Achilles: A Formal Framework of Leaking Secrets from Signature Schemes via Rowhammer

Junkai Liang[1,*], Zhi Zhang[2,*], Xin Zhang[1,*], Qingni Shen[1,†], Yansong Gao[2],
Xingliang Yuan[3], Haiyang Xue[4], Pengfei Wu[4], Zhonghai Wu[1,†]
[1]*Peking University,* [2]*The University of Western Australia,*
[3]*The University of Melbourne,* [4]*Singapore Management University*
{ljknjupku, zzhangphd}@gmail.com, zhangxin00@stu.pku.edu.cn,
qingnishen@pku.edu.cn, garrison.gao@uwa.edu.au, xingliang.yuan@unimelb.edu.au,
haiyangxc@gmail.com, pfwu@smu.edu.sg, wuzh@pku.edu.cn

Abstract

Signature schemes are a fundamental component of cybersecurity infrastructure. While they are designed to be mathematically secure against cryptographic attacks, they are vulnerable to Rowhammer fault-injection attacks. Since all existing attacks are ad-hoc in that they target individual parameters of specific signature schemes, it remains unclear about the impact of Rowhammer on signature schemes as a whole.

In this paper, we present Achilles, a formal framework that aids in leaking secrets in various real-world signature schemes via Rowhammer. Particularly, Achilles can be used to find potentially more vulnerable parameters in schemes that have been studied before and also new schemes that are potentially vulnerable. Achilles mainly describes a formal procedure where Rowhammer faults are induced to key parameters of a generalized signature scheme, called G-sign, and a post-Rowhammer analysis is then performed for secret recovery on it. To illustrate the viability of Achilles, we have evaluated six signature schemes (with five CVEs assigned to track their respective Rowhammer vulnerability), covering traditional and post-quantum signatures with different mathematical problems. Based on the analysis with Achilles, all six schemes are proved to be vulnerable, and two new vulnerable parameters are identified for EdDSA. Further, we demonstrate a successful Rowhammer attack against 3 of these schemes, using recent cryptographic libraries including `wolfssl`, `relic`, and `liboqs`.

1 Introduction

Signature schemes play a significant role in computer and network security, serving applications such as protecting personal information [1, 2], blockchain authentication [3, 4], and many others [5, 6, 7, 8]. As a result, their implementations have been the target of numerous Rowhammer fault attacks [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]. Rowhammer [20] stands out as the only fault, to date, that unprivileged software can effectively and reliably induce in the Dynamic Random Access Memory (DRAM) of commodity systems.

Rowhammer attacks against signature schemes: Existing attacks target individual parameters of a specific signature scheme. According to whether a parameter is public, the attacks can be classified into two categories: faulting public parameters (*pp*) and faulting secret parameters (*sp*).

In the first category, researchers have shown that the secret key from RSA can be recovered by faulting the message m being signed [13, 21]. Similarly, Poddebniak et al. [12] demonstrate the secret key recovery from EdDSA by faulting m. These works utilize Differential Fault Analysis (`DFA`), which computes the differences between a valid signature and a faulty signature after inducing faults in m, to reveal the secret key.

In the second category, some works fault the secret keys of various signature schemes (e.g., ECDSA [22], LUOV [23], and Dilithium [24]), while others focus on injecting faults into the nonce in ECDSA [25, 26]. With faulty signatures generated via faulting a secret parameter, a different technique called Signature Correction Analysis (`SCA`) is employed to recover bits of the secret. Since `SCA` can only extract one bit at a time by correcting a faulty signature, it requires many more faulty signatures compared to `DFA`.

Despite the growing threat of Rowhammer against signature schemes, we observe that all the aforementioned attacks are ad-hoc, each targeting a specific scheme with a narrow focus on limited parameters. The impact of Rowhammer on signature schemes as a whole remains unclear, and thus, we are interested in the following questions:

1. *Is there a formal framework that describes how Rowhammer can be exploited to leak secret keys from signature schemes?*
2. *If so, can this framework help uncover additional vulnerable parameters in previously studied schemes?*

---

*: The authors contribute equally to this paper.

†: Corresponding author. This work was supported by the National Key R&D Program of China under Grant No. 2022YFB2703301, PKU-OCTA Laboratory for Blockchain and Privacy Computing and the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant.

3. *Further, can it be used to identify new schemes that are susceptible to Rowhammer attacks?*

In this work: we offer affirmative responses by introducing a framework termed as Achilles. With Achilles, leaking secrets from signature schemes consists of the following four steps (in Section 4):

- *First*, we generalize various real-world signature schemes and derive an abstract framework (coined as G-sign). This framework encompasses a set of key parameters, each categorized as either pp or sp. With G-sign, a signing oracle $OSign$ is defined to generate valid signatures upon queries.
- *Second*, considering Rowhammer's characteristics, Achilles formalizes Rowhammer-based faults against G-sign, where a parameter is faulted. Based on faulted G-sign, the other signing oracle $OFSign$ is defined to generate faulty signatures.
- *Third*, with $OSign$ and $OFSign$, we build a game-based model to check how a secret key can be recovered after a fault. Further, whether a fault occurs to pp or sp can be decided through the model.
- *Last*, depending on whether a faulty parameter belongs to pp or sp, `DFA` or `SCA` is used for the secret key recovery. As `SCA` cannot recover every secret bit, the state-of-the-art (SOTA) solution [22] recovers the remaining w bits with a high memory complexity of $\mathcal{O}(\sqrt{2^w})$. We propose a more efficient SCA algorithm with a memory complexity of only $\mathcal{O}(1)$.

To demonstrate the viability of Achilles, we have analyzed six signature schemes—BB short, EdDSA, ML-DSA, RSA, Elgamal, BLS —using an automated tool to identify their potentially vulnerable parameters. The signature schemes cover traditional and post-quantum signatures with different mathematical problems. Section 5 provides an analysis of BB short [27], and EdDSA [28] while Appendix A provides ML-DSA [29], RSA [30], Elgamal [31] and BLS [32]. While EdDSA has been faulted before [12], BB short and ML-DSA have not yet been studied. Based on Achilles, we identify that the secret key d in BB short, s in EdDSA, s_1 in ML-DSA, the hash value or inputs h, R, P, m in EdDSA, c, μ, w in ML-DSA, secret x in BLS can be vulnerable to Rowhammer attacks. We identify hash inputs R, P and secret key s as potentially new vulnerable parameters in EdDSA that have not been exploited before and find BB short and ML-DSA are also vulnerable.

To verify the potential vulnerability of the newly identified parameters in the schemes, we perform end-to-end Rowhammer attacks against these schemes from recent popular cryptographic libraries, i.e., `wolfssl-5.6.6`, `relic-toolkit-0.6.0`, and `liboqs-0.10.0` (in Section 6). Due to the varying implementation details of each scheme, there exists a challenge when faulting some parameters, i.e., the fault must occur in a fixed time window during the victim signing process. The window refers to the time period between a parameter's initialization and its use in subsequent cryptographic computations. If the fault occurs outside the window, it can either crash the signing process or fail to recover the secret key. Unfortunately, the window for some parameters is too small for a fault to be injected successfully.

To address this challenge, we leverage OS signals. Specifically, the attacker process signals the OS kernel upon detecting the completion of the targeted parameter's initialization. When the kernel receives the signal, it switches the signing process off the CPU, suspending it for a period of time. If the OS kernel is signalled frequently, the signing process remains suspended longer, indicating that the targeted parameters are not yet involved in any computation and can be faulted during this period.

To mitigate the attacks, we discuss possible countermeasures in Section 7, including our proposed one that has been adopted by `wolfssl` as an official patch.

Contributions: we have made four main contributions:

- To the best of our knowledge, we introduce the first formal framework that enables the revelation of potentially new vulnerable parameters and schemes. Our work underscores the importance of systematizing Rowhammer attacks targeting cryptographic schemes.
- We propose a new algorithm for recovering remaining bits after `SCA` through hash collision. This algorithm is much more memory-efficient than the SOTA [22].
- We perform three representative case studies of 6 schemes. These schemes feature diverse mathematical structures, including bilinear groups, elliptic curves, and lattices. Our case studies show that Achilles uncovers potentially new vulnerable parameters in EdDSA and new vulnerable schemes, i.e., BB short and ML-DSA.
- We conduct end-to-end Rowhammer attacks against the schemes using their recent cryptographic libraries, i.e., `wolfssl`, `relic`, and `liboqs` and use instantiated `DFA` or `SCA` with `SCARA` for the secret key recovery. Specifically, we demonstrate a full secret-key recovery of EdDSA and ML-DSA by faulting one of their public parameters. When faulting their private parameter, we have leaked 78, 85, 38 secret bits in BB short, EdDSA, and ML-DSA, respectively.

1.1 Responsible Disclosure

We have disclosed our findings to the security teams of `wolfssl`, `relic`, `crypto++`, and `liboqs`. We have been assigned 5 different CVEs (CVE-2024-2881, CVE-2024-1545, CVE-2023-51939, CVE-2024-28285, CVE-2024-31510), which track a fault-injection vulnerability in EdDSA, RSA, BB short, Elgamal, and ML-DSA, respectively.

2 Background and Related work

In this section, we introduce Rowhammer and its attacks against signature schemes.

Rowhammer is a software-induced fault in DRAM that can cause bit flips in the main memory of a commodity system [20]. Existing Rowhammer attacks against signature schemes target flipping critical parameters. Based on whether a parameter is public or private, we classify the attacks into the following categories.

Faulting *pp*: This category of attacks injects faults into public parameters, resulting in faulty signatures. An attacker can recover the secret key by comparing a valid signature with a faulty one. These attacks are effective since only two signatures are needed, but they necessitate knowledge of specific parameters for fault injection. In the RSA scheme, Boneh et al. [18] described the Bellcore attack, which injects faults into Chinese Remainder Theorem (CRT) implementation. Through a differential analysis of the faulty signature, the modulus N can be factored, indicating a leakage of the secret key. Bhattacharya et al. [13] utilized this attack with Rowhammer. For EdDSA, Poddebniak et al. [12] injected faults to the message and recovered the key by solving a linear equation.

Faulting *sp*: The attacker exploits a fault injection technique to induce a slight leakage in the secret parameter. By manipulating the faulty signature to appear valid, the attacker attains partial disclosure. This attack has been demonstrated on the nonce in ECDSA. Studies have revealed that even a small amount of leaked nonce bits can be leveraged to recover the secret key in ECDSA signatures [14, 33, 34]. Additionally, the possibility of leaking bits by intentionally faulting the secret key exists. Mus et al. [22] introduced a novel attack vector involving the injection of faults into the secret key, subsequently leaking the key bit by bit. In post-quantum schemes, Mus et al. [23] conducted a fault attack on a secret matrix within LUOV, achieving complete key recovery through further analysis. Islam et al. [24] focused on Dilithium by faulting a secret polynomial and have recovered partial of the secret.

3 Threat Model and Assumptions

Aligned with existing Rowhammer attacks targeting cryptographic schemes [13, 22, 23, 24, 35], we assume an attacker can launch an arbitrary unprivileged user process without root privileges on a modern Linux operating system. The attacker is familiar with the Linux OS's behavior, including its signal mechanism and Buddy Allocator. The OS is functioning correctly without any vulnerabilities. The attacker's objective is to induce desired bit flips in the underlying physical memory used by a victim process running a targeted cryptographic signature scheme. For the fault to be exploitable, the attacker needs the fault occurs in a specific key parameter of the scheme within a specific time window. The attacker is assumed to have knowledge of the scheme, including its secret key bit-length and public key, and can query the scheme during runtime.

The hardware resources are managed by the OS and shared among user processes, including both the attacker and the victim. The CPU is x86-based, allowing the attacker to leverage existing tools [36, 37] to reverse-engineer the machine's DRAM address mapping function. The physical memory is supported by DRAM modules (e.g., DDR3 and DDR4) with default refresh rates and is assumed to be vulnerable to Rowhammer-induced bit flips. The location and direction of these bit flips are specific to the DRAM module in use.

While modern DRAM modules such as DDR4 and DDR5 are equipped with Target Row Refresh (TRR) to mitigate Rowhammer, prior works have demonstrated that they remain vulnerable [38, 39]. Similarly, Error-Correcting Code (ECC) memory, once thought to provide robust protection against Rowhammer on enterprise-grade machines, has been shown to be insecure against such attacks [40]. In our end-to-end demonstration, the attacker considers non-ECC DRAM and uses TRRespass [38] to bypass TRR.

The OS runs memory-light tasks, enabling the attacker to utilize most of its available memory, where numerous 2 MB contiguous memory blocks are allocated. This eliminates the need to access the privileged interface `pagemap`. Within these allocated 2 MB memory blocks, the attacker leverages the reverse-engineered DRAM address mapping function and TRRespass [38] to identify sufficient Rowhammer-induced bit flips (in Section 6.2). To induce targeted bit flips in the victim's memory promptly, the attacker first manipulates the Buddy Allocator to coerce the victim into reusing vulnerable physical pages for storing targeted parameters. The attacker then leverages the signal mechanism to execute Rowhammer within a constrained time window, flipping the desired bits in the targeted parameters (in Section 6.3).

4 Achilles

In this section, we begin with an overview of Achilles, followed by a detailed description.

4.1 Overview

Figure 1 illustrates how Achilles works in four main steps, as described below:

- *First*, we establish a formal framework (denoted as G-sign) for a paradigm of representative real-world signature schemes including those based on different mathematical structures, e.g., bilinear groups, elliptic curves, and lattices. G-sign maintains a set of key parameters, classified as *pp* (public parameters) or *sp* (secret parameters), according to whether they can be publicly available. Using G-sign, main parameters of a given real-world scheme can be treated as either *pp* or *sp* for subsequent analysis in Section 5. *OSign* is defined to be a signing oracle that retrieves correct signatures from G-sign.
- *Second*, we formalize a Rowhammer fault against G-sign, targeting a specific parameter in *pp* and *sp*. With a faulted parameter from G-sign, we then design a "special" signing

$Gen(1^n)$	$Sign(pk,sk,m)$	$Vrfy(pk,m,\sigma)$
1 : $(pk,sk) \leftarrow Setup(1^n)$	1 : $r \leftarrow \mathcal{R}$ **or** $F(sk,m)$	1 : **return** 1 **or** 0
2 : **return** (pk,sk)	2 : $h \leftarrow H(pk,sk,m,r)$	
	3 : $\sigma \leftarrow P(pk,sk,h,r,m)$	
	4 : **return** σ	

Figure 1: A paradigm of representative signature schemes (denoted as G-sign).

oracle, *OFSign*, to obtain faulty signatures from faulted G-sign.

- *Third*, with *OSign* and *OFSign*, we build a game-based model to perform post-Rowhammer analysis for secret key recovery. Through the model, whether a fault occurs to pp or sp can be decided.
- *Last*, when a parameter from pp is faulted, DFA is used to compare the differences between a valid signature and a faulty one to recover the secret key. When a parameter from sp is faulted, SCA is used to correct a faulty signature to a valid one, leaking one secret bit. Since not all DRAM cells are vulnerable to Rowhammer faults, SCA cannot recover all the secret bits. To address this limitation, we employ Pollard Rho's birthday attack algorithm, which is more memory-efficient than the current SOTA [22].

4.2 Formal Treatment of Signature Schemes

As depicted in Figure 1, G-sign consisting of three algorithms $Gen, Sign$, and $Vrfy$.

- $Gen(1^n)$: On input security parameter n, the algorithm runs the probabilistic algorithm $Setup$ and generates public verification key pk and secret key sk respectively.
- $Sign(pk,sk,m)$: On input pk,sk, and a message m, the algorithm first generates r from the random space $\mathcal{R}$ or computing a deterministic function F on sk and m. Then, it computes a hash digest h via a function H on possible inputs pk,sk,m, and r, and finally output $\sigma = P(pk,sk,h,r,m)$ as the signature where P is the proving procedure.
- $Vrfy(pk,m,\sigma)$ on input pk, a message m and a signature σ, outputs 1 or 0 indicating accept or reject, respectively.

Clearly, in G-sign, the pp contains three parameters pk,h, and m that can be made public. The sp includes sk and r that are targeted by the attacker. Especially, G-sign can instantiate various real-world digital signatures, including ECDSA [41], EdDSA [28], BB short [27], and ML-DSA [29]. The last three will be discussed in our case studies.

4.3 A Game-based Model against G-sign

We first formalize Rowhammer faults against G-sign in Figure 2. Then, we provide a cryptographic security game modelling two types of leakage in Figure 3 based on our former formulation.

In Figure 2, we first define a signing oracle *OSign* using symbols in G-sign. Then, we provide a set of functions to formally define the Rowhammer fault. With the fault functions, we allow the attacker to access *OFSign* to obtain faulty signatures. Below, we elaborate on *OSign*, fault function $f_i(x)$, and *OFSign* .

- *OSign*: $OSign(pk,sk,m)$ is defined as a signing oracle that returns a signature for any message queried by the attacker. It stores an input message m in a set M and returns a valid signature σ generated by computing $Sign(pk,sk,m)$.
- $f_i(x)$: $\{f_i(x)\}$ is defined as a set of fault functions, each inducing a bit flip on a parameter x. Initially, the set of fault information f_i^{set} is set empty. In line 2, it generates a random number in the range $(0,|x|-1)$ as the position of a bit flip. In line 3, a binary addition $\oplus$ is operated on the i-th bit of x and 1 to simulate the faulting process. In lines 4-5, f_i^{set} stores the position k and value $x[k]$ of the bit flip and then returns the faulted x. The subscript i ranges from 1 to 11, denoting a fault to respective parameters in each step of the scheme.
- *OFSign*: $OFSign(i,pk,sk,m)$ is defined as a signing oracle that faults the i-th parameter and returns a faulty signature for a message m. Specifically, in line 1, only one fault function, f_i, is considered active, while the other functions are set to be void, meaning that they simply pass the input through to the output without any modification. In lines 2-4, we apply the fault functions $f_1,\ldots,f_{11}$ to all 11 parameters in G-sign, respectively. In lines 5-6, the information of the fault f_i^{set} is recorded in the set F_f. The set MF contains all messages that are sent to this faulty signing oracle. We note that in *OFSign*, Rowhammer-induced faults are *permanent*. For instance, if sk is faulted in line 3, then h would be faulty, and this fault can further affect line 4, leading to a faulty σ. Existing fault formulation [42, 43] do not account for this persistent nature of Rowhammer-induced faults.

With *OSign* and *OFSign*, we formalize a game-based model shown in Figure 3. Exp^F can be run by a challenger who wants to ensure the security of a signature scheme. In line 1, the set MF (resp., M) tracks messages sent to the *OFSign* (resp., *OSign*) that returns a faulty (resp., benign) signature and F_f tracks the information of the faults. Line 2 generates a key pair sk,pk. In line 3, the function $\mathcal{A}^{OSign(\cdot),OFSign(i,\cdot)}$ is invoked with pk as input and it returns a tuple consisting of a signature pair (m^*,σ^*) and faulty information f^*. As

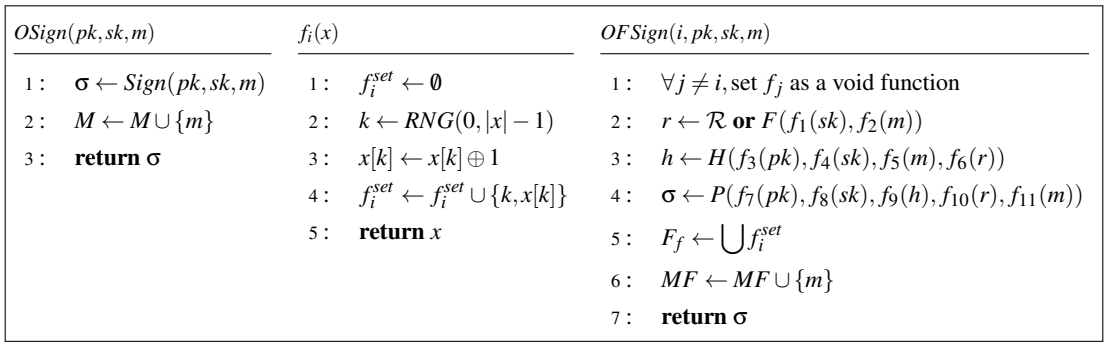

Figure 2: Formalizing Rowhammer faults against G-sign.

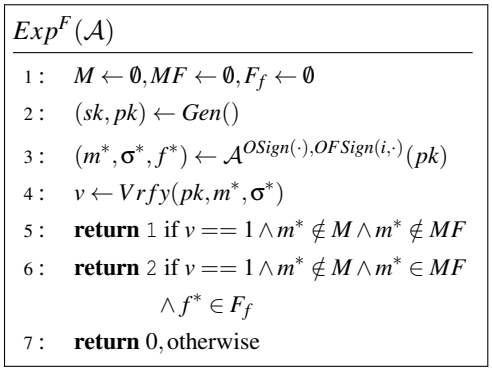

Figure 3: A game-based experiment modelling post-Rowhammer analysis.

discussed above, *OSign* and *OFSign* are oracles operated by the challenger, providing auxiliary information. In line 4, the challenger verifies the signature. If the verification succeeds, the verification result v will satisfy the condition check in either line 5 or line 6.

If the check on line 5 passes, three conditions are satisfied: $v == 1$, $m^* \notin M$ and $m^* \notin MF$. The condition $v == 1$ indicates that the signature pair (m^*, σ^*) is valid. The conditions $m^* \notin M$ and $m^* \notin MF$ confirm that the message m^* was not queried through the oracles *OSign* or *OFSign*. When these three conditions are met, it implies that m^* is a new message, and the attacker has successfully leaked *sk* and used it to generate a valid signature σ^* for m^*. Besides, the bit-flip fault denoted by $OFSign(i,\cdot)$ has occurred in pp. Leaking *sk* can be done through DFA, which assumes faults in pp and compares the difference between a faulty signature and a valid one to extract *sk*.

If the check on line 6 passes, four conditions are satisfied: $v == 1$, $m^* \notin M$, $m^* \in MF$ and $f^* \in F_f$. The first three conditions are similar to line 5 except that $m^* \in MF$ indicates the message m^* was previsouly queried to *OFSign*. The condition $f^* \in F_f$ signifies that the attacker knows the position and value of the bit flip when querying *OFSign* to generate a faulty signature for m^*. When these four conditions are met, it implies that m^* is a message previously sent to *OFSign* and that the attacker has succesfully corrected a faulty signature to a valid one using the faulty information f^*, leaking a bit of *sk*. As noted above, f^* represents bit-wise leakage of a faulty parameter, and in this case, the fault has occurred in *sp*. Otherwise, the attacker can compute *sk* via DFA and thus generate a valid signature without correcting the faulty one. The process of signature correction and *sk* recovery can be achieved through SCA.

4.4 Secret Key Recovery

In this section, we design DFA and SCA and explain them in detail as follows.

Analysing a faulty public parameter with DFA: In algorithm 1, the attacker faults a parameter in pp and obtains faulty and valid signatures through the signature query in lines 3-8 after initialization. In order to perform differential analysis and extract the secret key, the attacker makes a difference between the valid and faulty signatures. In line 7, $g(pp)$ is a function that can be computed with only knowledge of public parameters. If sk' has a linearity with $g(pp)$ such as $(\sigma - \sigma') = sk' \cdot g(pp)$, the secret key can be extracted.

Analysing a faulty secret parameter with SCA: In algorithm 2, the attacker tries to fault each secret parameter and output enough secret bits as leakage. Lines 1-2 are the initialization phase that defines and initializes certain symbols for the algorithm. Lines 3-5 find the index for each parameter in the for loop. Then, in lines 6-8, the attacker gains one faulty signature and begins to correct the faulty signature. To process one faulty signature, in lines 9-14, the attacker enumerates all possible faults and corrects them to be valid. If $Vrfy(pk \cdot g(pp, \Delta d), m, \sigma) = 1$ (where g is a public function that the attacker can compute), a successful correction is found, and the attacker can deduce the nature of the fault and store it as bit leakage. Note that in the signature scheme, *pk* is determined by *sk*, and a faulty *sk* can correspond to a different *pk*. Thus, we correct the signature by adding correction terms to *pk*. Finally, in line 16, if the quantity of the remaining bit is less than 50, SCA is finished. We observe that it is not always possible to completely leak all the secret bits, as it is difficult for Rowhammer to fault every single bit of a secret parameter. As such, we further propose an algorithm to

recover the remaining bits.

Algorithm 1: Post-Rowhammer analysis via `DFA`

Initially, $S = (Gen, Sign, Vrfy)$ is a signature scheme. $OFSign$ and $f_i(x)$ are function oracles defined in Figure 2.
Parse S to G-sign
foreach $p \in \{m, pk, h\}$ **do**
    `// get a valid and faulty signature.`
    $\sigma \leftarrow OSign(\cdot)$
    Use i to denote the index for parameter p
    $\sigma' \leftarrow OFSign(i, \cdot)$
    return $sk' = (\sigma - \sigma')/g(pp)$
end

4.4.1 SCARA: SCA remaining bits recovery

As SCA recover most secret bits, the time to recover the remaining bits is significant [22]. To address this issue for ECDSA, Mut et al. [22] proposed a modified baby-step giant-step. This algorithm improves efficiency for computing remaining bits, however, it uses a time/space tradeoff technique and the memory complexity is exponential to the number of remaining bits.

Here, we demonstrate that this is unnecessary by devising a generic algorithm to find the remaining bits through hash collision. The algorithm to find hash collision only needs two temporal variables in the while loop due to Theorem 1 (see below).

Our algorithm follows the paradigm of Pollard Rho's birthday attack algorithm for finding hash collisions. Through the use of a meticulously crafted hash function, our algorithm achieves the computation of remaining bits in time complexity $\mathcal{O}(\sqrt{2^w})$, where the number of remaining bits is denoted as w and employs constant memory complexity, $\mathcal{O}(1)$.

We divide x into two parts as the known and unknown bits of x, $x^{(k)}$ and $x^{(w)}$, denoted as $x = x^{(k)} + x^{(w)}$ where $k + w = n$. To take advantage of the k known bits of secret x, we modify public key pk to pk' with $x^{(k)}$. Then, the algorithm finds a hash collision based on pk'. For different schemes, we only need to customize a hash function to make sure that certain collisions can yield a solution for $x^{(w)}$. We give an example of BB short discussed in case study Section 5.1.

$$H_P(r_1, r_2) = g^{r_1} Q'^{r_2} \tag{1}$$

If a hash collision is found, which means $g^{r_1} Q'^{r_2} = g^{r'_1} Q'^{r'_2}$, the relation between g and Q' can be easily computed as $g^{x^{(w)}} = Q'$ where $x^{(w)} = (r_1 - r'_1) \cdot (r'_2 - r_1)^{-1}$.

A few technical details remain to be addressed. First, to achieve constant memory, we utilize a chain of hash values to find collisions. If a collision exists in two elements of the chain, the loop will terminate. The correctness is guaranteed by Theorem 1.

Algorithm 2: Post-Rowhammer analysis via `SCA`

Initialization: $S = (Gen, Sign, Vrfy)$ is a signature scheme; $OFSign$ is a function oracle defined in Figure 2; sk and r are key parameters defined in Figure 1; sk' stores the recovered bits and N is the number of recovered bits. Initially, sk' contains all 0 bit and N is 0; σ, Δ, i,j are all intermediate variables defined in the loop.
Parse S to G-sign
foreach $p \in \{sk, r\}$ **do**
    `// get a valid and faulty signature.`
    Use i to denote the index for parameter p
    do
        $\sigma' \leftarrow OFSign(i, \cdot)$
        `// enumerate the one-bit fault.`
        $\Delta \leftarrow 0$, $j \leftarrow 0$
        do
            if $Vrfy(pk \cdot g(pp, \Delta d), m, \sigma') = 1$ **then**
                $N{+}{+}$
                $sk' = sk' \oplus \Delta$
            end
            $\Delta \leftarrow \Delta << 1$, $j \leftarrow j + 1$
        while $j < |sk'|$
    while $|sk'| - recovered\_bits > 50$
end
return sk'

Theorem 1. Let $s_1, s_2, ..., s_q$ be a sequence of values with $s_m = f(s_{m-1})$, if $s_I = s_J$ with $1 < I < J \leq q$, then there is an $i < J$ such that $s_i = s_{2i}$. [44]

A hash output is used as input for the next hash, necessitating that the range of the hash function is a subset of its domain. Additionally, within the function domain, we impose the requirement that the bits not in the need-to-recover position be set to 0. This process is represented by functions E and H, ensuring that the return value of $H_{g,Q'}$ maps to the correct domain.

The complexity of this algorithm primarily depends on the number of iterations in the `while` loop. Since the hash value can be considered as randomly independent, the procedure for finding a collision essentially resembles a birthday attack, resulting in a time complexity of $\mathcal{O}(\sqrt{2^w})$. Moreover, due to using only two temporary variables during the loop, the space complexity remains at $\mathcal{O}(1)$.

5 Case Studies

In this section, we demonstrate how Achilles can be leveraged to analyse a given scheme's susceptibility to Rowhammer.

We have performed an analysis of 6 different schemes, i.e., BB short [27], EdDSA [28], ML-DSA [29], RSA [30], Elgamal [31] and BLS [32]. The analysis of BB short and EdDSA is provided below. For ML-DSA, RSA and Elgamal, they are included in Appendix A due to page limit.

Our case study is a walk-through of Achilles. First, we mathematically describe a signature scheme's implementation and transform it into G-sign, mapping its parameters to a list classified as pp or sp. Note that not all parameters in the scheme need to be mapped. Some immediate parameters in generating a signature are filtered out as they do not contribute to G-sign. Second, to satisfy the IF condition check in either line 5 or line 6 of Figure 3, we instantiate `DFA` or `SCA` to analyse each of the parameters that have been mapped. If the secret key can be covered by faulting a parameter, the parameter along with the scheme is potentially vulnerable. Based on our analysis, all the schemes above are potentially vulnerable to Rowhammer attacks, which are detailed below. In Section 6, we perform real-world attacks against each potentially vulnerable parameter. Besides, to accelerate the whole analysis, we have automated the second step of identifying potentially vulnerable parameters for a scheme, as detailed in Section 5.3.

5.1 Analysing BB short

In this analysis, we describe BB short [27], which is implemented in `relic-toolkit-0.6.0`, based on which we show how to leak the secret key via Rowhammer.

Gen(1^λ)**:** It generates a public key pk and a secret key sk. Particularly, this algorithm builds on groups $\mathbb{G}_1, \mathbb{G}_2$ with their generators g_1, g_2, respectively. The function e is defined as a bilinear map $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, which has holomorphic property such that $e(g^a, h^b) = e(g,h)^{ab}$. The function randomly picks s and further computes $q \leftarrow g_2^s$ and $z \leftarrow e(g_1, g_2)$. As such, the pk is generated as (g_1, g_2, e, q, z) and the sk is defined as s.

Sign*(pk, s, m)***:** It takes s and m as inputs, where s is the secret key and m is a message. Let H be a hash function. A signature σ is computed as $g_1^{1/(h+s)}$, where $h = H(m||q||z)$.

Vrfy*(pk, m, σ)***:** It takes a pair of (σ, m) and pk as inputs and generates 1 if the following equation holds:

$$e(\sigma, q \cdot g_2^h) = z \tag{2}$$

We first parse this scheme to G-sign as depicted in Table 1. `DFA` is not found because sk and h are on the denominator when calculating the signature. However, when analysing sk, we find BB short vulnerable to `SCA`.

When a single bit flip occurs to s right before $Sign(s, m)$ is invoked, the generated signature will become $\sigma' = g_1^{1/(h+s')}$, where σ' is a faulty signature caused by a faulty secret key s'.

Here, we denote s' as $s + \Delta s$ where Δs represents the injected fault. To make Equation 2 hold, Δs must satisfy the following equations:

Algorithm 3: `SCARA`: recover the remaining secret bits from `SCA`.

Input: public key pk; recovered t bits of secret key x; Hash function $H_1 : \{0,1\}^* \to \{0,1\}^n$; Hash function $H_P : \{0,1\}^n \times \{0,1\}^n$ is designed for underlying problem; a sample function $E : \{0,1\}^n \to \{0,1\}^n$, E set bits not reside in the need-to-recover position to 0; Represent function $f : \{0,1\}^n \to \{0,1\}^n$ as $f(s) \leftarrow H_P(E(H_1(s)))$.
Output: Secret key x.
Represent x as $x = x^{(k)} + x^{(w)}$
`// Eliminate known bits from SCA`
Compute $pk' = pk / g(x^{(k)})$
`// Use` $E(\{0,1\}^w)$ `to add extra` t `0s so that the result is` n `bits and the` w `bits are in the need-to-recover position`
$s_0 \leftarrow\$ E(\{0,1\}^w)$
$s := s' = s_0$
$i := 0$
do
  $s = f(s)$
  $s' = f(f(s'))$ `// now` $s = f^{(i)}(s_0)$ `and` $s' = f^{(2i)}(s_0)$
  $i = i + 1$
while $s \neq s'$
$s' := s$, $s := s_0$
for $j \leftarrow 1$ *to* i **do**
  if $f(s) = f(s')$ **then**
    $x^{(w)} = solve(f(s), f(s'))$
    return $x = x^{(k)} + x^{(w)}$
    break
  else
    $s = f(s)$
    $s' = f(s')$
  end
end

$$Vrfy(pk \cdot g(pp, \Delta s), m, \sigma') = 1 \tag{3}$$

and $g(pp, \Delta s) = g_2^{\Delta s}$. It is easy to see $g(pp, \Delta)$ can be computed without secret parameters and thus Equation 3 holds (as $e(\sigma', q \cdot g_2^h \cdot g_2^{\Delta s}) = 1$). We are able to find out the index of the bit flipped in s and thus recover its original bit. Finally, to recover the remaining bits, we use the hash collision algorithm with the hash function defined in Equation 1.

5.2 Analysing EdDSA

In this analysis, we describe the EdDSA implemented in `wolfSSL-5.6.6`, and we show how to leak the secret key via both `DFA` and `SCA`.

Scheme	pp			sp
BB short	h	m	q, g_1, g_2	s
G-sign	h	m	pk	sk

Table 1: Classification of potentially vulnerable parameters in BB short.

Gen(1^λ)**:** It generates a secret key (k,s) and a public key (P,B). In sk, k is used for hashing and s is generated by k. In pk, P is a curve point, and B is an ed25519 base point. Particularly, pk can be viewed as $pk=(P,B)$. Regarding sk, it is defined as $sk=(s,k)$. pk and sk satisfy $P=sB$ and $s=H(k)$.
Sign(pk,sk,m)**:** It takes pk,sk and m as inputs, utilizes H as hash function and generates a signature (R,σ) by computing $r=H(k||m), R=rB, h=H(R||P||m)$, and $\sigma=r+hs$.
Vrfy(pk,m,σ)**:** It takes a signature σ and $m, pk=(P,B)$ as inputs, and generates 1 if the following equation holds:

$$\sigma B = R + hP \tag{4}$$

Parameters in EdDSA can also be parsed as shown in Table 2. By faulting pp and sp, we find that EdDSA is vulnerable to both DFA and SCA.

Scheme	pp				sp
EdDSA	h	m	P,B	r,R	s
G-sign	h	m	pk	r	sk

Table 2: Classification of potentially vulnerable parameters in EdDSA.

Faulting public parameters with DFA: In EdDSA, R,P,M,h can be viewed as public parameters. When a single fault occurs in h, a differential analysis can result in the secret key s: $\sigma'=r+h's$. By combining the faulty signature with a valid one:

$$\sigma-\sigma' = (h-h')s, \tag{5}$$

s can be computed as $s=(\sigma-\sigma')\cdot(h-h')^{-1}$. Here $h-h'$ can be regarded as $g(pp)$ which satisfies the expression of DFA. Because R,P,m all serve as input for computing $h=H(R||P||m)$, we conclude that faulting these parameters can also yield a secret key.
Faulting secret parameters with SCA: When a single bit flip occurs to s when the *Sign* (m,pk,sk) function is invoked, the generated signature will become $\sigma'=r+hs'$, where (R,S') is a faulty signature caused by a faulty secret key s'.

Here, we denote s' as $s+\Delta s$ where Δs represents the injected fault. To make Equation 4 hold, Δs and $g(pp,\Delta)$ must satisfy the following equation:

$$\sigma' B = R + h(P + \Delta s B) \tag{6}$$

where $g(pp,\Delta)=\Delta sB$. When Equation 6 holds, we are able to find out the index of the bit flipped in s and thus recover its original bit.

Finally, to recover the remaining bits, we represent the secret key as $x=x^{(k)}+x^{(w)}$, compute $P'=P-s^{(k)}B$ and define the hash function for algorithm 3 as $H_P(r_1,r_2)=r_1B+r_2P'$. It is simple to verify that when a collision $H_p(r_1,r_2)=H_p(r_1',r_2')$ is found, we can compute $x^{(w)}$ as $(r_1-r_1')\cdot(r_2'-r_1)^{-1}$.

5.3 Identifying potentially vulnerable parameters automatically

The analysis described in Section 5.1 and Section 5.2 is performed manually. To accelerate the analysis and enable the automated identification of parameters' vulnerabilities, we have developed a Python program called *AutoVuln*. This program is designed to automatically identify potentially vulnerable parameters in a given signature scheme by leveraging mathematical symbols from the Python symbolic computation library `sympy`.

We assume that the program's users are either attackers or crypto-library developers seeking to analyze a scheme's vulnerability to Rowhammer efficiently. Accordingly, users are expected to have knowledge of the target scheme and must provide specific inputs about the scheme to the program for automatic analysis.
Inputs to AutoVuln: The format of AutoVuln invocation is as follows: *./AutoVuln -pk_list=[] -sk_list=[] -h_list=[] -m_list=[] -r_list=[] -sign_list=[] -vrfy_list=[]*. The program requires seven inputs. The first five inputs are five lists of parameters in the scheme mapped to (pk,sk,h,m,r) in G-sign. Thus, a program user must manually translate the scheme's implementation into the G-sign format, establishing a mapping between the scheme's parameters and the G-sign parameters. Examples of the mapping can be found in Table 1 and Table 2.

The sixth input is a list of relations among the scheme's mapped parameters during the signature generation. For instance, in EdDSA, these relations include σ=*r+h*s, R=r*B* and *P=s*B*. The final input is a list of relations among the parameters when verifying the signature. For EdDSA, it would be σ*B=R+h*P*. When analysing EdDSA, the provided inputs would be: *-pk=[P,B,R] -sk=[s] -h=[h] -m=[m] -r=[r] -sign_list=[*σ*=r+h*s,P=s*B,R=r*B] -vrfy_list=[*σ*B=R+h*P]*.
AutoVuln: With all the inputs, the program performs the three following steps.

- *Pre-processing*. In this step, the program converts the first five input lists (treated as strings) into *sympy* symbols and organizes them into corresponding symbol lists: *pk_sym_list*, *sk_sym_list*, *h_sym_list*, *m_sym_list*, and *r_sym_list*. Next, the program processes the equations in the last two input lists (*sign_list* and *vrfy_list*). Each equation is parsed into two expressions. For instance, the equation σ=*r+h*s* is split into *left_expr*(σ) and *right_expr*($r+h*s$). The program then uses the built-in *sympy* function *Eq(left_expr, right_expr)* to generate a

symbolized equation. These equations are stored in either *sign_sym_list* or *vrfy_sym_list*, depending on their source.

- *Analysing Public Parameters (pp).* In this step, the program performs DFA on each public parameter. For every parameter in *pk_sym_list*, *h_sym_list*, and *m_sym_list*, a new symbol fp is defined to represent a faulty version of the parameter. The program substitutes fp for the original parameter in the equations from *sign_sym_list*, producing a faulty signature expression. By applying symbolic extraction, the program relates the correct signature to the faulty signature, resulting in an expanded expression. For example, in EdDSA, if h is faulted, the substitution $h \rightarrow fp$ modifies the signature equation to σ=r+fp*s. Symbolic extraction yields: σ-σ'=$(h\text{-}fp)$*s. The program then checks whether all coefficients of the secret key are public parameters. If this condition is met (as in Line 7 of algorithm 1), the parameter is flagged as potentially vulnerable. In the case of EdDSA, the coefficient of s is h-fp, which is public. Thus, h is identified as a potentially vulnerable parameter.
- *Analysing Secret Parameters (sp).* In this step, the program performs SCA on each secret parameter. For every parameter in *sk_sym_list* and *r_sym_list*, the program introduces a new symbol df, representing a fault. This creates a symbolic expression for the parameter, such as s+df in EdDSA, which is substituted into the equations in *sign_sym_list* to generate a faulty signature equation. For example, in EdDSA, faulting s leads to the new expression s+df, modifying the signature equation to σ=r+h*$(s$+$df)$. The program extracts the coefficients of df and checks if they are public. If so, the coefficients with df can be subtracted from the faulty equation, allowing it to pass all the verification equations from *vrfy_sym_list* (satisfying Line 9 of algorithm 2). In the EdDSA example, since h is public, the term h*df can be subtracted, resulting in successful verification. As such, s is identified as a potentially vulnerable parameter.

Execution Results: The size of AutoVuln is 3.25 KB, and it has been evaluated against multiple cryptographic schemes, including BB short in `relic-toolkit-0.6.0`, EdDSA and RSA in `wolfSSL-5.6.6`, ML-DSA in `liboqs-0.10.0`, ElGamal in `cryptopp-8.9`, and BLS in `bls-signatures-2.0.3`. The results are summarized in Table 3. Execution for each scheme is efficient, completing in under 100 ms. Furthermore, the identified potentially vulnerable parameters align with the results from our manual analysis for BB short, EdDSA, and in Section 5.1 and Section 5.2, as well as for ML-DSA, Elgamal, RSA, and BLS in Appendix A.

Scheme	Library	Time	Identified Potentially Vulnerable Parameter				
			pk	*h*	*m*	*sk*	*r*
BB short	relic-toolkit-0.6.0	64.0 ms				s	
EdDSA	wolfSSL-5.6.6	75.2 ms	R,P	h	m	s	
ML-DSA	liboqs-0.10.0	85.0 ms	μ,w	c	m	s_1	
Elgmal	cryptopp-8.9	66.9 ms				x	k
RSA	wolfSSL-5.6.6	54.5 ms				s	
BLS	bls-signatures-2.0.3	52.2 ms				x	

Table 3: The execution time and potentially vulnerable parameters identified automatically by AutoVuln.

6 Evaluation

After identifying potentially vulnerable parameters in each scheme in Section 5, we now exploit Rowhammer to validate which parameter will lead to the recovery of the secret key. There are three main steps to achieve this as follows.

In the first step of *profiling memory*, we profile the physical memory allocated to the attacker, aiming to collect sufficient victim pages that contain Rowhammer bit-flips. In the second step of *faulting targeted parameter*, the attacker, co-located with the victim on the same system, tricks the system into using a victim page to store targeted parameters and then induces bit flips during signature generation. In the last step of *recovering secret key*, we respectively utilize DFA and SCA with SCARA to recover the secret.

In this section, we focus on faulting d in BB short, s,R,P,m,h in EdDSA and c,s_1,μ,w in ML-DSA, respectively. For each parameter, we will repeat the last two steps above.

6.1 Experimental Setup

We perform the experiments using a machine with Intel Core i3-10100 CPU (IceLake) and two Apacer DDR4-2666 8G DIMMs (part number: D12.2324WC.001). The machine runs default Ubuntu 22.04 with Linux kernel 6.1.66. The signature codes belong to recent popular libraries `relic-toolkit-0.6.0`, `wolfssl-5.6.6`, and `liboqs-0.10.0`. For BB short, we use the *Weiling* bilinear pairing and *secp224r1*, a.k.a, NIST P-224 elliptic curve. The secret key d has 224 bits. For EdDSA, we use a twisted Edwards curve where its order q is $2^{255} - 19$, a.k.a, curve25519. The secret key s in EdDSA is 256 bits. For ML-DSA, we use the parameter set `ML-DSA-44` where the secret key has 256 coefficients, and each coefficient ranges from -2 to 2.

6.2 Profiling Memory

In the attack process, we profile 12 GB of physical memory to identify sufficient victim pages containing bits susceptible to Rowhammer, along with their corresponding aggressor pages. To achieve this, it's crucial to know the mapping between a virtual address (VA) and a physical address (PA), as the physical address bits determine the DRAM bank and row. However, the OS kernel conceals the physical addresses from any unprivileged process, including the attacker.

To address this issue, we obtain 2 MB of contiguous memory blocks [45, 46] by leveraging the deterministic behaviour of the buddy allocator. In a 2 MB physical memory block,

a VA shares the least significant 21 bits with its corresponding PA. Based on prior work [36, 37], many x86 processors with different microarchitectures use these 21 bits in a PA to determine the corresponding DRAM bank. Therefore, we check whether the two VAs are in the same bank. Additionally, since the size of a DRAM row is much smaller than a 2 MB hugepage, we can determine whether two given VAs are in two adjacent rows.

To this end, we first use DRAMDig [37] to reverse engineer the DRAM address function of the machine. The results show that the bank index is determined by bit-wise operations against physical address bits of $(17,21),(16,20),(15,19),(14,18),(6,13)$, while the row index is determined directly by bits 18 to 33. Therefore, we have 32 banks decided by the five pairs of address bits and 2^{16} rows in a bank with each row of 2^{18} bytes, confirming that we can use a VA to decide its DRAM bank and whether two VAs are in the same row.

Next, we obtain 2 MB memory blocks as follows: we exhaust small free memory blocks (no greater than 2 MB) by using the `mmap` system call with the `MAP_POPULATE` flag. We then send memory requests of 2 MB via `mmap` to trick the kernel into splitting memory blocks of 4 MB, receiving multiple memory blocks of 2 MB where VAs and PAs have the same lowest 21 bits.

Last, we use TRRespass[3] to find an effective hammer pattern against the DIMMs. This refers to the number of aggressor rows that are needed to trigger bit flips. As our DIMMs are DDR4 modules and can be protected by target row refresh (TRR) [47], TRRespass can bypass TRR and return the expected hammer pattern for inducing bit flips. The results show that a double-sided hammer can trigger reproducible bit flips. With the identified hammer pattern, we start our hammering attempts to profile the memory within the process. Each hammering attempt is a finite loop of hammering the 4 pairs of virtual pages, with each pair hammered 250,000 rounds. After each hammering attempt, we scan other unhammered memory for bit flips. By doing so, we collect a set of victim pages with their paired aggressor pages. The page offset of a single-bit flip can differ in the collected victim pages. Figure 4 shows the distribution of bit-flip offsets accumulated over 4 KB-aligned pages.

6.3 Faulting Targeted Parameter

With the profiled memory, we release vulnerable pages back to the OS kernel and coerce it into reusing these pages to host targeted parameter, so-called *memory massaging* [46, 48, 49]. In our case, the pages have flippable bits that have the same 4 KB-page offset as the targeted parameter in the victim process (Note that for each parameter in a library binary, their offset within the page remains consistent, even though the base address of the virtual page where the parameter resides is randomized by address space layout randomization).

[3] https://github.com/vusec/trrespass

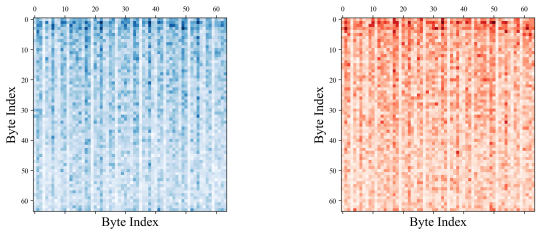

Figure 4: The distribution of flippable-bit offsets over 4 KB-aligned pages. Bit flips from 1 to 0 (blue) and bit flips from 0 to 1 (red) accumulated over 4 KB pages.

Particularly, we exploit the predictable behavior of the Linux Buddy Allocator, which allocates recently unmapped physical pages on the page frame cache (`per-cpu pageset`) in a first-in-last-out (FILO) order. Therefore, we unmap specific victim pages via `mmap` and trigger the victim process into using one of the unmapped pages to load the parameter.

The last step is to induce bit flips to all the pages, with one of them causing a fault to the parameter. However, the fault must occur in a limited time window between a parameter's initiation and its subsequent use in computations. If a bit flip occurs outside the window, it can either disrupt the signing process or render itself ineffective for recovering the secret key.

To address this issue, we opt for OS signals. Specifically, the attacker process running on one core registers a signal handler and then triggers the victim signing process on another core. In the meantime, the attacker sets up a timer by using an unprivileged high-resolution timer (e.g., `rdtsc`) to clock the time of the signing process. Upon the completion of a target parameter's initialization, the attacker process sends the registered signal to trick the OS kernel into switching the victim process off the core, extending the window. If the time window is small, the victim process can be switched off frequently.

6.4 Serect Key Recovery

After faulting a targeted parameter and receiving sufficient signatures, we use either `DFA` or `SCA` with `SCARA` to recover the secret key in each scheme.

Recovering the secret bits via `DFA`: When a public parameter is faulted, only one faulty signature is needed to recover the secret key. In EdDSA, `fe_inv`, `fe_mul` and `fe_add` functions defined in `fe_low_mem.c` are to perform field and curve operations. They are used to compute the secret key. With the operations and exploitable signature after faulting P and m, respectively, we have recovered the secret key. In ML-DSA, c is a ring polynomial, and we use `poly_invntt_tomont`, `poly_add` and `poly_sub` functions defined in `poly.c` to com-

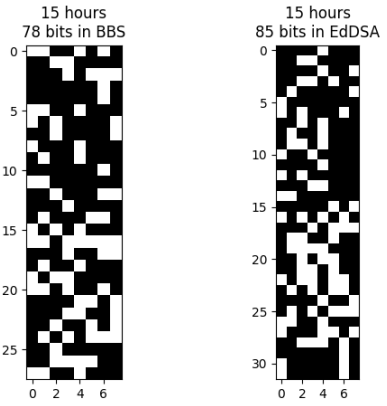

Figure 5: The number of recovered bits in 15 hours for the secret key in BB short and EdDSA is represented in the form of pixels, where white cells indicate recovered bits, and black cells indicate non-recovered bits.

pute the secret key. With the operations and exploitable signature after faulting c, we have recovered the secret key.

Recovering secret bits via SCA: When a private parameter is faulted, hundreds of unique faulty signatures are required (depending on the bit length of the secret key). Figure 5 shows the number of secret bits that have been recovered in 15 hours via SCA in BB short and EdDSA. Specifically, in BB short, we have recovered 78 unique bits using 631 faulty signatures out of a total of 1822. In EdDSA we have leaked 85 unique secret bits using 834 faulty signatures out of 4120 in total.

In ML-DSA, the secret key s_1 is defined as a 4-dimensional vector consisting of $s_1^{(1)}, s_1^{(2)}, s_1^{(3)}, s_1^{(4)}$. Each element in the vector is a polynomial of 256 degrees with coefficients in the range of $[-2, 2]$. Each coefficient is stored as an `int32_t`, but only the 3 LSBs represent useful values, while the other 29 bits serve only as a positive or negative flag. There are 5 possible values for three LSBs: $110, 111, 000, 001, 010$. In our experiments, we observe that if a fault occurs in the 16 MSBs of a coefficient, the rejection condition in line 18 in algorithm 5 terminates with high probability. If a fault occurs in bits 3 through 15, the recovered information is redundant, as these bits are the same as the third LSB.

Since we could not find exploitable bit flips in the 3 LSBs of our evaluated Apacer DDR4 DIMMs, we instead used a more vulnerable Samsung DDR3-1300 4G DIMM (part number: M473B5273DH0-YK0) deployed in a Lenovo T420 equipped with Intel Core i5-2430M CPU (Sandy Bridge). This machine operates on the same OS and kernel as the previously evaluated DDR4-based machine. Using DRAMDig [37], we reverse-engineered the DRAM address mapping of this machine. The results reveal that the bank index is determined by the physical address bits $(16, 20), (15, 19), (14, 18), (13, 17)$, while the row index is derived from bits 17 to 31. Within 6 hours of double-sided hammering on 0.6 GB of the DDR3 memory, we observed 1034 bit flips from 1 to 0 and 1339 from 0 to 1. Based on the profiling results, we successfully induced faults in the 3 LSBs of s_1, recovering 38 unique bits in 12 hours.

SCARA: Table 4 presents a comparison of brute-force, Jolt [22], and SCARA with respect to their memory and time overhead when recovering 55 remaining secret bits in BB short and EdDSA. While brute-force fails in the recovery for each scheme, SCARA only utilizes 0.65% memory of Jolt's [22] on average. The reason EdDSA requires more memory and time than BB short is that its elements curve are larger, and the operations are slower on the `secp256r1` curve than `secp192r1` curve in BB short. SCARA is unsuitable for ML-DSA because the secret key is stored in a large array containing 32,768 bits (4096×8), which significantly exceeds SCARA's capacity.

	BB short		EdDSA	
	Memory (MB)	Time (s)	Memory (MB)	Time (s)
brute-force	21.4	timeout	21.7	timeout
Jolt [22]	5810.2	960.5	7240.5	1201.1
SCARA	21.6	770.4	21.8	1160.2

Table 4: The memory and time overhead when recovering the remaining secret bits using brute-force, Jolt [22] and our SCARA, respectively.

7 Countermeasures

In this section, we first discuss specific solutions to mitigate fault-injection attacks against signature schemes. We then talk about general strategies to counteract Rowhammer.

7.1 Mitigating Faults on Signature Schemes

Leakage resilient cryptographic schemes: Assuming a small number of bits in sp that can be leaked via faults or side channels, some works have proposed new cryptographic primitives that are proven to be secure against a full leak of sk. Specifically, these primitives define a relatively long sk and establish a leakage threshold based on its length. Even if some bits of sk below this threshold are leaked, the entire sk remains secure against SCA. However, these primitives remain vulnerable to DFA, which requires only a single correct signature and a faulty one. The proposed primitives include signature schemes [50, 51], key encapsulation [52, 53], and secret sharing protocols [54].

Verifying after signing: When a fault occurs in either pp or sp, a faulty signature will be generated. Considering that verifying shown in Figure 1 can detect the fault by rejecting the faulty signature, the cryptographic library developer can append it right after signing. However, it can be bypassed if another fault occurs to specific verifying implementation [16]. Besides, verifying can incur significant overhead to signing.

Redundancy check: The redundancy check includes temporal and spatial redundancy check [24]. A temporal redundancy check involves re-executing signing and comparing the resulting signatures. If the signatures differ, a Rowhammer fault can be detected. However, this check can be bypassed if the fault occurs in all signatures, as Rowhammer induces a permanent fault. A spatial redundancy check stores multiple copies of the same secret parameters in randomly selected DRAM locations and compares them at the end of the signing. If the copies differ, a Rowhammer fault is detected. We proposed this check in `EdDSA`, and it has been adopted by `wolfssl` as an official patch.

In Section 7.3, we have demonstrated that both verifying-after-signing and redundancy check implemented by `wolfssl` are insufficiently secure against Rowhammer attacks.

7.2 Mitigating Rowhammer Attacks

RowHammer defenses can generally be categorized into hardware-based and software-only approaches (we refer the reader to [55] for more details.). Many hardware-based solutions [20, 56, 57] have been proposed in academia to strengthen DRAM and mitigate RowHammer-induced bit flips. These solutions require hardware modifications and have yet to be adopted in the industry due to cost and complexity. For industrially hardened DRAM modules such as DDR4 and DDR5, they have been shown to remain vulnerable to RowHammer attacks.

For instance, ECC DRAM can detect or correct bit flips in memory, providing effective mitigation but not complete immunity against RowHammer. ECCploit [40] demonstrated that RowHammer-induced bit flips persist even with ECC DRAM. Besides, ECC DRAM is predominantly used in servers and workstations and is rarely used in consumer PCs. Another widely used solution is TRR DRAM. It tracks row activations and refreshes adjacent rows when activation counts reach a pre-defined threshold, suppressing bit flips. However, TRR DRAM has also proven unable to eliminate RowHammer [38, 39, 58]. While these industrial solutions cannot fully eliminate RowHammer, they can influence the number and locations of bit flips in DRAM. This reduces the likelihood of attackers finding sufficient vulnerable locations to exploit RowHammer for targeting parameters identified by Achilles. Some types of RAM inherently avoid the root cause of RowHammer: electrical interference between adjacent cells caused by charge storage. For example, magnetic RAM (MRAM) stores data using magnetic states rather than electrical charges, rendering it immune to RowHammer. However, MRAM adoption remains limited and primarily confined to niche applications.

Unlike them, software-only solutions [59, 60, 61, 62, 63] require no hardware modifications, making them compatible with legacy hardware, including DDR3. One effective approach involves increasing the refresh rate of DRAM cells to suppress RowHammer-induced bit flips. For example, computer manufacturers such as Lenovo [64] have updated firmware to reduce the refresh period of DDR3 modules from 64 ms to 32 ms. While this significantly reduces RowHammer bit flips, it also impacts OS performance. Moreover, RowHammer-induced bit flips persist even with the reduced refresh period [59], making this approach cost-ineffective. Some software-based solutions [60, 61, 62, 63] do not attempt to suppress RowHammer. Instead, they mitigate its exploitation by preventing sensitive data or code (e.g., targeted parameters in Section 6.3) from being placed onto vulnerable rows. These solutions typically modify the OS memory allocator to enforce DRAM-aware memory isolation between software entities. The relevant example is RIP-RH [62], which isolates DRAM rows allocated to different processes by inserting guard rows in between. This prevents an attacker process from faulting key parameters or opcodes in a victim process, thus effectively mitigating our Rowhammer attacks. However, all these solutions fail to account for bit flips occurring more than 6 rows away from the hammered rows [65].

7.3 Serect Key Recovery from Hardened Signature Implementations

In this section, we target `wolfSSL-5.7.4`, which implements verifying-after-signing to harden RSA and a redundancy check to harden EdDSA. These hardened implementations are designed to detect the generation of faulty signatures. If a fault is detected, the signature scheme aborts, thereby mitigating Rowhammer attacks against key parameters. However, we demonstrate that both mitigations can be circumvented by faulting their respective critical instructions. Once the bypass is achieved, a targeted parameter can be faulted to recover the secret key (as detailed in Section 6.3). Below, we discuss the location of these critical instructions and the technique for faulting them in RSA and EdDSA.

RSA with verifying-after-signing: The verifying-after-signing mitigation is implemented in C as a macro named `WOLFSSL_CHECK_SIG_FAULTS`. This macro verifies a generated signature and returns the verification result in the `ret` variable. After executing the macro, RSA uses an `IF` condition to check the value of `ret`. If `ret` is not 0, the signature is deemed faulty, and the execution aborts. Otherwise, the execution continues.

By analyzing the compiled binary of RSA, we observe that the `IF` condition involves a `CMP` instruction and a `JE` instruction. The `CMP` instruction compares `ret` with 0. If `ret` equals 0, the `JE` (jump if equal) instruction directs execution to a specific location for continuation. Otherwise, the jump is not taken, and the execution aborts.

The `JE` instruction is a short jump with the opcode $0x74$. If this opcode is faulted to $0x75$ through a single-bit flip, the instruction changes to `JNE` (jump if not equal). This modification causes execution to continue even when the signature is

faulty, effectively bypassing verifying-after-signing. Consequently, we target the JE instruction for fault injection.

EdDSA with redundancy check: For EdDSA with redundancy checks, a similar mitigation is implemented using the macro WOLFSSL_EDDSA_CHECK_PRIV_ON_SIGN. This macro compares two arrays containing copies of the same secret key after the signing procedure. The result of the comparison is stored in the ret variable, which is returned. An IF condition then checks the value of ret. If ret equals 0, indicating the arrays are identical, the execution continues. Otherwise, the execution aborts.

Unlike RSA, the IF condition for EdDSA uses a CMP instruction and a JNE (jump if not equal) instruction. When a fault occurs in the secret key, ret is not 0, and the JNE instruction triggers a jump to abort execution. The JNE instruction is also a short jump with the opcode $0x75$. If this opcode is faulted to $0x74$ via a bit flip, the instruction changes to JE (jump if equal). This prevents the jump from being taken when the key is faulted, thereby bypassing the redundancy check. We note that while faulting the targeted opcode bypasses the mitigation, the corresponding signature scheme will abort if its secret parameter is not faulted.

Faulting Targeted Opcode: To fault the targeted opcode in RSA and EdDSA, we employ the memory massaging technique discussed in Section 6.3 to manipulate the OS into reusing a vulnerable page to host the targeted opcode. The vulnerable page, collected in Section 6.2, must contain a flippable bit that corresponds to the same 4 KB-page offset as the bit to be faulted in the targeted opcode. For RSA, we ran 1000 trials and succeeded on the 676th attempt by faulting the targeted opcode. Subsequently, we faulted its 1024-bit secret key d. In total, we obtained 861 faulty signatures in 8 hours and recovered 65 unique bits from them. For EdDSA, we also conducted 1000 trials and succeeded on the 908th attempt by faulting the targeted opcode. Following this, we faulted its 256-bit secret key s, resulting in 679 faulty signatures in 8 hours. From these signatures, 35 unique secret bits were successfully recovered.

8 Discussion

In this section, we discuss the limitations of our work and compare it with prior works on physical fault injection.

8.1 Limitations

Currently, Achilles models standardized signature schemes, including traditional and post-quantum schemes. Non-standardized signature schemes (e.g., threshold signatures [66] and ring signatures [67]), are excluded due to their unique characteristics. These schemes assume multiple signers and their signing queries are not modelled by G-sign. Extending Achilles to include non-standardized signature schemes will be valuable.

Besides, signature schemes exhibit varying levels of robustness against post-Rowhammer analysis. Schemes with longer secret keys are more resilient to SCA, as a longer key requires faults in more bits for a full recovery. However, these schemes, regardless of the key length, remain vulnerable to DFA, which derives the secret key by comparing a valid and a faulty signature and is unaffected by the key length.

Last, the demonstrated Rowhammer attacks are limited to the Linux environment, where predictable behaviours of Linux signal mechanisms and Buddy Allocator are abused. The evaluated DDR4 DIMMs are vulnerable to Rowhammer: bit flips from 1 to 0 or vice versa, can occur at most page offsets as shown in Figure 4.

8.2 Physical fault injection

There are numerous physical techniques for injecting faults, with optical radiation, clock glitches, voltage glitches, electromagnetic interference (EM), and heating being prominent examples [68]. Optical radiation employs a range of optical methods, such as laser beams [69], focused ion beams [70], X-rays [71], and flashes [72], to target specific areas of a device and disrupt its normal operations. Clock glitches [73] and voltage glitches [74] manipulate a device's clock and voltage signals, respectively, to induce faults that can result in malfunctions or unintended behavior. EM [75] involves using electromagnetic pulses to disrupt a device's circuitry, inducing faults in its operation. Heating [76] intentionally raises the temperature of specific device components, causing circuit malfunctions.

Physical fault-injection techniques can generally be categorized into invasive (e.g., optical radiation) and non-invasive methods (e.g., clock/voltage glitches, electromagnetic interference, and heating). Invasive techniques require direct physical access to a device's internal structure to induce faults, while non-invasive ones induce faults by manipulating the device's external environments. We now compare these fault-injection techniques with our Rowhammer attacks as follows.

Targeted device: Physical fault-injection techniques require proximity or direct access to the targeted device, along with specialized and costly equipment, making them well-suited for embedded systems such as FPGA SoCs and smart cards. In contrast, Rowhammer represents a software-induced, non-invasive fault injection approach that is cost-effective. It specifically targets DRAM modules commonly found in consumer-grade and server-grade platforms, making it especially relevant in scenarios such as multi-process commodity operating systems or multi-tenant public clouds.

Spatial and temporal precision: Optical radiation techniques provide high spatial and moderate temporal precision for inducing faults [68]. For instance, laser beam utilizes control devices like digital oscilloscopes to pinpoint targets at the bit level. While the timing of the fault can be partially controlled due to the speed and accuracy of optical radiation,

these techniques are unsuitable for Achilles. This is because certain key parameters in the signing process must be faulted within a narrow time window, which optical radiation cannot reliably detect or control. Instead, optical radiation techniques are typically applied to tasks like instruction skipping [77], which can be executed after the process has been loaded.

EM provides moderate spatial and temporal precision. It typically requires a motorized positioning table, a signal generation module, and an oscilloscope to ensure precise control over the fault location and timing. While EM is less accurate than optical radiation techniques due to the diffused nature of electromagnetic fields, it can still achieve single bit-flip faults [78]. The timing for inducing faults is comparable to that of optical radiation.

Clock and voltage glitches exhibit low spatial precision and moderate temporal precision. These techniques induce faults at the circuit level by disrupting input signals, leading to imprecise fault localization and causing multi-bit errors. Similarly, heating offers low spatial and temporal precision, as neither the specific fault location nor the fault timing can be controlled with fine granularity. Consequently, these methods are unsuitable for Achilles, which requires precise bit-level granularity and timing for inducing single-bit faults.

Unlike these methods, our Rowhammer attack, as discussed in Section 6, achieves high spatial precision by doing *memory profiling* and *memory massaging*. It advances existing Rowhammer attacks by leveraging Linux OS signals to overcome the challenge of the limited time window for fault injection, resulting in high temporal precision. Therefore, Achilles is uniquely powered by Rowhammer, rendering none of the aforementioned physical fault injection techniques applicable to Achilles.

9 Conclusion and Future Work

In this paper, we presented a formal framework, coined as Achilles, that describes a formal procedure to induce Rowhammer faults to a generalized signature scheme and perform a post-Rowhammer analysis for secret recovery. With Achilles, potentially vulnerable parameters in real-world signature schemes have been found. To validate their vulnerability, we perform Rowhammer attacks against the signature schemes and recover their secrets via `DFA` and `SCA` with `SCARA`.

Achilles specializes in signature schemes, which allow adversaries to submit multiple signing queries, with the resulting signatures being publicly available. This public accessibility makes such schemes vulnerable to fault-based secret recovery. Expanding Achilles to include non-standardized signature schemes and other cryptographic primitives, such as encryption and key agreement, would be valuable for uncovering and fixing more vulnerable schemes.

10 Ethics Statements and Open Science Policy Compliance

Ethics statements: In this paper, our case study and evaluation involve real-world attacks on cryptographic open-source projects. We have made disclosures to all the vendors that our attack would have influenced and we have provided countermeasures. We have collaborated with `wolfssl` security team to deploy patches in their library. Other libraries claim that fault injection is currently not part of their security model and have a notification for the users in their disclaimers. As our attack targets general signature schemes, there may be negative potential outcomes in other systems that use a signature scheme. We have discussed various countermeasures that can be used by real-world applications to mitigate the vulnerability. We hope this work raises awareness about Rowhammer fault injection when implementing and/or deploying cryptographic schemes and motivates the community to apply more corresponding countermeasures for Rowhammer attacks.
Open policy: We are dedicated to upholding the values of the Open Science Policy and strive to encourage transparency, reproducibility, and collaboration in scientific research. We release our artifacts including the source code and demos at `https://doi.org/10.5281/zenodo.14735639` as per the conference's requirements.

References

[1] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Protocol Architecture. RFC 4251, January 2006.

[2] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[3] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[4] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1:22–23, 2013.

[5] Hitesh Dhall, Dolly Dhall, Sonia Batra, and Pooja Rani. Implementation of ipsec protocol. In *International Conference on Advanced Computing & Communication Technologies*, pages 176–181, 2012.

[6] Craig Gentry. Certificate-based encryption and the certificate revocation problem. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 272–293. Springer, 2003.

[7] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM*

special interest group on data communication, pages 183–196, 2017.

[8] Jeremy Clark and Paul C Van Oorschot. Sok: Ssl and https: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Symposium on Security and Privacy*, pages 511–525, 2013.

[9] George Arnold Sullivan, Jackson Sippe, Nadia Heninger, and Eric Wustrow. Open to a fault: On the passive compromise of {TLS} keys via transient errors. In *USENIX Security Symposium*, pages 233–250, 2022.

[10] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a needle in the software stack. In *USENIX Security Symposium*, pages 1–18, 2016.

[11] 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.

[12] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. In *IEEE European Symposium on Security and Privacy*, pages 338–352, 2018.

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

[14] Keegan Ryan. Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 181–194, 2019.

[15] Weiqiong Cao, Hongsong Shi, Hua Chen, Jiazhe Chen, Limin Fan, and Wenling Wu. Lattice-based fault attacks on deterministic signature schemes of ecdsa and eddsa. In *Cryptographers' Track at the RSA Conference*, pages 169–195. Springer, 2022.

[16] Puja Mondal, Suparna Kundu, Sarani Bhattacharya, Angshuman Karmakar, and Ingrid Verbauwhede. A practical key-recovery attack on lwe-based key-encapsulation mechanism schemes using rowhammer. *arXiv preprint arXiv:2311.08027*, 2023.

[17] Gabrielle De Micheli and Nadia Heninger. Recovering cryptographic keys from partial information, by example. *Cryptology ePrint Archive*, 2020.

[18] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

[19] Michael Fahr Jr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, et al. When frodo flips: End-to-end key recovery on frodokem via rowhammer. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 979–993, 2022.

[20] 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 *International Symposium on Computer Architecture*, pages 361–372, 2014.

[21] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of cryptology*, 14:101–119, 2001.

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

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

[24] Saad Islam, Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. Signature correction attack on dilithium signature scheme. In *IEEE European Symposium on Security and Privacy*, pages 647–663, 2022.

[25] Nguyen and Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15:151–176, 2002.

[26] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. Big numbers-big troubles: Systematically analyzing nonce leakage in ({EC) DSA} implementations. In *USENIX Security Symposium*, pages 1767–1784, 2020.

[27] Dan Boneh and Xavier Boyen. Short signatures without random oracles and the sdh assumption in bilinear groups. *Journal of cryptology*, 21(2):149–177, 2008.

[28] Thomas Pornin. Deterministic usage of the digital signature algorithm (dsa) and elliptic curve digital signature algorithm (ecdsa). Technical report, 2013.

[29] National Institute of Standards and Technology. Module-lattice-based digital signature standard. Technical report, U.S. Department of Commerce, 2023.

[30] Evgeny Milanov. The rsa algorithm. *RSA laboratories*, pages 1–11, 2009.

[31] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[32] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2002.

[33] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. {TPM-FAIL}:{TPM} meets timing and lattice attacks. In *USENIX Security Symposium*, pages 2057–2073, 2020.

[34] Keegan Ryan. Return of the hidden number problem.: A widespread and novel key extraction attack on ecdsa and dsa. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 146–168, 2019.

[35] Chihun Song, Michael Jaemin Kim, Tianchen Wang, Houxiang Ji, Jinghan Huang, Ipoom Jeong, Jaehyun Park, Hwayong Nam, Minbok Wi, Jung Ho Ahn, et al. Tarot: A cxl smartnic-based defense against multi-bit errors by row-hammer attacks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 981–998, 2024.

[36] 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.

[37] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. Dramdig: A knowledge-assisted tool to uncover dram address mapping. In *Design Automation Conference*, 2020.

[38] 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 *IEEE Symposium on Security and Privacy*, pages 747–762, 2020.

[39] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. ZenHammer: Rowhammer attacks on AMD zen-based platforms. In *USENIX Security Symposium*, 2024.

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

[41] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1:36–63, 2001.

[42] Diego F Aranha, Claudio Orlandi, Akira Takahashi, and Greg Zaverucha. Security of hedged fiat–shamir signatures under fault attacks. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 644–674, 2020.

[43] Marc Fischlin and Felix Günther. Modeling memory faults in signature and authenticated encryption schemes. In *Topics in Cryptology–CT-RSA 2020: The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24–28, 2020, Proceedings*, pages 56–84. Springer, 2020.

[44] Jeong Han Kim, Ravi Montenegro, and Prasad Tetali. Near optimal bounds for collision in pollard rho for discrete log. In *Annual IEEE Symposium on Foundations of Computer Science*, pages 215–223, 2007.

[45] Yueqiang Cheng, Zhi Zhang, Surya Nepal, and Zhi Wang. Cattmew: Defeating software-only physical kernel isolation. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[46] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading bits in memory without accessing them. In *IEEE Symposium on Security and Privacy*, 2020.

[47] Micron, Inc. 8gb: x4, x8, x16 ddr4 sdram features-excessive row activation. https://www.micron.com/products/dram/ddr4-sdram, 2020.

[48] 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 *ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689, 2016.

[49] 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 *IEEE Symposium on Security and Privacy*, pages 245–261, 2018.

[50] Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, and Mor Weiss. Zk-pcps from leakage-resilient secret sharing. *Journal of Cryptology*, 35(4):23, 2022.

[51] Jianye Huang, Qiong Huang, and Willy Susilo. Leakage-resilient group signature: Definitions and constructions. *Information Sciences*, 509:119–132, 2020.

[52] Dana Dachman-Soled, S Dov Gordon, Feng-Hao Liu, Adam O'Neill, and Hong-Sheng Zhou. Leakage-resilient public-key encryption from obfuscation. In *Public-Key Cryptography*, pages 101–128. 2016.

[53] Xin Li, Fermi Ma, Willy Quach, and Daniel Wichs. Leakage-resilient key exchange and two-seed extractors. In *Annual International Cryptology Conference*, pages 401–429, 2020.

[54] Nishanth Chandran, Bhavana Kanukurthi, Sai Lakshmi Bhavana Obbattu, and Sruthi Sekar. Short leakage resilient and non-malleable secret sharing schemes. In *Annual International Cryptology Conference*, pages 178–207, 2022.

[55] Zhi Zhang, Decheng Cheng, Jiahao Qi, Yueqiang Cheng, Shijie Jiang, Yiyang Lin, Yansong Gao, Surya Nepal, Yi Zou, Jiliang Zhang, and Yang Xiang. SoK: Rowhammer on commodity operating systems. In *Asia Conference on Computer and Communications Security*, 2024.

[56] Michele Marazzi, Patrick Jattke, Solt Flavien, and Kaveh Razavi. PROTRR: Principled yet optimal in-dram target row refresh. In *IEEE Symposium on Security and Privacy*, 2022.

[57] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. Csi: Rowhammer-cryptographic security and integrity against rowhammer. In *IEEE Symposium on Security and Privacy*, pages 236–252, 2023.

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

[59] 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. In *Architectural Support for Programming Languages and Operating Systems*, pages 743–755, 2016.

[60] 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 *USENIX Security Symposium*, 2017.

[61] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: comprehensive and compatible software protection against rowhammer attacks. In *Operating Systems Design and Implementation*, pages 697–710, 2018.

[62] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. RIP-RH: Preventing rowhammer-based inter-process attacks. In *Asia Conference on Computer and Communications Security*, pages 561–572, 2019.

[63] Zhi Zhang, Yueqiang Cheng, Minghua Wang, Wei He, Wenhao Wang, Nepal Surya, Yansong Gao, Kang Li, Zhe Wang, and Chenggang Wu. Softtrr: Protect page tables against rowhammer attacks using software-only target row refresh. In *USENIX Annual Technical Conference*, 2022.

[64] LENOVO, Inc. Row hammer privilege escalation lenovo security advisory: Len-2015-009. https://support.lenovo.com/au/en/product_security/row_hammer, Aug. 2015.

[65] 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 *International Symposium on Computer Architecture*, 2020.

[66] Yehuda Lindell. Fast secure two-party ecdsa signing. In *Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part II 37*, pages 613–644, 2017.

[67] Shi-Feng Sun, Man Ho Au, Joseph K Liu, and Tsz Hon Yuen. Ringct 2.0: A compact accumulator-based (linkable ring signature) protocol for blockchain cryptocurrency monero. In *CEuropean Symposium on Research in Computer Security*, pages 456–474, 2017.

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

[69] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adèle Morisset, and Sébastien Ermeneux. Laser-induced fault injection on smartphone bypassing the secure boot-extended version. *IEEE Transactions on Computers*, 69(10):1449–1459, 2018.

[70] Huiyun Li, Guanghua Du, Cuiping Shao, Liang Dai, Guoqing Xu, and Jinlong Guo. Heavy-ion microbeam fault injection into sram-based fpga implementations of cryptographic circuits. *IEEE Transactions on Nuclear Science*, pages 1341–1348, 2015.

[71] Stéphanie Anceau, Pierre Bleuet, Jessy Clédière, Laurent Maingault, Jean-luc Rainard, and Rémi Tucoulou. Nanofocused x-ray beam to reprogram secure circuits. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 175–188, 2017.

[72] Oscar M Guillen, Michael Gruber, and Fabrizio De Santis. Low-cost setup for localized semi-invasive optical fault injection attacks: How low can we go? In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 207–222, 2017.

[73] Bodo Selmke, Florian Hauschild, and Johannes Obermaier. Peak clock: Fault injection into pll-based systems via clock manipulation. In *ACM Workshop on Attacks and Solutions in Hardware Security Workshop*, pages 85–94, 2019.

[74] Niek Timmers and Cristofaro Mune. Escalating privileges in linux using voltage fault injection. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 1–8, 2017.

[75] Karim M Abdellatif and Olivier Hériveaux. Silicontoaster: A cheap and programmable em injector for extracting secrets. In *Workshop on Fault Detection and Tolerance in Cryptography*, pages 35–40, 2020.

[76] Md Mahbub Alam, Shahin Tajik, Fatemeh Ganji, Mark Tehranipoor, and Domenic Forte. Ram-jam: Remote temperature and voltage fault attack on fpgas using memory collisions. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 48–55, 2019.

[77] Paul Grandamme, Pierre-Antoine Tissot, Lilian Bossuet, Jean-Max Dutertre, Brice Colombier, and Vincent Grosso. Switching off your device does not protect against fault attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024.

[78] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88, 2013.

A Supplemental Material

In this appendix, we demonstrate the vulnerability of fault injection related to ML-DSA, Elgamal, RSA and BLS.

A.1 Analysing ML-DSA

ML-DSA is a lattice-based post-quantum signature formally standardized by NIST implemented in `liboqs-0.10.0`. The underlying hard problem is the RLWE over modulus lattice. We first give a simplified description of the scheme to demonstrate our attack on ML-DSA. Since ML-DSA is rather complicated, we describe it in an algorithm and simplify some unnecessary steps. Please refer to [29] for a detailed description.

Gen(1^λ)**:** (described in algorithm 4) generates pk, sk related to the RLWE problem through a random seed ρ. The number theoretical transform (NTT) is used for acceleration during the generation and has homomorphic properties such that $NTT(ab) = NTT(a) \cdot NTT(b)$ and $NTT^{-1}(NTT(a)) = a$.

Algorithm 4: *Gen* algorithm of ML-DSA (simplified)

Output: Keypair pk, sk
$\zeta \leftarrow \{0,1\}^n, (\rho, \rho', K) \leftarrow H(\zeta)$
`// Generate the public matrix` $\hat{A}$ `in NTT form.`
$\hat{A} \in R_q^{k \times l} \leftarrow ExpandA(\rho)$
$(s_1, s_2) \in S_\eta^l \times S_\eta^k \leftarrow ExpandS(\rho')$
$t \leftarrow NTT^{-1}(\hat{A} \cdot NTT(s_1)) + s_2$
$pk \leftarrow (\hat{A}, t), sk \leftarrow (s_1, s_2, \zeta)$

Sign(pk, sk, m)**:** (depicted in algorithm 5) takes pk, sk and message m as inputs, and generates a signature (c, z). The parameters in Sign are considered targets for fault injection.

Algorithm 5: *Sign* algorithm of ML-DSA (simplified)

Input: Message m, private key $sk = (\rho, tr, s_1, s_2, t, K)$.
Output: Signature $\sigma = (z, c, h)$
`// Compute the matrix` $\hat{A}$ `in NTT form`
$\hat{A} \in R_q^{k \times l} \leftarrow ExpandA(\rho)$
$\hat{s_1}, \hat{s_2}, \hat{t} \leftarrow NTT(s_1), NTT(s_2), NTT(t)$
$\mu \leftarrow H(m, pk)$, $rnd \leftarrow \{0,1\}^{256}$
$\rho' \leftarrow H(K, u, rnd)$
$\kappa \leftarrow 0, (z, h) \leftarrow \perp$
do
  `// Generate y from` ρ', κ
  $y \leftarrow ExpandMask(\rho', \kappa)$
  $w \leftarrow NTT^{-1}(\hat{A} \cdot NTT(y))$
  $c \leftarrow H(\mu, w)$
  $\hat{c} \leftarrow NTT(c)$
  $\langle cs_1 \rangle \leftarrow NTT^{-1}(\hat{c} \cdot \hat{s_1})$, $\langle cs_2 \rangle \leftarrow NTT^{-1}(\hat{c} \cdot \hat{s_2})$
  $z \leftarrow y + \langle cs_1 \rangle$
  $r_0 \leftarrow LowBits(w - \langle cs_2 \rangle)$
  if $||z||_\infty \geq \gamma_1 - \beta$ *or* $||r_0||_\infty \geq \gamma_2 - \beta$ **then**
    $(z, h) \leftarrow \perp$
  else
    `// Generate` h `to compute` w
    $h \leftarrow MakeHint(-ct_0, w - cs_2 + ct_0)$
  end
while $(z, h) = \perp$

Vrfy(pk, m, σ)**:** (described in algorithm 6) takes a signature σ, message m and pk as input, output put 1 if the hash check pass and 0 if not.

We also map the parameters used in ML-DSA into G-sign, c, m, A, t, μ, w are public parameters and s, y are secret. We

Algorithm 6: $Vrfy$ algorithm of ML-DSA (simplified)

1 **Input:** pk, m, σ
2 **Output:** a decision bit
3 $\mu \leftarrow H(m, pk)$
4 $w' \leftarrow NTT^{-1}(\hat{A} \cdot NTT(z) - NTT(c) \cdot NTT(t))$
5 $w \leftarrow UseHint(h, w')$
6 **accept** iff $c = H(\mu, w)$

simulate fault injection to these parameters and find that c, μ, m, A is vulnerable to DFA and s is vulnerable to SCA. We show how to recover secrets from faulty signatures. The derivation is as follows.

Types	pp				sp
ML-DSA	c	m	A, t	μ, w	s_1, s_2, y, ζ
G-sign	h	m	pk	r	sk

Table 5: Classification of potentially vulnerable parameters in ML-DSA.

Faulting public parameters with DFA: Parse pp as A, t, μ, c, we find that when a single fault occurs to c, the output signature is converted to $z' = y + c's_1$, compared to a valid one $z = y + cs_1$, we can compute secret key s_1 as $s_1 = g(pp) \cdot (z' - z)$ and $g(pp) = (c' - c)^{-1}$.As there is a high probability that $(c' - c)$ is invertible, we can easily compute s_1 with two signature queries. We can also conclude that parameters that serve as input when computing c can also be vulnerable, such as m, μ, w.

Faulting secret parameters with SCA: Parse sp as s_1, s_2, y, ζ, and we manage to perform a SCA when faulting s_1 (other secrets can be computed by s_1 in lattice), If one bit flip occurs in s_1 before the signature generation, the faulty signature becomes $z' = y + cs'_1$. The difference between the faulty signature and its valid one is $\Delta z = c(s_1 - s'_1) = c\Delta s_1$. Here $g(pp, \Delta s)$ can be denoted as $c\Delta s$. Note that s_1 is defined over S_η^l and its elements are all polynomials. Without loss of generality, let the one-bit fault occur in the j-th coefficient in the i-th element in s_1, denoted as a_{ij}. The fault component is $\sum_{k=0}^{n-1} c_k x^k \cdot \Delta a_{ij} x^j$. Since c can be computed without a secret key, the attacker can eliminate the fault term to yield a valid signature, resulting in leakage of the secret key.

A.2 Analysing Elgamal

In this analysis, we describe Elgamal [31], which is implemented in the `Cryptopp-8.9` library, based on which we show how to leak the secret key via Rowhammer.

Gen(1^λ): Function Gen generates a public key pk and a secret key sk. It first chooses a key length N, and then a N-bit prime p. Then Choose a generator $g < p$ of the multiplicative group of integers modulo p, $\mathbb{Z}_p^*$. Using (g, p), it chooses x randomly from $\{1 \dots p-2\}$ and set $y = g^x \ mod \ p$. It sets pkas,

$$pk = (g, p, y) \tag{7}$$

Regarding sk, it is defined as:

$$sk = x \tag{8}$$

pk and sk satisfy the following equations:

$$y = g^x \ mod \ p \tag{9}$$

Sign(pk, sk, m): Function Sign takes pk, sk and m as inputs, utilizes H as hash function and k as a randomly chosen number, generates a signature (r, σ) as follows:

$$r = g^k \tag{10}$$

$$\sigma = (H(m) - xr)k^{-1} \tag{11}$$

Vrfy (pk, σ, m): Function Vrfy takes a signature σ and m, pk as inputs, and generates 1 (i.e., verification succeeds) if the following equation holds:

$$g^{H(m)} = y^r r^\sigma \tag{12}$$

We first parse the parameters to G-sign in Table 6. When simulating fault injection on pp, we have not found a successful DFA leakage because the signature and hash value in Equation 11 does not satisfy the linearity property in line 8 of algorithm 1. In SCA analysis, we successfully leak secret keys.

Scheme	pp			sp	
Elgamal	$H(m)$	m	g, p, y	r, k	x
G-sign	h	m	pk	r	sk

Table 6: Classification of potentially vulnerable parameters in Elgamal.

Faulting secret parameters x: When a single bit flip occurs to x before the $Sign$ (pk, sk, m) function is invoked, the generated signature will become as follows:

$$\sigma = (H(m) - x'r)k^{-1} \tag{13}$$

Here, we denote x' as $x + \Delta x$ where Δx represents the injected fault. To make verification hold, Δx must satisfy the following equation (a simplified form of expression of SCA in line 15 in algorithm 2):

$$g^{H(m)} = y^r r^\sigma \cdot g^{\Delta x} \tag{14}$$

When Equation 14 holds, we are able to find out the index of the bit flipped in x and thus recover its original bit.

Faulting secret parameters k: When a single bit flip occurs to k before the $Sign$ (pk, sk, m) function is invoked, the generated signature will become as follows:

$$\sigma = (H(m) - xr)k'^{-1} \tag{15}$$

In a similar way, we denote $k' = k + \Delta k$. We are able to recover Δk as the original bit of k if the following equation holds:

$$g^{H(m)} y^{-r} g^{\sigma^{-1}} = g^{(H(m)-xr)\cdot(H(m)-xr)^{-1}} \cdot g^{k+\Delta k} = r \cdot g^{\Delta k} \tag{16}$$

A.3 Analysing RSA

We briefly introduce RSA implemented in `wolfssl-5.6.6` and `openssl` and demonstrate how we launch a fault injection attack.

Gen(1^λ): Function Gen generates a public key pk and a secret key sk. a public key consists of integers (N,e), where $N = pq$ is a public modulus obtained from two large primes p and q, and e represents the public exponent used in signature verification. The private key d is the secret signing exponent obtained as $d = e - 1 \ mod \ \phi(N)$ where $\phi(N) = (p-1)(q-1)$. Set $pk = (N,e), sk = d$

Sign(pk, sk, m): Function Sign takes pk, sk and m as inputs, output a signature σ as:

$$\sigma = m^d \tag{17}$$

Vrfy (pk, σ, m): Function Vrfy takes a signature σ and m, pk as inputs, and generates 1 (i.e., verification succeeds) if the following equation holds:

$$m = \sigma^e \tag{18}$$

Scheme	pp			sp	
RSA	/	m	N,e	/	d
G-sign	h	m	pk	r	sk

Table 7: Classification of potentially vulnerable parameters in RSA.

We show the mapping table in Table 7. When faulting public parameters, we have not found any leakage to do `DFA` as RSA does not satisfy the linearity property in line 8 of algorithm 1. When faulting sk, we find exploitable leakage using `SCA`.

Faulting secret parameters d: When a single bit flip occurs to d before the *Sign* (pk, sk, m) function is invoked, the generated signature will become as follows:

$$\sigma = m^{d'} \tag{19}$$

We can correct the faulty signature to make verification holds by adding a Δd term as $m = \sigma^e \cdot m^{-e\Delta d}$. Then, use Δd as a leakage bit of secret key d.

A.4 Analysing BLS

We briefly introduce BLS implemented in `bls-signature-2.0.3` and demonstrate how we launch a fault injection attack.

Gen(1^λ): Function Gen generates a public key pk and a secret key sk. The public key consists of elements Q, g from group $\mathbb{G}$, a public hash function H and a bilinear map e. the secret key is a random integer $x, x \in [0, ..., |\mathbb{G}| - 1]$. Set $pk = (Q, g, \mathbb{G}, H, e), sk = x$, pk and sk satisfies the following equations:

$$Q = g^x \tag{20}$$

Sign(pk, sk, m): Function Sign takes pk, sk and m as inputs, compute $h = H(m)$ and output a signature σ as:

$$\sigma = h^x \tag{21}$$

Vrfy (pk, σ, m): Function Vrfy takes a signature σ and m, pk as inputs, compute $h = H(m)$ and generates 1 (i.e., verification succeeds) if the following equation holds:

$$e(\sigma, g) = e(h, Q) \tag{22}$$

Scheme	pp			sp	
BLS	h	m	Q, g	/	x
G-sign	h	m	pk	r	sk

Table 8: Classification of potentially vulnerable parameters in BLS.

We show the mapping table in Table 8. When faulting public parameters, we have not found any leakage to do `DFA` as BLS does not satisfy the linearity property in line 7 of algorithm 1. When faulting sk, we find exploitable leakage using `SCA`.

Faulting secret parameters x: When a single bit flip occurs to x before the *Sign* (pk, sk, m) function is invoked, the generated signature will become as follows:

$$\sigma = h^{x'} \tag{23}$$

We can correct the faulty signature to make verification holds by adding a Δx term as $e(\sigma', g) = e(\sigma, g) \cdot e(h^{\Delta x}, g)$. Then, use Δx as a leakage bit of secret key x.