IACR Communications in Cryptology
ISSN 3006-5496, Vol. 1, No. 4, 25 pages.

https://doi.org/10.62056/ab0ljbkrz

MAYO Key Recovery by Fixing Vinegar Seeds

Sönke Jendral and Elena Dubrova

KTH Royal Institute of Technology, Stockholm, Sweden

Abstract. As the industry prepares for the transition to post-quantum secure public key cryptographic algorithms, vulnerability analysis of their implementations is gaining importance. A theoretically secure cryptographic algorithm should also be able to withstand the challenges of physical attacks in real-world environments. MAYO is a candidate in the ongoing second round of the NIST post-quantum standardization process for selecting additional digital signature schemes. This paper demonstrates three first-order single-execution fault injection attacks on the official MAYO implementation on the ARM Cortex-M4. By using voltage glitching to disrupt the computation of the vinegar seed during the signature generation, we enable the recovery of the secret key directly from the faulty signatures. Our experimental results show that the success rates of the fault attacks in a single execution are 36%, 82%, and 99%, respectively. They emphasize the importance of developing countermeasures against fault attacks prior to the widespread deployment of post-quantum algorithms like MAYO.

Keywords: Fault injection · MAYO · Multivariate cryptography · Post-quantum digital signature · Key recovery attack

1 Introduction

The National Institute of Standards and Technology (NIST) recently concluded its competition for Post-Quantum Cryptographic (PQC) algorithms, resulting in the publication of standards for key encapsulation mechanism ML-KEM [Nat24b], and digital signature algorithms ML-DSA [Nat24a] and SLH-DSA [Nat24c]. To strengthen security through diversification and broaden the range of use cases for PQC signatures, NIST launched a second competition in 2022. The goal is to identify additional general-purpose PQC signature algorithms based on different underlying mathematical problems than ML-DSA and SLH-DSA, offering other key and signature sizes, and providing varied key generation, signing or verification performance [Nat23]. MAYO is one of the submissions selected by NIST as a second-round candidate in this competition. It is a multivariate quadratic digital signature scheme designed to be existentially unforgeable under chosen message attacks (EUF-CMA) in the random oracle model [Beu22]. EUF-CMA security means that an adversary with access to the public key and a signing oracle cannot generate a valid signature for a new message. The security of MAYO relies on the presumed hardness of the *Oil and Vinegar* (OV) problem and a variant of the *Multivariate Quadratic* (MQ) problem called the *multi-target whipped MQ* problem.

However, a theoretically secure cryptographic algorithm should also be able to withstand the challenges of physical attacks in real-world environments. Yet, numerous successful side-channel and fault attacks on implementations of PQC algorithms demonstrated over the past few years [PPM17, RRB$^+$19, GJN20] indicate that this is not always the case. It is important to identify which types of physical attacks are most relevant in real-world

E-mail: jendral@kth.se (Sönke Jendral), dubrova@kth.se (Elena Dubrova)

Received: 2024-10-04 Accepted: 2024-12-03

scenarios to focus efforts on designing effective and targeted countermeasures prior to the widespread deployment of PQC algorithms.

The idea behind multivariate signature schemes is to select a multivariate map $\mathcal{P}$ consisting of several multivariate polynomials as the public key and a specific preimage $\mathbf{s}$ with $\mathcal{P}(\mathbf{s}) = \mathbf{t}$, where $\mathbf{t}$ is a (salted) hash of the message, as the signature. As the secret key, OV-based schemes then use a trapdoor (a vector space $\mathbf{O}$, on which $\mathcal{P}$ vanishes, which is called the *oil space*), with which it is possible to find $\mathbf{s}$ efficiently. Concretely, finding $\mathbf{s}$ is done by first selecting at random a vector $\mathbf{v}$ of so-called *vinegar values* and then solving for a so-called *oil vector* $\mathbf{o} \in \mathbf{O}$ such that $\mathcal{P}(\mathbf{v} + \mathbf{o}) = \mathbf{t}$. It is generally assumed that finding $\mathbf{s}$ without knowledge of the trapdoor is difficult [BCD+23], therefore the scheme is considered secure. Multivariate schemes differ in how $\mathcal{P}$ is constructed. In general, $\mathcal{P}$ is chosen as $\mathcal{P} = \mathcal{S} \circ \mathcal{F} \circ \mathcal{T}$, where $\mathcal{F}$ is called the *central map* and $\mathcal{S}$ and $\mathcal{T}$ are affine invertible maps, though it has been shown that for Unbalanced Oil and Vinegar (UOV) (and derived schemes, such as MAYO), only the central map $\mathcal{F}$ and an affine invertible map $\mathcal{T}$ are required [BPB10]. Newer descriptions of UOV, such as the current specification [BCD+23], as well as the specification of MAYO [BCC+23], do not consider the maps $\mathcal{F}$ and $\mathcal{T}$ explicitly and instead compute $\mathcal{P}$ directly. MAYO additionally performs an optimisation that uses a larger map $\mathcal{P}^*$ that is created by “whipping up” a smaller map $\mathcal{P}$.

Previous attacks [HTS11, KL19, SK20, AKKM22, AMSU24] on multivariate signature schemes have shown that fault injections targeting the vinegar values can lead to recovery of a vector in the oil space and thereby to recovery of the full secret key. However, these attacks require multiple signatures or are limited to specific memory allocation strategies that are not always realistic. Furthermore, while it is known that setting the vinegar values to specific known values using fault injection can lead to key recovery, the techniques to accomplish this have not been exhaustively explored and depend on the target implementation. Therefore the applicability of previous attacks, particularly those conducted on other schemes, is not immediately obvious.

Contributions: In this paper, we present three first-order single-execution fault injection attacks on the official implementation of MAYO on the ARM Cortex-M4 processor [BCC+24]. All three attacks reveal the secret vinegar values by fixing the seed from which they are derived to a known value. The first attack fixes the seed to a constant by skipping the absorption phase during the computation of the seed, as in the attack on CRYSTALS-Dilithium in [JMD24]. The second attack aborts a loop during the absorption phase, thereby allowing the seed to be derived from public information. Both attacks are not limited to a specific memory allocation strategy and instead rely only on functionality dictated by the specification. The third attack skips the initialisation of one of the arguments for the computation of the seed, similarly allowing the seed to be derived from public information. This attack is limited to a less restrictive memory allocation strategy than previous attacks. We identified settings that consistently skip the necessary instructions without crashing the device or disrupting other steps of the signature generation.

All three attacks enable the recovery of the full secret key from a single faulty signature with probabilities of 82%, 36%, and 99%, respectively, using a trivial key recovery method.

We additionally propose a technique for classifying the results of a symbolic execution-based simulation that is able to identify potential target instructions for fault injection. Our approach uses loopy belief propagation on a factor graph to estimate per-bit probabilities of states. It allows us to identify frequently reachable states where the search space for the sponge contents is small. This approach was successfully used to identify the second attack in this paper and can also be used to find similar vulnerabilities in other implementations and algorithms. Finally, we propose countermeasures against the presented attacks.

Table 1: Comparison to previous fault attacks on multivariate signature schemes.

	Algorithm	#Signatures	#Faults	Evaluation[a]	Limitations[b]
Hashimoto et al. [HTS11]	Multiple	Multiple	Multiple	Theoretical	None
Krämer and Loiero [KL19]	UOV/ Rainbow	Multiple	Multiple	Theoretical	None
Shim and Koo [SK20]	UOV	44–103	Multiple	Theoretical	None
Mus et al. [MIS20]	LUOV	Multiple	Multiple	Practical	Key in $\mathbb{F}_2$ (not applicable to MAYO)
Aulbach et al. [AKKM22]	Rainbow	Multiple	1	Simulation	Exact memory reuse
Furue et al. [FKNT22]	UOV	Multiple	2–40	Simulation	Enumeration 2^{41}–2^{89}
Sayari et al. [SMA+24]	MAYO	2	1	Theoretical	Exact memory reuse
		2	1		Deterministic
Aulbach et al. [AMSU24]	MAYO	1	1	Practical	Zero-initialisation
		2	1		Exact memory reuse
This work	MAYO	1	1	Practical, Simulation	None
		1	1		None
		1	1		Similar memory location

[a]Indicates whether the attack is evaluated theoretically, simulated, or performed in practice.
[b]Indicates limitations of the attack regarding which algorithm and variant can be attacked, how complex the key recovery is, and which memory allocation strategy should be used for the attack to succeed.

Organisation of the paper: The rest of this paper is organised as follows. Section 2 describes previous work. Section 3 provides background information on the MAYO algorithm and voltage fault injection. Section 4 describes the simulation and classification technique. Section 5 presents the experimental setup. Section 6 describes the fault attacks. Section 7 introduces the secret key recovery method. Section 8 summarises the experimental results. Section 9 discusses possible countermeasures against the attacks. Section 10 concludes the paper.

2 Previous work

This section gives an overview of previous attacks on multivariate signature schemes, including MAYO, which make use of fault injection or side-channel analysis to recover the secret key. Table 1 provides a summary.

Hashimoto et al. [HTS11] presented two general fault attacks applicable to a number of multivariate schemes. Their first attack changes single coefficients in the central map through a fault. By decrypting random messages under the faulty map and reencrypting them under the original map, they are able to extract information about a part of the secret key from the differences. Their second attack targets the random values used in the signing process. By fixing these values to a constant using a fault, they are able to combine information from several faulty signatures and thereby reduce the complexity of the Kipnis-Shamir attack for recovering a part of the secret key. Krämer and Loiero [KL19] reevaluated these attacks in the context of UOV and Rainbow and found that the first attack is not applicable to schemes that omit one of the affine maps, such as UOV (and MAYO). They also propose additional countermeasures for the second attack. Shim and Koo [SK20] extended the second attack to achieve full key recovery from UOV with between 44 and 103 faulty signatures (depending on the fault model). The attack is not validated experimentally.

Mus et al. [MIS20] showed a Rowhammer-based bit flipping attack on LUOV. Their attack works by recovering a number of bits of the secret key by flipping individual bits and observing the resulting faulty signatures. By combining the partial knowledge of the key with an algebraic approach, they are able to recover all 11,229 bits of the key from 4116 bits obtained by bit flipping in 3hrs 49min and 49hrs of additional post-processing. They do not state the number of signing operations that were performed by the target device in the 3hrs 49min timeframe. As pointed out in [FKNT22], this attack is not applicable to UOV (or MAYO), as the secret key is not in a finite field of two elements.

Aulbach et al. [AKKM22] presented two practical fault attacks on Rainbow. The first one uses the same approach as [SK20] of fixing the vinegar values to reuse them across iterations, but applies a more efficient postprocessing technique. The second attack skips the linear transformation, thereby allowing it to be recovered through multiple faulty signatures. By applying the Kipnis-Shamir attack, they are able to recover the full secret key. They experimentally verify their results using simulation, but do not state the number of signatures required for the attacks.

Furue et al. [FKNT22] introduced a novel fault attack on UOV. Their attack works by injecting faults into parts of the secret key. By observing faulty signatures generated from the changed secret key, they are able to construct a reduced UOV instance, which can be attacked with lower complexity using either the Kipnis-Shamir attack [KS98] or an intersection attack [Beu21, Beu22]. They simulate their attack and find that the full secret key could be recovered with 2 to 40 faults and 2^{41} to 2^{89} enumerations with probabilities between 30% to 80%

Sayari et al. [SMA$^+$24] addressed two fault injection attacks in their hardware implementation of MAYO. The first of these concerns the reuse of vinegar values by skipping their sampling through fault injection. The difference between the original signature and a faulty signature can potentially be used to reveal a vector in the oil space and thus recover the secret key. They propose to shuffle the vinegar values after the signing procedure to prevent reuse. The second attack concerns skipping the addition of the oil values at the end of the signing procedure through fault injection, thereby revealing a vinegar value. If the deterministic signing mode is used, this vinegar value can be used to recover an oil vector and thus the secret key from the difference between the original signature and the faulty signature. As a countermeasure, they propose to check the validity of the signature, as the fault injection causes the signature to be invalid.

Recently, Aulbach et al. [AMSU24] presented two variants of a loop-abort fault injection attack on MAYO similar to the first attack by Sayari et al. [SMA$^+$24]. The idea is again to abort the loop that is used to sample the vinegar values, thus leaving some of the values uninitialised. Under the assumption that the vinegar values are initially set to a constant value or are reused across multiple invocations of the signing procedure, they are able to recover a vector in the oil space, and thereby the key of the scheme, either directly or from the difference of two signatures. They experimentally validated the attack using clock

Table 2: MAYO parameter sets from [BCC+23].

Parameter set	n	m	o	k	q	salt_len	digest_len	pk_seed_len	$f(z)$
MAYO_1	66	64	8	9	16	24b	32b	16b	$f_{64}(z)$
MAYO_2	78	64	18	4	16	24b	32b	16b	$f_{64}(z)$
MAYO_3	99	96	10	11	16	32b	48b	16b	$f_{96}(z)$
MAYO_5	133	128	12	12	16	40b	64b	16b	$f_{128}(z)$

glitching, but do not report a fault probability. Both attacks require only a single fault and one respective two signatures.

Aulbach et al. [ACK+23] also presented an attack making use of side-channel analysis. They exploit leakage during the multiplication of the vinegar values with known constants and are able to recover all vinegar values using a template-based attack. Using the vinegar values, they recover both a vector in the oil space and the full oil space $\mathbf{O}$. The latter is recovered using a combination of the Kipnis-Shamir attack [KS98] and the reconciliation attack [DYC+08]. They experimentally validate their attack on an STM32F303RCT7 processor and recover the full key from a single trace with a probability greater than 97%.

3 Background

This section describes the MAYO algorithm and the voltage fault injection method used in our experiments.

3.1 MAYO algorithm

MAYO is a multivariate quadratic digital signature scheme introduced by Beullens [Beu22]. It is based on the *Oil and Vinegar* (OV) signature scheme originally introduced by Patarin [Pat97] and is considered secure in the random oracle model based on the assumed hardness of the OV and *multi-target whipped Multivariate Quadratic* (MQ) problems. In OV schemes, the public key is a multivariate map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ of m n-variate quadratic polynomials $p_1(x), \ldots, p_m(x)$ over a finite field $\mathbb{F}_q$. The map features a trapdoor, which is a secret subspace $\mathbf{O}$ on which the map vanishes. Using the trapdoor, it is possible to efficiently find a preimage $\mathbf{s}$ of a hash $\mathbf{t}$ such that $\mathcal{P}(\mathbf{s}) = \mathbf{t}$ by selecting a vector $\mathbf{v} \in \mathbb{F}_q^n$ at random and then solving for a vector $\mathbf{o} \in \mathbf{O}$ with

$$\mathcal{P}(\mathbf{v}+\mathbf{o}) = \underbrace{\mathcal{P}(\mathbf{v})}_{\text{constant}} + \underbrace{\mathcal{P}(\mathbf{o})}_{=0} + \underbrace{\mathcal{P}'(\mathbf{v},\mathbf{o})}_{\text{linear in } \mathbf{o}} = \mathbf{t}$$

where $\mathcal{P}'$ is the polar form of $\mathcal{P}$, see [BCC+23]. The signature can then be computed as $\mathbf{s} = \mathbf{v} + \mathbf{o}$. Without knowledge of the trapdoor, finding a preimage is assumed to be difficult, which is known as the MQ problem. Distinguishing a map with such a trapdoor from a fully random map is similarly assumed to be difficult and the corresponding problem is known as the OV problem. To reduce the size of the public key, MAYO employs an optimisation that constructs a larger map $\mathcal{P}^*$ from a smaller map $\mathcal{P}$ before finding the preimage. Beullens refers to this process as "whipping up" the map and the resulting variant of the MQ problem that asks to find the preimage in $\mathcal{P}^*$ is thus known as the *multi-target whipped MQ* problem.

An overview of parameters for MAYO is given in Table 2. For further details we refer to the specification [BCC+23]. We are focusing on MAYO_1 in this paper, though variants MAYO_2, MAYO_3 and MAYO_5 can be approached similarly. A caveat that applies to MAYO_2 is addressed explicitly in Section 7.

Algorithm 1 MAYO.KeyGen() [Beu22]

```
Output: Public key $pk$, secret key $sk$
1: $\mathbf{O} \leftarrow \mathbb{F}_q^{(n-o)\times o}$
2: $\mathsf{seed}_{sk} \leftarrow \{0,1\}^\lambda$
3: $\mathsf{seed}_{pk} \leftarrow \mathsf{SHAKE256}(\mathsf{seed}_{sk})$
4: for $i$ from 1 to $m$ do
5:     $\mathbf{P}_i^{(1)} \leftarrow \mathsf{Expand}(\mathsf{seed}_{pk} \parallel P1 \parallel i)$
6:     $\mathbf{P}_i^{(2)} \leftarrow \mathsf{Expand}(\mathsf{seed}_{pk} \parallel P2 \parallel i)$
7:     $\mathbf{P}_i^{(3)} \leftarrow \mathsf{ToUpperTriangular}(-\mathbf{O}\mathbf{P}_i^{(1)}\mathbf{O}^T - \mathbf{O}\mathbf{P}_i^{(2)})$
8: return $(pk, sk) = ((\mathsf{seed}_{pk}, \{\mathbf{P}_i^{(3)}\}_{1\le i\le m}), (\mathsf{seed}_{sk}, \mathbf{O}))$
```

Algorithm 2 MAYO.Sign(sk, M) [Beu22]

```
Input: Secret key sk, message $M$
Output: Signature $\sigma$
1: $(\mathsf{seed}_{sk}, \mathbf{O}) \leftarrow \mathsf{sk}$
2: $\mathsf{seed}_{pk} \leftarrow \mathsf{SHAKE256}(\mathsf{seed}_{sk})$
3: for $i$ from 1 to $m$ do
4:     $\mathbf{P}_i^{(1)} \leftarrow \mathsf{Expand}(\mathsf{seed}_{pk} \parallel P1 \parallel i)$
5:     $\mathbf{P}_i^{(2)} \leftarrow \mathsf{Expand}(\mathsf{seed}_{pk} \parallel P2 \parallel i)$
6: $R \leftarrow \{0,1\}^r$                                    ▷ Deterministic variant: $R \leftarrow \{0\}^r$
7: $\mathsf{salt} \leftarrow \mathsf{SHAKE256}(M \parallel R \parallel \mathsf{seed}_{sk})$
8: $\mathbf{t} \leftarrow \mathsf{SHAKE256}(M \parallel \mathsf{salt})$
9: for $ctr$ from 0 to 255 do
10:    $\mathsf{V} \leftarrow \mathsf{SHAKE256}(M \parallel \mathsf{salt} \parallel \mathsf{seed}_{sk} \parallel ctr)$
11:    $\mathsf{v}_1, \dots, \mathsf{v}_k \leftarrow \mathsf{Decode}(\mathsf{V})$
12:    $(\mathbf{A}, \mathbf{y}) \leftarrow \mathsf{BuildLinearSystem}(\{\mathsf{v}_1, \dots, \mathsf{v}_k\}, \mathbf{O}, \mathbf{P}^{(1)}, \mathbf{P}^{(2)}, \mathbf{t})$
13:    $\mathsf{x} \leftarrow \mathsf{SampleSolution}(\mathbf{A}, \mathbf{y})$        ▷ Try to find $Ax = y$ (i.e. $\mathcal{P}^*(\mathbf{s}) = \mathbf{t}$)
14:    if $\mathsf{x} \neq \perp$ then break
15: $\mathbf{s} \leftarrow \{\mathsf{v}_i + \mathbf{O}\mathsf{x}_i \parallel \mathsf{x}_i\}_{1\le i\le k}$
16: return $\sigma = (\mathbf{s}, \mathsf{salt})$
```

The main components of the MAYO scheme are the key generation procedure, the signing procedure and the verification procedure. We provide simplified versions of these procedures here. For the full versions, including the definitions of functions not defined here, we refer to the specification [BCC+23].

3.1.1 Key generation (Algorithm 1)

The key generation samples a random matrix $\mathbf{O}$ that forms the oil space. It also samples a secret random seed seed_{sk}, and a public seed seed_{pk} from which the sequences of m matrices $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ of the multivariate quadratic map $\mathcal{P}$ are expanded pseudorandomly. This allows the public key to only contain the seed instead of the matrices, thereby reducing its size. Finally, the remaining sequence of m matrices $\mathbf{P}_i^{(3)}$ is chosen such that the map $\mathcal{P}$ vanishes on the oil space $\mathbf{O}$. The public key consists of the public seed seed_{pk} and the sequence of matrices $\mathbf{P}^{(3)}$. The secret key consists of the secret seed seed_{sk} and the matrix $\mathbf{O}$.

Algorithm 3 MAYO.Verify(pk, M, σ) [Beu22]

Input: Public key pk, message M, signature σ
Output: Boolean

```
1: $(\mathsf{seed}_{pk}, \mathbf{P}^{(3)}) \leftarrow pk$
2: for $i$ from 1 to $m$ do
3:     $\mathbf{P}_i^{(1)} \leftarrow \mathsf{Expand}(\mathsf{seed}_{pk} \parallel P1 \parallel i)$
4:     $\mathbf{P}_i^{(2)} \leftarrow \mathsf{Expand}(\mathsf{seed}_{pk} \parallel P2 \parallel i)$
5: $(\mathbf{s}, \mathsf{salt}) = \sigma$
6: $\mathbf{t} \leftarrow \mathsf{SHAKE256}(M \parallel \mathsf{salt})$
7: $\mathbf{t}' \leftarrow \mathsf{Evaluate}_{\mathbf{P}^{(1)},\mathbf{P}^{(2)},\mathbf{P}^{(3)}}(\mathbf{s})$        ▷ $\mathcal{P}^*(\mathbf{s}) = \mathbf{t}'$
8: return true if $\mathbf{t} = \mathbf{t}'$ else false
```

3.1.2 Signing (Algorithm 2)

The signing procedure extracts the oil space $\mathbf{O}$ and the sequences of matrices $\mathbf{P}^{(1)}$ and $\mathbf{P}^{(2)}$ from the secret key. It then uses SHAKE256 to compute the salt and from the salt, the target value $\mathbf{t}$. Using SHAKE256, the vinegar seed V is derived from the message M, the salt, the secret seed seed_{sk} and a counter value ctr. From there, a system of linear equations is constructed and solved, corresponding to finding $\mathbf{s}$ such that under the multivariate quadratic map $\mathcal{P}^*$, it holds that $\mathcal{P}^*(\mathbf{s}) = \mathbf{t}$. For certain choices of vinegar values, the matrix $\mathbf{A}$ that defines the system does not have full rank and thus the system cannot be solved. If this is the case, the signing process restarts with a different vinegar seed. Solving the system can be done efficiently, due to the knowledge of the oil space $\mathbf{O}$. Once a solution is found, the resulting signature consists of a sequence $\mathbf{s}$ of oil vectors masked by vinegar values $\mathsf{v}_1 + \mathbf{O}\mathsf{x}_1, \ldots, \mathsf{v}_k + \mathbf{O}\mathsf{x}_k$ concatenated with the corresponding arguments $\mathsf{x}_1, \ldots, \mathsf{x}_k$, and the salt.

3.1.3 Verification (Algorithm 3)

The verification procedure extracts and derives the sequences of matrices $\mathbf{P}^{(1)}$, $\mathbf{P}^{(2)}$ and $\mathbf{P}^{(3)}$ from the public key. It also extracts the argument $\mathbf{s}$, as well as the salt from the signature. It then derives the original target value $\mathbf{t}$ and evaluates the multivariate quadratic map $\mathcal{P}^*$ with the argument $\mathbf{s}$ to compute the value $\mathbf{t}'$. If the resulting values $\mathbf{t}$ and $\mathbf{t}'$ match, the signature is valid.

3.2 Voltage fault injection and fault model

Voltage fault injection manipulates the voltage supplied to a processor to induce faults. Fault injection approaches differ mainly in the required degree of precision for the timing of the glitch and in the offered degree of control over which parts of the program are affected. Techniques that require less precise timing, such as [BBPP09, BBBP13], which uniformly underpower the processor executing the program to slow down logic gates to cause faults, also offer less control over which instructions are affected and how. Techniques which require more precise timing, such as [O'F16, BFP19], use precise voltage spikes to cause faults, and allow affecting only specific instructions.

In this paper, we apply the fault injection technique of O'Flynn [O'F16], which uses a crowbar circuit to short the power rails of the processor and momentarily drop the voltage, thereby inducing oscillations in the target circuit which potentially cause faults. The fault model of this technique is single/multiple instruction skipping (i.e. the processor is forced to skip the execution of specific attacker-chosen instructions through a precise fault injection). While this technique can also cause instruction or data corruption, these are not relevant for the attacks presented in this paper.

3.3 SHAKE256 algorithm

SHAKE256 is an extendable output function (XOF) that is part of the FIPS202 standard [Nat15]. Extendable output functions generate pseudorandom sequences of output from a given input. SHAKE256 uses the sponge construction [BDPVA11] and an iterated one-way permutation function from the KECCAK family of permutations.

Fig. 2 shows the main components of the construction. It contains two buffers, which are zero-initialised, that form the state of the construction. During the absorption phase, blocks of the input are XORed into the state and the state is updated using the permutation function. This is repeated until all blocks of input have been absorbed. Then, output is generated in the squeezing phase by taking part of the state, extracting it as a block of output, and updating the state using the permutation function. The algorithm terminates once all output blocks have been generated.

4 Simulation method

A large number of fault simulation techniques have been proposed in the past. As they are too many to list, we focus on those relevant to this work.

Techniques such as [HSP21, HGA+21, Risnd, MTO24] perform concrete execution using emulation. These techniques are typically based on QEMU [Bel05] or the related Unicorn engine [ND15] and differ mainly in how the fault injection is configured and modelled. Other techniques, such as [LFBP24, Lan22, DHHB08, CDSLN20, PNKI13] instead use symbolic execution based on a variety of underlying frameworks. The simulation technique used in this paper closely matches the approach by Lancia [Lan22], which employs the same underlying framework with similar modifications, but focuses on various bit flip fault models instead of the instruction skipping fault model.

4.1 Symbolic execution

Symbolic execution is a simulation technique that substitutes computations on concrete values with computations on symbolic values [Kin76]. This allows for all possible branches of a program to be explored unconditionally, instead of only those reached under a given variable assignment. In the context of fault injection, symbolic execution can identify faults that can only be reached under certain conditions, which may be difficult or impossible to achieve with concrete execution, especially if data dependencies are only created as a result of the fault injection.

Our simulation technique uses the `angr` framework [SWS+16] for performing symbolic execution. We selected this framework due to its extensibility and compatibility with our existing Python-based tooling, as well as its support for a large number of architectures (including multiple ARM architectures, x86, RISC-V, MIPS and even domain-specific architectures like Tricore). Using the framework, we created a simulation engine that is able to skip a configurable number of instructions in given parts of the program. In our case, we consider the first part of the SHAKE256 computation, from the zero-initialisation of the sponge up until the absorption into the sponge is completed.

4.2 Reachability estimation using loopy belief propagation

As a part of the symbolic execution, the `angr` framework annotates each state with a list of constraints that are necessary to reach that state. These constraints are conditional expressions on concrete or symbolic bitvectors, modelled as an abstract syntax tree. For example, the constraint `<Bool reg_r4_8_32{UNINITIALIZED} <= 0x87>` indicates that the value of the 32-bit register `r4` must be less than or equal to `0x87`.

The idea behind our approach is to use the abstract syntax tree to construct a factor graph. Consider, for example, the expression `<Bool reg_r4_8_32{UNINITIALIZED} & 0x1 == reg_r5_8_32{UNINITIALIZED}>`. Its abstract syntax tree will contain the values `r4`, `0x8`, and 1, as well as the operations "`&`" and "`==`". To translate this into a factor graph, we first create a node for each bit value (i.e. 32 nodes for the value of `r4`, 32 nodes for the value of `0x1` and 32 nodes for the value `r5`, where all values are expressed using the same number of bits) and temporary or output values (i.e. 32 nodes for the output of the "`&`" operation and 32+1 nodes for the computation of the "`==`" operation). We then add a factor to the graph for each bit of the `&` operation. The first factor is connected to the first bit of `r4`, the first bit of `0x1` and the first bit of the output of the operation. The function realised by the first factor is 1 if the output bit is equal to the bitwise AND of the input bits (i.e. the probability of a given input assignment is 1 if the output is the bitwise AND of the inputs). This is repeated for each of the bits.

Then, we add a factor to the graph for each bit of the "`==`" operation. Here, the first factor is connected to the first bit of the output of the `&` operation, the first bit of `r5` and the first of the temporary bits of the "`==`" operation, as well as the additional temporary bit, which is used as a placeholder for the previous comparison (which does not exist for the first comparison, so it is set to 1). The function realised by this factor is 1 if the temporary output bit matches both of the input bits being equal and the previous comparison being 1 (i.e. the probability of a given assignment of bit values occurring is 1 if the output is 1 only if the current bits are equal and all previous bits were also equal). This is repeated for each of the bits, with the outputs of the previous comparison becoming the temporary bits for the previous comparison in the subsequent comparisons. Other operations, like addition or other comparisons, can similarly be implemented based on their corresponding bit operations. By applying loopy belief propagation on the factor graph, we are able to derive approximate marginal probabilities for the values of individual bits and thus estimate the probability of a constraint being satisfied.

An advantage of this approach is that it allows to represent arbitrary per-bit probabilities for input values. In our simulations, we assume that uninitialised bits (bits that are not assigned a specific value during the simulated part of the algorithm) are distributed uniformly at random, but it would be possible to achieve more accurate predictions by integrating additional information about the distributions, for example by sampling the register and memory values from a real device. We did not pursue this approach because, in our experiments, the target device runs a part of the cryptographic algorithm and no other tasks. Thus, any values gathered by sampling would be unlikely to be representative of a real device.

We found that an unmodified version of the loopy belief propagation algorithm performs poorly due to the presence of short loops caused, for example, by the comparison of neighbouring bits. This is a known limitation of the loopy belief propagation algorithm and a number of techniques have been proposed to address it, including [YFW00, KCN21]. For our use case, we found it sufficient to identify short loops and combine together all of their factors. In cases where conditional expressions contain more terms or more complex arithmetic operations that cause loops, the quality of the estimate will decrease substantially and combining loops into a single factor becomes infeasible.

4.3 Results

Within 81 minutes of simulation time, we identified 665 candidate states that successfully complete the first part of the SHAKE256 computation despite the injection of a fault. These 665 candidate states correspond to single instruction skips at 122 unique addresses. Recall that, under symbolic execution, a state is added for each branch in the program flow if the branch condition cannot be statically resolved (i.e. if the branch condition depends on a data or register value that is not unconditionally set during the program execution), thus the number of candidate states is higher than the number of single instruction skips. Of the 665 candidate states, we identified 75 states where the number of unknown bits in the sponge after the absorption phase is less than or equal to 32, corresponding to the injection of single instruction skips at eight unique addresses.

We take into account that the first $32 + 24$ bytes of the input to the SHAKE256 function are public and are thus allowed to occur in the sponge without affecting the search space, provided that their positions are known. These identified states are those in which an attacker could, with reasonable number of enumerations, recover the secret key using the technique described in Section 7. However, not all of the 75 states are reachable with high probability.

Using the loopy belief propagation approach, we found that 69 of the 75 states for which the search space for the sponge contents is small, are unlikely to be reached under the assumption that uninitialised memory and register values are uniformly and randomly distributed. Of these 69 unlikely reachable states, 61 are related to skipping the initialisation of register `r4` at the beginning of the program and thus require specific values for the uninitialised register in order for program execution to finish successfully.

Of the remaining eight unlikely reachable states, one state is related to skipping the initialisation of register `r8`, thus shifting the area of memory into which the data is absorbed into uninitialised memory. This state is wrongly annotated as requiring the register of `r8` to be zero, causing us to incorrectly deem it as unlikely reachable. A possible cause for the wrong annotation is that the default memory model in `angr` does not handle writes to memory with symbolic addresses correctly.

The remaining seven unlikely reachable states skip a subtraction operation that sets the condition flags during the XOR of data into the sponge, thus causing a loop abort when the previous condition flags are set to certain values. This is only possible if several specific bits of the first part of the input are zero. Note that, while the first part of the input to the SHAKE256 function (the message digest) is attacker controlled, setting specific bits in it to zero would require finding a preimage for such a value under SHAKE256, which is considered infeasible, so we do not consider the corresponding attack further.

The remaining six states are unconstrained and thus correctly identified as reachable. Of these, three states are related to skipping the branching to the absorption function or the initialisation of one of its parameters. The corresponding attack for these three states is described in Section 6.1. One state is related to skipping the branching to a subroutine called from the absorption function, instead of skipping the absorption function itself, with the same result. We did not pursue this attack further, as it requires skipping a single branching instruction without affecting any of the surrounding instructions, which is impractical in our experimental setup. Finally, two states are related to skipping a backwards branch in different iterations of a loop during the absorption, thus causing a loop abort and leaving the sponge partially initialised with the public part of the input. This is the same loop that is targeted by the faults that skip setting a condition flag mentioned earlier. The difference is that the branch here is skipped directly, thus removing the need for the condition flags to have certain values. The corresponding attack for these two states is described in Section 6.2.

Overall, we found that the loopy belief propagation technique can provide seemingly reasonable estimates for the probability of satisfying certain constraints under the assumption that uninitialised values are uniformly and randomly distributed. However, for the constraints we encountered in our simulation, these estimates are of limited use. Most of the constraints require registers to contain specific values where, for example, the four highest bits of a 32-bit register must be 1 and all other bits must be 0. Under the assumed distribution, the probability for such a value to occur is small (2^{-32} for the described case), thus differentiating between states based on their probabilities is not possible. Future work may consider alternative approaches for establishing statistical models for the distribution

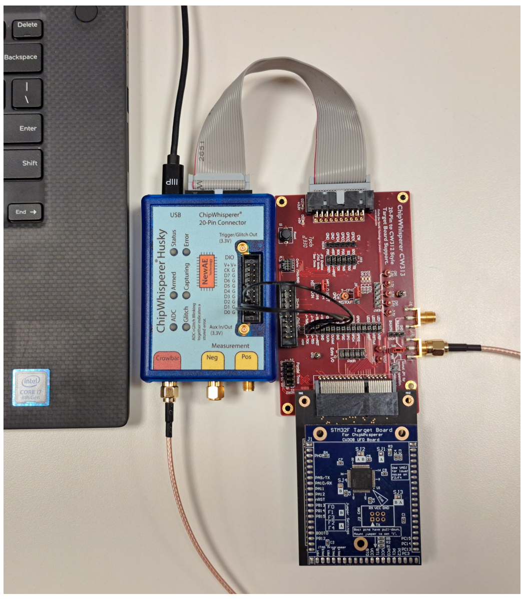

Figure 1: ChipWhisperer-Husky, CW313 adapter board and CW308T-STM32F4 board used in the experiments.

of uninitialised memory and register values, such as the sampling technique mentioned earlier. Additionally, neither the belief propagation approach itself, nor the combining approach for graphs that contain loops, scale well, preventing larger, more complex expressions from being used. Future work may consider applying other known techniques, such as [YFW00, KCN21], in this context, or reducing the estimation complexity by other means.

The approach we employed in this paper is to reuse the logic that splits constraints into individual bit operations and apply it to identify a subset of states have at most 32 unknown bits in the sponge, before applying the loopy belief propagation to only those states. The idea is to mark a bit as unknown if it is computed from a bit that is itself unknown. In this way, uninitialised values from input registers propagate to the output values if they are involved in relevant computations. While such an approach generally overestimates the number of unknown bits (for example during addition where certain bit combinations of the output and carry bits cannot occur but the bits are still marked as unknown, even though the value of one could be inferred from the other), it provides a fast and sufficiently good classification to identify states which could then be further analysed using the loopy belief propagation technique (though as mentioned previously, the insights gained from the latter are limited in our case). There may be other approaches to reduce the number of states that the loopy belief propagation is applied to or the size of the factor graph, that could be useful.

5 Experimental Setup

This section describes the equipment used for the experiments, as well as the target implementation of MAYO.

5.1 Equipment

The equipment used in our experiments is shown in Fig. 1. The target device is a CW308-STM32F4 board containing an ARM Cortex-M4 STM32F415RGT6 processor running at

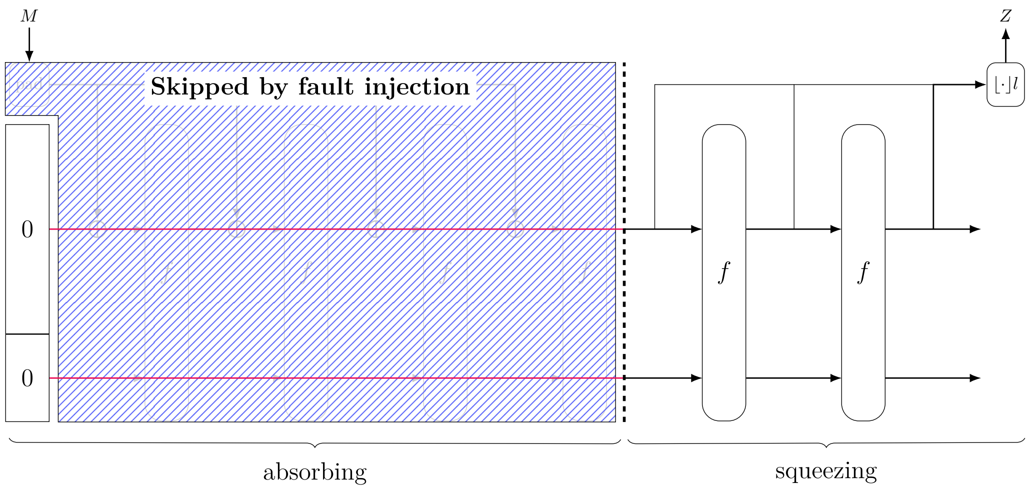

Figure 2: SHAKE256 sponge construction (adapted from [BDPVA11]). The fault injection skips the absorption step highlighted in blue, thus causing the output Z to be constant and independent of the input M.

a frequency of 24 MHz. It is mounted on a CW313 adapter board and faults are injected using a ChipWhisperer-Husky. The fault injection is triggered via ARM CoreSight DWT watchpoints, thus avoiding any modification of the assembly code otherwise caused by inserting a trigger. Alternative trigger sources, such as communication with peripheral devices or similarity of the power consumption to reference waveforms, could be used by an attacker that does not have control over the target device.

5.2 Target implementation

In our experiments, we use the MAYO implementation by Beullens et al. [BCC+24]. Specifically, we use the most recent commit (`fe46236`) of the `main` branch, not the `nibbling-mayo` branch. However, the changes introduced by the nibble representation do not affect any of the components that we consider in this paper, so we expect the attacks to translate to that version directly.

The implementation is compiled using `arm-none-eabi-gcc` with the highest optimization level `-O3` (recommended default).

6 Fault Injection Attacks

This section describes the three fault injection attacks on MAYO.

```
size_t keccak_inc_absorb(uint64_t *state, size_t bytes_not_permuted,
                         uint8_t *m, size_t mlen) {
  while (mlen + bytes_not_permuted >= 136) {
    KeccakF1600_StateXORBytes(state, m, bytes_not_permuted);
    mlen -= 136 - bytes_not_permuted;
    m += 136 - bytes_not_permuted;
    bytes_not_permuted = 0;
    KeccakF1600_StatePermute(state);
  }

  KeccakF1600_StateXORBytes(state, m, bytes_not_permuted, mlen);
  return bytes_not_permuted + mlen;
}
```

Listing 1: The C code of the `keccak_inc_absorb` procedure. The function targeted by absorption skipping attack is highlighted in green.

```
...
mov r1, r8
mov r3, r4
mov r0, r7
bl KeccakF1600_StateXORBytes
ldr r2, [sp, #208]
ldr r3, [sp, #212]
...
```

Listing 2: An excerpt of the assembly code of the `keccak_inc_absorb` procedure. The `bl` instruction targeted by the absorption skipping attack is highlighted in green.

6.1 Absorption skipping attack on SHAKE256

The first attack, which we call the *absorption skipping* attack, extends the technique for skipping the absorption of input data during the calculation of a hash introduced in the context of CRYSTALS-Dilithium in [JMD24] to MAYO.

The implementation of MAYO by Beullens et al. [BCC$^+$24] considered in this paper and the implementation of CRYSTALS-Dilithium by Abdulrahman et al. [AHKS22] considered in [JMD24] use the same SHAKE256 implementation from the `pqm4` project. The SHAKE256 implementation internally consists of four functions: `keccak_inc_init`, `keccak_inc_absorb`, `keccak_inc_finalize`, and `keccak_inc_squeeze`. The first function, `keccak_inc_init`, is called to zero-initialise the sponge. Then, the `keccak_inc_absorb` function (see Listing 1) absorbs arbitrary-sized input blockwise into the sponge. This function may be called multiple times to absorb data from different buffers. Next, the `keccak_inc_finalize` function is called once to prepare the sponge for squeezing. Finally, the `keccak_inc_squeeze` function, which may be called multiple times, extracts arbitrary-sized output blockwise by squeezing the sponge. For the computation of the vinegar seed, these four functions are invoked by calling the `shake256` function in lines 15 and 16 of Listing 4.

```
1   __KeccakF1600_StateXORLanes:
2   __KeccakF1600_StateXORLanes_LoopAligned:
3     ldr r4, [r1], #4
4     ldr r5, [r1], #4
5     ldrd r6, r7, [r0]
6     toBitInterleaving r4, r5, r6, r7, r3, 0
7     strd r6, r7, [r0], #8
8     subs r2, r2, #1
9     bne __KeccakF1600_StateXORLanes_LoopAligned
10    bx lr
```

Listing 3: The assembly code of the inner loop in the absorption procedure. The `bne` instruction targeted by the absorption abort attack is highlighted in orange.

The idea behind the absorption skipping attack is to prevent the absorption of data into the sponge through fault injection in order to fix the value of the vinegar seed to a known constant. If the branch to the `KeccakF1600_StateXORBytes` function (see line 11 of Listing 1 or the `bl` instruction in line 5 of Listing 2) is skipped, the sponge does not absorb any data. The loop in lines 3 to 9 of Listing 1 is never executed in the computation of the vinegar seed, because the length of the input $M \parallel \mathsf{salt} \parallel \mathsf{seed}_{sk} \parallel ctr$ is 81 bytes, which is less than the 136 bytes required to trigger a permutation (i.e. the input does not fill a full block). Due to the zero-initialisation performed by the `keccak_inc_init` function prior to the fault injection, skipping the absorption leaves the sponge in an initialised, but empty state. Hence squeezing the vinegar seed output from this sponge generates a constant sequence of bytes known to the attacker.

Fig. 2 shows the sponge construction with the absorbing and squeezing phases, as well as the two buffers of the sponge whose values are propagated to the squeezing phase by the fault injection. Note that, unlike in the first attack of Aulbach et al. [AMSU24], the presented attack is not limited to a specific memory allocation strategy.

6.2 Absorption abort attack on SHAKE256

The idea behind the second attack, which we call the *absorption abort* attack, is to abort the loop that performs the actual absorption of data into the sponge. By skipping a backwards branch (see `bne` instruction in line 9 of Listing 3), the loop exits early and the sponge only absorbs the first part of the data. Since the first two arguments to the SHAKE256 function in the computation of the vinegar seed are the public message digest and salt, the attacker can predict the contents of the sponge after the absorption and thus its output.

Note that, for both the absorption skipping and absorption abort attacks to be successful, the signing should not fail after the fault injection (i.e. there must be a solution to the system $\mathbf{Ax} = \mathbf{y}$). Otherwise the next iteration of the signing loop will overwrite the faulty seed. However, the failure probability for signing is known to be low (upper bound of $\simeq 1.55 \times 10^{-11}$ for MAYO_1 and MAYO_2, $\simeq 9.25 \times 10^{-19}$ for MAYO_3 and $\simeq 3.61 \times 10^{-21}$ for MAYO_5; see Lemma 3 of [Beu22]). Empirically, we observed no instances of failure during signing of 40,000 random messages. Therefore this is not an issue in practice.

6.3 Argument initialisation skipping attack on SHAKE256 via `memcpy`

```
1   shake256(tmp, digest_bytes, m, mlen); // M_digest
2   randombytes(tmp + digest_bytes, salt_bytes) // R
3
4   // Store M_digest ‖ R ‖ seed_sk contiguously in tmp
5   memcpy(tmp + digest_bytes + salt_bytes, seed_sk, sk_seed_bytes);
6   shake256(salt, salt_bytes, tmp,
7            digest_bytes + salt_bytes + sk_seed_bytes); // salt
8
9   // Reuse tmp to store M_digest ‖ salt contiguously
10  memcpy(tmp + digest_bytes, salt, salt_bytes);
11
12  ...
13  *(tmp + digest_bytes + salt_bytes + sk_seed_bytes) = ctr;
14  // Sample seed for vinegar values
15  shake256(V, k * v_bytes + r_bytes, tmp,
16           digest_bytes + salt_bytes + sk_seed_bytes + 1);
```

Listing 4: The C code for the computation of the salt, **t** and vinegar values. The computation of the vinegar seed is highlighted in purple. The function targeted by the argument initialisation skipping attack is highlighted in red.

```
1   ...
2   mov r1, r9
3   movs r2, #24
4   sub.w r0, r7, #452
5   bl memcpy
6   movs r3, #80
7   movs r1, #24
8   ...
```

Listing 5: An excerpt of the assembly code for the copying of the secret seed seed_{sk} into the buffer `tmp`. The `bl` instruction targeted by the argument initialisation skipping attack is highlighted in red.

The third attack, which we call the *argument initialisation skipping* attack, targets a single `memcpy` operation prior to the computation of the salt.

The SHAKE256 implementation used by Beullens et al. [BCC+24] requires all arguments of the `shake256` function to be stored contiguously in a single buffer. In practice, a buffer `tmp` is reused across multiple invocations of the `shake256` function. More specifically, the computation of the salt with the arguments $M \parallel R \parallel \mathsf{seed}_{sk}$ is realised by first outputting the message digest into the buffer `tmp` (see line 1 of Listing 4), then copying a random value R into the buffer (line 2) and finally copying the secret seed seed_{sk} into the buffer (line 5). The computation of the vinegar seed reuses `tmp` by overwriting the value R with the computed salt (line 10) and appending the value *ctr* (line 13) before calling the `shake256` function again. By skipping the copying of seed_{sk} in line 5 (i.e. skipping the branching instruction `bl` in line 5 of Listing 5), the corresponding section of `tmp` is left uninitialised.

At the end of the signing procedure, the implementation by Beullens et al. [BCC+24] zeroes most of the memory as a security measure. Thus, when the same section of memory is reused during the next invocation of the signing procedure, the uninitialised parts of `tmp` are set to zero despite not being initialised. As a consequence, all arguments of the `shake256` function during the computation of the vinegar seed can be predicted by an attacker. More specifically, the message digest can be derived from the message, the salt

Algorithm 4 RecoverSecretKey(σ, V)

```
1: $(\mathbf{s}_{\mathsf{enc}}, \mathsf{salt}) \leftarrow \sigma$
2: $\mathbf{s} \leftarrow \mathsf{Decode}(\mathbf{s}_{\mathsf{enc}})$
3: for $i$ from 1 to $k$ do
4:     $\mathsf{v}_i \leftarrow \mathsf{Decode}(n-o, \mathsf{V}[(i-1)*\mathsf{v\_bytes} : i*\mathsf{v\_bytes}])$
5:     $(\mathsf{v}_i + \mathbf{O}\mathsf{x}_i, \mathbf{x}_i) \leftarrow s[(i-1)*n : i*n]$
6:     $\mathbf{y}_i \leftarrow \mathbf{z}_i - \mathbf{v}_i$
7: $\mathbf{A} \leftarrow \begin{pmatrix} \mathsf{v}_1 + \mathbf{O}\mathsf{x}_1 - \mathsf{v}_1 & \cdots & \mathsf{v}_k + \mathbf{O}\mathsf{x}_k - \mathsf{v}_k \end{pmatrix}$
8: $\mathbf{X} \leftarrow \begin{pmatrix} \mathsf{x}_1 & \cdots & \mathsf{x}_k \end{pmatrix}$
9: Solve $\mathbf{XB} = \mathbf{I}_o$ as in Equation 1. If no solution exists, return $\perp$.
10: $\mathbf{O} \leftarrow \mathbf{AB}$
11: return $\mathbf{O}$
```

can be extracted from the signature, the secret seed seed_{sk} is zero by assumption and the value of the *ctr* could either be enumerated over all possible 256 values, or assumed to be zero, as the same justification regarding the failure probability of the signing from the first attack applies here. This allows the attacker to predict the vinegar seed.

Note that, unlike in the second attack method of Aulbach et al. [AMSU24], it is not necessary that `tmp` is allocated in the same section of memory. Instead, even shifts of several hundred bytes could cause `tmp` to be placed in zeroed memory.

7 Secret key recovery

All three attacks presented in Section 6 enable the attacker to predict the seed used for the sampling of the vinegar values. This section presents a novel approach for recovering the full oil space $\bar{\mathbf{O}}$ from a faulty signature generated with a known vinegar seed. Note that in MAYO (and newer descriptions of UOV [BCD+23]) the full oil space is defined as $\bar{\mathbf{O}} = \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_o \end{bmatrix}$, which does not significantly affect the security of the scheme [Beu22], but slightly increases the complexity of the recovery.

Previous work has shown that it is possible to recover the entire oil space from a small number of vectors. The reconciliation attack [DYC+08] provides a way to find additional vectors in the oil space given an initial vector. Aulbach et al. [ACK+23] use a combination of the reconciliation attack and the Kipnis-Shamir attack [KS98] to recover the full oil space from a single vector. Beullens' intersection attack [Beu21, Beu22] also combines ideas from the reconciliation attack and the Kipnis-Shamir attack to identify initial vectors for the reconciliation attack. Recently, Pébereau [Péb24] presented efficient polynomial-time algorithms for recovering the secret key from UOV schemes (including MAYO).

These techniques could also be applied to recover the secret key from the faulty signatures in the attacks presented in this paper. However, knowledge of the vinegar seed alongside the structure of the MAYO signature allows for an alternative approach of performing secret key recovery. We stress that this approach is *not a replacement* for existing techniques: It requires knowledge of all vinegar values and occasionally fails to recover the secret key. We present this approach mainly for completeness.

A MAYO signature contains the masked oil vectors $\mathsf{v}_1 + \mathbf{O}\mathsf{x}_1, \ldots, \mathsf{v}_k + \mathbf{O}\mathsf{x}_k$ concatenated with the vectors $\mathsf{x}_1, \ldots, \mathsf{x}_k$. The attacker recovers $\mathsf{v}_1, \ldots, \mathsf{v}_k$.

If the set $\{\mathsf{x}_1, \ldots, \mathsf{x}_k\}$ contains a subset S of o-many linearly independent vectors, then the set

$$\left\{ \begin{pmatrix} \mathbf{O}\mathsf{x}' \\ \mathsf{x}' \end{pmatrix} \mid \mathsf{x}' \in S \right\}$$

already defines an (equivalent) basis for the full oil space that can be used for signature forgery. However, in the following, we show how to compute the original full oil space without explicitly determining the subset S. Although the benefits of knowledge of the original basis of the oil space compared to knowledge of an equivalent basis are not immediately clear, it is possible that future attacks could leverage this information.

Let

$$\mathbf{A} \coloneqq \begin{bmatrix} \mathsf{v}_1 + \mathbf{O}\mathsf{x}_1 - \mathsf{v}_1 & \cdots & \mathsf{v}_k + \mathbf{O}\mathsf{x}_k - \mathsf{v}_k \end{bmatrix} \in \mathbb{F}_q^{n \times k}$$

be a matrix of oil vectors computed using knowledge of the vinegar values and

$$\mathbf{X} \coloneqq \begin{bmatrix} \mathsf{x}_1 & \cdots & \mathsf{x}_k \end{bmatrix} \in \mathbb{F}_q^{o \times k}.$$

Any right inverse of $\mathbf{X}$, which are the solutions $\mathbf{B} \in \mathbb{F}_q^{k \times o}$ of the linear system

$$\mathbf{X}\mathbf{B} = \mathbf{I}_o \tag{1}$$

where $\mathbf{I}_o$ is the $o \times o$ identity matrix, yields the original oil space $\mathbf{O}$ as

$$\mathbf{O} = \mathbf{A}\mathbf{B}$$

from which we can derive the original full oil space $\bar{\mathbf{O}}$ as

$$\bar{\mathbf{O}} = \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_o \end{bmatrix}.$$

In order for the system in Eq. 1 to be solvable, the $o \times k$ matrix $\mathbf{X}$ must have (row) rank o, which is equivalent to the set $\{\mathsf{x}_1, \ldots, \mathsf{x}_k\}$ containing a subset of o-many linearly independent vectors. In parameter set MAYO_2, where $k < o$, such a subset cannot exist and the method thus cannot be used. In the other parameter sets, it is possible for the vectors $\mathsf{x}_1, \ldots, \mathsf{x}_k$ to be linearly dependent such that $\mathbf{X}$ does not have rank o. In these cases, the method fails and key recovery must instead be performed using one of the previously mentioned techniques [ACK+23, Péb24].

The techniques [DYC+08, ACK+23, Beu21, Beu22, Péb24] are also capable of recovering the original basis $\mathbf{O}$. Compared with, e.g. the reconciliation attack [DYC+08], the theoretical benefit of the approach described above is that the secret key recovery only requires solving a single system of $o \times o$ equations with $k \times o$ variables for full secret key recovery, instead of solving a system of m equations with m variables for each of the $o - 1$ additional vectors that need to be recovered. In practice, however, the performance difference may be quite small. Additionally, the presented approach occasionally fails to recover the key, in which case a different technique must be used. Nonetheless, it may be beneficial in cases where all vinegar values are known.

8 Experimental Results

This section describes the results of the fault injection attacks and subsequent secret key recovery.

8.1 Fault injection success probability

We succeeded with skipping the execution of the `KeccakF1600_StateXORBytes` function in 81.9% of 1,000 attempts and aborting the loop during the absorption in 36.4% of 1,000 attempts. We further succeeded with skipping the execution of the `memcpy` function in 100% of 1,000 attempts.

The success probability of the absorption abort attack is substantially lower than the other two because the device crashes whenever the instruction following the skipped one is affected by the fault. Hence, the injected fault has to be very precise, which we found difficult to achieve in our experimental setup. The success probability of the first attack is higher than that presented in [JMD24] because we tweaked the offset parameter that controls the position of the rising edge of the glitch within the clock cycle (we use an offset of 600 units instead of 700 units). In practice, the success probabilities of fault attacks are not always consistent, so the probabilities reported here should be interpreted as a best-case scenario for an attacker.

8.2 Secret key recovery

We applied Algorithm 4 to the faulty signatures generated in the first phase of the attack.

As a guess for the vinegar seed in the absorption skipping attack, we use the output of SHAKE256 when invoked with the argument $\{0\}^{648}$, i.e. an all-zero input of 81 bytes, which is equivalent to the output generated after skipping the absorption phase. In SHAKE256, the length of the input is absorbed into the sponge after the absorption phase. As this step is not affected by the fault attack, the length must be the same as that of the original input to generate the correct output.

As a guess for the vinegar seed in the absorption abort attack, we use the output of SHAKE256 when invoked with the argument $M[0:128] \parallel \{0\}^{520}$, i.e. we take only the first 128 bits of the message digest and extend with zeros to achieve the same length input.

As a guess for the vinegar seed in the argument initialisation skipping attack, we use the output of SHAKE256 when invoked with the arguments $M \parallel \mathsf{salt} \parallel 0^{|\mathsf{seed}_{sk}|} \parallel 0$, i.e. we substitute the secret seed seed_{sk} and the ctr with 0. All of these guesses can easily be made by an attacker, because they are either constants or use public information contained in the faulty signatures.

Algorithm 4 successfully recovered the secret key from 818 out of 819 faulty signatures (99.88%) for the absorption skipping attack, 361 out of 364 faulty signatures (99.18%) for the absorption abort attack, and 993 out of 1,000 faulty signatures (99.30%) for the argument initialisation skipping attack. The remaining secret keys can be recovered using the techniques [ACK+23, Péb24].

9 Countermeasures

This section discusses possible countermeasures against the presented fault attacks.

9.1 Absorption skipping and absorption abort attacks on SHAKE256

The absorption skipping and absorption abort attacks targeting the SHAKE256 procedure can be mitigated by eliminating the branches that are targeted by the fault injection. For the absorption skipping attack, it is sufficient to inline the `KeccakF1600_StateXORBytes` subroutine into the `keccak_inc_absorb` function, as mentioned in [JMD24]. For the absorption abort attack, this would involve unrolling the loop during the absorption, which is possible because the input to the function during the computation of the vinegar seed has fixed length. However, due to the resulting increase in code size and the need for a separate implementation that is able to handle dynamic length input in other parts of the algorithm, this countermeasure may be impractical.

A different approach is to increase the probability of the signing loop being repeated. The attacks fail if the faulty seed is overwritten in a second iteration of the signing loop. However, the failure probability of the signing (and thus the probability of executing the signing loop more than once) is very low with the current parameter sets. It may be possible to select parameters that deliberately increase the probability of signing failure to make it more difficult for an attacker to identify the correct iteration of the signing loop for fault injection. Assuming an attacker is only able to inject a fault into a limited number of iterations, this lowers the success probability at the cost of an increased runtime of the signing procedure. It is worth pointing out that the tentative round 2 parameter sets for MAYO recently proposed in [Beu24] increase the probability of the signing loop being repeated. However, the new repeat probabilities ($\leq 2^{-12}$) are still too low to effectively prevent the attacks presented in this paper.

In [JMD24] it is also suggested that implementations verify that the sponge is not empty after absorbing data, thereby protecting against the absorption skipping attack. However, the absorption abort attack can bypass this countermeasure by allowing a small amount of data to be absorbed. It may thus be better to compare the data in the sponge to the input data. Implementing such a check correctly may be nontrivial, especially when absorbing later blocks of input into a non-empty sponge, as is the case for larger inputs.

To protect against the absorption abort attack, which leaves the sponge only partially initialised, it is possible to reorder the arguments to the SHAKE256 function to make the first part of the input unknown to the attacker. In our attack, the absorption is aborted after filling the sponge with the first 128 input bits. Thus, without knowing the input bytes, an attacker cannot enumerate the contents of the sponge. However, our attack targets the second iteration of the absorption loop, which absorbs 64 bits in each iteration. By instead targeting the first iteration, an attacker may be able to reduce the number of bits in the sponge. Additionally, other implementations may process fewer bits per iteration, thereby also reducing the search space for an attacker.

9.2 Argument initialisation skipping attack on SHAKE256 via `memcpy`

To protect against the argument initialisation attack, it is possible to use the incremental variant of the SHAKE256 function already found in the implementation [BCC+24], which splits the hash computation into several functions. This allows the `shake256_inc_absorb` function to be used, which can be called multiple times with different buffers, eliminating the need to copy the secret seed seed_{sk} to a common buffer. Alternatively, instead of zeroing buffers with sensitive information at the end of the signing, overwriting these buffers with random data would also prevent the attack.

9.3 Other countermeasures

For the sake of completeness, we also mention that selecting parameter sets with $o > k$ prevents the key recovery method presented in Section 7 from being used. However, this is not an effective countermeasure in general, because key recovery techniques, such as [ACK+23, Péb24], are not affected by setting $o > k$. As mentioned previously, those techniques can also be used for key recovery for the fault attacks in this paper.

Finally, we experimented with inserting random delays into the execution of the algorithm to make it more difficult for an attacker to identify the right time for the fault injection. The random delays were implemented as a buffer of `NOP`s into which the actual instructions were inserted at runtime with randomly chosen distances between them. However, we found that it is possible to reliably identify certain instructions based on reference power consumption waveforms during the execution of the algorithm, thereby breaking the countermeasure.

Additionally, the generation of a sequence of instructions with randomly inserted delays has a significant runtime overhead both for the randomisation, as well as the subsequent patching of relative branches to ensure the algorithm is executed correctly. There is also a significant memory and execution cost associated with this technique, especially if delays are chosen to be large enough to protect against attackers that can fault multiple instructions. Overall, the random delay insertion does not seem viable to protect against the attacks presented in this paper.

10 Conclusion

We presented three practical first-order single-execution fault injection attacks on an implementation of MAYO that can recover the full secret key of the scheme. Unlike previous work, two of our attacks are not limited by the memory allocation strategy of the device. The third requires a less restrictive memory allocation than previous attacks.

We introduced an alternative key recovery method that is simpler than previous techniques and can be used in cases where the vinegar seed is known. We also proposed a simulation technique that combines symbolic execution with loopy belief propagation on a factor graph to identify faults that allow an attacker to predict the vinegar seed.

Our work demonstrates that it is possible to recover the secret key of MAYO in a single attempt with a high probability, up to 99%. This highlights the importance of protecting the computations of seed values. Previous fault attacks on MAYO have focused on attacking the vinegar values derived from the seed instead of the seed itself.

Future work includes developing stronger countermeasures against fault attacks on implementations of PQC algorithms.

11 Acknowledgement

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation and by the Swedish Civil Contingencies Agency (Grant No. 2020-11632).

References

[ACK+23] Thomas Aulbach, Fabio Campos, Juliane Krämer, Simona Samardjiska, and Marc Stöttinger. Separating oil and vinegar with a single trace side-channel assisted Kipnis-Shamir attack on UOV. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):221–245, 2023. doi:10.46586/TCHES.V2023.I3.221-245.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*, volume 13269 of *Lecture Notes in Computer Science*, pages 853–871. Springer, 2022. doi:10.1007/978-3-031-09234-3_42.

[AKKM22] Thomas Aulbach, Tobias Kovats, Juliane Krämer, and Soundes Marzougui. Recovering Rainbow's secret key with a first-order fault attack. In Lejla Batina and Joan Daemen, editors, *Progress in Cryptology - AFRICACRYPT 2022*, pages 348–368, Cham, 2022. Springer Nature Switzerland. doi:10.1007/978-3-031-17433-9_15.

[AMSU24] Thomas Aulbach, Soundes Marzougui, Jean-Pierre Seifert, and Vincent Quentin Ulitzsch. MAYo or MAY-not: Exploring implementation security of the post-quantum signature scheme MAYO against physical attacks. In *Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2024, Halifax, NS, Canada, September 4, 2024*, pages 28–33. IEEE, 2024. doi:10.1109/FDTC64268.2024.00012.

[BBBP13] Alessandro Barenghi, Guido Marco Bertoni, Luca Breveglieri, and Gerardo Pelosi. A fault induction technique based on voltage underfeeding with application to attacks against AES and RSA. *J. Syst. Softw.*, 86(7):1864–1878, 2013. doi:10.1016/J.JSS.2013.02.021.

[BBPP09] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. Low voltage fault attacks on the RSA cryptosystem. In Luca Breveglieri, Israel Koren, David Naccache, Elisabeth Oswald, and Jean-Pierre Seifert, editors, *Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009, Lausanne, Switzerland, 6 September 2009*, pages 23–31. IEEE Computer Society, 2009. doi:10.1109/FDTC.2009.30.

[BCC$^+$23] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. MAYO, June 2023. URL: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/mayo-spec-web.pdf.

[BCC$^+$24] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. Nibbling MAYO: optimized implementations for AVX2 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):252–275, 2024. URL: https://doi.org/10.46586/tches.v2024.i2.252-275, doi:10.46586/TCHES.V2024.I2.252-275.

[BCD$^+$23] Ward Beullens, Ming-Shing Chen, Jintai Ding, Boru Gong, Matthias J. Kannwischer, Jacques Patarin, Bo-Yuan Peng, Dieter Schmidt, Cheng-Jhih Shih, Chengdong Tao, and Bo-Yin Yang. UOV: Unbalanced oil and vinegar, May 2023. URL: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/UOV-spec-web.pdf.

[BDPVA11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions, 2011. URL: https://keccak.team/files/CSF-0.1.pdf.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 41–46. USENIX, 2005. URL: http://www.usenix.org/events/usenix05/tech/freenix/bellard.html.

[Beu21] Ward Beullens. Improved cryptanalysis of UOV and Rainbow. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 348–373. Springer, 2021. doi:10.1007/978-3-030-77870-5_13.

[Beu22] Ward Beullens. MAYO: Practical post-quantum signatures from oil-and-vinegar maps. In Riham AlTawy and Andreas Hülsing, editors, *Selected Areas in Cryptography*, pages 355–376, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-030-99277-4_17.

[Beu24] Ward Beullens. MAYO: Overview + Updates. NIST PQC Seminar, September 2024. URL: https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/pqc-seminars/presentations/20-mayo-09242024.pdf.

[BFP19] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the glitch: Optimizing voltage fault injection attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):199–224, 2019. doi:10.13154/tches.v2019.i2.199-224.

[BPB10] Stanislav Bulygin, Albrecht Petzoldt, and Johannes Buchmann. Towards provable security of the Unbalanced Oil and Vinegar signature scheme under direct attacks. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010 - 11th International Conference on Cryptology in India, Hyderabad, India, December 12-15, 2010. Proceedings*, volume 6498 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2010. doi:10.1007/978-3-642-17401-8_3.

[CDSLN20] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. ProFIPy: Programmable software fault injection as-a-service. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 364–372, 2020. doi:10.1109/DSN48063.2020.00052.

[DHHB08] Ashish Darbari, Bashir Al Hashimi, Peter Harrod, and Daryl Bradley. A new approach for transient fault injection using symbolic simulation. In *2008 14th IEEE International On-Line Testing Symposium*, pages 93–98, 2008. doi:10.1109/IOLTS.2008.59.

[DYC+08] Jintai Ding, Bo-Yin Yang, Chia-Hsin Owen Chen, Ming-Shing Chen, and Chen-Mou Cheng. New differential-algebraic attacks and reparametrization of Rainbow. In Steven M. Bellovin, Rosario Gennaro, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security, 6th International Conference, ACNS 2008, New York, NY, USA, June 3-6, 2008. Proceedings*, volume 5037 of *Lecture Notes in Computer Science*, pages 242–257, 2008. doi:10.1007/978-3-540-68914-0_15.

[FKNT22] Hiroki Furue, Yutaro Kiyomura, Tatsuya Nagasawa, and Tsuyoshi Takagi. A new fault attack on UOV multivariate signature scheme. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography*, pages 124–143, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-17234-2_7.

[GJN20] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 359–386. Springer, 2020. doi:10.1007/978-3-030-56880-1_13.

[HGA+21] Florian Hauschild, Kathrin Garb, Lukas Auer, Bodo Selmke, and Johannes Obermaier. ARCHIE: A QEMU-based framework for architecture-independent evaluation of faults. In *18th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2021, Milan, Italy, September 17, 2021*, pages 20–30. IEEE, 2021. doi:10.1109/FDTC53659.2021.00013.

[HSP21] Max Hoffmann, Falk Schellenberg, and Christof Paar. ARMORY: fully automated and exhaustive fault simulation on ARM-M binaries. *IEEE Trans. Inf. Forensics Secur.*, 16:1058–1073, 2021. doi:10.1109/TIFS.2020.3027143.

[HTS11] Yasufumi Hashimoto, Tsuyoshi Takagi, and Kouichi Sakurai. General fault attacks on multivariate public key cryptosystems. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, pages 1–18, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-25405-5_1.

[JMD24] Sönke Jendral, John Preuß Mattsson, and Elena Dubrova. A single-trace fault injection attack on hedged module lattice digital signature algorithm (ML-DSA). In *Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2024, Halifax, NS, Canada, September 4, 2024*, pages 34–43. IEEE, 2024. doi:10.1109/FDTC64268.2024.00013.

[KCN21] Alec Kirkley, George T. Cantwell, and M. E. J. Newman. Belief propagation for networks with loops. *Science Advances*, 7(17):eabf1211, 2021. doi:10.1126/sciadv.abf1211.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. doi:10.1145/360248.360252.

[KL19] Juliane Krämer and Mirjam Loiero. Fault attacks on UOV and Rainbow. In Ilia Polian and Marc Stöttinger, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 193–214, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-16350-1_11.

[KS98] Aviad Kipnis and Adi Shamir. Cryptanalysis of the oil & vinegar signature scheme. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*, pages 257–266. Springer, 1998. doi:10.1007/BFB0055733.

[Lan22] Julien Lancia. Detecting fault injection vulnerabilities in binaries with symbolic execution. In *2022 14th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–8, 2022. doi:10.1109/ECAI54874.2022.9847500.

[LFBP24] Guilhem Lacombe, David Feliot, Etienne Boespflug, and Marie-Laure Potet. Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. *Journal of Cryptographic Engineering*, 14(1):147–164, April 2024. doi:10.1007/s13389-023-00310-8.

[MIS20] Koksal Mus, Saad Islam, and Berk Sunar. Quantumhammer: A practical hybrid attack on the LUOV signature scheme. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1071–1084, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372297.3417272.

[MTO24] Kit Murdock, Martin Thompson, and David F. Oswald. FaultFinder: Lightning-fast, multi-architectural fault injection simulation. In Chip-Hong Chang, Ulrich Rührmair, Jakub Szefer, Lejla Batina, and Francesco Regazzoni, editors, *Proceedings of the 2024 Workshop on Attacks and Solutions in Hardware Security, ASHES 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 78–88. ACM, 2024. doi:10.1145/3689939.3695788.

[Nat15] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical Report NIST FIPS 202, National Institute of Standards and Technology, Gaithersburg, MD, August 2015. doi:10.6028/NIST.FIPS.202.

[Nat23] National Institute of Standards and Technology. NIST announces additional digital signature candidates for the PQC standardization process, June 2023. URL: https://csrc.nist.gov/News/2023/additional-pqc-digital-signature-candidates.

[Nat24a] National Institute of Standards and Technology. Module-Lattice-Based Digital Signature Standard. Technical Report NIST FIPS 204, National Institute of Standards and Technology, Gaithersburg, MD, August 2024. doi:10.6028/NIST.FIPS.204.

[Nat24b] National Institute of Standards and Technology. Module-Lattice-Based Key Encapsulation Mechanism Standard. Technical Report NIST FIPS 203, National Institute of Standards and Technology, Gaithersburg, MD, August 2024. doi:10.6028/NIST.FIPS.203.

[Nat24c] National Institute of Standards and Technology. Stateless Hash-Based Digital Signature Standard. Technical Report NIST FIPS 205, National Institute of Standards and Technology, Gaithersburg, MD, August 2024. doi:10.6028/NIST.FIPS.205.

[ND15] Anh Quynh Nguyen and Hoang Vu Dang. Unicorn: Next generation CPU emulator framework. *BlackHat USA*, 476, 2015. URL: https://www.unicorn-engine.org/.

[O'F16] Colin O'Flynn. Fault injection using crowbars on embedded systems. *IACR Cryptol. ePrint Arch.*, page 810, 2016. URL: http://eprint.iacr.org/2016/810.

[Pat97] Jacques Patarin. The oil and vinegar signature scheme. In *Presented at the Dagstuhl Workshop on Cryptography September 1997*, 1997.

[Péb24] Pierre Pébereau. One vector to rule them all: Key recovery from one vector in UOV schemes. In Markku-Juhani O. Saarinen and Daniel Smith-Tone, editors, *Post-Quantum Cryptography - 15th International Workshop, PQCrypto 2024, Oxford, UK, June 12-14, 2024, Proceedings, Part II*, volume 14772 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2024. doi:10.1007/978-3-031-62746-0_5.

[PNKI13] Karthik Pattabiraman, Nithin M. Nakka, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. SymPLFIED: Symbolic program-level fault injection and error detection framework. *IEEE Transactions on Computers*, 62(11):2292–2307, 2013. doi:10.1109/TC.2012.219.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 513–533. Springer, 2017. doi:10.1007/978-3-319-66787-4_25.

[Risnd] Riscure. Riscure FiSim. https://github.com/Keysight/FiSim, n.d.

[RRB$^+$19] Prasanna Ravi, Debapriya Basu Roy, Shivam Bhasin, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Number "not used" once - practical fault attack on pqm4 implementations of NIST candidates. In Ilia Polian and Marc Stöttinger, editors, *Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3-5, 2019, Proceedings*, volume 11421 of *Lecture Notes in Computer Science*, pages 232–250. Springer, 2019. doi:10.1007/978-3-030-16350-1_13.

[SK20] Kyung-Ah Shim and Namhun Koo. Algebraic fault analysis of UOV and Rainbow with the leakage of random vinegar values. *IEEE Transactions on Information Forensics and Security*, 15:2429–2439, 2020. doi:10.1109/TIFS.2020.2969555.

[SMA+24] Oussama Sayari, Soundes Marzougui, Thomas Aulbach, Juliane Krämer, and Jean-Pierre Seifert. HaMAYO: A fault-tolerant reconfigurable hardware implementation of the MAYO signature scheme. In Romain Wacquez and Naofumi Homma, editors, *Constructive Side-Channel Analysis and Secure Design - 15th International Workshop, COSADE 2024, Gardanne, France, April 9-10, 2024, Proceedings*, volume 14595 of *Lecture Notes in Computer Science*, pages 240–259. Springer, 2024. doi:10.1007/978-3-031-57543-3_13.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (State of) The Art of War: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 138–157. IEEE Computer Society, 2016. doi:10.1109/SP.2016.17.

[YFW00] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Generalized belief propagation. In Todd K. Leen, Thomas G. Dietterich, and Volker Tresp, editors, *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA*, pages 689–695. MIT Press, 2000. URL: https://proceedings.neurips.cc/paper/2000/hash/61b1fb3f59e28c67f3925f3c79be81a1-Abstract.html.