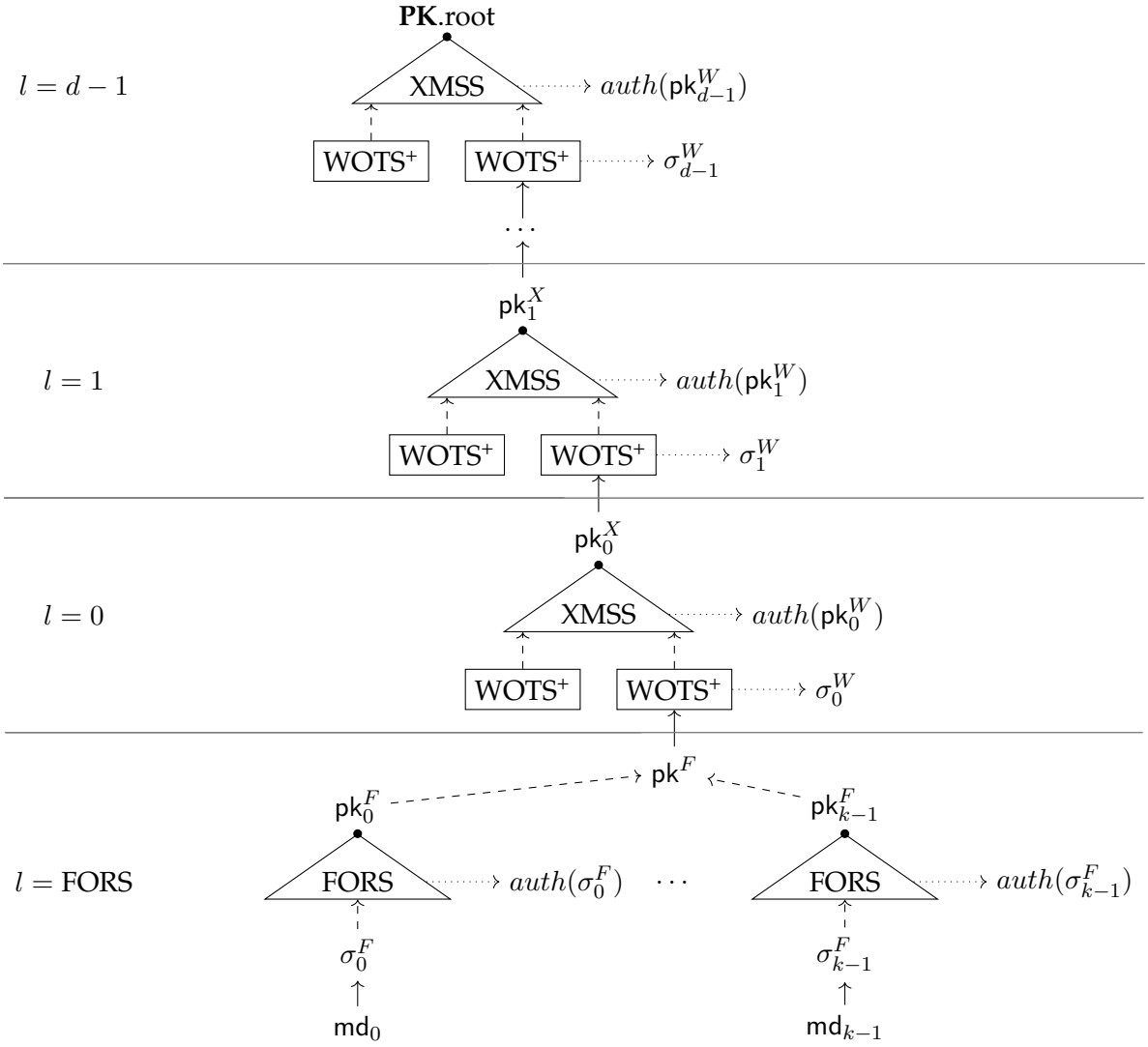


Figure 2.7: Overall SPHINCS⁺ structure. Solid lines denote signing, dashed lines denote authentication, and dotted lines denote outputs in the hypertree.

Table 2.1: Hash functions and pseudorandom functions used in SPHINCS⁺

Function	Parameters	Input	Output
$\mathsf{PRF}_{\mathsf{msg}}$	$(\mathbf{SK}.\text{prf}, \text{opt\_rand}) \in \mathbb{B}^n \times \mathbb{B}^n$	$m \in \mathbb{B}^*$	$R \in \mathbb{B}^n$
Generates a randomization value R for a message m.			
$\mathsf{H}_{\mathsf{msg}}$	$(R, \mathbf{PK}.\text{seed}, \mathbf{PK}.\text{root}) \in \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^n$	$m \in \mathbb{B}^*$	$R \in \mathbb{B}^n$
Generates a message digest md *for a message m to be signed.*			
PRF	$(\mathbf{PK}.\text{seed}, \mathbf{ADRS}) \in \mathbb{B}^n \times \mathbb{B}^{32}$	$\mathbf{SK}.\text{seed} \in \mathbb{B}^n$	$s \in \mathbb{B}^n$
Generates secret values used in WOTS⁺ and FORS.			
T_ℓ	$(\mathbf{PK}.\text{seed}, \mathbf{ADRS}) \in \mathbb{B}^n \times \mathbb{B}^{32}$	$m_1 \in \mathbb{B}^{\ell n}$	$y \in \mathbb{B}^n$
Hash function that maps an ℓn-byte message to an n-byte message.			
H	$(\mathbf{PK}.\text{seed}, \mathbf{ADRS}) \in \mathbb{B}^n \times \mathbb{B}^{32}$	$m_2 \in \mathbb{B}^{2n}$	$y \in \mathbb{B}^n$
Special case of T_ℓ that takes messages of lengths $2n$.			
F	$(\mathbf{PK}.\text{seed}, \mathbf{ADRS}) \in \mathbb{B}^n \times \mathbb{B}^{32}$	$x \in \mathbb{B}^n$	$y \in \mathbb{B}^n$
A hash function that takes an n-byte input and produces an n-byte output.			

Hash Functions and Pseudorandom Functions

The SPHINCS⁺ signature scheme uses six functions throughout its components: $\mathsf{PRF}_{\mathsf{msg}}$, $\mathsf{H}_{\mathsf{msg}}$, PRF, T_ℓ, H, and F. All of these functions are implemented using hash functions or Extensible Output Functions (XOFs) such as SHA-2 or SHAKE. Table 2.1 summarizes the hash functions and pseudorandom functions used in SPHINCS⁺. The **ADRS** structure is used to key the hash (pseudorandom, resp.) functions.

Helper Functions

The SPHINCS⁺ signature scheme uses three helper functions in its algorithms. The function pair `toInt` and `toByte` converts a sequence of bytes to an integer and vice versa. They are used to converting byte strings from hash digests to integers and back, representing a byte string as an integer in big-endian format.

Algorithm 8: toInt(X, n)

Input: n-byte string X.
Output: Integer value of X.

```
total ← 0
for i ← 0 to n − 1 do
    total ← 256 · total + X[i]
return total
```

Algorithm 9: toByte(x, n)

Input: Integer x, string length n.
Output: n-byte string S containing the big-endian representation of x.

$total \leftarrow x$
for $i \leftarrow 0$ **to** $n-1$ **do**
  $S[n-1-i] \leftarrow total \bmod 256$ `// least significant 8 bits of total`
  $total \leftarrow total \gg 8$ `// shift right by 8 bits`
return S

The function **base_2^b** represents a sequence of bytes as a sequence of base-2^b blocks. It is used to convert a byte string into an array of integers, where each integer represents a block of b bits.

Algorithm 10: base_2^b(X, b, out_len)

Input: Byte string X of length at least $\lceil \frac{out\_len \cdot b}{8} \rceil$, integer b, output length *out_len*.
Output: Array $base_b$ of length *out_len*, with each element in $[0, \ldots, 2^b - 1]$.

$in \leftarrow 0$
$bits \leftarrow 0$
$total \leftarrow 0$
for $out \leftarrow 0$ **to** $out\_len - 1$ **do**
  while $bits < b$ **do**
    $total \leftarrow (total \ll 8) + X[in]$ `// append next byte`
    $in \leftarrow in + 1$
    $bits \leftarrow bits + 8$
  $bits \leftarrow bits - b$ `// consume b bits`
  $base_b[out] \leftarrow \big(total \gg bits\big) \bmod 2^b$ `// extract top b bits`
return $base_b$

Addresses

Four of the functions described in Table 2.1 take an address **ADRS** as an additional parameter. **ADRS** is a 32-byte public value used in SPHINCS$^+$ to store addressing information for use in the signing and verification algorithms. It is used to key into hash functions and pseudorandom functions, where each call to the function takes a unique address. In the case of PRF, this allows the generation of a vast amount of different secret values from a single **SK**.seed. There are five different types of addresses used in SPHINCS$^+$: two for WOTS$^+$, one for XMSS, and two for FORS. We omit the FORS addresses here, as FORS is not the focus of this work.

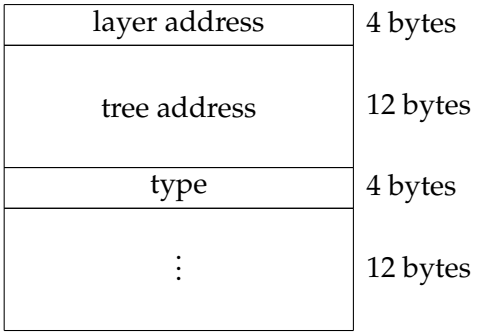


Figure 2.8: Structure of an **ADRS**.

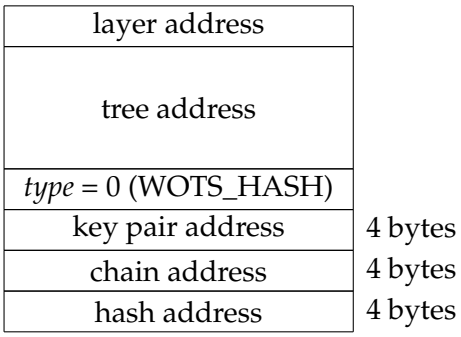


Figure 2.9: WOTS+ hash address.

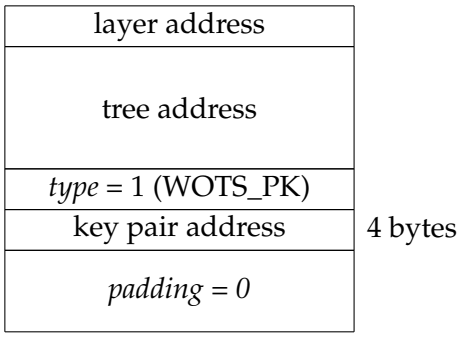


Figure 2.10: WOTS+ pk compression address.

All **ADRS** types hold a layer address, a tree address, and a type identifier (Figure 2.8). The layer address and tree address identify the current layer of the hypertree and the current tree address. The type identifier is used to specify the type of the address.

Figure 2.9 shows the WOTS+ address with the type identifier WOTS_HASH (*type* = 0). It is used to key the chaining hash function during signature generation and verification. In addition to the layer address, the type address, and the identifier, it also contains the key pair address, the chain address, and the hash address. The key pair address identifies the key pair in the WOTS+ signing procedure. The chain address identifies the current the chain of the WOTS+ instance. The hash address identifies the current step in the hash chain.

Figure 2.10 shows the WOTS+ public key compression address with the type identifier WOTS_PK (*type* = 1). It is used to key the hash function during public key compression, the last step in a WOTS+ chain. It is structurally similar to a WOTS+ hash address, but it uses a different type identifier and has the last two words set to zero.

Figure 2.11 shows the XMSS address with the type identifier TREE (*type* = 2), which is used during the XMSS authentication path computation. The first word after the type identifier is set to zero, and the current height and index in the tree are stored in the next two words.

layer address	
tree address	
type = 2 (TREE)	
padding = 0	4 bytes
tree height	4 bytes
tree index	4 bytes

Figure 2.11: XMSS tree address.

layer address	
tree address	
type = 5 (WOTS_PRF)	
key pair address	4 bytes
chain address	4 bytes
hash address = 0	4 bytes

Figure 2.12: WOTS$^+$ key generation address.

The WOTS$^+$ key generation address in Figure 2.12 with the type identifier WOTS_PRF (*type* = 5) is used to key the hash function during WOTS$^+$ key generation. It is similar to the WOTS$^+$ hash address, but has the hash address set to zero.

An address exposes several methods to access and manipulate its components. If X is a component of **ADRS**, then **ADRS**.getX() returns the value of X, and **ADRS**.setX(v) sets the value of X to v. For example, **ADRS**.getLayerAddress() returns the layer address, and **ADRS**.setLayerAddress(l) sets the layer address to l. One exception is the address type: the type of **ADRS** is set using **ADRS**.setTypeAndClear(t), which also sets every word after the type identifier to zero.

2.3.4 Winternitz One-Time Signature Scheme$^+$

The *Winternitz One-Time Signature Scheme*$^+$ (WOTS$^+$) is a hash-based one-time signature scheme. It is based on the original WOTS scheme [Merkle, 1990] and was adapted for use with SPHINCS$^+$. Signing a message using WOTS$^+$ involves applying a hash function to the secret key a message-dependent number of times.

A WOTS$^+$ instance is parameterized by:

- n: The security parameter that defines the output length of the hash function F, the length of messages to be signed, the length of secret key components, and the length of public key components.

- $\lg_w$: The block size, i.e., the number of bits encoded by each hash chain.

The security parameter n is defined to be one of 16, 24, or 32 bytes, while the block size $\lg_w$ is fixed to 4 for all parameter sets (see Table 2.2). Additionally, the following parameters are derived from the security parameter and the block size:

$w = 2^{\lg_w}$	The length of each hash chain
$\ell_1 = \left\lceil \frac{8n}{\lg_w} \right\rceil$	The number of $\lg_w$ bit message blocks
$\ell_2 = \left\lfloor \frac{\log_2((w-1)\ell_1)}{\lg_w} \right\rfloor + 1$	The number of $\lg_w$ bit blocks in the checksum
$\ell = \ell_1 + \ell_2$	The total number of $\lg_w$ bit blocks to be signed

For example, for a security parameter $n = 32$ and block size $\lg_w = 4$, we have $w = 16$, $\ell_1 = 64$, $\ell_2 = 3$, and $\ell = 67$.

A WOTS⁺ secret key consists of ℓ secret key components $s_i \in \mathbb{B}^n$ for $1 \leq i \leq \ell$. Each of these secret key components corresponds to the start of a hash chain with w steps. All the secret key components are derived from a single secret key seed **SK**.seed $\in \mathbb{B}^n$ using a pseudorandom function PRF.

To sign a message $m \in \mathbb{B}^*$, the message is first split into ℓ_1 $\lg_w$-bit blocks $(b_1, \ldots, b_{\ell_1})$. The message is then padded with a checksum c that is computed as $c = \sum_{i=1}^{\ell_1} w - 1 - b_i$. The checksum c is then, similarly to m, split into ℓ_2 $\lg_w$-bit blocks $(b_{\ell_1+1}, \ldots, b_\ell)$. The signature σ then consists of ℓ components $\sigma_i \in \mathbb{B}^n$ for $1 \leq i \leq \ell$, where b_i corresponds to the number of applications F to an initial state derived from the secret key s_i.

WOTS⁺ utilizes two helper functions: setting $b = \lg_w$, the function **base_2^b** (Algorithm 10) is used to convert the message and checksum into base-$2^{\lg_w}$ blocks. The function **chain** (Algorithm 11) applies F to its n-byte input s many times, updating the **ADRS** accordingly.

Algorithm 11: chain(X, i, s, **PK**.seed, **ADRS**)

Input: Input string X, start index i, number of steps s, public seed **PK**.seed, address **ADRS**.
Output: Value of F iterated s times on X.

```
tmp ← X
for j ← i to i + s − 1 do
    ADRS.setHashAddress(j)
    tmp ← F(PK.seed, ADRS, tmp)
return tmp
```

Algorithm 12 shows the key generation algorithm **wots_pkGen** for WOTS⁺. The public key pk is generated from both the secret **SK**.seed and the public **PK**.seed from SPHINCS⁺. The algorithm first applies the pseudorandom function PRF to **SK**.seed at the address

ADRS to generate the secret key components s_i. Afterwards, the algorithm applies **chain** to each s_i to generate the public key components pk_i. Finally, the public key pk is generated by compressing the public key components using the function T_ℓ.

Algorithm 12: wots_pkGen(**SK**.seed, **PK**.seed, **ADRS**)

Input: Secret seed **SK**.seed, public seed **PK**.seed, address **ADRS**.
Output: WOTS$^+$ public key pk.

```
1  skADRS ← ADRS                    // copy address to create sk address
2  skADRS.setTypeAndClear(WOTS_PRF)
3  skADRS.setKeyPairAddress(ADRS.getKeyPairAddress())
4  for i ← 0 to ℓ − 1 do
5      skADRS.setChainAddress(i)
6      sk ← PRF(PK.seed, SK.seed, skADRS)          // chain i secret value
7      ADRS.setChainAddress(i)
8      tmp[i] ← chain(sk, 0, w − 1, PK.seed, ADRS)  // chain i public value
9  wotspkADRS ← ADRS    // copy address to create WOTS+ pk address
10 wotspkADRS.setTypeAndClear(WOTS_PK)
11 wotspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress())
12 pk ← T_ℓ(PK.seed, wotspkADRS, tmp)              // compress public key
13 return pk
```

Algorithm 13 shows the signing algorithm **wots_sign** for WOTS$^+$. It takes a message M, a secret seed **SK**.seed, a public seed **PK**.seed, and an address **ADRS** as input. The algorithm first converts the message M to a base-w representation $\mathsf{msg} = (m_1, m_2, \ldots, m_{\ell_1})$. It then computes a checksum csum based on the message. The checksum is then converted to a base-w representation as well and appended to the message. The algorithm then computes the chain secret key values s_i using the pseudorandom function PRF on the seeds **SK**.seed and **PK**.seed, and the address **ADRS**. Afterwards, **chain** is called to apply F m_i times to chain s_i to compute the signature value σ_i^W for each chain.
Algorithm 14 shows the public key generation algorithm **wots_pkFromSig** for WOTS$^+$. It serves as the verification algorithm for WOTS$^+$ signatures, taking a WOTS$^+$ signature σ^W, a message M, a public seed **PK**.seed, and an address **ADRS** as input. Similarly to the signing algorithm, the message M is converted to its base-w representation msg and a checksum $csum$ is computed. The checksum is then appended to the message. The algorithm then completes the chain computation by applying **chain** to each signature component σ_i^W, stored in $tmp[i]$. Concluding, the public key pk_{sig} is generated by compressing the public key components. If the signature is valid, the extracted public key pk_{sig} will match the public key of the signing party.
If the underlying hash function is secure, then the WOTS$^+$ signature scheme is secure for one-time signing. In SPHINCS$^+$, WOTS$^+$ is used as a building block for the many-time

Algorithm 13: wots_sign(M, **SK**.seed, **PK**.seed, **ADRS**)

Input: Message M, secret seed **SK**.seed, public seed **PK**.seed, address **ADRS**.
Output: WOTS⁺ signature $\sigma^W = (\sigma_0^W, \ldots, \sigma_{\ell-1}^W)$.

```
csum ← 0
msg ← base_2^b(M, lg_w, ℓ_1)                          // convert message to base w
for i ← 0 to ℓ_1 − 1 do
    csum ← csum + w − 1 − msg[i]
csum ← csum ≪ ((8 − ((ℓ_2 · lg_w) mod 8)) mod 8)      // for lg_w = 4, left shift by 4
msg ← msg ‖ base_2^b(toByte(csum, ⌈(ℓ_2 · lg_w)/8⌉), lg_w, ℓ_2)   // append checksum in base w
skADRS ← ADRS                                          // copy address to create key-generation key address
skADRS.setTypeAndClear(WOTS_PRF)
skADRS.setKeyPairAddress(ADRS.getKeyPairAddress())
for i ← 0 to ℓ − 1 do
    skADRS.setChainAddress(i)
    sk ← PRF(PK.seed, SK.seed, skADRS)                 // compute chain i secret value
    ADRS.setChainAddress(i)
    σ_i^W ← chain(sk, 0, msg[i], PK.seed, ADRS)        // compute chain i signature value
return σ^W
```

Algorithm 14: wots_pkFromSig(σ^W, M, **PK**.seed, **ADRS**)

Input: WOTS$^+$ signature σ^W, message M, public seed **PK**.seed, address **ADRS**.
Output: WOTS$^+$ public key pk_{sig} derived from σ^W.

$csum \leftarrow 0$
$msg \leftarrow \mathtt{base\_2^b}\big(M, \lg_w, \ell_1\big)$ // convert message to base w
for $i \leftarrow 0$ **to** $\ell_1 - 1$ **do**
  $csum \leftarrow csum + w - 1 - msg[i]$ // compute checksum
$csum \leftarrow csum \ll \big(\big(8 - ((\ell_2 \cdot \lg_w) \bmod 8)\big) \bmod 8\big)$ // for $\lg_w = 4$, left shift by 4
$msg \leftarrow msg \,\|\, \mathtt{base\_2^b}\Big(\mathtt{toByte}\big(csum, \left\lceil \frac{\ell_2 \cdot \lg_w}{8} \right\rceil\big), \lg_w, \ell_2\Big)$ // append checksum in base w
for $i \leftarrow 0$ **to** $\ell - 1$ **do**
  ADRS.setChainAddress(i)
  $tmp[i] \leftarrow \mathtt{chain}\big(\sigma_i^W, msg[i], w - 1 - msg[i], \textbf{PK}.\text{seed}, \textbf{ADRS}\big)$
$wotspkADRS \leftarrow$ **ADRS** // copy address for WOTS$^+$ public key
$wotspkADRS$.setTypeAndClear(WOTS_PK)
$\mathsf{pk}_{sig} \leftarrow T_\ell\big(\textbf{PK}.\text{seed}, wotspkADRS, tmp\big)$
return pk_{sig}

signature scheme XMSS. We go into more detail in the next section, where we first start by introducing the XMSS scheme and then explain how to use it to generate *authentication paths* for WOTS$^+$ instances during the signing process.

2.3.5 Extended Merkle Signature Scheme

For everyday use, a one-time signature scheme such as WOTS$^+$ is not practical. To construct a many-time signature scheme from a one-time signature scheme, it is necessary to manage multiple one-time keys and ensure that each key is used to sign at most one distinct message. *Stateful* signature schemes keep book of the used keys and ensure that each key is used only once. *Stateless* signature schemes have to eliminate the need of keeping track of the used keys. Instead, the latter rely on the use of an exponentially large number of keys, making key reuse highly unlikely. However, the signing party has to convince the verifier that the key used to sign a message is actually derived from their secret key. In SPHINCS$^+$, this is achieved by managing one-time keys in a tree structure, generating *authentication paths* for WOTS$^+$ keys using an XMSS tree.

XMSS is a hash-based signature system based on *Merkle trees*. A Merkle tree is a binary tree where each leaf is labelled with the hash of an associated cryptographic key's public key. In the case of XMSS, WOTS$^+$ public keys are used as the leaves of the tree. Each parent

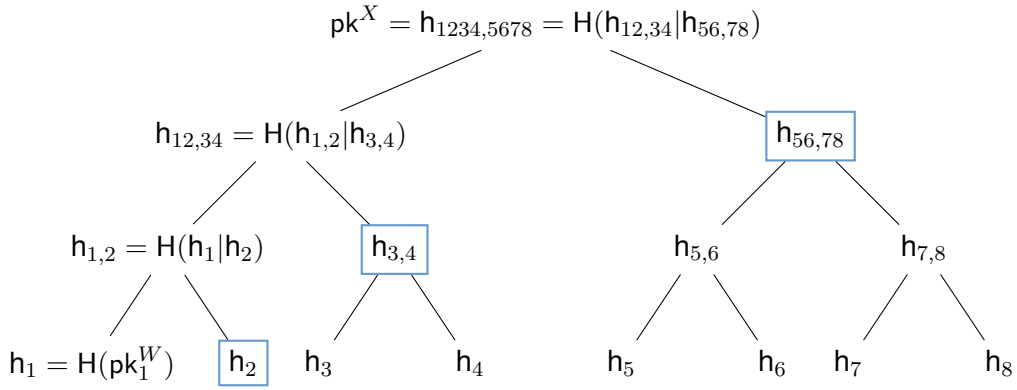


Figure 2.13: A Merkle tree of height 3 with $2^3 = 8$ leaves. The leave h_i represents the hash of a WOTS⁺ key pk_i^W. A parent node $\mathsf{h}_{i,j}$ represents the hash of the concatenation of its children $\mathsf{h}_i, \mathsf{h}_j$. The final hash is denoted as the root node pk^X and represents the public key of the scheme. The nodes forming the authentication path $auth(\mathsf{pk}_1^W)$ are highlighted with blue rectangles.

node is labelled with the hash of the concatenation of the hashes of its children. The root node of the tree is called the *public key* of the scheme. Using a Merkle tree of height h', $2^{h'}$ key pairs can be managed and authenticated (see Figure 2.13).

XMSS uses the **treehash**[1] function (Algorithm 15) to compute the nodes of a Merkle tree starting from its leaves. It takes as input a secret seed, a start index, a target height, a public seed, and an address structure **ADRS**. The algorithm computes the target height node of the tree by iteratively applying the WOTS⁺ key generation algorithm to the leaves and then hashing the resulting public keys up to the root.

Algorithm 16 shows the signing algorithm **xmss_sign** for XMSS. It takes a message M, a secret seed **SK**.seed, an index idx, a public seed **PK**.seed, and an address **ADRS** as input. The idx is the index of the WOTS⁺ key to be used for signing M, i.e., the leaf index in the Merkle tree. The algorithm first computes the authentication path corresponding to idx. It then produces the signature σ^W by signing the message M using the WOTS⁺ signing algorithm **wots_sign** with the secret seed **SK**.seed, the public seed **PK**.seed, and the address **ADRS**. The resulting signature σ^X is then constructed by concatenating the WOTS⁺ signature σ^W and the authentication path $AUTH$.

The function **xmss_pkFromSig** (Algorithm 17) computes the root node of an XMSS tree given a signature σ^X consisting of an authentication path, a message M, a public seed, and an address. It serves as the verification algorithm for XMSS signatures. It first extracts

[1]The **treehash** algorithm from the SPHINCS⁺ submission [Aumasson et al., 2022] is functionally equivalent to the `xmss_node` algorithm presented in the SLH-DSA standardization document [NIST, 2024], but is implemented iteratively instead of recursively. Since the iterative version better reflects the reference implementation of the SPHINCS⁺ submission, we use it throughout this thesis.

Algorithm 15: treehash(**SK**.seed, s, z, **PK**.seed, **ADRS**)

Input: Secret seed **SK**.seed, start index s, target height z, public seed **PK**.seed, address **ADRS**
Output: n-byte root node

if $s \bmod 2^z \neq 0$ **then**
  return -1
for $i \leftarrow 0$ **to** $2^z - 1$ **do**
  ADRS.setType(WOTS_HASH)
  ADRS.setKeyPairAddress$(s + i)$
  $node \leftarrow$ **wots_pkGen**(**SK**.seed, **PK**.seed, **ADRS**)
  ADRS.setType(TREE)
  ADRS.setTreeHeight(1)
  ADRS.setTreeIndex$(s + i)$
  while *height(top of Stack) = height(node)* **do**
    ADRS.setTreeIndex$\big((\textbf{ADRS}.\text{getTreeIndex} - 1)/2\big)$
    $node \leftarrow \mathsf{H}\big(\textbf{PK}.\text{seed}, \textbf{ADRS}, (Stack.\text{pop}() \parallel node)\big)$
    ADRS.setTreeHeight$\big(\textbf{ADRS}.\text{getTreeHeight} + 1\big)$
  $Stack$.push$(node)$
return $Stack$.pop()

Algorithm 16: xmss_sign(M, **SK**.seed, idx, **PK**.seed, **ADRS**)

Input: n-byte message M, secret seed **SK**.seed, index idx, public seed **PK**.seed, address **ADRS**.
Output: XMSS signature $\sigma^X = (\sigma^W \| AUTH)$.

for $j \leftarrow 0$ **to** $h' - 1$ **do**
  $k \leftarrow \lfloor idx/2^j \rfloor \oplus 1$
  $AUTH[j] \leftarrow$ **treehash**$\big(\textbf{SK}.\text{seed}, k, j, \textbf{PK}.\text{seed}, \textbf{ADRS}\big)$
ADRS.setTypeAndClear(WOTS_HASH)
ADRS.setKeyPairAddress(idx)
$\sigma^W \leftarrow$ **wots_sign**$\big(M, \textbf{SK}.\text{seed}, \textbf{PK}.\text{seed}, \textbf{ADRS}\big)$
$\sigma^X \leftarrow \sigma^W \parallel AUTH$
return σ^X

the WOTS⁺ public key from the key pair at leaf index idx using the signature σ^W. Then, it computes the root node of the XMSS tree using the extracted WOTS⁺ public key and the authentication path $AUTH$.

Algorithm 17: xmss_pkFromSig(idx, σ^X, M, **PK**.seed, **ADRS**)

Input: Index idx, XMSS signature $\sigma^X = (\sigma^W \| AUTH)$, n-byte message M, public seed **PK**.seed, address **ADRS**.
Output: n-byte root value $node[0]$.

1 **ADRS**.setTypeAndClear(WOTS_HASH) `// compute WOTS⁺ public key from signature`
2 **ADRS**.setKeyPairAddress(idx)
3 $\sigma^W \leftarrow \sigma^X$.getWOTSSig() `// extract WOTS⁺ signature`
4 $AUTH \leftarrow \sigma^X$.getXMSSAUTH() `// extract authentication path`
5 $node[0] \leftarrow$ **wots_pkFromSig**(σ^W, M, **PK**.seed, **ADRS**) `// WOTS⁺ public key`
6 **ADRS**.setTypeAndClear(TREE) `// compute root from WOTS⁺ pk and AUTH`
7 **ADRS**.setTreeIndex(idx)
8 **for** $k \leftarrow 0$ **to** $h' - 1$ **do**
9 **ADRS**.setTreeHeight($k + 1$)
10 **if** $\lfloor idx/2^k \rfloor$ *is even* **then**
11 **ADRS**.setTreeIndex(**ADRS**.getTreeIndex()/2)
12 $node[1] \leftarrow$ H(**PK**.seed, **ADRS**, $node[0] \| AUTH[k]$)
13 **else**
14 **ADRS**.setTreeIndex((**ADRS**.getTreeIndex() $- 1$)/2)
15 $node[1] \leftarrow$ H(**PK**.seed, **ADRS**, $AUTH[k] \| node[0]$)
16 $node[0] \leftarrow node[1]$
17 **return** $node[0]$

2.3.6 The SPHINCS⁺ Hypertree

SPHINCS⁺ requires up to 2^{68} WOTS⁺ keys for authenticating FORS signatures. Managing this many keys in a single XMSS tree is impractical, as the number of hash operations required to compute a signature would be too high. Instead, SPHINCS⁺ uses a *hypertree* structure. A hypertree is a tree structure where each node is a tree. The SPHINCS⁺ hypertree consists of d layers, with nodes being XMSS trees of height h'. Hence, the *total height* of the SPHINCS⁺ hypertree is $h = d \cdot h'$. The root node of the top layer is the public key of the scheme **PK**.root. The leaves of each layer are WOTS⁺ public keys, and are used to sign a tree on the next layer. This way, only one XMSS tree has to be computed per layer during signing.

The function **ht_sign** (Algorithm 18) shows the signing algorithm for the hypertree. It takes a message M, a secret seed **SK**.seed, a public seed **PK**.seed, a tree index idx_{tree}, and a leaf index idx_{leaf} as input. It first calls **xmss_sign** to sign the message M, producing a signature σ^X containing a WOTS$^+$ signature and an authentication path. Afterwards, the root node of the XMSS tree at layer 0 used for signing is computed using **xmss_pkFromSig**. The algorithm then iterates over the remaining layers of the hypertree, signing the root node of the previous layer using **xmss_sign** and appending the resulting signature to the hypertree signature σ^{HT}. Once the top layer is reached, the hypertree signature σ^{HT} contains the WOTS$^+$ signatures and authentication paths for all layers of the hypertree.

Algorithm 18: ht_sign(M, **SK**.seed, **PK**.seed, idx_{tree}, idx_{leaf})

Input: Message M, secret seed **SK**.seed, public seed **PK**.seed, tree index idx_{tree}, leaf index idx_{leaf}.
Output: Hypertree signature σ^{HT}.

ADRS $\leftarrow$ toByte$(0, 32)$ `// initialize address to 32-byte zero`
ADRS.setTreeAddress(idx_{tree})
$\sigma^X \leftarrow$ **xmss_sign**$(M,$ **SK**.seed, $idx_{\text{leaf}},$ **PK**.seed, **ADRS**$)$
$\sigma^{HT} \leftarrow \sigma^X$
$\mathsf{pk}^X \leftarrow$ **xmss_pkFromSig**$(idx_{\text{leaf}}, \sigma^X, M,$ **PK**.seed, **ADRS**$)$
for $j \leftarrow 1$ **to** $d-1$ **do**
  $idx_{\text{leaf}} \leftarrow idx_{\text{tree}} \bmod 2^{h'}$ `//` h' `least significant bits of` idx_{tree}
  $idx_{\text{tree}} \leftarrow idx_{\text{tree}} \gg h'$ `// remove` h' `least significant bits`
  ADRS.setLayerAddress(j)
  ADRS.setTreeAddress(idx_{tree})
  $\sigma^X \leftarrow$ **xmss_sign**$(\mathsf{pk}^X,$ **SK**.seed, $idx_{\text{leaf}},$ **PK**.seed, **ADRS**$)$
  $\sigma^{HT} \leftarrow \sigma^{HT} \,\|\, \sigma^X$
  if $j < d-1$ **then**
    $\mathsf{pk}^X \leftarrow$ **xmss_pkFromSig**$(idx_{\text{leaf}}, \sigma^X,$ **PK**.seed, **ADRS**$)$ `// compute next root`
return σ^{HT}

The function **ht_verify** (Algorithm 19) verifies a hypertree signature σ^{HT}. It takes a message M, a signature σ^{HT}, a public seed **PK**.seed, a tree index idx_{tree}, a leaf index idx_{leaf}, and the hypertree public key **PK**.root as input. It starts by extracting the first WOTS$^+$ signature from the hypertree signature σ^{HT} and computes the root node of the XMSS tree at layer 0 using **xmss_pkFromSig**. Then, it iterates over the remaining layers of the hypertree, extracting the WOTS$^+$ signatures and authentication paths from σ^{HT} and computing the root node of each layer using **xmss_pkFromSig**. Finally, it compares the computed root node with the hypertree public key **PK**.root and returns true if they match, or false otherwise.

Algorithm 19: ht_verify(M, σ^{HT}, **PK**.seed, idx_{tree}, idx_{leaf}, **PK**.root)

Input: Message M, signature σ^{HT}, public seed **PK**.seed, tree index idx_{tree}, leaf index idx_{leaf}, HT public key **PK**.root.
Output: Boolean (validity of σ^{HT}).

1 **ADRS** $\leftarrow$ **toByte**$(0, 32)$
2 **ADRS**.setTreeAddress(idx_{tree})
3 $\sigma^X \leftarrow \sigma^{HT}$.getXMSSSignature$(0)$ // $\sigma^{HT}[0 : (h' + \ell) \cdot n)]$
4 $node \leftarrow$ **xmss_pkFromSig**$\big(idx_{\text{leaf}}, \sigma^X, M,$ **PK**.seed, **ADRS**$\big)$ // compute first root
5 **for** $j \leftarrow 1$ **to** $d - 1$ **do**
6 $\quad idx_{\text{leaf}} \leftarrow idx_{\text{tree}} \bmod 2^{h'}$ // h' least significant bits of idx_{tree}
7 $\quad idx_{\text{tree}} \leftarrow idx_{\text{tree}} \gg h'$ // remove h' least significant bits
8 $\quad$ **ADRS**.setLayerAddress(j)
9 $\quad$ **ADRS**.setTreeAddress(idx_{tree})
10 $\quad \sigma^X \leftarrow \sigma^{HT}$.getXMSSSignature$(j)$ // extract j-th XMSS signature
11 $\quad node \leftarrow$ **xmss_pkFromSig**$\big(idx_{\text{leaf}}, \sigma^X, node,$ **PK**.seed, **ADRS**$\big)$ // compute next root
12 **if** $node =$ ***PK.root*** **then**
13 $\quad$ **return** true
14 **else**
15 $\quad$ **return** false

2.3.7 Forest Of Random Subsets

Forest of Random Subsets (FORS) is a hash-based few-time signature scheme that is used to sign the hash digest of the actual message in SPHINCS$^+$. While a WOTS$^+$ key can only be used once, FORS keys can be used multiple but overall few times. FORS uses a number of XMSS trees, a combination which is then called a *forest*, at addresses specified by **ADRS** to sign a message digest. The signature process is similar to XMSS, and the FORS public key is the hash of the concatenation of the public keys of the XMSS trees used in the forest. The FORS signature scheme is parameterized by the number of XMSS trees k in the forest and the total number of leaves 2^a.
Similarly to the WOTS$^+$ scheme, FORS defines three functions **`fors_keygen`**, **`fors_sign`**, and **`fors_pkFromSig`** to generate keys, sign messages, and extract a public key from a signature. We omit details about the FORS signature scheme here, as it is not the focus of the attack presented in this thesis, and we refer the reader to the NIST standardization document [NIST, 2024] for a detailed description of the FORS scheme.

2.3.8 SPHINCS$^+$ Interface

SPHINCS$^+$ exposes three functions to the user: a *key generation* function, a *signing* function, and a *verification* function.
Algorithm 20 shows the key generation algorithm **`sphincsp_keygen`** for SPHINCS$^+$. It takes a secret **SK**.seed, a PRF key **SK**.prf, and a public **PK**.seed as input. It generates the public key for the top-level XMSS tree by calling **`treehash`** with the secret seed, a zero index, the height h', the public seed, and an address **ADRS**.

Algorithm 20: sphincsp_keygen(**SK**.seed, **SK**.prf, **PK**.seed)

Input: Secret seed **SK**.seed, PRF key **SK**.prf, public seed **PK**.seed.
Output: SPHINCS$^+$ key pair (SK, PK).

1 **ADRS** $\leftarrow$ **`toByte`**$(0, 32)$ `// generate pk for top-level XMSS tree`
2 **ADRS**.setLayerAddress$(d-1)$
3 **PK**.root $\leftarrow$ **`treehash`**$($**SK**.seed, 0, h', **PK**.seed, **ADRS**$)$
4 **return** $(($**SK**.seed, **SK**.prf, **PK**.seed, **PK**.root$)$, $($**PK**.seed, **PK**.root$))$

The function **`sphincsp_sign`** (Algorithm 21) shows the signing algorithm for SPHINCS$^+$. It takes a message M, a private key SK = (**SK**.seed, **SK**.prf, **PK**.seed, **PK**.root), and an optional additional randomness $addrnd$ as input. The algorithm first generates a randomizer R using the pseudorandom function PRF on the PRF key **SK**.prf, the additional randomness *addrnd*, and the message M. It then computes a message digest using the hash function H_{msg} on the randomizer R, the public **PK**.seed, the public key **PK**.root, and

the message M. The digest is split into three parts: the first $\lceil k \cdot a/8 \rceil$ bytes are used as the message digest, the next $\lceil (h - h/d)/8 \rceil$ bytes are used as the tree index, and the last $\lceil h/(8d) \rceil$ bytes are used as the leaf index.

Algorithm 21: sphincsp_sign(M, SK, $addrnd$)

Input: Message M, private key SK = (**SK**.seed, **SK**.prf, **PK**.seed, **PK**.root), (optional) additional randomness $addrnd$.
Output: SLH-DSA signature σ.

ADRS $\leftarrow$ **toByte**$(0, 32)$ // initialize address to 32-byte zero
$opt\_rand \leftarrow addrnd$ // subst. $opt\_rand \leftarrow$ **PK**.seed for deterministic variant
$R \leftarrow \text{PRF}_{msg}(\textbf{SK}.\text{prf}, opt\_rand, M)$ // generate randomizer
$\sigma \leftarrow R$ // initialize signature with R
$digest \leftarrow \text{H}_{\text{msg}}(R, \textbf{PK}.\text{seed}, \textbf{PK}.\text{root}, M)$ // compute message digest
$md \leftarrow$ first $\lceil \frac{k \cdot a}{8} \rceil$ bytes of *digest*
$tmp\_idx_{\text{tree}} \leftarrow$ next $\left\lceil \frac{h - h/d}{8} \right\rceil$ bytes of *digest*
$tmp\_idx_{\text{leaf}} \leftarrow$ next $\lceil \frac{h}{8d} \rceil$ bytes of *digest*
$idx_{\text{tree}} \leftarrow$ **toInt**$(tmp\_idx_{\text{tree}}, \left\lceil \frac{h - h/d}{8} \right\rceil) \bmod 2^{h - h/d}$ // tree index
$idx_{\text{leaf}} \leftarrow$ **toInt**$(tmp\_idx_{\text{leaf}}, \lceil \frac{h}{8d} \rceil) \bmod 2^{h/d}$ // leaf index
ADRS.setTreeAddress(idx_{tree})
ADRS.setTypeAndClear(FORS_TREE)
ADRS.setKeyPairAddress(idx_{leaf})
$\sigma^F \leftarrow$ **fors_sign**$(md, \textbf{SK}.\text{seed}, \textbf{PK}.\text{seed}, \textbf{ADRS})$ // sign using FORS
$\sigma \leftarrow \sigma \,\|\, \sigma^F$ // append FORS signature
$\text{pk}^F \leftarrow$ **fors_pkFromSig**$(\sigma^F, md, \textbf{PK}.\text{seed}, \textbf{ADRS})$ // derive FORS public key
$\sigma^{HT} \leftarrow$ **ht_sign**$(\text{pk}^F, \textbf{SK}.\text{seed}, \textbf{PK}.\text{seed}, idx_{\text{tree}}, idx_{\text{leaf}})$ // sign using hypertree
$\sigma \leftarrow \sigma \,\|\, \sigma^{HT}$ // append HT signature
return σ

Algorithm 22 shows **sphincsp_verify**, the verification function for SPHINCS⁺ signatures. It takes a message M, a signature σ, and a public key PK = (**PK**.seed, **PK**.root) as input. The algorithm first splits the signature into the randomizer R, the FORS signature σ^F, and the hypertree signature σ^{HT}. It then computes the message digest using the hash function H_{msg} on R, **PK**.seed, **PK**.root, and M. Similarly to the signing algorithm, the digest is split into three parts: the message digest, the tree index, and the leaf index. Then, using the FORS signature, the public key for the FORS forest is computed using the function **fors_pkFromSig**. Finally, the hypertree signature is verified using the function **ht_verify**, which checks, using the authentication path from the signature, if the

computed root node matches the public key **PK**.root.

Algorithm 22: sphincsp_verify(M, σ, $PK = (\textbf{PK}.\text{seed}, \textbf{PK}.\text{root})$)

Input: Message M, signature σ, public key $PK = (\textbf{PK}.\text{seed}, \textbf{PK}.\text{root})$.
Output: Boolean expressing the validity of σ.

1 **if** $|\sigma| \neq \big(1 + k(1+a) + h + d \cdot \ell\big) \cdot n$ **then**
2 **return** false
3 **ADRS** $\leftarrow$ **toByte**$(0, 32)$ // initialize address to 32-byte zero
4 $R \leftarrow \sigma$.getR() // extract R (first n bytes of σ)
5 $\sigma^F \leftarrow \sigma$.getSIG_FORS() // extract FORS signature (n-byte blocks $[n : (1 + k(1+a)) \cdot n]$)
6 $\sigma^{HT} \leftarrow \sigma$.getSIG_HT() // extract HT signature (remaining bytes)
7 $digest \leftarrow \mathsf{H}_{\text{msg}}\big(R, \textbf{PK}.\text{seed}, \textbf{PK}.\text{root}, M\big)$ // compute message digest
8 md $\leftarrow$ first $\lceil \frac{k \cdot a}{8} \rceil$ bytes of *digest*
9 $tmp\_idx_{\text{tree}} \leftarrow$ next $\left\lceil \frac{h - h/d}{8} \right\rceil$ bytes of *digest*
10 $tmp\_idx_{\text{leaf}} \leftarrow$ next $\lceil \frac{h}{8d} \rceil$ bytes of *digest*
11 $idx_{\text{tree}} \leftarrow$ **toInt**$\big(tmp\_idx_{\text{tree}}, \lfloor (h - h/d)/8 \rfloor\big) \mod 2^{h-h/d}$ // tree index
12 $idx_{\text{leaf}} \leftarrow$ **toInt**$\big(tmp\_idx_{\text{leaf}}, \lfloor h/(8d) \rfloor\big) \mod 2^{h/d}$ // leaf index
13 **ADRS**.setTreeAddress(idx_{tree}) // set tree address for FORS
14 **ADRS**.setTypeAndClear(FORS_TREE) // select FORS tree type and clear other fields
15 **ADRS**.setKeyPairAddress(idx_{leaf}) // set key-pair (leaf) address
16 $\mathsf{pk}^F \leftarrow$ **fors_pkFromSig**$\big(\sigma^F, md, \textbf{PK}.\text{seed}, \textbf{ADRS}\big)$ // compute FORS public key
17 **return** **ht_verify**$\big(\mathsf{pk}^F, \sigma^{HT}, \textbf{PK}.\text{seed}, idx_{\text{tree}}, idx_{\text{leaf}}, \textbf{PK}.\text{root}\big)$ // verify hypertree signature

2.3.9 Parameter Sets

SPHINCS$^+$ defines several parameter sets for different hash function families and security levels. The original SPHINCS$^+$ submission proposes 36 parameter sets which fall into two categories: *fast* and *small*. While the former is optimized for signing speed, the latter is optimized for signature size. The categories are further divided into *simple* and *robust* variants. While the simple variants apply the hash functions once to the concatenation of parameters and inputs, the robust parameter sets are designed to be more secure against attacks in the random oracle model by first hashing the parameters and subsequently XOR-ing the inputs with the parameter digest before hashing again. Each combination of category and variant can be used with SHA2 [NIST, 2015a], SHAKE [NIST, 2015b], or

Haraka [Kölbl et al., 2016] as the hash function. Thus, for each hash function, 12 parameter sets are proposed.
However, not all parameter sets are approved by NIST. Table 2.2 of the approved parameter sets shows that only the *simple* variants of the SHA2 and SHAKE parameter sets were approved, while the Haraka parameter sets were not approved at all. In this work, we focus on the SHAKE-256s parameter set, which satisfies the highest security level of 5. Other parameter sets are only used to compare the performance of the attack against other parameter sets.
The specification defines the following parameters:

- n: the security parameter in bytes.
- h: the height of the hypertree. This determines the number of FORS key pairs available for signing, hence the probability that a FORS key pair is reused.
- d: the number of XMSS tree layers in the hypertree. It is purely a performance trade-off parameter and must divide h without remainder.
- h': the depth of an XMSS tree. This can be computed as $h' = \frac{h}{d}$, but is included in the parameter set for clarity.
- a, k: the security level of a FORS instance. A FORS forest consists of 2^a leaves, which are split into k trees. There is a performance trade-off between a and k, as a smaller value for a generally leads to smaller and faster signatures, but increase the signature size for a given security level.
- $\lg_w$: the 2-logarithm of the *Winternitz Parameter* w. It determines the number of bits that can be signed with a single WOTS+ chain.
- m: the output length of $\mathsf{H}_{\mathrm{msg}}$ in bytes.

2.3.10 Differences between SPHINCS+ and SLH-DSA

SPHINCS+ has been first submitted to NIST's PQC standardization process in 2017 [Bernstein et al., 2017]. The SPHINCS+ specification version 3.0 was later submitted to the third round of the NIST PQC standardization process in 2020. The specification was revised to version 3.1 [Aumasson et al., 2022] and standardized as SLH-DSA in 2024 [NIST, 2024]. Specification version 3.1 revision introduced several minor changes:

- Two new address types were introduced: *WOTS_PRF* and *FORS_PRF*. Those are used to derive the WOTS+ and FORS initial secrets from the secret key seed.

Table 2.2: Parameter sets for SLH-DSA

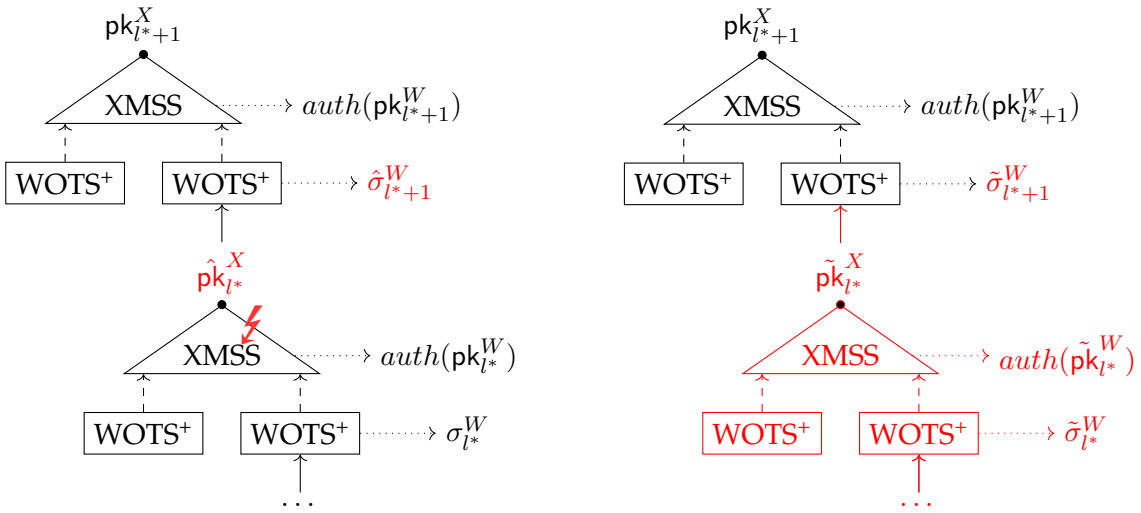
Name	n	h	d	h'	a	k	$\lg_w$	m	Security category	pk (bytes)	sig (bytes)
SHA2-128s SHAKE-128s	16	63	7	9	12	14	4	30	1	32	7856
SHA2-128f SHAKE-128f	16	66	22	3	6	33	4	34	1	32	17088
SHA2-192s SHAKE-192s	24	63	7	9	14	17	4	39	3	48	16224
SHA2-192f SHAKE-192f	24	66	22	3	8	33	4	42	3	48	35664
SHA2-256s SHAKE-256s	32	64	8	8	14	22	4	47	5	64	29792
SHA2-256f SHAKE-256f	32	68	17	4	9	35	4	49	5	64	49856

- **PK**.seed was added as an input to PRF to mitigate multi-key attacks.
- For the category 3 and 5 SHA2 parameter sets, SHA-256 was replaced by SHA-512 in H_{msg}, $\mathsf{PRF}_{\text{msg}}$, H, and T_ℓ.
- For all SHA2 parameter sets, the randomizer R and **PK**.seed were added as inputs when computing H_{msg} to mitigate second preimage attacks.

The SLH-DSA specification also differs from the revised submission version 3.1 in the method for extracting bits from the message digest while signing with the FORS scheme. Additionally, some minor bugs in the pseudocode of `wots_sign` and `wots_pkFromSig` were fixed in the SLH-DSA specification. The standard only approves the use of 12 of the 36 proposed parameter sets. Most notably, the Haraka-based parameter sets and the robust variants are not approved.

2.4 Grafting Tree Attack

The *grafting tree attack* is a fault attack on the SPHINCS and SPHINCS$^+$ signature schemes first described in [Castelnovi et al., 2018]. In this section, we give an overview of the attack and analyze its complexity. We first give an overview of the attack, and then analyze the complexity of each step in detail. We assume that, in addition to faulted SPHINCS$^+$ signatures, the attacker can also obtain valid signatures σ from a SPHINCS$^+$ signing oracle.



(a) Faulting the XMSS computation.

(b) Grafting an XMSS tree.

Figure 2.14: Grafting tree attack on SPHINCS⁺ [Genêt, 2023].

While this is not strictly necessary for the attack, it simplifies the analysis. For an analysis of the attack without the availability of valid signatures, we refer to [Genêt, 2023].

2.4.1 Attack Overview

We will now provide an overview of the attack, before describing the individual steps in the following subsections. Figure 2.14 illustrates the grafting tree attack on SPHINCS⁺. Recall that during the computation of an XMSS tree at layer l^*, the **`treehash`** function computes the tree's root node $\mathsf{pk}^X_{l^*}$ by iteratively applying a hash function to the tree's nodes, emitting an authentication path $auth(\mathsf{pk}^W_{l^*})$ on the way. The attacker then injects faults into one of these hash function calls, leading to a faulted root node $\hat{\mathsf{pk}}^X_{l^*}$. This value is subsequently signed using a WOTS⁺ instance at layer $l^* + 1$, producing a WOTS⁺ signature $\hat{\sigma}^W_{l^*+1}$. After the signing procedure is completed, the attacker then collects a faulted signature

$$\hat{\sigma} = (R, \sigma^F, \sigma^W_0, auth(\mathsf{pk}^W_0), \ldots, \sigma^W_{l^*}, auth(\mathsf{pk}^W_{l^*}), \hat{\sigma}^W_{l^*+1}, auth(\mathsf{pk}^W_{l^*+1}), \ldots).$$

If at least two different WOTS⁺ signatures for the same WOTS⁺ instance at layer $l^* + 1$ are encountered, we call the WOTS⁺ instance *compromised*, and we call the number of WOTS⁺ signatures collected for this instance the *number of WOTS⁺ collisions*. The attacker repeats the fault injection until a sufficient number of WOTS⁺ collisions is collected, having even more collisions reduces the complexity of the attack.

In order to mount the attack, the attacker has to identify WOTS$^+$ collisions from the collected signatures. For that, they first extract all WOTS$^+$ signatures from the collected signatures, grouping them by their **ADRS**. Then, they find compromised WOTS$^+$ instances by finding addresses mapping to at least two different WOTS$^+$ signatures. The attacker can then use a compromised WOTS$^+$ instance to sign a *grafted* XMSS tree with public key $\tilde{\mathsf{pk}}_{l*}^{X}$. However, they can only sign messages that are compatible with the compromised WOTS$^+$ instance.

Recall that when signing a message m in WOTS$^+$, the message and its checksum are first split into ℓ blocks of size w bits. Each block m_i is then signed using the **chain** function, applying a hash function F a number of m_i times to an initial chain secret s_i, producing the chain signature $\sigma^{(i)}$. A WOTS$^+$ signature σ^W then consists of the chain signatures $\sigma^{(i)}$ for each chain $0 \leq i < \ell$, and we call the $m_i - 1$ preimages of $\sigma^{(i)}$ under F the *WOTS$^+$ secret values* of chain i. If the attacker now acquires two different signatures $(\sigma^W, \hat{\sigma}^W)$ for a given WOTS$^+$ instance, they can combine the signatures of chain $0 \leq i < \ell$ to acquire the exposed WOTS$^+$ secret values $\hat{\theta}_i = \min\left\{\sigma^{(i)}, \hat{\sigma}^{(i)}\right\}$. If more than two signatures for a given WOTS$^+$ instance are available, the attacker can keep combining the chain signatures to expose more WOTS$^+$ secret values. We call the compromised WOTS$^+$ instance with the most exposed WOTS$^+$ secret values the *targeted WOTS$^+$ instance*. Now, the attacker simply needs to graft a compatible XMSS root $\tilde{\mathsf{pk}}_{l*}^{X}$ for all the exposed WOTS$^+$ secret values $\hat{\theta}_i$, corresponding to message blocks $\hat{m}_i$.: Find a $\tilde{\mathsf{pk}}_{l*}^{X}$ such that all message and checksum blocks $\tilde{r}_i$ of $\tilde{\mathsf{pk}}_{l*}^{X}$ satisfy $\tilde{r}_i \geq \hat{m}_i$ for all $0 \leq i < \ell$. The grafted XMSS tree is then signed using the **chain** function with the exposed WOTS$^+$ secret values $\hat{\theta}_i$. This tree can be used to sign arbitrary messages, constituting the universal forgery against SPHINCS$^+$.

In the following, we analyze the steps of the grafting tree attack on SPHINCS$^+$ signatures. We start with the identification of WOTS$^+$ collisions in Section 2.4.2. Subsequently, we analyze the complexity of the grafting step in Section 2.4.3. We see that the total complexity of the attack is dominated by the complexity of the grafting step, and that the attack is feasible with just a few WOTS$^+$ collision.

2.4.2 Identifying WOTS$^+$ Collisions

In this step, the attacker identifies WOTS$^+$ collisions from the collected signatures. First, the WOTS$^+$ signatures are extracted from the collected signatures and grouped by **ADRS**. Then, the attacker identifies addresses that map to at least two different WOTS$^+$ signatures. Subsequently, the attacker calculates the exposed WOTS$^+$ secret values for each **ADRS**. Let $(\sigma^W, \hat{\sigma}^W)$ be the valid and a faulted WOTS$^+$ signatures for the same WOTS$^+$ instance at **ADRS**, and let m_i be the message chunk corresponding to the message signed by σ^W in chain i. The attacker identifies WOTS$^+$ secret values in $\hat{\sigma}^W = (\hat{\sigma}^{(0)}, \ldots, \hat{\sigma}^{(\ell-1)})$

using the following exhaustive search [Genêt, 2023]:

1. For each $\hat{\sigma}^{(i)} \in \hat{\sigma}$: find $1 \leq k < m_i$ such that

$$\texttt{chain}\left(\hat{\sigma}^{(i)}, k, w-1-k, \mathbf{PK}.\text{seed}, \mathbf{ADRS}\right) = p_i.$$

2. If such k exists, then $\hat{\sigma}^{(i)}$ exposes the WOTS$^+$ secret values $k, \ldots, m_i - 1$ of chain i. If no k leads to a match, the WOTS$^+$ signature is discarded.

Complexity. Extracting the WOTS$^+$ signatures $\hat{\sigma}^W$ from a SPHINCS$^+$ signature faulted at hypertree layer $l^* \leq d-1$ requires running a truncated SPHINCS$^+$ signature verification with l^* layers. This amounts to an average number of hash function calls of:

$$\underbrace{2 + k(a+1)}_{\text{FORS verification}} + \underbrace{l^*(\ell \cdot (w-1)/2 + 1 + h')}_{\text{HT verification up to layer } l^*}.$$

Additionally, assuming that the secret values $\hat{\sigma}^{(i)}$ are distributed uniformly in the compromised keys, identifying the WOTS$^+$ secret values requires an average number of hash function calls of

$$\ell \cdot \left(\sum_{x=0}^{w-1} \frac{1}{w}(w-1-x)\right) = \ell \cdot \frac{w-1}{2}.$$

For the SHAKE-256s parameter set with target layer $l^* = 7$, this results in an average of $2^{11.85} + 2^{8.97} \approx 4200$ hash function calls.

2.4.3 Tree Grafting

After identifying the WOTS$^+$ secret values, the attacker performs the grafting tree attack. In this step, the attacker tries to find an XMSS public key that can be signed using the targeted WOTS$^+$ instance. This involves an exhaustive search for a secret $\hat{\mathbf{SK}}$.seed that produces an XMSS public key $\hat{\mathsf{pk}}^X$. However, only a limited number of XMSS trees can be signed, as the attacker can only sign messages that are compatible with the exposed chain secrets. Let $(\hat{\theta}_0, \hat{\theta}_1, \ldots, \hat{\theta}_{\ell-1})$ be the exposed secret elements of the targeted WOTS$^+$ instance, corresponding to message chunks $(\hat{m}_0, \hat{m}_1, \ldots, \hat{m}_{\ell-1})$. The algorithm to find a suitable XMSS tree is described in [Genêt, 2023] and works as follows:

1. Draw $\hat{\mathbf{SK}}$.seed uniformly at random.
2. Create a new XMSS tree with public key $\tilde{\mathsf{pk}}^X$ from the secret $\hat{\mathbf{SK}}$.seed.
3. Split $\tilde{\mathsf{pk}}^X$ and its WOTS$^+$ checksum into chunks $(\tilde{r}_0, \ldots, \tilde{r}_{\ell-1})$ of size w bits.

4. If $\tilde{r}_i \geq \hat{m}_i$ for all $0 \leq i < \ell$, return the grafted $\hat{\mathbf{SK}}$.seed. Repeat from step 1 otherwise.

Complexity. The complexity of the grafting tree attack depends on the number of exposed WOTS$^+$ secret values in the targeted WOTS$^+$ instance as well as on the target layer l^*. The computation of the root node of an XMSS tree using **`treehash`** requires the computation of $2^{h'}$ WOTS$^+$ public keys, each of which requires the computation of $\ell + \ell \cdot (w-1) + 1$ hash function calls. Additionally, the *tree walk*, i.e, calculating the inner nodes of the tree from its leaves needs $\sum_{i=1}^{h'} 2^{i-1}$ hash function calls to compute the root node of the XMSS tree. This amounts to a total complexity of the **`treehash`** function of

$$\mathbb{C}(\text{treehash}) = 2^{h'} \underbrace{(\ell + \ell \cdot (w-1) + 1)}_{\text{WOTS}^+ \text{ public key generation}} + \underbrace{\sum_{i=1}^{h'} 2^{i-1}}_{\text{Tree walk}} = 2^{h'}(\ell \cdot w + 2) - 1. \tag{2.1}$$

The probability that a random $\hat{\mathbf{SK}}$.seed leads to a suitable XMSS tree is given by the probability that all $\hat{r}_i$ are greater than or equal to the corresponding $\hat{m}_i$. Assuming that the attacker collects $M > 1$ different signatures for the targeted WOTS$^+$ instance and the exposed chain elements x are distributed uniformly, the probability of a grafting attempt to be successful is given by [Genêt, 2023] as:

$$\mathbb{P}(\text{Grafting}) \leq \frac{1}{w^\ell} \left(\sum_{x=0}^{w-1} \left(1 - \left(\frac{w-1-x}{w} \right)^M \right) \right)^\ell \approx e^{-\frac{\ell}{M+1}} + \mathcal{O}(1). \tag{2.2}$$

The grafting tree attack constitutes a Bernoulli experiment, where each sampled tree is a Bernoulli trial with success probability $\mathbb{P}(\text{Grafting})$. We can therefore estimate the expected number of trees sampled until a signable tree is found by taking the inverse of the success probability:

$$\mathbb{E}(\text{Sampled Trees}) = \frac{1}{\mathbb{P}(\text{Grafting})} \geq w^\ell \left(\sum_{x=0}^{w-1} \left(1 - \left(\frac{w-1-x}{w} \right)^M \right) \right)^{-\ell}.$$

Since each trial requires a number of hash computations given by Equation (2.1), this amounts to an expected attack complexity of

$$\begin{aligned} \mathbb{C}(\text{Grafting}) &= \mathbb{C}(\text{treehash}) \cdot \mathbb{E}(\text{Sampled Trees}) \\ &\geq \left(2^{h'}(\ell \cdot w + 2) - 1 \right) \cdot w^\ell \left(\sum_{x=0}^{w-1} \left(1 - \left(\frac{w-1-x}{w} \right)^M \right) \right)^{-\ell}. \end{aligned}$$

Plugging in the parameters of the different parameter sets introduced in Section 2.3.9

Table 2.3: Average complexity of tree grafting. The table shows the complexity of the XMSS tree grafting step for the different parameter sets.

	M = 2	4	6	8	10	16	32
SHAKE-128s	$2^{36.34}$	$2^{27.55}$	$2^{24.22}$	$2^{22.48}$	$2^{21.41}$	$2^{19.81}$	$2^{18.58}$
SHAKE-128f	$2^{30.34}$	$2^{21.55}$	$2^{18.22}$	$2^{16.48}$	$2^{15.41}$	$2^{13.81}$	$2^{12.58}$
SHAKE-192s	$2^{45.21}$	$2^{32.39}$	$2^{27.54}$	$2^{25.01}$	$2^{23.46}$	$2^{21.12}$	$2^{19.33}$
SHAKE-192f	$2^{39.21}$	$2^{26.39}$	$2^{21.54}$	$2^{19.01}$	$2^{17.46}$	$2^{15.12}$	$2^{13.33}$
SHAKE-256s	$2^{52.92}$	$2^{36.09}$	$2^{29.72}$	$2^{26.39}$	$2^{24.35}$	$2^{21.28}$	$2^{18.93}$
SHAKE-256f	$2^{48.92}$	$2^{32.09}$	$2^{25.72}$	$2^{22.39}$	$2^{20.35}$	$2^{17.28}$	$2^{14.93}$

and different values of M, we obtain the average expected complexities for the grafting tree attack shown in Table 2.3. Note that the complexity mostly matches those from [Genêt, 2023], except for the SHAKE-128s and SHAKE-192s parameter sets due to outdated parameters used in the original paper. The table shows that the attack is feasible if the attacker can collect a sufficient number of faulty signatures even for the most secure parameter sets.

3 SWAGE: An End-to-End Framework for Rowhammer Attacks

Conducting Rowhammer attacks in practice is challenging. As the attack interacts with different aspects of the target system, many moving parts need to be considered. Many tools have been proposed to facilitate Rowhammer attacks, but they often cover only a specific aspect of the attack, such as finding reproducible hammering patterns or reverse-engineering the physical memory layout. Most of the time, these tools are tightly coupled to their software design of the practical evaluation that is presented, making it difficult to reuse them in an end-to-end Rowhammer attack. Until now, a modular framework that covers all aspects of a Rowhammer attack while being easy to use and extend has been missing. In this chapter, we aim to close this gap by introducing SWAGE, a novel modular end-to-end framework for Rowhammer attacks. To contextualize the use case of SWAGE, a standard Rowhammer threat model is assumed, where the attacker has user-level access to a target system running a recent Linux kernel version and can execute arbitrary code – such as SWAGE – with user permissions. The attacker also has root access to a replicated target system during the offline phase, which is used to reverse-engineer the physical memory layout of the target system and find reproducible hammering patterns. Within this realistic and widely accepted threat landscape, we introduce a set of tools bundled with SWAGE that can be used either independently or as integrated components of a broader Rowhammer attack framework. SWAGE is implemented in Rust, C, and Python, and is designed to be modular and extensible. We provide SWAGE as an open-source project[2] and invite the community to contribute to its development.

Figure 3.1 shows the flowchart of a Rowhammer attack using SWAGE. The attack starts with the DRAM INSPECTOR (Section 3.1), which is used to reverse-engineer the physical memory layout of the target system. The physical memory layout is then used by the ALLOCATOR module (Section 3.2) to find contiguous memory regions that can be used for Rowhammer attacks. The HAMMERER module (Section 3.3) is used to find reproducible memory access patterns that induce bit flips in the target memory module. Lastly, the VICTIM module (Section 3.4) provides an API to implement victim-specific code to interact with the target program, e.g., to check the success of the attack.

[2] `https://git.uni-luebeck.de/its/research-projects/rowhammer/swage`

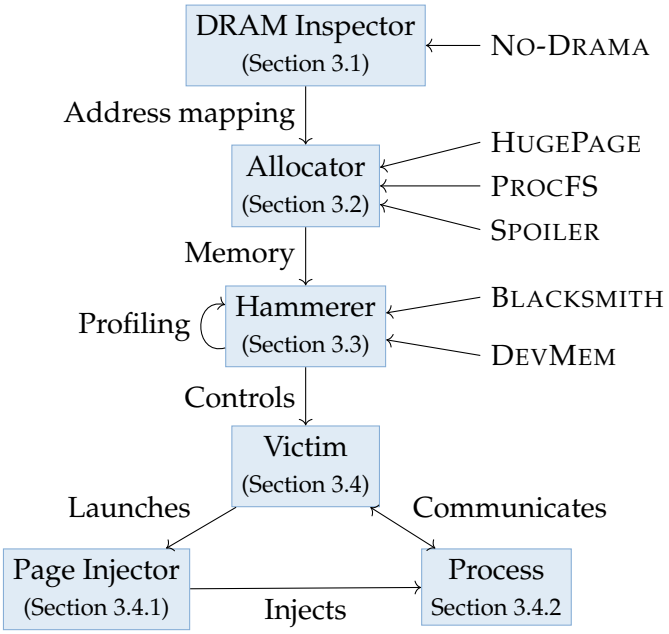


Figure 3.1: Flowchart showing the steps of a Rowhammer attack using SWAGE.

3.1 The DRAM INSPECTOR Module

SWAGE includes a DRAM INSPECTOR module that provides information about the physical memory layout of the target system. As discussed in Section 2.2.2, the operating system provides an abstraction of the physical memory layout to user applications using *virtual addresses*. Similarly, the memory controller abstracts physical memory to the operating system. Two rows with consecutive physical addresses are not necessarily physically adjacent in the memory module. To determine the address of a row in physical memory, the memory controller applies a *memory mapping* function to the physical address. This function is specific to any given CPU generation and usually not disclosed by manufacturers. However, there exist side channel attacks that can be used to reverse-engineer the physical memory layout. In the next section, we discuss DRAMA, a tool that uses a timing-based side channel to reverse-engineer the physical memory layout, as well as a graph-based bit detection scheme introduced in [Xiao et al., 2016] that can be used to determine the physical mapping function.

3.1.1 The DRAMA Attack

In 2016, Pessl et al. introduced DRAMA, a tool that uses the bank conflict timing side channel to reverse-engineer the physical memory layout [Pessl et al., 2016]. As we have seen in Section 2.2.1, memory is organized in banks, and each bank contains multiple rows. When

a row is accessed, the content of the requested row is loaded into the bank's row buffer. If another row in the same bank is accessed while the row buffer contains the content of another row, the memory controller has to write back the content of the row buffer to the-DRAM array before loading the new row. The write-back-then-read sequence takes longer than a regular access and can be measured using a timing side channel. DRAMA uses this timing side channel to measure the time it takes to access a pair of memory addresses. If the measured time above a given threshold, the rows are in the same bank. Otherwise, they are in different banks. After sampling a sufficient number of pairs of memory addresses, DRAMA can determine the physical mapping function by solving a system of linear equations.

3.1.2 Graph-Based Bank Bit Detection Scheme

In [Xiao et al., 2016], a graph-based bit detection scheme is introduced. It uses the same primitive as DRAMA, measuring the time it takes to access pairs of memory addresses. However, the graph-based bit detection scheme does not require a system of linear equations to be solved, but instead builds a graph where nodes represent bits of memory addresses and edges represent high-latency timing measurements between pairs of bits. Using the graph, the scheme determines the physical mapping function by identifying subgraphs representing combinations of bits involved in determining a bit of the row, column, and bank functions.
SWAGE incorporates NO-DRAMA [Wilke, 2023], combining techniques introduced in DRAMA and the graph-based bit detection scheme by Xiao et al. NO-DRAMA generates output describing the physical mapping function in a machine-readable format, which is parsed by SWAGE to provide information about the physical memory layout of the target system. The `struct DRAMAddr` represents a DRAM address as a tuple of bank, row, and column, providing blanket implementations for pointers and integer types to allow easy conversion between physical and DRAM addresses.

3.2 The ALLOCATOR Module

In order to conduct a practical Rowhammer attack, the attacker needs to allocate contiguous memory for the physical addressing function to work properly. The ALLOCATOR module in SWAGE provides several options for the allocation of contiguous memory blocks suitable for Rowhammer attacks. The ALLOCATOR module provides a unified interface for allocating contiguous memory blocks, which can be used in both the offline phase and the online phase of an attack.

Listing 3.1: The ConsecAlloc trait and related structs in SWAGE.

```
trait ConsecAllocator {
    fn block_size(&self) -> usize;
    fn alloc_consec_blocks(&mut self, size: usize) -> Result<ConsecBlocks>;
}
struct ConsecBlocks {
    pub blocks: Vec<Memory>
}
struct Memory {
    pub ptr: *mut u8,
    pub len: usize
}
```

Central to the ALLOCATOR module is the `trait ConsecAllocator` (Listing 3.1), which provides an interface for allocating contiguous memory blocks. The `block_size()` method returns the size of the contiguous memory blocks that can be allocated, while the `alloc_consec_blocks()` method allocates a number of contiguous memory blocks of the given size, so-called *split blocks*. Implementations of the `ConsecAllocator` trait must guarantee that the memory blocks returned by `alloc_consec_blocks()` are physically contiguous, i.e., each block is backed by a single contiguous physical memory region of size `block_size()`.

SWAGE comes with several implementations of the `ConsecAllocator` trait, each using a different strategy to allocate contiguous memory blocks. In the following subsections, we introduce allocation strategies implemented in SWAGE. In Section 3.2.1, we introduce *huge pages*, a mechanism provided by the OS to allocate large chunks of physically contiguous memory. Then, in Section 3.2.2, we discuss the `pagemap` interface, which can be used to find contiguous memory for users with root privileges. We discuss the `pagetypeinfo` and `buddyinfo` interfaces in Section 3.2.3, which can be used to find contiguous memory using the buddy allocator. Finally, in Section 3.2.4, we introduce the SPOILER attack, which uses a timing-based side channel to find contiguous memory. The SPOILER attack is experimentally determined to be the most reliable method to find contiguous memory without root level access, but still has some limitations and false positives. We discuss these limitations and how to mitigate them.

3.2.1 Let The Kernel Handle It: Huge Pages

The Linux kernel provides *huge pages*, a mechanism to allocate large chunks of physically contiguous memory. Huge pages are up to one gigabyte in size and are used to reduce the overhead of page table management. However, huge pages are mostly used in high-performance computing, e.g., for large in-memory databases, and need to be enabled in

the kernel configuration by the system administrator. Therefore, while huge pages can be used in the offline phase of an attack to find Rowhammer access patterns, they can not be assumed to be available on the target system.
Thus, the HUGEPAGE allocator implemented in SWAGE can be used in the offline phase of an attack to examine the reproducibility of Rowhammer access patterns.

3.2.2 First Generation Attacks: The `pagemap` Interface

Early attacks used the Linux kernel's `/proc/PID/pagemap` ProcFS interface to find memory suitable for a Rowhammer attack [Seaborn and Dullien, 2015]. This file provides the mapping between virtual and physical addresses for the process with process ID `PID`. In 2015, as a response to Rowhammer attacks exploiting the `pagemap` interface, the Linux kernel developers changed the interface to only be accessible to the root user [Shutemov, 2015].
While not usable during the online phase on the target system, SWAGE implements a PFN allocator that uses the `/proc/PID/pagemap` interface to find contiguous memory in a large buffer. The PFN allocator is the most reliable one, as it has no false positives, but requires root privileges. It is therefore a handy tool for the offline phase of an attack, for quick iterations of experiments, and for debugging.

3.2.3 Attacks Using the Buddy Allocator: `pagetypeinfo` and `buddyinfo`

Whenever a user process allocates memory, the kernel has to find a number of free memory pages to satisfy the request. Managing a simple list of free 4 KiB pages would be space inefficient: Assuming a system with only 32 GiB RAM, this would require a list of 8 000 000 entries. Additionally, some applications might have specific requirements for memory allocation, e.g., legacy drivers that require memory to be allocated in lower memory regions or kernel code requiring *unmovable* pages. Therefore, the kernel uses more sophisticated data structures to manage free memory.
The *buddy allocator* is a memory management algorithm in the Linux kernel. It manages physical memory blocks of 4 KiB to 4 MiB in size. At system start up, the buddy allocator divides the memory into blocks of 4 MiB. When a process requests memory, the buddy allocator finds the smallest available block that can satisfy the request. If no block of the requested size is available, the buddy allocator splits a larger block into two smaller *buddy blocks*, and serves the request using one (or both) of the buddies. When a block is freed, the buddy allocator merges it with its buddy block if it is also free.
The buddy allocator maintains lists of free blocks of different sizes, called *orders*. The size of a memory block of order i is 2^i pages, i.e., a block of order 0 is 4 KiB, while a block of

the highest order 10 is 4 MiB. It also groups the blocks by three categories *node*, *zone*, and *migration type*:

Node The physical memory node in a *Non-Uniform Memory Access* (NUMA) system.

Zone The memory zone a page belongs to: *DMA* (the lowest 24 MiB of memory), *DMA32* (the lowest 4 GiB of memory), or *Normal* (4 GiB and above).

Migration Type The type of memory page, e.g., *movable* (can be moved by the kernel), *unmovable* (cannot be moved), or *reclaimable* (cache pages that can be dropped or reloaded).

The ProcFS interface `/proc/pagetypeinfo` provides detailed information about the state of the *buddy allocator*, and can be used as a side channel disclosing whether a memory allocation was served from a contiguous block of memory [Kwong et al., 2020]. The `pagetypeinfo` interface provides information about the number of free memory blocks of *orders* 0 to 10, grouped by *node*, *zones*, and *migration type*.
During regular system operations, `/proc/pagetypeinfo` can be used by system operators to debug allocator issues and adapt software to compensate for fragmentation, e.g., by changing the size of memory pools. In an attack scenario, however, it can also be used to find contiguous memory. For that, an attacker observes the change in available blocks using `/proc/pagetypeinfo` while allocating a 4 MiB block of memory. If the number of available blocks of order 10 decreases by 1 and all other available block orders remain untouched, the attacker can assume that the allocation was served from a contiguous 4 MiB block of memory. However, this interface was made unavailable to non-privileged users in 2019 due to unrelated performance considerations [Hocko, 2019].
The `/proc/buddyinfo` interface, on the other hand, provides coarse-grained information about the number of available memory blocks of each order, and is still available to non-privileged users. Unlike `/proc/pagetypeinfo`, `/proc/buddyinfo` does not group the blocks by node, zones, and migration type; instead, it reports only the cumulative number of free blocks for each order. It can still be used to find contiguous memory, but is less precise than `/proc/pagetypeinfo`, as draining of lower order blocks is harder to detect, leading to a higher false-positive rate. While these can be mitigated by testing a candidate memory region for consistency with the reverse-engineered physical address mapping, using `/proc/buddyinfo` as a side channel is still less reliable than employing microarchitectural side channels. Nevertheless, it has been used as an alternative to `/proc/pagetypeinfo` in Rowhammer attacks [Tobah et al., 2022].
SWAGE still implements a BUDDYINFO allocator, which uses the `/proc/buddyinfo` interface to find contiguous memory, but since the BUDDYINFO allocator shows a high

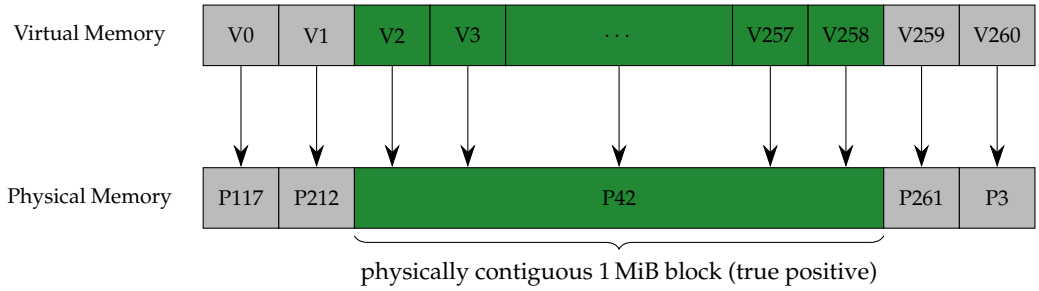


Figure 3.2: A single 1 MiB-aligned window (green) inside a larger buffer is backed by one contiguous 1 MiB physical block, which SPOILER correctly identifies using the load hazard timing side channel. Each square denotes a 4 KiB page.

false-positive rate due to the unreliability in finding locked pages, its use in end-to-end Rowhammer attacks is not recommended.

3.2.4 Exploiting Microarchitectural Leakage: The SPOILER Attack

The SPOILER attack [Islam et al., 2019] exploits pipeline hazards causing load stalls in the CPU to determine a prefix of the physical address of a memory location. Modern CPUs employ *speculative loads* and *load forwarding* to improve memory performance, executing `load` instructions speculatively before related `store` instructions. When a `load` leaves the instruction pipeline while a `store` to a potentially overlapping address is still buffered, the core must decide whether it is safe to forward data from the store buffer or whether the `load` must wait until the stores retire. The comparison between the speculative-load address and the buffered-store addresses is performed in several stages of increasing accuracy, i.e., the processor first checks only a subset of the physical-address bits (typically the page offset and the next 8 bits, giving a 1 MB aliasing window). If these partial bits match, the pipeline conservatively stalls the load for a few cycles until a full physical-address comparison has been completed. SPOILER creates long chains of stores followed by a timed load and looks for exactly these stalls: a prolonged latency reveals that the load's physical address shares the same 1 MB region as one of the preceding stores. By sweeping through virtual pages and measuring the resulting timing patterns, an unprivileged attacker can recover the 12-bit page offset plus up to 8 additional physical-address bits. This aliasing knowledge accelerates Rowhammer attacks by providing a primitive to find contiguous memory regions.
In SWAGE, we use the technique presented in SPOILER to find contiguous memory blocks of up to 4 MiB in size (Figure 3.2). However, there are some limitations to the SPOILER attack that we first need to address, and which we solve by combining the SPOILER primitive with the physical address mapping reverse-engineered in Section 3.1:

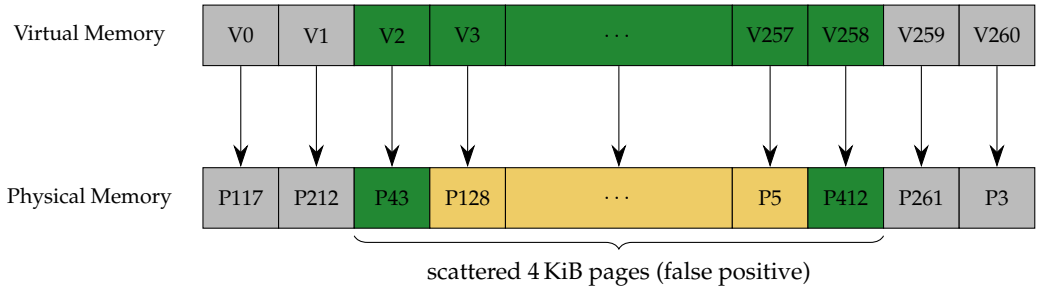


Figure 3.3: False positive: only the outer pages (green) share the 1 MiB alias; interior pages (yellow) reside in unrelated physical frames, yet SPOILER still flags the whole region as contiguous. Each square denotes a 4 KiB page.

(1) *Granularity*. SPOILER only finds contiguous memory blocks of 1 MiB. Depending on the concrete physical address mapping function, it may be necessary to find larger contiguous memory blocks, or at least find a set of memory blocks that is consistent with the physical address mapping function. We can solve this problem by using the SPOILER primitive to find a number of contiguous 1 MiB memory blocks, and then using the row buffer conflict timing side channel in Algorithm 1 to check whether the memory blocks are consistent with the physical address mapping function. This enhances the SPOILER primitive to find larger contiguous memory blocks of up to 4 MiB in size.

(2) *Alignment*. The high-latency pattern arises whenever the attack encounters a `load-store` pair sharing a physical page offset; the hit need not be aligned to 1 MiB boundaries. This means that the SPOILER attack may find a 1 MiB block, but the block may have an offset δ between virtual and physical addresses. One way to solve this problem is to use a 2 MiB *Transparent Huge Page* (THP) for the `load` instruction, which is guaranteed to have appropriate alignment. It is reasonable to assume that THP are available on the target system, as they are enabled by default in modern Linux kernels. However, the SPOILER attack can also be used without THP: After finding a 1 MiB block candidate, we can use the row buffer conflict timing side channel in Algorithm 1 to check whether the candidate is consistent with the physical address mapping function after applying an offset $0 \leq \delta \leq 2\,\text{MiB}$ to the virtual address. If we find a δ that is consistent with the physical address mapping function, we can assume that the 1 MiB block candidate is valid.

(3) *False positives*. Two 4 KiB pages that merely sit on the same 1 MiB aliasing produce the same timing peak as a truly contiguous 1 MiB region (Figure 3.3). We tackle this problem by performing a row buffer conflict check on the candidate memory region.

If the timing check is consistent with the physical address mapping function, i.e., if we observe timing peaks where we expect them, we find that the candidate memory region is actually backed by a contiguous 1 MiB physical page block.

SWAGE implements a SPOILER allocator including the means to overcome the limitations of the primitive, allowing for the allocation of contiguous 4 MiB memory blocks. As shown experimentally in Section 4.3.1, we find that SPOILER is a powerful tool and more reliable alternative to attacks using the `ProcFS` interfaces.

3.3 The HAMMERER Module

The HAMMERER module is the core of SWAGE and provides the functionality to hammer a reproducible memory access pattern and to interact with the victim process. The module is designed to be used in conjunction with the ALLOCATOR module, which provides contiguous memory blocks suitable for Rowhammer attacks, and the VICTIM module, which provides an API to interact with a victim process.

Listing 3.2: Hammering trait and related types in the HAMMERER module.

```
trait Hammering {
    fn hammer(&self, victim: &mut dyn HammerVictim)
        -> Result<HammerResult, HammerVictimError>;
}
struct HammerResult {
    pub attempt: u32,
    pub victim_result: VictimResult,
}
enum VictimResult {
    BitFlips(Vec<BitFlip>), // Non-empty list of bit flips
    String(String),  // a string returned by the victim process
    // ...other results that may be returned by the victim
}
enum HammerVictimError {
    NoFlips,
    IoError(std::io::Error), // I/O error when interacting with the victim
    // ...other errors that may occur during hammering
}
```

Listing 3.2 shows the `Hammering` trait and related types. The trait defines a simple API to implement custom hammering strategies, consisting of only a single method `hammer` that takes a mutable reference to a victim object and returns a `HammerResult`.
SWAGE comes with built-in support for BLACKSMITH [Jattke et al., 2022], a fuzzing-based software suite to find reproducible hammering patterns. Under the hood, it just-in-time

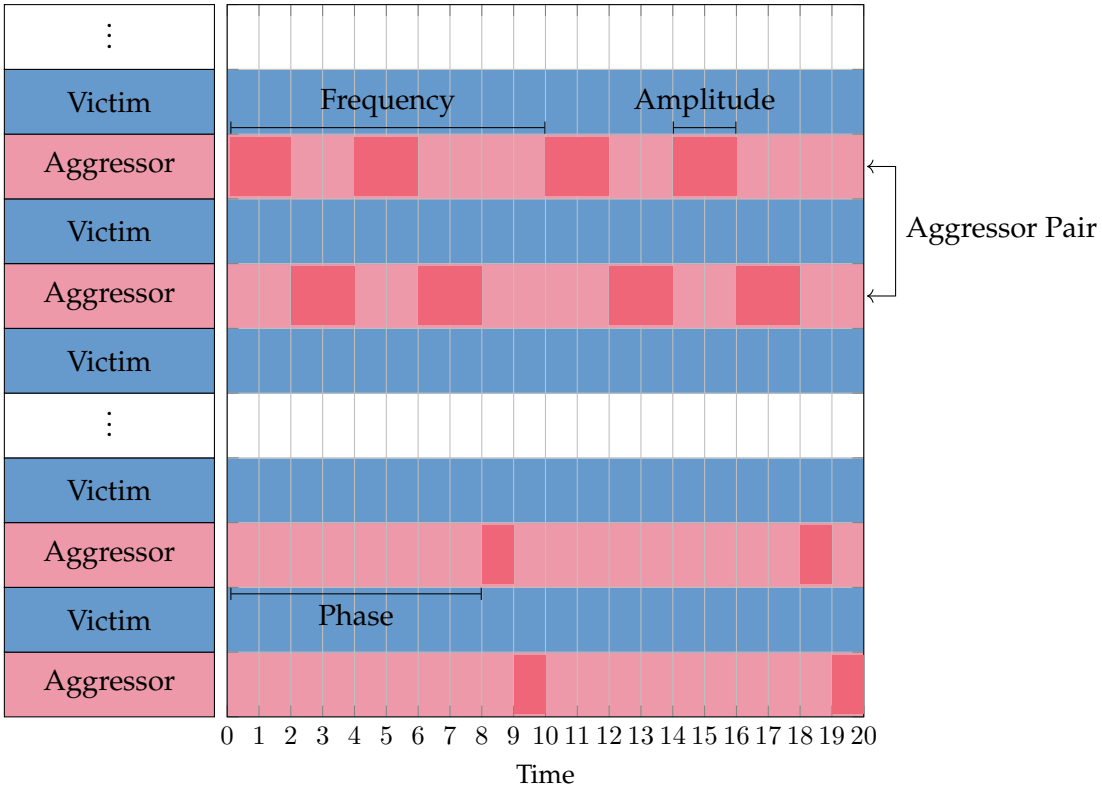


Figure 3.4: Non-uniform access pattern in BLACKSMITH. The figure shows a time series of memory accesses to aggressors rows (red). Victim rows are marked in blue . This memory access pattern overwhelms many TRR mechanisms by accessing aggressor rows in a non-uniform manner that is hard to predict.

compiles the access pattern and aggressor mapping found in a BLACKSMITH fuzzing run into an assembly function that can be executed on the target system. Figure 3.4 shows how BLACKSMITH generates *non-uniform access patterns* by randomizing three domains: How often an aggressor row is activated (frequency), the time between the start of the hammering pattern and the first activation of an aggressor row (phase), and how often an aggressor row is activated back-to-back (amplitude). BLACKSMITH empirically shows that non-uniform access patterns are effective in bypassing TRR in all 40 tested DDR4 memory modules. While the original BLACKSMITH suite uses huge pages for fuzzing reproducible hammering patterns, SWAGE uses the abstraction provided by the ALLOCATOR module, adapting the hammering pattern to contiguous memory blocks of arbitrary size. SWAGE also incorporates a profiling step in the hammering phase to verify the reproducibility of the hammering pattern under real-world conditions, such as split blocks provided by the ALLOCATOR module. For testing and debugging, the HAMMERER module also provides a DEVMEM hammerer that manipulates bits in memory directly.
The simple API defined by the `Hammering` trait allows easy implementation of other hammering strategies such as TRRESPASS [Frigo et al., 2020] or ROWPRESS [Luo et al., 2024]. In conjunction with SWAGE's allocator module, the HAMMERER module can be used to find reproducible memory access patterns on contiguous memory blocks suitable for an end-to-end Rowhammer attack.

3.4 The VICTIM Module

The VICTIM module provides an API to interact with and control a victim process. Implementations in this module serve two main purposes: First, they encapsulate the interaction with the victim process, such as synchronization with the hammering process and gathering results from the victim process after hammering. Second, they handle page injection, ensuring that an attacker-controlled target page is injected into the victim's address space before hammering.

Listing 3.3: HammerVictim trait in the VICTIM module.

```
1 trait HammerVictim {
2     fn start(&mut self) -> Result<(), HammerVictimError>;
3     fn init(&mut self);
4     fn check(&mut self) -> Result<VictimResult, HammerVictimError>;
5     fn stop(&mut self);
6 }
```

Listing 3.3 shows the `HammerVictim` trait, which defines the interface for a victim process. It consists of the following four methods:

`start()` starts the victim process, potentially conducting the page injection attack.

`init()` initializes the victim before a hammering round. This method allows the victim to prepare for the hammering phase, e.g., synchronize the victim process with the hammering process. After `init()` returns, the victim is ready to be hammered.

`check()` checks whether a hammering round was successful. This method is called after hammering and returns a `VictimResult` (Listing 3.2) if the hammering was successful. Determining the success of the hammering is victim-specific and usually involves communicating with the victim process, such as reading from its standard output or a TCP socket.

`stop()` stops the victim process. This method is called after the hammering is done to clean up the victim process.

SWAGE comes with a number of built-in victim implementations, such as `MemCheck`, which checks for bit flips in a target memory region, and `SphincsPlus`, which communicates with a SPHINCS$^+$ signing server to collect signatures and check for faults, writing them to a file for later analysis. However, the `HammerVictim` trait is designed to be extensible, allowing users to implement their own victim processes. For example, a custom victim component could interact with a deep neural network, injecting a backdoor into the model [Tol et al., 2023], or interact with an FPGA to directly test the robustness of hardware countermeasures against Rowhammer attacks [Weissman et al., 2020].

3.4.1 Page Injection

Page injection describes techniques that aim to manipulate a victim process and the OS allocator so that the bits the attacker knows how to flip land *exactly* inside a security-critical data object in the victim process. Depending on the target program, this can be a pointer, a data structure, or a byte array containing key material. Page injection is a crucial step in a Rowhammer attack and is therefore bundled by SWAGE in the VICTIM module.

Listing 3.4: PageInjector trait in the VICTIM module.

```
1 trait PageInjector<T> {
2     fn inject(&mut self) -> Result<T, std::io::Error>;
3 }
```

Listing 3.4 shows the `PageInjector` trait. It defines a single method `inject()`, which injects a target page into the victim's address space and returns a generic type `T`. The generic type allows the page injector to return a victim-specific object, such as a pointer

to the injected page, a status code indicating the success of the injection, or a socket to communicate with the launched victim process.
Several page injection techniques have been proposed. Most notably, an optimization in the Linux kernel for fast page allocation can be exploited to inject a target page into a victim process [Adiletta et al., 2024]. When a process allocates a page, the buddy allocator searches for an order-0 page in the free list. If no order-0 page is available, it will search for a larger page in the free list and split it into smaller pages. This operation requires a global lock on the free list, which is expensive. Therefore, the Linux kernel maintains a per-core free list for order-0 pages. When a process releases a page, it is added to the per-core free list of the CPU core that released it. When another process on the same core subsequently allocates a page, the buddy allocator will serve the page from the per-core free list. This behavior can be exploited to inject a target page into a victim process co-located on the attacker's core. SWAGE implements a `BuddyPageInjector`, which launches a victim process, injects a target page into the victim's address space, and returns a **`Child`** object to communicate with the launched victim process on success.
A recent novel page injection technique, RUBICON [Bölcskei et al., 2025], uses a combination of page injection and eviction strategies to implement a cross-CPU page injection attack. While RUBICON is not implemented in SWAGE yet, the framework provides an API to easily implement such techniques for cross-CPU threat models.
Page injection is a highly victim-specific attack and depends on the target program's memory layout and access patterns. SWAGE therefore ships with a set of tools to analyze a target program to find the number of *bait pages* required to successfully inject a target page at a required memory location, e.g., at a specific position in the stack. In an end-to-end attack, the results of this offline analysis can be used to determine parameters for a page injection attack.

3.4.2 Target Analysis

Before running the Rowhammer attack, the target program has to be investigated to find susceptible memory regions. However, finding memory regions that are suitable for Rowhammer attacks is a non-trivial task, as it involves knowledge about the protocol to be attacked, the memory layout of the program implementing the protocol, and interaction with the operating system.
One way to find suitable memory regions is via *static analysis* of the target program. While this approach is easy to implement, it can also become tedious and error-prone, as relevant target programs usually contain many variables and data structures. Also, the memory layout, cache eviction, and timing behavior of a program are not directly visible in the source code and can change between different function calls, environments, and com-

piler versions or optimization levels. Another way of finding suitable memory regions is using *dynamic analysis*. Dynamic analysis is usually performed by running the target program in a controlled environment and intercepting its memory accesses. This approach is more flexible and allows us to find suitable memory regions automatically. However, the reverse-mapping of memory accesses to the source code is also not trivial. Different compiler versions, optimization levels, and architectures can lead to different memory layouts. Additionally, combining dynamic analysis with instrumentation of the target program can change access patterns, rendering the results of the analysis invalid.
SWAGE provides a set of tools to analyze a target program and find suitable memory regions for Rowhammer attacks, combining static and dynamic analysis techniques. We distinguish between two dimensions of suitability:

Spatial suitability Rowhammer attacks on modern DDR4 memory modules usually flip up to two bits in a given victim row. However, the page offset of the bit flip cannot be selected arbitrarily and is sometimes limited by the physical properties of the DRAM cells. While some rows only flip bits at a specific offset, others contain multiple susceptible cells. Additionally, some flippy cells are susceptible to bidirectional bit flips, while others only flip in one direction. To ensure availability, we might even want to only flip specific row offsets in a specific direction, because rogue bit flips in a hammering pattern can lead to unintended side effects such as application crashes or even kernel panics. Therefore, we need to find a memory region that is large enough to cover the required row offset. This can be done by analyzing the source code of the target program and looking for large data structures that are used in the program, as well as dynamic analysis at runtime.

Temporal suitability Finding temporally suitable memory regions is usually more challenging. The target must remain in memory for a sufficiently long time to allow the Rowhammer attack to be effective. This rules out short-lived data structures kept in cache or registers. The target should also not be accessed too frequently, as this would lead to premature refreshes of the target row. Analyzing the target program's memory access patterns is crucial to find suitable memory regions. While this can also be done by analyzing the source code, it is usually more effective using instrumentation and dynamic analysis.

Combining a manual static analysis of a target program with dynamic analysis while incorporating low-footprint instrumentation is a promising approach to find suitable attack targets. To keep the instrumentation footprint minimal while still capturing the observations we need for Rowhammer profiling, SWAGE includes a lightweight C helper library MEMUTILS. It can be loaded dynamically into the target program or linked statically. The

library exposes a handful of primitives used to instrument the target program and debug a Rowhammer attack. It provides the following primitives to do so:

`get_physical_address()` translates a virtual address to a physical address using the `/proc/self/pagemap` interface. This yields a 48-bit DRAM address. Knowing the physical address of a variable is important for Rowhammer profiling, and allows the attacker to find the spatial suitability of an attack candidate in the offline phase.

`measure_access()` and the `FLUSH` macro are a pair of assembly helpers: The function `measure_access` wraps a configurable number of loads between two `rdtscp` instructions, while `FLUSH` evicts a given cache line with a single `clflush` instruction. The latency histogram we obtain lets us tell stable DRAM residency from cache hits and prefetch noise, mapping to temporal suitability. Those functions are also useful for measuring the latency of memory accesses in general, and can be used to debug cache-related issues with the Rowhammer attack.

`get_stack_offset()` parses `/proc/self/maps` to locate the stack region of the current process, then converts a virtual address into a page offset within that region. The region page offset is useful for target programs with numerous stack allocations, as it allows the attacker to identify the stack offset of a target variable. The stack offset can be used to configure the number of bait pages required for the page injection attack (Section 3.4.1).

`MEMUTILS_PRINT_OFFSET` bundles the above in a single `printf` statement for convenience. It prints the physical address of a given virtual memory address, the stack offset, and the latency of a memory access in machine-readable format. By instrumenting the target program with this macro, the attacker can easily collect execution traces of the target program and analyze them later with tooling provided by SWAGE.

`mtrr_*` helpers configure the *Memory Type Range Registers* (MTRRs) of the CPU. MTRRs can be used to control the cacheability of memory pages. There are two MTRR helper functions: `mtrr_set_cacheability()` sets the cacheability of a given memory region to `MTRR_UNCACHEABLE`, while the `mtrr_set_default()` function sets the default cacheability of all memory regions to `MTRR_WRBACK`. This function family is useful for debugging cache-related issues with the Rowhammer attack.

4 A Rowhammer-Based Universal Forgery Attack Against SPHINCS$^+$

In this chapter, we evaluate SWAGE by conducting an end-to-end universal forgery attack against the SPHINCS$^+$ signature scheme on a real-world target system.
After introducing the threat model and experimental setup in Section 4.1, we first discuss the offline phase of the attack in Section 4.2. The attacker starts the offline phase by reverse-engineering the physical memory layout of the target system, using NO-DRAMA to reverse-engineer the physical memory layout in Section 4.2.1. Using this layout, the attacker uses the BLACKSMITH fuzzing site to find reproducible access patterns. Afterwards, in Section 4.2.3, the SWAGE profiling tools are used to find suitable memory regions in the target program.
In the online phase discussed in Section 4.3, the attacker first allocates contiguous memory blocks for the attack (Section 4.3.1). They then profile the allocated memory for reproducible bit flips in a target page at suitable page offsets in Section 4.3.2. After finding a reproducible bit flip, the attacker performs a page injection attack to inject the target page into the victim's address space in Section 4.3.3. The attacker then collects signatures from the victim process in Section 4.3.4.
Finally, in Section 4.4, the collected signatures are analyzed in a post-processing phase to find reused one-time signature keys and the grafting tree attack is performed. This final part of the evaluation demonstrates the practicality of fault attacks exploiting the Rowhammer bug against SPHINCS$^+$, showing the necessity of countermeasures against Rowhammer attacks in cryptographic implementations.

4.1 Threat Model and Experimental Setup

We assume a standard Rowhammer threat model where the attacker has user-level access to the target system. The attacker is assumed to be CPU co-located with a victim process on the target system. They can execute arbitrary code with user permissions – including SWAGE – but cannot acquire root access to the target system. During the offline phase, they also have root access to a replicated target system. The target system provides accurate timers, such as the `rdtscp` instruction, to measure memory access latencies for timing side channels. The victim program is a server started by the attacker that accepts input via standard input and outputs results to standard output. The target system is a system with

recent DDR4 memory modules with employed hardware Rowhammer countermeasures (TRR). It is assumed that *Address Space Layout Randomization* (ASLR) is disabled on the target system, as previous works have studied the effect of ASLR on Rowhammer attacks [Amer et al., 2024], and it was shown that ASLR can be bypassed with little engineering efforts [Adiletta et al., 2024].

During the online phase, the attacker's goal is to inject a target page into the victim's address space and hammer the target page to induce bit flips during the victim's execution. Based on the output of the victim process, the success of an attack is determined by a custom victim module implemented by the attacker.

After the online phase, the attacker analyzes the collected signatures and conducts the grafting tree attack. For this attack, the attacker has access to a machine with hardware-accelerated cryptographic hash primitives, such as a recent CPU supporting the `AVX2` instruction set or a CUDA-enabled GPU.

Experimental Setup: The experiments in this evaluation are conducted on a machine equipped with an Intel i5-6400 CPU and a single G.Skill AEGIS 16 GB DDR4-2133 DIMM, running Ubuntu 20.04.3 with Linux kernel version 6.8.12-generic with default settings. The machine runs the SPHINCS+ submission v3.1 reference C implementation from the NIST submission package. A small signing server based on this reference implementation is provided that accepts signing requests via standard input and outputs signatures to standard output. The server is compiled to run the SPHINCS+ signature scheme with randomization enabled and parameters `SHAKE-256s` at the highest security level 5. For the grafting phase of the attack, a machine with an AMD EPYC 7763 CPU with 64 cores and 120 GiB of RAM is used. The machine is running Ubuntu 22.04.4 LTS with Linux kernel version 5.19.0-generic.

4.2 Offline Phase

In the offline phase, the attacker has root access to a replicated target system. To mount the attack, the attacker first needs to reverse-engineer the physical memory layout (Section 4.2.1) of the target system using NO-DRAMA. Using the knowledge gained, they then use BLACKSMITH to find a reproducible memory access pattern in Section 4.2.2. We show that the attacker can reproduce the access pattern using SWAGE's profiling module by using the *block splitting* technique. Block splitting allows the attacker to split the aggressor mapping of an access pattern into smaller memory blocks, abstaining from the use of huge pages. To conclude the offline phase, a fault analysis of the SPHINCS+ reference implementation is performed in Section 4.2.3 using the MEMUTILS tools provided in SWAGE to find suitable memory regions for the Rowhammer attack.

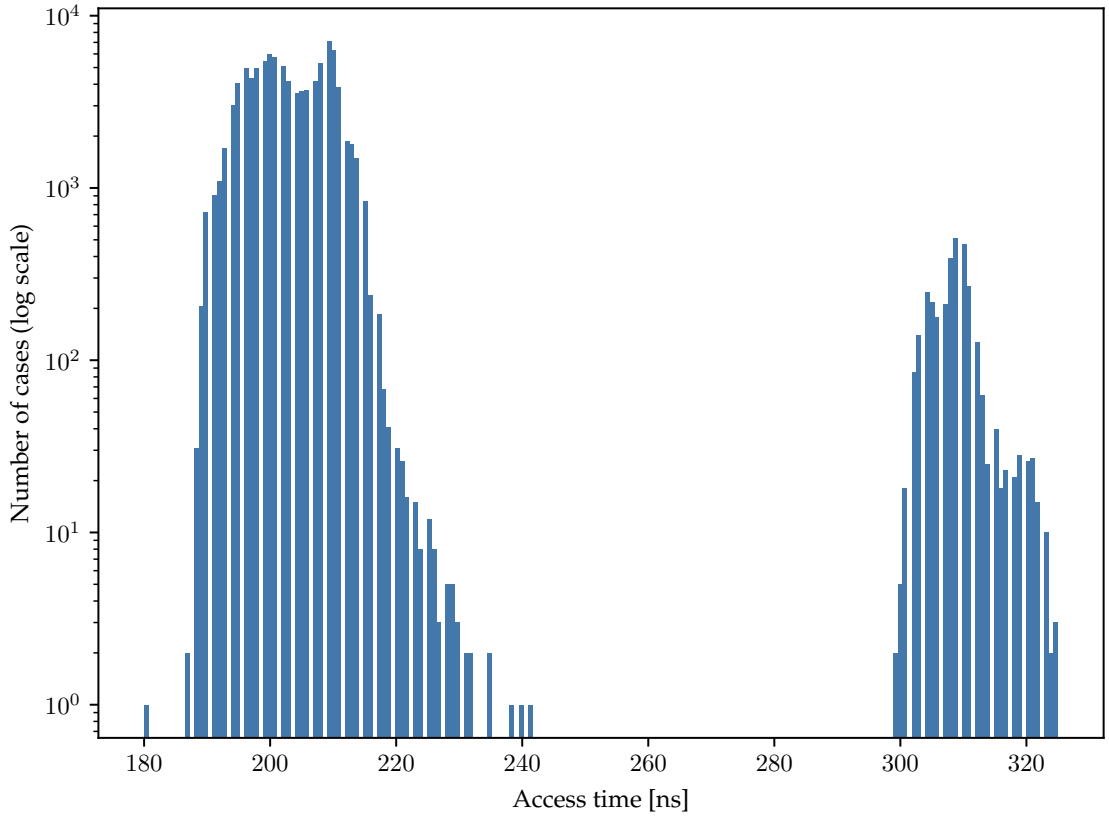


Figure 4.1: Log-scale histogram of NO-DRAMA access times measured over 10^5 random address pairs. The x-axis indicates access latency in nanoseconds, and the y-axis shows the count of address pairs on a logarithmic scale. The bimodal shape reflects two timing regimes: the lower-latency peak corresponds to inter-bank accesses, while the higher-latency peak corresponds to intra-bank accesses.

4.2.1 Reverse-Engineering the Physical Memory Layout

For the Rowhammer attack, the attacker needs to know the physical memory layout of our target system. The physical memory layout is determined by the mapping of physical address to memory banks, rows, and columns. The attacker reverse-engineers this mapping using NO-DRAMA.

Figure 4.1 shows the results of the timing-based side channel used in NO-DRAMA for 10^5 randomly chosen pairs of memory addresses on the target system. We can clearly see that the access time follows a bimodal distribution. The group with the fast timing contains pairs of addresses that are in different banks, while the group with the slow timing contains pairs of addresses that are in the same bank. The access time is significantly higher for pairs of addresses in the same bank, indicating that the memory controller has to write

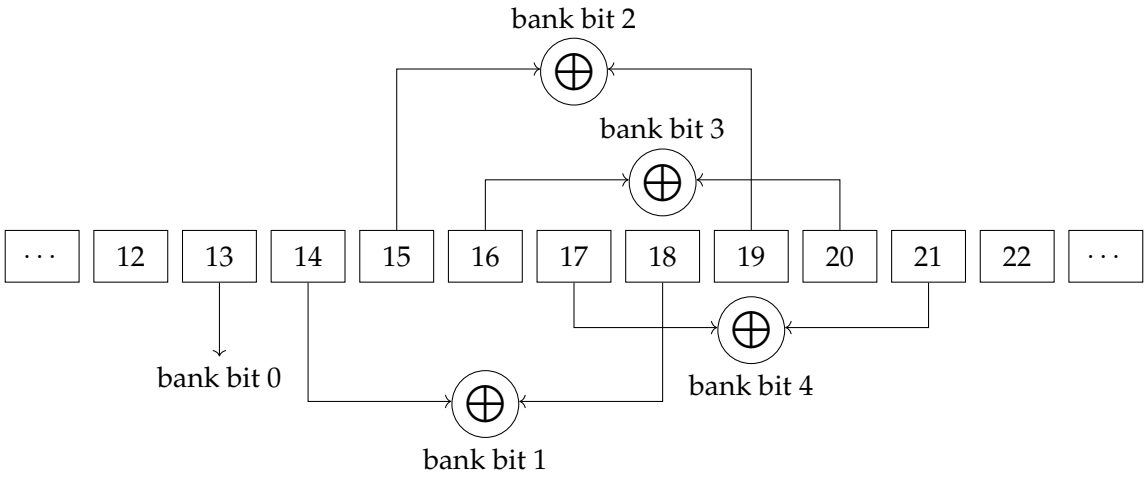


Figure 4.2: DRAM bank-selection mapping for the target system. Bank bit 0 is taken directly from physical bit 13, while bank bits 1-4 are computed as the XOR ($\oplus$) of bit pairs ($b_{14} \oplus b_{18}$, $b_{15} \oplus b_{19}$, $b_{16} \oplus b_{20}$, and $b_{17} \oplus b_{21}$), respectively. Dots indicate omitted neighboring physical address bits.

back the content of the row buffer to the DRAM array before loading the new row.

The attacker uses these timing measurements to determine the physical mapping function by solving a system of linear equations using NO-DRAMA. Figure 4.2 shows the reverse-engineered bank mapping function for the target system. The row mapping function for the target system is simple: the row index is determined by physical bits 18 to bit 29. Similarly, the column index is determined by physical bits 0 to 12.

In another experiment, the physical-to-DRAM mapping is verified on the target system by checking the timing of pairs of addresses with same or different bank indices. For two addresses with the same bank index, the access time is significantly higher than for two addresses with different bank indices. As it shows no significant outlier behavior, the mapping function is most likely correct. In conclusion, the attacker successfully reverse-engineered the physical memory layout of the target system using NO-DRAMA. They use this physical memory layout in the next step to find a reproducible memory access pattern.

4.2.2 Finding Reproducible Memory Access Patterns

After reverse-engineering the physical memory layout, the attacker searches for reproducible memory access patterns. For the target system at hand, experiments show that a multi-sided access pattern as generated by TRRESPASS is not sufficient to induce bit flips. However, the BLACKSMITH fuzzing suite is able to find reproducible non-uniform access patterns. Figure 4.3 shows the reproducible access patterns found by an eight-hour run

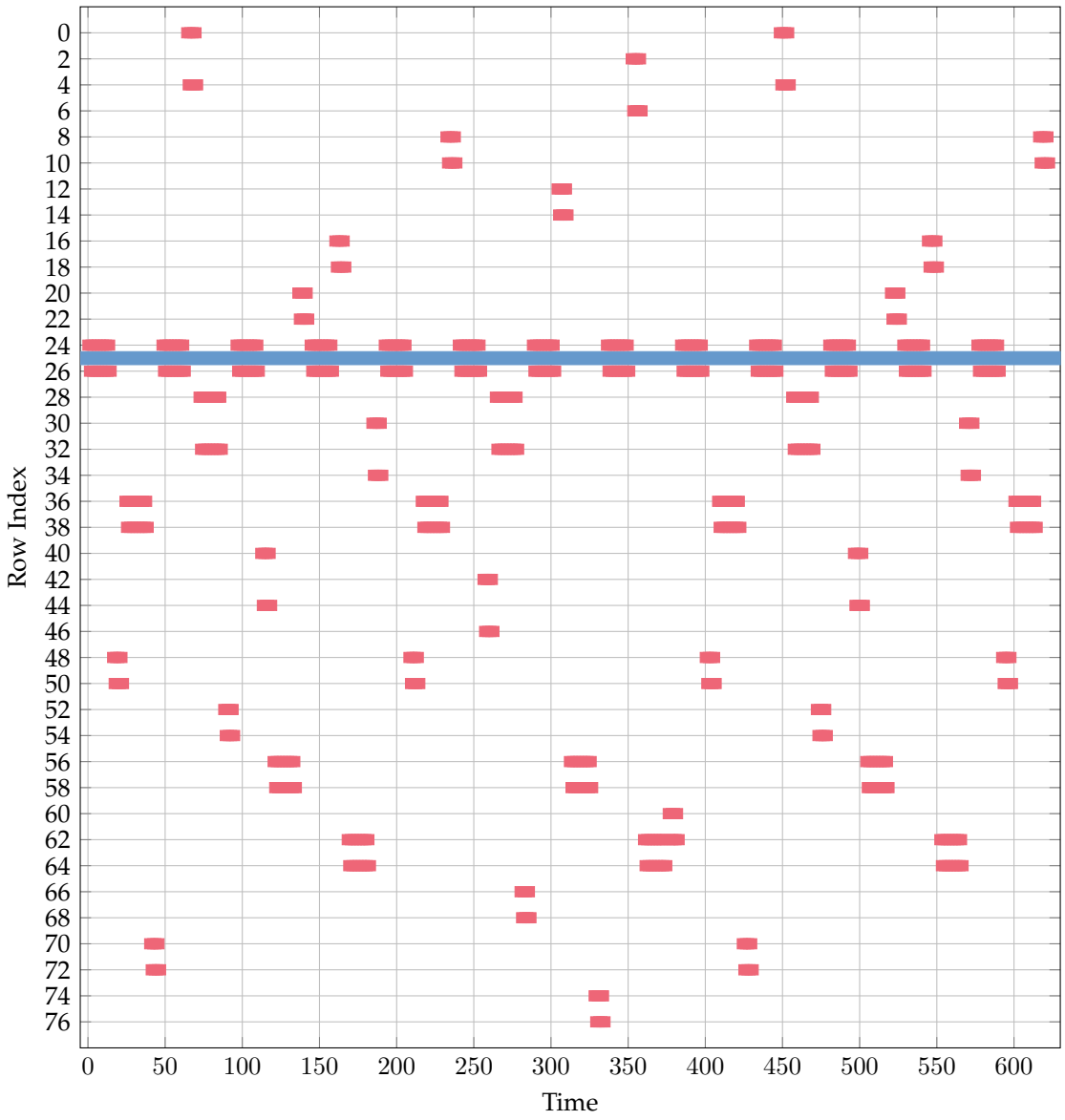


Figure 4.3: Reproducible BLACKSMITH access pattern found during an eight-hour fuzzing run. Each red rectangle marks an aggressor-row access, plotted by access sequence (x-axis) and logical row index (y-axis). Logical indices are translated to physical rows via the aggressor mapping. The horizontal blue band highlights the victim row.

Table 4.1: Results of an eight hour fuzzing run of BLACKSMITH. The table shows properties of the aggressor access patterns, their respective aggressor mappings, and the bit flips observed during the fuzzing run.

Pattern	Activations	Refresh Intervals	Mapping Identifier	Bank	Min Row	Max Row	#Flips
$\mathcal{A}$	304	4	1	21	662	821	1
$\mathcal{B}$	624	8	1	0	1486	1635	1
			2	2	400	478	1
$\mathcal{C}$	624	8	1	18	2061	2220	2
$\mathcal{D}$	624	8	1	11	2827	2986	1
			2	13	2944	3102	0
			3	17	756	914	0
$\mathcal{E}$	304	4	1	31	780	938	1
			2	1	1340	1489	0
$\mathcal{F}$	304	4	1	30	455	614	2
			2	2	1157	1315	0
$\mathcal{G}$	312	4	1	27	1177	1315	1
			2	29	2345	2504	0
			3	31	1289	1427	0
$\mathcal{H}$	312	4	1	23	684	842	1

of BLACKSMITH. The fuzzer differentiates between an *aggressor access pattern* and the *address mapping*. The aggressor access pattern is the sequence of accesses to logical aggressor indices that are used to induce bit flips, and the aggressor mapping is the mapping of a logical aggressor index to a DRAM address (bank, row, column).

Table 4.1 presents the hammering patterns found in an eight-hour fuzzing session. During this period, BLACKSMITH identified eight distinct access patterns reliably triggering bit flips. The reproducibility experiment in Table 4.2 indicate that these patterns vary in how consistently they induce bit flips during a *sweeping run* – a scenario where an aggressor mapping shifts across rows within the same bank. Such reproducibility serves as a strong proxy for how broadly an access pattern might be usable in a real-world attack scenario.

After the fuzzing run, BLACKSMITH exports the access patterns and aggressor mappings in a machine-readable format. SWAGE comes with an importer that can read these patterns and mappings and employs a just-in-time compiler to convert them into an assembly function that can be executed on the target system. Reproducing the pattern-mapping pairs with SWAGE yields the same bit flips observed during the fuzzing run with simi-

Table 4.2: Reproducibility results of BLACKSMITH using the best aggressor mapping per access pattern. Each candidate was tested over 10 rounds, each allowing up to 1000 pattern repetitions or terminating early upon the first encountered bit flip.

Candidate	Bitflips		Retries		Time (s)		
	Avg.	Max	Avg.	Max	Avg.	Max	Total
$\mathcal{A}_1$	1.3	2	1.4	4	1.22	3.54	12.20
$\mathcal{B}_1$	0.0	0	1000.0	1000	901.10	901.45	9011.03
$\mathcal{C}_1$	0.0	0	1000.0	1000	883.49	885.65	8834.95
$\mathcal{D}_1$	2.1	6	8.9	26	7.93	23.13	79.26
$\mathcal{E}_1$	1.1	2	271.6	780	236.13	678.13	2361.32
$\mathcal{F}_1$	0.2	1	863.9	1000	774.79	896.88	7747.86
$\mathcal{G}_1$	1.0	1	23.7	51	20.23	43.53	202.30
$\mathcal{H}_1$	1.1	2	10.8	34	9.52	29.97	95.19

lar reproducibility scores, confirming the effectiveness of the access pattern. Among all tested patterns, access pattern $\mathcal{G}$ with aggressor mapping 1 demonstrated the highest reproducibility. We call this pattern-mapping pair $\mathcal{G}_1$ and will use it for the subsequent stages of the attack.

In this stage of the attack, BLACKSMITH can utilize huge pages during the fuzzing phase. However, as discussed above, this approach is impractical for the online phase, where the attacker has to deallocate their target page and inject it into the victim's address space. After identifying reproducible access patterns and aggressor mappings with BLACKSMITH, the attacker therefore adjusts his memory allocation strategy to use *block splitting*. Specifically, they divide the huge page into 4 MiB chunks and allocate corresponding 4 MiB memory regions – using SPOILER or similar techniques described in Section 3.2 – on the target system. This allows the attacker to map all aggressors within each chunk to their respective 4 MiB memory blocks, facilitating effective exploitation during the online phase. However, as the number of allocatable 4 MiB blocks is limited, the aggressor mapping fuzzer is adjusted to only generate mappings covering a portion of the available address space, e.g., 40 MiB of the 1 GiB memory available. For $\mathcal{G}_1$, this results in a total of ten 4 MiB chunks, each containing a subset of the aggressors. The reproducibility experiment confirms that $\mathcal{G}_1$ remains effective even when applied to these smaller memory blocks.

In a last offline reproducibility experiment, the attacker deallocates the target page before initiating the hammering process, and then allocates it in a *dummy process*. This dummy process reserves a large stack-based array, fills it with a predetermined value, and then reads the array to verify whether the contents of the target page match the expected value before rewriting the array. This approach mimics the online phase of the attack, where a

victim process is anticipated to interact with the target page. The results of this experiment demonstrate that $\mathcal{G}_1$ maintains the reproducibility outlined in Table 4.2, underscoring its resilience to page walks, page table flushes, and other memory management operations carried out by the operating system.
Concluding the offline phase, the attacker has successfully identified a reproducible access pattern and aggressor mapping that can be used to induce bit flips in the target page. In the next section, the SPHINCS+ reference implementation is analyzed to find a spatially and temporally suitable memory region for the online phase of the attack.

4.2.3 Fault Analysis of the SPHINCS+ Reference Implementation

In [Castelnovi et al., 2018], the *grafting tree* fault attack on SPHINCS, the predecessor of SPHINCS+, is introduced, exploiting faults in the Merkle tree computation. Later experimental studies successfully demonstrate this attack on the official reference code [Genêt et al., 2018] and adapt the attack to SPHINCS+ [Genêt, 2023]; however, they rely on clock glitching on embedded devices running a truncated SPHINCS+ implementation. Contrasting, a Rowhammer attack operates via carefully crafted DRAM access patterns to induce bit flips, representing a fundamentally different fault model and attack surface. For example, clock glitching attacks can be used to reproducibly cause instruction skips, while Rowhammer attacks can only flip bits in (uncached) memory, limiting the attack surface. On the other hand, Rowhammer attacks can be mounted on commodity hardware without specialized hardware support, making them more accessible for practical attacks. In this section, we analyze the SPHINCS+ reference implementation to pinpoint memory regions most susceptible to a Rowhammer-based Merkle tree fault injection.
The SPHINCS+ reference implementation is written in C and provides a NIST-specified API for key generation, signing, and verification. We combine static and dynamic analysis methods to find suitable memory regions for the Rowhammer attack. As discussed in Section 3.4.2, we distinguish between two dimensions of suitability: *spatial* and *temporal* suitability.
To find spatially suitable memory regions, we map attack vectors introduced in prior works to the SPHINCS+ reference implementation. Subsequently, the DevMem hammerer in Swage is used to experimentally verify the spatial suitability. The DevMem hammerer allows us to simulate the Rowhammer attack by flipping bits in a given physical memory page and flushing the target page from the cache, effectively eliminating the constraints imposed by temporal suitability.
Finding temporally suitable memory regions is more challenging. For example, CPU-specific caching behavior leads to changes in the interaction between CPU and DRAM, which can have a significant impact on the success rate of a Rowhammer attack. In the

worst case, a CPU with large caches might always keep the target in cache, which would render the Rowhammer attack ineffective. Therefore, to verify temporal suitability, we combine static analysis with the MEMUTILS library shipped with SWAGE to instrument the SPHINCS$^+$ reference implementation and collect memory access traces for spatially suitable memory regions. The traces show the memory access patterns of the SPHINCS$^+$ reference implementation as well as potential cache-related issues by measuring the latency of memory accesses. By examining these traces – specifically the timing of individual accesses – we can observe the SPHINCS$^+$ implementation's memory-access patterns and identify any cache-related anomalies that might affect Rowhammer exploitability.
In the next subsection, we take a closer look at the Merkle tree computation in the SPHINCS$^+$ reference implementation and analyze how it can be exploited in a Rowhammer attack. We focus on the `treehashx1` function, which computes the root node of a Merkle tree, and show that the `stack` variable can be used as a spatially and temporally suitable target.

Fault Attack Against the Merkle Tree Computation in SPHINCS$^+$

In the hypertree phase of the SPHINCS$^+$ signature generation, **`treehash`** (Algorithm 15) computes the root node of the Merkle tree identified by an address **ADRS** and a secret **SK**.seed. In the C reference implementation, the **`treehash`** algorithm is implemented as a function `treehashx1` in file `utilsx1.c`. It takes as input the secret and public seed, the start index, the target height, and the **ADRS** of the tree. The function computes the root node of the Merkle tree bottom up, starting at the left-most WOTS$^+$ node and working its way up to the root. While walking up the tree, it uses a stack to store the nodes that are currently being processed.
To find a candidate with spatial suitability, we analyze the implementation of **`treehash`** in the SPHINCS$^+$ reference implementation. During the tree hash walk of a subtree of height z, the stack never holds more than one node at each level $0, 1, \ldots, z-1$. Concretely, each time a node of height i is generated or combined, it is either immediately merged with another node at height i (if one exists on the stack), or it is pushed to the stack. Since there are only z possible heights below the root, the stack can grow to at most z nodes. In the reference C implementation, this is realized by allocating `uint8_t stack[z * n]` where n is the hash-output length, i.e., the security parameter, in bytes.
While signing and hence calculating the root node of an XMSS tree, **`treehash`** is called with $s = 0$ and $z = h'$, where h' is the height of the tree. For the SHAKE-128f parameter set with $n = 16\,\mathrm{B}$ and $h' = 3$, the `stack` can hold up to $h' \cdot n = 48\,\mathrm{B}$ of data. However, for the SHAKE-256s parameter set, with $n = 32\,\mathrm{B}$ and $h' = 8$, `stack` has a capacity 256 B. This makes `stack` a spatially suitable target for a Rowhammer attack, as it covers

a reasonable portion of an 8 KiB target row, especially so for the SHAKE-256s parameter sets. The higher security levels of SPHINCS+ increase the size of the stack and thus the attack surface for a Rowhammer attack.

For the temporal suitability of the `stack` variable, we assume the signing party currently executes **`treehash`** to compute the public key pk^X of an XMSS tree at penultimate layer $d-1$, and just started iteration $i > 0$ of the outer loop, and `stack` is not empty. The algorithm calls the **`wots_pkGen`** function to compute the WOTS public key for the current index $s+i$. To compute the WOTS+ public key from the secret seed, the chaining hash function F is called repeatedly. For example, with SHAKE-256s parameter set, a WOTS+ instance consists of $\ell = \ell_1 + \ell_2$ hash chains with $\ell_1 = \left\lceil \frac{8n}{\lg_w} \right\rceil = 64$ message chains and $\ell_2 = \left\lfloor \frac{\log_2(\ell_1 \cdot (w-1))}{\lg_w} \right\rfloor = 3$ checksum chains. For comparison, using the SHAKE-128f parameter set, the WOTS+ instance consists of $\ell_1 = 32$ message chains and $\ell_2 = 3$ checksum chains. Each chain consists of $w = 16$ steps, where each step requires a hash computation. After all chains are completed, the WOTS+ public key is computed by concatenating the results of the message and checksum chains and applying the hash function T_ℓ to the concatenated result. Therefore, to compute the WOTS+ public key from the secret seed with the SHAKE-256s parameter set, a total of

$$\ell + \ell \cdot (w-1) + 1 = 67 + 67 \cdot 15 + 1 = 1071$$

hash function calls are required (resp. 561 hash function calls for SHAKE-128f). This gives an attacker a sufficiently large time frame to flip a bit in `stack`, which currently stores an intermediate `node` of the XMSS tree. Practical experiments on the target system with the SHAKE-256s parameter set show that the contents of `stack` are not held in cache. Again, the higher security levels of SPHINCS+ increase the number of hash function calls and thus increase the attack surface for a Rowhammer attack.

In the `treehashx1` function, `stack` is allocated as a $z \cdot n$ B array on the stack. The attacker wants to place `stack` in a target page and therefore performs a *page injection attack*. To mount this attack, the attacker first analyzes the memory access patterns of the `treehashx1` function to find the spatial and temporal characteristics of `stack` in the target program by instrumenting it using the MEMUTILS toolchain. Even though the instrumentation adds some overhead due to increased I/O operations, it allows the attacker to collect detailed memory access traces and is a first step towards finding a suitable memory region for the Rowhammer attack.

Figure 4.4 shows the access pattern for the `stack` array in the `treehashx1` function. The figure shows all write and read operations during each layer in a XMSS tree signing process. The `stack` array is accessed in a highly predictive manner. While lower offsets in the

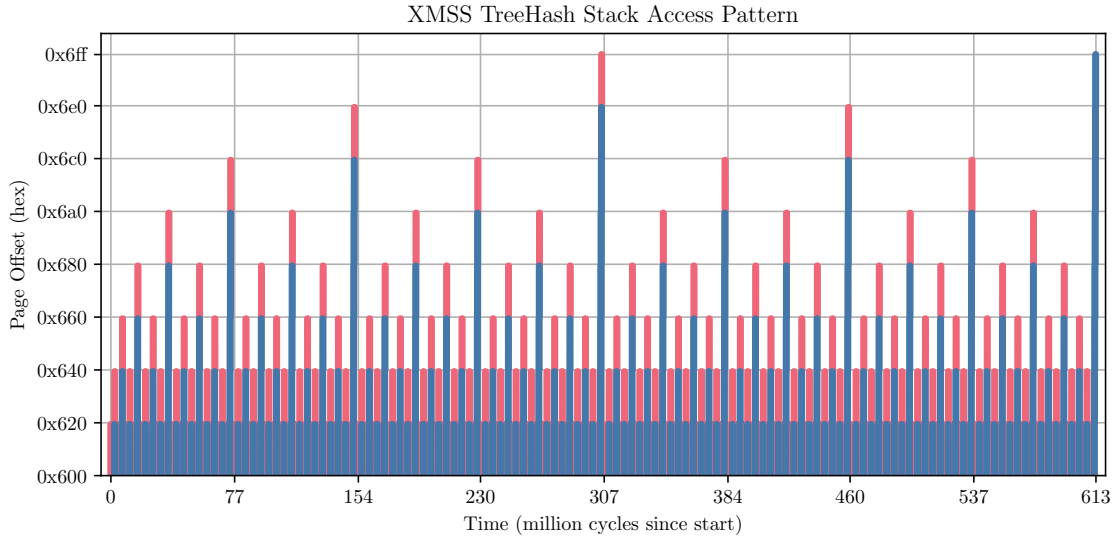


Figure 4.4: Access patterns for `stack` in the `treehashx1` function in the SPHINCS⁺ reference implementation. The figure shows all accesses to the `stack` array during generation of a XMSS public key. Write accesses are shown in red, read accesses in blue.

array are written to and read from frequently, the higher offsets are only written to and read from once per layer. This corresponds to the implementation of the stack in Algorithm 15, where it is used to store intermediate nodes of the Merkle tree. Additionally, the distance between write and read accesses to higher offsets in the `stack` array is relatively large. This gives the attacker a significant time window to flip a bit between offsets `0x6e0` and `0x700`, which corresponds to the last 32 B of the `stack` array.
Concluding, `stack` appears to be a spatially and temporally suitable target for a Rowhammer attack. In the next section, we discuss how the attacker ensures that the `stack` variable is allocated on a flippable victim page and how to inject the page into the victim's address space.

Page Injection Attack Against The SPHINCS⁺ Reference Implementation

After identifying the `stack` array as a suitable target for a Rowhammer attack, the attacker needs to ensure that it is allocated on a victim page with desirable flippiness characteristics. To achieve this, they need to perform a *page injection attack*, where the operating system's allocator state is manipulated to ensure that the `stack` array is placed on an attacker-controlled page.
As described in Section 3.4.1, the OS maintains a CPU-local list of recently freed pages,

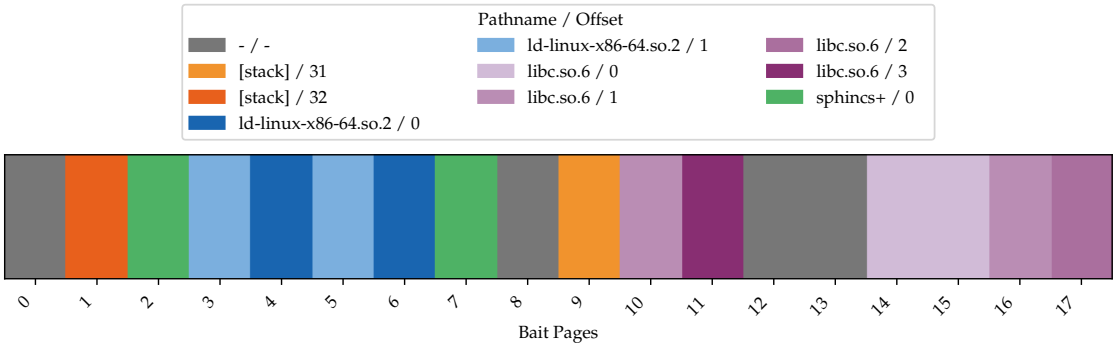


Figure 4.5: Heatmap of bait allocations.

which is used to serve page allocation requests. This mechanism can be exploited for a page injection attack by examining the allocation behavior of a target program and pushing a specific number of *bait pages* into the CPU-local free list before releasing the target page. When the victim process allocates a page, they end up with the target page mapped into their memory space after exhausting the bait pages. SWAGE provides an experiment to analyze the allocation behavior of a target program and to find the number of bait pages required to reproducibly inject a target page. It implements the following algorithm:

1. Allocate a 4 KiB *target page* and many *bait pages*.
2. Determine the physical address of the target page.
3. Release k bait pages followed by the target page.
4. Start the victim process.
5. Find the target page in the page map of the victim process, and return the region and page offset the target page was placed in.

Figure 4.5 shows a heatmap of the page injection experiment for the SPHINCS+ reference implementation. The heatmap shows the number of bait pages required to inject a target page at a specific memory region in the SPHINCS+ reference implementation. Note that some regions appear twice in the heatmap. This is due to access restrictions imposed by the operating system, mapping system libraries twice with differing access rights. For example, deallocating nine bait pages before the target page allows the attacker to inject the target page as the second-to-last page in the stack (region offset 31) of the SPHINCS+ reference implementation. Dynamic analysis using MEMUTILS shows that the `stack` array is allocated with region offset 32 on the stack, which corresponds to one bait page. Repeating the experiment shows consistent results, with the target page being injected at

the expected location, concluding that the attacker can reliably place the `stack` array in an attacker-controlled page.
Thus, we have shown that an attacker can use the tools provided by SWAGE to find a suitable memory region for a Rowhammer attack against the SPHINCS$^+$ reference implementation. Next up, the online phase of the attack is evaluated, where the attacker collects signatures from the SPHINCS$^+$ reference implementation and subsequently analyzes them to perform the grafting tree attack.

4.3 Online Phase

With the information gathered in the offline phase, the attacker now proceeds with the online phase of the attack. In the online phase, the attacker's goal is to inject a target page into the victim's address space and hammer the target page to induce bit flips during the victim's execution. The online phase consists of the following steps:

1. Allocate contiguous memory blocks for the attack (Section 4.3.1).
2. Profile the allocated memory for reproducible bit flips in a target page at suitable page offsets (Section 4.3.2).
3. Perform a page injection attack to inject the page into the victim's address space (Section 4.3.3).
4. Collect output from the victim process for post-attack analysis (Section 4.3.4).

After the online phase, the attacker analyzes the collected signatures to determine the number of reused one-time signatures and conduct a grafting tree attack for a universal forgery attack.

4.3.1 Allocating Contiguous Memory Blocks

In the offline phase, the attacker has full control over a replicated target system. In the online phase, however, the attacker is co-located with the victim and is only assumed to have user-level access to the target system. Therefore, the attacker cannot configure huge pages or otherwise instruct the kernel to allocate contiguous memory blocks. To overcome this limitation, the ALLOCATOR module in SWAGE is used to allocate contiguous memory blocks on the target system. The module implements the SPOILER allocator discussed in Section 3.2.4, which allows an attacker to allocate contiguous memory blocks of 4 MiB in size. Experiments on the target system show that the SPOILER allocator deterministically allocates contiguous memory blocks, given the optimizations discussed in Section 3.2.4

and the system having higher-order memory blocks available. Most notably, it significantly outperforms the `BuddyInfo` allocator, which shows probabilistic characteristics due to the noise introduced by co-located processes.

4.3.2 Profiling Memory for Reproducible Bit Flips

After allocating contiguous memory blocks, the attacker can now profile the memory for reproducible bit flips. The profiling is done by hammering the recently allocated memory blocks with SWAGE's profiler in the HAMMERER module. The attacker uses the profiler to identify memory regions that exhibit reproducible bit flips. In Section 4.2, the attacker has learned that a reproducible bit flip between page offsets `0x6e0` and `0x700` is optimal to perform a universal forgery attack, as this range covers the topmost 32 B of the `stack` array in the SPHINCS+ reference implementation. Assuming that reproducibly flippable bits are distributed uniformly across 4 KiB pages, the attacker can expect to find a bit flip in this region with a probability of

$$\mathbb{P}(\text{Profiling}) = \frac{\texttt{0x700} - \texttt{0x6e0}}{\texttt{0x1000}} = \frac{32}{4096} = \frac{1}{128}.$$

The attacker can therefore expect to find a bit flip in this region after profiling 128 pages. Recalling the memory layout of C programs introduced in Figure 2.3, we note that the environment variables are stored before the stack. This behavior can be exploited to speed up the profiling by defining a number of environment variables before launching the target program, shifting the stack down by the size of the injected environment variables. While this approach is tangential to the attack and a little stretch to the threat model, it allows the attacker to significantly increase the performance of the profiling step, as the attacker can manipulate the program's memory layout instead of having to find a bit flip at a suitable page offset.

4.3.3 Page Injection Attack

Once a reproducible bit flip at a suitable page offset was found, the page injection attack, the penultimate step of the online phase, can be performed. The page injection attack uses the allocation information gathered during the offline phase to inject the target page into a delicately crafted location in the victim's memory space. For the attack on the `stack` array in the SPHINCS+ reference implementation, the attacker aims to inject the target page into the stack of the victim process at offset 31. For this, according to the threat model introduced in Section 4.1, it is assumed that the attacker is able to start the signing server process on the target machine. SWAGE provides a page injection module that allows the attacker to deterministically inject a target page into the victim's address space. Practical

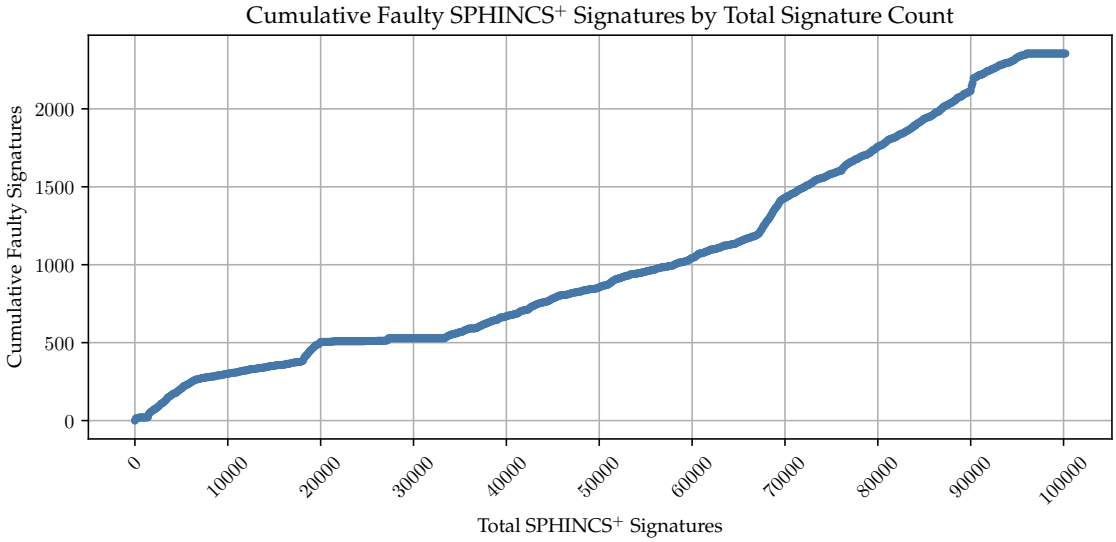


Figure 4.6: Number of non-verifiable signatures collected. The X-axis shows the number of signatures collected, while the Y-axis shows the number of non-verifiable signatures.

experiments show that the page injection attack is successful in 99 out of 100 cases. In case the page injection fails (most likely due to noise caused by allocations from different processes on the same CPU), the attacker restarts the online phase and tries again. Once the page injection is successful, the attacker proceeds to collect signatures from the signing server.

4.3.4 Collecting Signatures

The final step of the online phase is to collect signatures from the signing server. The signatures are collected by reading the victim's standard output and stored in a file for later analysis. The signing server is configured to run the SPHINCS$^+$ signature scheme with randomization enabled and parameters `SHAKE-256s` at the highest security level 5. The server accepts signing requests via standard input and outputs signatures to standard output, allowing for synchronization between the attacker process and the signing server. On the target machine, signing a message using the reference implementation takes about 2.6 seconds, which is sufficient to collect numerous signatures in a reasonable time frame. Figure 4.6 shows the number of non-verifiable signatures collected during the 72-hour online phase. The X-axis shows the number of signatures collected, while the Y-axis shows the number of non-verifiable signatures. In this experiment, 2355 non-verifiable signatures were observed – around 2.35% of the total 100 297 signatures (5.6 GiB) collected from the

signing server. While the number of non-verifiable signatures is a first indicator for the success of the attack, not all successful faults lead to a non-verifiable signature, as a fault may also affect the computation of the authentication path, leading to a verifiable signature that still compromises a WOTS+ key. However, the number of non-verifiable signatures is a still a useful indicator for the number of bit flips induced in the target memory region. In the grafting phase of the attack, the attacker analyzes the signatures to find reused one-time WOTS+ keys, which is covered in the next section.

4.4 Grafting Phase

After the online phase, the attacker analyzes the collected signatures to perform the grafting tree attack. In the attack presented in this thesis, the target layer is fixed to the penultimate layer $l^* = 7$ for the SHAKE-256s parameter set, as this layer is the most likely to have key collisions. As discussed in Section 2.4, the attack consists of several steps, which we describe in the following.

4.4.1 Identifying WOTS+ Key Collisions

In this step, WOTS+ signatures are extracted from the collected full signatures and collisions are identified, applying the method described in Section 2.4.2. Figure 4.7 shows the maximum number of WOTS+ key collisions observed during the online phase, as a function of the number of signatures collected. Similarly to the faulty signatures, we observe an increase in the number of key collisions as the number of signatures increases. The maximum number of key collisions is a good indicator for the complexity of the grafting tree attack, as it serves as a proxy for the number of exposed WOTS+ secrets.

4.4.2 Tree Grafting

In the tree grafting step (introduced in Section 2.4), the attacker randomly samples XMSS trees until they find one that is signable using a compromised WOTS+ instance. Equation (2.1) estimates the complexity of the grafting tree attack, and Equation (2.2) estimates the probability that a randomly sampled XMSS tree is signable using a compromised WOTS+ instance with M uniformly distributed collisions.
Practical experiments with the signatures collected during the online phase show that the attack complexity matches the theoretical estimates in Table 2.3, showing that the attacker can successfully graft a tree after collecting a few colliding WOTS+ signatures. For a WOTS+ instance with $M = 4$ collisions, the attacker can expect to find a signable XMSS tree after sampling about $2^{35.8}$ trees with the collisions encountered during the practi-

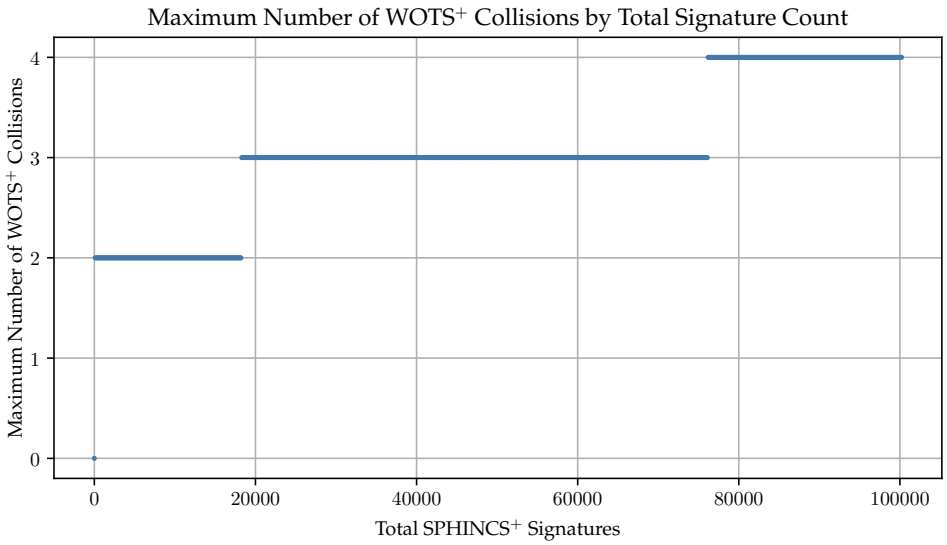


Figure 4.7: Maximum number of WOTS$^+$ key collisions observed as a function of the number of signatures collected. The X-axis shows the number of signatures collected, while the Y-axis shows the number of key collisions.

cal attack. This closely matches the theoretical estimate of $2^{36.09}$ in Table 2.3. On the machine used for the grafting phase, sampling a single XMSS tree takes about 62.5 ms, or 153 125 000 cycles at 2.45 GHz. With 128 threads available, the attacker can therefore expect to successfully graft a tree after about 416 days of continuous sampling.

While this is a time-consuming task, we note that the attack can be significantly accelerated through the use of hardware acceleration. For example, [Saarinen, 2024] introduces SLOTH, an FPGA-based hardware accelerator for SPHINCS$^+$, reducing the time to sample a single XMSS tree to 274 943 cycles. The authors implement the proposed accelerator on a Xilinx VCU118 FPGA board running at 250 MHz, where sampling a single XMSS tree thus only takes about 1.1 ms. [Wang et al., 2023] introduce GPU-based hardware acceleration for XMSS, achieving up to 0.426 ms per tree on an NVIDIA RTX 3090 GPU by employing multi-key parallelism, i.e., sampling multiple trees in parallel. The follow-up work [Wang et al., 2025] further improves the performance of the GPU implementation and introduces a GPU-based hardware accelerator for SPHINCS$^+$. However, the latter does not provide performance measurements for the parameter set used in this evaluation. Using the relatively modest hardware setup used in [Wang et al., 2023], the grafting attack could be accelerated to about 360 days. However, when employing state-of-the-art hardware accelerators, such as a cluster of NVIDIA RTX 5090 or NVIDIA H200 GPUs, the time to graft a tree could be reduced to a few days or even hours, depending on the number of WOTS$^+$ key collisions and cores available. We leave the exact performance evaluation of

the grafting phase using hardware accelerators and the analysis of Rowhammer attacks against a broader range of parameter sets for future work.
Concluding, the feasibility of a Rowhammer-based grafting tree attack against SPHINCS+, as demonstrated in this evaluation, highlights the severe impact of practical fault attacks on the hash-based signature scheme. By exploiting even a few key collisions induced through targeted bit flips, an attacker can achieve a universal forgery with relatively modest resources, especially considering high-performance hardware accelerators like GPUs. This underscores the need for robust fault attack countermeasures in both hardware and software implementations of cryptographic schemes. Without such protections, even mathematically secure algorithms remain vulnerable in real-world deployments, making the implementation of fault detection and mitigation strategies critical for maintaining the integrity and trustworthiness of digital signatures.

5 Conclusions

This thesis presents SWAGE, a comprehensive framework for performing Rowhammer attacks on real-world systems. SWAGE provides a user-friendly programming interface for performing Rowhammer attacks, allowing users to focus on the attack logic rather than the underlying hardware and operating system details. The framework is designed to be flexible and extensible, allowing users to easily adapt it to their specific needs. We release SWAGE as an open-source project, making it available for researchers and practitioners to use and extend. To our knowledge, SWAGE is the first fully open-source modular end-to-end framework for Rowhammer attacks, providing a complete and extensible solution for performing attacks on real-world systems.
SWAGE is evaluated by performing an end-to-end Rowhammer attack against the post-quantum signature scheme SPHINCS$^+$. The attack is demonstrated to be feasible and highly effective in practice, allowing an attacker to conduct a universal forgery attack against SPHINCS$^+$ by flipping bits in the signature generation process. This is, to the best of our knowledge, the first practical demonstration of a Rowhammer attack against SPHINCS$^+$ on a real-world system.

5.1 Related Work

The *grafting tree attack* was first described by [Castelnovi et al., 2018] who focussed on a theoretical analysis of the attack and its computational feasibility. A practical implementation of the grafting tree attack against SPHINCS was then presented by [Genêt et al., 2018]. Both of the aforementioned works discuss the susceptibility of SPHINCS$^+$ to the same fault attack, but do not show it in practice. This is made up for in [Genêt, 2023], where a theoretical analysis of the grafting tree attack against SPHINCS$^+$ and potential countermeasures are discussed and a practical implementation of the attack is presented, showing the feasibility of the attack against SPHINCS$^+$. However, the practical works do not consider Rowhammer as a potential fault injection method, but focus on an embedded target system in a hardware-supported *clock glitching* attack.
Rowhammer attacks have also been demonstrated against other post-quantum cryptographic schemes. In [Amer et al., 2024], the authors present Rowhammer-based end-to-end key recovery attacks against the post-quantum key encapsulation mechanisms BIKE and CRYSTALS-Kyber (ML-KEM) as well as the lattice-based post-quantum signature

scheme CRYSTALS-Dilithium (ML-DSA). The authors demonstrate that Rowhammer attacks can be used to recover secret keys from these schemes, highlighting the need for robust countermeasures against hardware-based attacks in post-quantum cryptography. In [Haidar et al., 2025], the authors present a Rowhammer-based key recovery attack against post-quantum signature scheme Falcon requiring only a single bit flip.

A comprehensive Rowhammer simulation framework is presented in [Tatar et al., 2018]. The authors introduce HAMMERTIME, a framework that simulates Rowhammer attacks and provides estimates on the number of expected hammering experiments required to flip a bit in a given page offsets range. However, while HAMMERTIME comes with several useful of tools for profiling and simulating Rowhammer attacks, it does not provide an end-to-end solution for performing Rowhammer attacks on real-world systems.

Regarding fault analysis tools, [Liang et al., 2025] introduce ACHILLES, a formal framework for fault analysis of signature schemes. The authors present a systematic approach to fault analysis by separating the fault model from the cryptographic algorithm. They do so by introducing a generalized signature scheme G-SIGN that categorizes signature scheme parameters into public parameters (*pp*) and secret parameters (*sp*). G-SIGN then uses signing oracles `OSign` and `OFSign` to generate valid and faulty signatures, respectively. The faults are injected according to a fault model applied to public or secret parameters. The faulty signatures are then post-processed using Differential Fault Analysis (DFA) and Signature Correction Analysis (SCA) to recover the secret key of the signature scheme. The authors demonstrate the feasibility of their framework by applying it to six different signature schemes, including post-quantum signature scheme ML-DSA. However, while ACHILLES appears to be a powerful tool for systematic fault analysis, its applicability to fault attacks against SPHINCS$^+$ is questionable, as it only considers key recovery attacks and not universal forgery attacks. Another notable work is RAINBOW [Looss, 2022], which aims to automate fault analysis by emulating the target program using the UNICORN CPU emulator framework [Unicorn, 2015]. The authors present a framework that allows users to specify fault injection parameters and automatically inject faults into the target program.

5.2 Discussion and Open Problems

The results of this thesis introduce SWAGE and demonstrate the feasibility of Rowhammer attacks against post-quantum signature schemes, using the example of SPHINCS$^+$. While the attack is demonstrated to be effective in practice, there are still some open problems and challenges that need to be addressed in future work. This thesis only covers the reference implementation of SPHINCS$^+$ without hardware acceleration such as `AVX2`. Em-

ploying hardware acceleration and parallelization techniques can significantly increase the performance of the signature generation process, which in turn might decrease the effectiveness of the Rowhammer attack. Future work could investigate the impact of hardware acceleration on the feasibility of Rowhammer attacks against SPHINCS$^+$ and other post-quantum signature schemes. Additionally, only the SPHINCS$^+$ reference implementation is considered in this thesis. Recently, OpenSSL v3.5 was released [OpenSSL, 2025], which adds support for SLH-DSA. Future work might focus on evaluating the feasibility of Rowhammer attacks against the SLH-DSA implementation in OpenSSL and against a broader range of different parameter sets.

The post-processing phase of the attack against SPHINCS$^+$ does currently not implement FPGA or GPU-based hardware acceleration techniques, which could significantly speed up the grafting tree attack. Thus, it would be interesting to implement hardware acceleration techniques for the grafting tree attack, such as the ones presented in [Wang et al., 2023, Saarinen, 2024, Wang et al., 2025].

Furthermore, the presented attack currently assumes a CPU co-located attacker for the page injection. A recent work [Bölcskei et al., 2025] has shown that page injection attacks can be also performed in a cross-CPU manner, allowing an attacker to inject faults into a victim process running on a different CPU core. This opens up new threat models for Rowhammer attacks. An open question would be to reproduce the page injection attack in a cross-CPU setting, potentially extending the attack to a wider range of systems.

Currently, binary analysis and instrumentation of the target process are performed manually. This workflow could be automated to enhance flexibility and usability. However, while RAINBOW [Looss, 2022] already supports memory access interception for fault simulation, this functionality incurs significant overhead due to its implementation in a Python wrapper around the UNICORN emulator. Integrating with RAINBOW may offer a promising path toward a high-performance framework for instrumentation and fault analysis.

References

[Adiletta et al., 2024] Adiletta, A. J., Tol, M. C., Doröz, Y., and Sunar, B. (2024). Mayhem: Targeted Corruption of Register and Stack Variables. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. ACM. https://dl.acm.org/doi/10.1145/3634737.3637638.

[Amer et al., 2024] Amer, S., Wang, Y., Kippen, H., Dang, T., Genkin, D., Kwong, A., Nelson, A., and Yerukhimovich, A. (2024). PQ-Hammer: End-to-End Key Recovery Attacks on Post-Quantum Cryptography Using Rowhammer. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3308–3323. IEEE Computer Society. https://www.computer.org/csdl/proceedings-article/sp/2025/223600a048/21B7QQRP39e.

[Arnold et al., 2024] Arnold, P., Berndt, S., Eisenbarth, T., and Orlt, M. (2024). Polynomial sharings on two secrets: Buy one, get one free. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):671–706. https://doi.org/10.46586/tches.v2024.i3.671-706.

[Aumasson et al., 2022] Aumasson, J.-P., Bernstein, D. J., Beullens, W., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.-L., Hülsing, A., Kampanakis, P., Kölbl, S., Lange, T., Lauridsen, M. M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., and Schwabe, P. (2022). SPHINCS+ Submission to the NIST post-quantum project, v.3.1. https://sphincs.org/data/sphincs+-r3.1-specification.pdf.

[Baksi et al., 2022] Baksi, A., Bhasin, S., Breier, J., Jap, D., and Saha, D. (2022). A Survey on Fault Attacks on Symmetric Key Cryptosystems. *ACM Comput. Surv.*, 55(4):86:1–86:34. https://dl.acm.org/doi/10.1145/3530054.

[Berndt et al., 2023] Berndt, S., Eisenbarth, T., Faust, S., Gourjon, M., Orlt, M., and Seker, O. (2023). Combined Fault and Leakage Resilience: Composability, Constructions and Compiler. In *Advances in Cryptology – CRYPTO 2023*, pages 377–409, Cham. Springer Nature Switzerland. https://link.springer.com/chapter/10.1007/978-3-031-38548-3_13.

[Bernstein et al., 2017] Bernstein, D. J., Dobrauig, Christoph, Eichlseder, M., Fluhrer, S., Gazdag, S.-L., Hülsing, A., Kampanakis, P., Kölbl, S., Lange, T., Lauridsen, M. M.,

Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., and Schwabe, P. (2017). SPHINCS+ Submission to the NIST post-quantum project. https://sphincs.org/data/sphincs+-specification.pdf.

[Bernstein et al., 2019] Bernstein, D. J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., and Schwabe, P. (2019). The SPHINCS+ Signature Framework. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2129–2146. ACM. https://dl.acm.org/doi/10.1145/3319535.3363229.

[Bölcskei et al., 2025] Bölcskei, M., Jattke, P., Wikner, J., and Razavi, K. (2025). Rubicon: Precise Microarchitectural Attacks with Page-Granular Massaging. In *EuroS&P*. https://comsec-files.ethz.ch/papers/rubicon_eurosp25.pdf.

[Castelnovi et al., 2018] Castelnovi, L., Martinelli, A., and Prest, T. (2018). Grafting Trees: A Fault Attack Against the SPHINCS Framework. In *Post-Quantum Cryptography*, volume 10786, pages 165–184. Springer International Publishing, Cham. https://link.springer.com/10.1007/978-3-319-79063-3_8.

[Cojocar et al., 2019] Cojocar, L., Razavi, K., Giuffrida, C., and Bos, H. (2019). Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 55–71. https://ieeexplore.ieee.org/document/8835222.

[Easttom, 2022] Easttom, C. (2022). Cryptographic Hashes. In *Modern Cryptography: Applied Mathematics for Encryption and Information Security*, pages 213–231. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-031-12304-7_9.

[Frigo et al., 2020] Frigo, P., Vannacc, E., Hassan, H., Der Veen, V. V., Mutlu, O., Giuffrida, C., Bos, H., and Razavi, K. (2020). TRRespass: Exploiting the Many Sides of Target Row Refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762, San Francisco, CA, USA. IEEE. https://ieeexplore.ieee.org/document/9152631/.

[Genêt, 2023] Genêt, A. (2023). On Protecting SPHINCS+ Against Fault Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 80–114. https://tches.iacr.org/index.php/TCHES/article/view/10278.

[Genêt et al., 2018] Genêt, A., Kannwischer, M. J., Pelletier, H., and McLauchlan, A. (2018). Practical Fault Injection Attacks on SPHINCS. *IACR Cryptol. ePrint Arch.*, page 674. https://eprint.iacr.org/2018/674.

[Gruss et al., 2018] Gruss, D., Lipp, M., Schwarz, M., Genkin, D., Juffinger, J., O'Connell, S., Schoechl, W., and Yarom, Y. (2018). Another Flip in the Wall of Rowhammer Defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261. https://ieeexplore.ieee.org/document/8418607.

[Gruss et al., 2016] Gruss, D., Maurice, C., and Mangard, S. (2016). Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, volume 9721 of *Lecture Notes in Computer Science*, pages 300–321. Springer. https://doi.org/10.1007/978-3-319-40667-1_15.

[Haidar et al., 2025] Haidar, C. A., Payet, Q., and Tibouchi, M. (2025). Crowhammer: Full Key Recovery Attack on Falcon with a Single Rowhammer Bit Flip. https://eprint.iacr.org/2025/1042.

[Hocko, 2019] Hocko, M. (2019). [PATCH 1/2] mm, vmstat: Hide /proc/pagetypeinfo from normal users. https://web.archive.org/web/20220711162355/https://lore.kernel.org/all/20191025072610.18526-2-mhocko@kernel.org/T/.

[Islam et al., 2019] Islam, S., Moghimi, A., Bruhns, I., Krebbel, M., Gülmezoglu, B., Eisenbarth, T., and Sunar, B. (2019). SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 621–637. USENIX Association. https://www.usenix.org/conference/usenixsecurity19/presentation/islam.

[Jattke et al., 2022] Jattke, P., Van Der Veen, V., Frigo, P., Gunter, S., and Razavi, K. (2022). Blacksmith: Scalable rowhammering in the frequency domain. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 716–734. IEEE. https://ieeexplore.ieee.org/abstract/document/9833772/.

[Jattke et al., 2024] Jattke, P., Wipfli, M., Solt, F., Marazzi, M., Bölcskei, M., and Razavi, K. (2024). ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/jattke.

[JEDEC, 2012] JEDEC (2012). JESD79-4: DDR4 SDRAM Standard. https://www.jedec.org/sites/default/files/docs/JESD79-4.pdf.

[Kim et al., 2014] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. (2014). 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. https://ieeexplore.ieee.org/document/6853210.

[Kogler et al., 2022] Kogler, A., Juffinger, J., Qazi, S., Kim, Y., Lipp, M., Boichat, N., Shiu, E., Nissler, M., and Gruss, D. (2022). Half-Double: Hammering From the Next Row Over. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3807–3824. https://www.usenix.org/conference/usenixsecurity22/presentation/kogler-half-double.

[Kölbl et al., 2016] Kölbl, S., Lauridsen, M. M., Mendel, F., and Rechberger, C. (2016). Haraka v2 – Efficient Short-Input Hashing for Post-Quantum Applications. *IACR Transactions on Symmetric Cryptology*, pages 1–29. https://tosc.iacr.org/index.php/ToSC/article/view/563.

[Konoth et al., 2018] Konoth, R. K., Oliverio, M., Tatar, A., Andriesse, D., Bos, H., Giuffrida, C., and Razavi, K. (2018). ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 697–710. https://www.usenix.org/conference/osdi18/presentation/konoth.

[Kwong et al., 2020] Kwong, A., Genkin, D., Gruss, D., and Yarom, Y. (2020). RAMBleed: Reading Bits in Memory Without Accessing Them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711, San Francisco, CA, USA. IEEE. https://ieeexplore.ieee.org/document/9152687/.

[Lange, 2021] Lange, T. (2021). Hash-based Signatures. https://hyperelliptic.org/tanja/teaching/pqcrypto21/slides/hash-mm-1.pdf.

[Liang et al., 2025] Liang, J., Zhang, Z., Zhang, X., Shen, Q., Gao, Y., Yuan, X., Xue, H., Wu, P., and Wu, Z. (2025). Achilles: A Formal Framework of Leaking Secrets from Signature Schemes via Rowhammer. *34th USENIX security symposium, USENIX Security 25, Seattle, WA, USA, August 13-15, 2025*. https://www.usenix.org/conference/usenixsecurity25/presentation/liang-achilles.

[Lipp et al., 2020] Lipp, M., Schwarz, M., Raab, L., Lamster, L., Aga, M. T., Maurice, C., and Gruss, D. (2020). Nethammer: Inducing Rowhammer Faults through Network Requests. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 710–719. https://ieeexplore.ieee.org/document/9229701.

[Looss, 2022] Looss, A. (2022). Integrating Fault Injection In Development Workflows. https://www.ledger.com/blog/fault-injection-simulation.

[Luo et al., 2024] Luo, H., Olgun, A., Yağlıkçı, A. G., Tuğrul, Y. C., Rhyner, S., Cavlak, M. B., Lindegger, J., Sadrosadati, M., and Mutlu, O. (2024). RowPress: Amplifying Read Disturbance in Modern DRAM Chips. http://arxiv.org/abs/2306.17061.

[Merkle, 1990] Merkle, R. C. (1990). A Certified Digital Signature. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 218–238, New York, NY. Springer. https://link.springer.com/chapter/10.1007/0-387-34805-0_21.

[NIST, 2015a] NIST (2015a). Secure Hash Standard. Technical Report NIST FIPS 180-4, NIST, Washington, D.C. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[NIST, 2015b] NIST (2015b). SHA-3 standard : Permutation-Based Hash and Extendable-Output Functions. Technical Report NIST 202, National Institute of Standards and Technology (U.S.), Washington, D.C. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[NIST, 2024] NIST (2024). Stateless Hash-Based Digital Signature Standard. Technical Report NIST FIPS 205, National Institute of Standards and Technology (U.S.), Washington, D.C. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.pdf.

[OpenSSL, 2025] OpenSSL (2025). OpenSSL 3.5 Final Release - Live | OpenSSL Library. https://web.archive.org/web/20250522090203/https://openssl-library.org/post/2025-04-08-openssl-35-final-release/.

[Pessl et al., 2016] Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. (2016). DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 565–581. USENIX Association. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.

[Saarinen, 2024] Saarinen, M.-J. O. (2024). Accelerating SLH-DSA by Two Orders of Magnitude with a Single Hash Unit. In *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part I*, volume 14920 of *Lecture Notes in Computer Science*, pages 276–304. Springer. https://doi.org/10.1007/978-3-031-68376-3_9.

[Seaborn and Dullien, 2015] Seaborn, M. and Dullien, T. (2015). Project Zero: Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[Shor, 1994] Shor, P. (1994). Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Santa Fe, NM, USA. IEEE Comput. Soc. Press. http://ieeexplore.ieee.org/document/365700/.

[Shutemov, 2015] Shutemov, K. A. (2015). [RFC, PATCH] pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace. https://lore.kernel.org/all/1425935472-17949-1-git-send-email-kirill@shutemov.name/.

[Tatar et al., 2018] Tatar, A., Giuffrida, C., Bos, H., and Razavi, K. (2018). Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer. In *Research in Attacks, Intrusions, and Defenses*, pages 47–66, Cham. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-030-00470-5_3.

[Tobah et al., 2022] Tobah, Y., Kwong, A., Kang, I., Genkin, D., and Shin, K. G. (2022). SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 681–698, San Francisco, CA, USA. IEEE. https://ieeexplore.ieee.org/document/9833802/.

[Tol et al., 2023] Tol, M. C., Islam, S., Adiletta, A. J., Sunar, B., and Zhang, Z. (2023). Don't Knock! Rowhammer at the Backdoor of DNN Models. In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network, DSN 2023, Porto, Portugal, June 27-30, 2023*, pages 109–122. IEEE. https://ieeexplore.ieee.org/document/10202628/.

[Unicorn, 2015] Unicorn (2015). Unicorn – The Ultimate CPU Emulator. https://www.unicorn-engine.org/.

[van der Veen et al., 2018] van der Veen, V., Lindorfer, M., Fratantonio, Y., Pillai, H. P., Vigna, G., Kruegel, C., Bos, H., and Razavi, K. (2018). GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, volume 10885 of *Lecture Notes in Computer Science*, pages 92–113. Springer. https://doi.org/10.1007/978-3-319-93411-2_5.

[Wang et al., 2023] Wang, Z., Dong, X., Chen, H., and Kang, Y. (2023). Efficient GPU Implementations of Post-Quantum Signature XMSS. *IEEE Trans. Parallel Distributed Syst.*, 34(3):938–954. https://ieeexplore.ieee.org/document/10004747.

[Wang et al., 2025] Wang, Z., Dong, X., Chen, H., Kang, Y., and Wang, Q. (2025). CUSPX: Efficient GPU Implementations of Post-Quantum Signature SPHINCS+. *IEEE Trans. Computers*, 74(1):15–28. https://ieeexplore.ieee.org/document/10677363.

[Weissman et al., 2020] Weissman, Z., Tiemann, T., Moghimi, D., Custodio, E., Eisenbarth, T., and Sunar, B. (2020). JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):169–195. https://doi.org/10.13154/tches.v2020.i3.169-195.

[Wilke, 2023] Wilke, L. (2023). Institut für IT Sicherheit / research-projects / Rowhammer / no-drama · GitLab. https://git.uni-luebeck.de/its/research-projects/rowhammer/no-drama.

[Xiao et al., 2016] Xiao, Y., Zhang, X., Zhang, Y., and Teodorescu, R. (2016). One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 19–35. USENIX Association. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/xiao.