ZKFault: Fault attack analysis on zero-knowledge based post-quantum digital signature schemes

Puja Mondal[1],
Supriya Adhikary[1], Suparna Kundu[2], and Angshuman Karmakar[1,2]

[1] Department of Computer Science and Engineering, IIT Kanpur, India
{pujamondal,adhikarys,angshuman}@cse.iitk.ac.in
[2] COSIC, KU Leuven, Kasteelpark Arenberg 10, Bus 2452, B-3001 Leuven-Heverlee, Belgium
{suparna.kundu}@esat.kuleuven.be

Abstract. Computationally hard problems based on coding theory, such as the syndrome decoding problem, have been used for constructing secure cryptographic schemes for a long time. Schemes based on these problems are also assumed to be secure against quantum computers. However, these schemes are often considered impractical for real-world deployment due to large key sizes and inefficient computation time. In the recent call for standardization of additional post-quantum digital signatures by the National Institute of Standards and Technology, several code-based candidates have been proposed, including LESS, CROSS, and MEDS. These schemes are designed on the relatively new zero-knowledge framework. Although several works analyze the hardness of these schemes, there is hardly any work that examines the security of these schemes in the presence of physical attacks.
In this work, we analyze these signature schemes from the perspective of fault attacks. All these schemes use a similar tree-based construction to compress the signature size. We attack this component of these schemes. Therefore, our attack is applicable to all of these schemes. In this work, we first analyze the LESS signature scheme and devise our attack. Furthermore, we showed how this attack can be extended to the CROSS signature scheme. Our attacks are built on very simple fault assumptions. Our results show that we can recover the entire secret key of LESS and CROSS using as little as a single fault. Finally, we propose various countermeasures to prevent these kinds of attacks and discuss their efficiency and shortcomings.

Keywords: Post-quantum cryptography · Post-quantum signature· Code-based cryptography· Fault attacks· LESS· CROSS

1 Introduction

Digital signature schemes are one of the most used and fundamental cryptographic primitives. The security of our current prevalent digital signature schemes based on integer factorization [33] or elliptic curve discrete logarithms [23] can be compromised using a large quantum computer [37,30]. Therefore, we need quantum computer-resistant digital signature algorithms. In 2022, the National Institute of Standards and Technology (NIST) selects three post-quantum digital signature schemes [2]

CRYSTALS-DILITHIUM [15], FALCON, and SPHINCS+ [5] for standardization. Among them, FALCON and DILITHIUM are based on lattices, and SPHINCS+ is a hash-based signature scheme.

A majority of these signature schemes are lattice-based. Therefore, a breakthrough result in the field of cryptanalysis of lattice-based cryptography could create a major dilemma in the transition from classical to post-quantum cryptography. Such incidents are not very rare. Some recent examples are Castryck *et al.*'s [10] attack on the post-quantum key-exchange mechanism based on supersingular isogeny Diffe-Hellman [19] problem or Beullens's attack [6] on post-quantum digital signature scheme Rainbow [14]. Both of these schemes were finalists of the NIST's post-quantum standardization procedure. Therefore, diversification in the underlying hard problems ensures that if one of the cryptographic schemes is compromised, others may remain secure. Another problem of the currently standardized signature schemes is their very large signature sizes compared to classical signatures. This renders them almost impractical for real-world use cases like SSL/TLS certificate chains. Recognizing the critical importance of diversification and the practical use of digital signatures, NIST has recently issued an additional call [25] for post-quantum secure digital signatures. In this call, NIST emphasizes the importance of small signature and fast verification to enhance practicality.

Linear Equivalence Signature Scheme (LESS) [7,35] is a submitted digital signature scheme aimed at increasing diversification and smaller signature and public key sizes. There are other code-based submissions like WAVE [36], enhanced pqsigRM [11], and CROSS [34]. These schemes are based on the Syndrome Decoding Problem (SDP) for linear codes. The hardness of SDP relies on different variants of Information Set Decoding (ISD) algorithms. On the other hand, LESS has avoided the SDP, and it is the first cryptographic scheme based on the Code Equivalence Problem (CEP). The CEP asks to determine if two linear codes are equivalent to each other. In the Hamming metric, the notion of equivalence is linked to the existence of a monomial transformation, often termed the Linear Equivalence Problem (LEP).

Due to the choice of this hard problem, the designers could choose parameters that lead to smaller key sizes without compromising security. The designers have also proposed different compression techniques to reduce the key sizes. LESS offers a balanced trade-off between the combined public key and signature size and the efficiency of signing and verification routines. Table 4 in Appendix A compares the key sizes and efficiency of LESS and other code-based digital signature schemes.

We want to note that LESS first introduced the novel problem CEP or LEP for cryptographic constructions. It uses a 3-round interactive sigma protocol between a prover and a verifier. Other signature schemes like MEDS and CROSS are also based on similar zero-knowledge identification schemes. Multiple rounds of the identification scheme are used here, which is converted into a signature scheme using the Fiat-Shamir transformation. However, using multiple rounds increases the signature size. Here, we have noticed that all three signature schemes, LESS, CROSS and MEDS [12], use the same compression technique that helped the designers ease the long-enduring bottleneck of large signature sizes in code-based cryptography. However, the implementation of this common compression technique has potential vulnerabilities

against fault attacks that we identified in this work. Our primary motivation in this work is to uncover potential vulnerabilities against a wide spectrum of fault attacks and propose suitable countermeasures for the schemes LESS and CROSS that use the protocols having the same compression technique. We are confident that this work will help to improve the LESS and CROSS signature schemes and be useful in the evaluation of NIST's standardization procedure. Further, we strongly believe that this will also be beneficial to other cryptographic signature schemes, such as MEDS, as it uses a similar technique. Below, we briefly summarize our contributions.

Fault analysis of LESS digital signature: We have explored several fault attack surfaces of the LESS signature scheme that could be exploited by an adversary. We found different attack surfaces in the signing algorithm of LESS, and attack strategies that can be utilized on those attack surfaces. We observed that the designers of LESS proposed a technique to compress the signature size. They used a binary tree called *Reference Tree* to fulfil this purpose. We show that the modification of the values in the tree during the signing algorithm leaks information about the secret key as part of the output signature. We further use this information to recover the full secret key.

Versatility of our fault attack: Our attack assumes a single fault injection model. We want to note that our focus was to develop the theoretical framework to recover the secret after the fault injection. In this regard, our attack can be realized using many different faults. Therefore, it is very versatile *i.e.* not skewed in favour of the attacker. In particular, we discuss the applicability of our attack using different types of faults, such as instruction skip, stuck-at-zero, and bit-flip. These types of faults can be realized using different mechanisms such as voltage glitch [13], Rowhammer [24,31], clock glitch [9,28], laser fault injection [8], electromagnetic fault injection [17,20] etc.

Strong mathematical analysis: We give detailed mathematical analysis to recover the secret key after the fault injections. We consider an arbitrary location for the fault injection, which is known to the attacker. Then, discuss the methods to recover the secret key in different scenarios. To further improve the effectiveness and practicality of our attack, we also provide a very effective method to remove noise from the experiments *i.e.* differentiating between effective and ineffective faults. This is a non-trivial problem in any fault injection attack. We mathematically derived the expected amount of secret information that can be recovered from a single effective fault.

Application to other zero-knowledge based signature schemes: Other code-based signature schemes in the NIST additional call for signatures such as CROSS [34] and MEDS [12], use a similar zero-knowledge framework as LESS. In these frameworks, the challenger and prover must communicate a series of challenges and responses for the soundness of schemes. This increases the signature size of the digital signature schemes designed using this framework. All these three signature schemes use a binary tree-based compression technique to reduce the signature size. As our attack targets this method, our attack strategy can also be extended to these schemes. We have explained this strategy for the CROSS signature scheme in this work.

Attack simulation: We have an end-to-end fault attack simulation on the reference implementation of LESS and CROSS signature schemes. For LESS, we have simulated the attack in a way so that it can count the number of secret matrix recovered with one faulted signature, the number of faulted signatures required to recover the whole

secret. Also, our simulation induces fault with varying successful fault probability. In both schemes, we modify a particular node of the binary tree structure and then recover the secret from the faulted signatures. We have shown that if we inject fault in a specific location, then the entire secret can be recovered from a single effective fault signature for all the parameter sets of LESS except the parameter LESS-1s. For the CROSS signature scheme, only one effective faulted signature is enough to recover the complete secret for all parameters.

Countermeasures: Finally, we discuss different countermeasures that can prevent such attacks. We show that these countermeasures are effective against the single-fault attack models. Our first countermeasure removes the primary source of vulnerability *i.e.* the generation of the *Reference Tree*. This rather simple method increases the signature size. The second countermeasure modifies the *Reference Tree* generation procedure such that the attack surfaces are eliminated. This method ensures that the signature sizes stay the same as the original LESS proposal [35]. Lastly, we implemented the second countermeasure for LESS and compared its performance with the original LESS implementation. The performance cost of our second countermeasure is the same as the cost of the original LESS implementation.

2 Preliminaries

$\mathbb{Z}_q$ denotes the ring of integers modulo q. Additionally, $\mathbb{F}_q$ and $\mathbb{F}_q^*$ have been used to signify the field with q elements and the multiplicative group of this field $\mathbb{F}_q$, respectively. The sets $\mathbb{F}_q^k$ and $\mathbb{F}_q^{k\times n}$ represent the collection of all vectors of size k and all matrices of dimension $k\times n$ over the field $\mathbb{F}_q$, respectively. We use calligraphic uppercase ($\mathcal{C}$) to denote a linear code.

The lowercase letters (a) and uppercase letters (A) denote the scalars and the ordered set of scalars, respectively. A^c represents the complement of the set A. We use bold lowercase ($\boldsymbol{a}$) to denote vectors in any domain, and the i-th entry of the vector $\boldsymbol{a}$ is denoted by $\boldsymbol{a}[i]$. We denote the i-th standard basis as $\boldsymbol{e}_i$. The transpose of a vector $\boldsymbol{a}$ is denoted by $\boldsymbol{a}^T$.

The bold uppercase letters ($\boldsymbol{A}$) represent matrices. Let $\boldsymbol{A}$ be a matrix, then $\boldsymbol{A}[i, j]$ represents the i, j-th entry of the matrix $\boldsymbol{A}$. Also, $\boldsymbol{A}[*, j]$ and $\boldsymbol{A}[i, *]$ represent the j-th column and i-th row of the matrix $\boldsymbol{A}$ respectively. Let $J\subset\mathbb{Z}_n$ be an ordered set of column indices of the matrix $\boldsymbol{A}$, then the notation $\boldsymbol{A}[*, J]$ represents the submatrix of $\boldsymbol{A}$ formed by selecting columns with indices specified in the set J. Similarly, if J is an ordered set of row indices of matrix $\boldsymbol{A}$, then the notation $\boldsymbol{A}[J, *]$ represents the submatrix of $\boldsymbol{A}$ formed by selecting rows with indices specified in the set J. The transpose of a matrix $\boldsymbol{A}$ is denoted by $\boldsymbol{A}^T$. The inner product of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ of same size is denoted by $\langle \boldsymbol{a}, \boldsymbol{b}\rangle$ and is defined by $\sum_i \boldsymbol{a}[i]\boldsymbol{b}[i]$. The set of all invertible matrices of order k over $\mathbb{F}_q$ is denoted by $GL_k(q)$.

2.1 Definitions

Definition 1 (Monomial matrix). *An $n\times n$ matrix $\boldsymbol{A}$ is called a monomial matrix if we can write $\boldsymbol{A} := (\boldsymbol{u}[0]\boldsymbol{e}_{\pi(0)} \mid \boldsymbol{u}[1]\boldsymbol{e}_{\pi(1)} \mid \cdots \mid \boldsymbol{u}[n-1]\boldsymbol{e}_{\pi(n-1)})$. Here, $\boldsymbol{u} \in \mathbb{F}_q^n$,*

$\pi : \mathbb{Z}_n \to \mathbb{Z}_n$ is a permutation and $\boldsymbol{u}[j]\boldsymbol{e}_{\pi(j)}$ is j-th column of $\boldsymbol{A}$. We represent the monomial matrix $\boldsymbol{A}$ with the pair $(\pi, \boldsymbol{u})$.

Definition 2 (Partial monomial matrix). *An $n \times k$ matrix $\boldsymbol{B}$ is called a partial monomial matrix if we can write the matrix $\boldsymbol{B} := (\boldsymbol{v}[0]\boldsymbol{e}_{\pi_*(0)} \mid \boldsymbol{v}[1]\boldsymbol{e}_{\pi_*(1)} \mid \cdots \mid \boldsymbol{v}[k-1]\boldsymbol{e}_{\pi_*(k-1)})$. Here, $n > k$, $\boldsymbol{v} \in \mathbb{F}_q^k$ and $\pi_* : \mathbb{Z}_k \to \mathbb{Z}_n$ is an injective mapping. We represent the partial monomial matrix $\boldsymbol{B}$ with the pair $(\pi_*, \boldsymbol{v})$.*

We denote the set of all invertible monomial matrices of order n and the set of all partial monomial matrices of order $n \times k$ over $\mathbb{F}_q$ by $M_n(q)$ and $M'_{n,k}(q)$ respectively.

Definition 3 (Reduced Row-Echelon form). *A matrix $\boldsymbol{A}$ of order $m \times n$ is said to be in Reduced Row-Echelon form (RREF) if the following conditions hold*

i. For each $0 \le i \le m-1$, $0 \le j \le n-1$, if the i-th row contains the first non-zero element at j-th position, then the first non-zero element of $(i+1)$-th row should be after the j-th position.
ii. The first non-zero element of any non-zero row is 1.
iii. The leading element is the only non-zero element of that column.

We can transfer any matrix $\boldsymbol{A}$ to its `RREF` form by applying some elementary row operations [22] on the matrix $\boldsymbol{A}$, and we denote this transformation by $\texttt{RREF}(\boldsymbol{A})$. Also, note that a matrix has a unique `RREF`. The first non-zero elements of $\texttt{RREF}(\boldsymbol{A})$ in each row are called *pivots* and the columns that contain pivot are called *pivot column* of the matrix $\texttt{RREF}(\boldsymbol{A})$. The remaining columns are called *non-pivot columns*.

Definition 4 (Lexicographically sorted order). *Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be two vectors of the same size over the field $\mathbb{F}_q$. We call the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ are in lexicographical order if $\boldsymbol{a}[i] < \boldsymbol{b}[i]$ holds, where i is the first position where two vectors differ. We denote it as $\boldsymbol{a} < \boldsymbol{b}$. Let there be r vectors $\boldsymbol{v}_0, \boldsymbol{v}_1, \cdots, \boldsymbol{v}_{r-1}$ over the field $\mathbb{F}_q$. We call these vectors in lexicographically sorted order if, for any $0 \le i, j < r$, $\boldsymbol{v}_i < \boldsymbol{v}_j$ holds whenever $i < j$.*

A matrix G is lexicographically sorted if its columns are in ascending lexicographical order. In this paper, the function `LexMinCol` makes each column of input matrix $\boldsymbol{G}$ to lexicography sorted order by multiplying the inverse of the first non-zero element of that column and `LexSort` function is used to sort the columns of $\boldsymbol{G}$ in lexicographically sorted order.

Definition 5 (Linear code). *An $[n, k]$-linear code $\mathcal{C}$ of length n and dimension k is a linear subspace of the vector space $\mathbb{F}_q^n$. It can be represented by a matrix $\boldsymbol{G} \in \mathbb{F}_q^{k \times n}$, which is called a generator matrix. For any $\boldsymbol{u} \in \mathbb{F}_q^k$, the generator matrix $\boldsymbol{G}$ maps it to a code-word $\boldsymbol{uG} \in \mathbb{F}_q^n$.*

Definition 6 (Linear code equivalence). *Let $\mathcal{C}$ and $\mathcal{C}'$ be two linear codes of length n and dimension k with generator matrices $\boldsymbol{G}$ and $\boldsymbol{G}'$ respectively. We call the codes $\mathcal{C}$ and $\mathcal{C}'$ linearly equivalent, if there exist matrices $\boldsymbol{Q} \in M_n(q)$, $\boldsymbol{S} \in GL_k(q)$ such that $\boldsymbol{G}' = \boldsymbol{SGQ}$.*

Definition 7 (Information Set (IS) of a Linear Code [27]). *Let $\mathcal{C}$ be a linear code with length n, and $J \subset \mathbb{Z}_n$ be a set with cardinality k. Consider $\boldsymbol{G}$ as the generator matrix of code $\mathcal{C}$. Define J as an information set corresponding to $\boldsymbol{G}$ if inverse of $\boldsymbol{G}[*, J]$ exists i.e., $\boldsymbol{G}[*, J]$ is non-singular.*

2.2 LESS signature scheme

The signature scheme LESS is based on the hardness of the Linear-code Equivalence Problem (LEP). LESS signature [35] uses a 3-round interactive sigma protocol [16] between a prover and a verifier to establish the message's authenticity and the Fiat Shamir transformation [1] to transform this interactive protocol into a signature scheme. In this section, we describe the key generation and the signature algorithm of the digital signature LESS as it is most relevant to our work. Meanwhile, the verification algorithm is described in Appendix B. The description of the LESS signature involves some additional functions that we describe below.

- $\texttt{CSPRNG}(\mathit{seed}, \cdot)$: This is a pseudo-random number generator, which takes a seed as input and outputs a pseudo-random string. The resulting output can be formatted according to preference, either as a string of seed values or a matrix. The uses of the function as $\texttt{CSPRNG}(\mathit{seed}, \mathbb{S}_{\texttt{RREF}})$, $\texttt{CSPRNG}(\mathit{seed}, \mathbb{S}_{t,w})$ and $\texttt{CSPRNG}(\mathit{seed}, M_n(q))$ represents sampling a generator matrix in RREF, sampling the fixed weight digest vector and sampling a monomial matrix, respectively using the provided *seed*.
- $\texttt{SeedTree}(\mathit{seed}, \mathit{salt})$: This function generates a tree of height $\lceil \log\, t \rceil$. It begins with λ bit input *seed* and uses the CSPRNG function to generate 2λ bits. This long string is divided into two parts: the first λ bits are used for the left child and the last λ bits for the right child. The bits corresponding to each child are again fed into the CSPRNG with *salt* to generate the next layer of the nodes in the tree. This process is repeated until the tree with height $\lceil \log\, t \rceil$ is constructed.
- $\texttt{PrepareDigestInput}(\boldsymbol{G}, \boldsymbol{Q'})$: This function takes the matrices $\boldsymbol{G}$ which is in RREF and a monomial matrix $\boldsymbol{Q'}$ as inputs. Then computes $\boldsymbol{G'}$ as $(\boldsymbol{G'}, \mathit{pivot\_column}) = \texttt{RREF}(\boldsymbol{G}\boldsymbol{Q'}^T)$. Let $J = \{\alpha_0, \alpha_1, \cdots, \alpha_{k-1}\}$ be the set of pivot column indices, which is essentially the information set (IS) of $\boldsymbol{G'}$. Then, compute the partial monomial matrix $\overline{\boldsymbol{Q}}'$ and the matrix $\overline{\boldsymbol{V}}'$ as $\overline{\boldsymbol{Q}}' = \boldsymbol{Q'}^T[*,\ J]$ and $\overline{\boldsymbol{V}}' = \texttt{LexSort}(\texttt{LexMinCol}(\boldsymbol{G'}[*,\ J^c]))$. After this computation, this function returns the partial monomial matrix $\overline{\boldsymbol{Q}}'$ and the matrix $\overline{\boldsymbol{V}}'$ as outputs.
- $\texttt{SeedTreePaths}(\boldsymbol{seed}, \boldsymbol{f})$: Given a seed tree $\boldsymbol{seed}$ and a binary string $\boldsymbol{f}$ representing the leaves to be disclosed, this procedure derives which nodes of the seed tree should be disclosed so that the verifier can rebuild all the leaves which have been marked by the binary string. A detailed description of this function is given in Alg. 4.
- CompressRREF and CompressMono: CompressRREF function is used to compress a matrix $\boldsymbol{G}$ in RREF, and similarly CompressMono is used to compress a monomial matrix. Each compression procedure have corresponding expansion procedure that converts the compressed information to its proper matrix form. Therefore, we can assume using or not using these function does not affect the functionality of key generation, signing or verification of LESS.

Algorithm 1 `LESS_KeyGen(`λ`)` [7,4]

Input: None
Output: SK$=(MSEED, gseed)$, PK$=(gseed, \boldsymbol{G}_1, ..., \boldsymbol{G}_{s-1})$

1: $MSEED \xleftarrow{\$} \{0,1\}^{\lambda}$
2: $\boldsymbol{mseed} \leftarrow \texttt{CSPRNG}(MSEED) \in \{0,1\}^{(s-1)\lambda}$
3: $gseed \xleftarrow{\$} \{0,1\}^{\lambda}$
4: $\boldsymbol{G}_0 \leftarrow \texttt{CSPRNG}(gseed, \mathbb{S}_{\texttt{RREF}})$
5: **for** $i=1; i<s; i=i+1$ **do**
6: $\quad \boldsymbol{Q}_i \leftarrow \texttt{CSPRNG}(\boldsymbol{mseed}[i], M_n(q))$
7: $\quad (\boldsymbol{G}_i, pivot\_column) \leftarrow \texttt{RREF}(\boldsymbol{G}_0(\boldsymbol{Q}_i^{-1})^T)$
8: $\quad \text{PK}[i] \leftarrow \texttt{CompressRREF}(\boldsymbol{G}_i, pivot\_column)$
9: Return (SK, PK)

Key Generation of LESS: It is presented in Alg. 1. Given a security parameter λ, the two outputs of this algorithm are the secret key SK and the public key PK. The first component of the secret key is the master key $MSEED \in \{0,1\}^{\lambda}$. Using the `CSPRNG` function, the vector $\boldsymbol{mseed} \in \{0,1\}^{(s-1)\lambda}$ is generated from the $MSEED$, which contains $s-1$ many λ-bit binary strings. Now, the i-th secret monomial matrix $\boldsymbol{Q}_i$ is generated from $\boldsymbol{mseed}[i] \in \{0,1\}^{\lambda}$. Note that these generated $\boldsymbol{Q}_i$'s are all secret monomial matrices. Also, the seed $gseed$ is employed in the generation of the public matrix $\boldsymbol{G}_0$. The remaining part of the public key consists of the matrices $\boldsymbol{G}_i$ for $1 \leq i \leq s-1$, which are generated using the process described in Alg. 1[3].

Signature algorithm of LESS: The signature algorithm shown in Alg. 2 takes a message string m of length len and the secret key SK$=(MSEED, gseed)$ as inputs and returns a corresponding signature τ. The main secret key component of SK is the master seed $MSEED$. All the $s-1$ monomial matrices $\boldsymbol{Q}_j$ are generated from the $MSEED$ and used to produce signatures. That is, instead of having information of $MSEED$, if we have the information of all of $s-1$ monomial matrices $\boldsymbol{Q}_j$, then we can construct the same valid signature. Therefore, these monomial matrices $\boldsymbol{Q}_j$ are considered equivalent to the secret key component $MSEED$. To reduce the signature size, the authors of LESS have incorporated a method involving tree construction. We explain this process briefly here.

First, we outline the procedure for generating a set of t ephemeral monomial matrices represented by $\widetilde{\boldsymbol{Q}}_0, \widetilde{\boldsymbol{Q}}_1, \cdots, \widetilde{\boldsymbol{Q}}_{t-1}$ through the generation of t random ephemeral seeds denoted as $\boldsymbol{ESEED}[i]$ for $0 \leq i < t$. The process involves the following steps:

- Start by sampling a random master seed $EMSEED \xleftarrow{\$} \{0,1\}^{\lambda}$.
- Build a tree of seed nodes using `SeedTree` procedure, with output tree named $\boldsymbol{seed}$. The height and the number of leaf nodes of the output tree are $\lceil \log(t) \rceil$ and $2l = 2^{\lceil \log(t) \rceil}$ respectively where the input seed is the master seed $EMSEED$.

[3] For simplicity and compactness, we follow the implementation of LESS instead of the specification document

Algorithm 2 `LESS_Sign`(m, SK)

Input: Message $m \in \mathbb{Z}_2^{len}$ and secret key SK $=(MSEED, gseed)$.
Output: The signature $\tau = (salt, cmt, \boldsymbol{TreeNode}, \boldsymbol{rsp})$.

1: $\boldsymbol{mseed} \leftarrow \texttt{CSPRNG}(MSEED) \in \{0,1\}^{(s-1)\lambda}$
2: $EMSEED \xleftarrow{\$} \{0,1\}^{\lambda}$, $salt \xleftarrow{\$} \{0,1\}^{\lambda}$
3: $\boldsymbol{seed} \leftarrow \texttt{SeedTree}(EMSEED, salt)$
4: $\boldsymbol{ESEED}$ = Leaf nodes of the $\boldsymbol{seed}$
5: $\mathbf{G}_0 \leftarrow \texttt{CSPRNG}(gseed, \mathbb{S}_{\texttt{RREF}})$
6: **for** $i=0$; $i<t$; $i=i+1$ **do**
7: $\quad \widetilde{\boldsymbol{Q}}_i \leftarrow \texttt{CSPRNG}(\boldsymbol{ESEED}[i], M_n(q))$
8: $\quad (\overline{\boldsymbol{Q}}_i, \overline{\boldsymbol{V}}_i) \leftarrow \texttt{PrepareDigestInput}(\boldsymbol{G}_0, \widetilde{\boldsymbol{Q}}_i)$
9: $cmt \leftarrow \texttt{H}(\overline{\boldsymbol{V}}_0, ..., \overline{\boldsymbol{V}}_{t-1}, m, len, salt)$
10: $\boldsymbol{d} \leftarrow \texttt{CSPRNG}(cmt, \mathbb{S}_{t,w})$
11: **for** $i=0$; $i<t$; $i=i+1$ **do**
12: $\quad$ **if** $\boldsymbol{d}[i]=0$ **then**
13: $\quad\quad \boldsymbol{f}[i]=0$
14: $\quad$ **else**
15: $\quad\quad \boldsymbol{f}[i]=1$
16: $\boldsymbol{TreeNode} \leftarrow \texttt{SeedTreePaths}(\boldsymbol{seed}, \boldsymbol{f})$ ▷ (Alg. 4)
17: $k=0$
18: **for** $i=0$; $i<t$; $i=i+1$ **do**
19: $\quad$ **if** $\boldsymbol{d}[i] \neq 0$ **then**
20: $\quad\quad j = \boldsymbol{d}[i]$
21: $\quad\quad \boldsymbol{Q}_j \leftarrow \texttt{CSPRNG}(\boldsymbol{mseed}[j], M_n(q))$
22: $\quad\quad \boldsymbol{Q}_k^* \leftarrow \boldsymbol{Q}_j^T \overline{\boldsymbol{Q}}_i$
23: $\quad\quad \boldsymbol{rsp}[k] \leftarrow \texttt{CompressMono}(\boldsymbol{Q}_k^*)$
24: $\quad\quad k=k+1$
25: Return $\tau = (salt, cmt, \boldsymbol{TreeNode}, \boldsymbol{rsp})$

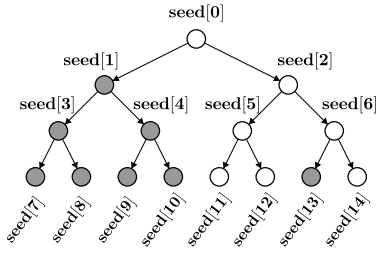

Fig. 1: Example of seed tree

- Select the first t leaf nodes of the $\boldsymbol{seed}$ as the ephemeral seeds $\boldsymbol{ESEED}[i]$, where $\boldsymbol{ESEED}[i] = \boldsymbol{seed}[2l-1+i]$ for $0 \leq i < t$.
- Using `CSPRNG` function $\widetilde{\boldsymbol{Q}}_i$ is prepared for each $\boldsymbol{ESEED}[i]$.

Lines 6-8 of Alg. 2, correspond to generating the partial monomial matrices $\overline{\boldsymbol{Q}}_i$, the matrices $\overline{\boldsymbol{V}}_i$ having the information of the non-pivot part corresponding to the matrix $\boldsymbol{G}_0 \widetilde{\boldsymbol{Q}}_i^T$. Using the information of all $\overline{\boldsymbol{V}}_i$ matrices, message m, message length len and

salt, the digest $\boldsymbol{d} \in \mathbb{Z}_s^t$ is prepared. This digest vector $\boldsymbol{d}$ has fixed weight w, where weight of the vector $\boldsymbol{d}$ is defined as $wt(\boldsymbol{d}) := |\{i\colon \boldsymbol{d}[i] \neq 0\}|$. We will briefly discuss the `SeedTreePaths` procedure in Alg. 4, as our attack is based on exploting this procedure. This `SeedTreePaths` (Alg. 4) helps to reduce the size of the signature. Finally, the signature will return $\boldsymbol{Q}_{\boldsymbol{d}[i]}^T \overline{\boldsymbol{Q}_i}$ whenever $\boldsymbol{d}[i] \neq 0$, and it also reveals the seed nodes so that $\widetilde{\boldsymbol{Q}}_i$ can be generated from the revealed seed nodes for all i such that $\boldsymbol{d}[i] = 0$. Note that whenever we try to return $\widetilde{\boldsymbol{Q}}_i$, it is enough to return $\boldsymbol{ESEED}[i]$'s instead. Also, having the information of any ancestor node of the seed $\boldsymbol{ESEED}[i] (= \boldsymbol{seed}[2l-1+i])$, we can get the information of $\boldsymbol{ESEED}[i]$. This is the idea behind the minimization of the number of seeds that are to be sent. This minimized set is returned as $\boldsymbol{TreeNode}$. Consider the example in Fig. 1, where the leaf nodes are $\boldsymbol{ESEED}[i]$'s and the shaded leaf nodes represent all those positions where $\boldsymbol{d}$ takes the value 0 *i.e.*, these are the $\boldsymbol{ESEED}[i]$'s that are to be revealed. Observe that revealing only $\boldsymbol{TreeNode} = (\boldsymbol{seed}[1], \boldsymbol{seed}[13])$ is enough, as the required $\boldsymbol{ESEED}[i]$'s can be regenerated at the time of verification. Consequently, this minimizes the signature size.

Now, in lines 18-24 of Alg. 2, the $\boldsymbol{rsp}$ is prepared by appending the partial monomial matrices $\boldsymbol{Q}_{\boldsymbol{d}[i]}^T \overline{\boldsymbol{Q}}_i$ for all i such that $\boldsymbol{d}[i]$ is non-zero. Since the length and the weight of the fixed weight digest $\boldsymbol{d}$ are t and w respectively, the vector $\boldsymbol{d}$ has exactly w many non-zero elements and $t-w$ many zero elements. Therefore, the signature will contain the component $\boldsymbol{rsp}$ having exactly w many matrices of the form $\boldsymbol{Q}_{\boldsymbol{d}[i]}^T \overline{\boldsymbol{Q}}_i$. After all of these computations, ($salt$, cmt, $\boldsymbol{TreeNode}$, $\boldsymbol{rsp}$) is generated as the signature.

2.3 Parameter set

There are three security levels of LESS [35] and their corresponding parameter sets, which are shown in Table 1. Here, the code parameters are given by n: the length of the code, k: the dimension of the code, q: prime modulus corresponding to the finite field $\mathbb{F}_q$, $2l$: the number of leaf nodes of the seed tree, where $2l = 2^{\lceil \log t \rceil}$, t: the length of the digest $\boldsymbol{d}$, w: the fixed weight of the digest $\boldsymbol{d}$ and s: $s-1$ is the number of secret monomial matrices. According to the LESS documentation [35], multiple parameter sets are defined for each security level of LESS, and the optimization criteria for each of these parameter sets are different. The "b" version (e.g., LESS-1b) refers to the parameter set with balanced public key and signature size, the "s" (e.g., LESS-1s) version refers to the parameter set with smaller signature size, and the "i" (only LESS-1i) version refers to the parameter set with intermediate public key and signature size.

3 Our Work: Fault analysis of LESS

One of the strongest physical attacks on the digital signature schemes is to recover the secret or signing key, as the adversary can compute any valid message and signature pair using the recovered signing key. In general, only the key generation and the signing algorithm involve the secret key. However, only the signing algorithm uses the long-term secret key (the same secret key is used multiple times), making it most suitable for performing a physical attack [9,29,38,18]. In this work, our objective is

Security level	Parameter set	Parameters							Public key (PK) (KiB)	Signature (τ) (KiB)
		n	k	q	l	t	w	s		
1	LESS-1b	252	126	127	128	247	30	2	13.7	8.1
	LESS-1i					244	20	4	41.1	6.1
	LESS-1s					198	17	8	95.9	5.2
3	LESS-3b	400	200	127	512	759	33	2	34.5	18.4
	LESS-3s					895	26	3	68.9	14.1
5	LESS-5b	548	274	127	1024	1352	40	2	64.6	32.5
	LESS-5s				512	907	37	3	129.0	26.1

Table 1: Parameter set of LESS [35] for different security levels

to mount a fault attack on the zero-knowledge based digital signature schemes. In this attack model, the adversary would query the faulted signature oracle (which outputs a signature with some injected faults) multiple times. In this section, we will progressively describe our fault attack strategy to recover the secret monomial matrices for the LESS signature scheme. Later in Section 4, we show that the same attack strategy can be employed in other zero-knowledge based signature schemes, such as CROSS, to recover the signing key.

3.1 An observation on LESS

LESS signature algorithm presented in Alg. 2 returns either the information of the monomial matrix $\widetilde{\boldsymbol{Q}}_j$ or the multiplication $\boldsymbol{Q}_{\boldsymbol{d}[j]}^T\overline{\boldsymbol{Q}}_j$ for any $j \in \mathbb{Z}_t$. Here, $\overline{\boldsymbol{Q}}_j$ is a *partial monomial* matrix that is generated from the matrix $\widetilde{\boldsymbol{Q}}_j$ by using the `PrepareDigestInput` function. If we manage to get a pair $(\widetilde{\boldsymbol{Q}}_j, \boldsymbol{Q}_{\boldsymbol{d}[j]}^T\overline{\boldsymbol{Q}}_j)$ for some $\boldsymbol{d}[j] \neq 0$, then we can construct the pair $(\overline{\boldsymbol{Q}}_j, \boldsymbol{Q}_{\boldsymbol{d}[j]}^T\overline{\boldsymbol{Q}}_j)$. This pair $(\overline{\boldsymbol{Q}}_j, \boldsymbol{Q}_{\boldsymbol{d}[j]}^T\overline{\boldsymbol{Q}}_j)$ leaks some information of matrix $\boldsymbol{Q}_{\boldsymbol{d}[j]}^T$ that is directly follows from the following lemma.

Lemma 1. *Let $\boldsymbol{A} = (\pi, \boldsymbol{u}) \in M_n(q)$ be a monomial matrix and $\boldsymbol{B} = (\pi', \boldsymbol{u'}) \in M'_{n,k}(q)$ be a partial monomial matrix. Let $\boldsymbol{C} = (\pi'', \boldsymbol{u''}) \in M'_{n,k}(q)$ be the partial monomial matrix defined by $\boldsymbol{C} = \boldsymbol{A}^T\boldsymbol{B}$. Given the matrices $\boldsymbol{B}$ and $\boldsymbol{C}$, we can compute exactly k many columns of the monomial matrix $\boldsymbol{A}^T$. More specifically, for all $0 \leq j < k$, we can compute $\pi^{-1}(\pi'(j))$ and $\boldsymbol{u}[\pi^{-1}(\pi'(j))]$.*

Proof. For the monomial matrix $\boldsymbol{A}$ represented by $(\pi, \boldsymbol{u})$, the transpose of $\boldsymbol{A}$ is the following matrix

$$\boldsymbol{A}^T = [\boldsymbol{u}[\pi^{-1}(0)]\boldsymbol{e}_{\pi^{-1}(0)} \mid \boldsymbol{u}[\pi^{-1}(1)]\boldsymbol{e}_{\pi^{-1}(1)} \mid \cdots \mid \boldsymbol{u}[\pi^{-1}(n-1)]\boldsymbol{e}_{\pi^{-1}(n-1)}]$$

The multiplication of the monomial matrix $\boldsymbol{A}^T$ with the partial monomial matrix $\boldsymbol{B}$ is given by

$$\boldsymbol{A}^T\boldsymbol{B} = [\boldsymbol{u}[\pi^{-1}(\pi'(0))]\boldsymbol{u'}[0]\boldsymbol{e}_{\pi^{-1}(\pi'(0))} \mid \cdots \mid \boldsymbol{u}[\pi^{-1}(\pi'(k-1))]\boldsymbol{u'}[k-1]\boldsymbol{e}_{\pi^{-1}(\pi'(k-1))}]$$

Since $\boldsymbol{C}=\boldsymbol{A}^T\boldsymbol{B}$, so for all $0\le j<k$ we have $\boldsymbol{C}[*,\, j]=(\boldsymbol{A}^T\boldsymbol{B})[*,\, j]$, which implies $\boldsymbol{u''}[j]\boldsymbol{e}_{\pi''(j)}=\boldsymbol{u}[\pi^{-1}(\pi'_*(j)]\boldsymbol{u'}[j]\boldsymbol{e}_{\pi^{-1}(\pi'_*(j))}$. This gives us the following

$$\begin{aligned}\boldsymbol{u''}[j]&=\boldsymbol{u}[\pi^{-1}(\pi'(j))]\boldsymbol{u'}[j]\\ \pi''(j)&=\pi^{-1}(\pi'(j))\end{aligned}$$

Since $\boldsymbol{B}$ and $\boldsymbol{C}$ are known, we have the information of each $\pi'(j)$, $\pi''(j)$, $\boldsymbol{u'}[j]$ and $\boldsymbol{u''}[j]$ where $0\le j<k$. Therefore for all $0\le j<k$ we have,

$$\begin{aligned}\boldsymbol{u}[\pi^{-1}(\pi'(j))]&=\boldsymbol{u''}[j](\boldsymbol{u'}[j])^{-1}\\ \pi^{-1}(\pi'(j))&=\pi''(j)\end{aligned}\tag{1}$$

Note that we have computed $\pi'(j)$-th column of the matrix $\boldsymbol{A}^T$ for all $0\le j<k$. □

For simplicity, in this part, we will use consider the matrices $\widetilde{\boldsymbol{Q}}_j$, $\overline{\boldsymbol{Q}}_j$ and $\boldsymbol{Q}_{\boldsymbol{d}[j]}$ as the matrices $\widetilde{\boldsymbol{Q}}$, $\overline{\boldsymbol{Q}}$ and $\boldsymbol{Q}$ respectively. Recall the `prepareDigestInput` function, it was taking $\boldsymbol{G}_0$ and a monomial matrix $\widetilde{\boldsymbol{Q}}=(\widetilde{\pi},\, \widetilde{\boldsymbol{v}})$ as input and $\overline{\boldsymbol{Q}}$ is one of the outputs of the function. The $\overline{\boldsymbol{Q}}$ is computed in a way that $\boldsymbol{G}_0\overline{\boldsymbol{Q}}=\boldsymbol{G}_0(\widetilde{\boldsymbol{Q}})^T[*,J^\dagger]$, where $J^\dagger$ is an IS of $\boldsymbol{G}_0(\widetilde{\boldsymbol{Q}})^T$. From the definition of IS, we can say that $\boldsymbol{G}_0\overline{\boldsymbol{Q}}$ is a non-singular matrix. Observe that,

$$\boldsymbol{G}_0\overline{\boldsymbol{Q}}=[\overline{\boldsymbol{v}}_0\cdot\boldsymbol{g}_{\overline{\pi}(0)}\mid\overline{\boldsymbol{v}}_1\cdot\boldsymbol{g}_{\overline{\pi}(1)}\mid\cdots\mid\overline{\boldsymbol{v}}_{k-1}\cdot\boldsymbol{g}_{\overline{\pi}(k-1)}]\tag{2}$$

Where $\overline{\boldsymbol{Q}}$ is a partial monomial matrix represented by $(\overline{\pi},\, \overline{\boldsymbol{v}})$. Since the matrix representation in Eq. 2 is non-singular, the set $J=\{\overline{\pi}(i)\,:\,i\in\mathbb{Z}_k\}$ is the IS of $\boldsymbol{G}_0$.

Now consider we are given the pair $(\widetilde{\boldsymbol{Q}},\, \boldsymbol{Q}^T\overline{\boldsymbol{Q}})$, where $\boldsymbol{Q}$ represented by $(\pi,\, \boldsymbol{v})$ and $\overline{\boldsymbol{Q}}$ is generated from $\widetilde{\boldsymbol{Q}}$ using the function `prepareDigestInput`. Now, $\boldsymbol{Q}^T\overline{\boldsymbol{Q}}$ is a partial monomial matrix and let it be represented by $(\pi_*,\, \boldsymbol{v}_*)$ then from Lemma 1, we can write that for any $j\in\mathbb{Z}_k$

$$\begin{aligned}\pi^{-1}(\overline{\pi}(i))&=\pi_*(i)\\ \boldsymbol{v}[\pi^{-1}(\overline{\pi}(i))]&=\boldsymbol{v}_*[i](\overline{\boldsymbol{v}}[i])^{-1}.\end{aligned}\tag{3}$$

This Eq. 3 gives us the partially recovered secret *i.e.* only k many columns of $\boldsymbol{Q}^T$. According to the definition of $\overline{\pi}$, the set $\{\overline{\pi}(i)\,:\,i\in\mathbb{Z}_k\}$ is the set J which is the information set of $\boldsymbol{G}_0$. Now, if $\boldsymbol{Q}$ is a secret monomial then from the key generation of LESS, we can say that $\widehat{\boldsymbol{G}}=\mathrm{RREF}(\boldsymbol{G}_0(\boldsymbol{Q}^T)^{-1})$ is a part of the public key. We can further write $\widehat{\boldsymbol{G}}=\boldsymbol{S}\boldsymbol{G}_0(\boldsymbol{Q}^T)^{-1}$ for some non-singular matrix $\boldsymbol{S}$. Consider $\widehat{\boldsymbol{G}}=[\widehat{\boldsymbol{g}}_0\mid\widehat{\boldsymbol{g}}_1\mid\cdots\mid\widehat{\boldsymbol{g}}_{n-1}]$ then for all $i\in\mathbb{Z}_n$ we have $\widehat{\boldsymbol{g}}_i=\boldsymbol{S}\cdot\big((\boldsymbol{v}[i])^{-1}\cdot\boldsymbol{g}_{\pi(i)}\big)$ which implies that for all $i\in\mathbb{Z}_n$,

$$\widehat{\boldsymbol{g}}_{\pi^{-1}(i)}=\boldsymbol{S}\cdot\big((\boldsymbol{v}[\pi^{-1}(i)])^{-1}\cdot\boldsymbol{g}_i\big)\tag{4}$$

Consider that the set J have the elements j_0, j_1, $\cdots$, j_{k-1}, and we take the matrix $\boldsymbol{G}^*=[\widehat{\boldsymbol{g}}_{\pi^{-1}(j_0)}\mid\widehat{\boldsymbol{g}}_{\pi^{-1}(j_1)}\mid\cdots\mid\widehat{\boldsymbol{g}}_{\pi^{-1}(j_{k-1})}]$ and also take the matrix

$$\boldsymbol{G}'=[(\boldsymbol{v}[\pi^{-1}(j_0)])^{-1}\cdot\boldsymbol{g}_{j_0}\mid(\boldsymbol{v}[\pi^{-1}(j_1)])^{-1}\cdot\boldsymbol{g}_{j_1}\mid\cdots\mid(\boldsymbol{v}[\pi^{-1}(j_{k-1})])^{-1}\cdot\boldsymbol{g}_{j_{k-1}}]$$

From Eq. 4, we have $\boldsymbol{G}^* = \boldsymbol{S}\boldsymbol{G}'$ and since J is an IS of $\boldsymbol{G}_0$, so $\boldsymbol{G}'$ is a non-singular matrix. Also $\boldsymbol{G}'$ and $\boldsymbol{G}^*$ are both computable as for each $j \in J$, $\pi^{-1}(j)$ and $\boldsymbol{v}[\pi^{-1}(j)]$ are already recovered. Therefore, we can compute $\boldsymbol{S} = \boldsymbol{G}^* \cdot (\boldsymbol{G}')^{-1}$. Finally, we have $\boldsymbol{S}^{-1}\widehat{\boldsymbol{G}} = \boldsymbol{G}_0(\boldsymbol{Q}^T)^{-1}$, where $\boldsymbol{S}$, $\boldsymbol{G}_0$ and $\widehat{\boldsymbol{G}}$ are known. Using Alg. 3, we can recover the full secret.

Algorithm 3 `getColumnPermutation`$(\widehat{\boldsymbol{G}}, \boldsymbol{G}_0, \boldsymbol{S})$

Input: The partially recovered secret $\pi : J_* \to J$ and $\boldsymbol{v}[j]$ $\forall j \in J_*$, where $J_* = \{\pi^{-1}(i) : i \in J\}$, public information $\boldsymbol{G}_0$ and $\widehat{\boldsymbol{G}}$, recovered matrix $\boldsymbol{S}$
Output: Outputs rest of the secret $\pi : J_*^c \to J^c$ and $\boldsymbol{v}[j]$ $\forall j \in J_*^c$

```
[g_0 | g_1 | ··· | g_{n-1}] ← G_0
[ĝ_0 | ĝ_1 | ··· | ĝ_{n-1}] ← Ĝ
for j ∈ J^c do
    for i ∈ J_*^c do
        for a ∈ F_q do
            if g_j = a·(S^{-1} ĝ_i) then
                assign π(i) ← j
                assign v[i] ← a
```

We can conclude that from one pair $(\widetilde{\boldsymbol{Q}}_j,\ \boldsymbol{Q}^T_{\boldsymbol{d}[j]}\overline{\boldsymbol{Q}}_j)$, we can recover the secret monomial matrix $\boldsymbol{Q}^T_{\boldsymbol{d}[j]}$, where $\boldsymbol{d}[j] \neq 0$. However, we will not receive the pair $(\widetilde{\boldsymbol{Q}}_j,\ \boldsymbol{Q}^T_{\boldsymbol{d}[j]}\overline{\boldsymbol{Q}}_j)$ if the signatures are generated by executing the signing algorithm properly. Therefore, we must find strategies to disrupt the normal flow of execution to help us get such pairs. Also, note that, if the number of secret monomial matrices$(s-1)$ is greater than one, then receiving only one such pair is not enough to retrieve all the secret monomials. So, we may require multiple faulted signatures to receive several such pairs and finally recover all the secret monomial matrices. All of these analysis are briefly described in the later sections.

3.2 Identification of attack surfaces

As we observed that having one pair of the form $(\widetilde{\boldsymbol{Q}}_j,\ \boldsymbol{Q}^T_{\boldsymbol{d}[j]}\overline{\boldsymbol{Q}}_j)$ is enough to recover the secret matrix $\boldsymbol{Q}^T_{\boldsymbol{d}[j]}$, where $\boldsymbol{d}[j] \neq 0$. Also, observe that, in LESS, there are $s-1$ secret monomial matrices $\boldsymbol{Q}_i$ for $1 \leq i \leq s-1$, and t ephemeral monomial matrices $\widetilde{\boldsymbol{Q}}_j$ for $0 \leq j < t$ as described in Section 2.2. Hence, our goal is to find at least one pair of the form $(\widetilde{\boldsymbol{Q}}_j,\ \boldsymbol{Q}^T_{\boldsymbol{d}[j]}\overline{\boldsymbol{Q}}_j)$, where $1 \leq \boldsymbol{d}[j] \leq s-1$ and $0 \leq j < t$ by manipulating the signing algorithm.

Note that, LESS is a code-based signature scheme based on the sigma-protocol with Fiat-Shamir transformation. In Alg. 2, the signer generates the random challenge $\boldsymbol{d}$ (fixed weight digest), from commitment (cmt) using the pseudo-random function `CSPRNG`. Any fault injection before the challenge generation may modify the challenge value, but that is an output of a pseudo-random function. This would not help, as

we need to recover the secret key. Therefore, we have targeted to inject a fault after the generation of $\boldsymbol{d}$.

Modification of the vector $\boldsymbol{d}$: As we can see from Alg. 2, the digest $\boldsymbol{d}$ ($\boldsymbol{d}[i]$ for $0 \leq i < t$) value decides whether $\widetilde{\boldsymbol{Q}}_i$ is revealed or $\boldsymbol{Q}^T_{\boldsymbol{d}[i]}\overline{\boldsymbol{Q}}_i$ is revealed. Therefore, the most obvious target for fault injection is the digest $\boldsymbol{d}$ to reveal both $\widetilde{\boldsymbol{Q}}_i$ and $\boldsymbol{Q}^T_{\boldsymbol{d}[i]}\overline{\boldsymbol{Q}}_i$ for some i. If we modify some value $\boldsymbol{d}[i]$ of $\boldsymbol{d}$ (line 11 in Alg. 2) from 0 to some non-zero value r by injecting fault, then we will get the information of $\boldsymbol{Q}^T_r\overline{\boldsymbol{Q}}_i$ instead of getting information of $\widetilde{\boldsymbol{Q}}_i$. Similarly, if we change the value of $\boldsymbol{d}[i]$ from non-zero value r to 0, then we will get the information of $\widetilde{\boldsymbol{Q}}_i$ instead of getting information about $\boldsymbol{Q}^T_r\overline{\boldsymbol{Q}}_i$. In both cases, we do not receive $\widetilde{\boldsymbol{Q}}_i$ and $\boldsymbol{Q}^T_r\overline{\boldsymbol{Q}}_i$ together. Therefore, modifying the $\boldsymbol{d}$ value does not satisfy our purpose.

Algorithm 4 `SeedTreePaths`

Input: The *Seed Tree* $\boldsymbol{seed}$ and the vector $\boldsymbol{f}$.
Output: Outputs $\boldsymbol{TreeNode}$ a subset of *Seed Tree* which consists only the $\texttt{seed}_i$'s that does not correspond to $\boldsymbol{f}[i]=1$.

1: **for** $i=0$; $i<4l-1$; $i=i+1$ **do**
2: $\quad\boldsymbol{x}[i]=0$
3: $\boldsymbol{x} \leftarrow$ `compute_seeds_to_publish`$(\boldsymbol{f}, \boldsymbol{x})$ ▷ (Alg. 5)
4: $j=0$
5: **for** $i=0$; $i<4l-1$; $i=i+1$ **do**
6: $\quad$**if** ($\boldsymbol{x}[i]=0$ and $\boldsymbol{x}[Parent(i)]=1$) **then**
7: $\quad\quad\boldsymbol{TreeNode}[j]=\boldsymbol{seed}[i]$
8: $\quad\quad j=j+1$
9: return $\boldsymbol{TreeNode}$

One might think of using the cases $\boldsymbol{d}[i]=0$ bypassing the check $\boldsymbol{d}[i]\neq 0$ (line 19) using a fault. However, $\boldsymbol{mseed}[0]$ does not exist and might cause an error during execution. Therefore, modifying anything from lines 18-24 would not benefit us. Now, we analyse the remaining steps (lines 11-16) of Alg. 2. In these steps, we can modify the value of the vector $\boldsymbol{f}$. Also, the `SeedTreePaths` algorithm is another potential candidate for fault injection, which is presented in Alg. 4. It uses an auxiliary function `compute_seeds_to_publish` described in Alg. 5. In the `SeedTreePaths` procedure, a tree $\boldsymbol{x}$ of size $4l-1$ is initialized with all zero. We call this tree as *Reference Tree*. In Alg. 5, the values of the leaf nodes of the *Reference Tree* are updated according to the $\boldsymbol{f}$ *i.e.*, $\boldsymbol{x}[2l-1+i]$ are assigned the value $\boldsymbol{f}[i]$ for all $0 \leq i < t$. The remaining nodes of the *Reference Tree* are assigned the value using the formula $\boldsymbol{x}[i]=\boldsymbol{x}[2i+1] \vee \boldsymbol{x}[2i+2]$, signifying that if either child has a value of 1, the corresponding parent will be assigned 1. In this way, the value of the *Reference Tree* $\boldsymbol{x}$ has been updated in a bottom-up approach. Now, some locations in *Seed Tree* are to be published as $\boldsymbol{TreeNode}$ with the help of the *Reference Tree*. Alg. 4 checks if the i-th node of *Reference Tree* $\boldsymbol{x}[i]$ is

Algorithm 5 `compute_seeds_to_publish`

Input: A vector $\boldsymbol{f}$ of size t and the *Reference Tree* $\boldsymbol{x}$.
Output: Modified *Reference Tree* $\boldsymbol{x}$.

```
for i=0; i<t; i=i+1 do
    x[2l-1+i] = f[i]
for i=2l-2; i≥0; i=i-1 do
    x[i] = x[2i+1] ∨ x[2i+2]
return x
```

zero and its parent $\boldsymbol{x}[Parent(i)]$ is 1, where the function $Parent(\cdot)$ is defined as follows:

$$Parent(i) = \begin{cases} 0 & \text{if } i = 0 \\ \lfloor \frac{i-1}{2} \rfloor & \text{otherwise} \end{cases}$$

If the validity check is satisfied, then $\boldsymbol{seed}[i]$, the i-th node of the *Seed Tree* is appended to $\boldsymbol{TreeNode}$.

Example 1. In Fig. 2, we have given an example for leaf nodes $2l = 8$ and the vector $\boldsymbol{d}$ is chosen as (0, 3, 1, 1, 0, 0, 0, 0). Then, the vector $\boldsymbol{f}$ will be (0, 1, 1, 1, 0, 0, 0, 0). From Fig. 2, we can see that for $i = 2, 7$ the condition "$\boldsymbol{x}[i] = 0$ and $\boldsymbol{x}[Parent(i)] = 1$" is satisfied. Therefore, the vector $\boldsymbol{TreeNode} = (\boldsymbol{seed}[2], \boldsymbol{seed}[7])$ and $\boldsymbol{rsp} = (\boldsymbol{Q}^T_{\boldsymbol{d}[1]}\overline{\boldsymbol{Q}}_1, \boldsymbol{Q}^T_{\boldsymbol{d}[2]}\overline{\boldsymbol{Q}}_2, \boldsymbol{Q}^T_{\boldsymbol{d}[3]}\overline{\boldsymbol{Q}}_3) = (\boldsymbol{Q}^T_3\overline{\boldsymbol{Q}}_1, \boldsymbol{Q}^T_1\overline{\boldsymbol{Q}}_2, \boldsymbol{Q}^T_1\overline{\boldsymbol{Q}}_3)$ will be extracted from *Seed Tree* is revealed at the end. From the seeds $\boldsymbol{seed}[2]$ $\boldsymbol{seed}[7]$, we can compute the leaf seeds $\boldsymbol{seed}[7]$, $\boldsymbol{seed}[11]$, $\boldsymbol{seed}[12]$, $\boldsymbol{seed}[13]$, $\boldsymbol{seed}[14]$ which are equals to the leaf seeds $\boldsymbol{ESEED}[0]$, $\boldsymbol{ESEED}[4]$, $\boldsymbol{ESEED}[5]$, $\boldsymbol{ESEED}[6]$, $\boldsymbol{ESEED}[7]$ respectively. From these seeds, we can compute the matrices $\widetilde{\boldsymbol{Q}_0}$, $\widetilde{\boldsymbol{Q}_4}$, $\widetilde{\boldsymbol{Q}_5}$, $\widetilde{\boldsymbol{Q}_6}$, $\widetilde{\boldsymbol{Q}_7}$. From the output of `LESS_Sign` algorithm, we will get either $\widetilde{\boldsymbol{Q}_j}$ or $\boldsymbol{Q}^T_{\boldsymbol{d}[j]}\overline{\boldsymbol{Q}_j}$.

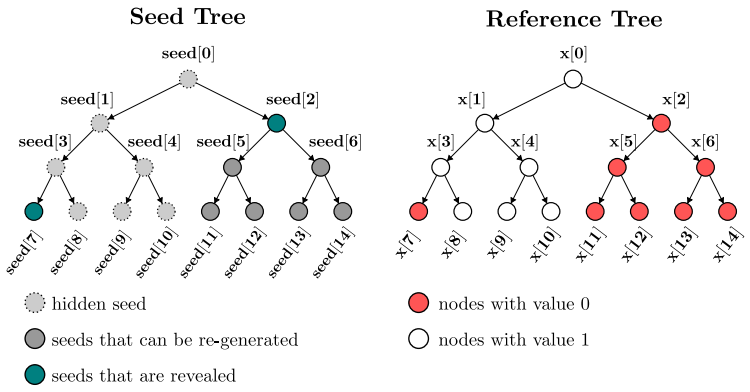

Fig. 2: Example for extraction of $\boldsymbol{TreeNode}$ from Seed Tree using Reference Tree, following Alg. 4

As we can observe from Alg. 4, the seeds from *Seed Tree* that are revealed as $\boldsymbol{TreeNode}$ are directly associated with the values in $\boldsymbol{f}$ and the *Reference Tree* $\boldsymbol{x}$. Therefore, we can try injecting faults in various locations of $\boldsymbol{f}$ or $\boldsymbol{x}$.

Modification of any node of the Reference Tree $\boldsymbol{x}$: Here, we investigate the effect of modification of some fixed i-th value of the tree $\boldsymbol{x}$ in Alg. 5. Without loss of generality, assume $\boldsymbol{x}[i_0]$, $\boldsymbol{x}[i_1]$, $\cdots$, $\boldsymbol{x}[i_{r-1}]$ be the leaf nodes of the subtree with root node $\boldsymbol{x}[i]$, where $r \geq 1$. Now, suppose we inject a fault in the signature algorithm to modify the value of i-th node of the *Reference Tree* $\boldsymbol{x}$. In that case, the signature algorithm will give us the faulted signature. However, even if we try to inject a fault in a physical machine, the fault can only occur with a certain probability. If we assume that the fault injection is successful, even then, there are several cases:

- **Case 1:** The node $\boldsymbol{x}[i]$ is 0 in the non-faulted case. In this case, since the actual value $\boldsymbol{x}[i]$ is 0, all the leaf nodes in the subtree rooted at $\boldsymbol{x}[i]$ must be zero. Hence, the vectors $\boldsymbol{f}$ and $\boldsymbol{d}$ do not have any non-zero value at the positions corresponding to the leaf nodes $\boldsymbol{x}[i_0]$, $\boldsymbol{x}[i_1]$, $\cdots$, $\boldsymbol{x}[i_{r-1}]$. Therefore, the $\boldsymbol{rsp}$ does not contain multiplication of any secret monomial matrix with the partial monomial matrix $\overline{\boldsymbol{Q}}_{i_j-2l+1}$, where $0 \leq j < r$ *i.e.*, we can not get any information about the secret matrices.
- **Case 2:** The node $\boldsymbol{x}[i]$ is 1 in the non-faulted case, and after the fault injection, it has changed to 0. Since the *Reference Tree* is updated in a bottom-up approach, the modification of the i-th node $\boldsymbol{x}[i]$ may affect the ancestors of $\boldsymbol{x}[i]$. Consequently, it may change the root node $\boldsymbol{x}[0]$. In this case, assume that it changes the value of the root node $\boldsymbol{x}[0]$ to 0. This case can occur only if all non-zero leaves fall under the subtree rooted at $\boldsymbol{x}[i]$. Since the value of the root node is zero, all the ancestors of $\boldsymbol{x}[i]$ including $\boldsymbol{x}[0]$ are zero. Therefore, neither $\boldsymbol{seed}[i]$ nor any of its ancestors in *Seed Tree* is released because the Alg. 4 requires the parent of $\boldsymbol{x}[j]$ to be 1 if we want to release the $\boldsymbol{seed}[j]$, *i.e.* such fault does not provide any advantage to us. Therefore, the nodes in the subtree rooted at $\boldsymbol{x}[i]$ do not affect the fault injection, so no extra information can be achieved from the released seeds corresponding to this subtree.

  Example 2. We consider the fixed digest vector $\boldsymbol{d}$, $\boldsymbol{f}$, and the *Reference Tree* $\boldsymbol{x}$ of Example 1 in a non-faulted scenario. We modify the value of $\boldsymbol{x}[1]$ from $1 \rightarrow 0$ that changes the value of $\boldsymbol{x}[0]$ from $1 \rightarrow 0$. Fig. 3 represents the *Reference Tree* and the related node of *Seed Tree* in faulted case. In this case, only $\boldsymbol{x}[7]$ satisfies the condition "$\boldsymbol{x}[7]=0$ and $\boldsymbol{x}[Parent(7)]=1$". Therefore, $\boldsymbol{TreeNode}$ will be ($\boldsymbol{seed}[7]$) and $\boldsymbol{rsp} = (\boldsymbol{Q}^T_{\boldsymbol{d}[1]}\overline{\boldsymbol{Q}}_1, \boldsymbol{Q}^T_{\boldsymbol{d}[2]}\overline{\boldsymbol{Q}}_2, \boldsymbol{Q}^T_{\boldsymbol{d}[3]}\overline{\boldsymbol{Q}}_3) = (\boldsymbol{Q}^T_3\overline{\boldsymbol{Q}}_1, \boldsymbol{Q}^T_1\overline{\boldsymbol{Q}}_2, \boldsymbol{Q}^T_1\overline{\boldsymbol{Q}}_3)$. None of the monomial matrices $\widetilde{\boldsymbol{Q}}_1, \widetilde{\boldsymbol{Q}}_2, \widetilde{\boldsymbol{Q}}_3$ can be generated from $\boldsymbol{seed}[7]$. Therefore, we are unable to recover any secret key-related information from this faulted signature.
- **Case 3:** The node $\boldsymbol{x}[i]$ is 1 in the non-faulted case. After the fault injection, it has changed to 0, but $\boldsymbol{x}[0]$ remains 1. Since the actual value of $\boldsymbol{x}[i]$ is 1, there exists some leaf node $\boldsymbol{x}[i_j]$ such that $\boldsymbol{x}[i_j]=1$. Therefore, it follows that $\boldsymbol{f}[i_j-2l+1]$ is non-zero and consequently $\boldsymbol{d}[i_j-2l+1]$ is also non-zero. Without loss of generality,

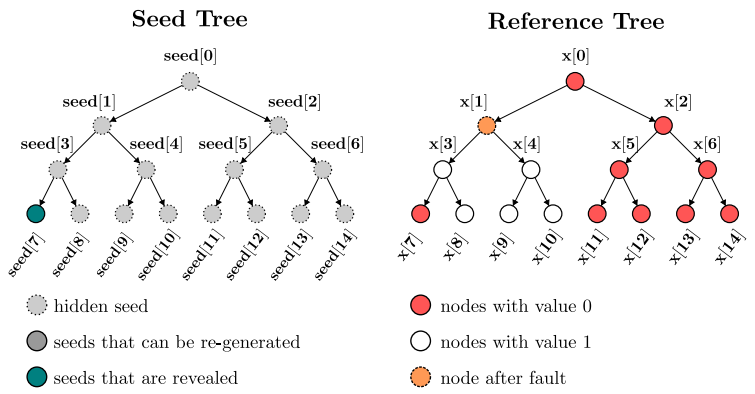

Fig. 3: Example of Case 2

assume that $i_j - 2l + 1 = k'$ then $\boldsymbol{rsp}$ contains $\boldsymbol{Q}^T_{\boldsymbol{d}[k']}\overline{\boldsymbol{Q}}_{k'}$. Also, since the faulted value of $\boldsymbol{x}[i]$ is 0 and $\boldsymbol{x}[0] = 1$, so $\boldsymbol{TreeNode}$ will contain either $\boldsymbol{seed}[i]$ or any of its ancestors in *Seed Tree* from which we can generate the leaf node $\boldsymbol{seed}[i_j]$ of the subtree rooted at $\boldsymbol{seed}[i]$. Hence, the ephemeral key $\boldsymbol{ESEED}[k']$ and consequently the monomial matrix $\widetilde{\boldsymbol{Q}}_{k'}$ can be generated. Therefore, we retrieve the pair $(\widetilde{\boldsymbol{Q}}_{k'}, \boldsymbol{Q}^T_{\boldsymbol{d}[k']}\overline{\boldsymbol{Q}}_{k'})$.

Example 3. We consider the fixed digest vector $\boldsymbol{d}$, $\boldsymbol{f}$, and the *Reference Tree* $\boldsymbol{x}$ of Example 1 in a non-faulted scenario. We modify the value of $\boldsymbol{x}[3]$ from $1 \rightarrow 0$ that does not change the value of $\boldsymbol{x}[0]$. Fig. 4 represents the *Reference Tree* and the related node of *Seed Tree* in the faulted case. From this Fig. 4 we can see that for $i = 2$, 3 the condition "$\boldsymbol{x}[i] = 0$ and $\boldsymbol{x}[Parent(i)] = 1$" is satisfied. Therefore, $\boldsymbol{TreeNode}$ will be ($\boldsymbol{seed}[2]$, $\boldsymbol{seed}[3]$) and $\boldsymbol{rsp} = (\boldsymbol{Q}^T_3\overline{\boldsymbol{Q}}_1, \boldsymbol{Q}^T_1\overline{\boldsymbol{Q}}_2, \boldsymbol{Q}^T_1\overline{\boldsymbol{Q}}_3)$. Now $\boldsymbol{seed}[3]$ is contained in the signature component *Reference Tree* that we can generate the seed $\boldsymbol{seed}[8]$ to the $8 - 2l + 1 = 8 - 8 + 1 = 1$-st monomial matrix $\widetilde{\boldsymbol{Q}}_1$. So, from this faulted signature, we found the pair $(\widetilde{\boldsymbol{Q}}_1, \boldsymbol{Q}^T_3\overline{\boldsymbol{Q}}_1)$ that help us find the information of the matrix $\boldsymbol{Q}^T_3$.

Modification of the vector $\boldsymbol{f}$: The vector $\boldsymbol{f}$ is computed by using the fixed digest vector $\boldsymbol{d}$. If the i-th element of $\boldsymbol{d}$ holds a non-zero value, $\boldsymbol{f}[i]$ is assigned the value of 1; otherwise, it is set to zero. If we modify the i-th value $\boldsymbol{f}[i]$ by injecting fault, then the $(2l - 1 + i)$-th leaf node $\boldsymbol{x}[2l - 1 + i]$ of the *Reference Tree* will be changed. The effect of this fault will be the same as the above modification of any leaf node of the Reference Tree $\boldsymbol{x}$.

From the above, we can observe that the attack surfaces are different as in the second attack component, we change the value of any $\boldsymbol{f}[i]$, and in the first attack component, we change the value of any $\boldsymbol{x}[i]$. However, we can say that modifying the value of any value $\boldsymbol{f}[i]$ is imposing the same effect as modifying the corresponding leaf node of $\boldsymbol{x}[2l - 1 + i]$. Therefore, from now onwards, we only discuss the modification of any node of the *Reference Tree* $\boldsymbol{x}$.

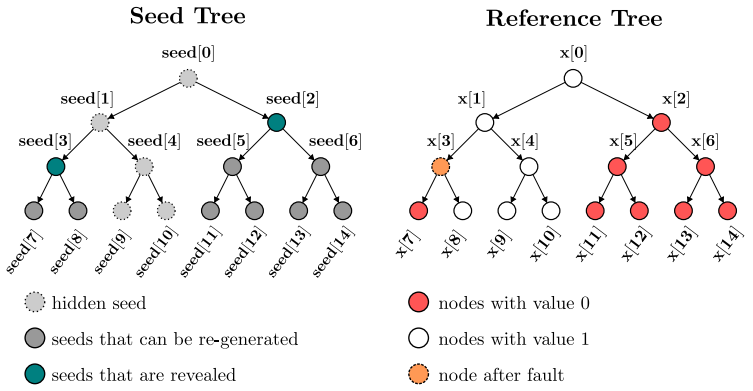

Fig. 4: Example of Case 3

3.3 Fault models

In this section, we describe the fault models that will help us to recover the secret key. Our attack just requires changing a bit ($1 \rightarrow 0$ for LESS). Here we discussed in detailed how each fault model can be utilize to realize our attack. Mainly, our fault assumptions can be realized by "skipping one condition check" or "forcing one data corruption in $\boldsymbol{f}$ or $\boldsymbol{x}$". We assume that the faulted location is arbitrary but known to the attacker.

Skip the validity check condition in Alg. 4: If we skip the check "$\boldsymbol{x}[i] = 0$ and $\boldsymbol{x}[Parent(i)] = 1$" in Alg. 4 for a fixed i, then $\boldsymbol{seed}[i]$ will always contained in $\boldsymbol{TreeNode}$. Without loss of generality, let $\boldsymbol{seed}[i_0]$, $\cdots$, $\boldsymbol{seed}[i_{r-1}]$ be the leaf nodes of the subtree rooted the node $\boldsymbol{seed}[i]$ of *Seed Tree* and $\boldsymbol{x}[i_0]$, $\cdots$, $\boldsymbol{x}[i_{r-1}]$ be the corresponding leaf nodes of the subtree rooted the node $\boldsymbol{x}[i]$ of the *Reference Tree*. If $\boldsymbol{x}[i] = 1$, then there exists the leaf node say $\boldsymbol{x}[i_{j'}]$, where $0 \leq j' < r$ and $\boldsymbol{x}[i_{j'}] = 1$. This implies $\boldsymbol{f}[i_{j'} - 2l + 1] = 1$ and so, $\boldsymbol{d}[i_{j'} - 2l + 1]$ must be non-zero. Therefore, $\boldsymbol{rsp}$ must contain the matrix multiplication $\boldsymbol{Q}^T_{\boldsymbol{d}[i_{j'}-2l+1]} \overline{\boldsymbol{Q}}_{i_{j'}-2l+1}$. Since $\boldsymbol{d}[i_{j'} - 2l + 1]$ is non-zero, we are not supposed to have the information about the ephemeral matrix $\widetilde{\boldsymbol{Q}}_{i_{j'}-2l+1}$ in non-faulted case. However, from the $\boldsymbol{seed}[i]$, we can generate all leaf nodes of the subtree rooted in this node. Now, $\boldsymbol{seed}[i_{j'}]$ is the $i_{j'} - 2l + 1$-th leaf node $\boldsymbol{ESEED}[i_{j'} - 2l + 1]$ of the *Seed Tree*, and that helps us find the ephemeral monomial matrix $\widetilde{\boldsymbol{Q}}_{i_{j'}-2l+1}$. So, in this case, we can find some information of secret monomial matrix $\boldsymbol{Q}^T_{\boldsymbol{d}[i'_j-2l+1]}$ from the pair $(\widetilde{\boldsymbol{Q}}_{i_{j'}-2l+1}, \boldsymbol{Q}^T_{\boldsymbol{d}[i'_j-2l+1]} \overline{\boldsymbol{Q}}_{i_{j'}-2l+1})$. This is a model that we can use to mount the attack.

Previously, many works [28,9] have shown that instruction skips can be easily done with clock glitches, and the fault happens with very high probability. Mainly, in these works they have skipped the condition check instructions and store instructions. Recently, Keita et al. in [39] bypassed the validity check in the decapsulation procedure in post-quantum the key-encapsulation mechanism Kyber [3]. However, one may argue that skipping the validity check is the most important part as if we can skip this validity check for $i = 0$ in line-6 in Alg. 4, then $\boldsymbol{seed}[0]$ will be revealed. Henceforth

all $\widetilde{\boldsymbol{Q}}_i$'s would have been revealed. Therefore, one may want to protect this checking at any cost. In fact, the need to protect this validity check was previously noted by Oder et al. [26] for different post-quantum schemes (e.g. Kyber). Nevertheless, the data in $\boldsymbol{d}$ and *Reference Tree* $\boldsymbol{x}$ can be corrupted by skipping the storing instruction and forcing the data not to change.

Skip the store instruction in Alg. 5 to corrupt $\boldsymbol{x}$: All the nodes of the *Reference Tree* $\boldsymbol{x}$ are initialized by zero. If we can skip the store instruction $\boldsymbol{x}[i] = \boldsymbol{x}[2i+1] \vee \boldsymbol{x}[2i+2]$ for any i, then vaule of $\boldsymbol{x}[i]$ will remain zero. However, if the value of $\boldsymbol{x}[i]$ is supposed to be 1 in the non-faulted case, then $\boldsymbol{x}[i]$ will be modified after injecting this store instruction fault. As we discussed earlier, we can find the information of the secret matrix if the fault changes the value of $\boldsymbol{x}[i]$ from 1 to 0 and $\boldsymbol{x}[0]$ remains 1. A similar instruction skipping attack has been shown in [28].

Stuck-at-zero fault model to corrupt $\boldsymbol{x}$: A possible attack avenue is exploiting effective faults in the stuck-at model, where an attacker can try to alter the i-th intermediate value $\boldsymbol{x}[i]$ to a particular known value, e.g., to zero using stuck-at-zero fault [13,17,20] using voltage glitches or electromagnetic attacks. The effect of this fault is equivalent to the above "store instruction skipping" fault. So, this fault will allow us to find the secret matrix.

Rowhammer attack model to corrupt $\boldsymbol{x}$: Rowhammer [31] is a hardware bug identified in DRAMS (dynamic random access memory), where repeated row activations can cause bitflips in adjacent rows. This can also be a possible attack where bitflips ($1 \rightarrow 0$) can be employed to corrupt the data $\boldsymbol{x}[i]$. Recently, such an attack on Kyber using rowhammer has been shown in [24].

As we discussed, any one of the above fault models can generate effective faulted signatures. However, the definition of successful fault always depends on the fault model. For example, if we work on the first fault model, *i.e.*, "Skip the validity check condition in Alg. 4", then the successful-fault will be: successfully skipped the checking condition "$\boldsymbol{x}[i] = 0$ and $\boldsymbol{x}[Parent(i)] = 1$" for a known i. But, if we work on the second fault model, then the successful fault will be: successfully skipped the store instruction "$\boldsymbol{x}[i] = \boldsymbol{x}[2i+1] \vee \boldsymbol{x}[2i+2]$" for a known i.

From now on, we will discuss the second fault model to inject a fault, *i.e.*, we inject a fault to skip the i-th store instruction $Ins(i)$: "$\boldsymbol{x}[i] = \boldsymbol{x}[2i+1] \vee \boldsymbol{x}[2i+2]$" for a fixed known i. Since all the values of the *Reference Tree* $\boldsymbol{x}$ are initialized by zero, therefore for each successful fault, the value of $\boldsymbol{x}[i]$ will always be zero, where the position of fault location $\boldsymbol{x}[i]$ is known to the attacker. In the practical setup of this store instruction skip fault model, the following cases may arise:

- Successfully skipped the instruction $Ins(i)$, for the known i and outputs the signature we call it a successful faulted signature. This fault could be an effective or ineffective fault.
- Could not skip the instruction $Ins(i)$, for the known fixed i. In this case, we call the output signature an unsuccessful faulted signature.

In a physical device, faults can be induced with varying success rates. Even if there is a successful fault, the resulting faulty signature may or may not provide information about the secret key, as we observed earlier. A successful fault is called "effective"

if it reveals secret key information and "ineffective" if it does not. More explicitly, we will say that a successful fault is effective if the fault changes the value of $\boldsymbol{x}[i]$ from 1 to 0, but the value of the root node $\boldsymbol{x}[0]$ remains unchanged, (i.e., 1). Otherwise, the fault will be ineffective. We must identify the effective faulted signature from the received signature to find the errorless secret matrix. In the next Section 3.4, we will discuss the effective faulted signature detection method.

3.4 Effective fault detection

Let $\tau' = (salt,\ cmt,\ \boldsymbol{TreeNode'},\ \boldsymbol{rsp})$ be the received signature corresponding to the message m. We need to detect if the signature is generated from an "effective" or "ineffective" fault. The injected fault only affects the *Reference Tree* $\boldsymbol{x}$, so the signature components *salt*, *cmt* and $\boldsymbol{rsp}$ remain the same with the corresponding non-faulted signature components. We can compute the fixed digest vector $\boldsymbol{d}$ corresponding to the signature τ' from *cmt*. From the fixed digest vector $\boldsymbol{d}$, we can compute the vector $\boldsymbol{f}$ and the *Reference Tree* $\boldsymbol{x}$ for the non-faulted case. However, we can compute the successful faulted *Reference Tree* $\boldsymbol{x'}$ from the *Reference Tree* $\boldsymbol{x}$ by assigning the value of $\boldsymbol{x'}[i]=0$ and updating the ancestors of $\boldsymbol{x'}[i]$ accordingly, *i.e.*, $\boldsymbol{x'}$ should be the *Reference Tree* if the instruction $Ins(i)$ is skipped. Then, we will distinguish the "effective" faulted signature and "ineffective" faulted signature with the following process:

- **Step-1:** First, we will check whether $\boldsymbol{x}[i] = 0$ or not. If $\boldsymbol{x}[i] = 0$, then this is already a case of "ineffective fault", and we reject the signature. Otherwise, we will go to the next step.
- **Step-2:** Next, we will check whether $\boldsymbol{x'}[0]=0$ or not. If $\boldsymbol{x'}[0]=0$, then this is a case of "ineffective fault", and we reject the signature. Otherwise, from the *Reference Tree* $\boldsymbol{x'}$, we compute the size of successfully faulted $\boldsymbol{TreeNode'}$, say Δ_{exp} and the size of received $\boldsymbol{TreeNode'}$ say Δ_{rec}. We compare the values Δ_{exp} with Δ_{rec}.
- **Step-3:** If $\Delta_{rec} \neq \Delta_{exp}$, then the fault is unsuccessful, and we reject the signature. Otherwise, using *salt*, $\boldsymbol{TreeNode'}$ and $\boldsymbol{x'}$ we compute all the $\widetilde{\boldsymbol{Q}}_j$ where $\boldsymbol{d}[j]=0$. We apply the verification using these $\widetilde{\boldsymbol{Q}}_j$'s and $\boldsymbol{rsp}$. If the verification is successful, then we take the received signature as an effective faulted signature. Otherwise, we reject the signature.

Note that, in **Step-3** of the above process, $\boldsymbol{x}[i]$ is changed from $1 \rightarrow 0$ and $\boldsymbol{x}[0]=1$, but we still consider it as "unsuccessful" fault. This is because we want our fault detection method to detect whether our fault has been successfully injected exactly at the i-th location or not. If $\boldsymbol{x}[i]$ changed from $1 \rightarrow 0$, then there are two cases.

- *Case-1:* fault was successfully injected at i-th location.
- *Case-2:* fault was injected at a j-th location for $j \neq i$ and it has changed $\boldsymbol{x}[i]$.

We only consider *Case-1* as "successful-fault", but not *Case-2* as the fault is not injected at the i-th location in that case. In this procedure, we can detect that the faulted signature that is generated by successfully skipping the instruction $Ins(i)$ and that leaks the information about the secret matrix. Note that the targeted faulted location i

is arbitrary but known to the attacker. For simplicity, we will fix the targeted fault position i. We will check whether this fixed i-th store instruction $Ins(i)$ skipped and that leaks the information about the secret matrix or not. If this detection method passes, then we will use this signature. Otherwise, we will again query for another signature.

3.5 Attack template

In this section, first, we will describe how to obtain the secret monomial matrices from an effective faulted signature $\tau = (salt, cmt, \boldsymbol{TreeNode}, \boldsymbol{rsp})$ in Alg. 6. Let $\boldsymbol{x}[i]$ be the node in *Reference Tree* with height h, and $L_{\boldsymbol{x}[i]}$ be the set of all leaf nodes of the subtree rooted at $\boldsymbol{x}[i]$. We only need the leaf nodes from $L_{\boldsymbol{x}[i]}$ that coincide with the first t (the length of the digest $\boldsymbol{d}$) many leaf nodes of the full *Reference Tree*. Without loss of generality, assume that there are v many such leaves, and let the set of indices of these leaves be $I^{(i)}_{\text{leaf}} = \{j_1, j_2, \cdots, j_v\}$. From this effective faulted signature, all the secret matrices $\boldsymbol{Q}^T_{\boldsymbol{d}[j-2l+1]}$ will be recovered with Alg. 6, where $j \in I^{(i)}_{\text{leaf}}$ and $\boldsymbol{d}[j-2l+1] \neq 0$.

Algorithm 6 `Recover_Secret_Matrices`(τ, PK)

Input: Signature $\tau = (salt, cmt, \boldsymbol{TreeNode}, \boldsymbol{rsp})$, public key $PK = (gseed, \boldsymbol{G}_1, \cdots, \boldsymbol{G}_{s-1})$.
Output: The columns of secret matrices $\boldsymbol{Q}^T_{\boldsymbol{d}[j-2l+1]}$, where $j \in I^{(i)}_{\text{leaf}}$ and $\boldsymbol{d}[j-2l+1] \neq 0$.

1: $\boldsymbol{d} \leftarrow$ `CSPRNG`$(cmt, \mathbb{S}_{t,w})$
2: $\boldsymbol{seed} \leftarrow$ `SeedTreeUpdate`$(\boldsymbol{TreeNode}, salt, \boldsymbol{d})$
3: $\boldsymbol{ESEED} \leftarrow$ Leaf nodes of $\boldsymbol{seed}$ corresponding to $\boldsymbol{seed}[i]$
4: $\boldsymbol{G}_0 \leftarrow$ `CSPRNG`$(gseed, \mathbb{S}_{\texttt{RREF}})$
5: **for** $r = 1$; $r \leq v$; $r = r+1$ **do**
6: **if** $\boldsymbol{d}[j_r - 2l+1] \neq 0$ **and** $\boldsymbol{Q}_{\boldsymbol{d}[j_r-2l+1]}$ is not recovered **then**
7: $\widetilde{\boldsymbol{Q}}_{j_r-2l+1} \leftarrow$ `CSPRNG`$(\boldsymbol{ESEED}[j_r - 2l+1], M_n(q))$
8: $(\overline{\boldsymbol{Q}}_{j_r-2l+1}, \overline{\boldsymbol{V}}_{j_r-2l+1}) \leftarrow$ `PrepareDigestInput`$(\boldsymbol{G}_0, \widetilde{\boldsymbol{Q}}_{j_r-2l+1})$
9: $\boldsymbol{Q}^* = \boldsymbol{Q}^T_{\boldsymbol{d}[j_r-2l+1]} \overline{\boldsymbol{Q}}_{j_r-2l+1} \leftarrow$ `ExpandToMonomAction`$(\boldsymbol{rsp})$
10: Compute $\boldsymbol{Q}^T_{\boldsymbol{d}[j_r-2l+1]}$ from $(\boldsymbol{Q}^*, \overline{\boldsymbol{Q}}_{j_r-2l+1})$ ▷ following Section 3.1

Here, `SeedTreeUpdate` function takes the $\boldsymbol{TreeNode}$, *salt* and digest $\boldsymbol{d}$ and generates all the ephemeral seeds assuming the modified *Reference Tree* after effective fault. Since $\boldsymbol{seed}[i]$ is revealed in $\boldsymbol{TreeNode}$ after effective fault, we can say that $\boldsymbol{ESEED}[j-2l+1]$ for all $j \in I^{(i)}_{\text{leaf}}$ are revealed.

In this attack model, we are able to get into the victim's device and introduce the fault that causes it to bypass the $Ins(i)$ instruction. The `LESS_KeyGen` (Alg. 1) is a one-time operation from where the secret key $\text{SK} = (MSSED, gseed)$ and public key $\text{PK} = (gseed, \boldsymbol{G}_1, \cdots, \boldsymbol{G}_{s-1})$ are generated. But with this private key SK, the `LESS_Sign` (Alg. 2) can execute more than once. We follow the following subsequent actions to find the secret monomial matrices:

- **Step-1:** We generate a message, signature pair (m, τ) from the victim device.

- **Step-2:** After receiving the pair (m, τ), we will determine whether or not τ is an effective faulted signature. Go back to **Step-1** if the signature is not effective. If yes, then go to **Step-3**.
- **Step-3:** Using this signature τ, we will run the `Recover_Secret_Matrices` algorithm (Alg. 6) to determine the hidden monomial matrices.
- **Step-4:** Next, we will calculate whether or not the whole secret monomial matrices were obtained. We terminate the process if the secret matrices are recovered. Otherwise, we repeat the same procedure to obtain the remaining non-recovered columns.

3.6 Secret recovery from single fault

In this section, we calculate the expected number of secret monomials recovered from one effective faulted signature where the fault is injected at a node $\boldsymbol{x}[i]$ $(0 \leq i \leq 4l-2)$. Now, if there are m many non-zero leaves with distinct values in the subtree rooted at $\boldsymbol{x}[i]$, then we will get exactly m many pairs of the form $(\widetilde{\boldsymbol{Q}}_j, \boldsymbol{Q}^T_{\boldsymbol{d}[j]}\overline{\boldsymbol{Q}}_j)$ *i.e.* we recover m many secret monomials. In this section, first, we will estimate the value of m.

Suppose $L_{\boldsymbol{x}[i]}$ the set of leaf nodes in the subtree rooted at $\boldsymbol{x}[i]$ and let $|L_{\boldsymbol{x}[i]}| = \ell$. Let W be the random variable representing the number of leaf nodes in $L_{\boldsymbol{x}[i]}$ with non-zero value. X be a random variable that represents the number of distinct non-zero values of the leaf nodes in $L_{\boldsymbol{x}[i]}$. Then for any $0 \leq m \leq s-1$, we have

$$\Pr[X=m] = \sum_{r=m}^{w} \Pr[X=m \mid W=r] \cdot \Pr[W=r]$$

Where w is the weight of $\boldsymbol{d}$ and therefore $L_{\boldsymbol{x}[i]}$ can only have at most w many non-zero valued leaf nodes. First, we will calculate $\Pr[X=m \mid W=r]$, which is the probability that r many non-zero leaves take exactly m many distinct values. These m distinct values can be chosen from $(s-1)$ possible values in $\binom{s-1}{m}$ ways. Now, we have to assign all these m values to the r many leaf nodes. We first partition the r locations into m many non-empty subsets, which can be done in $S(r, m)$ many ways. This $S(r, m)$ is a *Stirling number of the second kind* [32]. Now, each of the m many subsets can be assigned a unique non-zero value, which can be done in $m!$ ways. So, the r many leaf nodes can be assigned m distinct value in $m!\binom{s-1}{m}S(r, m)$ ways. Therefore

$$\Pr[X=m \mid W=r] = \frac{m!\binom{s-1}{m}S(r, m)}{(s-1)^r}$$

Now, $\boldsymbol{d}$ has weight w and $\Pr[W=r]$ is the probability that the ℓ many locations of $\boldsymbol{d}$ corresponding to the leaf nodes in $L_{\boldsymbol{x}[i]}$ has exactly r many non-zero values and the last $t-\ell$ many locations has $w-r$ many non-zero values. Therefore,

$$\Pr[W=r] = \frac{\binom{\ell}{r}\binom{t-\ell}{w-r}}{\binom{t}{w}}$$

Now we can calculate $\Pr[X=m]$ for all $0 \leq m \leq s-1$. However, we are interested in finding the expected number of secret monomials with one single fault, which is the expectation of the random variable X.

$$\begin{aligned}\mathbb{E}[X] &= \sum_{m=1}^{s-1} m \cdot \Pr[X=m] \\ &= \sum_{m=1}^{s-1} m \left(\sum_{r=m}^{w} \frac{m! \binom{s-1}{m} S(r,\, m)}{(s-1)^r} \cdot \frac{\binom{\ell}{r}\binom{t-\ell}{w-r}}{\binom{t}{w}} \right)\end{aligned}$$

With only one single faulted signature the expected number of secret monomials that we recover is $\mathbb{E}[X]$ but the total number of secret monomials is $(s-1)$. Therefore, we need multiple faulted signatures to recover all the secret monomials.

4 Extending our attack to CROSS

CROSS uses $\mathbb{E}^n$ a commutative group isomorphic to $(\mathbb{F}_z^n, +)$, where n, z are parameters of the signature and G is a subgroup of $\mathbb{E}^n$. Here $\boldsymbol{e}$ is a long-term secret vector which is used to generate signatures. Therefore, the attacker can generate multiple valid signatures using the secret $\boldsymbol{e}$ information. Similar to the attack on LESS, the target here is to find the information of the secret vector $\boldsymbol{e}$. In the Alg. 7, we can observe that if we have information of one single pair $(\boldsymbol{e'}^{(i)},\, f^{(i)} = (\boldsymbol{y}^{(i)},\, \sigma^{(i)},\, c_1^{(i)}))$, then we can compute the secret $\boldsymbol{e}$ by $\boldsymbol{e} = \sigma^{(i)}(\boldsymbol{e'}^{(i)})$. Therefore, we aim to find one such pair corresponding to any i for full key recovery.

In Alg. 7, the function `publish_seeds` (line 22) works equivalent to the function `SeedTreePaths` (Alg. 4) used in LESS signature. Using the digest vector $\boldsymbol{b}$, the function `publish_seeds` first creates a *Reference Tree* say $\boldsymbol{y}$ in a bottom-up approach like LESS signature. The only difference in this `Reference Tree` $\boldsymbol{y}$ is that the flag of the published seed is defined as 1 and unpublished seed notation as 0, whereas in LESS, the authors define the opposite. However, Both use equivalent concepts. Like LESS, we require the modification of any node of the *Reference Tree* from 1(flag of the unpublished seed) $\rightarrow$0(flag of the published seed) to get an effective faulted signature. Therefore, we need the modification from $0 \rightarrow 1$ to get an effective faulted signature. We can detect the effective faulted signature here using a similar technique that we used in Section 3.4 to detect effective fault for LESS signature.

Let us assume that we apply fault injection to the CROSS signature of a victim's device such that the value of $\boldsymbol{y}[i]$ has been changed from $0 \rightarrow 1$. Consider an effective faulted signature as $\tau = \left\{ Salt,\, c_0,\, c_1,\, h,\, \boldsymbol{SeedPath},\, \boldsymbol{MerkleProofs},\, \left\{ f^{(i)} \right\}_{i \notin J} \right\}$. Since τ is an effective-faulted signature, therefore we will get the seed $\boldsymbol{seed}[i]$. All the leaf nodes of the subtree say $L_{\boldsymbol{x}[i]} = \{\boldsymbol{x}[j_1],\, \cdots,\, \boldsymbol{x}[j_v]\}$ rooted as $\boldsymbol{seed}[i]$ can be computed from $\boldsymbol{seed}[i]$. i.e., $\boldsymbol{ESEED}[j_1 - 2l],\, \cdots,\, \boldsymbol{ESEED}[j_v - 2l]$ will be the corresponding leaf ephemeral seeds. Now, we will find the secret key $\boldsymbol{e}$ using the Alg. 8. The function `SeedTreeUpdate` works the same way we defined it in Section 3.5.

Algorithm 7 `CROSS_Sign` $(Msg, \boldsymbol{e})$

Input: Secret key $\boldsymbol{e} \in G$ and message Msg where $G \subset \mathbb{E}^n$, $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$ are public key satisfying $\boldsymbol{s} = \boldsymbol{e}\boldsymbol{H}^T$

Output: Signature $\tau = \left\{ Salt,\, c_0,\, c_1,\, h,\, \boldsymbol{SeedPath},\, \boldsymbol{MerkleProofs},\, \left\{ f^{(i)} \right\}_{i \notin J} \right\}$

Sample $MSeed \xleftarrow{\$} \{0,1\}^{\lambda}$, $Salt \xleftarrow{\$} \{0,1\}^{2\lambda}$
Generate $\boldsymbol{Seed} = \texttt{SeedTree}(MSeed, Salt)$
$\boldsymbol{ESEED}[1], \cdots, \boldsymbol{ESEED}[t] =$ Leaf nodes of $\boldsymbol{Seed}$
for $i=1,\, i \le t,\, i=i+1$ **do**
    Sample $(Seed^{(\boldsymbol{u}')}, Seed^{(\boldsymbol{v}')}) \xleftarrow{\boldsymbol{ESEED}[i]} \{0,1\}^{2\lambda}$
    Sample $\boldsymbol{u}'^{(i)} \xleftarrow{Seed^{(\boldsymbol{u}')}} \mathbb{F}_p^n$, $\boldsymbol{e}'^{(i)} \xleftarrow{Seed^{(\boldsymbol{e}')}} G$
    Compute $\sigma^{(i)} \in G$ such that $\sigma^{(i)}(\boldsymbol{e}'^{(i)}) = \boldsymbol{e}$
    Set $\boldsymbol{u}^{(i)} = \sigma^{(i)}(\boldsymbol{u}'^{(i)})$
    Compute $\widetilde{\boldsymbol{s}}^{(i)} = \boldsymbol{u}^{(i)}\boldsymbol{H}^T$
    Set $c_0^{(i)} = \texttt{Hash}(\widetilde{\boldsymbol{s}}^{(i)}, \sigma^{(i)}, Salt, i)$
    Set $c_1^{(i)} = \texttt{Hash}(\boldsymbol{u}'^{(i)}, \boldsymbol{e}'^{(i)}, Salt, i)$
Set $\mathcal{T} = \texttt{Merkle Tree}(c_0^{(1)}, \cdots, c_0^{(t)})$
Compute $c_0 = \mathcal{T}.\texttt{Root}()$
Compute $c_1 = \texttt{Hash}(c_1^{(1)}, \cdots, c_1^{(t)})$
Generate $(\beta^{(1)}, \cdots, \beta^{(t)}) = \texttt{GenCh}_1(c_0, c_1, Msg, Salt)$
for $i=1,\, i \le t,\, i=i+1$ **do**
    Compute $\boldsymbol{y}^{(i)} = \boldsymbol{u}'^{(i)} + \beta^{(i)}\boldsymbol{e}'^{(i)}$
    Compute $h^{(i)} = \texttt{Hash}(\boldsymbol{y}^{(i)})$
Compute $h = \texttt{Hash}(h^{(1)}, \cdots, h^{(t)})$
Generate $(\boldsymbol{b}[1], \cdots, \boldsymbol{b}[t]) = \texttt{GenCh}_2(c_0, c_1, \beta^{(1)}, \cdots, \beta^{(t)}, h, Msg, Salt)$
Set $J = \{i : \boldsymbol{b}[i] = 1\}$
Set $\boldsymbol{SeedPath} = \texttt{publish\_seeds}(MSeed, Salt, J)$
for $i \notin J$ **do**
    $f^{(i)} := (\boldsymbol{y}^{(i)}, \sigma^{(i)}, c_1^{(i)})$
Compute $\boldsymbol{MerkleProofs} = \mathcal{T}.\texttt{Proofs}(\{1, \cdots, t\} \setminus J)$
Return $\tau = \left\{ Salt,\, c_0,\, c_1,\, h,\, \boldsymbol{SeedPath},\, \boldsymbol{MerkleProofs},\, \left\{ f^{(i)} \right\}_{i \notin J} \right\}$

5 Simulation result

In this section, we discuss the simulation procedure of our fault attack on LESS and CROSS signatures; *i.e.*, we apply our fault assumption inside the LESS and CROSS signature algorithms to imitate the corresponding practical attack scenario. The simulation code is available at GitHub [4].

In the previous Sections 3.6 and 4, we have analyzed the effect of modification of the values $\boldsymbol{x}[i]$ (for LESS) and $\boldsymbol{y}[i]$ (for CROSS) to 0 and 1 respectively. For LESS and CROSS, this can be achieved by stuck at zero/stuck at one or instruction skip fault. So, in the simulation code, we have assumed the values 0 and 1 of the nodes

[4] `https://github.com/s-adhikary/zkfault_simulation`

Algorithm 8 `Recover_Secret_CROSS` (τ, PK)

Input: $\tau = \left\{ Salt,\, c_0,\, c_1,\, h,\, \boldsymbol{SeedPath},\, \boldsymbol{MerkleProofs},\, \left\{ (\boldsymbol{y}^{(i)},\, \sigma^{(i)},\, c_1^{(i)}) \right\}_{i \notin J} \right\}$

Output: The secret vector $\boldsymbol{e}$.

Generate $(\beta^{(1)}, \cdots, \beta^{(t)}) = \mathtt{GenCh}_1(c_0, c_1, Msg, Salt)$
Generate $(\boldsymbol{b}[1], \cdots, \boldsymbol{b}[t]) = \mathtt{GenCh}_2(c_0, c_1, \beta^{(1)}, \cdots, \beta^{(t)}, h, Msg, Salt)$
Set $J = \{i : \boldsymbol{b}[i] = 1\}$
$\boldsymbol{ESEED}[j_1 - 2l], \cdots, \boldsymbol{ESEED}[j_v - 2l] \leftarrow \mathtt{SeedTreeUpdate}(\boldsymbol{seed}[i], Salt, \boldsymbol{b})$
for $i = j_1 - 2l, \cdots, j_v - 2l$: **do**
    if $i \notin J$ **then**
        Sample $(Seed^{(\boldsymbol{u}')}, Seed^{(\boldsymbol{v}')}) \xleftarrow{\boldsymbol{ESEED}[i]} \{0,1\}^{2\lambda}$
        Sample $\boldsymbol{e}'^{(i)} \xleftarrow{Seed^{(\boldsymbol{e}')}} G$
        Compute $\boldsymbol{e} = \sigma^{(i)}(\boldsymbol{e}'^{(i)})$
        return $\boldsymbol{e}$

$\boldsymbol{x}[i]$ and $\boldsymbol{y}[i]$ respectively. After receiving this faulted signature τ, we compute the corresponding secrets of CROSS (secret $\boldsymbol{e}$) and LESS (secret monomial matrices) with the help of the respective algorithms Alg. 6 and Alg. 8.

Note that the attack is valid if we target any $\boldsymbol{x}[i]$ ($\boldsymbol{y}[i]$) for fault injection in LESS (CROSS) signature, where i is an arbitrary but fixed location. But in our simulation code we have fixed the location as $i = 1$. One may change this location and the simulation code accordingly. However, in that case the results in Table. 2 would change according to our result in Section 3.6. We provide the simulation results for all the versions of LESS and CROSS in Table 2. We have run the simulation code multiple times to recover all secrets with (multiple) faulted signatures. We take the average of the number of faulted signatures required to recover all secrets, which we denote with N_{avg}. We have also included the average number of secrets recovered from one single fault ($\mathbb{E}[X]$) in the table.

Scheme	**Security Level**	**Parameter Set**	**Optim. Corner**	**Number of Secrets**	$\mathbb{E}[X]$	$N_{\mathbf{avg}}$
LESS [7]	1	LESS-1b	-	1	1	1
		LESS-1i	-	3	2.91	1.05
		LESS-1s	-	7	5.55	2.09
	3	LESS-3b	-	1	1	1
		LESS-3s	-	2	2	1
	5	LESS-5b	-	1	1	1
		LESS-5s	-	2	2	1
CROSS [34]	1, 3, & 5	CROSS-R-SDP	fast/small	1	1	1
		CROSS-R-SDP(G)	fast/small	1	1	1

Table 2: Simulation result of full secret monomial matrices recovery of LESS and CROSS signature [7,34]

.

Our analysis is based on the fact that each time we query the faulted signature oracle, we get an effectively faulted signature. However, in a practical fault attack, this is not the case. In the real world, there is a probability that an injected fault is successful, say p_1, and also there is a probability that a successfully injected fault is effective, say p_0. Then

$$\begin{aligned} &\Pr[\text{ effective fault} \wedge \text{successful fault }] \\ =&\Pr[\text{ effective fault} \mid \text{successful fault }]\cdot\Pr[\text{ successful fault }] \\ =&p_0p_1 \end{aligned} \tag{5}$$

Let us consider $p=p_0p_1$. Therefore, in a practical scenario to get one faulted signature, the approximate number of queries to the faulted signature oracle needed would be $N_{\text{trial}}=\frac{1}{p}$. Moreover, to get N_{avg} many faulted signatures, we need $N_{\text{total}}=\frac{N_{\text{avg}}}{p}$ many queries. For example if we consider $p=0.01$, then $N_{\text{trial}}=100$ and consequently, $N_{\text{total}}=100\cdot N_{\text{avg}}$.

6 Countermeasures

In the previous section, we have seen that the primary attack surface *Reference Tree* $\boldsymbol{x}$ is initialized by 0. If we inject fault to skip store instruction line-5 of Alg. 5 i.e. $\boldsymbol{x}[i]=\boldsymbol{x}[2i+1]\wedge\boldsymbol{x}[2i+2]$, then $\boldsymbol{x}[i]$ does not change the value and stays 0. Hence, one may suggest initializing the *Reference Tree* with all 1. The instruction skip fault does not work in this case, but we can apply bit-flip fault or stuck-at-zero faults and apply the same attack analysis. Since many practical fault attacks are applicable, countermeasures against one type of fault may not serve our purpose. Therefore, first, we must identify the main reason for the existence of the attack vector.

After the digest computation in Alg. 2, the values of vector $\boldsymbol{d}$ are checked twice. First, by checking whether the value of each $\boldsymbol{d}[i]$ is zero or not, they published the component $\boldsymbol{TreeNode}$. Completing this procedure, again, each $\boldsymbol{d}[i]$ is checked to publish the component $\boldsymbol{rsp}$. Therefore, if an attacker injects a fault at the time of computing $\boldsymbol{TreeNode}$ and somehow succeeds in disclosing the seed $\boldsymbol{ESEED}[i]$ without altering the vector $\boldsymbol{d}$, then the information about the secret matrix $\boldsymbol{Q}^T_{\boldsymbol{d}[i]}$ is susceptible to leakage. To mitigate potential attacks, we must publish either the response $\widetilde{\boldsymbol{Q}}_i$ or $\boldsymbol{Q}^T_{\boldsymbol{d}[i]}\overline{\boldsymbol{Q}}_i$ after a single verification of the value $\boldsymbol{d}[i]$.

In the following sections, we will offer concise explanations for two countermeasures incorporated within the LESS scheme that protect the scheme up to one fault.

6.1 Countermeasure with larger signature size

The most straightforward countermeasure would be not using the tree construction at all. In this version, the preparation of digest $\boldsymbol{d}$ is the same as Alg. 2. After the digest preparation, for each $0\le i<t$ we only check the value of $\boldsymbol{d}[i]$, and set

$$\boldsymbol{rsp}[i]=\begin{cases}\widetilde{\boldsymbol{Q}}_i & \text{if } \boldsymbol{d}[i]=0\\ \boldsymbol{Q}^T_{\boldsymbol{d}[i]}\overline{\boldsymbol{Q}}_i & \text{otherwise}\end{cases}.$$

Then, we cannot get both $\widetilde{\boldsymbol{Q}}_i$ and $\boldsymbol{Q}_{\boldsymbol{d}[i]}^T\overline{\boldsymbol{Q}}_i$ for any i and the attack can be prevented. However, in this case, the size of the signature will be $|cmt|+wk(\lceil\log n\rceil+\lceil\log(q-1)\rceil)+(t-w)\lambda$. This signature size will be larger than the submitted version of LESS [35].

6.2 Countermeasure with same small signature size

Here, we introduce another countermeasure that will keep the signature size the same as the submitted version of LESS [35]. In this countermeasure, our main target is that after computation of the *Reference Tree* $\boldsymbol{x}$, we check the *Reference Tree* only once to compute the response for the signature. Note that, according to the construction of the *Reference Tree* $\boldsymbol{x}$, the path from a leaf node to the root node should be of form 0^y1^z because if the value of any node on this path is 1, then all the ancestors of that node will be 1.

After the preparation of the *Reference Tree* $\boldsymbol{x}$, we modify the signature generation method. First, observe that for any leaf node $\boldsymbol{x}[2l-1+i]$ of the *Reference Tree*, let the path to the root be $\boldsymbol{x}[i_0]\boldsymbol{x}[i_1]\cdots\boldsymbol{x}[i_p]$, where $p=\lceil\log_2(l)\rceil+1$ is the height of the *Reference Tree*. Here, $i_0=2l-1+i$ and $\boldsymbol{x}[i_j]$ is the ancestor of $\boldsymbol{x}[2l-1+i]$ at the height j, this means that $\boldsymbol{x}[i_p]$ is the root. The following is the signature generation process:

- **Step-1:**We start from the leftmost leaf node.
- **Step-2:** check the path $\boldsymbol{x}[i_0]\boldsymbol{x}[i_1]\cdots\boldsymbol{x}[i_p]$
- **Step-3:** If $\boldsymbol{x}[i_j]=1$ for $0\leq j\leq p$, then we store $\boldsymbol{Q}_{\boldsymbol{d}[i]}^T\overline{\boldsymbol{Q}}_i$ in $\boldsymbol{rsp}$ and select the next leaf node as $\boldsymbol{x}[2l+i]$ and goto **Step-2**, else go to **Step-4**.
- **Step-4:** We find $\boldsymbol{x}[i_h]$, which is the highest ancestor of $\boldsymbol{x}[i_0]$ with the value zero.
- **Step-5:** Since $\boldsymbol{x}[i_h]=0$, all the 2^h leaf nodes of the subtree rooted $\boldsymbol{x}[i_h]$ must be zero. We store the seed $\boldsymbol{seed}[i_h]$ in $\boldsymbol{TreeNode}$ and we select the next leaf node as $\boldsymbol{x}[2l-1+i+2^h]$ and go to **Step-2**. If no more leaf nodes are left, then we stop.
- **Step-6:** return the pair $\boldsymbol{rsp}$ and $\boldsymbol{TreeNode}$.

The digest $\boldsymbol{d}$ and the *Reference Tree* are prepared in the same process here as it is prepared in Alg. 2. Only the vectors $\boldsymbol{rsp}$, $\boldsymbol{TreeNode}$ are prepared using Alg. 9. At the end, (cmt, $salt$, $\boldsymbol{rsp}$, ,$\boldsymbol{TreeNode}$) is generated as the signature.

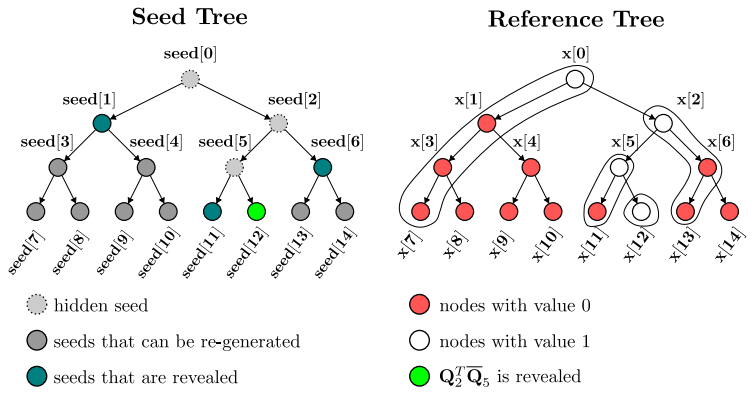

Fig. 5: Example for extraction of response $\boldsymbol{rsp}$, $\boldsymbol{TreeNode}$ according to Alg. 9

Algorithm 9 `LESS_Gen_rsp_update`

Input: The fixed weight digest vector $\boldsymbol{d}$, The secret monomial matrices $\boldsymbol{Q}_i$, $\forall i \in \mathbb{Z}_s$, where $\boldsymbol{Q}_0 = \boldsymbol{I}_n$, The *Seed Tree* $\boldsymbol{seed}$, and the partial monomial matrices $\overline{\boldsymbol{Q}}_j$, $\forall j \in \mathbb{Z}_t$.
Output: The response $\boldsymbol{rsp}$ and $\boldsymbol{TreeNode}$

for $i=0$; $i<t$; $i=i+1$ **do**
  if $\boldsymbol{d}[i]=0$ **then**
    $\boldsymbol{f}[i]=0$
  else
    $\boldsymbol{f}[i]=1$
for $i=0$; $i<4l$; $i=i+1$ **do**
  $\boldsymbol{x}[i]=0$
$\boldsymbol{x} \leftarrow$ `compute_seeds_to_publish`$(\boldsymbol{f}, \boldsymbol{x})$
$i=0$, $j=0$, $j'=0$
while $i<t$ **do**
  $c=2l-1+i$, $h=0$, $h'=0$
  while $Parent(c) \neq 0$ **do**
    if $\boldsymbol{x}[c]=0$ **then**
      $c'=c$, $h'=h+1$
    $c=Parent(c)$, $h=h+1$
  if $h'=0$ **then**
    $\boldsymbol{rsp}[j'] =$ `CompressMono`$(\boldsymbol{Q}_{\boldsymbol{d}[i]}^T \overline{\boldsymbol{Q}}_i)$
    $i=i+1$, $j'=j'+1$
  else
    $\boldsymbol{TreeNode}[j] = \boldsymbol{seed}[c']$
    $i=i+2^{h'-1}$, $j=j+1$
Return $\boldsymbol{rsp}$, $\boldsymbol{TreeNode}$

Example 4. Given a fixed signature digest vector represented as $\boldsymbol{d}=(0, 0, 0, 0, 0, 2, 0, 0)$. First, we construct the *Reference Tree* $\boldsymbol{x}$, which is illustrated in Fig. 5. Begin by checking leaf nodes from the left side. First, we take the leftmost leaf node $\boldsymbol{x}[7]$. The path from $\boldsymbol{x}[7]$ to root is $\boldsymbol{x}[i_0]\boldsymbol{x}[i_1]\boldsymbol{x}[i_2]\boldsymbol{x}[i_3]=\boldsymbol{x}[7]\boldsymbol{x}[3]\boldsymbol{x}[1]\boldsymbol{x}[0]$, and it is valued 0001. In Fig. 5, we can see that the height of the last ancestor valued 0 is $h'=3$, and the node is $\boldsymbol{x}[1]$. We store $\boldsymbol{seed}[1]$ in response $\boldsymbol{TreeNode}$ and select the next leaf node as $\boldsymbol{x}[7+2^{h'-1}]=\boldsymbol{x}[11]$. Final response will be calculated as $\boldsymbol{rsp}=(\boldsymbol{Q}_2^T\overline{\boldsymbol{Q}}_{12})$ and $\boldsymbol{TreeNode}=(\boldsymbol{seed}[1], \boldsymbol{seed}[11], \boldsymbol{seed}[6])$.

Suppose we inject a fault at the node $\boldsymbol{x}[i]$ and alter its value from 1 to 0. Then some of its ancestors may change. Let $\boldsymbol{x}[i_1]$, $\boldsymbol{x}[i_2]$, $\cdots$, $\boldsymbol{x}[i_h]$ be the list of all ancestors of $\boldsymbol{x}[i]$, where $\boldsymbol{x}[i_j]$ is ancestor of $\boldsymbol{x}[i_{j-1}]$ for all $j \in [2, h]$ and $\boldsymbol{x}[i_h]$ is the root. Suppose $\boldsymbol{x}[i_y]$ is the highest ancestor in the list to have the value zero. Now, consider the leftmost leaf node $\boldsymbol{x}[r]$ of the subtree rooted at $\boldsymbol{x}[i_y]$, then $\boldsymbol{x}[i_y]$ is the highest node with value zero in the path from $\boldsymbol{x}[r]$ to root. Hence, according to Alg. 9, $\boldsymbol{seed}[i_y]$ is appended to $\boldsymbol{TreeNode}$ and all the leaf nodes in the subtree rooted at $\boldsymbol{x}[i_y]$ are skipped.

Observe that the fault at $\boldsymbol{x}[i]$ only affects the subtree rooted at $\boldsymbol{x}[i_y]$, the rest of the *Reference Tree* is unchanged. The subtree rooted at $\boldsymbol{x}[i_y]$ is skipped after revealing $\boldsymbol{seed}[i_y]$, and $\boldsymbol{seed}[i_y]$ can only be used to generate the ephemeral seeds

that do not have any information about the secret monomial matrices. Therefore, the attack will not be possible with just one fault.

We only change the attack surface part to protect the LESS scheme against our attack. The attack surface of the CROSS signature scheme is similar to LESS. We can use the proposed countermeasure for CROSS also. We only need to modify the update method of $\boldsymbol{rsp}$ and $\boldsymbol{TreeNode}$ according to the CROSS signing algorithm.

Cost of the countermeasure Here we will compare the cost analysis of our proposed countermeasure with the original LESS implementation (Alg. 2). The *Reference Tree* generation process in our proposed method is the same as the original LESS proposal. We have only changed the $\boldsymbol{TreeNode}$ and $\boldsymbol{rsp}$ generation process but result is same in both cases *i.e*, for a particular *Seed Tree*, *Reference Tree* pair, our method and original LESS implementation, both generate the same $\boldsymbol{TreeNode}$ and $\boldsymbol{rsp}$. First of all, we consider the following computation costs:

- C_{check}: cost of any condition checking
- C_{mono}: cost of computation of a monomial multiplication followed by a `CompressMono` function and storing the result
- C_{seed}: cost of storing seeds from *Seed Tree* condition checking

Also, we fix a *Seed Tree* and a *Reference Tree* and we assume that r many seeds from *Seed Tree* are to be stored in $\boldsymbol{TreeNode}$. Assume that the total number of nodes in the *Reference Tree* is N.

Cost of LESS original implementation (Alg. 2): As we can see in Alg. 2, the $\boldsymbol{TreeNode}$ is generated using Alg. 4. We are going to ignore the cost of `compute_seeds_to_publish` function (Alg. 5), as it has also been used in our countermeasure. For each node in the *Reference Tree*, Alg. 4 checks the node and its parent node which takes $2N \cdot C_{\text{check}}$ computations. As we have assumed earlier there are r many seeds which are to be stored in $\boldsymbol{TreeNode}$, which takes $r \cdot C_{\text{seed}}$ computations. After that, Alg. 2 checks each value of the vector $\boldsymbol{d}$ which takes $t \cdot C_{check}$ computations and computes the monomial multiplication and calls the `CompressMono` function for each $\boldsymbol{d}[i] \neq 0$, which takes $w \cdot C_{\text{mono}}$ computations. Therefore the total cost is $(2N+t) \cdot C_{\text{check}} + r \cdot C_{\text{seed}} + w \cdot C_{\text{mono}}$.

Cost of our proposed method (Alg. 9): In each iteration of the while loop, we first check the full path from the leaf node to the root, this takes $\log_2 N \cdot C_{\text{check}}$ computations. In each iteration, the algorithm can either update $\boldsymbol{TreeNode}$ or update $\boldsymbol{rsp}$ *i.e.*, the total number of iterations in the while loop is $(r+w)$. The condition checking takes total $(r+w) \cdot \log_2 N \cdot C_{\text{check}}$ computations. Now the $\boldsymbol{rsp}$ is updated for w many iterations, which takes total $w \cdot C_{\text{mono}}$ computations. Similarly, updating $\boldsymbol{TreeNode}$ takes $r \cdot C_{\text{seed}}$ computations. Therefore, the total cost is $(r+w) \cdot \log_2 N \cdot C_{\text{check}} + r \cdot C_{\text{seed}} + w \cdot C_{\text{mono}}$.

Observe that the cost of the countermeasure may vary with the value of r, which is the number of seeds published from *Seed Tree*. We have benchmarked the performance

of the LESS-sign algorithm with and without our countermeasure for all parameter sets of LESS in Table 3. As we can see, including our countermeasure does not degrade the performance of the LESS-sign algorithm.

Security Level	**Parameter Set**	**Average cpucycles ($\times 10^6$ cycles)**	
		Our Countermeasure	**Original LESS**
1	LESS-1b	1162.31	1162.19
	LESS-1i	1148.03	1147.94
	LESS-1s	931.57	931.72
3	LESS-3b	9563.01	9564.23
	LESS-3s	11285.14	11283.65
5	LESS-5b	44031.11	44036.56
	LESS-5s	29544.22	29542.31

Table 3: LESS-sign performance comparison with our countermeasure against original LESS implementation

.

For benchmarking, we have used an HP Elite Tower 600 G9 Desktop with an Intel Core i7-12700 CPU running at 2.1 GHz and 32 GB physical memory, which was running Ubuntu 22.04.4 LTS. The test codes were executed on a single core with Turbo Boost and hyperthreading disabled.

7 Discussion and future direction

In this study, we have assumed a single-fault model where an attacker can only inject a fault in one single location. The countermeasure we have provided is based on that assumption. We emphasize the necessity of future investigations into higher-order fault models, side-channel attacks using power, electromagnetic radiation [20,28], and combined (side-channel assisted fault attack) attack. This study is the first research study enhancing the security of the digital signature scheme LESS and CROSS against a broader spectrum of fault attacks. LESS has $(s-1)$ secret monomial matrices, and we've shown that one pair can recover some information about one secret matrix. So, we need multiple targeted pairs to retrieve all secret matrices. This number of required pairs depends on various parameters. Therefore, we require more than one effective faulted signature for some parameter sets of LESS. For CROSS, there's only one secret $\boldsymbol{e}$ for all the parameter sets. It can be recovered with just one targeted pair.

In this work, we have done a fault analysis of the LESS [35] signature scheme that has been submitted to NIST. However, the authors of LESS have updated the scheme in the LESS project's site [21]. We observe that our mentioned attack surface, *i.e.*, the computation of $\boldsymbol{TreeNode}$ by using the function `SeedTreePaths`, are present there too. So, our attack is still applicable to their updated version. Another code-based

signature scheme MEDS (Matrix Equivalence Digital Signature) [12] based on the zero-knowledge protocol. Like LESS and CROSS, the Sign algorithm of MEDS uses a similar tree construction to reduce the signature size. In this case, the response ($\tilde{\boldsymbol{A}}_i$ or $\boldsymbol{Q}\cdot\tilde{\boldsymbol{A}}_i$) is constructed depending on some fixed weight digest vector $\boldsymbol{d}$, where $\boldsymbol{Q}$ is a secret component. It involves the same seed tree and *Reference Tree* to store some seeds corresponding to the response $\tilde{\boldsymbol{A}}_i$ in a similar manner. So, the same attack model can also be applied to the MEDS signature scheme. However, we have not completely analyzed how many faulted signatures are needed to find the entire secret. We left this part for future work.

We have shown a fault detection method where we have fixed a position $\boldsymbol{x}[i]$ of the `Reference Tree` and injected fault at that location. The detection method in Section 3.4 can detect a successful and effective fault at the location $\boldsymbol{x}[i]$ for any chosen i, where $1 \leq i < 4l-1$. Moreover, this method can determine the occurrence of an effective fault at any arbitrary location within the reference tree by applying the detection procedure for each $1 \leq i < 4l-1$. Given that $l \in \{128, 512, 1024\}$ (according to Table 1), this approach is computationally feasible. However, a mathematical analysis for this scenario has not been included and is left for future work.

Acknowledgements

This work was partially supported by Horizon 2020 ERC Advanced Grant (101020005 Belfort), CyberSecurity Research Flanders with reference number VR20192203, BE QCI: Belgian-QCI (3E230370) (see beqci.eu), Intel Corporation, Secure Implementation of Post-Quantum Cryptosystems (SECPQC) DST-India and BELSPO. Angshuman Karmakar is funded by FWO (Research Foundation – Flanders) as a junior post-doctoral fellow (contract number 203056 / 1241722N LV). Puja Mondal is supported by C3iHub, IIT Kanpur.

References

1. Abdalla, M., An, J.H., Bellare, M., Namprempre, C.: From Identification to Signatures via the Fiat-Shamir Transform: Minimizing Assumptions for Security and Forward-Security. In: Knudsen, L.R. (ed.) Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2332, pp. 418–433. Springer (2002). https://doi.org/10.1007/3-540-46035-7_28, https://doi.org/10.1007/3-540-46035-7_28
2. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Liu, Y.K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D.: Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. Online. Accessed 26th January, 2024 (2022), https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413-upd1.pdf
3. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehle, D.: CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation (version 3.02). Online (2021), https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf
4. Beckwith, L., Wallace, R., Mohajerani, K., Gaj, K.: A High-Performance Hardware Implementation of the LESS Digital Signature Scheme. In: Johansson, T., Smith-Tone, D. (eds.) Post-Quantum Cryptography - 14th International Workshop, PQCrypto 2023, College Park, MD, USA, August 16-18, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14154, pp. 57–90. Springer (2023). https://doi.org/10.1007/978-3-031-40003-2_3, https://doi.org/10.1007/978-3-031-40003-2_3
5. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS+ Signature Framework. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 2129–2146. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3363229, https://doi.org/10.1145/3319535.3363229
6. Beullens, W.: Breaking Rainbow Takes a Weekend on a Laptop. Cryptology ePrint Archive, Paper 2022/214 (2022), https://eprint.iacr.org/2022/214, https://eprint.iacr.org/2022/214
7. Biasse, J.F., Micheli, G., Persichetti, E., Santini, P.: LESS is More: Code-Based Signatures Without Syndromes. In: Nitaj, A., Youssef, A. (eds.) Progress in Cryptology - AFRICACRYPT 2020. pp. 45–65. Springer International Publishing, Cham (2020)
8. Breier, J., Hou, X.: How Practical are Fault Injection Attacks, Really? Cryptology ePrint Archive, Paper 2022/301 (2022), https://eprint.iacr.org/2022/301, https://eprint.iacr.org/2022/301
9. Bruinderink, L.G., Pessl, P.: Differential Fault Attacks on Deterministic Lattice Signatures. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(3), 21–43 (2018). https://doi.org/10.13154/TCHES.V2018.I3.21-43, https://doi.org/10.13154/tches.v2018.i3.21-43
10. Castryck, W., Decru, T.: An Efficient Key Recovery Attack on SIDH. In: Advances in Cryptology – EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part V. p. 423–447. Springer-Verlag, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-30589-4_15, https://doi.org/10.1007/978-3-031-30589-4_15
11. Cho, J., No, J.S., Lee, Y., Koo, Z., Kim, Y.S.: Enhanced pqsigRM: Code-Based Digital Signature Scheme with Short Signature and Fast Verification for Post-

Quantum Cryptography. Cryptology ePrint Archive, Paper 2022/1493 (2022), https://eprint.iacr.org/2022/1493, https://eprint.iacr.org/2022/1493
12. Chou, T., Niederhagen, R., Persichetti, E., Randrianarisoa, T.H., Reijnders, K., Samardjiska, S., Trimoska, M.: Take your MEDS: Digital Signatures from Matrix Code Equivalence. Cryptology ePrint Archive, Paper 2022/1559 (2022), https://eprint.iacr.org/2022/1559, https://eprint.iacr.org/2022/1559
13. Clavier, C.: Secret External Encodings Do Not Prevent Transient Fault Analysis. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4727, pp. 181–194. Springer (2007). https://doi.org/10.1007/978-3-540-74735-2_13, https://doi.org/10.1007/978-3-540-74735-2_13
14. Ding, J., Schmidt, D.: Rainbow, a New Multivariable Polynomial Signature Scheme. In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) Applied Cryptography and Network Security. pp. 164–175. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
15. Ducas, L., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehle, D.: CRYSTALS – Dilithium: Digital Signatures from Module Lattices. Cryptology ePrint Archive, Paper 2017/633 (2017), https://eprint.iacr.org/2017/633, https://eprint.iacr.org/2017/633
16. Galbraith, S.D., Petit, C., Silva, J.: Identification Protocols and Signature Schemes Based on Supersingular Isogeny Problems. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10624, pp. 3–33. Springer (2017). https://doi.org/10.1007/978-3-319-70694-8_1, https://doi.org/10.1007/978-3-319-70694-8_1
17. Genêt, A., Kannwischer, M.J., Pelletier, H., McLauchlan, A.: Practical Fault Injection Attacks on SPHINCS. IACR Cryptol. ePrint Arch. p. 674 (2018), https://eprint.iacr.org/2018/674
18. Groot Bruinderink, L., Hülsing, A., Lange, T., Yarom, Y.: Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. In: Gierlichs, B., Poschmann, A.Y. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2016. pp. 323–345. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
19. Jao, D., Feo, L.D.: Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In: PQCrypto. Lecture Notes in Computer Science, vol. 7071, pp. 19–34. Springer (2011)
20. Kundu, S., Chowdhury, S., Saha, S., Karmakar, A., Mukhopadhyay, D., Verbauwhede, I.: Carry Your Fault: A Fault Propagation Attack on Side-Channel Protected LWE-based KEM. IACR Cryptol. ePrint Arch. p. 1674 (2023), https://eprint.iacr.org/2023/1674
21. LESSProjectSite: LESS project (2023), https://www.less-project.com/
22. Meyer, C.: Matrix Analysis and Applied Linear Algebra. Other Titles in Applied Mathematics, Society for Industrial and Applied Mathematics (2000), https://books.google.co.in/books?id=HoNgdpJWnWMC
23. Miller, V.S.: Use of Elliptic Curves in Cryptography. In: Williams, H.C. (ed.) Advances in Cryptology — CRYPTO '85 Proceedings. pp. 417–426. Springer Berlin Heidelberg, Berlin, Heidelberg (1986)
24. Mondal, P., Kundu, S., Bhattacharya, S., Karmakar, A., Verbauwhede, I.: A practical key-recovery attack on LWE-based key-encapsulation mechanism schemes using Rowhammer. CoRR **abs/2311.08027** (2023). https://doi.org/10.48550/ARXIV.2311.08027, https://doi.org/10.48550/arXiv.2311.08027

25. NIST: NIST Announces Additional Digital Signature Candidates for the PQC Standardization Process. Online. Accessed 26th January, 2024 (2023), https://csrc.nist.gov/news/2023/additional-pqc-digital-signature-candidates
26. Oder, T., Schneider, T., Pöppelmann, T., Güneysu, T.: Practical CCA2-Secure and Masked Ring-LWE Implementation. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(1), 142–174 (2018). https://doi.org/10.13154/TCHES.V2018.I1.142-174, https://doi.org/10.13154/tches.v2018.i1.142-174
27. Persichetti, E., Santini, P.: A New Formulation of the Linear Equivalence Problem and Shorter LESS Signatures, pp. 351–378 (12 2023). https://doi.org/10.1007/978-981-99-8739-9_12
28. Pessl, P., Prokop, L.: Fault Attacks on CCA-secure Lattice KEMs. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(2), 37–60 (2021). https://doi.org/10.46586/TCHES.V2021.I2.37-60, https://doi.org/10.46586/tches.v2021.i2.37-60
29. Poddebniak, D., Somorovsky, J., Schinzel, S., Lochter, M., Rösler, P.: Attacking Deterministic Signature Schemes using Fault Attacks. Cryptology ePrint Archive, Paper 2017/1014 (2017), https://eprint.iacr.org/2017/1014, https://eprint.iacr.org/2017/1014
30. Proos, J., Zalka, C.: Shor's discrete logarithm quantum algorithm for elliptic curves. Quantum Inf. Comput. **3**(4), 317–344 (2003). https://doi.org/10.26421/QIC3.4-3, https://doi.org/10.26421/QIC3.4-3
31. Qiao, R., Seaborn, M.: A new approach for rowhammer attacks. In: 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 161–166 (2016). https://doi.org/10.1109/HST.2016.7495576
32. Rennie, B., Dobson, A.: On stirling numbers of the second kind. Journal of Combinatorial Theory **7**(2), 116–121 (1969). https://doi.org/https://doi.org/10.1016/S0021-9800(69)80045-1, https://www.sciencedirect.com/science/article/pii/S0021980069800451
33. Rivest, R.L., Shamir, A., Adleman, L.M.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Commun. ACM **21**(2), 120–126 (1978). https://doi.org/10.1145/359340.359342, http://doi.acm.org/10.1145/359340.359342
34. Schemes, N.P.Q.C.D.S.: CROSS: Codes and Restricted Objects Signature Scheme - Specification Document (Jan 2022), https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/CROSS-spec-web.pdf
35. Schemes, N.P.Q.C.D.S.: Less: Linear equivalence signature scheme - Specification Document (Jan 2022), https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/less-spec-web.pdf
36. Schemes, N.P.Q.C.D.S.: WAVE: Round 1 Submission - Specification Document (Jan 2022), https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/wave-spec-web.pdf
37. Shor, P.W.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994. pp. 124–134. IEEE Computer Society (1994). https://doi.org/10.1109/SFCS.1994.365700, https://doi.org/10.1109/SFCS.1994.365700
38. Sullivan, G.A., Sippe, J., Heninger, N., Wustrow, E.: Open to a fault: On the passive compromise of TLS keys via transient errors. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 233–250. USENIX Association, Boston, MA (Aug 2022), https://www.usenix.org/conference/usenixsecurity22/presentation/sullivan
39. Xagawa, K., Ito, A., Ueno, R., Takahashi, J., Homma, N.: Fault-Injection Attacks Against NIST's Post-Quantum Cryptography Round 3 KEM Candidates. In: Tibouchi, M., Wang, H. (eds.) Advances in Cryptology – ASIACRYPT 2021. pp. 33–61. Springer International Publishing, Cham (2021)

Supplementary material

A Comparison of LESS with other code-based signature schemes

Table 4 compares the key sizes and performance of LESS with other code-based digital signature schemes submitted to NIST's additional call for digital signatures [25].

Category	Scheme	Performance (M Cycle)		Size (Bytes)	
		Sign.	Verify	Signature	Public Key
Level I	WAVE [36]	1161	205.9	822	3677390
	MEDS [12]	518.1	515.6	9896	9923
	CROSS [34]	22	10.3	10304	61
	LESS [35]	263.6	271.4	5325	98202
Level III	WAVE [36]	3507	464.1	1249	7867598
	MEDS [12]	1467	1462	41080	41711
	CROSS [34]	46.5	18.3	23407	91
	LESS [35]	2446.9	2521.4	14438	70554
Level V	WAVE [36]	7397	813.3	1644	13632308
	MEDS [12]	1629.8	1612.6	132528	134180
	CROSS [34]	74.8	26.1	43373	121
	LESS [35]	10212.6	10458.8	26726	132096

Table 4: Comparison of code-based signature schemes in terms of performance and size.

B Verification algorithm of LESS

The verification algorithm in Alg. 10 takes a message m and signature τ and the public key PK as inputs and returns `1`, if the τ is a valid signature of the message m otherwise, it will return `0`.

Algorithm 10 LESS_Vrfy(m, τ, PK)

Input: A message m, the public key PK and the signature $\tau = (salt, cmt, \boldsymbol{TreeNode}, \boldsymbol{rsp})$.
Output: It will return 1, if (m, τ) is a valid message signature pair. Otherwise, it will return 0.

$\boldsymbol{d'} \leftarrow$ CSPRNG(cmt, $\mathbb{S}_{t,w}$)
for $i=0$; $i<t$; $i=i+1$ **do**
  if $\boldsymbol{d'}[i]=0$ **then**
    $\boldsymbol{f'}[i]=0$
  else
    $\boldsymbol{f'}[i]=1$
$\boldsymbol{ESEED'} \leftarrow$ regenerate_leaves($salt$, $\boldsymbol{TreeNode}$, $\boldsymbol{f'}$)
$\boldsymbol{G}_0 \leftarrow$ CSPRNG($gseed$, $\mathbb{S}_{\text{RREF}}$)
$k=0$
for $i=1$; $i<t$; $i=i+1$ **do**
  if $\mathbf{d}'[i]=0$ **then**
    $\widetilde{\boldsymbol{Q}}'_i \leftarrow$ CSPRNG($\boldsymbol{ESEED'}[i]$, $\boldsymbol{M}_n$)
    $(\overline{\boldsymbol{Q}}'_i, \overline{\boldsymbol{V}}'_i) \leftarrow$ PreparedDigestInput($\boldsymbol{G}_0$, $\widetilde{\boldsymbol{Q}}'_i$)
  else
    $j = \boldsymbol{d'}[i]$
    $\boldsymbol{G}_j \leftarrow$ ExpandRREF(PK[j])
    $\boldsymbol{Q}^* \leftarrow$ ExpandToMonomAction($\boldsymbol{rsp}[k]$)
    Compute $J \leftarrow \{\alpha_i : \boldsymbol{Q}^*[\alpha_i, *]=0\}$
    $\widehat{\boldsymbol{G}} \leftarrow (\boldsymbol{G}_j\boldsymbol{Q}^* \mid \boldsymbol{G}_j[*, J])$
    $(\widehat{\boldsymbol{G}}, pivot\_column) \leftarrow$ RREF($\widehat{\boldsymbol{G}}$)
    $NP=0$, $\overline{\boldsymbol{V}}'_i = \boldsymbol{O}$
    for $c=0$; $c<n$; $c=c+1$ **do**
      if $pivot\_column[c]=0$ **then**
        $\overline{\boldsymbol{V}}'_i \leftarrow$ LexMin($\widehat{G}$, $\overline{\boldsymbol{V}}'_i$, NP, c)
        $NP=NP+1$
    $\overline{\boldsymbol{V}}'_i \leftarrow$ LexSortCol($\overline{\boldsymbol{V}}'_i$)
    $k=k+1$
$cmt' \leftarrow$ H($\overline{\boldsymbol{V}}'_0$, $\cdots$, $\overline{\boldsymbol{V}}'_{t-1}$, m, len, $salt$)
if $cmt = cmt'$ **then**
  Return 1
else
  Return 0