Software-Induced Fault Attacks on Post-Quantum Signature Schemes

Saad Islam

A Dissertation
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy
in
Electrical and Computer Engineering

April 2022

APPROVED:

Professor Berk Sunar
Advisor
Worcester Polytechnic Institute

Assistant Professor Fatemeh Ganji
Committee Member
Worcester Polytechnic Institute

Dr. Mehmet Sinan Inci
Committee Member
NVIDIA Corporation

Abstract

A digital signature is a digital equivalent of a handwritten signature or stamp which is used to validate the authenticity and integrity in digital communications like an email, a credit card transaction, or a digital document. Digital signatures are mathematical schemes whose security is based on conjectured hard problems like discrete log or RSA moduli factorization. Unfortunately, these public-key cryptosystems are not quantum secure and large-scale quantum computers will be able to solve the underlying hard problems. In 2021, IBM has released "Eagle", a 127-qubit quantum processor and has a roadmap of 1K-1M+ qubits beyond 2024. NIST has already realized the quantum threat and announced a competition for Post-Quantum Cryptography Standardization Process in 2016. It is currently in round 3 and expected to be finalized with the announcement of KEM and Digital Signature standards by 2022. Apart from algorithmic security, significant attention has been given to implementation attacks such as side-channel and fault attacks. To counter classical Differential Fault Attacks (DFA), which only work for deterministic schemes, the schemes are now offering randomized versions.

The goal of this dissertation is to investigate these randomized post-quantum signature schemes against fault attacks. The study has identified a number of vulnerabilities in several post-quantum schemes in the NIST

competition. We are able to recover the entire key of the LUOV (round 2 finalist) signature scheme in less than 4 hours of Rowhammer attack, followed by our novel bit-tracing algorithm and divide and conquer attack. We have named this hybrid attack QuantumHammer. More recently, we have proposed the "Signature Correction Attack" on the Dilithium signature scheme (round 3 finalist) and successfully reduced its security strength from 2^{128} to 2^{81}. Rowhammer attack does not require physical access and poses a significant threat to shared cloud servers. The identified vulnerabilities are however generic and can work as long as required faulty signatures are collected using any fault mechanism. The main idea of both bit-tracing and signature correction is to utilize a single faulty signature to mathematically trace back to the fault, revealing the secret key bit. We achieve this by trying to correct a faulty signature for all possible faults in the secret key using the verification algorithm as an oracle. This technique does not need any correct signature counterpart as needed in traditional DFA attacks.

In all of our Rowhammer experiments on post-quantum schemes, we have used SPOILER for finding the contiguous memory required for double-sided Rowhammer. SPOILER is a hardware bug we discovered in all *Intel* generations, starting from 1^{st} Gen (2008) of Intel core processors, stemming from the speculative load operations. SPOILER reveals critical physical address information to userspace processes which boosts Rowhammer and cache attacks.

Acknowledgments

This work is supported by U.S. Department of State, Bureau of Educational and Cultural Affair's Fulbright Program and by the National Science Foundation under grants CNS- 1814406 and CNS-2026913. It was a great experience of exchanging culture and education in the past five years of my PhD program.

I would like to extend my sincere gratitude to Professor Berk Sunar for his continuous support and guidance that made this dissertation possible. I really acknowledge his frequent visits to the lab and having healthy discussions which kept me on track and made me think that I can do it. He has always taken a lot of interest in my work and helped me accomplish it. I have learned from him to aim high and keep advancing toward my goals. I have always felt very comfortable sharing any kind of problems with him and his advice was always helpful. He is truly an inspiration and I will keep following his words in the future.

I am very grateful to my dissertation committee members Professor Berk Sunar, Professor Fatemeh Ganji, and Dr. Mehmet Sinan Inci for their invalu-

able time and guidance. Their constructive feedback and suggestions have substantially improved the quality of this work. Despite their busy schedules, they have always responded whenever I needed them.

I would like to thank Professor Thomas Eisenbarth (Worcester Polytechnic Institute and the University of Lübeck) and his team Moritz Krebbel and Ida Bruhns from the University of Lübeck, Germany, for working with me. I am grateful to my co-authors Daniel Moghimi, Berk Gulmezoglu, Andreas Zankl, M. Caner Tol, Koksal Mus, Ziming Zhang, Richa Singh, and Patrick Schaumont. I thank Muhammad Ali Siddiqi, who worked with me on my first research paper during my undergraduate, leading to the best paper award.

I feel lucky to be part of Vernam Lab and working with such amazing people around me. We have spent a lot of time together, had great discussions, and always tried to help each other as a family. In not any particular order, I would like to thank the faculty of Vernam Lab including Yarkin Doröz, Thomas Eisenbarth, Fatemeh (Saba) Ganji, William J. Martin, Koksal Mus, Patrick Schaumont, Berk Sunar, Shahin Tajik, my fellow students Andrew J. Adiletta, Ramazan Kaan Eren, Mohammad Hashemi, Pantea Kiaei, Zhenyuan (Charlotte) Liu, Dev Mehta, Tahoura Mosavirik, Dillibabu Shanmugam, Richa Singh, M. Caner Tol, Koray Yurtseven, Zane Weissman, and alumni, Archanaa Santhana Krishnan, Jacob Grycel, Yuan Yao, Daniel Moghimi, Berk Gulmezoglu, Gizem Selcan Cetin, Mehmet Sinan Inci, Michael Moukarzel, Wei Dai and Cong Chen.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, quantum computers have made steady progress to the point where they are considered a threat to traditional public-key cryptosystems based on the conjectured hardness of problems such as integer factorization and discrete logarithm. In a landmark result, Shor introduced an algorithm [154] that can solve the classically conjectured hard problems of factorization and discrete logarithm in polynomial time with the aid of a quantum computer. Symmetric-key systems will also be affected, albeit to a lesser extent. Using Grover's algorithm [64] one may recover symmetric keys by searching through the key-space with square-root time complexity. Hence one may overcome Grover, by equivalently doubling key lengths of symmetric schemes and output sizes of hash functions. As Key Encapsulation Mechanism (KEM) uses public-key schemes to exchange the symmetric keys, there is a need to develop schemes based on quantum-secure hard problems.

To aid the transition to post-quantum cryptography (PQC), the US NIST announced a PQC standardization process in 2016 [129]. The process started with 82 submissions for public-key encryption (PKE), KEM, and digital signatures. 69 schemes were passed into Round 1, 26 were able to get into Round 2 and currently, there are seven finalists and eight alternate candidates in Round 3 expected to be completed by the end of 2022. Similar schemes were merged together and some were attacked by the cryptographic community and were taken out of the competition [44, 147, 40, 63, 5, 156, 157, 136, 91]. There are five categories based on the underlying hard problems: lattice-based, code-based, hash-based, isogeny-based, and multivariate schemes. These schemes offer varying key sizes under varying performance figures. Multivariate is known to be very efficient for resource constraint devices but on the other hand, the key sizes are quite large. Lattice-based schemes have comparatively compact keys and exhibit better performance. Five out of seven finalists belong to the lattice category; two of the lattice schemes Dilithium [49] and Falcon [54] are digital signatures. The only remaining signature scheme is Rainbow, which is already attacked by Ward Beullens [14] in 2022. Dilithium belongs to the CRYSTALS family having another finalist KYBER which is a KEM. Both are based on the conjectured hard module Learning With Errors (LWE) problem.

The cryptographic community as well as companies have started integrating the finalists from the NIST competition into existing cryptographic libraries like OpenSSL. An open-source project named Open Quantum Safe

(OQS) [159] aims to support the development and prototyping of quantum-resistant cryptography. PQShield [141] is providing four different products for hardware and firmware for embedded devices, SDK for mobile and server technologies, and encryption solution for messaging platforms. Another company, QuSecure [144], is providing a software solution to protect the data at rest. The transition from classic to post-quantum algorithms is urgently needed to ensure forward secrecy.

According to the status report on the Second Round of the NIST PQC standardization process [3], evaluation is based on three criteria: 1) Security. 2) Cost and performance. 3) Algorithm and implementation characteristics. The third criterion is very important since even if a scheme is mathematically secure, it may succumb to side-channel and fault attacks targeting the implementation. Indeed, in recent years, numerous side-channel attacks e.g. [27, 137, 21, 145, 139, 142, 51, 41, 131, 7, 94, 98] and fault attacks e.g. [126, 28, 146, 140, 105, 17, 50, 19, 147, 59] have been demonstrated by the research community on PQC schemes. These include cache attacks, power and EM side-channels, EM and laser injections, clock glitches, and the Rowhammer attack. A major challenge in applying these attacks on PQC schemes is that PQC schemes have massive key sizes (kB) while the attacks can reveal only a few bits per attempt. Yet, even a few revealed key bits may reduce the security strength below the level specified by the PQC standard. Another challenge for side-channel attacks is that all the finalists have constant-time AVX2 implementations for example they do not have secret

dependent branches or other timing variations based upon the secret key. Also, the schemes in Round 3 have withstood more than five years of cryptanalysis by the cryptographic community and the underlying hard problems have been analyzed for decades. For the fault attacks like Differential Fault Attacks (DFA), PQC schemes already have mitigation by randomizing the nonce values. The NIST Round 2 version of LUOV, specifically added a random salt for every message and required randomly generated vinegars to defend against the side-channel and fault injection attacks. To prevent DFA, Round 2 Dilithium added signature randomization by using a random nonce in every signature generation. DFA works on a principle of taking the difference between the correct and faulty pair of output and mathematically recovering the secret key. After this mitigation, the same message signed or encrypted twice gives a different signature or ciphertext and the attacker is unable to collect a faulty and correct pair of the same message.

In this dissertation, we show that nonce randomization mitigation is insufficient for fault attacks. We have proposed bit-tracing attack for the LUOV scheme and Signature Correction Attack for Dilithium scheme that only needs faulty signatures. The main idea is to correct the faulty signature for all possible bit-flips in the secret key by using verification algorithm as an oracle. We achieve this by identifying the unique impact of every faulty secret key bit on the signature and subtracting it from the faulty signature. We keep verifying our tries using verification algorithm and end up recovering the secret key bits. This technique recovers one secret key bit for one

faulty signature and may not be used to recover the full secret key. However, we have demonstrated that by analytical approaches and techniques, we can amplify our attack to recover the full secret key in case of LUOV and successfully reduced the security level of Dilithium from 2^{128} to 2^{81}.

Both bit-tracing and Signature Correction attacks are independent of the fault mechanisms and only need faulty signatures with single-bit faults in the secret keys. We have chosen Rowhammer to demonstrate these attacks which is a software-induced fault mechanism, capable of injecting hardware faults into the memory. Rowhammer does not require any physical access and can be carried out remotely even in cloud scenarios [86, 55, 42, 89, 143, 167]. In all of our Rowhammer experiments, we have used SPOILER for contiguous memory detection. Contiguous memory is a requirement to efficiently implement the Rowhammer attack. Previous works use HugePages for the same purpose but that needs special settings. However, SPOILER can detect contiguous memory in normal settings and without any special privileges. The Discovery of SPOILER in this dissertation has led us practically demonstrate the Rowhammer attack on multiple post-quantum signature schemes.

1.1 Contributions

This dissertation investigates post-quantum signature schemes against software-induced fault attacks. The schemes investigated in this work already incorporate nonce randomization fault countermeasure. However, this

research identifies a number of vulnerabilities in the implementations as well as in the design of these schemes. The primary contributions of this dissertation are summarized below and detailed contributions are mentioned in each chapter:

- We have discovered a novel microarchitectural leakage named SPOILER (CVE-2019-0162) in all generations of Intel Core processors starting from the 1^{st} generation (2008). It leaks critical physical address information stemming from the false dependency hazards during speculative load operations.

- We demonstrate that SPOILER can be used to find contiguous memory with normal user-level privilege and without any special settings like HugePages. Contiguous memory is required by double-sided Rowhammer attack to identify consecutive rows inside the DRAM banks.

- We have identified a number of DRAM modules, vulnerable to the Rowhammer attack. These Rowhammer setups have helped us identify and practically demonstrate a number of vulnerabilities in post-quantum signature schemes.

- We have introduced and implemented bit-tracing attack on the LUOV signature scheme. The attack is purely mathematical that uses faulty signatures to trace and recover secret key bits. We then amplify the efficiency of our attack using an analytical approach for full key recovery and call this hybrid attack as QuantumHammer.

- We practically demonstrate the Rowhammer attack on constant-time AVX2 optimized implementation of LUOV and collect the faulty signatures. Our attack induces hardware fault but through software, i.e. we do not assume any physical access to the device. This also permits remote attacks on shared cloud servers or in sandboxed environments.

- We have introduced and implemented Signature Correction Attack on Dilithium signature scheme reducing its security strength from 2^{128} to 2^{81}. The attack makes use of the faulty signatures and mathematically traces back to the secret key bits. The idea looks similar to the bit-tracing attack, however, the mathematics is completely different. This is because LUOV is a multivariate scheme and Dilithium is based on lattices.

- We practically demonstrate the faults in the Dilithium scheme by Rowhammer attack and collect the faulty signatures. The Signature Correction Attack, however, still applies if the faulty signatures are collected through some other fault mechanism. The only requirement is that they must have single-bit faults in the secret key of Dilithium.

- We demonstrate faults in the Dilithium signature scheme by applying the Plundervolt attack. The attack is based on undervolting the CPU voltages through software. We have successfully collected faulty signatures on various CPU voltages.

- We have discovered an interesting behavior of Plundervolt which af-

fects the memory `write` operations stemming from the out-of-order execution of a branch.

1.2 Outline

This dissertation is the result and combination of publications [86, 126, 87, 70, 168] in various top-tier peer-reviewed security conferences. First, we describe the necessary background in Chapter 2. Chapter 3 presents SPOILER, a collaborative work with Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth and Berk Sunar. This work was published in USENIX Security 2019. Chapter 4, QuantumHammer, is the result of collaborative work with Koksal Mus and Berk Sunar, published in CCS 2020. In Chapter 5, Signature Correction Attack is presented, a collaborative work with Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. This work is accepted in EuroS&P 2022. Chapter 6 presents Plundervolt attack on the Dilithium signature scheme and Chapter 7 concludes the dissertation. We would like to mention our collaborative work with Berk Gulmezoglu et al. on user privacy in mobile devices, published in ACM ASIACCS 2019 [70] and with M. Caner Tol et al. on backdoor injection attacks on DNNs using Rowhammer [168].

Chapter 2

Background

This dissertation implements software-induced fault attacks such as Rowhammer and Plundervolt on post-quantum signature schemes. In this chapter, first we provide background information on computer architecture required to understand SPOILER. Then we explain the software-induced fault attacks and finally the structure of the LUOV and Dilithium signature schemes.

2.1 Computer Architecture

2.1.1 Memory Management

The virtual memory manager shares the DRAM across all running tasks by assigning isolated virtual address spaces to each task. The assigned memory is allocated in pages, which are typically 4 kB each, and each virtual page

will be stored as a physical page in DRAM through a virtual-to-physical page mapping. Memory instructions operate on virtual addresses, which are translated within the processor to the corresponding physical addresses. The page offset comprising the least significant 12 bits of the virtual address is not translated. The processor only translates the bits in the rest of the virtual address, the virtual page number. The OS is the reference for this translation, and the processor stores the translation results inside the TLB. As a result, repeated translations of the same address are performed more efficiently.

2.1.2 Cache Hierarchy

Modern processors incorporate multiple levels of caches to avoid the DRAM access latency. The cache memory on Intel processors is organized into sets and slices. Each set can store a certain number of lines, where the line size is 64 Bytes. The 6 Least Significant Bits (LSBs) of the physical address are used to determine the offset within a line and the remaining bits are used to determine which set to store the cache line in. The number of physical address bits that are used for mapping is higher for the LLC, since it has a large number of sets, e.g., 8192 sets. Hence, the untranslated part of the virtual address bits which is the page offset, cannot be used to index the LLC sets. Instead, higher physical address bits are used. Further, each set of LLC is divided into multiple slices, one slice for each logical processor. The mapping of the physical addresses to the slices uses an undocumented

function [84]. When the processor accesses a memory address, a cache hit or miss occurs. If a miss occurs in all cache levels, the memory line has to be fetched from DRAM. Accesses to the same memory address would be served from the cache unless other memory accesses evict that cache line. In addition, we can use the `clflush` instruction, which follows the same memory access check as other memory operations, to evict our own cache lines from the entire cache hierarchy.

2.1.3 Prime+Probe Attack

In the Prime+Probe attack, the attacker first fills an entire cache set by accessing memory addresses that are mapped to the same set, an *eviction set*. Later, the attacker checks whether the victim program has displaced any entry in the cache set by accessing the eviction set again and measuring the execution time. If this is the case, the attacker can detect congruent addresses, since the displaced entries cause an increased access time. However, finding the eviction sets is difficult due to the unknown translation of virtual addresses to physical addresses. Since an unprivileged attacker has no access to HugePages [80] or the virtual-to-physical page mapping such as the `pagemap` file [113], knowledge about the physical address bits greatly speeds up the eviction set search.

2.1.4 Memory Order Buffer

The processor manages memory operations using the Memory Order Buffer (MOB). MOB is tightly coupled with the data cache. The MOB assures that memory operations are executed efficiently by following the Intel memory ordering rule [122] in which memory `stores` are executed in-order and memory `loads` can be executed out-of-order. These rules have been enforced to improve the efficiency of memory accesses while guaranteeing their correct commitment. Figure 2.1 shows the MOB schematic according to Intel [2, 1]. The MOB includes circular buffers, *store buffer*[1] and *load buffer* (LB). A `store` will be decoded into two micro-ops to store the address and data, respectively, in the store buffer. The store buffer enables the processor to continue executing other instructions before commitment of the `stores`. As a result, the pipeline does not have to stall for the `stores` to complete. This further enables the MOB to support out-of-order execution of the `load`.

Store forwarding is an optimization mechanism that sends the `store` data to a `load` if the `load` address matches any of the store buffer entries. This is a speculative process, since the MOB cannot determine the true dependency of the `load` on `stores` based on the store buffer. Intel's implementation of the store buffer is undocumented, but a potential design suggests that it will only hold the virtual address, and it may include part of the physical address [2, 1, 104]. As a result, the processor may falsely forward the data, al-

[1]Store buffer consists of *Store Address Buffer (SAB)* and *Store Data Buffer (SDB)*. For simplicity, we use *Store Buffer* to mention the logically combined SAB and SDB units.

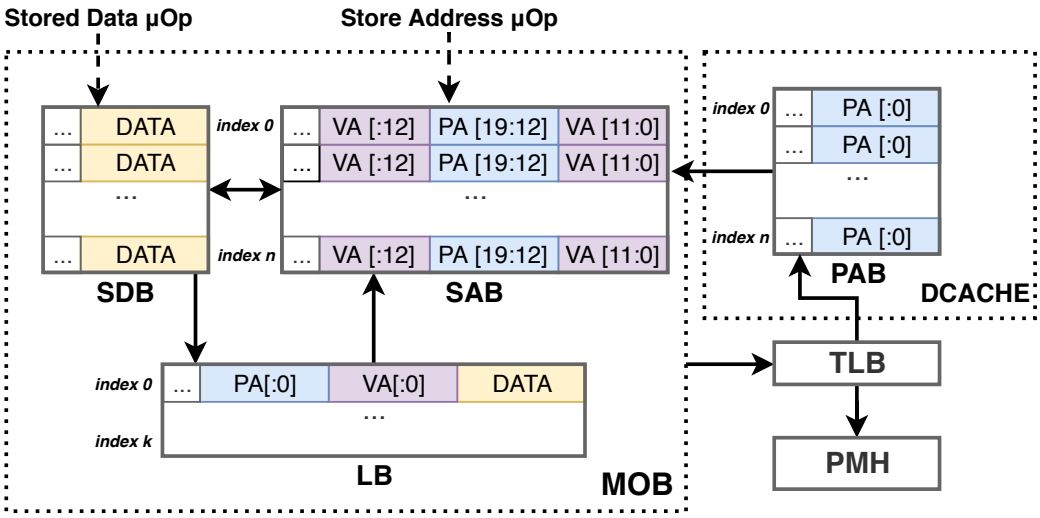


Figure 2.1: The Memory Order Buffer includes circular buffers SDB, SAB and LB. SDB, SAB and PAB of the DCACHE have the same number of entries. SAB may initially hold the virtual address and the partial physical address. MOB requests the TLB to translate the virtual address and update the PAB with the translated physical address.

though the physical addresses do not match. The complete resolution will be delayed until the `load` commitment, since the MOB needs to ask the TLB for the complete physical address information, which is time-consuming. Additionally, the data cache (DCACHE) may hold the translated store addresses in a Physical Address Buffer (PAB) with an equal number of entries as the store buffer.

2.1.5 Speculative Load Hazards

As we mentioned earlier, memory `loads` can be executed out-of-order and before the preceding memory `stores`. If one of the preceding `stores` modifies the content of a location in memory, the memory `load` address is referring

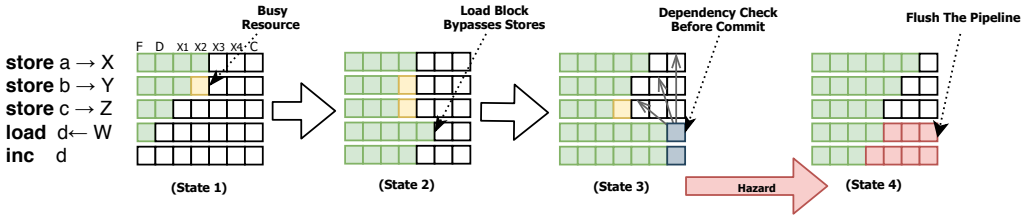


Figure 2.2: The speculative load is demonstrated on a hypothetical processor with 7 pipeline stages: F = Fetch, D = Decode, X_{1-4} = Executions, and C = Commit. When the memory `stores` are blocked competing for resources (State 1), the `load` will bypass the `stores` (State 2). The load block including the dependent instructions will not be committed until the dependency of the address W versus X,Y,Z are resolved (State 3). In case of a dependency hazard (State 4), the pipeline is flushed and the `load` is restarted.

to, out-of-order execution of the `load` will operate on stale data, which results in invalid execution of a program. This out-of-order execution of the memory `load` is a speculative behavior, since there is no guarantee during the execution time of the `load` that the virtual addresses corresponding to the memory `stores` do not conflict with the `load` address after translation to physical addresses. Figure 2.2 demonstrates this effect on a hypothetical processor with 7 pipeline stages. As multiple `stores` may be blocked due to limited resources, the execution of the `load` and dependent instructions in the pipeline, the *load block*, will bypass the `stores` since the MOB assumes the load block to be independent of the `stores`. This speculative behavior improves the memory bottleneck by letting other instructions continue their execution. However, if the dependency of the `load` and preceding `stores` is not verified, the load block may be computed on incorrect data which is either falsely forwarded by store forwarding (false dependency), or loaded from

a stale cache line (unresolved true dependency). If the processor detects a false dependency before committing the `load`, it has to flush the pipeline and re-execute the load block. This will cause observable performance penalties and timing behavior.

Dependency Resolution

Dependency checks and resolution occur in multiple stages depending on the availability of the address information in the store buffer. A `load` instruction needs to be checked against all preceding `stores` in the store buffer to avoid false dependencies and to ensure the correctness of the data. A potential design [76, 104],[2] suggests the following stages for the dependency check and resolution, as shown in Figure 2.3:

1. **Loosenet**: The first stage is the *loosenet* check where the page offsets of the `load` and `stores` are compared[3]. In case of a loosenet hit, the compared `load` and `store` may be dependent and the processor will proceed to the next check stage.

2. **Finenet**: The next stage, called *finenet*, uses upper address bits. The *finenet* can be implemented to check the upper virtual address bits [76], or the physical address tag [104]. Either way, it is an intermediate stage,

[2]The implementation of the MOB used in Intel processors is unpublished and therefore we cannot be certain about the precise architecture. Our results agree with some of the possible designs that are described in the Intel patents.

[3]According to `Ld_Blocks_Partial:Address_Alias` Hardware Performance Counter (HPC) event[81], loosenet is defined by Intel as the mechanism that only compare the page offsets.

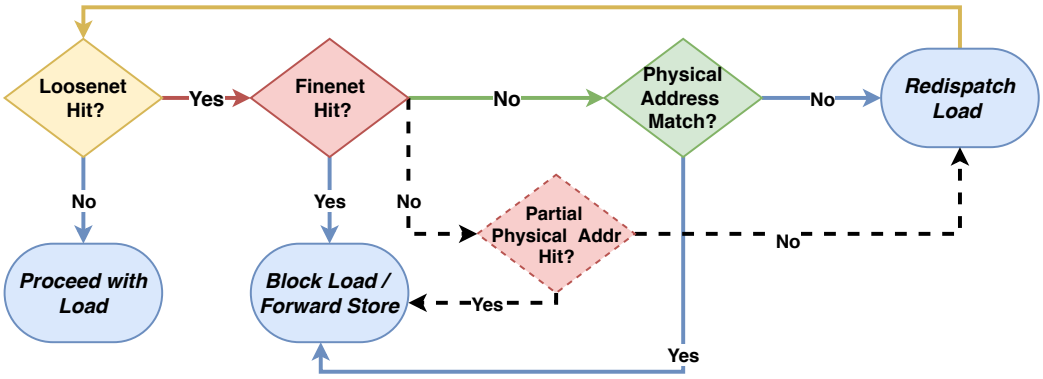


Figure 2.3: The dependency check logic: *loosenet* initially checks the least 12 significant bits (page offset) and the *finenet* checks the upper address bits, related to the page number. The final dependency using the physical address matching might still fail due to partial physical address checks.

and it is not the final dependency resolution. In case of a *finenet* hit, the processor blocks the `load` and/or forwards the store data, otherwise, the dependency resolution will go into the final stage.

3. **Physical Address Matching**: At the final stage, the physical addresses will be checked. Since this stage is the final chance to resolve potential false dependencies, we expect the full physical address to be checked. However, one possible design suggests that if the physical addresses are not available, the physical address matching returns true and continues with the store forwarding [76].

Since the page offset is identical between the virtual and physical address, loosenet can be performed as soon as the `store` is decoded. [2] suggests that the store buffer only holds bit 19 to 12 of the physical address. Although

the PAB holds the full translated physical address, it is not clear in which stage this information can be available to the MOB. As a result, the *finenet* check may be implemented based on checking the partial physical address bits. As we verify later, the dependency resolution logic may fail to resolve the dependency at multiple intermediate stages due to the unavailability of the full physical address.

2.2 Software-Induced Fault Attacks

In this section, we provide the background of two software-induced fault mechanisms, Rowhammer and Plundervolt.

2.2.1 Rowhammer Attack

We are using Rowhammer as a tool to inject faults. We briefly review the concept and operation of the Rowhammer attack, covering memory management, DRAM organization, address translation and applicability on cloud environments.

Every process has its own virtual address space which is divided into virtual pages, typically of size 4 kB. Memory Management Unit (MMU) translates the virtual addresses into physical addresses and keeps track in form of page tables. The memory controller integrated in modern processor then translates these physical addresses into channels, ranks and banks inside the DRAM. This DRAM addressing varies from system to system and is

not publicly disclosed for Intel CPUs, although the DRAM addressing was reverse-engineered for some of the systems by Pessl *et al.* in 2016 [138]. Each bank then further consists of rows and columns sharing the same row buffer. A DRAM row consists of 64K cells and a cell is composed of a transistor and a capacitor. Data is stored in these capacitors in form of charge and interpreted as a zero or a one according to predefined threshold levels. As capacitors leak charge over time, there is a refresh mechanism to restore the charge of all the DRAM cells every 64ms.

As the DRAM manufacturers are trying to make memories more compact, these rows of cells are getting physically closer leading to disturbance errors from one DRAM row to another. If one row is accessed repeatedly, it might cause electrical interference with the neighboring row due to insufficient insulation and the cells in the neighboring row may leak faster. If the leakage is faster than the refresh frequency, the cells can not maintain their state, which may lead to bit-flips. This is known as the Rowhammer effect which was first introduced by Kim *et al.* in 2014 [100]. Using Rowhammer, an attacker with access to a row next to the victim row in DRAM is able to cause bit-flips in the victim's memory, even when the attacker resides in a process completely separate from the victim process. If the attacker hammers one row which causes bit-flips in the neighboring row, it is called single-sided Rowhammer.

After this discovery, Seaborn *et al.* [151] introduced the double-sided Rowhammer which is far more effective than the earlier single-sided Rowham-

mer. In a double-sided Rowhammer, the attacker hammers two rows sandwiching the victim row, leaking the victim cells even faster. Veen *et al.* [173] in 2016 showed that it is also applicable on mobile platforms. Gruss *et al.* [65] introduced one-location hammering and achieved root access with opcode flipping in `sudo` binary in 2018. Gruss *et al.* [67] and Ridder *et al.* [42] have shown that Rowhammer can be applied through JavaScript remotely. Tatar *et al.* [164] and Lip *et al.* [115] have proved that it can be executed over the network. Rowhammer is also applicable in cloud environments [181, 36] and heterogeneous FPGA-CPU platforms [176]. In 2020, Kwong *et al.* [107] demonstrated that Rowhammer is not just an integrity problem but also a confidentiality problem.

There have been many efforts on Rowhammer detection [85, 35, 186, 75, 133, 68, 9, 38] and neutralization [67, 173, 25]. Gruss *et al.* [65] have shown that all of these countermeasures are ineffective. Some countermeasures require hardware modification, bootloader or BIOS update [25, 8, 97, 100, 61, 79] but they are not all implemented. Cojocar *et al.* [37] in 2019 reverse-engineered the Error Correction Code (ECC) memories showing that ECC countermeasure is not secure either. Another hardware countermeasure Target Row Refresh (TRR) has also been recently bypassed by Frigo *et al.* [57] using many-sided Rowhammer on DDR4 chips. The same work has been extended by Ridder *et al.* [42] to attack TRR-enabled DDR4 chips from JavaScript. They claim that more than 80% of the DRAM chips in the market are still vulnerable to the Rowhammer attack.

Impact of Technology Scaling

As the DRAM chip densities are increasing in every generation of DDR (see Figure 2.4), they are becoming more vulnerable to Rowhammer vulnerability [99, 127, 55]. The first Rowhammer attack by Kim et al. [100] was on DDR3 memories, whereas DDR and DDR2 do not have this vulnerability. Kim et al. [99] have demonstrated that the minimum number of hammers required to induce a bit-flip (Hammer Count) greatly reduces in newer DRAM chips (From 69.2k to 22.4k in DDR3, 17.5k to 10k in DDR4, and 16.8k to 4.8k in LPDDR4).

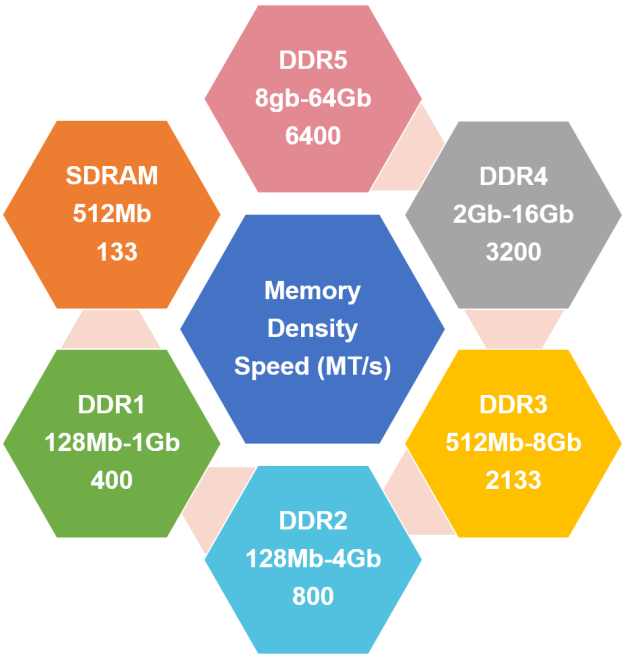


Figure 2.4: DRAM Memory Densities Comparison [162].

Cojocar et al. [37] have demonstrated that Memory Controller-based

ECC is susceptible to Rowhammer attack. Newer generations such as DDR5 and LPDDR4 now incorporate on-die ECC to reduce data corruption rates, which operates entirely within the DRAM. It would be worth investigating if on-die ECC memories have Rowhammer vulnerability [187]. DDR5 also has an optional Refresh Management (RFM) as a mitigation for Rowhammer attack. RFM is similar in concept to TRR in DDR4 which monitors excessive activations (ACT) per row, RFM is instead based on ACT per bank. Another concern with ECC is that DRAM manufacturers are relying on ECC for memory integrity and lowering the DRAM refresh rate to save power [37]. A lower refresh rate dramatically increases the number of bit-flips [100], making it easier to bypass ECC [37].

Graphics DDR (GDDR) is a type of DRAM specifically designed for applications requiring high bandwidth such as Graphics Processing Units (GPUs). There are no Rowhammer bit-flips reported in literature so far on GDDR memories. However, Zhang et al. [187] write in their future work to mount an end-to-end attack against the DNN model of the victim tenant by inducing bit-flips in GDDR5.

There is also a need to investigate latest 3D-stacked memory technologies such as High Bandwidth Memory (HBM) [90]. 3D stacking enables stacking of volatile memory like DRAM directly on top of a microprocessor, significantly reducing the transmission delay. It would be worth investigating if there is any electromagnetic coupling between these vertically stacked DRAM chips in this third dimension.

2.2.2 Plundervolt Attack

Dynamic voltage and frequency scaling (DVFS) has been introduced to manage heat and power consumption in modern systems. For this reason, the CPUs exposed a privileged interface to software for dynamic voltage and frequency scaling. In 2020, Murdock et al. [125] have shown that this interface can be abused to compromise Intel SGX integrity, dubbed Plundervolt. There is another similar concurrent work named V0LTpwn by Kenjar et al. [96] which also attacks the Intel SGX. Both groups had a responsible disclosure with Intel and a security advisory INTEL-SA-00289 / CVE-2019-11157 [82] has been assigned to the vulnerability. As a mitigation strategy, Intel disabled the mailbox interface from the software and released microcode and BIOS updates. *However, Murdock et al. believe that this may not cover the root cause for Plundervolt.*

The idea of Plundervolt and V0LTpwn is not completely new, there was a similar software-induced fault attack on ARM devices by Tang et al. [163] in 2017, named as CLKscrew. However, it was unclear whether a similar technique may work for Intel devices. Also, CLKscrew was based on frequency changes, whereas Plundervolt is based on undervolting the CPUs. Both however are capable of compromising the integrity of Trusted Execution Environments (TEE) and recovering secret keys from cryptographic algorithms. This led to a new class of software-induced fault attacks that does not require physical access to the victim machine and can be executed remotely. As compared to the Rowhammer attack, which causes bit-flips in

memory, this new class of fault attacks is capable of inducing computation faults which are also repeatable with high probability.

2.3 Post-Quantum Signature Schemes

In this section, we introduce two of the post-quantum signature schemes, LUOV [15] and Dilithium [49]. Their key generation, signature generation and verification algorithm are described briefly. To give an idea about the massive key and signature sizes of post-quantum signature schemes, we have compared the round 2 candidates with ECDSA in Figure 2.5.

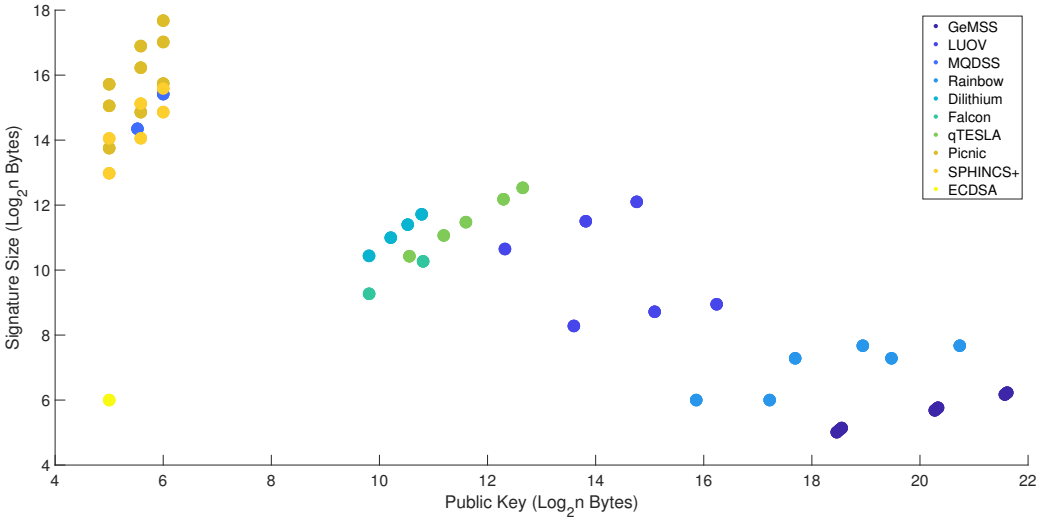


Figure 2.5: Public key and signature size comparison between PQC signature schemes and ECDSA. Both x-axis and y-axis are on logarithmic scale.

2.3.1 Lifted Unbalanced Oil and Vinegar (LUOV)

Consider a system of m Multivariate Quadratic (MQ) polynomials with n variables $x_1, \ldots, x_n$

$$p^k(x_1, \ldots, x_n) = \sum_{i=1}^{n} \sum_{j=i}^{n} p_{ij}^k \cdot x_i x_j + \sum_{i=1}^{n} p_i^k \cdot x_i + p_0^k \tag{2.1}$$

Note that, since we are using boolean equations, we reserved the exponent for use as an index.

Solving the MQ system is conjectured hard for sufficiently large m and n. The MQ challenge by Yasuda et al. [184] gives a way to gauge the difficulty of solving real-life MQ instances with moderate size instances. A multivariate signature scheme may be built around the MQ system: the coefficients represent the public key $\mathcal{P}$, the system is solved for the hash of the message, the variable values that satisfy the equation (the solution to the MQ system) represents the signature. It is hard to solve this system and find a signature for the desired message unless we have a trapdoor $\mathcal{P} = \mathcal{S} \circ \mathcal{F} \circ \mathcal{T}$, where $\mathcal{S}$ and $\mathcal{T}$ are the secret invertible linear transformations and $\mathcal{F}$ is the secret quadratic map having a special structure given as

$$f^k(x_1, \ldots, x_n) = \sum_{i=1}^{v} \sum_{j=i}^{n} \alpha_{ij}^k \cdot x_i x_j + \sum_{i=1}^{n} \beta_i^k \cdot x_i + \gamma^k \tag{2.2}$$

Here, n variables $x_1, \ldots, x_n$ are divided into two parts, $x_1, \ldots, x_v$ as the vinegar variables and $x_{v+1}, \ldots, x_n$ as the m oil variables where $n = v + m$.

The parameters α_{ij}^k, β_i^k and γ^k are chosen randomly from a finite field $\mathbb{F}$ where k ranges from 1 to m. The specialty of this structure is that there is no quadratic term with multiplication of two oil variables. So, if vinegar variables are chosen randomly and inserted into the system, it collapses into a linear system which can be easily solved for the remaining oil variables using Gaussian elimination. Note that, oil variables are public whereas vinegars are kept secret. The structure of $\mathcal{F}$ is then hidden using a secret linear transformation $\mathcal{T}$.

The first Oil and Vinegar scheme was proposed by Patarin [132] in 1997 which was broken by Kipnis and Shamir [102] in 1998. The modified version of the scheme named UOV was then proposed by Kipnis et al. [101] in 1999. The main difference was to unbalance the number of oil and vinegar variables by increasing the number of vinegar variables to render the attack ineffective.

The public keys of UOV are prohibitively large to prevent wide-scale deployment. This motivated another proposal named LUOV by Beullens et al. [15]. LUOV was submitted to NIST for the PQC standardization process and is a Round 2 finalist. One of the main innovations of LUOV is to reduce the large key sizes in UOV in the way that keys are generated and stored. Instead of storing and transferring large public keys every time, LUOV makes use of the idea that generating the keys whenever needed using a sponge type hash function and using a private seed for the private key and public seed and additional $Q_2 \in \mathbb{F}_2^{m \times m(m+1)/2}$ matrix for the public key. Here we give a brief description of the LUOV scheme. A detailed description and supporting

documentation can be found in [16].

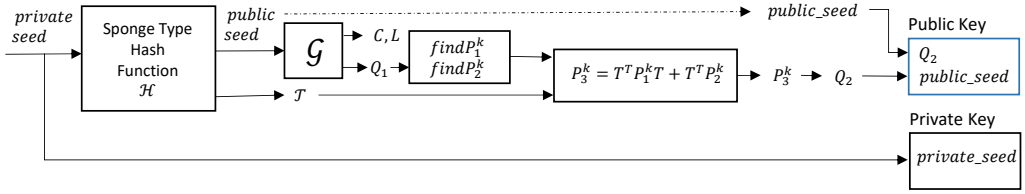


Figure 2.6: LUOV public and private key generation processes.

Key Generation

Key generation process is depicted in Figure 2.6. Briefly, *private_seed* is hashed by a sponge type hash function $\mathcal{H}$ generating *public_seed* and $v \times m$ private binary secret linear transformation matrix $\mathcal{T}$. Another hash function $\mathcal{G}$ generates public parameters $C \in \mathbb{F}_2^m$, $L \in \mathbb{F}_2^{m \times n}$ and $Q_1 \in \mathbb{F}_2^{m \times v(v+1)/2+vm}$ by hashing the *public_seed*. A $v \times v$ upper triangular matrix P_1^k and $v \times m$ matrix P_2^k are generated by $findP_1^k$ and $findP_2^k$ algorithms respectively using Q_1 and an integer counter k. The details of the algorithms can be found in [16]. In this dissertation, we do not need the details of generation of P_1^k and P_2^k, hence, we will consider P_1^k and P_2^k as given fixed random binary matrices. P_1^k and P_2^k are given as:

$$P_1^k = \begin{pmatrix} a_{1,1}^k & a_{1,2}^k & \cdots & a_{1,v}^k \\ 0 & a_{2,2}^k & \cdots & a_{2,v}^k \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{v,v}^k \end{pmatrix}_{v,v}, P_2^k = \begin{pmatrix} b_{1,1}^k & b_{1,2}^k & \cdots & b_{1,m}^k \\ b_{2,1}^k & b_{2,2}^k & \cdots & b_{2,m}^k \\ \vdots & \vdots & \ddots & \vdots \\ b_{v,1}^k & b_{v,2}^k & \cdots & b_{v,m}^k \end{pmatrix}_{v,m}$$

Intermediate $m \times m$ matrix P_3^k is generated by the formula $P_3^k = -T^T P_1^k T + T^T P_2^k$ where $k = 1, \ldots, m$. Therefore, $(i,j)^{th}$ element of P_3^k is

$$p_3^k(i,j) = \sum_{\alpha=1}^{v} t_{\alpha,j} \sum_{l=1}^{\alpha} t_{l,i} a_{l,\alpha} + \sum_{\gamma=1}^{v} t_{\gamma,i} b_{\gamma,j} \quad \text{for} \quad i,j \in \{1, \cdots, m\}. \tag{2.3}$$

$m \times \frac{m(m+1)}{2}$ binary public key matrix Q_2 is generated by Equation 2.4. It is important to emphasize that P_3^k constitutes the k^{th} row of Q_2.

$$Q_{2(k,\beta_{i,j})} = \begin{cases} p_3^k(i,j) & , i = j \\ p_3^k(i,j) \oplus p_3^k(j,i) & , i < j \end{cases} \tag{2.4}$$

where $\beta_{i,j} = (i-1)m + j - \sum_{\alpha=0}^{i-1} \alpha$, $i,j,k \in \{1, \cdots, m\}$ and $\beta_{i,j} = 1, \ldots, m(m+1)/2$.

For instance the kth row of Q_2 is of the following form:

$(p_3^k(1,1), p_3^k(1,2) \oplus p_3^k(2,1), p_3^k(1,3) \oplus p_3^k(3,1), \cdots, p_3^k(2,2),$
$p_3^k(2,3) \oplus p_3^k(3,2), \cdots, p_3^k(3,3), \cdots, p_3^k(m,m))$.

Key generation algorithm outputs *private_seed* as the private key and *public_seed* and Q_2 as the public key. Public map $\mathcal{P}$ needed for signature verification is the concatenation of C, L, Q_1 and Q_2.

Signature Generation

Signature generation primitive of LUOV is shown in Figure 2.7 and explained in Algorithm 1 which is divided into four parts, Parameter Generation, Augmented Matrix Generation, Gaussian Elimination and Generation of the Signature for the sake of simplicity. It is important to note that o is publicly available in the signature. Therefore, it is known to the adversary.

Algorithm 1 LUOV Signature Generation

Input: *private_seed* , Message M
Output: Signature $(S||salt)$

1: **Parameter Generation:** Binary linear transformation $\mathcal{T}$ and *public_seed* are generated by the hash of random *private_seed*. Then, the hash of *public_seed* outputs C, L and Q_1. Concatenation of message M and a random *salt* hashed by $\mathcal{H}$ produces message h to be signed.
2: **Augmented Matrix Generation:** Insert randomly chosen vinegar variables v into the MQ system $\mathcal{F}(s') = h$ which collapses to a linear system. The augmented matrix generation algorithm is explained in Algorithm 14.
3: **Gaussian Elimination:** Linear system can be easily solved by Gaussian elimination which gives oil variables o. Note that, oil variables depend on h and v since the other parameters are generated by the same *private_seed*.
4: **Generation of the Signature:** Signature S is the concatenation of $s = \mathcal{T} \cdot o + v$, o and *salt*.
return $(s||o||salt)$

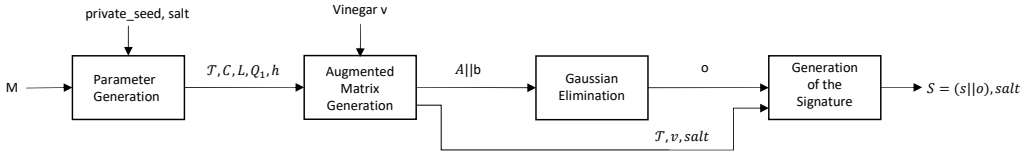


Figure 2.7: Signature generation algorithm explained in four steps.

Signature Verification

The verifier generates C, L and Q_1 from the *public_seed* using the hash function $\mathcal{G}$. These parts are then combined with the publicly available Q_2 to form the public map $\mathcal{P}$. Similar to the signing algorithm, the message M and the *salt* are concatenated, then hashed using $\mathcal{H}$ to form the digest h. If $\mathcal{P}(s) = h$, then the signature is verified, otherwise rejected.

2.3.2 CRYSTALS - Dilithium

The Cryptographic Suite for Algebraic Lattices (CRYSTALS) consists of two cryptographic schemes, Kyber [23], a KEM and Dilithium [49], a digital signature algorithm. The suite has been submitted to the NIST PQC competition by the Crystals team and both the CRYSTALS are among the Round 3 finalists. These algorithms are based on hard problems over module lattices. The security of Dilithium is based on two problems, namely, Learning With Errors (LWE) problem and SelfTargetMSIS problem. Dilithium is essentially based on Bai-Galbraith scheme proposed by Bai and Galbraith [10] in 2014. The design of the scheme is based on "Fiat-Shamir with Aborts" [120]. Dilithium has three security levels 2, 3 and 5 and also have AES versions instead of SHAKE for performance purposes. We shall just briefly explain the key generation, signing and verification algorithms of Dilithium scheme. We refer the reader to the original specifications for details [49].

Key Generation

The secret key vectors s_1 and s_2 of lengths l and k are sampled randomly from a uniform distribution. Each element of these vectors is a polynomial in the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ and the coefficients are of size η, where $q = 2^{23} - 2^{13} + 1$ and $n = 256$. Next, a $k \times l$ matrix $\mathbf{A}$ is generated whose entries are also from R_q with relatively larger coefficients in range q. Then the LWE vector t is computed, part of which is kept secret as t_0 while the other part t_1 is made public. The matrix $\mathbf{A}$ is also made public while s_1 and s_2 are kept secret. Dilithium key generation process can be seen in Algorithm 2 where it outputs pk as public key and sk as secret key. Unlike the Bai-Galbraith scheme, where the whole t was made public, Dilithium just makes t_1 public to reduce the size of the public key. The signature size however, is relatively increased by a small factor.

Algorithm 2 Dilithium Key Generation [49]

Output: pk - Public Key, sk - Secret Key
$\zeta \leftarrow \{0,1\}^{256}$
$(\rho, \varsigma, K) \in \{0,1\}^{256\times 3} \leftarrow H(\zeta)$
$(s_1, s_2) \in S_\eta^l \times S_\eta^k \leftarrow H(\varsigma)$
$\mathbf{A} \in R_q^{k\times l} \leftarrow ExpandA(\rho)$
$t \leftarrow \mathbf{A}s_1 + s_2$
$(t_1, t_0) \leftarrow Power2Round_q(t, d)$
$tr \in \{0,1\}^{384} \leftarrow CRH(\rho \| t_1)$
return $(pk = (\rho, t_1), sk = (\rho, K, tr, s_1, s_2, t_0))$

Signature Generation

Dilithium signing has two modes of operation, deterministic which is the default and randomized, recommended for side-channel and fault attacks scenarios. The nonce y is generated using a seed ρ' which is either deterministic or randomized depending upon the mode of operation. The signature z is generated using the expression $z = y + c \cdot s_1$, where c is the challenge vector derived as depicted in Algorithm 3. An important part of the signing operation is the rejection sampling which checks if the signature z does not leak any secret information. The rejection sampling loop runs for approximately 4 to 7 times until a secure signature is generated. There is a rejection counter κ which is incremented in every loop to generate a different nonce y in each iteration.

Signature Verification

The Dilithium verification algorithm computes the challenge vector $\tilde{c}$ and compares it to the $\tilde{c}$ provided in the signature. Also, it checks the range of coefficients of signature z and the weight of the hint h. If all the three conditions are met, the signature is verified, otherwise rejected. The hint h is not kept secret since it is needed by the verifier to makeup for t_0. We refer to the Dilithium specification for details [49].

Algorithm 3 Dilithium Signature Generation [49]

Input: sk - Secret Key, M - Message
Output: σ - Signature
$\mathbf{A} \in R_q^{k \times l} \leftarrow ExpandA(\rho)$
$\mu \in \{0,1\}^{384} \leftarrow CRH(tr \parallel M)$
$\kappa \leftarrow 0, (z,h) \leftarrow \perp$
$\rho' \in \{0,1\}^{384} \leftarrow CRH(K \parallel \mu)$ (or $\rho' \leftarrow \{0,1\}^{384}$ randomized)
while $(z,h) = \perp$ **do**
    $y \in S_{\gamma 1}^l \leftarrow ExpandMask(\rho', \kappa)$
    $w \leftarrow \mathbf{A}y$
    $w_1 \leftarrow HighBits_q(w, 2\gamma_2)$
    $\tilde{c} \in \{0,1\}^{256} \leftarrow H(\mu \parallel w_1)$
    $c \in B_\tau \leftarrow SampleInBall(\tilde{c})$
    $z \leftarrow y + c \cdot s_1$
    $r_0 \leftarrow LowBits_q(w - c \cdot s_2, 2\gamma_2)$
    if $\|z\| \geq \gamma_1 - \beta$ or $\|r_0\|_\infty \geq \gamma_2 - \beta$ **then**
        $(z,h) \leftarrow \perp$
    else
        $h \leftarrow MakeHint_q(-c \cdot t_0, w - c \cdot s_2 + c \cdot t_0, 2\gamma_2)$
        if $\|c \cdot t_0\|_\infty \geq \gamma_2$ or the # of 1's in $h > \omega$ **then**
            $(z,h) \leftarrow \perp$
        end if
    end if
    $\kappa \leftarrow \kappa + l$
end while
return $\sigma = (z, h, \tilde{c})$

Algorithm 4 Dilithium Signature Verification [49]

Input: pk - Public Key, M - Message, σ - Signature
Output: Verify / Reject
$\mathbf{A} \in R_q^{k \times l} \leftarrow ExpandA(\rho)$
$\mu \in \{0,1\}^{384} \leftarrow CRH(CRH(\rho \parallel t_1 \parallel M)$
$c \leftarrow SampleInBall(\tilde{c})$
$w_1' \leftarrow UseHint_q(h, \mathbf{A}z - ct_1 \cdot 2^d, 2\gamma_2)$
return $[\|z\|_\infty < \gamma_1 - \beta]$ and $[\tilde{c} = H(\mu \parallel w_1')]$ and [# of 1's in $h \leq \omega$]

Chapter 3

SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks

Modern microarchitectures incorporate optimization techniques such as *speculative loads* and *store forwarding* to improve the memory bottleneck. The processor executes the `load` speculatively before the `stores`, and forwards the data of a preceding `store` to the `load` if there is a potential dependency. This enhances performance since the `load` does not have to wait for preceding `stores` to complete. However, the dependency prediction relies on partial address information, which may lead to false dependencies and stall hazards.

In this chapter, we show that the dependency resolution logic that serves

the speculative load can be exploited to gain information about the physical page mappings. Microarchitectural side-channel attacks such as Rowhammer and cache attacks like Prime+Probe rely on the reverse engineering of the virtual-to-physical address mapping. We propose the SPOILER attack which exploits this leakage to speed up this reverse engineering by a factor of 256. Then, we show how this can improve the Prime+Probe attack by a 4096 factor speed up of the eviction set search, even from sandboxed environments like JavaScript. Finally, we improve the Rowhammer attack by showing how SPOILER helps to conduct DRAM row conflicts deterministically with up to 100% chance, and by demonstrating a double-sided Rowhammer attack with normal user's privilege. The latter is due to the possibility of detecting contiguous memory pages using the SPOILER leakage.

3.1 Motivation

Microarchitectural attacks have evolved over the past decade from attacks on weak cryptographic implementations [13] to devastating attacks breaking through layers of defenses provided by the hardware and the Operating System (OS) [170]. These attacks can steal secrets such as cryptographic keys [12, 135] or keystrokes [112]. More advanced attacks can entirely subvert the OS memory isolation to read the memory content from more privileged security domains [114], and to bypass defense mechanisms such as Kernel Address Space Layout Randomization (KASLR) [66, 52]. Rowham-

mer attacks can further break the data and code integrity by tampering with memory contents [100, 151]. While most of these attacks require local access and native code execution, various efforts have been successful in conducting them remotely [164] or from within a remotely accessible sandbox such as JavaScript [130].

Memory components such as DRAM [100] and cache [134] are not the only microarchitectural attack surfaces. *Spectre* attacks on the branch prediction unit [103, 121] imply that side-channels such as caches can be used as a primitive for more advanced attacks on speculative engines. Speculative engines predict the outcome of an operation before its completion, and they enable execution of the following dependent instructions ahead of time based on the prediction. As a result, the pipeline can maximize the instruction level parallelism and resource usage. In rare cases where the prediction is wrong, the pipeline needs to be flushed resulting in performance penalties. However, this approach suffers from a security weakness, in which an adversary can fool the predictor and introduce arbitrary mispredictions that leave microarchitectural footprints in the cache. These footprints can be collected through the cache side-channel to steal secrets.

Modern processors feature further speculative behavior such as *memory disambiguation* and speculative loads [48]. A `load` operation can be executed speculatively before preceding `store` operations. During the speculative execution of the `load`, false dependencies may occur due to the unavailability of physical address information. These false dependencies need to be resolved

to avoid computation on invalid data. The occurrence of false dependencies and their resolution depend on the actual implementation of the memory subsystem. Intel uses a proprietary memory disambiguation and *dependency resolution logic* in the processors to predict and resolve false dependencies that are related to the speculative load. We discover that the dependency resolution logic suffers from an unknown false dependency independent of the 4K aliasing [123, 161]. The discovered false dependency happens during the 1 MB aliasing of speculative memory accesses which is exploited to leak information about physical page mappings.

The state-of-the-art microarchitectural attacks [83, 138] either rely on knowledge of physical addresses or are significantly eased by that knowledge. Yet, knowledge of the physical address space is only granted with root privileges. Cache attacks such as *Prime+Probe* on the Last-Level Cache (LLC) are challenging due to the unknown mapping of virtual addresses to cache sets and slices. Knowledge about the physical page mappings enables more attack opportunities using the Prime+Probe technique. Rowhammer [100] attacks require efficient access to rows within the same bank to induce fast row conflicts. To achieve this, an adversary needs to reverse engineer layers of abstraction from the virtual address space to DRAM cells. Availability of physical address information facilitates this reverse engineering process. In sandboxed environments, attacks are more limited, since in addition to the limited access to the address space, low-level instructions are also inaccessible [67]. Previous attacks assume special access privileges only granted

through weak software configurations [83, 113, 173] to overcome some of these challenges. In contrast, SPOILER only relies on simple operations, `load` and `store`, to recover crucial physical address information, which in turn enables Rowhammer and cache attacks, by leaking information about physical pages without assuming any weak configuration or special privileges.

3.1.1 Contributions

We have discovered a novel microarchitectural leakage that reveals critical information about physical page mappings to userspace processes. The leakage can be exploited by a limited set of instructions, which is visible in all Intel generations starting from the 1^{st} generation of Intel Core processors, independent of the OS and also works from within virtual machines and sandboxed environments. In summary, this chapter:

1. exposes a previously unknown microarchitectural leakage stemming from the false dependency hazards during speculative load operations.

2. proposes an attack, SPOILER, to efficiently exploit this leakage to speed up the reverse engineering of virtual-to-physical mappings by a factor of 256 from both native and JavaScript environments.

3. demonstrates a novel eviction set search technique from JavaScript and compares its reliability and efficiency to existing approaches.

4. achieves efficient DRAM row conflicts and the first *double-sided*

Rowhammer attack with normal user-level privilege using the contiguous memory detection capability of SPOILER.

5. explores how SPOILER can track nearby load operations from a more privileged security domain right after a context switch.

3.1.2 Related Work

Kosher et al. [103] and Maisuradze et al. [121] have exploited vulnerabilities in the speculative branch prediction unit. Transient execution of instructions after a fault, as exploited by Lipp et al. [114] and Bulck et al. [170], can leak the memory content of protected environments. Similarly, transient behavior due to the lazy store/restore of the FPU and SIMD registers can leak register contents from other contexts [160]. New variants of both Meltdown and Spectre have been systematically analyzed [29]. The Speculative Store Bypass (SSB) vulnerability [77] is a variant of the Spectre attack and relies on the stale sensitive data in registers to be used as an address for speculative loads which may then allow the attacker to read this sensitive data. In contrast to previous attacks on speculative and transient behaviors, we discover a new leakage in the undocumented memory disambiguation and dependency resolution logic. SPOILER is **not** a Spectre attack. The root cause for SPOILER is a weakness in the address speculation of Intel's proprietary implementation of the memory subsystem which directly leaks timing behavior due to physical address conflicts. Existing spectre mitigations would

therefore not interfere with SPOILER.

The timing behavior of the 4K aliasing false dependency on Intel processors has been studied [53, 183]. *MemJam* [123] uses this behavior to perform a side-channel attack, and Sullivan et al. [161] demonstrate a covert channel. These works only mention the 4K aliasing as documented by Intel [81], and the authors conclude that the address aliasing check is a two stage approach: Firstly, it uses page offset for the initial guess. Secondly, it performs the final resolution based on the exact physical address. On the contrary, we discover that the undocumented *address resolution* logic performs additional partial address checks that lead to an unknown, but observable aliasing behavior based on the physical address.

Several microarchitectural attacks have been discovered to recover virtual address information and break KASLR by exploiting the Translation Lookaside Buffer (TLB) [78], Branch Target Buffer (BTB) [52] and Transactional Synchronization Extensions (TSX) [88]. Additionally, Gruss et al. [66] exploit the timing information obtained from the `prefetch` instruction to leak the physical address information. The main obstacle to this approach is that the `prefetch` instruction is not accessible in JavaScript, and it can be disabled in native sandboxed environments [185], whereas SPOILER is applicable to sandboxed environments including JavaScript.

Knowledge of the physical address enables adversaries to bypass OS protections [95] and ease other microarchitectural attacks [113]. For instance, the `procfs` filesystem exposes physical addresses [113], and HugePages al-

locate contiguous physical memory [116, 83]. *Drammer* [173] exploits the Android *ION memory allocator* to access contiguous memory. However, access to the aforementioned primitives is restricted on most environments by default. We do not have any assumptions about the OS and software configuration, and we exploit a hardware leakage with minimum access rights to find virtual pages that have the same least significant 20 physical address bits. *GLitch* [56] detects contiguous physical pages by exploiting row conflicts through the GPU interface. In contrast, our attack does not rely on a specific integrated GPU configuration, and it is widely applicable to any system running on an Intel CPU. We use SPOILER to find contiguous physical pages with a high probability and verify it by producing row conflicts. SPOILER is particularly helpful for attacks in sandboxed low-privilege environments such as JavaScript, where previous methods require a time-consuming brute-forcing of the memory addresses [130, 151, 67].

3.2 The SPOILER Attack

The attack model for SPOILER is the same as Rowhammer and cache attacks where the attacker's code is needed to be executed on the same underlying hardware as of the victim. As described in Section 2.1.5, speculative loads may face other aliasing conditions in addition to the 4K aliasing, due to the partial checks on the higher address bits. To confirm this, we design an experiment to observe timing behavior of a speculative load based on

higher address bits. For this purpose, we propose Algorithm 5 that executes a speculative load after multiple `stores` and further makes sure to fill the store buffer with addresses that cause 4K aliasing during the execution of the `load`. Having w as the window size, the algorithm iterates over a number of different memory pages, and for each page, it performs `stores` to that page and all previous w pages within a *window*. Since the size of the store buffer varies between different processor generations, we choose a big enough window ($w = 64$) to ensure that the `load` has 4K aliasing with the maximum number of entries in the store buffer and hence maximum potential conflicts. Following the `stores`, we measure the timing of a `load` operation from a different memory page, as defined by x. Since we want the `load` to be executed speculatively, we can not use a store fence such as `mfence` before the `load`. As a result, our measurements are an estimate of execution time for the speculatively `load` and nearby microarchitectural events. This may include a negligible portion of overhead for the execution of `stores`, and/or any delay due to the dependency resolution. If we iterate over a diverse set of addresses with different virtual and physical page numbers, but the same page offset, we should be able to monitor any discrepancy.

3.2.1 Speculative Dependency Analysis

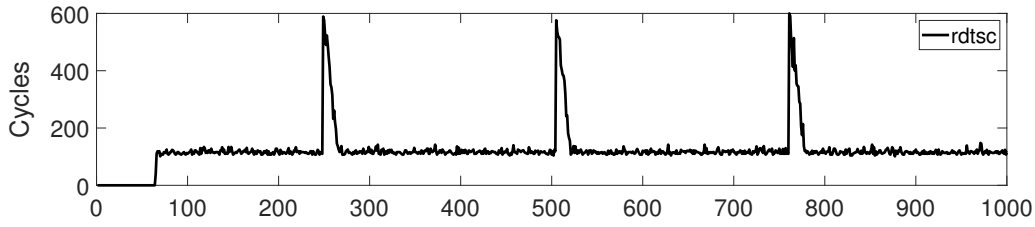
In this section, we use Algorithm 5 and Hardware Performance Counters (HPC) to perform an empirical analysis of the dependency resolution logic. HPCs can keep track of low-level hardware-related events in the CPU. The

Algorithm 5 Address Aliasing

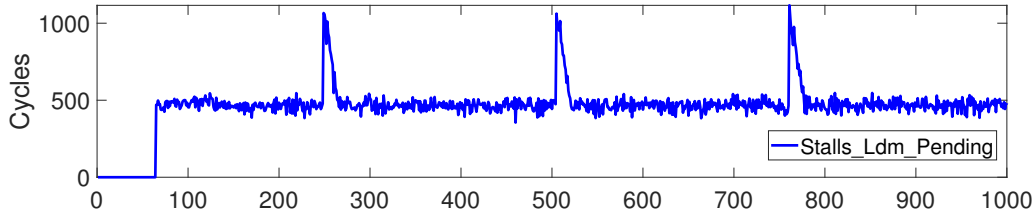
```
1: for p from w to PAGE_COUNT do
2:     for i from w to 0 do
3:         data --store--> buffer[(p − i) × PAGE_SIZE]
4:     end for
5:     t1 = rdtscp()
6:     data <--load-- buffer[x × PAGE_SIZE]
7:     t2 = rdtscp()
8:     measure[p] ← t2 − t1
9: end for
10: return measure
```

counters are accessible via special purpose registers and can be used to analyze the performance of a program. They provide a powerful tool to detect microarchitectural components that cause bottlenecks. Software libraries such as Performance Application Programming Interface (PAPI) [165] simplifies programming and reading low-level HPC on Intel processors. Initially, we execute Algorithm 5 for 1000 different virtual pages. Figure 3.1(a) shows the cycle count for each iteration with a set of 4 kB aliased store addresses. Interestingly, we observe multiple step-wise peaks with a very high latency. Then, we use PAPI to monitor 30 different performance counters listed in Table A.1 in the Appendix while running the same experiment. At each iteration, only one performance counter is monitored alongside the aforementioned timing measurement. After each speculative load, the performance counter value and the `load` time are both recorded. Finally, we obtain the timings and performance counter value pairs as depicted in Figure 3.1.

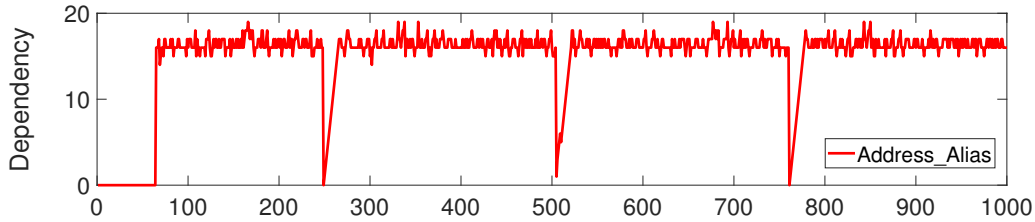
To find any relation between the observed high latency and a particular



(a) Step-wise peaks with a very high latency can be observed on some of the virtual pages



(b) Affected HPC event: `Cycle_Activity:Stalls_Ldm_Pending`



(c) Affected HPC event: `Ld_Blocks_Partial:Address_Alias`

Figure 3.1: SPOILER's timing measurements and hardware performance counters recorded simultaneously.

event, we compute correlation coefficients between counters and the timing measurements. Since the latency only occurs in the small region of the trace where the timing increases, we only need to compute the correlation on these regions. When an increase of at least 200 clock cycles is detected, the next s values from timing and the HPC traces are used to calculate the correlations, where s is the number of steps from Table 3.1 and 200 is the average execution time for a load.

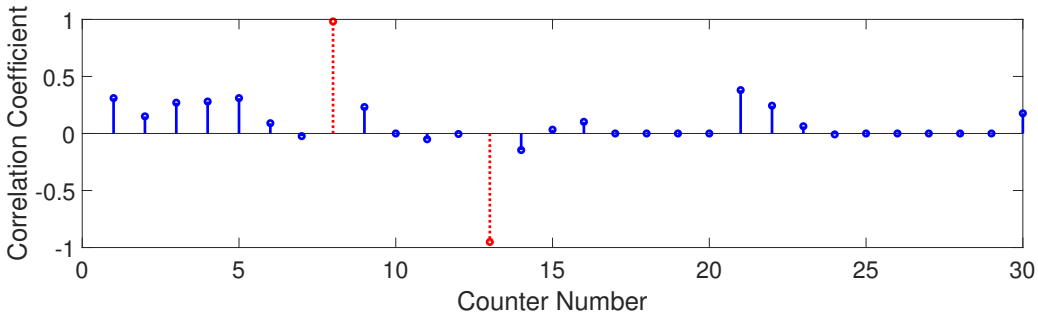


Figure 3.2: Correlation with HPCs listed in Table A.1 in the Appendix. `Ld_Blocks_Partial:Address_Alias` and `Cycle_Activity:Stalls_Ldm_Pending` (both dotted red) have strong positive and negative correlations, respectively.

As shown in Figure 3.2, two events have a high correlation with the leakage: `Cycle_Activity:Stalls_Ldm_Pending` has the highest correlation of 0.985. This event shows the number of cycles for which the execution is stalled and no instructions are executed due to a pending `load`. `Ld_Blocks_Partial:Address_Alias` has an inverse correlation with the leakage. This event counts the number of false dependencies in the MOB when loosenet resolves the 4K aliasing condition. Separately, `Exe_Activity:Bound_on_Stores` increases with more number of `stores` within the inner window loop in Algorithm 5, but it does not have a correlation with the leakage. The reason behind this behavior is that the store buffer is full, and additional store operations are pending. However, since there is no correlation with the leakage, this shows that the timing behavior is not due to the `stores` delay. We also attempt to profile any existing counters related to the memory disambiguation. However, the events

`Memory_Disambiguation.Success` and `Memory_Disambiguation.Reset` are not available on the modern architectures that are tested.

3.2.2 Leakage of the Physical Address Mapping

In this experiment, we evaluate whether the observed step-wise latency has any relationship with the physical page numbers by observing the `pagemap` file. As shown in Figure 3.3, we observe step-wise peaks with a very high latency which appear once in every 256 pages on average.The 20 least significant bits of physical address for the `load` matches with the physical addresses of the `stores` where high peaks for virtual pages are observed. In our experiments, we always detect peaks with different virtual addresses, which have the matching least 20 bits of physical address. This observation clearly discovers the existence of 1 MB aliasing effect based on the physical addresses. This 1 MB aliasing leaks information about 8 bits of mapping that were unknown to the user space processes.

Matching this observation with the previously observed `Cycle_Activity:Stalls_Ldm_Pending` with a high correlation, the speculative load has been stalled to resolve the dependency with conflicting store buffer entries after the occurrence of a 1 MB aliased address. This observation verifies that the latency is due to the pending `load`. When the latency is at the highest point, `Ld_Blocks_Partial:Address_Alias` drops to zero, and it increments at each down step of the peak. This implies that the loosenet check does not resolve the rest of the store dependencies whenever

CPU Model	Architecture	Steps	SB Size
Intel Core i7-8650U	Kaby Lake R	22	56
Intel Core i7-7700	Kaby Lake	22	56
Intel Core i5-6440HQ	Skylake	22	56
Intel Xeon E5-2640v3	Haswell	17	42
Intel Xeon E5-2670v2	Ivy Bridge EP	14	36
Intel Core i7-3770	Ivy Bridge	12	36
Intel Core i7-2670QM	Sandy Bridge	12	36
Intel Core i5-2400	Sandy Bridge	12	36
Intel Core i5 650	Nehalem	11	32
Intel Core2Duo T9400	Core	N/A	20
Qualcomm Kryo 280	ARMv8-A	N/A	*
AMD A6-4455M	Bulldozer	N/A	*

Table 3.1: 1 MB aliasing on various architectures: The tested AMD and ARM architectures, and Intel Core generation do not show similar effects. The Store Buffer (SB) sizes are gathered from Intel Manual [81] and *wikichip.org* [177, 179, 178].

there is a 1 MB aliased address in the store buffer.

3.2.3 Evaluation

In the previous experiment, the execution time of the `load` operation that is delayed by 1 MB aliasing decreases gradually in each iteration (Figure 3.3). The number of steps to reach the normal execution time is consistent on the same processor. When the first store in the window loop accesses a memory address with the matching 1 MB aliased address, the latency is at its highest point, marked as "1" in Figure 3.3. As the window loop accesses this address later in the loop, it appears closer to the `load` with a lower latency like the steps marked as 5, 15 and 22. This observation matches the *carry*

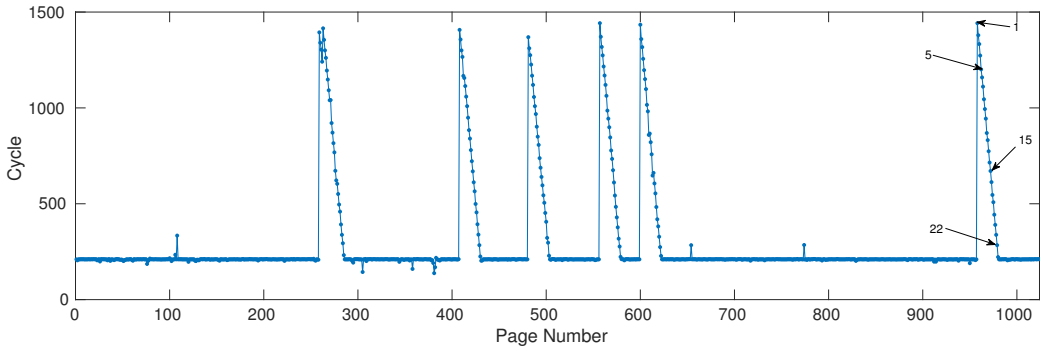


Figure 3.3: Step-wise peaks with 22 steps and a high latency can be observed on some of the pages (*Core i7-8650U* processor).

chain algorithm described by Intel [76] where the aliasing check starts from the most recent `store`. As shown in Table 3.1, experimenting with various processor generations shows that the number of steps has a linear correlation with the size of the store buffer which is architecture dependent. While the leakage exists on all Intel Core processors starting from the first generation, the timing effect is higher for the more recent generations with a bigger store buffer size. The analyzed ARM and AMD processors do not show similar behavior[1].

As our time measurement for speculative load suggests, it is not possible to reason whether the high timing is due to a very slow `load` or commitment of store operations. If the step-wise delay matches the store buffer entries, this delay may be either due to the dependency resolution logic performing a pipeline flush and restart of the `load` for each 4 kB aliased entry starting from the 1 MB aliased entry, or due to the `load` waiting for all the remaining

[1]We use `rdtscp` for Intel and AMD processors and the `clock_gettime` for ARM processors to perform the time measurements.

`stores` to commit because of an unresolved hazard. To explore this further, we perform an additional experiment with all store addresses replaced with non-aliased addresses except for one. This experiment shows that the peak disappears if there is only a single 4 kB and 1 MB aliased address in the store buffer.

Lastly, we run the same experiments on a shuffled set of virtual addresses to assure that the contiguous virtual addresses may not affect the observed leakage. Our experiment with the shuffled virtual addresses exactly matches the same step-wise behavior suggesting that the upper bits in virtual addresses do not affect the leakage behavior, and the leakage is solely due to the aliasing on physical address bits.

Comparison of Address Aliasing Scenarios

We further test other address combinations to compare additional address aliasing scenarios using Algorithm 5. As shown by Figure 3.4, when `stores` and the `load` access different cache sets without aliasing, the `load` is executed in 30 cycles, which is the typical timing for an L1 data cache load including the `rdtscp` overhead. When the `stores` have different memory addresses with the same page offset, but the `load` has a different offset, the `load` takes 100 cycles to execute. This shows that even memory addresses in the store buffer having 4K Aliasing conditions with each other that are totally unrelated to the speculative load create a memory bottleneck for the `load`. In the next scenario, 4K aliasing between the `load` and all `stores`, the average

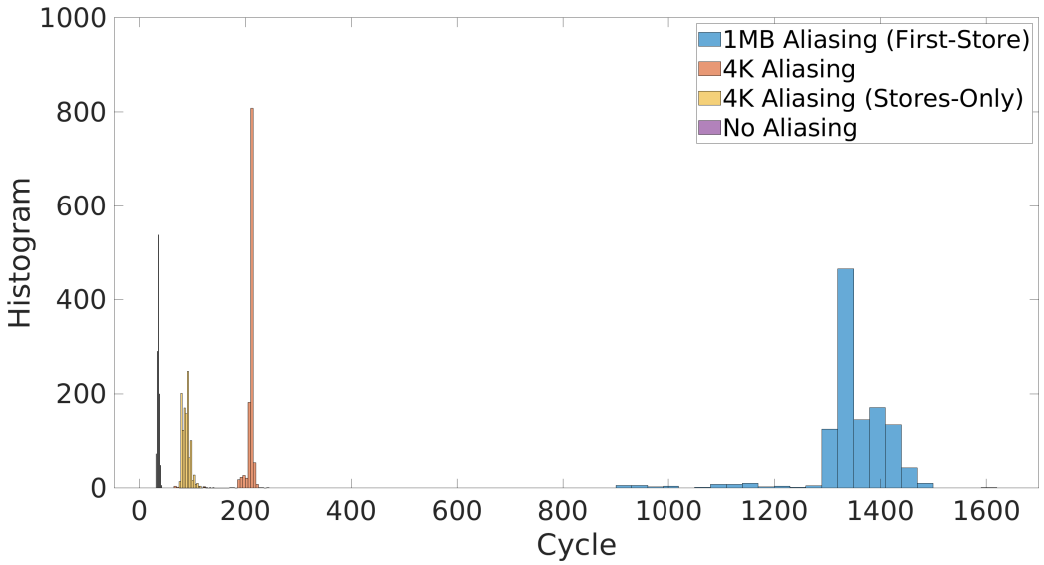


Figure 3.4: Histogram of the measurement for the speculative load with various store addresses. `Load` will be fast, 30 cycles, without any dependency. If there exists 4K aliasing only between the `stores`, the average is 100. The average is 200 when there is 4K aliasing of `load` and `stores`. The 1 MB aliasing has a distinctive high latency.

load time is about 200 cycles. While the aforementioned 4K aliasing scenarios may leak cross-domain information about memory accesses (Section 3.5), the most interesting scenario is the 1 MB aliasing which takes more than 1200 cycles for the highest point in the peak. For simplicity, we refer to the 1 MB an aliased address as *aliased address*, in the rest of the chapter.

3.2.4 Discussion

The Curious Case of Memory Disambiguation

The processor uses an additional speculative engine, called the *memory disambiguator* [48, 106], to predict memory false dependencies and reduce the chance of their occurrences. The main idea is to predict if a `load` is independent of preceding `stores` and proceed with the execution of the `load` by ignoring the store buffer. The predictor uses a hash table that is indexed with the address of the `load`, and each entry of the hash table has a saturating counter. If the pre-commitment dependency resolution does not detect false dependencies, the counter is incremented, otherwise, it will be reset to zero. After multiple successful executions of the same `load` instruction, the predictor assumes that the `load` is safe to execute. Every time the counter resets to zero, the next iteration of the `load` will be blocked to be checked against the store buffer entries. Mispredictions result in performance overhead due to pipeline flushes. To avoid repeated mispredictions, a watchdog mechanism monitors the success rate of the prediction, and it can temporarily disable the memory disambiguator.

The predictor of the memory disambiguator should go into a stable state after the first few iterations, since the memory `load` is always truly independent of any aliased store. Hence the saturating counter for the target speculative load address passes the threshold, and it never resets due to a false prediction. As a result, the memory disambiguator should always fetch

the data into the cache without any access to the store buffer. However, since the memory disambiguation performs speculation, the dependency resolution at some point verifies the prediction. The misprediction watchdog is also supposed to only disable the memory disambiguator when the misprediction rate is high, but in this case, we should have a high prediction rate. Accordingly, the observed leakage occurs after the disambiguation and during the last stages of dependency resolution, i.e., the memory disambiguator only performs prediction on the 4K aliasing at the initial loosenet check, and it cannot protect the pipeline from 1 MB aliasing that appears at a later stage.

Hyperthreading Effect

Similar to the 4K Aliasing [123, 161], we empirically test whether the 1 MB aliasing can be used as a covert/side-channel through logical processors. Our observation shows that when we run our experiments on two logical processors on the same physical core, the number of steps in the peaks is exactly halved. This matches the description by Intel [81] where it is stated that the store buffer is split between the logical processors. As a result, the 1 MB aliasing effect is not visible and exploitable across logical cores. [104] suggests that loosenet checks mask out the `stores` on the opposite thread.

3.3 SPOILER from JavaScript

Microarchitectural attacks from JavaScript have a high impact as drive-by attacks in the browser can be accomplished without any privilege or physical proximity. In such attacks, co-location is automatically granted by the fact that the browser loads a website with malicious embedded JavaScript code. The browsers provide a sandbox where some instructions like `clflush` and `prefetch` and file systems such as `procfs` are inaccessible, limiting the opportunity for attack. Genkin et al. [60] showed that side-channel attacks inside a browser can be performed more efficiently and with greater portability through the use of *WebAssembly*.Yet, WebAssembly introduces an additional abstraction layer, i.e. it emulates a 32-bit environment that translates the internal addresses to virtual addresses of the host process (the browser). WebAssembly only uses addresses of the emulated environment and similar to JavaScript, it does not have direct access to the virtual addresses. Using SPOILER from JavaScript opens the opportunity to puncture these abstraction layers and to obtain physical address information directly. Figure 3.5 shows the address search in JavaScript using SPOILER. Compared to native implementations, we replace the `rdtscp` measurement with a timer based on a shared array buffer [73]. We cannot use any fence instruction such as `lfence`, and as a result, there remains some negligible noise in the JavaScript implementation. However, the aliased addresses can still be clearly seen, and we can use this information to improve the state-of-the-art eviction set cre-

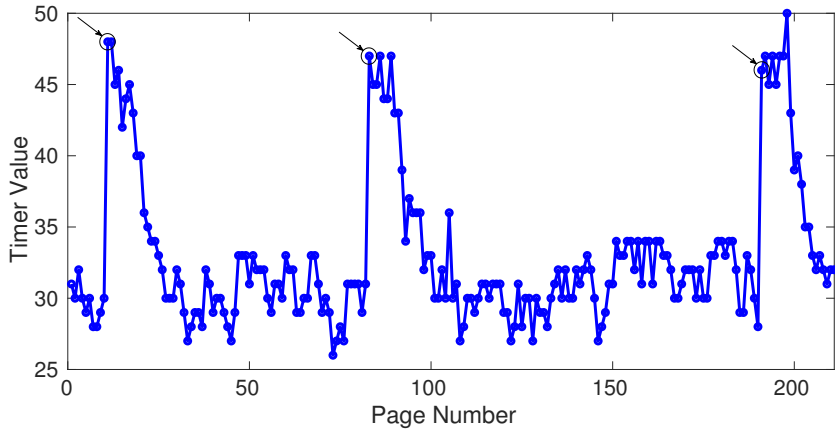


Figure 3.5: Reverse engineering physical page mappings in JavaScript. The markers point to addresses having same 20 bits of physical addresses being part of the same eviction set.

ation for both Rowhammer and cache attacks.

3.3.1 Efficient Eviction Set Finding

We use the algorithm proposed in [60]. It is a slight improvement to the former state-of-the-art brute force method[130] and consists of three phases:

- *expand*: A large pool of addresses P is allocated with the last twelve bits of all addresses being zero. A random address is picked as a witness t and tested against a candidate set C. If t is not evicted by C, it is added to C and a new witness will be picked. As soon as t gets evicted by C, C forms an eviction set for t.

- *contract*: Addresses are subsequently removed from the eviction set. If the set still evicts t, the next address is removed. If it does not evict

t anymore, the removed address is added back to the eviction set. At the end of this phase, we have a minimal eviction set of the size of the set associativity.

- *collect*: All addresses mapping to the already found eviction set are removed from P by testing if they are evicted by the found set. After finding 128 initial cache sets, this approach utilizes the linearity property of the cache: For each found eviction set, the bits 6-11 are enumerated instead. This provides 63 more eviction sets for each found set, leading to full cache coverage.

We test this approach on an Intel Core i7-4770 with four physical cores and a shared 8MB 16-way L3 cache with Chromium 68.0.3440.106, Firefox 62 and Firefox Developer Edition 63. The approach yields an 80% accuracy rate to find all 8192 eviction sets when starting with a pool of 4096 pages. The entire eviction set creation process takes an average of 46 s. We improve the algorithm by **1)** using the addresses removed from the eviction set in the contract phase as a new candidate set and **2)** removing more than one address at a time from the eviction set during the contract phase. The improved eviction set creation process takes 35 s on average.

Evaluation

The probability of finding a congruent address is $P(C) = 2^{\gamma-c-s}$, where c is the number of bits determining the cache set, γ is the number of bits

attackers know, and s is the number of slices[174]. Since SPOILER allows us to control $\gamma \geq c$ bits, we are only left with uncertainty about a few address bits that influence the slice selection algorithm [84]. In theory, the eviction set search is sped up by a factor of 4096 by using aliased addresses in the pool, since on average one of 2^8 instead of one of 2^{20} addresses is an aliased address. Additionally, the address pool is much smaller, where 115 addresses are enough to find all the eviction sets. In native code, the overhead involved in finding the aliased addresses is negligible, less than a second in our experiments. However, in JavaScript, due to the noise, it takes 9s for finding aliased addresses and then 3s for eviction set as compared to the baseline of 46s for classic method in Table 3.2. Success rate however is 100% with SPOILER as compared to 80% for the classic method. Besides, success rate of the classical method can be affected by the availability and consumption of memory on the system.

From each aliased address pool, 4 eviction sets can be found (corresponding to the 4 slices which are the only unknown part in the mapping). These can be enumerated again to form 63 more eviction sets since we still kept the bits 6-11 fixed. To accomplish full cache coverage, the aliased address pool has to be constructed 32 times. The SPOILER variant for finding eviction sets is more susceptible to system noise, which is why it needs more repetitions i.e. R rounds to get reliable values. On the other hand, it is less prone to values deviating largely from the mean, which is a problem in the classic eviction set creation algorithm. The classic method does not succeed

Algorithm	R	t_{total}	t_{AAS}	t_{ESS}	Success
Classic[130]	3	46s	-	100%	80%
Improved [60]	3	35s	-	100%	80%
AA (ours)	10	10s	54%	46%	67%
AA (ours)	20	12s	75%	25%	100%

Table 3.2: Comparison of different eviction set finding algorithms on an Intel Core i7-4770. Classic is the method from [130], improved is the same method with slight improvement, *Aliased Address (AA)* uses SPOILER. t_{AAS} is the time percentage used for finding aliased addresses. t_{ESS} is the time percentage for finding eviction sets. R is the number of Rounds.

about one out of five times in our experiments, as shown in Table 3.2. The unsuccessful attempts occur due to aborts if the algorithm takes much longer than statistically expected. As a result, SPOILER can be incorporated in an end-to-end attack such as drive-by key-extraction cache attacks by Genkin et al. [60]. SPOILER increases both speed and reliability of the eviction set finding and therefore the entire attack.

3.4 Rowhammer Attack using SPOILER

To perform a Rowhammer attack, the adversary needs to efficiently access DRAM rows adjacent to a victim row. In a single-sided Rowhammer attack, only one row is activated repeatedly to induce bit-flips on one of the nearby rows. For this purpose, the attacker needs to make sure that multiple virtual pages co-locate on the same bank. The probability of co-locating on the same bank is low without the knowledge of physical addresses and their mapping to memory banks. In a double-sided Rowhammer attack, the attacker tries

to access two different rows $n+1$ an $n-1$ to induce bit-flips in the row n placed between them. While double-sided Rowhammer attacks induce bit-flips faster due to the extra charge on the nearby cells of the victim row n, they further require access to contiguous memory pages. In this section, we show that SPOILER can help boost both single and double-sided Rowhammer attacks by its additional 8-bit physical address information and resulting detection of contiguous memory.

3.4.1 DRAM Bank Co-location

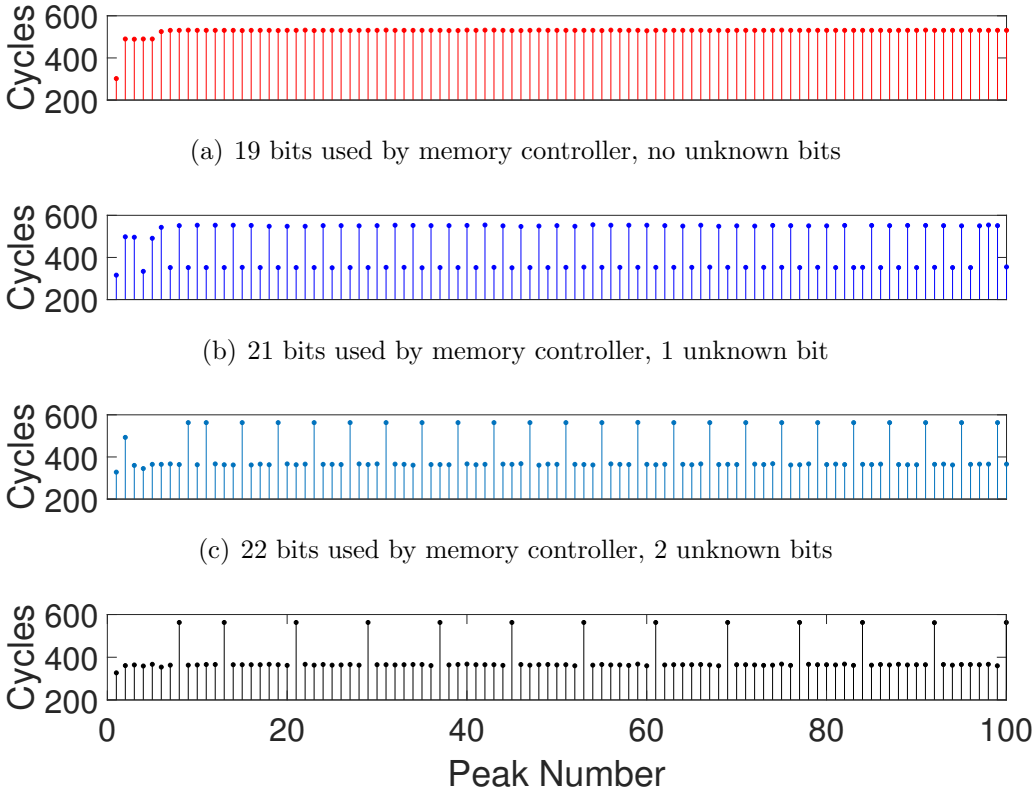
DRAMA [138] reverse engineered the memory controller mapping. This requires elevated privileges to access physical addresses from the `pagemap` file. The authors have suggested that prefetch side-channel attacks [66] may be used to gain physical address information instead. SPOILER is an alternative way to obtain partial address information and is still feasible when the `prefetch` instruction is not available, e.g. in JavaScript. In our approach, we use SPOILER to detect aliased virtual memory addresses where the 20 LSBs of the physical addresses match. The memory controller uses these bits for mapping the physical addresses to the DRAM banks [138]. Even though the memory controller may use additional bits, the majority of the bits are known using SPOILER. An attacker can directly hammer such aliased addresses to perform a more efficient single-sided Rowhammer attack with a significantly increased probability of hitting the same bank. As shown in Table 3.3, we reverse engineer the DRAM mappings for different hardware

System Model	DRAM Configuration	# of Bits
Dell XPS-L702x (Sandy Bridge)	1 x (4GB 2Rx8)	21
	2 x (4GB 2Rx8)	22
Dell Inspiron-580 (Nehalem)	1 x (2GB 2Rx8) **(b)**	21
	2 x (2GB 2Rx8) **(c)**	22
	4 x (2GB 2Rx8) **(d)**	23
Dell Optiplex-7010 (Ivy Bridge)	1 x (2GB 1Rx8) **(a)**	19
	2 x (2GB 1Rx8)	20
	1 x (4GB 2Rx8) **(e)**	21
	2 x (4GB 2Rx8)	22

Table 3.3: Reverse engineering the DRAM memory mappings using DRAMA tool, *# of Bits* represents the number of physical address bits used for the bank, rank and channel [138].

configurations using the DRAMA tool, and only a few bits of physical address entropy beyond the 20 bits will remain unknown.

To verify if our aliased virtual addresses co-locate on the same bank, we use the row conflict side-channel as proposed in [56] (timings in the Appendix, Section A.2). We observe that whenever the number of physical address bits used by the memory controller to map data to physical memory is equal to or less than 20, we always hit the same bank. For each additional bit the memory controller uses, the probability of hitting the same bank is divided by 2 as there is one more bit of entropy. In general, we can formulate that our probability p to hit the same bank is $p = 1/2^n$, where n is the number of unknown physical address bits in the mapping. We experimentally verify the success rate for the setups listed in Table 3.3, as depicted in Figure 3.6. In summary, SPOILER drastically improves the efficiency of finding



(a) 19 bits used by memory controller, no unknown bits

(b) 21 bits used by memory controller, 1 unknown bit

(c) 22 bits used by memory controller, 2 unknown bits

(d) 23 bits used by memory controller, 3 unknown bits

Figure 3.6: Bank co-location for various DRAM configurations (a), (b), (c) & (d) from Table 3.3. The regularity of the peaks shows that the allocated memory was contiguous, which is coincidental.

addresses mapping to the same bank without administrative privilege or reverse engineering the memory controller mapping.

3.4.2 Contiguous Memory

For a double-sided Rowhammer attack, we need to hammer rows adjacent to the victim row in the same bank. This requires detecting contiguous mem-

ory pages in the allocated memory, since the rows are written to the banks sequentially. Without contiguous memory, the banks will be filled randomly and we will not be able to locate neighboring rows. We show that an attacker can use SPOILER to detect contiguous memory using 1 MB aliasing peaks. For this purpose, we compare the physical frame numbers to the SPOILER leakage for 10000 different virtual pages allocated using `malloc`. Figure 3.7 shows the relation between 1 MB aliasing peaks and physical page frame numbers. When the distance between the peaks is random, the trend of frame numbers also changes randomly. After around 5000 pages, we observe that the frame numbers increase sequentially. The number of pages between the peaks remains constant at 256 where this distance comes from the 8 bits of physical address leakage due to 1 MB aliasing.

We also compare the accuracy of obtaining contiguous memory detected by SPOILER, by analyzing the actual physical addresses from the `pagemap` file. By checking the difference between physical page numbers for each detected virtual page, we can determine the accuracy of our detection method: the success rate for finding contiguous memory is above 99% disregarding the availability of the contiguous pages. For detailed experiment on the availability of the contiguous pages, see Section A.3 in the Appendix.

3.4.3 Double-Sided Rowhammer with SPOILER

As double-sided Rowhammer attacks are based on the assumption that rows within a bank are contiguous, we mount a practical double-sided Rowham-

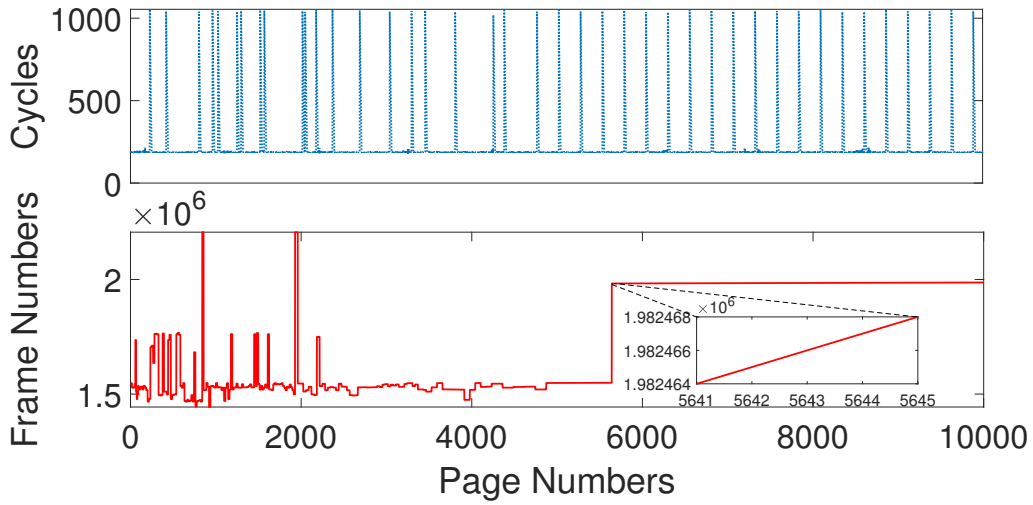


Figure 3.7: Relation between leakage peaks and the physical page numbers. The dotted plot shows the leakage peaks from SPOILER. The solid plot shows the decimal values of the physical frame numbers from the `pagemap` file. Once the peaks in the dotted plot become regular, the solid plot is linearly increasing, which shows contiguous memory allocation.

mer attack on several DRAM modules using SPOILER without any root privileges. First, we use SPOILER to detect a suitable amount of contiguous memory. If enough contiguous memory is available in the system, SPOILER finds it, otherwise a double-sided Rowhammer attack is not feasible. In our experiments, we empirically configure SPOILER to detect 10 MB of contiguous memory. Second, we apply the row conflict side-channel only to the located contiguous memory, and get a list of virtual addresses which are contiguously mapped within a bank. Finally, we start performing a double-sided Rowhammer attack by selecting 3 consecutive addresses from our list. While we have demonstrated the bit-flips in our own process, we can free that memory which can then be assigned to a victim process by using previously

DRAM Model	Architecture	Flippy
M378B5273DH0-CK0	Ivy Bridge	✓
M378B5273DH0-CK0	Sandy Bridge	✓
M378B5773DH0-CH9	Sandy Bridge	✓
M378B5173EB0-CK0	Sandy Bridge	×
NT2GC64B88G0NF-CG	Sandy Bridge	×
KY996D-ELD	Sandy Bridge	×
M378B5773DH0-CH9	Nehalem	✓
NT4GC64B8HG0NS-CG	Sandy Bridge	×
HMA41GS6AFR8N-TF	Skylake	×

Table 3.4: DRAM modules susceptible to double-sided Rowhammer attack using SPOILER

known techniques like spraying and memory waylaying [65]. As the bit-flips are highly reproducible, we can again flip the same bits in the victim process to demonstrate a full attack. Table 3.4 shows some of the DRAM modules susceptible to Rowhammer attack.

The native version of Rowhammer is also applicable in JavaScript. The JavaScript-only variant implementation of Rowhammer by Gruss et al. [67], named rowhammer.js[2], can be combined with SPOILER to implement an end-to-end attack. In the original rowhammer.js, 2MB HugePages were assumed to get a contiguous chunk of physical memory. With SPOILER, this assumption is no longer required as explained in Section 3.4.3.

Figure 3.8 shows the number of hammers compared to the amount of bit-flips for configuration (e) in Table 3.3. We repeat this experiment 30 times for every measurement and the results are then averaged out. On every

[2]`https://github.com/IAIK/rowhammerjs`

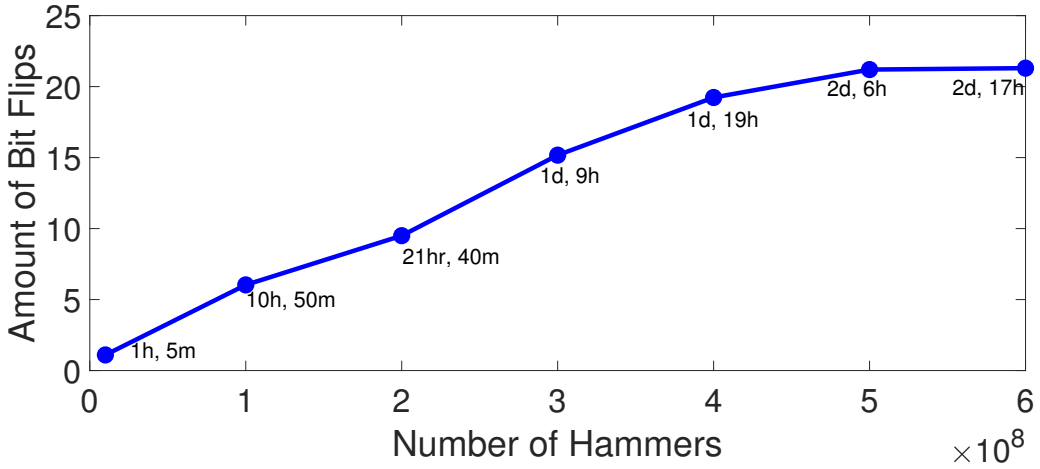


Figure 3.8: Amount of bit-flips increases with the increase in number of hammerings. The timings do not include the time taken for reboots and 1 minute sleep time.

experiment, the system is rebooted using a script because once the memory becomes fragmented, no more contiguous memory is available. The number of bit-flips increases with more number of hammerings. Hammering for 500 million times is found to be an optimal number for this DRAM configuration, as the continuation of hammering is not increasing bit-flips.

3.5 Tracking Speculative Loads With SPOILER

Single-threaded attacks can be used to steal information from other security contexts running before/after the attacker code on the same thread [33, 124]. Example scenarios are I) context switches between processes of different

users, or II) between a user process and a kernel thread, and III) Intel Software Guard eXtensions (SGX) secure enclaves [124, 172]. In such attacks, the adversary puts the microarchitecture to a particular state, waits for the context switch and execution of the victim thread, and then tries to observe the microarchitectural state after the victim's execution. We propose an attack where the adversary **1)** fills the store buffer with arbitrary addresses, **2)** issues the victim context switch and lets the victim perform a secret-dependent memory access, and **3)** measures the execution time of the victim. Any correlation between the victim's timing and the load address can leak secrets [183]. Due to the nature of SPOILER, the victim should access the memory while there are aliased addresses in the store buffer, i.e. if the `stores` are committed before the victim's speculative load, there will be no dependency resolution hazard.

We first perform an analysis of the depth of the operations that can be executed between the `stores` and the `load` to investigate the viability of SPOILER. In this experiment, we repeat a number of instructions between `stores` and the `load` that are free from memory operations. Figure 3.9 shows the number of stall steps due to the dependency hazard with the added instructions. Although `nop` is not supposed to take any cycle, adding 4000 `nop` will diffuse the timing latency. Then, we test `add` and `leal`, which use the Arithmetic Logic Unit (ALU) and the Address Generation Unit (AGU), respectively. Figure 3.9 shows that only 1000 `adds` can be executed between the `stores` and `load` before the SPOILER effect is lost. Since each `add`

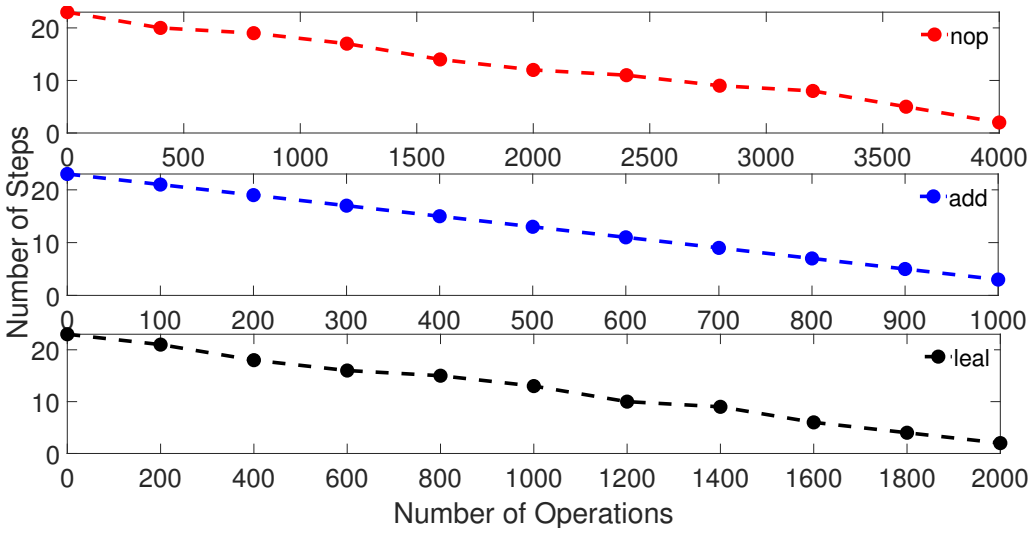


Figure 3.9: The depth of SPOILER leakage with respect to different instructions and execution units.

typically takes about 1 cycle to execute, this roughly gives a 1000 cycle depth for SPOILER. Considering the observed depth, we discuss potential attacks that can track the speculative load in the following two scenarios.

3.5.1 SPOILER Context Switch

In this attack, we are interested in tracking a memory access in the privileged kernel environment after a context switch. First, we fill the store buffer with addresses that have the same page offset, and then execute a system call. During the execution of the system call, we expect to observe a delayed execution if a secret `load` address has aliasing with the `stores`. We utilize SPOILER to iterate over various virtual pages, thus some of the pages have more noticeable latency due to the 1 MB aliasing. We analyze

multiple `syscalls` with various execution times. For instance, Figure 3.10 shows the execution time for `mincore`. In the first experiment (red/1 MB Conflict), we fill the store buffer with addresses that have aliasing with a memory `load` operation in the kernel code space. The 1 MB aliasing delay with 7 steps suggests that we can track the address of a kernel memory `load` by the knowledge of our arbitrary filled store addresses. The blue (No Conflict) line shows the timing when there is no aliasing between the target memory `load` and the attackers `store`. Surprisingly, only by filling the store buffer, the system call executes much slower: the normal execution time for `mincore` should be around 250 cycles (cyan/No Store). This proof of concept shows that SPOILER can be used to leak information from more privileged contexts, however, this is limited only to `loads` that appear at the beginning of the next context.

3.5.2 Negative Result: SPOILER SGX

In this experiment, we try to combine SPOILER with the *CacheZoom* [124] approach to create a novel single-threaded side-channel attack against SGX enclaves with high temporal and spatial resolution (4-Byte) [123]. We use *SGX-STEP* [171] to precisely interrupt every single instruction. *Nemesis* [172] shows that the interrupt handler context switch time is dependent on the execution time of the currently running instruction. On our test platform, *Core i7-8650U*, each context switch on an enclave takes about 12000 cycles to execute. If we fill the store buffer with memory addresses that match the

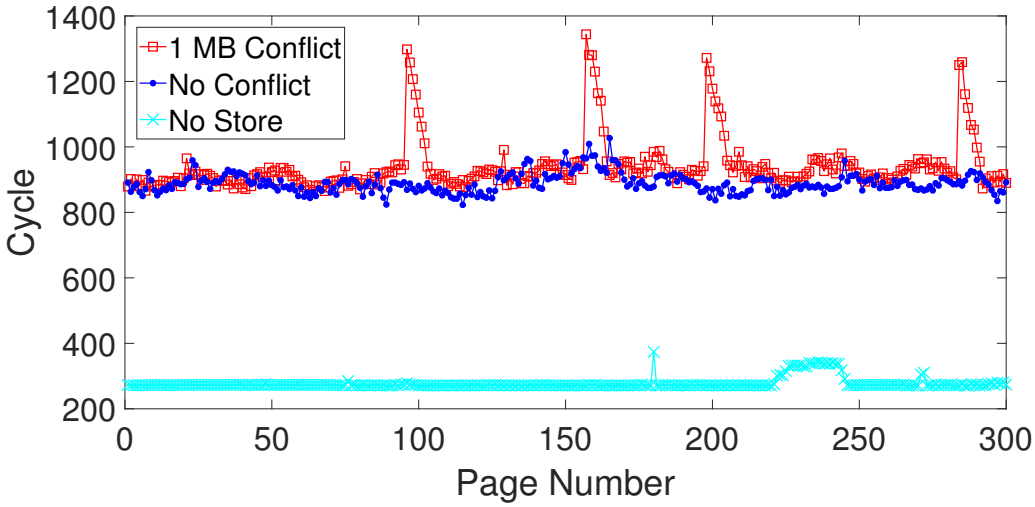


Figure 3.10: Execution time of `mincore` system call. When a kernel `load` address has aliasing with the attacker's `stores` (red/1MB Conflict), the step-wise delay will appear. These timings are measured with Kernel Page Table Isolation disabled.

page offset of a `load` inside the enclave in the interrupt handler, the context switch timing is increased to about 13500 cycles. While we cannot observe any correlation between the matched 4 kB or 1 MB aliased addresses, we do see unexpected periodic downward peaks with a similar step-wise behavior as SPOILER (Figure 3.11). We later reproduce a similar behavior by running SPOILER before an `ioctl` routine that flushes the TLB on each call. Intel SGX also performs an implicit TLB flush during each context switch. We can thus infer that the downward peaks occur due to the TLB flush, especially since the addresses for the downward peaks do not have any address correlation with the `load` address. This suggests that the TLB flush operation itself is affected by SPOILER. This effect eliminates the opportunity to

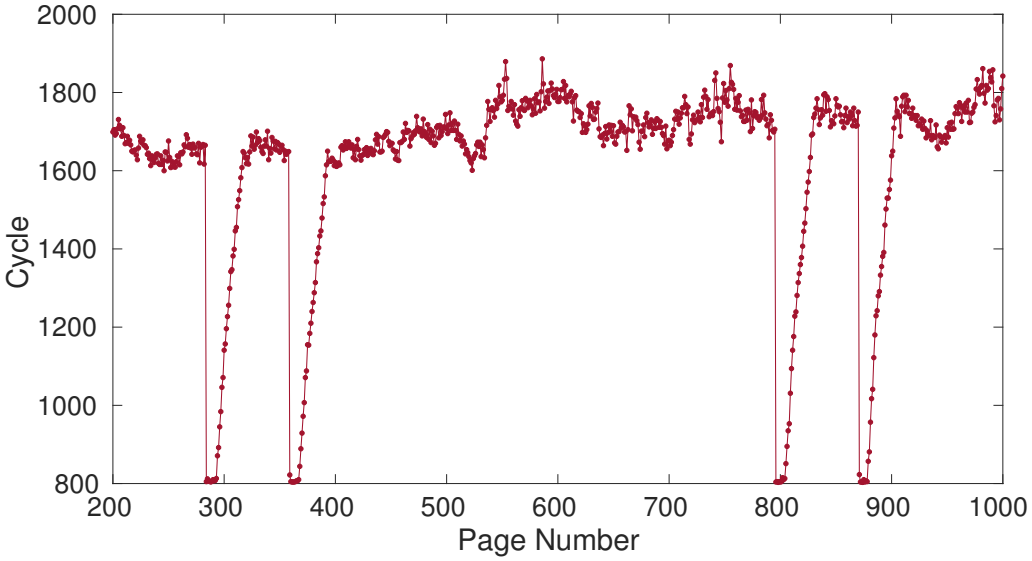


Figure 3.11: The effect of SPOILER on TLB flush. The execution cycle always increases for 4 kB aliased addresses, except for some of the virtual pages inside in the store buffer where we observe step-wise hills.

observe any potential correlation due to the speculative load. As a result, we can not use SPOILER to track memory accesses inside an enclave. Further exploration of the root cause of the TLB flush effect can be carried out as future work.

3.6 Mitigations

3.6.1 Software Mitigations

The attack exploits the fact that when there is a `load` instruction after a number of `store` instructions, the physical address conflict causes a high timing behavior. This happens because of the speculatively executed `load`

before all the `stores` are finished executing. There is no software mitigation that can completely erase this problem. While the timing behavior can be removed by inserting store fences between the `loads` and `stores`, this cannot be enforced to the user's code space, i.e., the user can always leak the physical address information. Another yet less robust approach is to execute other instructions between the `loads` and `stores` to decrease the depth of the attack. However, both of the approaches are only applicable to defend against attacks such as the one described in Section 3.5.

As for most attacks on JavaScript, removing accurate timers from the browser would be effective against SPOILER. Indeed, some timers have been removed or distorted by jitters as a response to attacks [114]. There is however a wide range of timers with varying precision available, and removing all of them seems impractical [150, 56].

When it is not possible to mitigate the microarchitectural attacks, developers can use dynamic tools to at least detect the presence of such leakage [35, 186, 26]. One of the dynamic approaches is gained by monitoring hardware performance counters in real-time. As explained in Section 3.2.1, two of the counters `Ld_Blocks_Partial:Address_Alias` and `Cycle_Activity:Stalls_Ldm_Pending` have high correlations with the leakage.

3.6.2 Hardware Mitigations

The hardware design for the memory disambiguator may be revised to prevent such physical address leakage, but modifying the speculative behavior may cause performance impacts. For instance, partial address comparison was a design choice for performance. Full address comparison may address this vulnerability, but will also impact performance. Moreover, hardware patches are difficult to be applied to legacy systems and take years to be deployed.

3.7 Conclusion

In this chapter, we introduced SPOILER, a novel approach for gaining physical address information by exploiting a new information leakage due to speculative execution. To exploit the leakage, we used the speculative load behavior after jamming the store buffer. SPOILER can be executed from user space and requires no special privileges. We exploited the leakage to reveal information on the 8 least significant bits of the physical page number, which are critical for many microarchitectural attacks such as Rowhammer and cache attacks. We analyzed the causes of the discovered leakage in detail and showed how to exploit it to extract physical address information.

Further, we showed the impact of SPOILER by performing a highly targeted Rowhammer attack in a native user-level environment. We further demonstrated the applicability of SPOILER in sandboxed environments by

constructing efficient eviction sets from JavaScript, an extremely restrictive environment that usually does not grant any access to physical addresses. Gaining even partial knowledge of the physical address will make new attack targets feasible in browsers even though JavaScript-enabled attacks are known to be difficult to realize in practice due to the limited nature of the JavaScript environment. Broadly put, the leakage described in this chapter will enable attackers to perform existing attacks more efficiently, or to devise new attacks using the novel knowledge. The source code for SPOILER is available on GitHub[3].

Responsible Disclosure We informed the *Intel Product Security Incident Response Team* (iPSIRT) of our findings. iPSIRT thanked for reporting the issue and for the coordinated disclosure. iPSIRT then released the public advisory and CVE. Here is the time line for the responsible disclosure:

- **12/01/2018:** We informed our findings to iPSIRT.
- **12/03/2018:** iPSIRT acknowledged the receipt.
- **04/09/2019:** iPSIRT released public advisory (INTEL-SA-00238) and assigned CVE (CVE-2019-0162).

[3] `https://github.com/saadislamm/SPOILER`
`https://github.com/UzL-ITS/Spoiler`

Chapter 4

QuantumHammer: A Practical Hybrid Attack on the LUOV Signature Scheme

Multivariate signatures is one of the main categories in NIST's post-quantum cryptography competition. Among the four round 2 candidates in this category, the LUOV and Rainbow schemes are based on the Oil and Vinegar scheme, first introduced in 1997 which has withstood over two decades of cryptanalysis. Beyond mathematical security and efficiency, security against side-channel attacks is a major concern in the competition. The current sentiment is that post-quantum schemes may be more resistant to fault-injection attacks due to their large key sizes and the lack of algebraic structure. We show that this is not true.

We introduce a novel hybrid attack, QuantumHammer, and demonstrate it on the constant-time implementation of LUOV (round 2 finalist). The QuantumHammer attack is a combination of two attacks, a bit-tracing attack enabled via Rowhammer fault injection and a divide and conquer attack that uses bit-tracing as an oracle. Using bit-tracing, an attacker with access to faulty signatures collected using Rowhammer attack, can recover secret key bits albeit slowly. We employ a divide and conquer attack which exploits the structure in the key generation part of LUOV and solves the system of equations for the secret key more efficiently with few key bits recovered via bit-tracing.

We have demonstrated the first successful in-the-wild attack on LUOV recovering all 11K key bits with less than 4 hours of an active Rowhammer attack. The post-processing part is highly parallel and thus can be trivially sped up using modest resources. QuantumHammer does not make any unrealistic assumptions, only requires software co-location (no physical access), and therefore can be used to target shared cloud servers or in other sandboxed environments.

4.1 Contributions

We have discovered a *practical* technique that recovers all secret key bits in LUOV. QuantumHammer proceeds by injecting faults, collecting faulty signatures, followed by the divide and conquer attack. The faults are achieved

using a realistic software only approach via a Rowhammer attack. In summary, in this chapter:

1. We introduce a simple technique that uses faulty signatures to mathematically trace and recover key bits. Each faulty signature yields a key bit. While not efficient, the technique gives us a tool we then amplify the efficiency of our attack using an analytical approach.

2. The analytical attack exploits structures in the generation of the public key using a small number of recovered key bits (using a modest number of faults injections), the complexity of attacking the overall multivariate system is reduced to a number of much smaller MV problems, which are tractable with modest resources using brute force.

3. Our attack is software only, i.e. we do not assume any physical access to the device. This also permits remote attacks on shared cloud servers or in browsers. We assume that the memory module is susceptible to Rowhammer and that faulty signatures can be recovered.

4. Earlier fault attacks on post-quantum schemes assumed hypothetical faults. We present a successful end-to-end Rowhammer attack on *constant-time* AVX2 optimized implementation of the multivariate post-quantum signature scheme LUOV.

5. We have demonstrated full key recovery of 11,229 bits for LUOV-7-57-197 in less than 4 hours of online Rowhammer attack and 49 hours of

offline post-processing.

6. This attack is applicable to all the variants of LUOV Scheme including the updates [175] after Ding et al. attack [47].

4.2 Related Work

In 2019, Ding et al. [47, 44] presented a (purely) algebraic attack on LUOV, i.e. the subfield differential attack. Without any side-channel information, the attack managed to significantly reduce the security level of LUOV. Specifically, for LUOV-8-58-237, the complexity is reduced from 2^{146} to 2^{105} which is lower than the minimum security level criteria established by NIST for the post-quantum competition. The updated version of LUOV now uses finite fields $GF(2^r)$, where r is a prime, which renders the subfield differential attack inapplicable[1].

On Rainbow-like schemes, Ding et al. [46] introduced an algebraic Reconciliation attack as an early work in 2008. Afterward, as for fault attacks on multivariate schemes, only a few results exist: In 2011 by Hashimoto et al. [74] on Big Field type and Stepwise Triangular System (STS) including UOV and Rainbow. In 2019, Kramer et al. [105] have also worked on UOV and Rainbow extending the earlier work. We will only talk about UOV and Rainbow in this section and not the Big Field type schemes. Reconciliation is an algebraic attack whereas the other two works assume physical fault at-

[1]The updated version is available at the author's website [175].

tacks, first introduced by Boneh et al. [20] but there are no details on fault injection technique. Kramer et al. claimed that randomness of vinegar variables and also the layers in Rainbow provide good protection against fault attacks. These studies consider three attack scenarios:

Scenario 1 (Algebraic Attack) In this scenario [46], a purely algebraic attack improves on brute force but does not assume any physical fault or any side-channel information. Specifically, the aim is to invert the public map $\mathcal{P}$ by finding a sequence of change of basis matrices. $\mathcal{P}$ is decomposed into a series of linear transformations which are recovered step by step which significantly reduces the security level.

Scenario 2 (Central Map) It assumes that a coefficient of the secret quadratic central map $\mathcal{F}$ has been faulted. By signing randomly chosen messages with the faulty $\mathcal{F}'$ and verifying the signatures with the correct public key $\mathcal{P}$, partial information about the secret linear transformation matrix $\mathcal{S}$ can be recovered using $\delta = \mathcal{S} \circ (\mathcal{F}' - \mathcal{F}) \circ \mathcal{T}$, where $\mathcal{T}$ is another secret linear transformation matrix. As $(\mathcal{F}' - \mathcal{F})$ is sparse, $\mathcal{S}$ can be partially recovered. At least $m-1$ faults are required to recover some part of the secret key matrix $\mathcal{S}$, where m is the number of equations in the system. Both [74] and [105] have an assumption that the attack can induce faults in either $\mathcal{S}$, $\mathcal{F}$ or $\mathcal{T}$ and provided the success probabilities of hitting the central map $\mathcal{F}$. Kramer et al. have additionally assumed a stronger attacker who can directly attack $\mathcal{F}$ or even specific coefficients of $\mathcal{F}$ to avoid unwanted scenarios. Kramer et al. [105] refute a claim made earlier by Hashimoto et al. [74] and claim that

UOV is immune to the fault attack on the central map. It is because the attack is recovering part of $\mathcal{S}$ and not $\mathcal{T}$, which is not present in the UOV scheme.

Scenario 3 (Fixed Vinegar) This scenario assumes that the attacker is able to fix part of randomly chosen vinegar variables from $(x_{v-u+1}, \ldots, x_v)$, where u is the number of vinegar variables fixed out of total v vinegar variables during multiple signature computation sessions. After that, message/signature pairs are generated and utilized to recover the secrets. $n-u+1$ pairs are needed to recover part of $\mathcal{T}$. As the attack recovers partial information about $\mathcal{T}$, it is applicable to both the UOV and Rainbow schemes but still not sufficient to recover the secret key.

Shim et al. [153] have presented an algebraic fault analysis attack on the UOV and Rainbow schemes. They have assumed a similar scenario of fixed (reused) vinegar but they have two more scenarios as well: revealed and set to zero vinegar. They are also assuming physical faults and there are no details on how the faults are injected. Based upon the number of faulty vinegar values, they give the complexities of the attacks. For UOV, 59 Bytes of faulty vinegar are needed for full key recovery. They also provide the results for LUOV which are the only fault attack results so far on LUOV scheme. Due to the large parameter sizes, the results are not very promising to obtain a practical attack to target real life deployments. Assuming 171 Bytes and 169 Bytes of faulty vinegar values for LUOV-8-63-256 and LUOV-8-49-242, the complexities drop from 2^{181} and 2^{192} to 2^{127} and 2^{109}, respectively.

The authors have not demonstrated the fault attack. In practice, fixing a large contiguous portion of vinegar values by physical fault injection or Rowhammer is very hard to achieve if at all possible.

Our attack scenario is different from those presented in existing works [46, 74, 105, 20]. We are inducing faults in the last stage of the signing algorithm in the linear transformation $\mathcal{T}$ of LUOV scheme. We have actually verified the assumption, i.e., we implemented an attack that induces bit-flips in $\mathcal{T}$. Note that the attack does not have any control in the position of the bit-flips within $\mathcal{T}$ as assumed by our attack scenario. Also, we have the ability to detect if the bit-flip was in $\mathcal{T}$ or not. We have practically demonstrated this model by inducing the bit-flips using the Rowhammer attack and not just assuming the faults as in previous research. To the best of our knowledge, this is the first work that actually induces bit-flips (faults) through software in post-quantum cryptographic schemes. The goal here is to make use of the faulty signatures to track back to the flipped bits and leak the secret bits of $\mathcal{T}$. We do not need any correct and faulty signature pairs. Rather we are able to correct the faulty signature by modifying the public signature values and verifying the modified signatures using signature verification mechanism as an oracle. Some recovered bits from this bit-tracing attack are used to decrease the complexity of the solution of Multivariate Quadratic (MQ) system to a practically solvable smaller MQ and Multivariate Linear (ML) systems by using a divide and conquer attack to recover the rest of the private key bits. We call this hybrid attack as QuantumHammer.

4.3 A Novel Bit-Tracing Attack on LUOV

In this section we outline a novel fault injection attack on LUOV. The attack succeeds in efficiently recovering secret key bits from faulty signatures whereas faults may be injected through software only Rowhammer attack. The attack consists of three main phases, pre-processing, online and post-processing phase.

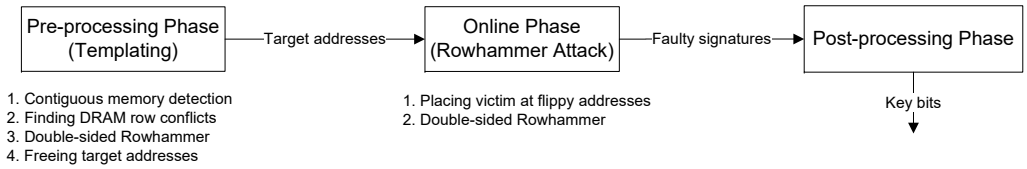


Figure 4.1: Phases of novel bit-tracing attack on LUOV

The pre-processing phase which includes templating, needs to be carried out on the same machine on which the victim will be running. The purpose of this phase is to collect the physical addresses of the memory locations susceptible to Rowhammer. The victim does not need to be present or running in the pre-processing phase. The victim can then be placed at those addresses in the online phase when the victim process starts running. In the online phase, the victim is first forced to be placed at the target addresses and then the Rowhammer attack induces bit-flips in a particular area of the victim while the victim is carrying out the signing operations. This causes the victim to generate faulty signatures which are public and collected by the attacker. After collecting a number of faulty signatures, our novel bit tracing algorithm is carried out in the post-processing phase which can be

done offline on any other machine or cluster.

The DRAM modules installed in the system are susceptible to Rowhammer attack. The attacker and victim processes are co-located on the same DRAM chip. The attacker can induce bit-flips in the linear transformation $\mathcal{T}$ of LUOV scheme and is able to collect the faulty signatures. The attacker has no control or knowledge over the position of the bit-flips within $\mathcal{T}$ and the $\mathcal{T}$ matrix is huge e.g. 11,229 bits for LUOV-7-57-197 [175]. Also, the attacker does not know the value of the flipped bit. The target of the attacker is to trace back to the position of the flipped bit as well as to recover the value of the bit by just using the faulty signatures. The attacker has no knowledge of the correct signatures and can only use the public parameters to perform the attack. Moreover, the attacker is not using HugePages for contiguous memory. Also, she does not have any knowledge of the DRAM mappings which convert physical addresses to DRAM ranks, banks, rows and columns. The bit-tracing is summarized in Figure 4.1 and then each step is explained in detail along-with results.

4.3.1 Pre-processing Phase (Templating)

The pre-processing (templating) phase of the attack is carried out on the machine where the attacker and the victim are co-located, sharing the same DRAM module. The victim does not need to be present or running in this phase. As double-sided Rowhammer requires contiguous chunk of physical memory, we allocate a 256 MB buffer and look for an 8 MB of contiguous

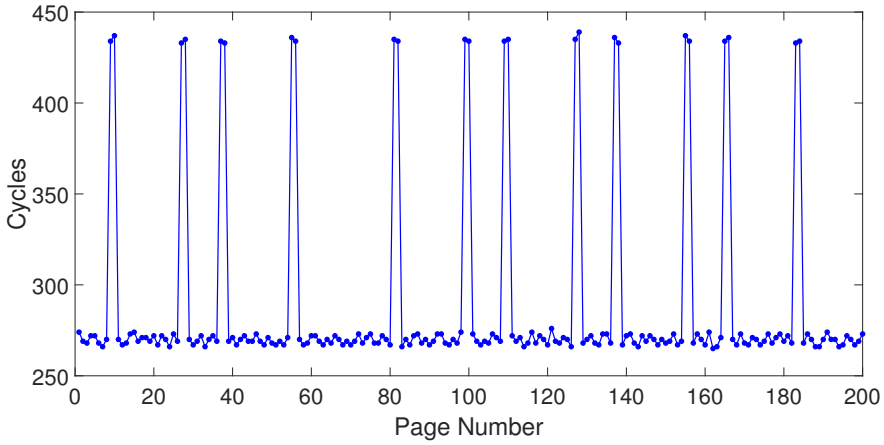


Figure 4.2: Row conflicts for the pages from the detected contiguous memory. The higher timings indicate that the pages are mapped to the same DRAM bank which are the target for the Rowhammer attack.

memory using SPOILER [86]. After that, row conflicts are found to identify the virtual addresses mapped to the same DRAM bank. This is achieved using a side-channel since the data coming from the same bank will take longer as compared to the data coming from the other banks. As the data from the row buffer needs to be copied back to the original row before the data from another row within the same bank is loaded into the row buffer, it creates additional delay. The measurements are shown in Figure 4.2 and a threshold value of 380 cycles is set in our experiments. This threshold value may vary from one machine to another.

Once we find the virtual addresses mapped to the same DRAM bank, we start the process of double-sided Rowhammer to find the DRAM rows suitable for Rowhammer. We found 125 rows in 8 MB of contiguous memory which are mapped to the same bank. Our results indicate that the rows in

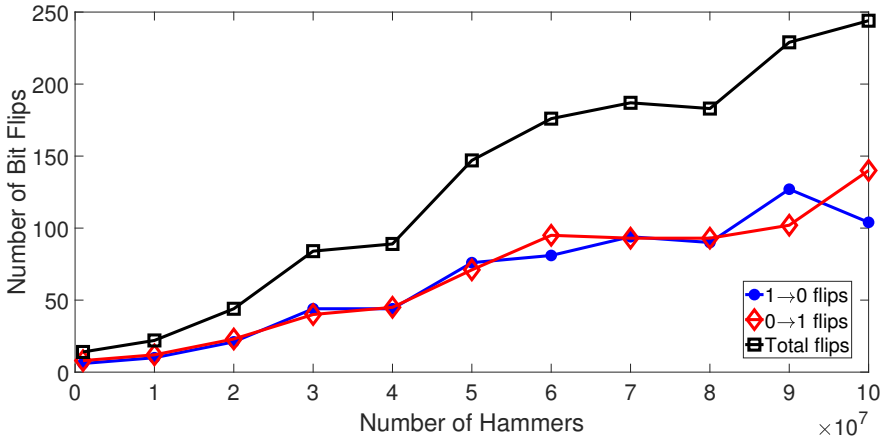


Figure 4.3: Number of bit-flips increases with the increase in number of hammers. The experiment is repeated 30 times for each number of hammers on an 8 MB contiguous chunk of memory and the results are then averaged out.

the DRAM are ordered sequentially if the targeted memory is contiguous. These rows are then taken 3 at a time with aggressor rows on the sides and the victim row in the middle and aggressor rows are accessed (hammered) repeatedly to get flips in the victim row. The number of bit-flips found within this contiguous chunk can be seen in Figure 4.3 against the number of hammers. It is observed that the number of bit-flips increases with the number of hammers. To find the susceptible memory locations in the pre-processing phase we set a value for number of hammers as 10^6. The other observation is that there is not much difference between the number of $1 \rightarrow 0$ flips and $0 \rightarrow 1$ flips. To achieve bidirectional flips, we fill the aggressor rows with all zeros and the victim row with all ones for $1 \rightarrow 0$ flips and aggressor rows with all ones and the victim with all zeros for $0 \rightarrow 1$ flips as explained

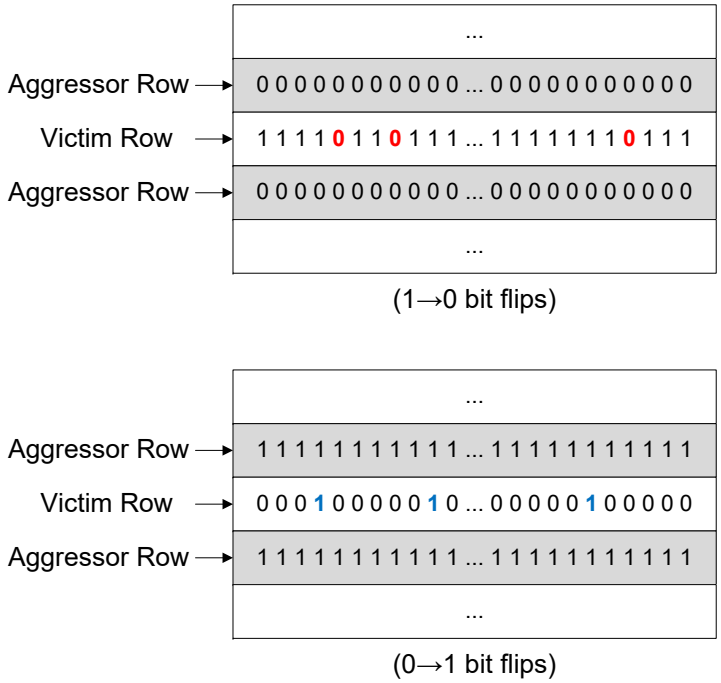


Figure 4.4: Double-sided Rowhammer with different data patterns. If the attacker rows are filled with all zeros, the bit-flips occur in $1 \rightarrow 0$ direction and if the attacker rows are filled with all ones, the bits are flipped from $0 \rightarrow 1$. This strategy helps to recover the values of the bit positions of $\mathcal{T}$ traced by the bit-tracing attack.

in Figure 4.4.

The final step of the pre-processing phase is to free the vulnerable memory pages from the attacker process so that the victim can be placed at that location for the online attack. We do this by using *munmap* instruction for every 8 kB row. As the bit-flips are highly reproducible, Rowhammer will flip the same bits again but in the victim process in the online phase.

The experiments are carried out on a Haswell system with DDR3 memory, running Ubuntu OS. 17,129 vulnerable physical addresses are found in

5.7 hours. These experiments are done repeatedly using a script as 8 MB of contiguous memory is not enough for gathering these many addresses. So, 256 MB of memory is allocated again and again out of which 8 MB of contiguous chunk is detected. Each chunk is then checked for all possible bit-flips. A big single chunk of contiguous memory is hard to find in a live system running various processes.

4.3.2 Online Phase (Rowhammer attack)

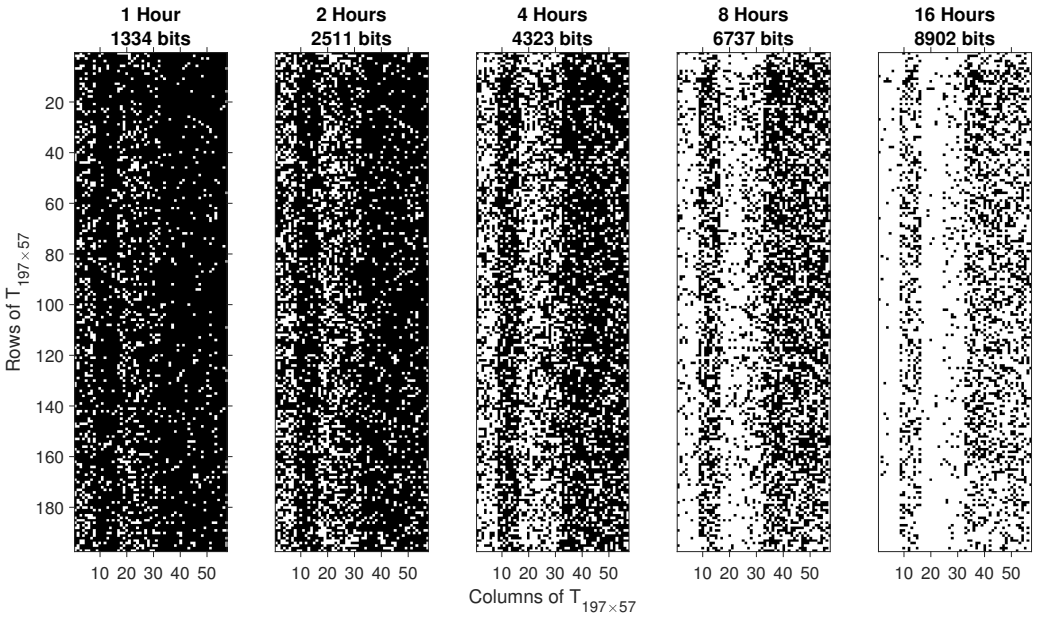


Figure 4.5: Online phase of Rowhammer attack. The plot depicts the bit-flips in the $\mathcal{T}$ matrix in the form of pixels, where white pixels indicate the flipped bits. Approximately 80% of the key bits are flipped in 16 hours.

The pre-processing phase gives a list of vulnerable physical addresses and the goal of the online phase is to first place the target linear transformation $\mathcal{T}$

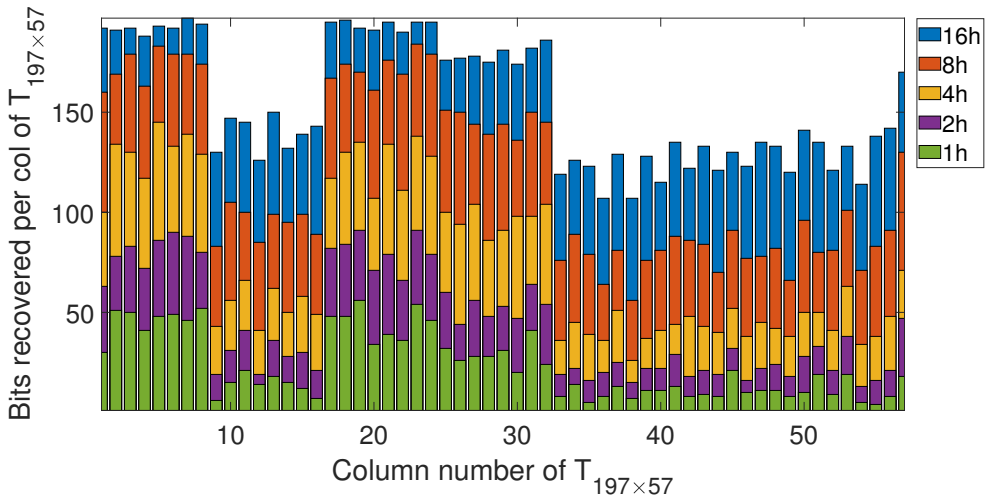


Figure 4.6: Number of bits recovered per column of $\mathcal{T}$ in 16 hours of online phase.

of LUOV scheme at one of those physical addresses and then do the double-sided rowhammer again to get bit-flips in $\mathcal{T}$. For experimental purposes, we achieve this by keep allocating memory pages for the $\mathcal{T}$ with in the victim process until it either gets in one of the target addresses or one page next to a target address. This is because one DRAM row comprises 8 kB having two 4 kB pages and the size of the $\mathcal{T}$ matrix is less than a 4 kB page. For LUOV-7-57-197, the size of the linear transformation matrix $\mathcal{T}$ is $(57 \times 197)/8 = 1,404$ Bytes. Hence, if $\mathcal{T}$ gets in either of the two pages of the target row, we can start doing the Rowhammer attack. This process is time consuming as a large number of memory pages are allocated until $\mathcal{T}$ is mapped to the desired target address.

The placement of victim can also be achieved by using other techniques

present in the literature like spraying [67, 151, 181], grooming [173] and memory-waylaying [65, 182, 107]. Figure 4.5 shows the number of $\mathcal{T}$ bits flipped against time. The number of bit-flips do not increase linearly with time as we start getting the same bit-flips over and over again. Out of 25,335 bit-flips, only 8,902 were unique in 16 hours of the online phase. We can see that in the first hour we get 1,334 bit-flips, little less than a double in two hours and after that the bit-flips are getting repeated more often. Still, we are able to recover approximately 80% of the $\mathcal{T}$ bits in 16 hours. Figure 4.6 indicates the number of bit-flips per column of $\mathcal{T}$ which will be used by QuantumHammer in Section 4.4. The working of the attack is verified when the victim and attacker process are running independently in different terminals but due to the system crashes, memory constraints, disk errors and synchronization problems, the attacker and the victim process are combined as we needed to run the experiments for l6 hours continuously. For example, in a 2GB memory in which only 25% memory is available in a running system, two separate processes start taking the swap partition. This makes the system slow and unresponsive.

4.3.3 Post-processing Phase

The post-processing phase takes the faulty signatures collected in the online phase and is able to recover the key bits of $\mathcal{T}$. We consider it a weakness of the LUOV scheme because the faulty public signatures should not lead back to the secret key bits of $\mathcal{T}$. In the LUOV scheme, if $\mathcal{T}$ is recovered,

the secret central map $\mathcal{F}$ can be easily computed using the public map $\mathcal{P}$, as $\mathcal{P} = \mathcal{F} \circ \mathcal{T}$. Thus, recovering $\mathcal{T}$ is enough to break the scheme and forge any signature. The bit-tracing algorithm can be executed offline on any other system or cluster independently.

In the last stage of LUOV, there is a linear transformation $\mathcal{T}$ which gives the signature as the output. The intuition behind the bit-tracing attack is to flip bits in $\mathcal{T}$ and observe the effect on the signature values. Once we get a faulty signature, the signature verification algorithm is utilized as an oracle to correct the signature by iteratively modifying the faulty signature. When the correct signature is found and the verification test is passed, bit-tracing algorithm mathematically tracks back to the flipped bit and is able to get information about the position of the flipped bit. By filling the attacker rows with all ones and the victim row with all zeros, we can tell that the flipped bit was a zero or vice versa.

We target the last part of the signature generation algorithm of LUOV which is a linear transformation $s_{v\times 1} = \mathcal{T}_{v\times m} \times o_{m\times 1} \oplus v_{v\times 1}$ or in the matrix form as Equation 4.1.

$$\begin{bmatrix} s_1 \\ \vdots \\ s_v \end{bmatrix} = \begin{bmatrix} t_{11} & \dots & t_{1m} \\ \vdots & \ddots & \vdots \\ t_{v1} & \dots & t_{vm} \end{bmatrix} \times \begin{bmatrix} o_1 \\ \vdots \\ o_m \end{bmatrix} \oplus \begin{bmatrix} v_1 \\ \vdots \\ v_v \end{bmatrix} \tag{4.1}$$

Our bit-tracing algorithm for LUOV is given in Algorithm 6 which takes $v \times m$ signature verifications to trace 1 bit of $\mathcal{T}$ for 1 bit-flip. The inputs

to the algorithm are all public parameters: 1) the faulty signature S which we get after flipping the bit using Rowhammer attack, 2) the message M, 3) public map $\mathcal{P}$. The algorithm finds the correct signature by replacing each element of s with the XOR of itself and each element of the oil variables. On successful verification, the indexes of the bit-flip (r, c) in $\mathcal{T}$ are returned which indicates the bit-flip position in $\mathcal{T}$.

Algorithm 6 Bit-tracing algorithm for LUOV - Offline

```
1: procedure TRACEBIT(S, salt)
   Input: (S, salt)                                  ▷ Faulty signature
   M - Message  ,  P - Public map
   Output: Returns (r, c)          ▷ Recovered bit-flip position in T
2:     h ← H(M||0x00||salt)
3:     for r from 1 to v do
4:         for c from 1 to m do
5:             S[r] ← S[r] ⊕ S[c + v]
6:             if P(S) ≠ h then
7:                 S[r] ← S[r] ⊕ S[c + v]
8:             else
9:                 return r, c
10:                break
11:            end if
12:        end for
13:    end for
14: end procedure
```

If there is a bit-flip somewhere in $\mathcal{T}$, say at index (r, c), multiplication of r^{th} row of $\mathcal{T}$ and o results in a difference in s which is o_c at the term s_c. As the o and s are public, we can try all potential differences which are the elements of o XORed with all elements of the s to check which one of the oil variable caused the error due to a bit-flip in $\mathcal{T}$. We achieve this by replacing

each element of s with its XOR of all elements of o one by one and pass it to the signature verification oracle. Once, the signature gets verified, we get the indexes of the flipped bit in $\mathcal{T}$, which are (r, c). The value of the bit can be recovered by knowing the direction of the bit-flip. A $0 \rightarrow 1$ flip means that the key bit was originally 0 and a $1 \rightarrow 0$ bit-flip means that the key bit was 1. The amount of time needed for this offline post-processing bit-tracing algorithm is shown in Table 4.1 for all variants of LUOV AVX2 optimized implementations.

For 2-bit scenario, Algorithm 6 can be modified to recover 2 bits of $\mathcal{T}$ if $v \times m$ verifications fail to correct the signature. In this scenario, there are two cases. First one is that 2 bit-flips are in the different rows of $\mathcal{T}$ which requires us to take all combinations of elements of s, 2 at a time which is $\binom{v}{2}$. For each combination, we need m^2 verifications by XORing both elements of the combination with all elements of o. The first scenario hence needs $m^2 \times \binom{v}{2}$ verifications. For 2 bit-flips in the same row, the error is just in one element of s. For each element of s, we need to XOR all combinations of o, 2 at a time with the element of s, until we find the correct signature. This scenario requires $v \times \binom{m}{2}$ verifications. In total, we need $vm + m^2\binom{v}{2} + v\binom{m}{2}$ signature verifications for 1 bit and 2 bit scenarios combined. If there are multiple bit-flips in $\mathcal{T}$ in the online phase, they can be controlled by changing the data patterns in the aggressor rows and turning on and off certain bit-flips. We have successfully tested this method via an independent experiment. But found that it increases the duration of the online phase. It was more efficient

to just ignore the rare cases of more than 2 bit-flips.

Table 4.1: Post computation times for bit-tracing attack, Algorithm 6 on LUOV. This computation is done offline and can easily be parallelized and distributed. The measurements are taken on a single machine with a Skylake Intel Core i5-6440HQ CPU @2.6GHz processor. Note that these timings are for $v \times m$ verifications which is the worst case scenario. In practice, the bits are traced in fewer iterations depending upon the position of the bit-flip in $\mathcal{T}$.

Implementation	LUOV Variant	1-bit Tracing Offline(Sec)
AVX2	`luov-7-57-197-chacha`	1.58
	`luov-7-57-197-keccak`	11.44
	`luov-7-83-283-chacha`	10.46
	`luov-7-83-283-keccak`	58.22
	`luov-7-110-374-chacha`	35.19
	`luov-7-110-374-keccak`	239.34
AVX2 (precompute)	`luov-7-57-197-chacha`	0.36
	`luov-7-57-197-keccak`	0.36
	`luov-7-83-283-chacha`	1.64
	`luov-7-83-283-keccak`	1.63
	`luov-7-110-374-chacha`	4.98
	`luov-7-110-374-keccak`	4.99

4.3.4 Performance

Table 4.1 summarizes the time it takes to perform the post-processing time, i.e. the bit-tracing step. The computation is performed offline and can easily be parallelized since all this step does is to search for the fault location using the faulty signature. Enabled by Rowhammer, the bit-tracing attack manages to effectively recover bits of $\mathcal{T}$, the secret key matrix. Assuming

single faults, each recovered secret key bit requires a successful Rowhammer fault injection, which takes significant amount of time, i.e. we get about 23 flips per minute on our target platform in the first hour, while the flipping performance degrades with time, see Figure 4.5. Remember that for LUOV-7-57-197, we have 11,229 key bits to recover. Recovering the entire signature key bit-by-bit would take more than 16 hours of live observation which is unrealistic.

Alternatively, if we try to reduce the complexity of the LUOV MQ equation system to enable SAT solving then the best strategy would be to target specific rows of $\mathcal{T}$ using Rowhammer. Using each fully recovered row, we can recover a vinegar variable. As the original oil and vinegar scheme with an equal number of oil and vinegar variables already was shown to be breakable by Patarin, we need to eliminate $v - m$ variables which means $v - m$ rows of $\mathcal{T}$ need to be recovered using Rowhammer attack. This approach too is costly.

Rather than trying to recover the entire key or to eliminate vinegar variables until the security collapses, we introduce a novel attack, i.e., QuantumHammer as described in the following section, that uses the bit-tracing attack as an oracle.

4.4 QuantumHammer

We present QuantumHammer attack that significantly reduces the complexity of the LUOV MQ system by splitting it into smaller MQ problems. This is achieved by using the bit-tracing attack as an oracle to recover a small number of specifically chosen key bits. Overall attack complexity is drastically reduced compared to an attack that only uses bit-tracing. Next we delve into the details of the LUOV construction. Specifically, we analyze the key generation process to obtain a simpler formulation.

4.4.1 Divide-and-Conquer Attack

Let $MQ(v, m)$ and $ML(v, m)$ represent systems of m quadratic and m linear equations of v unknowns, respectively. Our aim is to attack key generation part of LUOV explained in Section 2.3.1 and recover boolean private linear transformation matrix $\mathcal{T}$. The public parameter Q_2 is generated from the intermediate $m \times m$ the boolean matrix P_3^k by Equation 2.4. P_3^k is formulated in terms of P_1^k, P_2^k and $\mathcal{T}$ where P_1^k and P_2^k are publicly re-generatable from public parameter Q_1. Therefore, for a direct attack, we need to solve a $MQ(v \cdot m, \frac{m^3+m^2}{2})$ in which equations are from Equation 2.4 and unknowns are the elements of $\mathcal{T}$. For the NIST Round 2 submission LUOV-7-57-197, with parameters $m = 57$ and $v = 197$ solving the overall quadratic system appears infeasible unless there is a major breakthrough.

Instead of trying to attack the mv-bit secret key matrix $\mathcal{T}$ as a whole,

or recovering some part of $\mathcal{T}$ by bit-tracing attack and applying exhaustive search to the rest, we gain a more powerful attack, QuantumHammer, by exploiting the relation between the public matrices P_1^k, P_2^k, Q_2, where k from 1 to m and private linear transformation matrix $\mathcal{T}$ (remember the LUOV key generation process in Figure 2.6).

We start by making some observations on the structure of Q_2.

4.4.2 Observations on the structure of Q_2

Even though Q_2 yields a large $MQ(v \cdot m, \frac{m^3+m^2}{2})$ system, one can divide Q_2 column by column and consider it as a set of combination of discrete, smaller MQ systems in terms of columns of $\mathcal{T}$, i.e., set of $MQ(v, m)$ and $MQ(2v, m)$ systems by Equation 2.4 and 2.3.

Assuming bit-tracing attack recovers x bits from a column of $\mathcal{T}$, it is possible to reduce the related systems into one of $MQ(v-x, m)$, $ML(v-x, m)$ and $ML(v, m)$ systems. These equations have certain structure that we wish to exploit to recover the entire $\mathcal{T}$, column by column. The following definitions and observations will lead us to divide and conquer attack:

1. Define $\mathcal{A}_i$ as the set of m equations of v variables, $MQ(v, m)$ where equations are $Q_{2_{k,\beta_{i,i}}} = p_3^k(i, i)$ for k from 1 to m and variables are the i^{th} column of $\mathcal{T}$, i.e., $t_{1i}, \cdots, t_{vi}$.

2. Suppose x elements of i^{th} column of $\mathcal{T}$ are known/recovered. Define $\mathcal{A}_i(x)$ as a reduced system of $\mathcal{A}_i$ by inserting the x recovered bits into

$\mathcal{A}_i$. Note that, inserting x variables into $\mathcal{A}_i$ reduces the system to $MQ(v-x,m)$ from $MQ(v,m)$.

3. Define $\mathcal{B}_{i,j}$ as the set of m equations of $2v$ variables, $MQ(2v,m)$ where the equations are $Q_{2(k,\beta_{i,j})} = p_3^k(i,j) \oplus p_3^k(j,i)$ for k from 1 to m and variables are the i^{th} and j^{th} columns of $\mathcal{T}$, i.e., $t_{1i}, \cdots, t_{vi}$, $t_{1j}, \cdots, t_{vj}$.

4. Suppose i^{th} column of $\mathcal{T}$, i.e. $t_{1i}, \cdots, t_{vi}$, is known. Inserting these variables into $\mathcal{B}_{i,j}$ reduces the system from quadratic $MQ(2v,m)$ system to a linear $ML(v,m)$ system, where the unknowns are $t_{1j}, \cdots, t_{vj}$. We denote the insertion of the i^{th} column of $\mathcal{T}$ into $\mathcal{B}_{i,j}$ by $\mathcal{B}_{i,j}(t_i, 0)$. Note that, this reduces the hard problem $MQ(2v,m)$ into underdetermined linear $ML(v,m)$ system.

5. Suppose x elements of the j^{th} column of $\mathcal{T}$ and the entire i^{th} column of $\mathcal{T}$ are known. Inserting these known variables into $\mathcal{B}_{i,j}$ reduces the system from $MQ(2v,m)$ to $ML(v-x,m)$. The new system is denoted by $\mathcal{B}_{i,j}(t_i, x)$. If $x \geq v-m$ then the system reduces to an overdetermined linear system from an underdetermined one. Therefore, the new system has a unique solution and is efficiently solvable.

4.4.3 A Practical Divide and Conquer Attack

We are going to use bit-tracing attack as an oracle to recover some bits of some column in matrix $\mathcal{T}$. Informally, QuantumHammer proceeds as

follows:

Bit-tracing (Section 4.3):

Suppose x bits in some column of $\mathcal{T}$ is enough to reduce $MQ(v, m)$ system into a solvable $MQ(v - x, m)$ system. When x bits are recovered via bit-tracing in some column, we stop bit-tracing and recover the bits as explained in Section 4.3. Apply bit-tracing attack, and recover bits of $\mathcal{T}$ until the highest number of recovered bits from a column is $v - m$. Pick the highest $\lceil \frac{v}{m} \rceil$ columns. Assume the highest number of recovered bits are x_1, x_2, x_3 and x_4 bits in α_1, α_2, α_3 and α_4^{th} columns of $\mathcal{T}$, respectively. Note that, bit-tracing recovers additional bits from different columns of $\mathcal{T}$. But, having $\lceil \frac{v}{m} \rceil$ columns of $\mathcal{T}$ is enough to reduce the MQ systems into ML systems and can efficiently solve it. Therefore, we do not need to use the remaining bits recovered by bit-tracing in different columns of $\mathcal{T}$.

Quadratic Steps (Algorithm 7):

Algorithm 7 Quadratic Steps

1: **procedure** QUADSTEPS($(\alpha_1, x_1),\cdots,(\alpha_\kappa, x_\kappa)$)

Input: High recovered columns from Bit-tracing

Output: $(t_{\alpha_i}, \cdots, t_{\alpha_\kappa})$ ▷ entire columns of input vectors

2: $\mathcal{A}_{\alpha_1}(x_1) \leftarrow MQ\_Gen(\alpha_1, x_1)$ ▷ Algorithm 10 in Appx

3: $t_{\alpha_1} \leftarrow Eqn\_Solver(\mathcal{A}_{\alpha_1}(x_1), \emptyset)$ ▷ Algorithm 12 in Appx

4: **for** i **from** 2 **to** $\kappa = \lceil \frac{v}{m} \rceil$ **do**

5: $\mathcal{A}_{\alpha_i}(x_i) \leftarrow MQ\_Gen(\alpha_i, x_i)$. ▷ Quadratic Step

6: **for** j **from** 1 **to** i-1 **do**

7: $\mathcal{B}_{\alpha_i,\alpha_j}(x_i, t_j) \leftarrow ML\_Gen((\alpha_i, x_i), (\alpha_j, t_j))$ ▷ Algorithm 11 in Appx

8: **end for**

9: $t_i \leftarrow Eqn\_Solver(\mathcal{A}_{\alpha_i}(x_i), \bigcup_{j=1}^{i-1} \mathcal{B}_{\alpha_i,\alpha_j}(x_i, t_j))$

10: **end for**

11: **end procedure**

1. Consider $\mathcal{A}_{\alpha_1}$, more specifically, consider the elements of $\beta_{\alpha_1,\alpha_1} = (\alpha_1 - 1)m + \frac{\alpha_1(\alpha_1+1)}{2}^{th}$ column of Q_2 which are $p_3^k(\alpha_1, \alpha_1)$ terms of P_3^k for k from 1 to m and α_1 is the highest column of $\mathcal{T}$. Inserting x_1 recovered bits into the system $\mathcal{A}_{\alpha_1}$ reduces the $MQ(v, m)$ system into $MQ(v - x_1, m)$. We recover the remaining $v - x_1$ elements of α_1^{th} column $\mathcal{T}$ which are $t_{1\alpha_1}, \cdots, t_{v\alpha_1}$.

2. Insert recovered α_1^{th} column of $\mathcal{T}$ into $\mathcal{B}_{\alpha_1,\alpha_2}$ and x_2 recovered bits of α_2^{th} column of $\mathcal{T}$ into the systems $\mathcal{B}_{\alpha_1,\alpha_2}$ and $\mathcal{A}_{\alpha_2}$ reducing the systems into $\mathcal{B}_{\alpha_1,\alpha_2}(t_{\alpha_1,x_2})$ and $\mathcal{A}_{\alpha_2}(x_2)$, respectively. Thus, the system reduces to practically solvable $MQ(v-x_2,m)\bigcup ML(v-x_2,m)$. The solution of the reduced system gives the full α_2^{nd} column of $\mathcal{T}$ which are $t_{1\alpha_2},\cdots,t_{v\alpha_2}$. Note that, even though solving $MQ(v-x_2,m)$ is harder than solving $MQ(v-x_1,m)$, there are m additional linear equations from $ML(v-x_2,m)$ which decrease the number of unknowns from $v-x_2$ to $v-x_2-m$. Therefore, $MQ(v-x_2,m)\bigcup ML(v-x_2,m)$ is a much easier system to solve than $MQ(v-x_1,m)$.

3. Apply the same strategy to α_3^{th} column of $\mathcal{T}$, i.e., insert α_1 and α_2^{th} columns of $\mathcal{T}$ which are recovered in the first two steps, into the systems $\mathcal{B}_{\alpha_1,\alpha_3}$, $\mathcal{B}_{\alpha_2,\alpha_3}$ and $\mathcal{A}_{\alpha_3}$. The complexity reduces to $\mathcal{B}_{\alpha_1,\alpha_3}(t_1,x_3)\bigcup\mathcal{B}_{\alpha_2,\alpha_3}(t_2,x_3)\bigcup\mathcal{A}_{\alpha_3}(x_3)$. Thus, the system reduces to $ML(v-x_3,2m)\bigcup MQ(v-x_3,m)$ which has the solution of x_3^{th} unknowns from the α_3 column which are $t_{1\alpha_3},\cdots,t_{v\alpha_3}$. Note that, the solution of the system is equivalent to the solution of $MQ(v-x_3-2m,m)$ which is much easier than the previous steps.

4. The same strategy can be applied to recover α_4^{th} column of $\mathcal{T}$ by using previously recovered columns of $\mathcal{T}$ in addition to recovered x_4 bits of the α_4^{th} column in bit-tracing attack. Inserting the known elements will reduce the complexity to $ML(v-x_4,3m)\bigcup MQ(v-x_4,m)$. This is

a solvable system since it is equivalent to $MQ(v - x_4 - 3m, m)$. The solution gives us α_4^{th} column elements $t_{1\alpha_3}, \cdots, t_{v\alpha_3}$.

After $\lceil \frac{v}{m} \rceil$ steps, $\lceil \frac{v}{m} \rceil$ recovered columns of $\mathcal{T}$ are enough to reduce the smaller MQ systems of remaining columns into overdetermined ML systems. In the following steps, we are going to explain how one can reduce any small MQ system to a ML if $\lceil \frac{v}{m} \rceil$ columns are recovered.

Linear Steps (Algorithm 8):

Algorithm 8 Linear Steps

1: **procedure** LINEARSTEPS($(\alpha_1, t_{\alpha_1}),\cdots,(\alpha_\kappa, t_{\alpha_\kappa})$)
   Input: Recovered Columns
   Output: $(t_1, \cdots, t_m)$ ▷ columns of $\mathcal{T}$
2: **for** i **from** 1 **to** m **do** ▷ except $\{\alpha_1, \cdots, \alpha_\kappa\}$
3: **for** j **from** 1 **to** κ **do**
4: $\mathcal{B}_{i,\alpha_j}(\emptyset, t_{\alpha_j}) \leftarrow ML\_Gen((i, \emptyset), (\alpha_j, t_{\alpha_j}))$ ▷ $ML(v, m)$
5: **end for**
6: $t_i \leftarrow Eqn\_Solver(\emptyset, \bigcup_{j=1}^{\kappa} \mathcal{B}_{i,\alpha_j}(0, t_{\alpha_j}))$ ▷ $ML(v, \kappa \cdot m)$
7: **end for**
8: **return** t_i
9: **end procedure**

Suppose there are $\lceil \frac{v}{m} \rceil$ recovered columns of $\mathcal{T}$ from the quadratic steps. Inserting the bits of recovered columns into the related systems will give us

the following reduced ML system:

$$\bigcup_{i=1}^{\lceil \frac{v}{m} \rceil} \mathcal{B}_{\alpha_i,\beta}(t_i, 0)$$

where α_i's are the column numbers of recovered columns of $\mathcal{T}$ and β is the column number of attacked column of $\mathcal{T}$. This gives us an overdetermined $ML(v, \lceil \frac{v}{m} \rceil \cdot m)$ system which can be solved efficiently.

Note that, by the linear steps, we can recover the rest of $\mathcal{T}$ columns one by one in $m - \lceil \frac{v}{m} \rceil$ steps.

4.5 Experimental Results

Table 4.2: Exhaustive search timing for different sizes of $MQ(n, n)$ taken on Nvidia GTX 1080Ti GPU.

$n = m$	Time	$n = m$	Time	$n = m$	Time
40	2.7s	52	1h 32m	55	6h 15m
43	12s	53	3h 3m	56	24h 45m
49	11m 33s	54	3h 6m	57	49h 30m

Bit-tracing:

We have attacked the constant-time AVX2 optimized implementation of LUOV-7-57-197 [175] on a Haswell system equipped with Intel Core i7-4770 CPU @ 3.40GHz, 2 GB DDR3 DRAM, model Samsung (M378B5773DH0-CH9). Pre-processing (templating) step is performed in 5.7 hours to find

17,129 physical addresses vulnerable to Rowhammer. After that, 16 hours of online phase is carried out in which the victim is running and performing signing operations. Using bit-tracing attack, we recover 4,116 bits with 3 hours and 49 minutes of online observation. The faulty signatures are processed offline on a separate machine to recover the key bits [2].

Note that the attack recovers up to 140 bits in any column of $\mathcal{T}$ which is enough for a successful QuantumHammer attack. The distribution of the bits in the 57 columns of $\mathcal{T}$ is given in Figure 4.7. Some columns of $\mathcal{T}$ have been located in DRAM buffers that are more flippy than others.

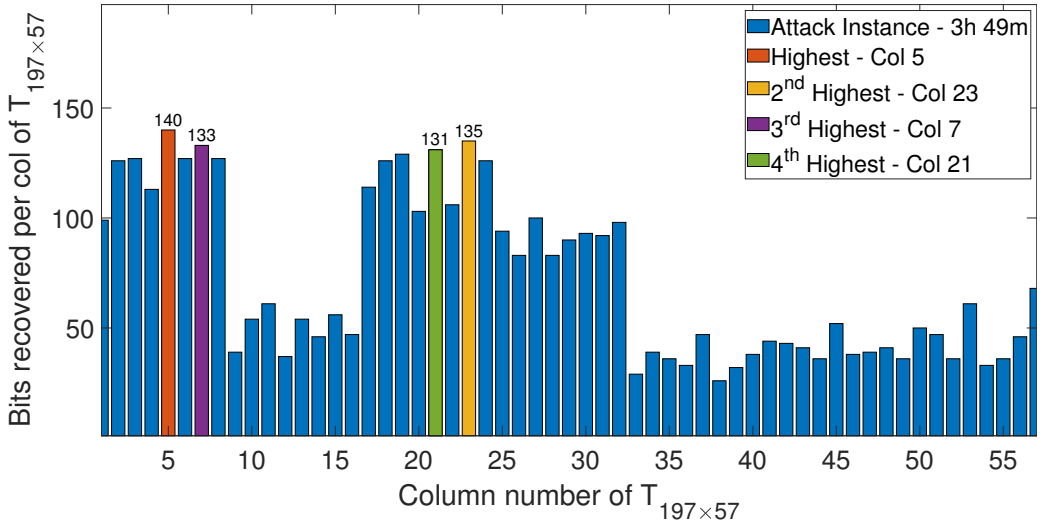


Figure 4.7: Bit-tracing attack recovers up to 140 key bits per column of $\mathcal{T}$ in less than 4 hours of Rowhammer on a 2 GB DDR3 Samsung DRAM (M378B5773DH0-CH9).

Quadratic Steps:

[2]The source code for QuantumHammer is made available at `http://github.com/VernamLab/QuantumHammer`.

In preparation for QuantumHammer, $p_3^k(i,j)$ were generated by the Equation 2.3 using the coefficients from P_1^k and P_2^k. MQ systems are generated by Equation 2.4 using $p_3^k(i,j)$ equations. To solve the generated system of equations, we focused on $\lceil \frac{v}{m} \rceil = \lceil \frac{197}{57} \rceil = 4$ columns with the highest number of recovered bits: columns 5, 23, 7 and 21 with 140, 135, 133 and 131 recovered bits, respectively. In every step of quadratic and linear steps, we recover a column of $\mathcal{T}$. Experimental results of quadratic steps are given in Table 4.3.

It is important to note that, in the first step, we recover the 5^{th} column of $\mathcal{T}$, by solving a $MQ(57,57)$ system reduced from the underdetermined $MQ(197,57)$ thanks to 140 recovered bits obtained by the bit-tracing attack. Without it, it would not be possible to recover the rest of the 5^{th} column. The system is solved by exhaustive search in roughly 49 hours on i7 Intel CPU with Nvidia GTX 1080 Ti GPU. In Table 4.2 exhaustive search timing for different sizes of $MQ(n,n)$ is given. We used the GPU implementation of [24] compiled using the Nvidia CUDA 10.0 framework. The offline exhaustive search can be trivially sped up by employing multiple GPUs since the search is fully parallelizable.

In the second step, we targeted 23^{rd} column with the 135 bits recovered bits from bit-tracing. With these 135 bits, the system starts out as $MQ(62,57)$. Next, we insert the values obtained from the 5^{th} column to reduce the complexity to $MQ(5,57)$. We can instantly solve this system via exhaustive search. At this point, by inserting the recovered bits in the

first two steps, we reduced the remaining equations into (over-defined) linear systems only.

Linear Steps:

In the quadratic steps, we recovered 4 columns of $\mathcal{T}$. Inserting these values into remaining equations will give us an underdetermined $ML(197, 57)$ system. We end up with 228 linear equations with 197 unknowns which can be solved via Gaussian elimination. Even though we can generate more linear equations by using more bits of $\mathcal{T}$ previously recovered by bit-tracing, we do not need any extra equations to solve the system. In 53 steps, all the remaining columns of $\mathcal{T}$ are recovered as summarized in Table 4.4.

4.6 Countermeasures

The effectiveness of QuantumHammer requires us to consider practical countermeasures at various levels:

Preventing Rowhammer: The most effective solution to prevent any Rowhammer fault-injection attack is to implement stronger isolation, such as using dedicated instances for any sensitive processes. If isolation is not possible, an effective alternative approach to reduce the impact of Rowhammer is increasing the DRAM row refresh rate. DDR3 and DDR4 refresh each row at least every 64 ms. That said many systems permit the refresh rates at 32 or 16 ms for better memory stability.

Online Detection of Rowhammer: One may also seek to employ active

countermeasures for online detection of Rowhammer. For this, Hardware Performance Counters (HPCs) can be used to monitor counters like cache hits and cache misses to detect Rowhammer.

Suppressing Faulty Signatures: Another way to counter faults in the signature schemes is to verify the signatures at sender side before sending it but that will involve additional processing. A faster approach can be to repeat the final linear transformation stage of the signing operation with an independently generated $\mathcal{T}$ and check if the signatures are identical. Clearly, in this case one must ensure that the checking mechanism itself does not become a target itself.

4.7 Discussion

The first algebraic attack targeting UOV type schemes that does not require any physical access is Reconciliation attack introduced by Ding et al. [46]. The attack aims to invert the public map. Decomposing the public map $\mathcal{P}$ into the multiplication of a series of specific linear transformations allows the attacker to recover every transformation one-by-one by exhaustive search algorithms such as F4/F5 or FXL. The result is a purely algebraic attack that significantly reduces the assumed security margin of LUOV.

In the Divide-and-Conquer Attack, we follow a similar approach in the sense that we exploit one of the innovations of LUOV, i.e. the structure of the public key Q_2. Being empowered by Rowhammer and the bit-tracing attack,

we take the attack further into full recovery of all key bits. This is achieved by converting the MQ system into smaller under-determined MQ systems which are in the same form as the original MQ system. Instead of decomposing the matrix, we regroup the equations into a discrete set of variables. Without the amplification of the fault attack, it would not be possible to solve the smaller MQ systems since they are underdetermined. In this sense, our overall QuantumHammer attack represents a novel approach.

Preventing Algebraic Collapse: What enables QuantumHammer is that the MQ equations use a small subset of the key bits in the way the key generation primitive is defined for LUOV. Hence, recovering a small fraction of the key bits via Rowhammer and the bit-tracing attack was sufficient to collapse the MQ system to smaller size tractable MQ systems. In many scenarios, a small fraction of the key bits may be recovered using side-channel attacks. Hence this attack poses a serious threat to real-life deployments. To prevent such collapse it would be prudent to check the resulting MQ system underlying the security of the scheme *at design time* under the assumption that any fixed size subset of the key bits are compromised.

4.8 Conclusion

This chapter shows that both hardware and cryptographic security are of utmost importance for cryptosystems. LUOV signature scheme, a round 2 finalist of NIST's PQC standardization process is based on the well known

oil and vinegar scheme which withstood over two decades of cryptanalysis. We have analyzed the scheme both mathematically and implementation wise and found weaknesses in both areas. The QuantumHammer attack combines both weaknesses to launch a successful attack recovering the full secret key of the scheme. There is a need to evaluate the hardware and software implementations of the cryptosystems in combination with the mathematical evaluation.

Table 4.3: Quadratic steps in our experimental QuantumHammer on LUOV-7-57-197. In every step, table lists the targeted column of $\mathcal{T}$, number of recovered bits during bit-tracing, size of ML system obtained after inserting previously recovered columns, complexity of the solution for the linear part, number of linear equations and unknowns, parameters for the quadratic part, and the complexity of the overall system after using ML to reduce the unknowns in quadratic part.

Step	Target Col	Num. of Rec. bits	Linear Part					Quadratic Part				Overall Complexity
			Insrtd Col	Equation System	Complexity	ML System		Equation System	Complexity	MQ System		
						Linear Eqns	Unk			Quad Eqns	Unk	
1	5	140	-	-	-	-	-	$\mathcal{A}_5(140)$	$MQ(57,57)$	57	57	$MQ(57,57)$
2	23	135	5	$\mathcal{B}_{5,23}(197,135)$	$ML(62,57)$	57	62	$\mathcal{A}_{23}(135)$	$MQ(62,57)$	5	62	$MQ(5,57)$
3	7	133	5	$\mathcal{B}_{5,7}(197,133)$	$ML(64,57)$	114	64	$\mathcal{A}_7(133)$	$MQ(64,57)$	57	64	$ML(64,114)$
			23	$\mathcal{B}_{23,7}(197,133)$	$ML(64,57)$							
4	21	131	5	$\mathcal{B}_{5,21}(197,131)$	$ML(66,57)$	171	66	$\mathcal{A}_{21}(131)$	$MQ(66,57)$	57	66	$ML(66,171)$
			23	$\mathcal{B}_{23,21}(197,131)$	$ML(66,57)$							
			7	$\mathcal{B}_{7,21}(197,131)$	$ML(66,57)$							

Table 4.4: Linear steps in our experimental QuantumHammer on LUOV-7-57-197. In every step, table lists the targeted column of $\mathcal{T}$, inserted columns used to generate ML system, and resultant equation systems, the size of the generated ML systems, and the number of equations and unknowns in the overall linear system and overall complexity are given.

Step Nmbr	Target Col	Linear Part					Overall Complexity
		Inserted Col	Equation System	Equivalent System	ML System Linear Equations	ML System Unknowns	
5	1	5	$\mathcal{B}_{5,1}(197,0)$	$ML(197,57)$	228	197	$ML(197,228)$
		23	$\mathcal{B}_{23,1}(197,0)$	$ML(197,57)$			
		7	$\mathcal{B}_{7,1}(197,0)$	$ML(197,57)$			
		21	$\mathcal{B}_{21,1}(197,0)$	$ML(197,57)$			
⋮	⋮				⋮	⋮	⋮
57	57	5	$\mathcal{B}_{5,57}(197,0)$	$ML(197,57)$	228	197	$ML(197,228)$
		23	$\mathcal{B}_{23,57}(197,0)$	$ML(197,57)$			
		7	$\mathcal{B}_{7,57}(197,0)$	$ML(197,57)$			
		21	$\mathcal{B}_{21,57}(197,0)$	$ML(197,57)$			

Chapter 5

Signature Correction Attack on Dilithium Signature Scheme

In this chapter, we introduce a novel Signature Correction Attack that not only applies to the deterministic version but also to the randomized version of Dilithium and is effective even on constant-time implementations using `AVX2` instructions. The Signature Correction Attack exploits the mathematical structure of Dilithium to recover the secret key bits by using faulty signatures and the public-key. It can work for any fault mechanism which can induce single bit-flips. For demonstration, we are using Rowhammer induced faults. Thus, our attack does not require any physical access or special privileges, and hence could be also implemented on shared cloud servers. Using Rowhammer attack, we inject bit-flips into the secret key s_1 of Dilithium, which results in incorrect signatures being generated by the signing algo-

rithm. Since we can find the correct signature using our Signature Correction algorithm, we can use the difference between the correct and incorrect signatures to infer the location and value of the flipped bit without needing a correct and faulty pair. To quantify the reduction in the security level, we perform a thorough classical and quantum security analysis of Dilithium and successfully recover 1,851 bits out of 3,072 bits of secret key s_1 for security level 2. Fully recovered bits are used to reduce the dimension of the lattice whereas partially recovered coefficients are used to to reduce the norm of the secret key coefficients. Further analysis for both primal and dual attacks shows that the lattice strength against quantum attackers is reduced from 2^{128} to 2^{81} while the strength against classical attackers is reduced from 2^{141} to 2^{89}. Hence, the Signature Correction Attack may be employed to achieve a practical attack on Dilithium (security level 2) as proposed in Round 3 of the NIST post-quantum standardization process.

5.1 Contributions

We introduce the Signature Correction Attack on the Dilithium signature scheme which recovers secret key bits using only the faulty signatures and the public key. The attack works by first inducing bit-flips in the signing process, then collecting the faulty signatures and finally recovers the secret key bits while trying to correct the faulty signature using verification algorithm as an oracle. The faults are induced using a practical and software

only Rowhammer attack to produce the faulty signatures. In summary, in this chapter:

1. We introduce the Signature Correction Attack on Dilithium signature scheme on both randomized as well as deterministic version. The Signature Correction Attack only requires faulty signatures and the public key to mathematically locate single-bit faults on the secret key and to reveal the exact value of the bit-flip independent of the fault mechanism used.

2. We practically demonstrate the Rowhammer attack as a fault injection mechanism for Signature Correction on constant-time AVX2 implementation of Dilithium to generate the faulty signatures. Unlike physical fault mechanisms like EM, laser or clock-glitches, Rowhammer does not require any physical access which permits remote attacks on shared servers and is also applicable through JavaScript.

3. We recover partial secret key of 883 bits out of 3,072 bits for Dilithium security level 2 in about 2 hours of online Rowhammer attack and negligible amount of post-processing.

4. Careful analysis of the encoding of the secret key allows us to increase the number of recovered bits from 883 to 1,522. Additionally, analysis on the positions of the recovered bits reveal an additional 329 bits hence significantly extending the key material. Detailed analysis is given in Section 5.5.

5. Further analysis of lattice attacks shows a much reduced security for Dilithium security level 2 below the NIST's requirements, i.e. from 2^{128} to 2^{81}. Hence a partial key material collection and recovery with Signature Correction Attack followed by a lattice attack may indeed compromise Dilithium level 2 in practice.

6. Our Signature Correction Attack is applicable to all variants of Dilithium currently in Round 3 including the randomized versions recommended for side-channel and fault attacks.

7. We propose countermeasures to detect and prevent the Signature Correction Attack by temporal and spatial redundancy techniques as well as through Rowhammer mitigations.

5.2 Related Work

To the best of our knowledge, we are the first to demonstrate a fault attack on the randomized version of Dilithium in Round 3, which is also applicable to the deterministic version. Previous fault attacks on Dilithium [28, 146, 147] are only applicable to the deterministic version of Dilithium in Round 1. DFA requires a pair of faulty and correct signatures which can be collected by signing the same message twice and faulting in the second iteration. To prevent this DFA, Round 2 Dilithium introduced signature randomization by using a different nonce for every signature generation. Our proposed Signa-

ture Correction Attack is independent of the nonce and hence applicable to both randomized and deterministic versions of Dilithium. Bruinderink *et al.* [28] based their analysis on hypothetical faults without experimental confirmation. Ravi *et al.* [146, 147] have experimented using EM fault injection on the reference implementation for ARM-Cortex-M4. All of these attacks require physical access to induce the faults.

We propose Signature Correction Attack on Dilithium and demonstrate it on the constant-time AVX2 implementation using a Rowhammer attack. Our Signature Correction Attack can work with any single fault injection mechanism. We have chosen Rowhammer, because it is a software-only fault attack that can be launched remotely. Also, it has not been mitigated and can be dangerous in cloud scenarios where different users shares the same DRAM [181, 36].

5.3 Signature Correction Attack on Dilithium

To the best of our knowledge, there is no published work yet summarizing a fault attack on Dilithium that can work on randomized version of Dilithium. The randomized version randomly generates the nonce for each signing operation, which gives a different signature every time we sign the same message. Hence a standard DFA is not possible in case of randomized Dilithium as the attacker cannot recover a faulty and another correct signature for the same

message for the same nonce. Our novel Signature Correction Attack however is independent of the nonce, hence it is applicable to both randomized and deterministic versions of Dilithium.

The Signature Correction Attack exploits the mathematical structure of Dilithium to recover the secret key bits by using just the faulty signatures and the public key. Thus the attack can be executed offline after collecting sufficiently many faulty signatures from an active fault attack. The attack is independent of the concrete fault injection technique. The only requirement is that the faults should be single bit and induced before the signing step 13 of Algorithm 3 in secret key s_1. First we define the attacker model and then explain the phases of our Signature Correction Attack.

5.3.1 Attacker Model

When multiple tenants in cloud environments reside on the same server, they may share the same DRAM. The Rowhammer attack requires the attacker process and victim process to share a DRAM. The attacker process can then induce bit-flips by just reading its own memory repeatedly [181, 36, 176]. Moreover, the DRAM must be vulnerable to Rowhammer attack which means that its memory cells must be susceptible to the hammering effect. Most types of DRAMs have been shown to be vulnerable in [57, 42]. We are not using HugePages for contiguous memory as most of the servers are not configured to use HugePages. We will explain how we detect contiguous memory in Section 5.3 as it is required for the double-sided Rowhammer to

locate the neighboring rows in a DRAM bank. Also, the attacker has no knowledge of the DRAM mapping which is different for different memory controllers and DRAM configurations. The DRAM mapping maps physical addresses to actual DRAM ranks, banks, rows and columns which can be used by the attacker to co-locate with the victim in the same DRAM bank. In Section 5.3, we will explain how we use the row conflict side-channel for bank co-location. The attacker can induce bit-flips in the secret key s_1 of Dilithium but she has no control over the position of bit-flip within the s_1. For security level 2 for example, the size of s_1 is 4 kB and the attacker has no knowledge of location of the bit-flip within this 4 kB memory. Also, she has no idea of the value of the flipped bit. The attacker can just induce bit-flips from her own process and is able to collect the faulty signatures from the victim. She can only use these faulty signatures along with the pubic parameters to recover the secret key bits.

5.3.2 Phases of the Signature Correction Attack

There are three phases in the Signature Correction Attack. First, we identify vulnerable memory locations called as templating. Then, we perform double-sided Rowhammer attack on the victim in the online phase and collect the faulty signatures. Finally, we post-process the faulty signatures and recover the flipped secret key bits by Signature Correction algorithm.

1. **Templating Phase:** In a pre-processing phase of the Rowhammer attack, the attacker will identify vulnerable memory locations. The victim needs not to be present during this phase.

2. **Online Phase:** In the online phase, the victim process is forced to map onto the identified vulnerable memory locations from the templating phase. Then the attacker induces bit-flips inside the victim process and collects the faulty signatures generated by the victim.

3. **Post-processing Phase:** In this phase, the attacker uses the faulty signatures and the public key to recover the secret key bits using the *Signature Correction algorithm*. This phase can be carried out offline and can be parallelized and run on distributed systems for performance.

We will first explain our novel Signature Correction algorithm. Next, we describe the templating and online phase of Rowhammer to practically demonstrate the fault injection.

5.3.3 Signature Correction Algorithm for Dilithium

Signature Correction is a way to recover the flipped secret key bits using faulty signatures. Since challenge c is public, the generated error in the signature can be used to find the position of the flipped bit in the secret key. The error in the faulty signature can be some certain multiples of c. Therefore, if we somehow correct the faulty signature, we are able to find the position of the bit-flip. The main idea of Signature Correction is to

find the faulted bit in the secret key by the process of correcting the faulty signature by checking it using signature verification algorithm. The main difference between the standard DFA and Signature Correction Attack is that the attacker does not need to know the original signature. Finding the position of the flipped bit by the fault is different for every algorithm. In Algorithm 9, we explain it specifically for Dilithium.

How to trace back to the flipped bit using a faulty signature

s_1 is defined in S_η^l in Algorithm 2 step 4. Let $s_1 = (s_1^{(1)}, \cdots, s_1^{(l)})$ in vector form where $s_1^{(i)} = \Sigma_{j=0}^{n-1} a_j^{(i)} x^j$ and $-\eta \leq a_j^{(i)} \leq \eta$, $1 \leq i \leq l$ and $0 \leq j \leq n-1$. In Algorithm 3 step 13, signature is generated by $z = y + c \cdot s_1$ where $c = \Sigma_{j=0}^{n-1} c_j x^j$ is a constant challenge vector. If one bit in s_1 is flipped before the signature generation, it faults the output signature $\bar{z} = y + c \cdot \bar{s_1}$. Then, the difference of the faulty and original signatures is $\Delta z = z + \bar{z} = c \cdot (s_1 + \bar{s_1}) = c \cdot \Delta s_1$. Since just one bit is flipped in s_1, Δz has just one non-zero component which is $c_t \bar{a}_r^{(i)} x^{t+r}$, where $\bar{a}_r^{(i)}$ is the one bit difference, x^r is the position of the flip in s_1^i and c_t is the relevant component of the flipped bit in c. Note that, because of x^r term, c shifts to the right r times. Additionally, $\bar{a}_r$ is a power of 2 since it is the 1-bit difference.

For instance, if the flip is in the first coefficient of s_1, the changes in z appear at the same indices at which c is non-zero. If it is in the second coefficient of s_1, the changes appear at the non-zero indexes of one bit shifted version of c and so on. This observation makes it possible to trace back to

the faulty bit by just using the faulty signature and the public key. We can not only locate the position of the bit-flip but also the value of the flipped bit because both have a unique effect on the error.

To recover the secret key bit by just using the faulty signature σ', first we unpack the faulty signature to get the unpacked faulty signature z' and the challenge information $\tilde{c}$. Next we sample $\tilde{c}$ to get the challenge vector c and copy it to a temporary variable $\overline{c}$ as we will need to modify it. The idea is to add all n shifted versions of c in the faulty signature z' one by one and try to correct the faulty signature. We can verify the correctness using the Dilithium verification, Algorithm 4, as an oracle. When the signature with the attempted correction verifies, we can tell that this is the index of the flipped coefficient. We can also tell the value of the flipped bit by trying both addition and subtraction of the shifted versions of c. We need to repeat this step for all of the L elements of s_1 to trace the flipped bit for any of the elements of s_1.

This procedure works if the bit-flip occurs in the LSB of the coefficients. If the flip is the second or third LSB, we need to add a *multiplier*, which is 2^{bit_index}. This multiplier is first multiplied with the shifted version of the challenge vector $\overline{c}$ and then added to the faulty signature z'. In the Dilithium implementation, the coefficients of s_1 are stored as `int32_t`, but the values of the coefficients range up to four bits depending upon the security level. Hence, we need to check up to three or four bits, we call this number as B. The algorithm however is capable of going further but there is no useful

information on the MSB side as the remaining bits are the same as last useful LSB. So, we need to keep modifying the public challenge $\bar{c}$, multiply it with the *multiplier*, add it to the faulty signature z' and verify to see if the signature is corrected using the verification oracle. If the signature is correct, the algorithm returns the recovered bit of secret key s_1 as output. The algorithm needs at most $2 \times B \times L \times n$ number of verification to recover one bit of secret key. In practice however, the code breaks earlier upon finding the location. Algorithm 9 summarizes our attack.

5.3.4 Templating Phase

Signature Correction Attack needs a fault mechanism that can provide faulty signatures. We are practically inducing faults using Rowhammer, a software-only technique that does not require any physical access to the target machine. Recent research has shown that it can be applied over the network [164, 115] and even remotely through JavaScript [67, 42]. There is no effective countermeasure to prevent Rowhammer completely in DRAM chips so far. Recent research has demonstrated that it is possible to apply Rowhammer even on DDR4 memories with TRR [57] mitigation as well as on ECC memories [37]. Templating phase involves three steps: contiguous memory detection, bank co-location and double-sided hammering.

Algorithm 9 Novel Signature Correction Algorithm for Dilithium

Input: σ' - Faulty Signature, M - Message, pk - Public Key
Output: (row, col, bit_index, value) - Recovered secret key bit
$(z', h, \tilde{c}) \leftarrow unpack(\sigma')$
$c \leftarrow SampleInBall(\tilde{c})$
$\overline{c} \leftarrow c$
for bit_index **from** 1 **to** 32 **do**
    $multiplier \leftarrow 2^{bit\_index-1}$
    for row **from** 1 **to** L **do**
        for col **from** 1 **to** N **do**
            $\overline{z}[row] \leftarrow z'[row] + multiplier \times \overline{c}$
            $\overline{\sigma} \leftarrow pack(\overline{z}, h, c)$
            if $sig\_verify(pk, M, \overline{\sigma}) = true$ **then**
                return $(row, col, bit\_index, 1)$
            else
                $\overline{c} \leftarrow circ\_shift\_right(\overline{c})$
            end if
        end for
        for col **from** 1 **to** N **do**
            $\overline{z}[row] \leftarrow z'[row] - multiplier \times \overline{c}$
            $\overline{\sigma} \leftarrow pack(\overline{z}, h, c)$
            if $sig\_verify(pk, M, \overline{\sigma}) = true$ **then**
                return $(row, col, bit\_index, 0)$
            else
                $\overline{c} \leftarrow circ\_shift\_right(\overline{c})$
            end if
        end for
    end for
end for

Contiguous Memory Detection

For a double-sided Rowhammer, the attacker needs to allocate the rows exactly one above and one below around the victim in the actual DRAM. For this purpose, contiguous memory is a requirement for double-sided Rowhammer. It can be achieved using HugePages but that requires special configuration and privileges. We achieve contiguous memory detection using SPOILER [86] from normal user space without any special privileges. When the SPOILER peaks become equally distant apart, the physical addresses become contiguous. Figure 5.1 shows the frame numbers of memory pages inside a buffer. We can see the contiguous memory where the frame numbers are linearly increasing. A detailed description of this approach can be found in [86].

Bank Co-location

A DRAM is organized in banks and every bank has a row buffer. Rowhammer attack works when both the attacker and the victim are sharing the same bank. To find the virtual addresses mapping to the same bank, we use a side-channel which is based on the row conflict. When two addresses from the same bank are accessed, it takes longer as compared to the accesses from different banks. This is because one row loaded inside the row buffer needs to be written back to its original position before loading another row. The CPU cycles taken for accessing one address and the remaining are shown in Figure 5.2. Depending on the maximum values of the peaks, we can set a

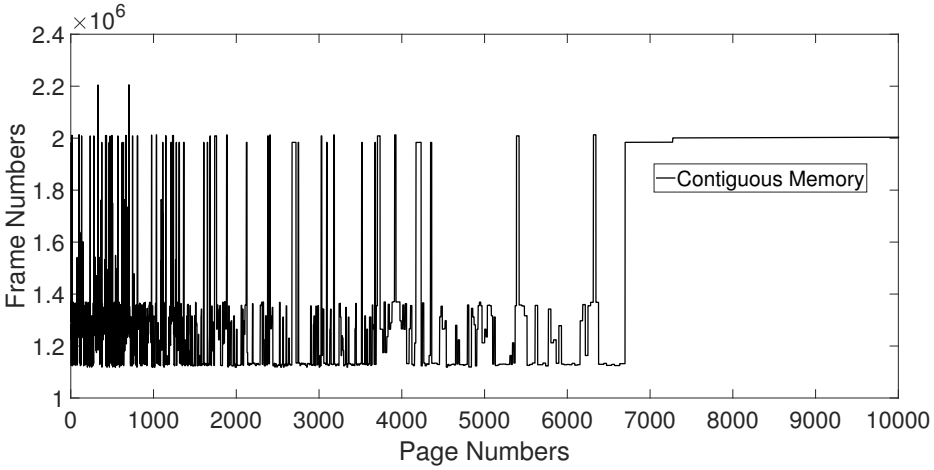


Figure 5.1: Contiguous memory detection. x-axis shows the page numbers of the allocated memory buffer, each page being 4 kB. On y-axis are the frame numbers of these pages in integer form. The straight line shows a linear increase in frame numbers; it is not a horizontal line.

threshold to extract the addresses mapped to the same bank.

Double-sided Hammering

Once we identify the contiguous memory within a bank, we can start taking three rows at a time from this memory and apply double-sided Rowhammer on them. We hammer the top and bottom row and expect the bit-flips in the middle row. In our experiments, we have kept the number of hammers equal to 10^6. While keeping the record of the vulnerable rows, we keep moving onto the next three rows until for our identified contiguous memory. We have used the typical Rowhammer instruction sequences without the `mfence` as shown in Listing 5.1. Without the `mfence`, the number of bit-flips are more as compared to with `mfence`. This is because the DRAM accesses become

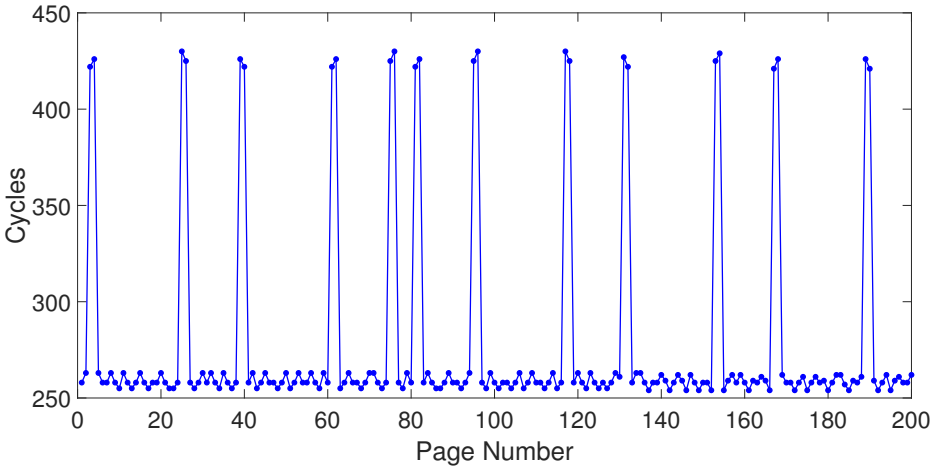


Figure 5.2: When two DRAM rows are accessed which reside in the same bank, we get a peak due to the row conflict. A threshold can be set to separate these rows using this side-channel information. In our experiments, we have set the `THRESHOLD_ROW_CONFLICT` value as 380 cycles.

Listing 5.1: Typical Rowhammer instruction sequence [36]

```
loop:
    movzx rax, BYTE PTR [rcx]
    movzx rax, BYTE PTR [rdx]
    clflush BYTE PTR [rcx]
    clflush BYTE PTR [rdx]
    mfence
    jmp loop
```

faster which results in quicker leakage of the charge stored in the memory cells. The number of flips with and without `mfence` are compared in Figure 5.3. The number of CPU cycles and the time taken by one Rowhammer instruction sequence is given in Table 5.1.

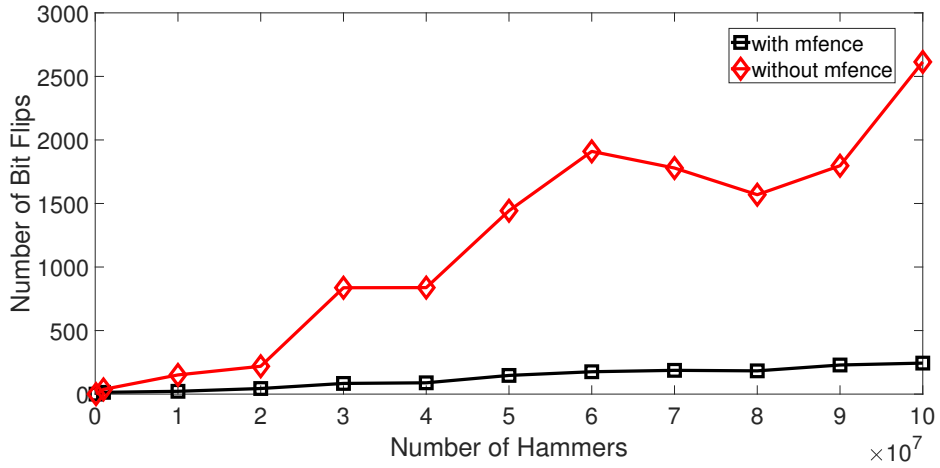


Figure 5.3: The number of bit-flips in 1 MB of memory in a DRAM bank out of the 8 MB contiguous chunk spread across 8 banks as a function of the number of hammers. The number of bit-flips increases with the number of hammers and without mfence sequence gives much more bit-flips. Approximately 0.03% of the DRAM cells are found to be vulnerable to Rowhammer attack on the DRAM model we profiled.

Table 5.1: CPU cycles and time taken by a typical Rowhammer instruction sequence on our platform.

Instruction Sequence (mV)	CPU Cycles	Time (μs)
With mfence	635	0.18
Without mfence	480	0.14

5.3.5 Online Phase

As the Rowhammer attack is highly reproducible, we first place the victim into our target vulnerable location inside the memory and repeat the double-sided Rowhammer attack by hammering the neighboring addresses. This induces the bit-flips in the actual victim, and faulty signatures are produced by the victim in response. The online phase consists of two steps, first is the

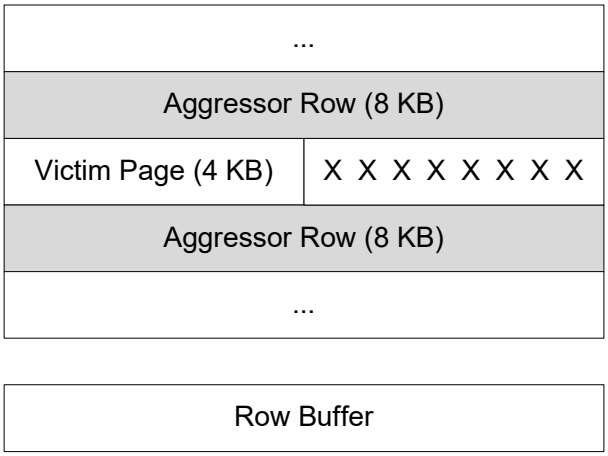


Figure 5.4: Victim placement and double-sided Rowhammer. To flip the bits from $1 \rightarrow 0$ inside the victim page, the attacker rows are needed to be filled with all zeros and for $0 \rightarrow 1$ flips, the attacker rows must be filled with all ones. Empirically, cells which flip both ways are very rare. Hence, a $0 \rightarrow 1$ flip may not happen in a $1 \rightarrow 0$ bad cell and vice versa.

victim placement and the second is the double-sided Rowhammer.

Victim Placement

Once the attacker finds vulnerable DRAM rows, it frees the row using `munmap`. Now it can either wait for the victim page to take that space in the memory or use standard techniques like spraying [67, 151, 181], grooming [173] or memory waylaying [65, 107, 182] to force the victim to come at the target address. We achieve this by repeatedly mapping the secret key s_1 of the victim until it lands on the target page as shown in Figure 5.4. The physical addresses are checked using the `pagemap` file.

Double-sided hammering

When the victim is mapped to the attacker's desired vulnerable memory location, the attacker can now apply the double-sided Rowhammer again. While the victim is signing the messages, the attacker now hammers the same rows that she found in the offline phase but this time it flips the bits in the victim process. This is because of the fact that Rowhammer effect is highly reproducible which means if you have found the vulnerable cells once, their values can be flipped again later. Finally, the victim starts producing the faulty signatures due to the bit-flips in the secret key which are collected by the attacker.

When a bit is flipped on the MSB side of s_1, it is likely that the rejection sampling condition in step 15 of Algorithm 3 repeatedly becomes true or takes too many iterations to output a faulty signature. This can create a *denial of service* scenario and can cause the victim to stuck in a loop and never output a signature unless the victim is moved to another memory location in the DRAM, making our attack harder. To counter this situation, we have set a limit on κ in Algorithm 3 to prevent the victim from going into an infinite loop. However, if there is a side-channel attack running in parallel is collecting side information, this scenario can be useful as the nonce y is changing in each iteration.

5.4 Experimental Results

In this Section, first we mention our Rowhammer experimental setup and then mention the results of our Signature Correction Attack experiments[1].

5.4.1 Experimental Setup

All the Rowhammer experiments are performed on a Haswell system with Intel(R) Core(TM) i7-4770 CPU @ 3.4GHz with 2 GB Samsung DDR3 part number M378B5773DH0-CH9. We have used Haswell because the AVX2 support start from Haswell and it also supports DDR3 memories. Our underlying operating system is Ubuntu 16.04 LTS.

We have performed all the post-processing on a Skylake system with Intel(R) Core(TM) i5-6440HQ CPU @ 2.60GHz having 8 GB DDR4 memory running Ubuntu 16.04 LTS using only a single core. The post-processing performance can be improved using multicore, GPUs or distributed computing.

5.4.2 Key recovery with Signature Correction Attack

We have successfully applied Rowhammer on s_1 of size 1024×32 bits for the AVX2 implementation of the Dilithium security level 2. After collecting 6,853 single-bit faulty signatures in 2.19 hours of online Rowhammer attack, we have recovered 3,735 unique bits of secret key s_1 using our Signature Correction algorithm as shown in Figure 5.5. Note that, the faults we can

[1]The source code for Signature Correction Attack is made available at `http://github.com/VernamLab/SignatureCorrection`.

inject are far from uniform. In fact, there are locations that are unflippable. The spatial bias is highly dependent on the technology of the DRAM. In our target DRAM (M378B5773DH0-CH9), we observed heavy spatial correlations (dark vertical stripes in Figure 5.5). Also rejection sampling prevents faulty signatures with flips at higher locations to be released. Hence, even if we force s_1 to relocate in memory as explained in Section 5.3.5, this does not allow recovery of all s_1 bits. We start recovering the same key bits and while others that wander through unflippable locations are never recovered. Therefore, we stop the online phase and do post-processing after all flippy locations are recovered. Among the 3,735 recovered bits, 2,454 are the 0's (green pixels) and 1,281 are the 1's (red pixels). Each sub-figure represents an element (polynomial) of s_1 up to $l = 4$ for Dilithium security level 2. Each polynomial has 256 coefficients on y-axis and 32 bits per coefficient on the x-axis. Every faulty signature gives one bit of secret key. The difference of 3,118 bits is because of the repetition of the faults at the same memory location as the attacker has no control over the locations within the s_1. 883 out of these 3,735 bits reside in the first three LSBs which should contain the actual key information. The rest of the bits from bit 4 to bit 32 are redundant, same as bit 3.

However, as the remaining bits from bit 4 to bit 32 are all same as bit 3 for each coefficient, if any of the bits are recovered from this region, we can consider it a bit recovery for LSB 3. This increases our useful bit recovery number significantly from 883 to 1,522 bits. Finally, we can say that by

Table 5.2: Post computation times for Signature Correction Attack on a single CPU. These offline computations can be performed on a distributed system or GPUs for performance improvement.

AVX2 Implementations	Average CPU Cycles (1 Verification)	Time (Sec) (1 Signature Correction)
dilithium2	36595	0.094
dilithium3	70397	0.267
dilithium5	67719	0.263
dilithium2-AES	28901	0.071
dilithium3-AES	47614	0.177
dilithium5-AES	49479	0.200

analyzing the positions of recovered bits in the coefficient, we can increase the number of recovered bits from 1,522 to 1,851, see Section 5.5.2 and Section 5.5.3 for details. As a summary, we have successfully recovered 1,851 bits out of the total 3,072 bits of s_1, 3-bits each of 1024 coefficients. The results and distribution of recovered bits up to the secret key coefficients is provided in Table 5.8.

Table 5.2 shows the offline computation time needed to trace one bit of secret key for all the variants of Dilithium. These timings are for the worst case scenario of $2 \times B \times L \times n$ verification as explained in Section 5.3. The search is however stopped earlier once a bit is located. We have computed the post-computation times for all variants but demonstrated the Rowhammer attack on only Dilithium security level 2. However our Signature Correction Attack is applicable to all variants, modes and security levels of Dilithium Round 3, where modes are randomized and deterministic, variants are SHAKE and AES and the security levels 2, 3 and 5.

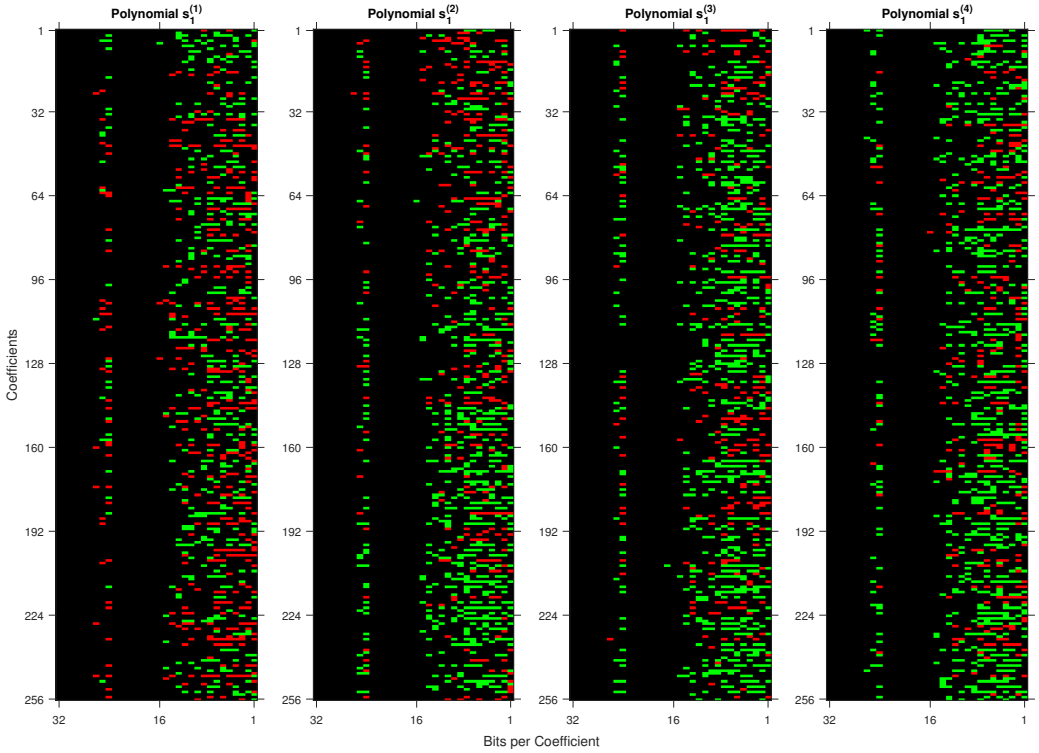


Figure 5.5: Recovered bits of secret key s_1 for Dilithium (security level 2). 3,735 in total with 2,454 0's (green pixels) and 1,281 1's (red pixels).

5.5 Estimating the Diminished Security Level of Dilithium

5.5.1 Lattice Security with Reduced Dimension

The Signature Correction Attack can be used iteratively to recover the secret key-bits. There are however two caveats in applying Signature Correction in practice:

- Each Signature Correction recovers only one secret key bit. For full-key

recovery we need at least 1024×3 unique faulty signatures which is rather time-consuming.

- As described below in practice we inject faults using Rowhammer, which prevents precise targeting of bits. Thus, we need many more Signature Correction iterations (and time consuming page re-allocations) in practice.

To overcome both problems, we instead opt to recover only a fraction of the key-bits to diminish the security level of Dilithium to a point where the remaining key bits can be recovered using lattice attacks.

Here we estimate the new security level of Dilithium by exploiting the recovered bits by Signature Correction Attack. Briefly, Dilithium is based on the hardness of the MLWE and MSIS problems under the Strong Unforgeability under Chosen Message Attack (SUF-CMA) model. We follow the cost estimation approach of [4, 49], i.e., the MLWE problem is analyzed as an LWE problem and the security level is estimated using standard lattice hardness estimation techniques. Specifically, we base our estimate on the so-called primal and dual attacks [34, 149] and use BKZ for lattice reduction. Cost estimation of the attacks are given in [4, 49]. Note that these estimates ignore SVP oracle calls. Instead, core-SVP hardness which is the cost of one call to an SVP oracle in dimension b is taken into account. For Quantum attacks, the Sieve algorithm is used to estimate the core hardness of underlying open problem. For the quantum sieve algorithm the heuristic complexity is

$\sqrt{3/2}^{b+O(b)} \approx 2^{0.292b}$ [11, 108]. Grover's quantum search algorithm reduces the complexity down to $2^{0.265b}$ [109, 110]. Cost of solving SVP in classical attack bound is $2^{0.2075b} \approx 2^{39}$ for the best-known algorithm [34].

The Signature Correction Attack allows us to recover certain number of bits of the private key s_1. By analyzing the distribution of the recovered bits, we can recover additional bits. We follow the general methodology from [4, 22] to analyze the reduced security of Dilithium with side information recovered using the Signature Correction Attack as described in Section 5.3. The reduced security levels can be determined by following the analysis in [49, 4, 22]. The analysis converts the equation system into an LWE instance of dimensions $256 \cdot l$ and $256 \cdot k$ by taking the coefficients of polynomial elements in the MLWE problem as the vectors of coefficients in LWE problem. Hence, the problem is reduced to finding the coefficient vectors

$$s_1 \ , \ s_2 \in \left(\mathbb{Z}^{256 \cdot l} \times \mathbb{Z}^{256 \cdot k}\right)$$

from $\bar{A}$ and coefficient vector of t. Here $\bar{A} \in \mathbb{Z}_q^{256 \cdot k \times 256 \cdot l}$ is obtained by replacing all entries $a_{ij} \in R_q$ of A by the rotation of the coefficient vectors of a_{ij}. One can show that the private key coefficients recovered using the Signature Correction Attack can be used to reduce the dimension n of the lattice formed by the equation system

$$As_1 + s_2 = t \tag{5.1}$$

by inserting the recovered coefficients of the secret key into its polynomial form. Moreover, inserting the recovered bits which are not used to find the coefficients can also reduce the norm of the coefficient vector. The security estimates for the scheme reduced for a given number of coefficients recovered using Signature Correction is given in Table 5.9. From the table, we deduce that recovering 8 coefficients of secret key reduces the attack complexity to around half of the overall complexity on average. In other words, recovering approximately 320 coefficients by Signature Correction Attack is enough to reduce the attack complexity to a practical level, i.e. 80 bits. Note that, we take the norm of secret key coefficients as $\zeta = \sqrt{10}$. Estimated time to recover is given in Section 5.4.2. [2]

To recover 320 coefficients, we need 960 bits in the same coefficients. Therefore, we cannot conclude that any 960-bit recovery is enough to break the scheme since we do not have any control on the location of the recovered bits.

5.5.2 Exploiting the Redundant Encoding to Recover More Coefficients

In this section, we focus on how we can use the recovered bits in the most effective way to diminish the security level of Dilithium. For this purpose, we divide the coefficients of the secret key polynomial into 3 groups:

[2] The script "scripts/PQsecurity.py" which estimates the cost of primal and dual attacks can be found at [4].

- **Group 1** has the fully recovered coefficients, i.e, 3 out of 3 bits are known in the coefficients. Number of recovered coefficients in Group 1 can directly be used to **reduce the dimension** n of the LWE system (Section 5.5.2).

- **Group 2** consists the coefficients in which 1 or 2 out of 3 bits are known in each coefficient. The recovered bits in this category fall short in reducing the LWE dimension further, yet they can still be used to **reduce the norm** of the secret key coefficients, i.e, unique SVP solution in LWE system (Section 5.5.3).

- **Group 3** is the collection of coefficients with no recovered bits. hence yielding no information about the coefficients.

When we estimate the security level, our calculations are based on the number of recovered coefficients and the norm of the remaining unknown coefficients. Secret key is defined as an l dimensional vector of n^{th} degree polynomials with coefficients in the range $[-\eta, \eta]$. In our experiments, we consider Dilithium security level 2 in which parameters are set to $\eta = 2$, $l = 4$ and $n = 256$ [49], i.e., Each coefficient of the secret key is in $\{-2, -1, 0, 1, 2\}$ but is encoded in the reference implementation as 32-bit words $\{1\cdots1110, 1\cdots1111, 0\cdots0000, 0\cdots0001, 0\cdots0010\}$, respectively. Therefore we have a highly redundant representation, where the per coefficient entropy of secret key encoding is only 2.25 bits (in 32 logical bits). The recovered bits are distributed over the last three bits of 1024 coefficients

Table 5.3: Recovering an additional bit by using recovered 2-bit info by Rowhammer. Shaded rows has the additional bit recovery, i.e., full coefficient is recovered by 2-bit info.

Known bits of xyz	Possible coefficients	# of possible coefficients
$00z$	$00z$	2
$01z$	010	1
$10z$	N/A	0
$11z$	$11z$	2
$0y0$	$0y0$	2
$0y1$	001	1
$1y0$	110	1
$1y1$	111	1
$x00$	000	1
$x01$	001	1
$x10$	$x10$	2
$x11$	111	1

Table 5.4: Number of additional full coefficient recoveries by 2-bit info. Highlighted bit shows the additional bit recovery.

xyz	Rec Coeffs	$s_1^{(1)}$	$s_1^{(2)}$	$s_1^{(3)}$	$s_1^{(4)}$	Total
01**z**	01**0**	10	11	6	5	32
0**y**1	0**0**1	12	5	6	8	31
1**y**0	1**1**0	7	7	6	9	29
1**y**1	1**1**1	10	9	5	11	35
x00	**0**00	1	0	0	0	1
x01	**0**01	0	0	0	0	0
x11	**1**11	0	0	0	1	1
Total # of Rec Coeffs		**40**	**32**	**23**	**34**	**129**

Table 5.5: Distribution of 1,522 bits recovered by Signature Correction Algorithm to the # secret key in polynomial coefficients. Total of 99 coefficients are recovered with another 857 coefficients yielding only partial information.

	$s_1^{(1)}$	$s_1^{(2)}$	$s_1^{(3)}$	$s_1^{(4)}$	**#bits**	**#coefs**
No recovery	19	14	18	17	0	**68**
1 bit rec	122	126	131	110	489	**489**
2 bits rec	95	89	86	98	736	**368**
Full rec	20	27	21	31	297	**99**
Total	**372**	**385**	**366**	**399**	**1522**	**1024**

given in Figure 5.5. Additionally, distribution of number of recovered bits up to the coefficients is given in Table 5.5.

Even though the recovered 1,522 bits are expected to give us information for about 507 coefficients, just 99 coefficients fully recovered, since only 297 out of 1,522 bits are concentrated in 99 coefficients. The remaining 1,423 bits are distributed over the remaining 857 different coefficients. On the other hand, 2-bit recovered in any coefficient yields either 0 or 1 bits of information on a coefficient as summarized in Table 5.3. Here coefficients are represented by xyz where z denotes the least significant bit (LSB) of the coefficient, and x represents the most significant bit (MSB) if we represent the coefficients by the last three bits. All higher order bits will be identical to x, i.e. the sign bit of the coefficient. In certain cases, with a 2-bit information of a coefficient we can recover the full 3-bit coefficient as shown in Table 5.3. For instance, if we recovered the first two bits as in the case of $01z$, then due to the encoding the only possible value z can take is 0. We can fully recover a coefficient from 2-bits of information in 7 out of 12 cases as shown in the

Table 5.6: Recovering an additional bit by using 1-bit recovered by Rowhammer. Shaded rows yield an extra bit.

Known bit of xyz	Possible coefficients	# of possible coefficients
$1yz$	$11z$	2
$0yz$	$00z$ or 010	3
$x1z$	$11z$ or 010	3
$x0z$	$00z$	2
$xy1$	001 or 111	2
$xy0$	$x10$ or 000	3

Table 5.7: Number of additional bit recovery by 1-bit info. Highlighted bit shows the recovered bit.

xyz	Rec Coeffs	$s_1^{(1)}$	$s_1^{(2)}$	$s_1^{(3)}$	$s_1^{(4)}$	Total
1yz	1**1**z	51	46	56	37	190
x0z	**0**0z	2	1	2	0	5
xy1	**xx**1	2	1	1	1	5
Total # of Rec bits		**55**	**48**	**59**	**38**	**200**

shaded rows in Table 5.3. With this approach, we managed to recover an additional 129 coefficients of the secret key as summarized in Table 5.4. The total number of recovered coefficients is increased significantly, i.e. from 99 to 228. You can find the number of recovered coefficients in Table 5.8.

5.5.3 Reducing the Norm of the Coefficients

In cases where we recover 1-bit out of a coefficient the information is not sufficient to recover the entire coefficient. However, we can still gain information useful in reducing the attack complexity. Specifically, we can reduce

the norm of the target vector by removing known bits from it. This reduces the complexity of the lattice search problem.

Further in certain cases the 1-bit knowledge may facilitate recovery of an additional bit of the coefficient. In Table 5.6, these special cases are shown in shaded rows. Analyzing the experimentally recovered bits gives us 200 of these special cases, i.e., two bits of 200 coefficients are recovered by 1-bit information. In Table 5.7, the number of coefficients in which extra bit recovery is possible is shown. By analyzing the recovered bits, we recovered 1 bit of 289 coefficients and 2 bits of 439 coefficients. There are 68 coefficients that we have no extra information about. Number of recovered bits and coefficients by Rowhammer and extra bit recovery method is given in Table 5.8. When we insert these recovered bits into the Lattice formulation the norm of secret key coefficients in the reduced system is decreased to

$$\zeta = \frac{68}{796} \times 3 + \frac{289}{796} \times 2 + \frac{439}{796} \times 1 = 1.53392.$$

By analyzing the encoding (Section 5.4.2), we increased the number of recovered bits from 883 to 1,522. This was achieved by taking recovered bits from 4 to 32 as the sign bit, i.e. x. Then we further increased from 1,522 to 1,851 by considering the positions in the recovered bits in the last 3 bits of the coefficient. In total, the number of fully recovered coefficients are increased from 99 to 228, and the number of coefficients with 2 bits known are increased from 368 to 439. By analyzing the encoding, we partially or

fully recovered 956 coefficients of 1024 secret key coefficients, in total. The breakdown is given in Table 5.8.

The diminished security level of Dilithium with the fully recovered coefficients (reduced dimension $\bar{n}$) and reduced average norm ζ is listed in Table 5.9. Note that with the fully recovered coefficients the reduced security level is 124-bits for classical and 112-bits for quantum attackers. By also exploiting the encoding to increase the fully recovered coefficients from 99 to 228 and partially recovered coefficients to reduce the norm from $\zeta = \sqrt{10}$ to $\zeta = 1.53392$, we managed to significantly degrade the security level: **89-bits (classical) and 81-bits (quantum)**.

5.6 Discussion

5.6.1 Is the weakness inherent to Dilithium?

In our attack we exploited the linear structure of Step 13 in the Dilithium Signing Algorithm:

$$z \leftarrow y + c \cdot s_1 \ .$$

To this end, we compute and check possible fault patterns as they would appear as additive terms in the faulty signature $\bar{z}$. This approach is enabled by the *linearly additive* secret mask y and the publicly known challenge vector c. Clearly, the presented signature correction algorithm is specific to Dilithium. However, we have also tried to produce a similar technique in the GeMSS

Table 5.8: Recovered Information by Signature Correction up to the number of coefficients. Highlighted rows show the number of coefficients with additional bit recovery.

	$s_1^{(1)}$	$s_1^{(2)}$	$s_1^{(3)}$	$s_1^{(4)}$	**#coefs**
Group 3: Coefficients with **no bit** recovery. **68** coefficients in total.					
No recovery	19	14	18	17	**68**
Group 2: Coefficients with **1** bit recovery. **289** bits in **289** coefficients in total.					
1 bit rec by 1 bit	67	78	72	72	**289**
Group 2: Coefficients with **2** bit recovery. **878** bits in **439** coefficients in total.					
2 bits rec by 1 bit	55	48	59	38	**200**
2 bits rec by 2 bits	55	57	63	64	**239**
Group 1: Full coefficient recovery. **684** bits in **228** coefficients in total.					
Full Coef rec by 2 bits	40	32	23	34	**129**
Full Coefs rec by 3 bits	20	27	21	31	**99**
Total#recbits(1851)	**467**	**465**	**448**	**471**	**1024**

Table 5.9: The reduced security level of Dilithium using the Signature Correction Attack. The value $\bar{n}$ denotes the reduced lattice dimension, b the block dimension of BKZ, and m the number of samples. Cost is given in log base 2 and is the smallest cost for all possible choices of m and b. Shaded rows show improvements: 124-bits (classical) and 112-bits (quantum) with plain Signature Correction, 89-bits (classical) and 81-bits (quantum) by also exploiting the encoding in addition to Signature Correction.

Dilithium Security Level II (128 bit) parameters: $q = 2^{23} - 2^{13} + 1$, $n = 1024$										
			Primal Attack				Dual Attack			
#Rec coeffs	$\bar{n}$	ζ	m	b	Classical	Quantum	m	b	Classical	Quantum
0	1024	$\zeta = \sqrt{10}$	1090	485	141	128	1089	484	141	128
0	1024	$\zeta = 1.53392$	1001	429	125	113	1027	428	125	113
Reduced Complexities with # Recovered coefficients and Reduced Norm										
1	1023	$\zeta = \sqrt{10}$	1129	484	141	128	1132	483	141	128
2	1022	$\zeta = \sqrt{10}$	1075	484	141	128	1074	483	141	128
4	1020	$\zeta = \sqrt{10}$	1062	483	141	128	1062	482	140	127
8	1016	$\zeta = \sqrt{10}$	1089	480	140	127	1090	479	140	127
64	960	$\zeta = \sqrt{10}$	1025	446	130	118	1037	445	130	118
99	925	$\zeta = \sqrt{10}$	981	425	124	112	997	424	124	112
128	896	$\zeta = \sqrt{10}$	933	408	119	108	947	407	119	107
192	832	$\zeta = \sqrt{10}$	919	369	107	97	885	369	107	97
228	796	$\zeta = \sqrt{10}$	863	348	101	92	843	348	101	92
288	736	$\zeta = \sqrt{10}$	799	313	91	83	788	313	91	83
320	704	$\zeta = \sqrt{10}$	744	295	86	78	810	294	86	78
352	672	$\zeta = \sqrt{10}$	745	276	80	73	742	276	80	73
99	925	$\zeta = 1.53392$	902	375	109	99	957	374	109	99
228	796	$\zeta = 1.53392$	782	306	89	81	773	306	89	81

[30] and Rainbow [45] schemes which gave insufficient partial information. While the approach is generic, the particulars of the signing algorithm may still make it hard to trace the fault to the output without causing the search space to grow exponentially, thus preventing efficient signature correction. While the presented attack utilizes faulty signatures to recover secret key bits we have also exploited the highly redundant encoding of the coefficients to gain significant advantage in reducing the security level of Dilithium. This weakness is not rooted in the algorithm itself, but rather due to the choice of representation used in the implementation.

5.6.2 Further Reducing the Attack Complexity

Dachman-Soled et al. [40] introduced a framework for cryptanalysis of lattice-based schemes when side-information in the form of "hints" about the secret and/or error is available. The framework allows the primal lattice reduction attack and allows progressive integration of hints before running a lattice reduction step. What we refer to as "recovered coefficient" and "partially recovered coefficient" correspond to "Modular hints" and "Approximate hints", respectively. Along with the framework the authors introduced techniques for progressively sparsifying the lattice, projecting onto and intersecting with hyperplanes, and/or altering the distribution of the secret vector. One may apply these more advanced techniques to gain advantage and further degrade the security level.

5.7 Countermeasures

Every novel attack sheds light onto how to strengthen a cryptographic scheme, and in this perspective, a discussion on countermeasures is very important. We can find considerable work on countermeasures against fault attacks on PQC schemes [28, 146, 50, 17]. In particular, Bindel *et al.* [19] have written an exhaustive literature review on countermeasures for fault attacks on lattice-based signature schemes.

For our Signature Correction Attack, there are two ways to detect and prevent the fault attack. First, we can prevent or detect the fault injection mechanism, which means that we would prevent or detect Rowhammer faults. Second, we can prevent or detect the exploitation of an injected fault. This requires an algorithmic countermeasure, such as preventing faulty signatures from being returned by the signer. Algorithmic countermeasures are required because our attack is independent of the fault mechanism used. In the following, we discuss the Rowhammer countermeasures followed by algorithmic countermeasures. Then, we provide a literature review of countermeasures against implementation attacks on lattice-based signature and encryption schemes in Table 5.10. In this table, we have shown countermeasures that work against timing, cache and fault attacks with a green tick mark and which don't work with a red cross mark. We show that post-quantum schemes are broadly vulnerable to three kinds of fault attacks, DFA, Instruction Skip and single-bit trace analysis. The table describes how countermeasures help

against these known attacks, which include an attack on an older round-2 PQ scheme [126] as well as our proposed Signature Correction.

5.7.1 Rowhammer Countermeasures

We discuss two approaches to counter Rowhammer attack. One technique detects the Rowhammer attack through hardware monitors, while the second technique prevents Rowhammer from happening in the first place.

Hardware Performance Counters (HPC) HPCs are special purpose registers that store the hardware events inside the CPU like cache hits and cache missed. As the Rowhammer bypasses the cache and directly hits the DRAM, there will be a significant increase in the number of cache misses which can be used to detect the Rowhammer attack. These HPCs, when paired with machine learning techniques, can detect Rowhammer attack with high accuracy. Gulmezoglu *et al.* [69] have shown an accuracy of 100% using the performance counter *LLC_Miss*.

Increasing DRAM Refresh Rate DDR3 and DDR4 specifications require that each DRAM row must be refreshed after at least 64ms to retain its values [158]. However, as we have seen this refresh rate is not sufficient in Rowhammer scenarios where hammering the neighboring rows cause the cells to leak faster than normal and are unable to retain their charge. So, an immediate mitigation can be to decrease the refresh interval to 32ms or

Table 5.10: An Overview of Countermeasures against Implementation Attacks on Lattice-Based Post-Quantum Cryptography; ✓Countermeasure works, ✗Countermeasure doesn't work

Countermeasures	**Implementation Attacks**					
			Fault			
	Timing [152, 155, 41]	**Cache** [27, 137, 152, 18]	**DFA** [31, 28]	**Instruction Skip** [147, 28, 146, 17, 50, 180, 169, 93]	**Quantum Hammer** [126] (LUOV Round2)	**Signature Correction Attack** (this work)
Constant run-time & data-oblivious accesses [27]	✓	✓	✗	✗	✗	✗
Key-independent control flow & memory accesses [152]	✗	✓	✗	✗	✗	✗
Nonce Randomization [28, 146]	✗	✗	✓	✓	✗	✗
Temporal Redundancy [31, 146, 92]	✗	✗	✓	✓	✗	✗
Spatial Redundancy [31, 92]	✗	✗	✓	✗	✓	✓
Verify-after-sign [31, 146]	✗	✗	✓	✓	✓	✓
HPC [69]	✗	✗	✗	✗	✓	✓
DRAM Refresh Rate [128]	✗	✗	✗	✗	✓	✓

16ms. Many systems allow this configuration from the BIOS for better memory stability. However, there are two downsides for this approach. The first one is that the power consumption will increase and the second one is the reduction of data transfer rate. This is because while the cells are refreshed, the data can not be read or written. Also, the Rowhammer can not be fully mitigated by this countermeasure. At most, one can significantly reduce the chances of getting bit-flips. Mutlu *et al.* [128] have shown that to fully mitigate Rowhammer using the refresh rate, one needs to set the refresh rate as 8.2 ms which is 7.8 times lower than 64 ms. This will cause significant burden on power consumption and quality of service which researchers are already trying to improve [32, 117].

5.7.2 Algorithmic Countermeasures

Here, we discuss algorithmic countermeasures for PQC signature schemes, specifically Dilithium against general fault attacks as well as our Signature Correction Attack. These countermeasures include adding randomization, temporal and spatial redundancy techniques and verify-after-sign approach.

Randomized Signing Due to the fault attacks based on determinism like [28, 146], Dilithium added this mitigation in Round 2 for DFAs as listed in step 6 of Algorithm 3. Here, the nonce y is generated randomly instead of generating using the message M, recommended for side-channel and fault attacks [49]. The idea is that if the attacker can not collect the faulty and

correct signature pair for the same message, the DFA will not work. However, this mitigation will not work for our Signature Correction Attack as it is independent of the nonce y. Whatever the value of the y is, we get the same error in the faulty signature depending upon the position of the fault in secret key s_1.

Temporal Redundancy Temporal Redundancy requires re-execution of same task after a certain amount of time and comparing their results. If they don't match then the output of the algorithm is disabled in order to prevent the attacker from accessing any information from the faulty signature. It makes it harder for the attacker to inject the same fault in redundant computations. However, as the Rowhammer attack faults the memory and can induce permanent faults, if multiple signatures are generated using the same faulty secret key in memory, they will still match, fault remains undetected. An algorithm with such a countermeasure can provide fault tolerance against transient faults but not permanent faults. Hence, it is recommended to add spatial redundancy.

Spatial Redundancy Spatial Redundancy involves simultaneous execution of N instances of a critical task for N level of redundancy and comparing their results for fault detection. If we store redundant copies of the original secret key during the key generation process at different memory locations, they can be used in parallel computations of signature generation using spatial redundancy technique. To bypass this approach, the attacker will need

to fault the same exact bits at both memory locations which will be much harder because every cell in the DRAM is not faulty. An important point here is that for deterministic version of Dilithium, this approach can work in a straightforward manner because the same nonce y is generated for the same message signed twice. However, for the randomized version, we will also need to store a copy of the nonce y for redundant computation. On the performance side, computation based on spatial redundancy will have significant overhead than the normal computation because of the increased complexity of the algorithm. Therefore, the level of spatial redundancy needed to detect faults should be taken into consideration.

Verify-after-Sign If there is any bit-flip in the secret key s_1 of Dilithium, it will generate a faulty signature which will not be verified by the verification algorithm. Hence, the verification can be used as a fault detection mechanism and if done on the signing side, the sender can easily detect the existence of the attacker. As compared to double signing, this approach is approximately three times faster as the verification algorithm takes less time as compared to the signing algorithm. There is still a possibility that the attacker also faults the verification in a way that results in a valid signature but to the best of our knowledge, there has not been such an algorithm developed so far for Dilithium. This approach may also fail if the comparison instruction is skipped using an instruction skip fault similar to [146]. Verify-after-sign can also detect instruction skip faults on signing step 13 and the

rejection sampling step 15 in Algorithm 3 [146]. Rejection sampling step is critical because if it is bypassed without a mitigation in place, it can reveal information about the secret key [49].

5.7.3 Applicability on Glitching Attacks

It would be worth investigating these countermeasures against glitching attacks. Glitching attacks are capable of faulting both instructions and data by introducing physical disturbances. Glitches can be injected by changing the supply voltage, optical probing with lasers, clock glitches or introducing an electromagnetic pulse. Precise glitches may lead to instruction skipping attacks which can be applied to bypass for example kernel's signature check [43] or to disable secure boot [39, 166]. Glitches have been leveraged to compromise gaming consoles such as XBOX 360 [6], Playstation 3 [111], Playstation Vita [119], Nintendo Switch [148, 58, 72] and tracking devices such as Apple's AirTag [71].

5.8 Conclusion

In this chapter, we have proposed the Signature Correction Attack targeting Dilithium a Round 3 finalist in the NIST PQC competition. The attack requires single bit-flips in the secret key vector, which we have implemented using Rowhammer targeting the constant-time AVX2 implementation of Dilithium. By analyzing the faulty signatures and exploiting redun-

dancy in the secret key encoding, our attack successfully recovered 1,851 bits (out of 3,072 bits) of the secret key. This enabled us to reduce the post-quantum security level to 81-bits (quantum) and 89-bits (classical) for both primal and dual lattice attacks. The attack is also applicable to other variants and security levels. We have demonstrated the first fault attack which works on randomized as well as deterministic versions of Dilithium. Our Signature Correction Attack is independent of the fault mechanism but we have used Rowhammer to demonstrate the attack as it is a software only attack and does not need physical access. This can be very critical in case of cloud scenarios where the attacker can launch an attack remotely and leak secret information by using only faulty signatures. We have shown how few bits of secret key significantly reduce the security strength of Dilithium using the lattice attacks especially when the secret key encoding is exploited. Further, randomizing the nonce, a countermeasure commonly implemented in the design of signature schemes to thwart fault attacks, is not sufficient in practice as demonstrated by our attack.

Chapter 6

Plundervolt Attack on Dilithium

Apart from Rowhammer attack, Plundervolt and V0LTpwn come in the category of software-induced fault attacks. Although Intel has released a patch for the attacks relying on undervolting, it is still important to see the effect of these attacks on post-quantum schemes. Only the software access to modify the `MSR` register has been taken which is used to undervolt. If the system is undervolted through hardware or the undervolting gets re-enabled by an attack in future or a post-quantum scheme is running on a different hardware than Intel, we should know the effects of undervolting on these schemes. In this chapter, we have successfully induced fault attacks in AVX2 implementation of Dilithium in Round 3.

6.1 Contributions

We have demonstrated that a well-known software-induced fault attack Plundervolt is capable of producing faulty signatures in Dilithium. Although Plundervolt is patched through software, the root cause still exists. In this chapter:

1. We demonstrate computation faults in Dilithium signature scheme and successfully collected faulty signatures at different CPU voltage levels using Plundervolt.

2. We have discovered another interesting behavior of Plundervolt which affects the memory `write` operations because of the out-of-order execution of a branch.

6.2 Plundervolt Attack on Dilithium

6.2.1 Threat Model

For the Plundervolt attack, the attacker and the victim both need to reside on the same machine. The microcode and BIOS update for Plundervolt should not be installed and the attacker must have the root access to modify the `MSR` register values in order to undervolt the system. The assumption of root access can be made in case of TEE like Intel SGX where the adversary should not be able to get any secret information from inside the enclave even if the

operating system is compromised. The attacker should be able to collect the messages signed by the victim.

6.2.2 Experimental Setup

First we needed to find a system that is vulnerable to the Plundervolt attack and checked if the undervolting is working. A list of CPUs has been given by Intel [82]. Then we reproduced the Plundervolt faults in multiplications which required tuning some parameters like operands, start voltage, end voltage, iterations etc. Our Plundervolt system is Kaby Lake with Intel(R) Core(TM) i7-7700 CPU @ 3.6GHz. It has a 16GB DDR4 memory and the operating system is Ubuntu 20.01.1 LTS. The microcode version is `0xde`, CPU ID `906E9` and the BIOS version 1.0.4.

6.2.3 Finding crash points

After setting up a system for Plundervolt, the first thing to do is to estimate the crash points of the system. This is important because we should remain below the crash point during our experiments to avoid any system halts. As the operating voltage of the CPU varies by varying the operating frequency, we have found the crash points for the whole range of our CPU starting from 0.8GHz to 3.6GHz as shown in Figure 6.1. Once we have found the crash points, we can safely operate on any frequency by undervolting the CPU while keeping some margin with the crash point. If the system gets halted,

the only solution is to hard reboot by pressing the power button. Sometimes, the system just reboots at the crash point but then it is still operating on the same voltage and crashes again. While working remotely, we use MSNSwitch Remote Power Switch (Model 622B) to reboot the machine.

6.2.4 Temperature Variations

As mentioned in the previous research [125, 96], the CPU temperature has an effect on the faults and the crash points. In our experiments, if a freshly booted system has a CPU temperature of 40°C, it increases up to 45°C while running the experiments which affects the results. If the weather is cold and the window is left open, it is hard to find any fault. We kept the windows shut and relied on room's temperature control to minimize the temperature variations.

6.2.5 Experimental Results

The attacker starts undervolting the system while the victim is running the Dilithium AVX2 signing operations and expect the faulty signatures. Under the current setup, the attacker starts undervolting the system from -170mV while the signing operations are running in parallel by the victim. For -170mV undervolting, 4 out of 1 million signatures get faulted as depicted in Table 6.1. Then the attacker starts reducing the CPU voltage by 2mV step at a time. As the operating voltage gets lower and lower, the number of faults

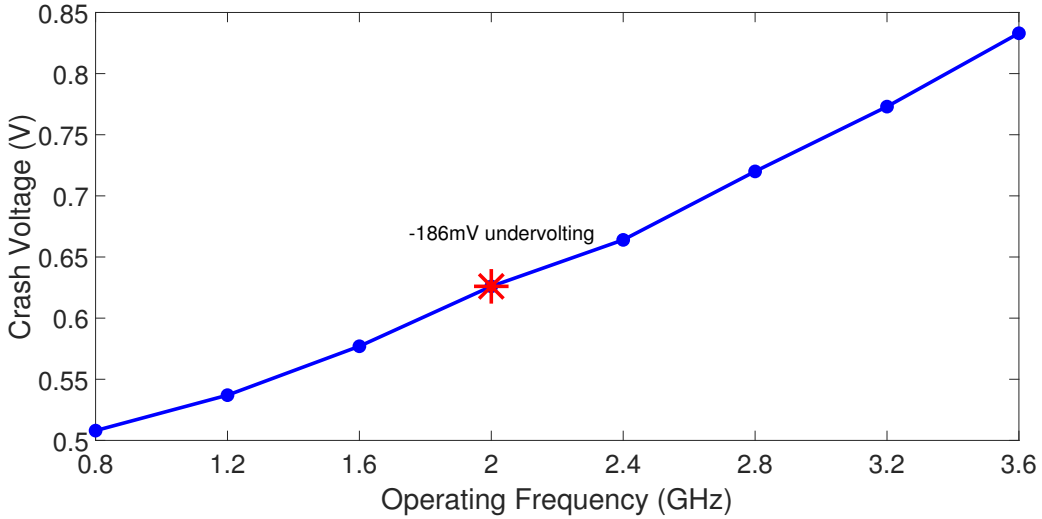


Figure 6.1: The crash voltage increases when the operating frequency is set to a higher value. It also depends upon the CPU temperature. Here the CPU temperature was ranging from 40-45°C. The temperature variations are hard to control, hence the results may vary if tried to reproduce. In our experiments, the operating frequency is always set to 2GHz.

increases. At -182mV however, the system crashes which is 4mV below the crash point corresponding to 2GHz frequency in Figure 6.1. This is due to the dilithium signing process is running, which raises the crash point. The more processes are running on a system, the more the crash point is affected.

6.3 Novel Observation of Plundervolt

Apart from the known fault behaviors in multiplications and AES-NI extensions in Plundervolt attack [125] or the AVX instruction sequences in V0LTpwn [96], we have discovered a special instruction sequence that is capable of faulting the `write` operations. The C code of this sequence is listed

Table 6.1: Number of faulty signatures in Dilithium. In our experiments, we have set the operating frequency to 2GHz and the corresponding voltage is set automatically by DVFS to 0.81V. Blue values till -176mV represent that it is safe to operate on these voltages whereas we do experience crashes beyond that (red values).

Undervolting (mV)	Number of Faulty Signatures out of 10^6
-170	4
-172	25
-174	53
-176	183
-178	281
-180	10306
-182	crash

in Listing 6.1 and its assembly version can be seen in Appendix C.1 for deeper understanding.

In the code sequence in Listing 6.1, we have declared a character array of size 8,000,000. We are writing characters 'a', 'b' and 'c' in a for loop in each element of the array on the same index in a for loop. Then we check if the last value which is 'c' is correctly written or not. This procedure is run in a while loop and should run forever under normal scenario as the branch should never be taken. However, we see that due to undervolting, the code does return the character 'b' which is very interesting.

To understand this behavior, we performed several experiments. First, we put an `mfence` before the *if* statement to make sure that it does not execute out of order. As a result, we did not see any faults. This indicates that the

Listing 6.1: Novel Observation with Undervolting

```
 1: int main() {
 2:     char arr[SIZE];
 3:     while (1) {
 4:       for (int i = 0; i < SIZE; i++) {
 5:         arr[i] = 'a';
 6:         arr[i] = 'b';
 7:         arr[i] = 'c';
 8:         if(arr[i] != 'c') {
 9:           return arr[i];
10:         }
11:       }
12:     }
13: }
```

if statement is executed out of order before the character 'c' is written in the array. Then we turned off the undervolting inside the *if* statement and printed out *arr[i]* multiple times to see if the character 'c' gets written after some delay but it did not. Character 'b' was permanently stored in the *arr[i]* causing a permanent memory violation. We also flushed the cache line to make sure that the character 'b' is not just being read from the cache but it is actually written in the DRAM. Further, we removed the *if* statement and just returned the sum of all the array elements and verified if the sum is equal to the expected value or not. It was always equal indicating that the fault is occurring due to the out-of-order execution of the branch. We repeated the experiments for `malloc` instead of an array and found similar results. Finally, other than `je` we tried other variants of conditional jumps like `jle`, `jg` and `jne` and found that we get faults with all of them.

6.4 Conclusion

In this chapter, we have demonstrated successful fault injection in Dilithium signature scheme using Plundervolt attack and collected the faulty signatures. Plundervolt is a software-induced fault mechanism but different in nature as compared to Rowhammer. It induces faults in the computations such as multiplications, unlike memory faults in case of Rowhammer. We have discovered another interesting behavior of Plundervolt affecting `write` operations, which stems from the out-of-order execution of a branch.

Chapter 7

Conclusion

In this dissertation, we have investigated randomized post-quantum signature schemes from the NIST's Post-Quantum Cryptography Standardization Process against fault attacks. The research has identified a number of vulnerabilities and practically demonstrated successful key recovery attacks on these schemes. We have utilized Rowhammer attack as our fault mechanism and utilized the faulty signatures to algebraically recover the secret key bits. In all of our Rowhammer experiments, we have used SPOILER for contiguous memory detection. SPOILER is a hardware vulnerability, we discovered in Intel processors. Intel has released a public advisory (INTEL-SA-00238) and assigned CVE (CVE-2019-0162).

The SPOILER vulnerability stems from the dependency resolution logic that serves the speculative loads. It leaks 8-bits of physical address information from the userspace without any special privileges. We have demon-

strated how this information is useful in microarchitectural attacks such as Prime+Probe attack. Software-induced fault attacks such as Rowhammer attack requires contiguous memory which we locate using SPOILER. We have demonstrated double-sided Rowhammer without HugePages and `pagemap` with just normal user privileges. We have identified a number of DRAM chips vulnerable to Rowhammer attack and developed a software tool to induce bit-flips in these chips. As the DRAM is shared between multiple processes in cloud environments, process isolation is very critical. However, Rowhammer has the capability to cross the process boundaries by inducing bit-flips in the neighboring DRAM rows. There have been a number of mitigations such as TRR and ECC but the Rowhammer problem has not been completely solved so far.

With the rise of quantum threat, NIST is running a Post-Quantum Cryptography Standardization Process for KEMs and Digital Signatures. Apart from algorithmic security, they are giving significant importance to side-channel and fault attacks. In this dissertation, we have proposed and demonstrated a novel bit-tracing attack for a multivariate signature scheme, LUOV (round 2 finalist). The main idea of bit-tracing is to use single-bit faulty signatures and mathematically trace back the location of the fault. We have used Rowhammer attack to practically induce the bit-flips in the signing process of LUOV. We are able to fully recover the secret of LUOV signature scheme by bit-tracing attack, followed by an analytical approach. We call this hybrid approach as QuantumHammer. We have investigated a sim-

ilar approach on Dilithium (round 3 finalist), a completely different scheme based on lattices. We have successfully identified a technique to trace the faults in Dilithium and we call it Signature Correction Attack. The attack significantly reduces the security strength of the scheme from 2^{128} to 2^{81}. Finally, we have successfully injected faults in Dilithium with another software-induced fault mechanism Plundervolt.

We conclude that only algorithmic security is not sufficient for the security of the scheme. Fault injection or side-channel attacks, if combined with algebraic attacks may lead to successful hybrid attacks. Apart from independently analyzing the algorithm and its implementation, there is a need to investigate in a hybrid fashion.

Bibliography

[1] Jeffery M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, and Paul D Madland. Method and apparatus for performing a store operation, April 23 2002. US Patent 6,378,062.

[2] Jeffrey M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, Paul D Madland, David B Papworth, and Michael A Fetterman. Method and apparatus for dispatching and executing a load operation to memory, February 10 1998. US Patent 5,717,882.

[3] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the second round of the nist post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2020.

[4] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key Exchange—A new hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, Austin, TX, August 2016. USENIX Association.

[5] Daniel Apon, Ray Perlner, Angela Robinson, and Paolo Santini. Cryptanalysis of ledacrypt. In *Annual International Cryptology Conference*, pages 389–418. Springer, 2020.

[6] Free60 Wiki archive. *The Xbox 360 reset glitch hack*. `https://free60.org/Reset_Glitch_Hack/`.

[7] Amund Askeland and Sondre Rønjom. A side-channel assisted attack on ntru. *Cryptology ePrint Archive*, 2021.

[8] Jedec Solid State Technology Association. *Low Power Double Data Rate 4 (LPDDR4)*, January 2020. `https://www.jedec.org/standards-documents/docs/jesd209-4b`.

[9] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. Anvil: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices*, 51(4):743–755, 2016.

[10] Shi Bai and Steven D Galbraith. An improved compression technique for signatures based on learning with errors. In *Cryptographers' Track at the RSA Conference*, pages 28–47. Springer, 2014.

[11] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, page 10–24, USA, 2016. Society for Industrial and Applied Mathematics.

[12] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 75–92, Berlin, Heidelberg, 2014. Springer.

[13] Daniel J Bernstein. Cache-timing attacks on aes, 2005.

[14] Ward Beullens. Breaking rainbow takes a weekend on a laptop. *Cryptology ePrint Archive*, 2022.

[15] Ward Beullens and Bart Preneel. Field lifting for smaller uov public keys. In *International Conference on Cryptology in India*, pages 227–246. Springer, 2017.

[16] Ward Beullens, Alan Szepieniec, Frederik Vercauteren, and Bart Preneel. Luov: Signature scheme proposal for nist pqc project.

[17] Nina Bindel, Johannes Buchmann, and Juliane Krämer. Lattice-based signature schemes and their sensitivity to fault attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 63–77. IEEE, 2016.

[18] Nina Bindel, Johannes Buchmann, Juliane Krämer, Heiko Mantel, Johannes Schickel, and Alexandra Weber. Bounding the cache-side-channel leakage of lattice-based signature schemes using program semantics. In *International Symposium on Foundations and Practice of Security*, pages 225–241. Springer, 2017.

[19] Nina Bindel, Juliane Kramer, and Johannes Schreiber. Special session: hampering fault attacks against lattice-based signature schemes-countermeasures and their efficiency. In *2017 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–3. IEEE, 2017.

[20] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 37–51. Springer, 1997.

[21] Jonathan Bootle, Claire Delaplace, Thomas Espitau, Pierre-Alain Fouque, and Mehdi Tibouchi. Lwe without modular reduction and improved side-channel attacks against bliss. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 494–524. Springer, 2018.

[22] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from lwe. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1006–1018, 2016.

[23] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[24] Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir, and Bo-Yin Yang. Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 203–218. Springer, 2010.

[25] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 117–130, Vancouver, BC, 2017. USENIX Association.

[26] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 224–235, New York, NY, USA, 2018. ACM.

[27] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 323–345. Springer, 2016.

[28] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 21–43, 2018.

[29] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. *arXiv preprint arXiv:1811.05441*, 2018.

[30] Antoine Casanova, Jean-Charles Faug'ere, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret, and Jocelyn Ryckeghem. *GeMSS: A Great Multivariate Short Signature*. PhD thesis, PhD thesis, UPMC-Paris 6 Sorbonne Universités, 2017.

[31] Laurent Castelnovi, Ange Martinelli, and Thomas Prest. Grafting trees: a fault attack against the sphincs framework. In *International Conference on Post-Quantum Cryptography*, pages 165–184. Springer, 2018.

[32] Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti, Alaa R Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. Improving dram performance by parallelizing refreshes with accesses. In *2014*

IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pages 356–367. IEEE, 2014.

[33] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.

[34] Yuanmi Chen and Phong Q. Nguyen. Bkz 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 1–20, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[35] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

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

[37] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 55–71. IEEE, 2019.

[38] Jonathan Corbet. Defending against Rowhammer in the kernel, October 2016. `https://lwn.net/Articles/704920/`.

[39] Ang Cui and Rick Housley. {BADFET}: Defeating modern secure boot using {Second-Order} pulsed electromagnetic fault injection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[40] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. Lwe with side information: attacks and concrete security estimation. In *Annual International Cryptology Conference*, pages 329–358. Springer, 2020.

[41] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. Timing attacks on error correcting codes in post-quantum schemes. In *Proceedings of ACM Workshop on Theory of Implementation Security Workshop*, TIS'19, page 2–9, New York, NY, USA, 2019. Association for Computing Machinery.

[42] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized many-sided rowhammer attacks from JavaScript. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1001–1018. USENIX Association, August 2021.

[43] Eric DeBusschere and Mike McCambridge. Modern game console exploitation. *Technical Report, Department of Computer Science, University of Arizona*, 2012.

[44] Jintai Ding, Joshua Deaton, Kurt Schmidt, Vishakha, and Zheng Zhang. Cryptanalysis of the lifted unbalanced oil vinegar signature scheme. Cryptology ePrint Archive, Report 2019/1490, 2019. `https://eprint.iacr.org/2019/1490`.

[45] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *International conference on applied cryptography and network security*, pages 164–175. Springer, 2005.

[46] Jintai Ding, Bo-Yin Yang, Chia-Hsin Owen Chen, Ming-Shing Chen, and Chen-Mou Cheng. New differential-algebraic attacks and reparametrization of rainbow. In *Applied Cryptography and Network Security*, pages 242–257. Springer, 2008.

[47] Jintai Ding, Zheng Zhang, Joshua Deaton, Kurt Schmidt, and F Vishakha. New attacks on lifted unbalanced oil vinegar. In *The 2nd NIST PQC Standardization Conference*, 2019.

[48] Jack Doweck. Inside intel® core microarchitecture. In *Hot Chips 18 Symposium (HCS), 2006 IEEE*, pages 1–35. IEEE, 2006.

[49] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 238–268, 2018.

[50] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Loop-abort faults on lattice-based fiat-shamir and hash-and-sign signatures. In *International Conference on Selected Areas in Cryptography*, pages 140–158. Springer, 2016.

[51] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on bliss lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1857–1874, 2017.

[52] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 40:1–40:13, Piscataway, NJ, USA, 2016. IEEE Press.

[53] Agner Fog. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering*, pages 02–29, 2012.

[54] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru. *Submission to the NIST's post-quantum cryptography standardization process*, 36(5), 2018.

[55] P. Frigo, E. Vannacc, H. Hassan, V. der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi. Trrespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762, 2020.

[56] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU*, page 0, Washington, DC, USA, 2018. IEEE, IEEE Computer Society.

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

[58] Andreas Galauner. *Glitching the Switch*, June 2018. `https://media.ccc.de/v/c4.openchaos.2018.06.glitching-the-switch`.

[59] Aymeric Genêt, Matthias J Kannwischer, Hervé Pelletier, and Andrew McLauchlan. Practical fault injection attacks on sphincs. *IACR Cryptology ePrint Archive*, 2018:674, 2018.

[60] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *International Conference on Applied Cryptography and Network Security*, pages 83–102. Springer, 2018.

[61] Mohsen Ghasempour, Mikel Lujan, and Jim Garside. Armor: A runtime memory hot-row detector, 2015.

[62] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, London, 2004.

[63] Aurélien Greuet, Simon Montoya, and Guénaël Renault. Attack on lac key exchange in misuse situation. In *International Conference on Cryptology and Network Security*, pages 549–569. Springer, 2020.

[64] Lov K Grover. A fast quantum mechanical algorithm for database search. *arXiv preprint quant-ph/9605043*, 1996.

[65] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261. IEEE, 2018.

[66] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 368–379, New York, NY, USA, 2016. ACM.

[67] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.

[68] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[69] Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning. *arXiv preprint arXiv:1907.03651*, 2019.

[70] Berk Gulmezoglu, Andreas Zankl, M Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. Undermining user privacy on mobile devices using ai. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019.

[71] hackaday. *APPLE AIRTAG SPILLS ITS SECRETS*, May 2021. `https://hackaday.com/2021/05/10/apple-airtag-spills-its-secrets`.

[72] hackaday. *EM-GLITCHING FOR NINTENDO DSI BOOT ROMS*, September 2021. `https://hackaday.com/2021/09/11/em-glitching-for-nintendo-dsi-boot-roms`.

[73] Lars T Hansen. Shared memory: Side-channel information leaks, 2016.

[74] Yasufumi Hashimoto, Tsuyoshi Takagi, and Kouichi Sakurai. General fault attacks on multivariate public key cryptosystems. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, pages 1–18, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[75] N Herath and A Fogh. These are not your grand daddys cpu performance counters–cpu hardware performance counters for security. *Black Hat Briefings*, 2015.

[76] Sebastien Hily, Zhongying Zhang, and Per Hammarlund. Resolving false dependencies of speculative load instructions, October 13 2009. US Patent 7,603,527.

[77] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[78] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[79] IBM. Ibm chipkill memory: Advanced ecc memory for the ibm netfinity 7000 m10, 2019. `http://ps-2.kev009.com/pccbbs/pc_servers/chipkilf.pdf`.

[80] Mehmet Sinan İnci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 368–388, Berlin, Heidelberg, 2016. Springer.

[81] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual.

[82] Intel. *Intel® Processors Voltage Settings Modification Advisory*, October 2019. `https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html`.

[83] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 591–604, Washington, DC, USA, 2015. IEEE Computer Society.

[84] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *2015 Euromicro Conference on Digital System Design (DSD)*, pages 629–636. IEEE, 2015.

[85] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Mascat: Preventing microarchitectural attacks before distribution. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, page 377–388, New York, NY, USA, 2018. Association for Computing Machinery.

[86] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative load hazards boost rowhammer and cache attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 621–637, Santa Clara, CA, August 2019. USENIX Association.

[87] Saad Islam, Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. Signature correction attack on dilithium signature scheme. *arXiv preprint arXiv:2203.00637*, 2022.

[88] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–392. ACM, 2016.

[89] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *2022 IEEE Symposium on Security and Privacy (SP)*, volume 1, 2022.

[90] Spec JEDEC. High bandwidth memory (hbm) dram. *JESD235*, pages 0018–9340, 2013.

[91] Daniel Kales and Greg Zaverucha. Forgery attacks on mqdssv2. 0, 2019.

[92] A. A. Kamal and A. Youssef. Strengthening hardware implementations of ntruencrypt against fault analysis attacks. *Journal of Cryptographic Engineering*, 3:227–240, 2013.

[93] Abdel Alim Kamal and Amr M. Youssef. Fault analysis of the ntrusign digital signature scheme. *Cryptography Commun.*, 4(2):131–144, June 2012.

[94] Emre Karabulut and Aydin Aysu. Falcon down: Breaking falcon post-quantum signature scheme through side-channel attacks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 691–696, 2021.

[95] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium*, pages 957–972, 2014.

[96] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0ltpwn: Attacking x86 processor integrity from software. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1445–1461, 2020.

[97] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural support for mitigating row hammering in dram memories. *IEEE Computer Architecture Letters*, 14(1):9–12, 2014.

[98] Il-Ju Kim, Taeho Lee, Jaeseung Han, Bo-Yeon Sim, and Dong-Guk Han. Novel single-trace ml profiling attacks on nist 3 round candidate dilithium. *IACR Cryptol. ePrint Arch.*, 2020:1383, 2020.

[99] Jeremie S Kim, Minesh Patel, A Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 638–651. IEEE, 2020.

[100] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[101] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 206–222. Springer, 1999.

[102] Aviad Kipnis and Adi Shamir. Cryptanalysis of the oil and vinegar signature scheme. In *Annual International Cryptology Conference*, pages 257–266. Springer, 1998.

[103] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael

Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[104] Steffen Kosinski, Fernando Latorre, Niranjan Cooray, Stanislav Shwartsman, Ethan Kalifon, Varun Mohandru, Pedro Lopez, Tom Aviram-Rosenfeld, Jaroslav Topp, Li-Gao Zei, et al. Store forwarding for data caches, November 29 2016. US Patent 9,507,725.

[105] Juliane Krämer and Mirjam Loiero. Fault attacks on uov and rainbow. In Ilia Polian and Marc Stöttinger, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 193–214, Cham, 2019. Springer International Publishing.

[106] Evgeni Krimer, Guillermo Savransky, Idan Mondjak, and Jacob Doweck. Counter-based memory disambiguation techniques for selectively predicting load/store conflicts, October 1 2013. US Patent 8,549,263.

[107] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[108] Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, pages 3–22, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[109] Thijs Laarhoven. *Search problems in cryptography: from fingerprinting to lattice sieving*. PhD thesis, Mathematics and Computer Science, February 2016. Proefschrift.

[110] Thijs Laarhoven, Michele Mosca, and Joop Pol. Finding shortest lattice vectors faster using quantum search. *Des. Codes Cryptography*, 77(2–3):375–400, December 2015.

[111] Nate Lawson. How the ps3 hypervisor was hacked, 2010. `https://rdist.root.org/2010/01/27/how-the-ps3-hypervisor-was-hacked/`.

[112] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in

sandboxed javascript. In *Computer Security – ESORICS 2017*, pages 191–209. Springer, 2017.

[113] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, 2016. USENIX Association.

[114] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

[115] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing rowhammer faults through network requests. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 710–719. IEEE, 2020.

[116] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 605–622, Washington, DC, USA, 2015. IEEE Computer Society.

[117] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. Raidr: Retention-aware intelligent dram refresh. *ACM SIGARCH Computer Architecture News*, 40(3):1–12, 2012.

[118] Errol L. Lloyd and Michael C. Loui. On the worst case performance of buddy systems. *Acta Informatica*, 22(4):451–473, Oct 1985.

[119] Yifan Lu and Davee. *Viva la Vita Vida: Hacking the most secure handheld console*, December 2018. `https://media.ccc.de/v/35c3-9364-viva_la_vita_vida`.

[120] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 598–616, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[121] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122. ACM, 2018.

[122] Intel 64 Architecture Memory Ordering White Paper. `http://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf`, 2008. Accessed: 2018-11-26.

[123] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations in SGX. In *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*, pages 21–44, 2018.

[124] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 69–90. Springer, 2017.

[125] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482. IEEE, 2020.

[126] Koksal Mus, Saad Islam, and Berk Sunar. Quantumhammer: A practical hybrid attack on the luov signature scheme. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1071–1084, 2020.

[127] Onur Mutlu. The rowhammer problem and other issues we may face as memory becomes denser. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1116–1121, 2017.

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

[129] NIST. Post-quantum cryptography standardization. `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization`, 2017.

[130] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1406–1418, New York, NY, USA, 2015. ACM.

[131] Aesun Park, Kyung-Ah Shim, Namhun Koo, and Dong-Guk Han. Side-channel attacks on post-quantum signature schemes based on multivariate quadratic equations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 500–523, 2018.

[132] Jacques Patarin. The oil and vinegar signature scheme. In *Dagstuhl Workshop on Cryptography September, 1997*, 1997.

[133] Mathias Payer. Hexpads: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*, pages 138–154. Springer, 2016.

[134] Colin Percival. Cache missing for fun and profit, 2005.

[135] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "make sure dsa signing exponentiations really are constant-time". In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1639–1650, New York, NY, USA, 2016. ACM.

[136] Ray Perlner and Daniel Smith-Tone. Rainbow band separation is better than we thought. *Cryptology ePrint Archive*, 2020.

[137] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To bliss-b or not to be: Attacking strongswan's implementation of post-quantum signatures. In *Computer and Communications Security*, pages 1843–1855. ACM, 2017.

[138] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security Symposium*, pages 565–581, 2016.

[139] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In *International Conference on*

Cryptology and Information Security in Latin America, pages 130–149. Springer, 2019.

[140] Peter Pessl and Lukas Prokop. Fault attacks on cca-secure lattice kems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 37–60, 2021.

[141] PQShield. Think openly, build securely post-quantum standards ready. `https://pqshield.com`, 2021.

[142] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 513–533. Springer, 2017.

[143] Salman Qazi, Yoongu Kim, Nicolas Boichat, Eric Shiu, and Mattias Nissler. Introducing half-double: New hammering technique for dram rowhammer bug, 2021.

[144] QuSecure. Scalable cybersecurity for the post-quantum enterprise. `https://www.qusecure.com`, 2021.

[145] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. Side-channel assisted existential forgery attack on dilithium-a nist pqc candidate. *IACR Cryptology ePrint Archive*, page 821, 2018.

[146] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. Exploiting determinism in lattice-based signatures: practical fault attacks on pqm4 implementations of nist candidates. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 427–440, 2019.

[147] Prasanna Ravi, Debapriya Basu Roy, Shivam Bhasin, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Number "not used" once-practical fault attack on pqm4 implementations of nist candidates. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 232–250. Springer, 2019.

[148] Gauvain Tanguy Henri Gabriel Roussel-Tarbouriech, Noel Menard, Tyler True, Tini Vi, et al. Methodically defeating nintendo switch security. *arXiv preprint arXiv:1905.07643*, 2019.

[149] C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66(2):181–199, September 1994.

[150] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: high-resolution microarchitectural attacks in javascript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267. Springer, 2017.

[151] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.

[152] Johanna Sepulveda, Andreas Zankl, and Oliver Mischke. Cache attacks and countermeasures for ntruencrypt on mpsocs: Post-quantum resistance for the iot. In *2017 30th IEEE International System-on-Chip Conference (SOCC)*, pages 120–125, 2017.

[153] K. Shim and N. Koo. Algebraic fault analysis of uov and rainbow with the leakage of random vinegar values. *IEEE Transactions on Information Forensics and Security*, pages 1–1, 2020.

[154] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[155] Joseph H. Silverman and William Whyte. Timing attacks on ntruencrypt via variation in the number of hash calls. In Masayuki Abe, editor, *Topics in Cryptology – CT-RSA 2007*, pages 208–224, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[156] Yongha Son. A note on parameter choices of round5. *IACR Cryptol. ePrint Arch.*, 2019:949, 2019.

[157] Yongha Son and Jung Hee Cheon. Revisiting the hybrid attack on sparse and ternary secret lwe. *IACR Cryptol. ePrint Arch.*, 2019:1019, 2019.

[158] JEDEC Standard. Double data rate (ddr) sdram specification. *JEDEC Solid State Technology Assoc*, 2005.

[159] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. In *International Conference on Selected Areas in Cryptography*, pages 14–37. Springer, 2016.

[160] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[161] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in iaas clouds. In *Network and Distributed Systems Security (NDSS) Symposium*. The Internet Society, 2018.

[162] Synopsys. *DDR5/4/3/2: How Memory Density and Speed Increased with each Generation of DDR*, April 2022. `https://blogs.synopsys.com`.

[163] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, Vancouver, BC, 2017. USENIX Association.

[164] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, 2018. USENIX Association.

[165] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[166] Niek Timmers and Albert Spruyt. Bypassing secure boot using fault injection. *Black Hat Europe*, 2016, 2016.

[167] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin. Spechammer: Combining spectre and rowhammer for new speculative attacks.

In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 1362–1379, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.

[168] M. Caner Tol, Saad Islam, Berk Sunar, and Ziming Zhang. An optimization perspective on realizing backdoor injection attacks on deep neural networks in hardware. *CoRR*, abs/2110.07683, 2021.

[169] Felipe Valencia, Tobias Oder, Tim Güneysu, and Francesco Regazzoni. Exploring the vulnerability of r-lwe encryption to fault attacks. In *Proceedings of the Fifth Workshop on Cryptography and Security in Computing Systems*, CS2 '18, page 7–12, New York, NY, USA, 2018. Association for Computing Machinery.

[170] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium. USENIX Association*, 2018.

[171] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2Nd Workshop on System Software for Trusted Execution*, SysTEX'17, pages 4:1–4:6, New York, NY, USA, 2017. ACM.

[172] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195. ACM, 2018.

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

[174] Pepe Vila, Boris Köpf, and José Francisco Morales. Theory and practice of finding eviction sets. *arXiv preprint arXiv:1810.01497*, 2018.

[175] Beullens Ward, Preneel Bart, Szepieniec Alan, and Vercauteren Fréderik. LUOV - MQ signature scheme, 2020. https://www.esat.kuleuven.be/cosic/pqcrypto/luov/.

[176] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):169–195, Jun. 2020.

[177] WikiChip. Ivy Bridge - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/ivy_bridge_(client). Accessed: 2019-02-05.

[178] WikiChip. Kaby Lake - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake. Accessed: 2019-02-05.

[179] WikiChip. Skylake (client) - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client). Accessed: 2019-02-05.

[180] Keita Xagawa, Akira Ito, Rei Ueno, Junko Takahashi, and Naofumi Homma. Fault-injection attacks against nist's post-quantum cryptography round 3 KEM candidates. *IACR Cryptol. ePrint Arch.*, page 840, 2021.

[181] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *USENIX Security Symposium*, pages 19–35, 2016.

[182] Lai Xu, Rongwei Yu, Lina Wang, and Weijie Liu. Memway: in-memorywaylaying acceleration for practical rowhammer attacks against binaries. *Tsinghua Science and Technology*, 24(5):535–545, 2019.

[183] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[184] Takanori Yasuda, Xavier Dahan, Yun-Ju Huang, Tsuyoshi Takagi, and Kouichi Sakurai. Mq challenge: Hardness evaluation of solving multivariate quadratic problems. *IACR Cryptology ePrint Archive*, 2015:275, 2015.

[185] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[186] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *Research in Attacks, Intrusions, and Defenses*, pages 118–140. Springer, 2016.

[187] Zhi Zhang, Jiahao Qi, Yueqiang Cheng, Shijie Jiang, Yiyang Lin, Yansong Gao, Surya Nepal, and Yi Zou. A retrospective and futurespective of rowhammer attacks and defenses on dram. *arXiv preprint arXiv:2201.02986*, 2022.

Appendix A

SPOILER

A.1 Tested Hardware Performance Counters

A list of tested performance counters is given in Table A.1.

A.2 Row conflict Side-Channel

The row conflict side-channel retrieves the timing information of the CPU while doing direct accesses (using `clflush`) from the DRAM. A higher timing indicates that the two addresses are mapped to the same bank in the DRAM because reading an address from the same bank forces the row buffer to copy the previous contents back to the original row and then load the newly accessed data into the row buffer. Whereas, a low timing indicates that two addresses are not in the same bank (not sharing the same row buffer) and

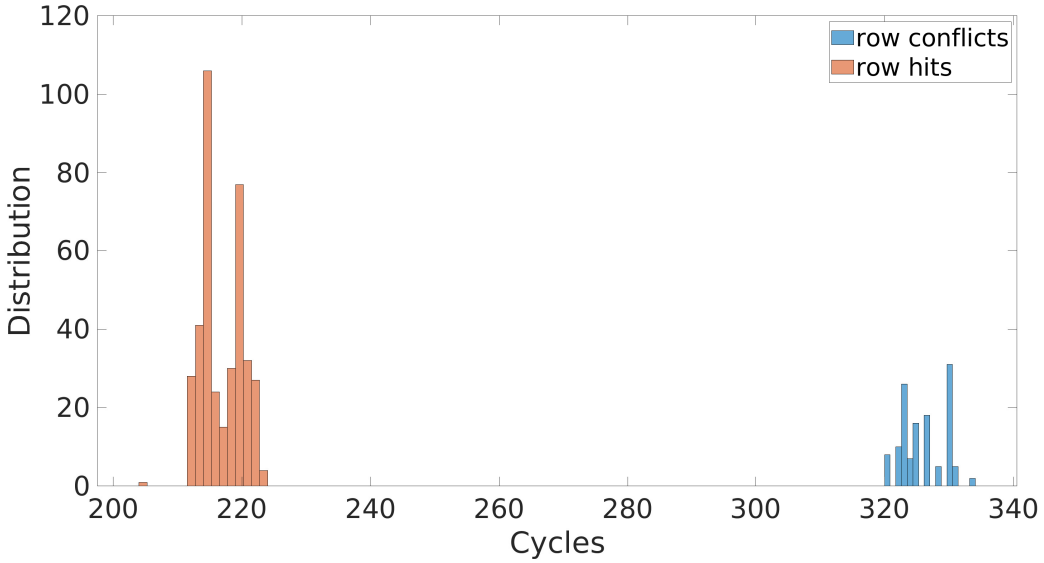


Figure A.1: Timings for accessing the aliased virtual addresses (random addresses where 20 LSB of the physical address match). Row hits (orange/low timings) are clearly distinguishable from row conflicts (blue/high timings).

are loaded into separate row buffers. Figure A.1 shows a wide gap (around 100 cycles) between row hits and row conflicts.

A.3 Memory Utilization and Contiguity

The probability of obtaining contiguous memory depends on memory utilization of the system. We conduct an experiment to examine the effect of memory utilization on availability of contiguous memory. In this experiment, 1 GB memory is allocated. During the experiment, the memory utilization of the system is increased gradually from 20% to 90%. We measure the probability of getting the contiguous memory with two methods. The first one

is checking the physical frame numbers from `pagemap` file to look for 520 kB of contiguous memory. The second method is using SPOILER to find the 520 kB of contiguous memory. This 520 kB is required to get three consecutive rows within a bank for a DRAM configuration having 256 kB row offset and 8 kB row size.

Figure A.2 and Figure A.3 show that when the memory has been fragmented after intense memory usage, it gets more difficult to allocate a contiguous chunk of memory. Even decreasing the memory usage does not help to get a contiguous block of memory. Figure A.3 depicts that after the memory utilization has been decreased from 70% to 60% and so on, there is not enough contiguous memory to mount a successful double-sided Rowhammer attack. Until the machine is restarted, the memory remains fragmented which makes a double-sided Rowhammer attack difficult, especially on targets like high-end servers where restarting is impractical.

The observed behavior can be explained by the *binary buddy allocator* which is responsible for the physical address allocation in the Linux OS [62]. This type of allocator is known to fragment memory significantly under certain circumstances [118]. The Linux OS uses a *SLAB/SLOB allocator* in order to circumvent the fragmentation problems. However, the allocator only serves the kernel directly. Userspace memory therefore still suffers from the fragmentation that the buddy allocator introduces. This also means that getting the contiguous memory becomes more difficult if the system under attack has been active for a while.

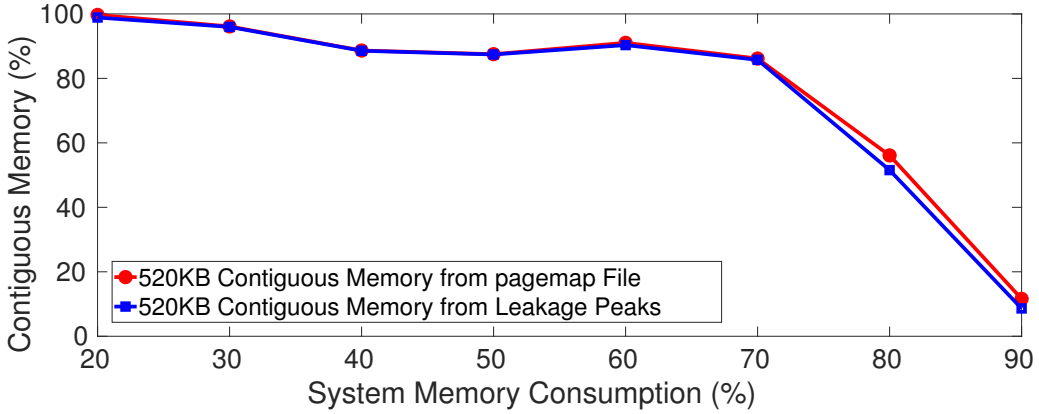


Figure A.2: Finding contiguous memory of 520 kB with increasing memory utilization. The overlap between the red and blue plot indicates the high accuracy of the contiguous memory detection capability of SPOILER as verified by the `pagemap` file.

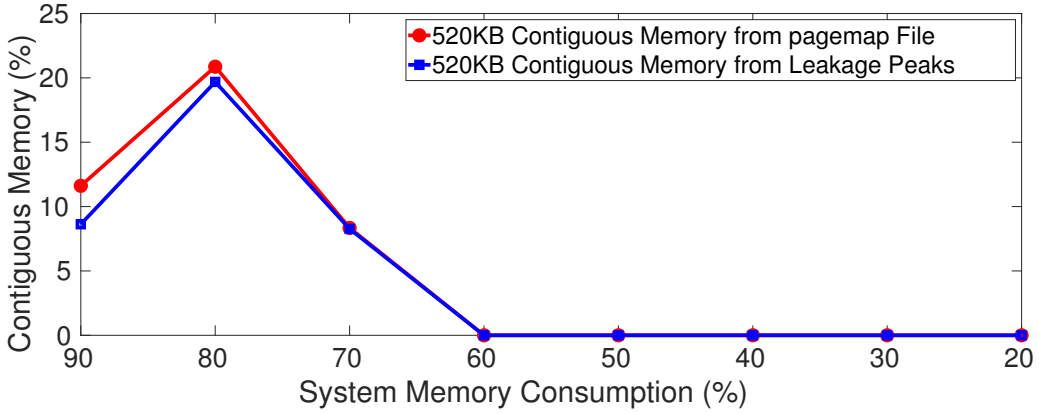


Figure A.3: Finding contiguous memory of 520 kB with decreasing memory utilization.

Counters	*Correlation*
UNHALTED_CORE_CYCLES	0.3077
UNHALTED_REFERENCE_CYCLES	0.1527
INSTRUCTION_RETIRED	0.2718
INSTRUCTIONS_RETIRED	0.2827
BRANCH_INSTRUCTIONS_RETIRED	0.3143
MISPREDICTED_BRANCH_RETIRED	0.0872
CYCLE_ACTIVITY:CYCLES_L2_PENDING	-0.0234
CYCLE_ACTIVITY:STALLS_LDM_PENDING	0.9819
CYCLE_ACTIVITY:CYCLES_NO_EXECUTE	0.2317
RESOURCE_STALLS:ROB	0
RESOURCE_STALLS:SB	-0.0506
RESOURCE_STALLS:RS	-0.0044
LD_BLOCKS_PARTIAL:ADDRESS_ALIAS	-0.9511
IDQ_UOPS_NOT_DELIVERED	-0.1455
IDQ:ALL_DSB_CYCLES_ANY_UOPS	0.0332
ILD_STALL:IQ_FULL	0.1021
ITLB_MISSES:MISS_CAUSES_A_WALK	0
TLB_FLUSH:STLB_THREAD	0
ICACHE:MISSES	0
ICACHE:IFETCH_STALL	0
L1D:REPLACEMENT	0.3801
L2_DEMAND_RQSTS:WB_HIT	0.2436
LONGEST_LAT_CACHE:MISS	0.0633
CYCLE_ACTIVITY:CYCLES_L1D_PENDING	-0.0080
LOCK_CYCLES:CACHE_LOCK_DURATION	0
LOAD_HIT_PRE:SW_PF	0
LOAD_HIT_PRE:HW_PF	0
MACHINE_CLEARS:CYCLES	0
OFFCORE_REQUESTS_BUFFER:SQ_FULL	0
OFFCORE_REQUESTS:DEMAND_DATA_RD	0.1765

Table A.1: Counters profiled for correlation test

Appendix B

QuantumHammer

B.1 Divide-and-Conquer Attack

Algorithm 10 MQ Generator

1: **procedure** MQ_Gen(i, x)

i: column# in $\mathcal{T}$

x: known elements in i^{th} column of $\mathcal{T}$

Output: $\mathcal{A}_i(x)$

2: $A_i \leftarrow GenMQ(i)$ ▷ use Equation 2.3 for $i = j$

3: $A_i(x) \leftarrow InsertVec(A_i, x)$ ▷ Section 4.4.2, Item 2

4: **return** $A_i(x)$

5: **end procedure**

Algorithm 11 ML Generator

1: **procedure** ML_GEN((i, x),(j, y))

i, j: column# in $\mathcal{T}$,

x: known elements in the x^{th} column of $\mathcal{T}$

y: known elements in the y^{th} column of $\mathcal{T}$

Output: $\mathcal{B}_{i,j}(x, y)$

2: $B_{i,j} \leftarrow EqnGen(i, j)$ ▷ Section 4.4.2, Item 3

3: $B_{i,j}(x, y) \leftarrow InsertVec(B_{i,j}, x, y)$ ▷ Section 4.4.2,Item 5

4: **return** $\mathcal{B}_{i,j}(x, y)$

5: **end procedure**

Algorithm 12 Equation Solver

1: **procedure** EQN_SOLVER($\mathcal{A}_i(x), \bigcup_{j=1}^{i-1} \mathcal{B}_{i,j}(x, y_j)$)

Output: t_j or **fail**

2: **if** $v - x \leq m$ **then**

Solve $\mathcal{A}_i(x)$ ▷ $\sim MQ(v - x, m)$

3: **else if** $v - x - (i-1)m \leq 0$ **then**

Solve $\bigcup_{j=1}^{i-1} \mathcal{B}_{i,j}(x, y_j)$ ▷ $\sim MQ((i-1)m, v - x)$

4: **else if** $0 \leq v - x - (i-1)m \leq m$ **then**

Solve $\mathcal{A}_i(x) \cup \bigcup_{j=1}^{i-1} \mathcal{B}_{i,j}(x, y_j)$ ▷ $\sim MQ(v - x - m, m)$

5: **else**

6: **return fail** ▷ Not a solvable system

7: **end if**

8: **return** t_j

9: **end procedure**

Algorithm 13 Coefficient Matrix Generator

1: **procedure** MATRIX_GEN(Q_1)

Output: P_k^3, Q_2

2: $Pk1 \leftarrow findPk1(Q_1, k)$ ▷ [16]

3: $Pk2 \leftarrow findPk2(Q_1, k)$ ▷ [16]

4: $Pk3 \leftarrow GenPk3(Pk1, Pk2, k)$ ▷ $\sim MQ(v-x, v-x)$

5: $Q_2 \leftarrow GenQ2(Pk3)$ ▷ Equation 2.4

6: **return** P_k^3, Q_2

7: **end procedure**

B.2 LUOV - Build Augmented Matrix

Algorithm 14 LUOV - Build Augmented Matrix

procedure BUILDAUGMENTED($C, L, Q_1, \mathcal{T}, h, v$)

Output: $LHS||RHS = (A||b)$

$RHS \leftarrow h - C - L_s(v||0)^T$

$LHS \leftarrow L \begin{pmatrix} -T \\ \\ 1_m \end{pmatrix}$

for k from 1 to m **do**

$P_{k_1} \leftarrow findPK1(k, Q_1)$

$P_{k_2} \leftarrow findPK2(k, Q_1)$

$RHS[k] \leftarrow RHS[k] - v^t P_{k,1} v$

$F_{k,2} \leftarrow (P_{k,1} + P_{k,1}^T)\mathcal{T} + P_{k,2}$

$LHS[k] \leftarrow LHS[k] + vF_{k,2}$

end for

return $LHS||RHS$

end procedure

Appendix C

Plundervolt

C.1 Assembly version of C code in Listing 6.1

Listing C.1: Assembly version of code sequence in Listing 6.1

```
 0:    endbr64
 4: push   rbp
 5: mov    rbp,rsp
 8: lea    r11,[rsp-0x7a1000]
10: sub    rsp,0x1000
17: or     QWORD PTR [rsp],0x0
1c: cmp    rsp,r11
1f: jne    10 <main+0x10>
21: sub    rsp,0x220
28: mov    rax,QWORD PTR fs:0x28
31: mov    QWORD PTR [rbp-0x8],rax
35: xor    eax,eax
```

```
37: mov    DWORD PTR [rbp-0x7a1214],0x0
41: jmp    a5 <main+0xa5>
43: mov    eax,DWORD PTR [rbp-0x7a1214]
49: cdqe
4b: mov    BYTE PTR [rbp+rax*1-0x7a1210],0x61
53: mov    eax,DWORD PTR [rbp-0x7a1214]
59: cdqe
5b: mov    BYTE PTR [rbp+rax*1-0x7a1210],0x62
63: mov    eax,DWORD PTR [rbp-0x7a1214]
69: cdqe
6b: mov    BYTE PTR [rbp+rax*1-0x7a1210],0x63
73: mov    eax,DWORD PTR [rbp-0x7a1214]
79: cdqe
7b: movzx  eax,BYTE PTR [rbp+rax*1-0x7a1210]
83: cmp    al,0x63
85: je     9e <main+0x9e>
87: mov    eax,DWORD PTR [rbp-0x7a1214]
8d: mov    rdx,QWORD PTR [rbp-0x8]
91: xor    rdx,QWORD PTR fs:0x28
9a: je     b8 <main+0xb8>
9c: jmp    b3 <main+0xb3>
9e: add    DWORD PTR [rbp-0x7a1214],0x1
a5: cmp    DWORD PTR [rbp-0x7a1214],0x7a11ff
af: jle    43 <main+0x43>
b1: jmp    37 <main+0x37>
b3: call   b8 <main+0xb8>
b8: leave    b9:  ret
```