LEAPFROG: The Rowhammer Instruction Skip Attack

Andrew Adiletta
MITRE
aadiletta@mitre.org

M. Caner Tol
Worcester Polytechnic Institute
mtol@wpi.edu

Kemal Derya
Worcester Polytechnic Institute
kderya@wpi.edu

Berk Sunar
Worcester Polytechnic Institute
sunar@wpi.edu

Saad Islam
Worcester Polytechnic Institute
saad.islam@fulbrightmail.org

***Abstract*—Since its inception, Rowhammer exploits have rapidly evolved into increasingly sophisticated threats compromising data integrity and the control flow integrity of victim processes. Nevertheless, it remains a challenge for an attacker to identify vulnerable targets (i.e., Rowhammer gadgets), understand the outcome of the attempted fault, and formulate an attack that yields useful results.**

In this paper, we present a new type of Rowhammer gadget, called a LeapFrog gadget, which, when present in the victim code, allows an adversary to subvert code execution to bypass a critical piece of code (e.g., authentication check logic, encryption rounds, padding in security protocols). The LeapFrog gadget manifests when the victim code stores the Program Counter (PC) value in the user or kernel stack (e.g., a return address during a function call) which, when tampered with, repositions the return address to a location that bypasses a security-critical code pattern.

This research also presents a systematic process to identify LeapFrog gadgets. This methodology enables the automated detection of susceptible targets and the determination of optimal attack parameters. We first show the attack on a decision tree algorithm to show the potential implications. Secondly, we employ the attack on OpenSSL to bypass the encryption and reveal the plaintext. We then use our tools to scan the Open Quantum Safe library and report on the number of LeapFrog gadgets in the code. Lastly, we demonstrate this new attack vector through a practical demonstration in a client/server TLS handshake scenario, successfully inducing an instruction skip in a client application. Our findings extend the impact of Rowhammer attacks on control flow and contribute to developing more robust defenses against these increasingly sophisticated threats.

1. Introduction

The miniaturization of DRAM technology has inadvertently increased the susceptibility to bit flips and reliability issues. To mitigate data corruption, DRAM rows are refreshed at regular intervals, typically every 64 milliseconds. However, Kim et al. [30] discovered that rapid and repeated access to adjacent rows could accelerate charge leakage, leading to bit flips before the scheduled refresh, a phenomenon known as the Rowhammer effect [30]. Expanding on this, Seaborn et al. [53] demonstrated an even more efficient method known as the double-sided Rowhammer, which exacerbates the issue.

Further developments in exploiting the Rowhammer vulnerability have been numerous. Gruss et al. [22] achieved root privileges by flipping opcodes in the `sudo` binary using a single-location hammering technique. Furthermore, Gruss et al. [23] and Ridder et al. [18] demonstrated the feasibility of launching Rowhammer attacks remotely via JavaScript. Tatar et al. [57] and Lip et al. [36] extended the reach of Rowhammer to network-based attacks. Its applicability has also been demonstrated in cloud environments [65], [14] and on hybrid FPGA-CPU platforms [64]. Importantly, Kwong et al. [34] revealed that Rowhammer poses not only an integrity threat but also compromises confidentiality.

Efforts to detect [27], [13], [69], [26], [45], [24], [5], [17] and neutralize [23], [61], [9] Rowhammer attacks have been substantial. However, Gruss et al. [22] demonstrated the ineffectiveness of these countermeasures. Moreover, Cojocar et al. [15] questioned the security of ECC as a countermeasure. The Target Row Refresh (TRR) hardware countermeasure was recently also circumvented, as shown by Frigo et al. [21] and further exploited by Ridder et al. [18] to target DDR4 chips with TRR. In particular, a recent study by Kogler et al. [32] highlighted the feasibility of hammering beyond adjacent locations to bypass TRR defenses.

The software exploits enabled by Rowhammer were further studied in recent research. Tobah et al. [58] introduced the notion of **Rowhammer gadgets** and a specialized attack. Specifically, if a victim code is designed to return benign data to an unprivileged user, and uses nested pointer dereferences, Rowhammer can be used to flip these pointers thereby gaining arbitrary read access in the victim's address space. Adiletta et al. [2] demonstrated that even internal CPU elements such as register values, which are occasionally saved to the stack, can be vulnerable to Rowhammer when they are temporarily stored in the stack and flushed to memory. Upon reloading, these corrupted values are returned to the registers, potentially leading to the execution of faulty stack variables and security breaches.

The threat of physical fault injection attacks has been acknowledged in the cryptographic community for some time [8]. For instance, `OpenSSL` incorporated error checks in CRT-based exponentiation early on to combat Bellcore attacks [8]. However, fault injection techniques have successfully compromised Elliptic Curve Parameters in the `OpenSSL` library [56]. Similarly, Rowhammer-

induced fault attacks in WolfSSL, leading to ECDSA key exposure, were revealed in [43], [19]. The vulnerabilities occurred during the TLS handshake process, involving the signing operation with private ECC keys. WolfSSL responded by introducing WOLF_SSL_CHECK_SIG_FAULTS and WOLFSSL_BLIND_PRIVATE_KEY, a series of checks during the signing stages to detect data tampering [40], [41].

A recent work by Adiletta et al. [2] targeted sensitive stack variables via Rowhammer threatening data integrity. In this paper, we instead target the control-flow-integrity (CFI) and subvert the execution flow for malicious ends, e.g. to bypass sensitive sections of code user authentication and data encryption. For this, we introduce LEAPFROG a new Rowhammer attack vector that targets the PC when stored in the stack during function calls and context switches. Not all PC manipulations will yield useful results, as some jumps within the code will result in errors, like segfaults, or simply will not bypass the intended code logic. To explore the massive attack surface, we introduce an automated tool that dynamically analyzes code to detect this type of Rowhammer gadget [58].

1.1. Our contributions

We introduce a novel approach for identifying LEAPFROG Rowhammer gadgets capable of corrupting the PC, utilizing a combination of GDB, the Intel Pintool, and the Linux Process Interface.

Our contributions are fivefold:

1) We introduce the concept of LEAPFROG gadgets, which allows an attacker to bypass security critical areas of code by faulting the PC value stored in stack.
2) We introduce the first simulation tool designed to identify LeapFrog gadgets. This tool represents an improvement over existing methodologies [67] by systematically analyzing binaries with our Intel Pin-based tool called *MFS* and incorporating time-domain analysis in simulations.
3) We scan the Open Quantum Safe library signature scheme, OpenSSL encryption, and a machine learning model - and quantify the number of potential LeapFrog gadgets present in the code.
4) We validate the feasibility of this attack in practical scenarios by successfully bypassing a TLS handshake in standard OpenSSL implementations.
5) We propose and evaluate countermeasures against the LeapFrog attack, offering insights into enhancing the resilience of systems against such advanced Rowhammer based exploits.

2. Background

Rowhammer DRAM is stored in an array architecture of memory cells, where a capacitor and a transistor form each cell in the array. Each cell, capable of storing a value of either 1 or 0, is connected along word lines that extend across the row. Additionally, bit lines intersect the word lines perpendicularly, linking them to each cell. When the bit lines are brought into opposition (where one goes high and one goes low), positive feedback from a sense amplifier sets the state of the cell to be high or low. The sense amplifier consists of two cross-connected inverters between the cells. For the cell to retain its state, the sense amplifier must be disconnected [1].

The sense amplifiers must be disconnected to read the cell, so the target word-line must be brought high. The charge in the capacitor will bring one of the bit lines high if the cell has a value of 1 during reading, then the sense amplifier will be reconnected, and the row will have the sense amplifier outputs latched [47].

The Rowhammer attack works by abusing the sense amplifier's ability to set the state of the cell. The capacitor will leak voltage and must undergo a refresh of voltage every 64 ms (or less) according to the standard JETEC convention. Reading the cell introduces noise into the system due to fluctuating voltages, and the noise can be amplified by the sense amp. As semiconductor technology improves, transistors shrink in size and the operating voltage also shrinks. The resulting cells are then able to operate at a higher speed, and the noise margins are reduced. Shrinking in size increases the signal coupling across traces and devices and magnifies crosstalk. When combined with the lower operating voltage and sharper edges due to higher speeds, the ratio of the crosstalk to the supply voltage increases significantly as technology improves making it easier for an adversary to exploit this type of attack. These can result in errors being written to the DRAM [1].

Instruction Skipping Instruction-skipping attacks are a type of fault attack that targets the normal execution flow of a program, particularly in embedded systems and secure circuits. Historically these techniques often needed physical access due to the timing and precision required to skip instructions. For example, laser fault injection has been demonstrated in inducing instruction skips in AES (Advanced Encryption Standard) that resulted in leaking private encryption keys [11]. In another example [51], researchers demonstrate how electromagnetic fault injection can effectively induce instruction skipping in the ARMv7-M architecture, specifically targeting AES.

In terms of countermeasures for instruction skips specifically, [42] addresses the vulnerability of embedded processors to instruction skip attacks. The paper acknowledges that while countermeasures based on temporal redundancy have been proposed, they are not entirely effective against double fault injections over extended time intervals.

Kernel Stack vs User Stack The user stack operates in user mode and is operated by user-level processes. Each user process has its own stack that stores local variables, function parameters, return addresses, and the control flow of the program. This stack is limited in size and is specific to the user space process, ensuring isolation and security from other processes.

On the other hand, the kernel stack operates in kernel mode, a privileged mode of operation for the system's kernel. Each thread of a process has its own kernel stack. This stack is used when the process executes system calls or when it is interrupted and the kernel needs to perform operations on behalf of the process. The kernel stack handles system-critical lower-level operations such as interrupt handling, system call implementation, and managing hardware interactions. It is kept separate from

the user stack for security and stability, ensuring that user processes cannot directly access or interfere with kernel-level operations. Data stored in the kernel stack includes CPU context for system calls, interrupt state information, and other kernel-specific data, while the user stack holds user-level process data like function calls and local variables. This separation reinforces the security and stability of the operating system by isolating user applications from the core kernel functions.

Program Counters Program Counters (PCs), also known as Instruction Pointers, hold the memory address of the next instruction to be executed by the CPU. This mechanism ensures that instructions are executed in the correct sequence. The value of the Program Counter is typically not stored in the stack; rather, it's stored in a dedicated register within the CPU. During the execution of a program, the PC is automatically incremented after each instruction is fetched, pointing to the subsequent instruction. However, during certain operations like function calls and interrupts, the PC value may be changed abruptly to a new address. In such cases, the return address (the original PC value) is often stored in the stack to enable the program to return to the correct point in the program after the operation is complete. For user function calls, the PC value is pushed to the user stack, but if there is an exception, signal handler, or system call, the PC gets pushed to the kernel stack. This mechanism facilitates the smooth flow of program execution.

Process Degradation Process degradation in computing refers to the intentional slowing down of a processor to create favorable conditions for certain types of attacks. A notable contribution in this field, HyperDegrade [3], combines previous approaches [4] with the use of simultaneous multithreading (SMT) architectures to significantly slow down processor performance, achieving a slowdown that is orders of magnitude greater than previous methods. It utilizes collateral Self Modifying Code (SMC) events to induce "machine clears", where the entire CPU pipeline is flushed, resulting in severe performance penalties. This process is triggered by cache line eviction, causing the invalidation of instructions in the victim's L1 instruction cache, which the CPU may interpret as an SMC event. This mechanism amplifies the degradation effect, as instructions are sometimes fetched multiple times, leading to substantial slowdowns in CPU performance. This slowdown enhances the time granularity for FLUSH+RELOAD [25] attacks, enabling more effective exploitation of side-channel vulnerabilities in systems. The attack not only explores the implementation of this technique but also investigates the root causes of performance degradation, particularly focusing on cache eviction. Their findings have substantial implications in the realm of cryptography, as evidenced by the amplification of the Raccoon attack [39] on TLS-DH key exchanges and other protocols.

3. Related Work

To attack the binary during runtime, we had to overcome timing challenges as well as different detection problems to find vulnerable areas in the code. In [22] researchers used `mmap` to map the target binary into a vulnerable page in memory, demonstrating how memory waylaying and memory-chasing techniques can force the mapped binary into the target page. This attack can potentially be mitigated by making the process execute only, and thus cannot be mapped with the `mmap` command. In contrast, our work can attack binaries that are unreadable from userland and are executed only. Additionally, our attack works on fundamentally different mechanics, so targets not susceptible to [22] may be susceptible to ours.

Another related work [58] demonstrates that code using nested pointer dereferences can corrupt bits in these pointers to reveal data to an unprivileged user. They demonstrate this vulnerability on `ioctl` given they can flood the kernel heap with data by spawning processes (a method they call "spraying"), increasing the probability a single bit-flip will point to malicious data in the heap that points to the location of secret data. Our work compliments and improves upon this prior work by increasing the number of vulnerable code patterns since their work relies on the presence of specific code patterns that may not be present in the victim code.

Lastly, [68] demonstrates a Rowhammer attack methodology where researchers emulated Rowhammer bitflips on targets. They introduced the idea of simulating a flip in the `EIP` register value in the stack, which can force the execution to jump from kernel code to user code, like the ret2usr attack [33]. However, attacks that cause privilege escalation by jumping from kernel code to user code are mitigated by `SMAP` [16], which prevents the kernel from executing userland instructions. Our attack forces a process to jump within its own code space and privilege space and thus is not affected by `SMAP` and introduces attack surfaces on new code patterns.

4. Threat Model

Similar to other Rowhammer attacks we assume the attacker is co-located on the same system as the victim [30], [22], [65], [14], [23]. Co-location is a common threat model for many micro-architectural side-channel attacks and fault attacks [35], [31], [12], [60], [62]. We do not assume root privilege or physical access to the machine. We only assume that the system has TRR enabled, and bypasses TRR with a many-sided attack [21].

5. LeapFrog Attack

LeapFrog gadgets are exploitable in scenarios where a process undergoes a context switch or executes a function call, leading to the storage of the PC value in either the kernel or user stack. The ingenuity of LeapFrog gadgets lies in their susceptibility to Rowhammer-induced bit flips due to them being stored in DRAM, enabling an attacker to alter the PC value subtly. This manipulation is designed to redirect the execution flow to a different code segment, ideally with minimal bit changes due to the blunt nature of Rowhammer and the higher probability of finding a faulty memory location with few or one faulty bits in the right location. In this paper, we assume that we can successfully find a 1-bit flip within a page that is in the right location to fault the PC value to force the intended instruction to skip.

In Figure 1 the storage of the PC value occurs in the kernel stack during the execution of `wait_receive`. In this scenario, a malicious server can hold the client process at the `wait_receive` function while hammering the PC value to force the process to jump to a new location upon returning from the function. In our assembly code analysis in Figure 1, we observe the original PC value is an address `0x555555555478`. Through strategic bit flips, this value can be altered to `0x555555555578`, effectively enabling an instruction skip (skipping one or more instructions) and jumping from the function call in wait_receive directly to a later point in the execution, bypassing the critical server authentication check. The practicability of such attacks, however, hinges on the feasibility of achieving the desired bit flips, a central challenge to the effectiveness of LeapFrog gadgets in real-world scenarios. In this scenario, flipping the PC from `0x555555555478` to `0x555555555578` only requires a 1-bit flip, which is a reasonable assumption for Rowhammer.

However, tiny variations in the C code can change the resulting assembly code significantly. For example, consider the first approach for a TLS handshake, where the process allocates memory for a message to be signed. The C code and its corresponding assembly code are shown in Listing 1. Alternatively, using a different method to allocate memory for the message results in a variation in the assembly code. This alternative approach and its corresponding assembly code are presented in Listing 2.

In the source code space, the alternative approach (Listing 2) takes `0x55555555541f - 0x555555555413` or 12 bytes of instructions, while the original approach (Listing 1) occupies 8 bytes of assembly instruction (this excludes the size of the last instruction). Given the assumption that only one bit per page can be reliably flipped, identifying useful instruction skips that require a single bit change, as illustrated in Figure 2, is crucial. This example illustrates the challenge of manually inspecting source code to determine the impact of tiny variations on assembly instruction distances. Hence, profiling binaries becomes an important tool in this context.

```
<main>:
...
   0x555555555469: xor    %ecx,%ecx
   0x55555555546b: mov    $0x80,%edx
   0x555555555470: mov    %r12,%rsi
   0x555555555473: callq  <recv@plt>
   0x555555555478: mov    %rax,%rbx
   0x55555555547b: test   %rax,%rax
   0x55555555547e: jg     <main+208>
...
   0x555555555571: callq  <ECDSA_SIG_free@plt>
   0x555555555576: xor    %eax,%eax
   0x555555555578: callq  <auth_success>
   0x55555555557d: jmp    <main+442>
   0x55555555557f: lea    0xb5a(%rip),%rdi
...
```

Figure 1: LeapFrog gadget in TLS handshake $addr_{src}$, the PC value that fault is injected into, is highlighted in blue. The new value is highlighted in red. The fault is injected during the execution of the function call highlighted in green.

Listing 1: Combined C and Assembly code for original memory allocation

```
1 // C Code
2 unsigned char message[32] = "This is a message
      to be signed";
3 int ret = send(client_fd, message, sizeof(
      message),0);
4
5 // Assembly Code
6 0x555555555413: movdqa 0xce5(%rip),%xmm0
7 0x55555555541b: mov $0x20,%edx
```

Listing 2: Combined C and Assembly code for alternative memory allocation

```
1 // C Code
2 unsigned char *message;
3 message = "This is a message to be signed";
4 int ret = send(client_fd, message, sizeof(
      message),0);
5
6 // Assembly Code
7 0x555555555413: lea 0xc6e(%rip),%r14
8 0x55555555541a: lea 0x60(%rsp),%r13
9 0x55555555541f: mov %r14,%rsi
```

5.1. Offline Memory Profiling

Finding Contiguous Memory Virtual to physical address mappings are stored in *pagemap* file in Linux OSs and it requires root privileges to access these translations. Using the SPOILER tool [28], we can reliably leak the information about the first 8 bits in physical addresses after the page offset bits. This allows us to find contiguous memory chunks in physical address space. In Figure 3, the page numbers with the peaks in the y-axis are contiguous pages' physical address space.

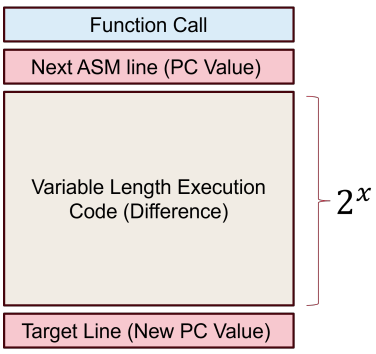

Figure 2: The best LeapFrog gadgets require a single-bit flip, where the distance between the two lines of code is a power of 2.

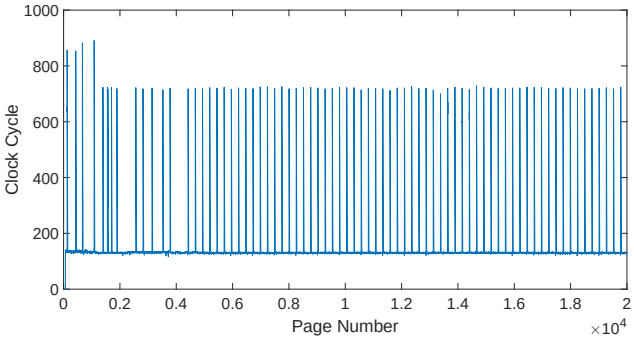

Figure 3: Timing peaks on virtual addresses detected by SPOILER [28] attack. Virtual addresses on the peaks are contiguous on physical address space.

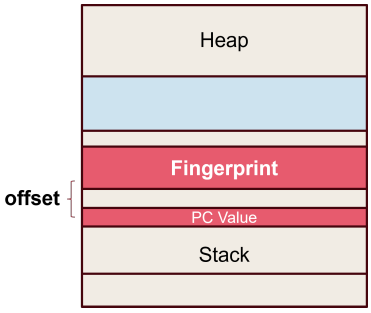

Figure 4: Once the fingerprint is located, there is a constant offset from the fingerprint regardless of ASLR, and this can be used for bait page profiling for the eventual attack

Finding Memory In the Same Banks To find physical memory pages within the same bank, we employ techniques first described in [46]. DRAM is structured in multiple banks that are physically isolated from each other, so while SPOILER can give an attacker physically contiguous chunks, the memory is distributed across multiple banks. This is a problem because hammering rows that are not adjacent within the same bank will not result in bit flips.

We use the row conflict side channel to find co-located memory address accesses within the same bank. We take measurements by iterative reading between two addresses, and if the reading results in high latency (assuming we are not hitting cache), it means DRAM is clearing the row buffer and the addresses are located in the same bank.

Alternatively, if we repeatedly read from two addresses and they are in different banks, the values will be loaded into their respective row buffers for their respective banks, and the reading time will have a lower latency.

Executing the Rowhammer Bit Flip in a Many-Sided Context Despite modern mitigation techniques against Rowhammer like Target Row Refresh (TRR), we are still able to induce flips in DDR4 memory by using a many-sided [21] approach.

In the final phase of our attack, the task is to induce bit flips in the target memory location. This step marks the culmination of the profiling and memory manipulation processes. The challenge lies in the fact that while we can ascertain the occurrence of bit flips in a given row (a row that we deem "flippy"), pinpointing the exact memory bits affected after the attack is not straightforward. This is due to the inherent nature of Rowhammer, where the attacker does not possess direct control over the specific memory areas being altered.

However, the success of the attack is often evident through observable changes in the process' state. For instance, a successful execution might manifest as an unauthorized bypass of security measures, or broken encryption output. This indirect outcome serves as a confirmation of the attack's effectiveness. We further expand on this in section 7.

6. Locating the PC in the Stack

To flip bits in the PC value with Address Space Layout Randomization (ASLR) enabled, the page that contains

Stack

Stack Address	Value
0x7ffd00a45c58	0x1bd000
0x7ffd00a45c60	0x1
0x7ffd00a45c68	0x215000
0x7ffd00a45c70	0x21b000
0x7ffd00a45c78	0x21a888
0x7ffd00a45c80	0x227e50
0x7ffd00a45c88	0x1000
0x7ffd00a45c90	0x214000

Fingerprint

Stack Value	Space
0x7ffff7fccda5	0
0x21b000	184
0x21a888	192
0x214000	216
0x7ffff7fd090e	752
0x7fffffffcf90	768
0x7ffff7fccb0e	784
0x7ffff7fcbc83	1584
0x210003	1616

Figure 5: Finding constant values in the stack to create a fingerprint

the PC value needs to be placed into the page with the bit that will flip during the Rowhammer attack. To do this, we use a method similar to that proposed in [50] where we deallocate a series of pages from the attacker process, launch the victim process, and experimentally determine some probability that the target data (in this case the PC) lands in the target location (the row with the flippy bits). We term the deallocation of pages "baiting" in this paper.

The profiling to determine the proper number of bait pages starts by allocating pages within the attacker's process space, designated to be released as bait. The procedure involves releasing a substantial number of bait pages, recording their physical addresses, and then correlating these with the physical address of the target variable in the victim process. The number of pages consumed by the victim process before allocating the target variable was determined through this correlation.

In a recent work [2], the victim's source code was altered to assign a unique value to the target register or stack variable, thereby making it identifiable in the memory during the profiling stage. This method is not possible with PC values as they are dependent on the compiler, so we introduce a new method to determine the number of bait pages required for the PC value.

The dynamic nature of the PC under ASLR implemented in the Linux kernel necessitates a novel approach that involves identifying invariant values within the stack that serve as reliable *fingerprints*. These fingerprints are used to determine the PC's offset relative to these constants, thereby facilitating the estimation of the required number of bait pages for effective targeting.

6.1. Fingerprinting the Stack

As the PC's address and value fluctuate with each process execution due to ASLR, our strategy leverages the relative stability of certain stack values and correlates an offset from those values. We first profile with ASLR disabled, knowing the target PC value in the stack from an assembly dump with GDB. We then determine an offset from the fingerprint as seen in Figure 4. Then with ASLR enabled, even with the PC value changing, the fingerprint remains identifiable and the offset from the fingerprint remains constant. The outcome is a refined understanding of the number of bait pages required to strategically position the PC, thus enhancing the precision of our Rowhammer attack in an ASLR-enabled environment. Fingerprinting only needs to be done once and is machine-independent.

The process begins by capturing snapshots of the stack at different instances and identifying unique values that persist across these snapshots. We implemented a Python script to automate this analysis. The script compares consecutive stack states, isolating values that remain unchanged— these become features of our fingerprints as seen in Figure 5. By calculating the address differences between these consistent values and tracking their occurrence across multiple iterations, we build a comprehensive profile of the stack's layout. This profile is instrumental in pinpointing the location of the PC relative to the identified fingerprints and is versatile enough to be used on virtually any binary.

7. Automatic Detection of LeapFrog Gadgets with *MFS*

Based on how the LeapFrog gadgets occur in the binary described in section 5, we develop a custom tool we call *MFS* (Multidimensional Fault Simulator) that relies on dynamic binary instrumentation and analysis. Since the attack happens on program counters and registers, which are invisible to high-level code, such as C/C++, it is not possible to do a static analysis of the source code. We put together a set of rules that enables us to collect, filter, and pinpoint the potential LeapFrog gadgets. The overall design is shown in Figure 6.

❶ First, *MFS* collects the instruction traces, specifically, the address of instructions executed, for different inputs. To detect the gadgets that cause security exploits, *MFS* chooses critical input pairs that cause differences in the program's control flow. Such inputs can be correct/incorrect private key pairs or passphrases for authentication programs. Together with the instruction addresses, we collect the execution time of each function executed. Since the return addresses of the functions with larger execution times will stay in the memory for a longer duration, they are potentially more viable targets.

❷ *MFS* then computes the difference between two instruction traces to find the instruction addresses that are executed with correct input(s) but not executed with incorrect input(s). Note that this is an optional step to reduce the complexity of the following steps, and it comes with a cost of false negatives. Moreover, depending on the program and type of exploit, it may not always be possible to get multiple different traces; see section 8.2. Alternatively, the whole instruction trace can be considered instead of only the difference.

❸ Regarding the choice of the fault model on PC values, the probability analysis in previous work [59] is relevant. Given a sequence of bit offsets $b_0, b_1, ..., b_{k+l-1}$ within a memory page and assuming a defective memory cell can flip solely in one direction, the conditional probability of identifying a compatible target page t amongst N susceptible pages can be expressed as:

$$p\big(t|\{b_{n_{0\to1}}\} \in \{0 \to 1\}, \{b_{n_{1\to0}}\} \in \{1 \to 0\}\big) = 1 - \left(1 - \prod_{i=0}^{k-1} \frac{n_{0\to1} - i}{S - i} \times \prod_{j=0}^{l-1} \frac{n_{1\to0} - j}{S - k - j}\right)^N, \quad (1)$$

where $n_{0\to1}$ and $n_{1\to0}$ represent the average counts of error-prone cells on a page that can be flipped from 0 to 1 and from 1 to 0 respectively, k and l denote the counts of bit locations needing flips from 0 to 1 and 1 to 0 respectively, and "S" signifies the total bit count per page.

First, we calculate the probability of finding a target page t for each N value and three different $k + l$ values. Note that $k + l$ is the number of bit offsets within a page. Figure 7 shows that for 1 bit per page, 2200 pages are enough to achieve 99.99% accuracy for a DDR4 DRAM with an average of 100 flips per page. For 2 and 3 bits per page, the same number of pages gives 2% and 0.006% probability, respectively.

MFS looks for address pairs that hold the following conditions:

$$d_{HD}(\text{addr}^i_{\text{exec}}, \text{addr}^j_{\text{return}}) = 1 \quad (2)$$

where $addr^i_{\text{exec}}$ is the address of the i^{th} instruction that is executed, $addr^j_{\text{return}}$ is the return addresses of the j^{th} call instruction, and d_{HD} is the Hamming distance between two addresses. i and j are bounded by the number of all instructions executed (n) and the number of call instructions executed (m), respectively. Although this operation has $O(m^n)$ complexity, it can be implemented with bitwise xor and can be parallelized using multiple processor cores. The condition given in Equation (2) is determined by the Rowhammer fault model. Given that finding a suitable memory page in memory is only realistic with single-bit flip fault models, *MFS* assumes we can only flip a single bit. Yet, the method is generic enough to cover other potential fault models, such as optical fault injection or electromagnetic fault injection, where multiple-bit flips are more likely [10]. This step generates a list of pairs of addresses in the following format: $\{< addr^k_{src}, addr^k_{dest} >\}$ where $addr^k_{src}$ is the k^{th} instruction address that *MFS* targets in the binary's execution with the input that we want to affect the control flow of, such as an incorrect private key, and $addr^k_{dest}$ is the corrupted instruction address after fault injection.

❹ For each address pair we get from the list generated in the previous step, *MFS* starts a simulation session. *MFS* executes the binary again with the incorrect input and simulates a bit flip on the instruction address $addr_{src}$ to make it $addr_{dest}$. Certain instructions may be executed multiple times in a single execution. To correctly cover that case in our fault model, we keep a counter variable for a specific instruction that increments every time the binary

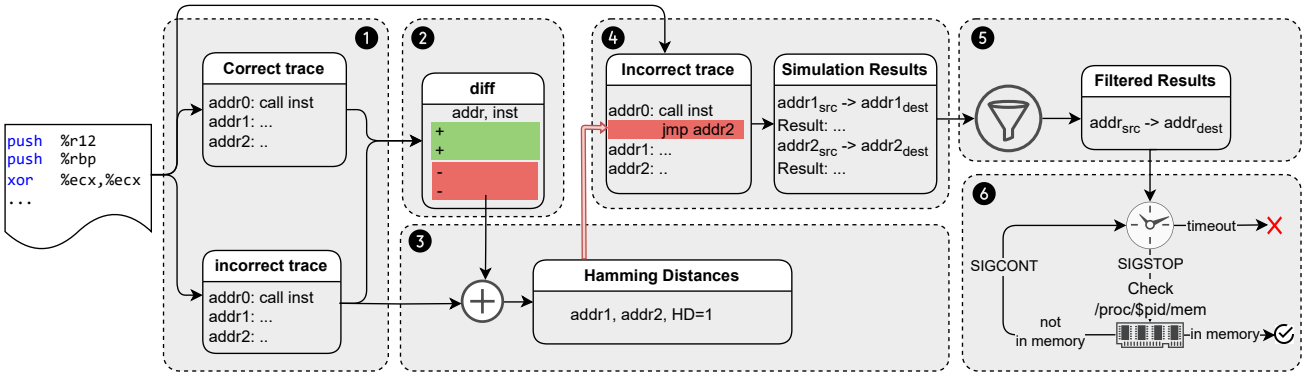

Figure 6: LeapFrog gadget detection using *MFS* framework

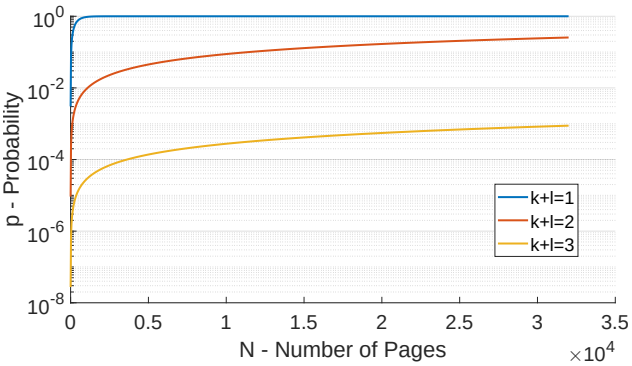

Figure 7: Probability of finding a page among N pages for different k+l values. k+l states the number of targeted bit offsets in a page.

executes the same instruction. In a single execution of the original binary, if an instruction is executed N times, we attempt the fault simulation $N + 1$ times, until we no longer see the same instruction in the trace.

❺ After the bit flip simulation, *MFS* continues the execution of the binary without further faults and observes the new behavior. The analysis of the new behavior is not a trivial task. There are several options where we can observe changes compared to the original execution. For instance, we can observe changes in the total number of executed instructions, the number of instructions that match with the correct input execution trace, the return code of the program, outputs to standard streams, ports that are accessed, function calls, authentication results, etc. The choice of observable depends on the program under test. In this work, *MFS* uses the return codes, standard outputs/errors, and authentications on different case studies.

❻ Once *MFS* has a list of PC values that potentially result in misauthentication or bypass with a single-bit flip, it then evaluates if they are practical to attack from a timing perspective. In some cases, a single bit flip will result in the desired behavior in a process but the attack window of time is too short to effectively attack the target. Additionally, the attack window needs to be long enough to allow for noise in the system - as processes will often take a variable amount of time to execute and get to the vulnerable area in the code where the PC value is shelved in the stack. *MFS* uses process degradation to increase the viability of LeapFrog gadgets, as slowing down a process artificially increases the attack window time. Note that this step is system-specific and it can be affected by the current processor/memory load. Although it is necessary to find viable targets in the list for an end-to-end attack, it does not guarantee that the other targets are not viable in different system configurations, or different systems.

MFS starts the victim process and then immediately stops it with a SIGSTOP signal and it checks if the PC value is currently in the stack of the process. If not, the process is killed and restarted, and stopped after a slightly longer period, in a process we call time sweeping. The challenge is sending a SIGSTOP with the highest timing precision possible. Different implementations of signals will yield different timing resolutions. For example, Python has a signal library that can be used to generate signals similar to a bash script, but there is considerable delay and imprecision in the time it takes to send a signal.

7.1. Tool Implementation

We used Intel's dynamic binary instrumentation framework, `Pin` [38], which allows for process analysis without altering its core behavior to implement ❶ and ❹ of *MFS*. Using Pin also makes it possible to find LeapFrog gadgets in binaries that do not have a source code since it does not require recompiling. In the context of *MFS*, Pin's capabilities are harnessed to monitor the execution trace of a binary. This integration allows for a thorough analysis of potential LeapFrog gadgets by observing how changes in PC values influence program behavior. For each executed instruction, our tool outputs the virtual address of the instruction and disassembly of the machine code. If the instruction is a call instruction, it also outputs the return address of the call. The return address of the call is usually the PC value that is pushed onto the stack before executing the called routine. For every write to STDOUT and STDERR, the tool forwards a copy of the buffer to a text file for further analysis. To avoid the effect of overhead caused by instruction-based instrumentation, function timings are collected in a separate session on every function entry and exit.

❷ is a simple comparison operation on the correct and incorrect execution traces implemented with `diff` command line tool in Linux.

❸ is implemented in Python. *MFS* parses the instruction traces and computes the Hamming distance between the return addresses and instruction addresses of all executed instructions in the correct trace or the list of addresses we get from ❷. The Hamming distances are calculated using the native `bit_count` function in Python followed by `bitwise_xor` in `numpy` library. The operation is parallelized on multiple cores to speed up the analysis.

The bit flip simulation part of *MFS* (❹) is done using `Pin` which takes the address pairs and simulates every fault independently. The faults on PC values are implemented as direct jumps to the corrupted addresses by adding `jmp` $addr_{dest}$ after function returns. Since we add a direct jump to the target address by injecting a line of assembly with the Pin tool, it is functionally equivalent to corrupting the PC value in memory.

❺ filters the simulation results depending on the program and targeted exploit type. For different types of exploits, we filter by return code (section 8.4), value in STDOUT (section 8.2).

❻ The last stage of *MFS* takes the list of PC values generated from the previous steps and determines which are practical from a timing perspective. It does this by sweeping the process in the time domain determining when it needs to stop the process to find particular PC values in the stack.

We begin by defining when we want to start our sweep, and what interval we want to sweep at. For OpenSSL as an example, we started our sweep at 0ns and had an interval of 100ns. Generally, the higher the resolution of the sweep, the longer the simulation takes. However, a smaller interval increases the likelihood that we will successfully send a SIGSTOP at a time when the target PC value is in the stack.

To determine if a PC value is in the stack, we start the victim process as a non-root user on a sibling core to a core that we are attacking with SMC to degrade the performance. For example, in our tests we isolated cores 6 and 14 and triggered SMC events on core 14 while running the victim process on core 6. Once we have the process identification number (PID) of the process and send a SIGSTOP, we use the Linux process interface to check the stack for the PC value. We do this by looking at the `/proc/[pid]/maps` file to determine which offsets in the victim process's address space contain the stack, and then we read from `/proc/[pid]/mem` at the offsets determined by `/proc/[pid]/maps` to find the PC values. The tool will generate a dictionary of stack addresses/values for the victim process that we can search through.

If during a sweep the tool finds the PC value in the stack, it will simulate a flip by overwriting that value with the new PC value determined by the previous steps to verify that the gadget does result in the intended behavior (privilege escalation, data leak, mis-authentication, etc...).

Generally, if *MFS* can successfully pass all stages of filtering with a particular LeapFrog gadget, we believe that it can be attacked and flipped with Rowhammer to cause the desired behavior.

8. Experiments

Experiment Setup The experiments are conducted on a system with Ubuntu 22.04.2 LTS with 6.2.0-37-generic Linux kernel installed. The system uses an Intel Core i9-9900K CPU with a Coffee Lake microarchitecture. We used a dynamic clock frequency instead of a static clock frequency to improve the practicality of the attack. End-to-end attack experiments are performed on a single DIMM Corsair DDR4 DRAM chip with part number CMU64GX4M4C3200C16 and 16GB capacity. DRAM row refresh period is kept at 64ms, which is the default value in most systems. In all the experiments, we used 100s simulation timeout, since the fault simulations rarely cause infinite loops. We empirically observe that using the Python signals library, the target process could complete 34M cycles before the attacker can stop it, with a standard deviation of 2.7M cycles. Alternatively, using a bash script, the victim process can only complete 18M cycles before it is stopped, with a standard deviation of 0.3M cycles. There is an order-of-magnitude difference in precision stopping a process with bash vs. with Python.

8.1. ML Misclassification

In this section, we investigate the potential implications of instruction skipping in the machine learning domain, specifically for decision tree algorithms. A decision tree is an ML model used to make predictions based on a series of binary choices, effectively splitting data into increasingly specific groups. It starts with a single node, which branches into possible outcomes based on the features of the data. Each branch represents a decision pathway, and each node in the pathway represents a test on a specific attribute. This process continues until a leaf node is reached, which provides the predicted outcome. They are widely used in various applications, from financial forecasting [37] to medical diagnosis [54] due to their interpretability, and efficiency for a variety of tasks such as classification, and feature importance ranking. We choose a decision tree for proof of concept yet instruction skipping attacks can be effective in every kind of model implementation.

Classification algorithms may be vulnerable to the LeapFrog attack under the threat model that an attacker is co-located on the server with the victim process running the model, and the attacker would like to force a particular output. If the attacker faults the victim process program counter and forces a jump in the code, the result may be a misclassification or a forced classification of a particular output. This attack is different from other Rowhammer attacks on machine learning models [59] because for this attack we do not need to know the model weights before hand, and we consider this a gray box model.

In this experiment, we use a public implementation[44] as our target. We simulate program counter flips and observe the effects on the model output. We follow a similar procedure to previous examples, where we experiment with a hammering distance of 1, 2, and 3 and determine the number of successful LeapFrog gadgets with each of these distances. In Table 1, we can see various number of LeapFrog candidate gadgets that

might result in a misclassification. After simulating these gadgets, we found 23 of the 1363 potential gadgets within 1 hammer distance would result in a misclassification.

Target	Size	#Inst.$_{exec}$	d_{HD}	# Candidates ❷ on	# Candidates ❷ off
Decision Tree	99KB	38417	1	N/A	1363
			2	N/A	8667
			3	N/A	32326

TABLE 1: Number of gadget candidates found in decision tree algorithm with different Hamming distances.

8.2. OpenSSL Encryption Bypass

We analyze `openssl` command line tool that uses OpenSSL v1.1.1w for block cipher and stream cipher implementations. For each cipher, we give a simple plaintext that contains the `helloworld` string and run encryption without salt with a simple passphrase. We aim to find LeapFrog gadgets in the binary that can be exploited for bypassing encryption steps in the ciphers, revealing the plaintext.

First, we scan the binary using *MFS* as described in section 7. Since we do not aim for any authentication bypass in this scenario, and the execution traces are deterministic for fixed inputs, step ❷ is not applicable. Instead, in step ❸, we compare the return addresses in a single trace against all the instruction addresses in the same trace to look for targets with $d_{HD} = 1$. This means that ultimately, for each return address, we test 12 different jumps, a trial for each bit in the page offset. Table 2 shows the total number of gadgets for each hammer distance, with a hammer distance of 1 having a total of 2700 candidates.

We scanned the binary with 135 different ciphers available in OpenSSL. Most of the time the binary was not affected by the simulated bit flip and correctly produced the ciphertext.

Target	Size	#Inst.$_{exec}$	d_{HD}	# Candidates ❷ on	# Candidates ❷ off
OpenSSL	818KB	49431	1	N/A	2700
			2	N/A	20208
			3	N/A	70475

TABLE 2: Number of gadget candidates found by *MFS* in for fault models with different Hamming distances. We ran OpenSSL with `aria-128-cbc` cipher.

Figure 10 illustrates one of the LeapFrog gadgets found in the `openssl` command line tool. When we corrupt a single bit in `0x55555559c4c5`, the return address of `opt_cipher` function, to make it `0x55555559c0d5`, the function returns to the corrupted return address, skipping three instructions in between. Similarly, another single-bit corruption to (`0x55555559c0c5`) causes the function to return to an earlier point in the program. We verified that both of these bit flips cause the binary to skip the whole encryption and instead output the plaintext. Similarly, *MFS* detected LeapFrog gadgets that are used in 36 ciphers including block ciphers and stream ciphers. The ciphers with

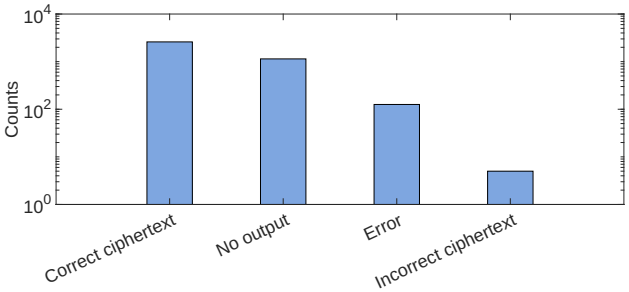

Figure 8: `aes-256-ctr` simulation results

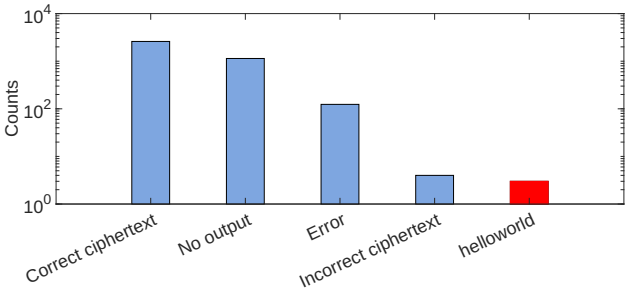

Figure 9: `aria-256-ctr` simulation results. Plaintext `helloworld` is revealed three times.

LeapFrog gadgets that revealed full or partial plaintext are listed in Table 3. Figure 8 and 9 summarize the simulation results for `aes-256-ctr` and `aria-256-ctr` respectively.

Even with ASLR enabled, these gadgets are reproducible because ASLR does not randomize the last 12 bits of the code space (the page offset). We only simulated faults in the last 12 bits (which should be the same across all x86 machines the process is compiled for), thus, the LeapFrog gadgets should work across machines without the need for rescanning.

8.3. Post-Quantum Cryptography Schemes

NIST announced the standards for Post-Quantum Cryptography (PQC) in FIPS 204 [48], and FIPS 205 [49]. These standards are used for digital signatures to pro-

Recovered	Cipher
`helloworld`	aria-128-cbc, aria-128-cfb,aria-128-cfb1 aria-128-cfb8, aria-128-ctr, aria-128-ofb aria192, aria-192-cbc, aria-192-cfb aria-192-cfb1, aria-192-cfb8, aria-192-ctr aria-192-ofb, aria256, aria-256-cbc aria-256-cfb, aria-256-cfb1, aria-256-cfb8 aria-256-ctr, aria-256-ofb, bf-ofb rc2-ofb, rc4, rc4-40
`hellowor...`	bf-cfb, rc2-cfb
`hdlmowor...`	idea-cfb, idea-ofb
`oworhell...`	bf, bf-cbc, bf-ecb, blowfish
`?rl#a?gy?...`	chacha20, des-ede3-ofb, des-ede-ofb, des-ofb

TABLE 3: 36 ciphers implemented in OpenSSL that are vulnerable to LeapFrog attack. Each given cipher reveals the plaintext fully or partially in the ciphertext due to skipped encryption steps.

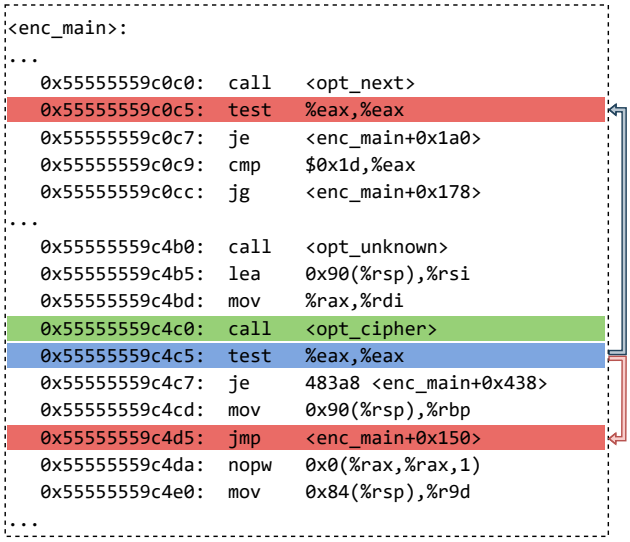

Figure 10: LeapFrog gadget in OpenSSL command line tool resulting in encryption bypass in aria-128-cbc block cipher. The PC value that fault is injected into, $addr_{src}$, is highlighted in blue. The new value after the fault injected,$addr_{dest}$, is highlighted in red. The fault is injected during the execution of the function call highlighted in green.

tect against quantum attacks. We use Open Quantum Safe (`liboqs version 0.11.1-dev`) library [55], an open source library for PQC algorithms, to find LEAPFROG gadgets on FIPS standards using *MFS* tool.

One of the algorithms selected by NIST for standardization is CRYSTALS-Dilithium, which serves as a digital signature scheme providing post-quantum security guarantees. Dilithium relies on the hardness of structured lattice problems, such as the Learning With Errors (LWE) problem, which is believed to be intractable for quantum computers. Another prominent algorithm is FALCON, which offers smaller key sizes and signatures by employing the NTRU lattice, making it a competitive choice for constrained environments. Our analysis of these algorithms reveals that, despite their robust design against quantum attacks, they still exhibit vulnerabilities at the implementation level, susceptible to hardware fault injections like the LEAPFROG used in Rowhammer-based exploits.

In digital signature schemes, we find gadgets that produce several failure modes in the Open Quantum Safe Library. The most critical error is a bypass of the signature verification. Note, that while we experimented with Post-Quantum encryption schemes, theoretically LeapFrog gadgets should work on classical encryption schemes as well.

The "Magic Number Mismatch" column in Table 4 highlights instances where the injected fault corrupts memory regions containing predefined magic numbers used for integrity checks. This mismatch signifies unintended memory corruption caused by the LeapFrog gadget, which can lead to unpredictable behavior or system crashes. According to Table 4, all signature schemes also contain gadgets for this failure mode, with Dilithium 3 containing the most number of LeapFrog gadgets.

Failures during key generation ("Key Gen. Fail"), signature generation ("Sig. Gen. Fail"), and signature verification ("Sig. Verif. Fail") were also identified. Such failures can be exploited to disrupt normal cryptographic operations, resulting in denial-of-service (DoS) attacks or weakening cryptographic strength by producing invalid or insecure keys and signatures. All schemes contain this type of gadget.

The "Incorrect Verification" column denotes scenarios where invalid signatures are erroneously accepted as valid. This occurs when a LeapFrog gadget alters the control flow of the verification routine, enabling attackers to perform impersonation attacks by forging signatures that bypass standard validation checks.

Lastly, the "Verification Bypass" column in Table 4 highlights instances where the signature verification routine can be entirely circumvented using LeapFrog gadgets. Similar to the TLS attack scenario described in Section 8.4, this allows an attacker to craft an invalid signature and have it accepted as valid at the client's end. By flipping bits in the Program Counter (PC) values using the LEAPFROG within the client's memory space, the attacker effectively bypasses the signature verification routine. This vulnerability poses a significant security risk by enabling impersonation attacks and facilitating unauthorized access or actions within the system. Notably, Dilithium3 exhibits the highest number of LeapFrog gadgets for this threat, indicating a greater susceptibility to such attacks. An example of such a LeapFrog gadget in Dilithium is seen in Figure 11.

We find LEAPFROG gadgets on FIPS 204 standard, also on other PQC digital signatures schemes, FALCON [20], MAYO [7], and CROSS [6].

Table 4 summarizes LEAPFROG gadgets found in the `liboqs` library for different PQC digital signature schemes. Compared to Dilithium, ML-DSA, the implementation of the FIPS 204 standard, had fewer LEAPFROG gadgets, suggesting that its implementation might be more resilient to the specific fault attacks we conducted. However, this does not imply immunity, as the gadgets found were still capable of bypassing critical functions. The relatively lower number of vulnerabilities in ML-DSA could also be attributed to its simpler structure, which reduces the surface area for potential control flow subversion attacks.

We also evaluated SPHINCS+, a hash-based signature scheme standardized in FIPS 205. SPHINCS+ offers a different security foundation, relying on the hardness of hash-based constructions rather than lattice problems. While this scheme is robust against certain classes of attacks, our analysis uncovered several LeapFrog gadgets capable of bypassing signature verification. This suggests that even though the algorithm itself is designed to withstand quantum and classical cryptanalytic attacks, practical vulnerabilities arise due to implementation flaws that allow Rowhammer-based attacks to alter execution paths. Interestingly, the number of LEAPFROG gadgets identified in SPHINCS+ varied significantly based on its parameter set, with some configurations being more resilient than others. This highlights the importance of parameter selection in mitigating the risk of physical attacks.

SPHINCS+ has more gadgets compared to the FALCON-1024 configuration, but in some configurations, it has fewer gadgets than FALCON-512, another selected algorithm that is not standardized. Overall, our findings

Scheme	# Instructions	Candidate Gadgets	Magic Number Mismatch	Key Gen. Fail	Sig. Gen. Fail	Sig. Verif. Fail	Incorrect Verification	Verification Bypass
Dilithium2	43474	17472	18	3	11	294	5	8
Dilithium3	42800	25010	36	6	22	654	10	20
Dilithium5	44171	18591	12	8	14	355	4	13
Falcon-512	60647	18641	18	16	50	106	4	16
Falcon-1024	60794	9360	8	8	22	40	2	7
Falcon-padded-512	60678	9396	8	9	21	41	2	6
Falcon-padded-1024	61128	9456	8	6	19	42	2	9
MAYO-1	43542	6924	8	3	4	33	2	8
MAYO-2	43533	6984	8	4	5	32	2	8
MAYO-3	46823	6996	8	3	4	35	2	6
MAYO-5	44302	6528	6	4	3	34	3	11
ML-DSA-44	43668	17616	0	3	1	21	1	5
ML-DSA-44-ipd	43338	8820	9	3	4	172	3	8
SPHINCS+-SHA2-128f	37109	8052	8	3	5	112	6	9
SPHINCS+-SHA2-128s	37379	8052	8	3	8	109	5	7
SPHINCS+-SHA2-192f	42143	8460	7	3	9	113	5	7
SPHINCS+-SHA2-192s	42535	8484	7	3	10	109	4	6
SPHINCS+-SHA2-256f	42437	8532	8	3	4	94	3	8
SPHINCS+-SHA2-256s	42758	8556	8	3	6	98	3	7
cross-rsdp-128-balanced	44024	10032	9	4	4	209	2	8

TABLE 4: Results from scans on the `liboqs` library, showing various issues encountered during signature operations for each digital signature scheme, along with the total number of assembly executions and candidate gadgets.

```
<OQS_randombytes_system>:
...
   0x5555555586e1:  mov    %r12,%rdi
   0x5555555586e4:  mov    %rax,%rbp
   0x5555555586e7   callq  <fread@plt>
   0x5555555586ec:  cmp    %rax,%rbx
   0x5555555586ef:  ja     <OQS_randombytes_system>

...
```

Figure 11: LeapFrog gadget detected in `liboqs` binary for Dilithium PQC Digital Signature Scheme. The PC value that fault is injected into, $addr_{src}$, is highlighted in blue. The new value after the fault injected, $addr_{dest}$, is highlighted in red. The fault is injected during the execution of the function call highlighted in green.

Target	Size	#Inst.$_{\text{exec}}$	d_{HD}	# Candidates ❷ on	# Candidates ❷ off
TLS	29KB	5328007	1	315	2493
			2	2240	14413
			3	21841	67421

TABLE 5: Number of gadget candidates found in TLS scenario for fault models with different Hamming distances.

indicate that there are generally more LEAPFROG gadgets that enable bypassing signature verification compared to those that can falsely verify an invalid signature, indicating higher feasibility for DoS attacks with lower security impact compared to impersonation attacks.

8.4. TLS Handshake

In a full end-to-end attack example, we illustrate the potency of the attack by applying it within a client/server authentication framework, specifically using OpenSSL for signature verification. Here, we consider a scenario where the attacker shares a physical computing space with the client. The goal of the attacker is to manipulate the client's signature verification mechanism, causing it to erroneously validate a corrupted signature as genuine. This manipulation forms part of a broader man-in-the-middle strategy, aimed at deceiving the client into believing they are securely connected to the intended server.

In the standard communication flow, the client initiates contact with the server by dispatching a `ClientHello` message. The server replies with a `ServerHello` message, which carries its public key and a digital signature of the handshake process. The client's role is then to authenticate this signature using the server's public key. Under normal circumstances, a verified signature would indicate a secure channel, prompting the client to transmit sensitive data to the server. However, in our attack scenario, the attacker strategically alters the signature verification process at the client's end. By inducing a single-bit error during this process, the client is misled into accepting a fraudulent signature as valid. As a result, the client erroneously trusts the communication channel and proceeds to send sensitive information to the attacker.

Figure 12 illustrates a standard interaction where the client establishes a connection with the server, sends a request, and then receives a server-signed message, enabling server authentication. A critical aspect to note is the client's susceptibility to a Rowhammer attack while it awaits the server's response. This waiting period, which can last several milliseconds, is primarily dictated by the server's response time. During this interval, an attacker has the opportunity to exploit the Rowhammer vulnerability by targeting the client's memory.

We first profile the target memory to determine where the client places the PC value in recently deallocated pages. We use a process we call *baiting* [50] where we allocate a series of pages as the attacker. Then, we

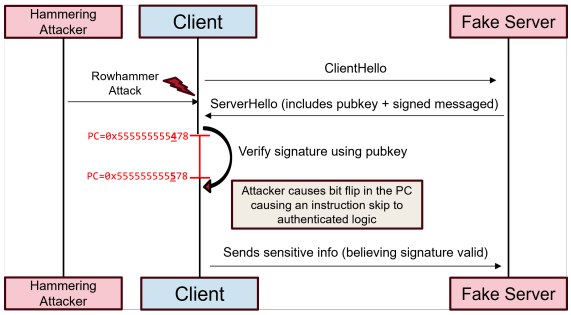

Figure 12: TLS Handshake: The client attempts to authenticate the server, and a colocated rowhammer attacker flips the PC value causing an instruction skip resulting in a misauthentication - this is an end-to-end attack

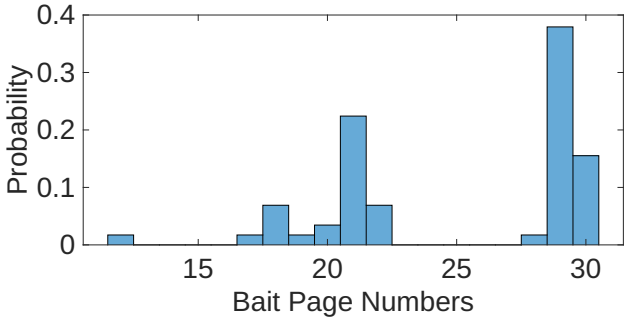

Figure 13: Probability distribution of bait page numbers.

deallocate the flippy page, forcing the victim process to use recently-released flippy locations, where the PC values are stored. You can see our results in Figure 13 where we see that releasing 29 pages results in nearly a 30% chance of a PC value being stored in the next page released (which would be the flippy page).

We scan the client binary using *MFS* while the server is using the correct and incorrect private key. With step ❷ on, we found 315 unique gadget candidates with $d_H = 1$. When the server uses the correct key, the client binary terminates with return code `0`, and when the server uses an incorrect key, the client returns with code `1`. In step ❺, we look for PC corruptions that cause the client to return with value `0`, meaning it incorrectly authenticates the server. After the simulation, we found that one of the candidates was a LeapFrog gadget that caused false authentication of the malicious server with an incorrect key.

Then, we scan the client with step ❷ off. With this mode, *MFS* detected 2493 gadget candidates with $d_H = 1$. After the simulation steps, we verified that 21 of those candidates were LeapFrog gadgets that caused the client to return with `0`, including the one found earlier. The number of candidates for different Hamming distance values is given in Table 5.

The total time for the end-to-end attack to induce a successful misauthentication of the TLS handshake was 12 hours and 25 minutes, as seen in Table 6. This time included profiling the system for the proper flippy pages with the correct offset, meaning the actual online time was around 2 hours. The experiment found a total of 1647 unique flippy pages, and over the course of the 2 hours of online attacking, we saw 2206 attacks where the program counter was baited into the correct page we were attacking before it flipped.

Category	Result
Total Time	12 hrs 25 mins
Online Time	1 hr 54 mins
Total Flippy Pages	1647
Total Attacks w/ Correct # of Bait pages	2206

TABLE 6: Results from the end-to-end attack on code using OpenSSL client/server signature verification with LeapFrog gadget

9. Countermeasures

Rowhammer Resistant Hardware. Increasing the DRAM refresh rate is a commonly cited countermeasure to prevent Rowhammer attacks. Standard DRAM refresh is 64ms, meaning that a Rowhammer attack has 64ms to flip a bit before the row refreshes. Thus, a faster refresh rate will result in a shorter time window for the Rowhammer attack to be performed and should result in fewer flips. This is not an ideal solution, however, because a faster refresh rate will lead to worse power usage and performance overall. Alternative methods such as probabilistic row refresh [63] and parallel row refresh [66] are not available in consumer systems. Additionally, upgrading hardware to newer DDR5 technology has also proven to be ineffective [29].

A novel countermeasure against Rowhammer attacks is the Randomized Row-Swap (RRS) method [52]. This approach fundamentally disrupts the spatial connection between aggressor and victim DRAM rows, thereby offering a robust defense against complex Rowhammer access patterns, including those not mitigated by victim-focused methods like the Half-Double attack. RRS operates by periodically swapping aggressor rows with randomly selected rows within the DRAM memory, limiting the potential damage to any single locality. While RRS can be implemented in conjunction with any tracking mechanism, its effectiveness has been demonstrated when paired with a Misra-Gries tracker, targeting a Row Hammer Threshold of 4.8K activations, akin to state-of-the-art attacks.

Initial beliefs held that the Error Correcting Code (ECC) would serve as an effective defense against Rowhammer attacks. However, subsequent research has shown that ECC, despite its prevalence in server environments, falls short of a comprehensive solution. This inadequacy primarily arises due to ECC's vulnerability to scenarios involving triple bit flips, a phenomenon well-documented in the literature [15]. Additionally, ECC, while standard in server-grade hardware, is typically absent in consumer-grade DRAM systems.

Adding `nops` To Code. A mitigation against the LeapFrog attack specifically would be patching the source code or binary such that it is no longer vulnerable. Given the single-bit flip requirement of Rowhammer on the PC values, adding enough `nops` within the LEAPFROG gadget to prevent instruction skips that only require a single-bit flip would potentially mitigate the attack. Adding `nop` instructions to source code is not trivial when the compiler optimizations are enabled since the compiler may

reorder the critical parts in a different way, which makes the patch ineffective. A mitigation tool that adds `nops` to binary itself may overcome the compiler effect. Yet, adding new instructions to a binary will result in a change in the address of all the following instructions, which may introduce new LEAPFROG gadgets. Therefore, the patched binary needs to be re-evaluated if the new version still has gadgets. Although a LeapFrog-aware compiler may potentially generate a LEAPFROG proof binary, we claim it is not a sound and reliable approach.

Adding Redundancy to the Control Flow. Since LeapFrog gadgets are hard to mitigate in the source code and binary manually, we need a generic mitigation that can be implemented at the compiler level. The main target in LeapFrog is the program counter values that are temporarily pushed into the stack. Pushing multiple copies of the program counter to the stack and making sure the ultimate decision to return to an address is made on the combination of these copies would potentially make the attack impractical.

10. Conclusion

In this work, we introduced LEAPFROG, a specific type of Rowhammer exploit that directly targets the control flow of programs by manipulating the Program Counter stored in the stack. This novel approach marks a significant shift in the understanding of Rowhammer threats, moving beyond traditional data integrity attacks to those that can alter program execution. Our successful demonstration of this attack in an OpenSSL TLS handshake scenario highlights its practical effectiveness and potential impact on widely used security protocols.

Furthermore, we proposed a systematic approach to identify LeapFrog gadgets in real-world software. Using our *MFS* analysis tool, we scanned multiple OpenSSL ciphers, Open Quantum Safe signature schemes, and machine learning classification algorithms and quantified the number of LeapFrog gadgets in this software. Even though the identification of vulnerable software is relatively straightforward thanks to our detection tool, mitigation of LEAPFROG is not a trivial task since it is not transparent to the developers on a source code level. Instead, dedicated Rowhammer-resistant DRAM hardware or Rowhammer-aware compiler tools will be required to prevent LeapFrog attacks.

11. Disclaimer

Andrew Adiletta's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions, or viewpoints expressed by the author. All references are public domain.

References

[1] Andrew Adiletta. Rowhammer; a review of the exploit used to access protected, inaccessible memory, 2021.

[2] Andrew J. Adiletta, M. Caner Tol, Yarkın Doröz, and Berk Sunar. Mayhem: Targeted corruption of register and stack variables. In *Proceedings of the 2024 ACM Asia Conference on Computer and Communications Security*, 2024.

[3] Alejandro Cabrera Aldaya and Billy Bob Brumley. Hyperdegrade: From ghz to mhz effective cpu frequencies. *arXiv preprint arXiv:2101.01077*, 2022.

[4] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 422–435, 2016.

[5] 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.

[6] Marco Baldi, Alessandro Barenghi, Sebastian Bitzer, Patrick Karl, Felice Manganiello, Alessio Pavoni, Gerardo Pelosi, Paolo Santini, Jonas Schupp, Freeman Slaughter, et al. Cross-codes and restricted objects signature scheme. In *2024 Spring Eastern Sectional Meeting*. AMS, 2023.

[7] Ward Beullens. Mayo: practical post-quantum signatures from oil-and-vinegar maps. In *International Conference on Selected Areas in Cryptography*, pages 355–376. Springer, 2021.

[8] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14:101–119, 2015.

[9] 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, August 2017. USENIX Association.

[10] Jakub Breier and Xiaolu Hou. How practical are fault injection attacks, really? *IEEE Access*, 10:113122–113130, 2022.

[11] Jakub Breier, Dirmanto Jap, and Chien-Ning Chen. Laser profiling for the back-side fault attacks: With a practical laser skip instruction attack on aes. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*. ACM, 2015.

[12] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[13] 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.

[14] 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.

[15] 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.

[16] Jonathan Corbet. Supervisor mode access prevention. https://lwn.net/Articles/517475/, Sep 2012. Accessed: 2024-01-10.

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

[18] 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.

[19] Kemal Derya, M Caner Tol, and Berk Sunar. Fault+ probe: A generic rowhammer-based bit recovery attack. *arXiv preprint arXiv:2406.06943*, 2024.

[20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, Zhenfei Zhang, et al. Falcon: Fast-fourier lattice-based compact signatures over ntru. *Submission to the NIST's post-quantum cryptography standardization process*, 36(5):1–75, 2018.

[21] 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.

[22] 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.

[23] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 300–321. Springer, 2016.

[24] 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.

[25] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[26] Nishad Herath and Anders Fogh. These are not your grand Daddys cpu performance counters–cpu hardware performance counters for security. *Black Hat Briefings*, 2015.

[27] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping microarchitectural attacks before execution. *IACR Cryptol. ePrint Arch.*, 2016:1196, 2016.

[28] 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.

[29] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. ZenHammer: Rowhammer attacks on AMD zen-based platforms. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1615–1633, Philadelphia, PA, August 2024. USENIX Association.

[30] 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. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[31] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[32] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-double: Hammering from the next row over. In *31st USENIX Security Symposium: USENIX Security'22*, 2022.

[33] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[34] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711. IEEE, 2020.

[35] 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)*, 2018.

[36] 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.

[37] Jiaming Liu, Chengzhang Li, Peng Ouyang, Jiajia Liu, and Chong Wu. Interpreting the prediction results of the tree-based gradient boosting models for financial distress prediction with an explainable machine learning approach. *Journal of Forecasting*, 42(5):1112–1137, 2023.

[38] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[39] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon attack: Finding and exploiting {Most-Significant-Bit-Oracles} in {TLS-DH (E}). In *30th USENIX Security Symposium (USENIX Security 21)*, pages 213–230, 2021.

[40] MITRE. CVE-2022-42961. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-42961, 2022. Accessed: 2024-10-12.

[41] MITRE. CVE-2024-5288. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-5288, 2024. Accessed: 2024-10-12.

[42] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4:145–156, 2014.

[43] Koksal Mus, Yarkın Doröz, M Caner Tol, Kristi Rahman, and Berk Sunar. Jolt: Recovering tls signing keys via rowhammer faults. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1719–1736. IEEE, 2023.

[44] Okay Demir. Decision tree implementation. https://github.com/okaydemir/binary-classification, 2016. Accessed: 2024-10-11.

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

[46] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, August 2016. USENIX Association.

[47] Labonnah F Rahman, Mamun Bin Ibne Reaz, Chang Tae Gyu, and Mohd Marufuzzaman. Design of sense amplifier for non volatile memory. *Revue Roumaine Des Sciences Techniques*, 58(2):173–182, 2013.

[48] Gina M Raimondo and Laurie E Locascio. Module-lattice-based digital signature standard. *National Institute of Standards and Technology, Gaithersburg*, 2023.

[49] Gina M Raimondo and Laurie E Locascio. Stateless hash-based digital signature standard. *National Institute of Standards and Technology, Gaithersburg*, 2023.

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

[51] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High precision fault injections on the instruction cache of armv7-m architectures. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 62–67. IEEE, 2015.

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

[53] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15:71, 2015.

[54] Kareemulla Shaik, Janjhyam Venkata Naga Ramesh, Miroslav Mahdal, Mohammad Zia Ur Rahman, Syed Khasim, and Kanak Kalita. Big data analytics framework using squirrel search optimized gradient boosted decision tree for heart disease diagnosis. *Applied Sciences*, 13(9):5236, 2023.

[55] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, pages 14–37, Cham, 2017. Springer International Publishing.

[56] Akira Takahashi and Mehdi Tibouchi. Degenerate fault attacks on elliptic curve parameters in openssl. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, pages 371–386. IEEE, 2019.

[57] Andrei Tatar, Radhesh Krishnan Konoth, 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)*, pages 213–226, Boston, MA, July 2018. USENIX Association.

[58] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. Go go gadget hammer: Flipping nested pointers for arbitrary data leakage. In *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA, August 2024. USENIX Association.

[59] M Caner Tol, Saad Islam, Andrew J Adiletta, Berk Sunar, and Ziming Zhang. Don't knock! rowhammer at the backdoor of dnn models. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 109–122. IEEE, 2023.

[60] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

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

[62] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[63] Z. Wang, W. Liu, and Y. Wang. Discreet-para: Rowhammer defense with low cost and high efficiency. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 1–8. IEEE, 2021.

[64] 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.

[65] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 19–35, Austin, TX, August 2016. USENIX Association.

[66] A Giray Yağlikçi, Ataberk Olgun, Minesh Patel, Haocong Luo, Hasan Hassan, Lois Orosa, Oğuz Ergin, and Onur Mutlu. Hira: hidden row activation for reducing refresh latency of off-the-shelf dram chips. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 815–834. IEEE, 2022.

[67] Keun Soo Yim. The rowhammer attack injection methodology. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pages 1–10, 2016.

[68] Keun Soo Yim. The rowhammer attack injection methodology. In *2016 IEEE 35th symposium on reliable distributed systems (SRDS)*, pages 1–10. IEEE, 2016.

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