Practical End-to-end Fault Attacks on PERK

Tanguy Stekke[1], Durba Chatterjee[2] and Lejla Batina[2]

[1] Université libre de Bruxelles, Bruxelles, Belgium,
[2] Radboud University, Nijmegen, Netherlands

Abstract. This paper presents the first practical end-to-end fault injection attacks on the post-quantum signature scheme PERK, based on the MPC-in-the-Head paradigm and relies on GGM tree expansions for efficient randomness generation. While GGM trees reduce memory requirements, they introduce implementation-level deviations from the theoretical model. We show that these implementation choices fundamentally alter the fault surface and enable new attack vectors that are not captured by the original security assumptions. We propose two attacks targeting distinct stages of the GGM tree construction. The first attack fixes the root seed, resulting in deterministic tree generation and enabling full reconstruction of all leaf labels. The second attack induces reuse of GGM roots via an instruction skip, allowing recovery of hidden leaf values across consecutive rounds. Both attacks require only a single fault to recover the secret key.
We demonstrate both attacks end-to-end on the `pqm4` reference implementation compiled with optimization levels `-O3` as well as `-Os` for ARM Cortex-M4 microcontrollers. Our evaluation is performed on two hardware platforms (ChipWhisperer-Lite with STM32F303 and Nucleo-L4R5ZI-P) using clock glitching and electromagnetic fault injection (EMFI). We achieve success probabilities of 100% and 85% for the two attacks, respectively. Finally, we propose countermeasures for both attacks.

Keywords: Fault Injection Attack, Post-Quantum Cryptography, MPCitH, PERK

1 Introduction

The proliferation of Internet-of-Things (IoT) devices and embedded systems has significantly increased the demand for secure and efficient authentication mechanisms. Digital signatures serve as a fundamental building block in achieving this goal. However, the constrained nature of such devices imposes strict limitations on the choice of cryptographic primitives. Classical schemes such as Elliptic Curve Digital Signature Algorithm (ECDSA) [JMV01] have long been favored due to their efficiency. At the same time, deployments in embedded settings often assume that the adversary has physical proximity, making implementations vulnerable to physical attacks such as side-channel analysis (SCA) and fault injection (FI). The susceptibility of classical cryptographic implementations to physical attacks has been extensively explored with numerous works demonstrating practical key-recovery [JD25, SJDN26, MAKK24]. These findings have led to a mature understanding of implementation-level vulnerabilities and the development of corresponding countermeasures.

As the cryptographic community transitions toward post-quantum cryptography (PQC), similar concerns naturally arise regarding the physical security of newly proposed schemes. This transition has been largely driven by the standardization efforts of the National Institute of Standards and Technology (NIST), culminating in the selection of ML-DSA [oSND+24] and ML-KEM [oSN24a] in August 2024. Following this milestone, NIST initiated a second call for PQC digital signature schemes [oSN24b], aiming to diversify the design space and explore alternative constructions based on different hardness assumptions.

A key requirement in this phase is the development of lightweight signature schemes suitable for constrained environments.

Among the emerging approaches, MPC-in-the-Head (MPCitH) based signature schemes have gained significant attention. These schemes rely on zero-knowledge proofs of different underlying mathematical problems. They use the Fiat–Shamir transform [FS86] to transform interactive zero-knowledge proofs into non-interactive digital signatures. Concretely, the prover simulates a multi-party computation protocol locally, commits to the views of virtual parties, and selectively reveals subsets of these views to convince the verifier. Recent candidates in the second round, including FAEST [BBB+23], Mirath [AAB+23], MQOM [BBFR25], PERK [ABB+25], RYDE [ABB+23], and SDitH [AMBB+23], instantiate this paradigm using different underlying *hard* mathematical problems.

MPCitH-based signatures offer advantages such as compact public keys. These benefits, however, come at the cost of large signature sizes and computationally expensive signing. Furthermore, practical implementations incur significant overhead from randomness expansion and state management. To address these challenges, existing works employ optimizations such as Goldreich-Goldwasser-Micali (GGM) tree constructions and more advanced paradigms like Threshold-Computation-in-the-Head (TCitH) and VOLE-in-the-Head (VOLEitH). Notably, all current second-round candidates incorporate TCitH and/or VOLEitH variants, with PERK being among the most recent to adopt VOLEitH.

At the same time, the complexity and structure of MPCitH-based constructions introduce new attack surfaces at the implementation level. Given the realistic assumption of physically capable adversaries in embedded settings, evaluating these schemes solely in terms of computational efficiency is insufficient. Their resilience against physical attacks must also be carefully examined.

In this work, we analyze the `pqm4` implementation of PERK v2.0.0, a digital signature scheme based on the Permuted Kernel Problem (PKP) and the MPCitH paradigm, and investigate its susceptibility to fault injection attacks. While PERK incorporates advanced design techniques such as VOLEitH, we show that implementation-level optimizations introduce subtle deviations from the theoretical model. In particular, intermediate values that are assumed to be persistent in the specification are recomputed on demand in practice to reduce memory usage. This discrepancy alters the execution flow and exposes new fault surfaces that are not captured by the original security assumptions. While prior works have identified potential fault vulnerabilities in MPCitH-based schemes, no prior work demonstrates practical end-to-end fault injection attack on PERK implementation.

In this paper, we bridge this gap by presenting the first end-to-end fault injection attack on the `pqm4` implementation of PERK. We demonstrate it on two hardware platforms (ChipWhisperer-Lite with STM32F303 and Nucleo-L4R5ZI-P) using two distinct techniques for fault injection i.e. clock glitching and EMFI. Our experiments demonstrate that the attacks work with a substantial success probabilities.

1.1 Contributions

Our main contributions are as follows:

1. **Practical Fault Injection Attacks on PERK.** We present two novel fault injection attacks targeting different stages of the GGM tree construction in PERK. The first attack forces the master seed to a known value by skipping the `randombytes` call, rendering the entire tree deterministic. The second attack targets the second response phase by skipping a PRG call, inducing collisions between GGM tree roots and enabling reconstruction of hidden values from partial openings.

2. **Single-Fault Key Recovery.** We develop efficient key-recovery procedures that exploit the resulting faulty signatures. Both attacks require only a single fault per

signature to recover the secret key. The recovery algorithms are up to 37 and 3.4 times faster than the signature scheme.

3. **End-to-End Experimental Validation.** We experimentally validate our attacks on the `pqm4` implementation of PERK on ARM Cortex-M4 microcontrollers. Our evaluation spans two hardware platforms and two fault injection techniques, namely clock glitching and electromagnetic fault injection. The first and second attacks reached success rates of 100% and 85% using EMFI, compared to 10% and 75% using clock glitching.
4. **Countermeasures and Overhead Analysis.** We propose practical countermeasures to mitigate the identified vulnerabilities and evaluate their implementation overhead on constrained devices.

Organization. Section 2 introduces the necessary background, and Section 3 reviews the related work on fault attacks on PQC schemes in the MPCitH paradigm. Section 4 presents the proposed fault attacks on the GGM tree construction in PERK. Section 5 provides end-to-end experimental validation on two platforms using clock glitching and EMFI. Countermeasures are presented in Section 6, and Section 7 concludes the paper.

2 Background

This section presents a brief overview of the PERK signature scheme, its core building blocks, and the Permuted Kernel Problem (PKP) that underpins its security.

Notation. Scalars are denoted by lowercase letters (e.g., q), vectors by bold lowercase letters (e.g., $\mathbf{x}$), and matrices by bold uppercase letters (e.g., $\mathbf{H}$). We denote by $\mathbb{F}_q$ a finite field of order q. For a positive integer n, we write $[n] = \{1, \ldots, n\}$. Given a vector $\mathbf{x} \in \mathbb{F}_q^n$ and a permutation $\pi \in S_n$, we denote by $\pi[\mathbf{x}]$ the permuted vector obtained by reordering the coordinates of $\mathbf{x}$ according to π. The symmetric group on n elements is denoted by S_n. For $i \in [t]$, $\mathbf{x}_i \in \mathbb{F}_q^n$ denotes input vectors and $\mathbf{y}_i \in \mathbb{F}_q^m$ the corresponding outputs.

In the context of PERK, τ is the number of GGM trees, and N the number of leaves in each tree or the number of parties in the MPC-in-the-Head protocol. We denote by $\theta^{(e)}$ the root of the e-th tree, where the superscript $e \in [\tau]$ indexes the tree. The corresponding leaf nodes are denoted by $\theta_i^{(e)}$, where $i \in [N]$ indexes the leaves of the tree.

2.1 The Permuted Kernel Problem

The security of PERK is relies on the Inhomogeneous Permuted Kernel Problem (IPKP), a variant of the classic PKP problem [Sha89]. Given a public matrix $\mathbf{H} \in \mathbb{F}_q^{m \times n}$ and a set of t input vectors, $\mathbf{x}_i \in \mathbb{F}_q^n$. The goal is to find the secret permutation $\pi \in S_n$ where S_n is the group of permutations of the set $[n]$ such that:

$$\mathbf{H}(\pi[\mathbf{x}_i]) = \mathbf{y}_i, \quad \forall\, i \in [t].$$

In PERK, the secret key is the seed that is used to sample the permutation π, while the public key consists of the seed `pk_seed` that is used to sample matrix $\mathbf{H}$ as well as $\mathbf{y}_i, i \in [t]$.

A relaxed variant of this problem, referred to as r-IPKP, is used to enable more efficient constructions. In this variant, the permutation is not required to satisfy the above relation for every individual pair $(\mathbf{x}_i, \mathbf{y}_i), i \in [t]$. Instead, it suffices that the relation holds for a non-trivial linear combination of the $(\mathbf{x}_i, \mathbf{y}_i)$ pairs. This relaxation reduces the computational complexity of the problem while preserving its hardness for cryptographic use.

2.2 Seed Derivation and GGM Trees

Goldreich-Goldwasser-Micali (GGM) trees are used to distribute the secret shares among x virtual parties. These trees are generated from top to bottom with a single τ-bit master seed (`mseed`), which is used to generate all randomness for the τ rounds of the protocol, each randomness for a different GGM tree. Each node in the tree construction has two child nodes except for the leaves. The steps involved in the tree construction are as follows:

- *Root Seed Generation*: From `mseed`, and a public salt, a Pseudo-Random Generator (PRG) derives τ tree root seeds:

$$(\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(\tau)}) \leftarrow \text{PRG}(\texttt{salt}, \texttt{mseed})$$

- *GGM Tree Expansion*: For each tree $e \in \{1, \ldots, \tau\}$, the root seed $\theta^{(e)}$ is expanded into N leaf seeds $\{\theta_0^{(e)}, \ldots, \theta_{N-1}^{(e)}\}$.

$$\{\theta_1^{(e)}, \ldots, \theta_N^{(e)}\} = \text{TreePRG}(\texttt{salt}, \theta^{(e)})$$

- *Leaf Expansion*: Each leaf seed $\theta_i^{(e)}, i \in \{1, \cdots, N\}$ is then fed to another PRG to produce the specific shares for the ith virtual party.

$$(\pi_i^{(e)}, \mathbf{v}_i^{(e)}) \leftarrow \text{PRG}(\texttt{salt}, \theta_i^{(e)})$$

  where,

  - $\pi_i^{(e)}$ (the permutation share): This is the ith component shared with virtual party i, to be used in the signature and verification algorithms.
  - $\mathbf{v}_i^{(e)}$ (the blinding vector): This vector provides the randomness used to mask intermediate computations in the scheme.

For further details, we refer the reader to [ABB+25]. From this construction, two interesting properties emerge. The first one is that if a root seed $\theta^{(e)}$ is known, then all subsequent values in that round become deterministic. This means that the secret key π can be recovered through reconstruction and chain composition. The second property concerns two distinct rounds $e, e' \in [1, \tau]$ with $e \neq e'$ if their root seeds coincide, i.e., $\theta^{(e)} = \theta^{(e')}$ then both GGM trees can be expanded, producing identical leaf sequences. The signature reveals all except one leaf from the partial opening for both, allowing an attacker to recover as well the secret permutation π.

2.3 The PERK Signature Scheme

The PERK signature scheme consists of three algorithms: KeyGen, Sign, and Verify. It is built from a five-round zero-knowledge proof of knowledge (ZNPOK) for r-IPKP, repeated τ times in parallel. The scheme uses Fiat-Shamir (FS) transformation to make the proof non-interactive.

2.3.1 Key Generation

The key generation algorithm generates an output pair of keys (`pk`, `sk`). The key `sk` is the private signing key used for signing, and `pk` is the public key used by the verifier. The first step involves generating two seeds (`pk_seed` and `sk_seed`) using a PRG. The secret permutation π is derived via PRG from `sk_seed`. `pk_seed` is used to sample the matrix $\mathbf{H} \in \mathbb{F}_q^{m \times n}$ and the input vectors $\mathbf{x}_i \in \mathbb{F}_q^n$ for $i \in [t]$. The corresponding outputs are then computed using the secret permutation as $\mathbf{y}_i = \mathbf{H}(\pi[\mathbf{x}_i])$ for all $i \in [t]$, so that $(\mathbf{H}, \mathbf{x}_i, \mathbf{y}_i)$ forms a valid IPKP instance. The public key consists of `pk_seed` (from which $\mathbf{H}$ and the $\mathbf{x}_i$ are regenerated) together with the $\mathbf{y}_i$, while the secret key is `sk_seed`.

2.3.2 Signing

To sign a message m, the signing algorithm requires a secret signing key as input. The process begins with the signer sampling a fresh random salt and a master seed `mseed`. Notably, to render the proof non-interactive, the signer locally simulates the roles of both the Prover and the Verifier. At a high level, the scheme proceeds through the following main steps:

1. *Commitment.* The Prover samples a salt and master seed to derive party seeds $\theta_i^{(e)}$ via a GGM tree. For each iteration $e \in [\tau]$, the Prover derives shares of the secret permutation π and vectors $\mathbf{v}_i^{(e)}$ using the PRG. These are used to compute a combined vector $v^{(e)}$ and its syndrome $\mathbf{Hv^{(e)}}$. The Prover commits to these values by sending h_1, which is a hash of all individual party commitments $cmt_{1,i}^{(e)}$ and the syndrome commitments $cmt_1^{(e)}$.

2. *First challenge.* The Verifier provides challenges $(\kappa_1^{(e)}, \ldots, \kappa_t^{(e)})$. These act as coefficients for a linear combination of the public key values x_j that will be used to compute the simulated initial state $s_0^{(e)}$.

3. *First response.* The Prover computes the internal states of the N parties. For each $i \in [N]$, the Prover computes $s_i^{(e)} = \pi_i^{(e)}[s_{i-1}^{(e)}] + v_i^{(e)}$. The first response h_2 is sent, which is a hash of the first challenge and all simulated states $s_i^{(e)}$.

4. *Second challenge.* The Verifier sends a second challenge consisting of indices $\alpha^{(e)} \in [N]$, selecting one party's view to remain hidden in each iteration.

5. *Second Response.* For each iteration, the Prover sends $rsp^{(e)}$, which includes the state $s_\alpha^{(e)}$, the commitment $cmt_{1,\alpha}^{(e)}$, and the seeds θ_i for all $i \neq \alpha$ (allowing the Verifier to reconstruct all views except the hidden one). If $\alpha = 1$, the Prover specifically omits π_1. The Verifier then checks the consistency of the reconstructed views and verifies that they hash to the values provided in h_1 and h_2.

2.3.3 Verifying

Given a message m, public key pk and a signature σ, the verifier re-derives the challenge coefficient from commitment h_1 and the hidden indices from h_2. For each iteration e, the verifier reconstructs all party shares except the hidden one (denoted by $z_1^{(e)}$) from the partial seed opening and uses it for the hidden party's value. Then recomputes all commitments and both challenge hashes h_1 and h_2. If both values are correct, the final output of the algorithm is one otherwise it returns zero.

2.4 MPC-in-the-Head and Compositional Sharing

PERK is a digital signature scheme built on the zero-knowledge proof of knowledge system and adopts the MPC-in-the-Head paradigm. The prover simulates a multi-party computation protocol in its head, commits to the views of all virtual parties, and then selectively reveals a subset of those views to the verifier. To prove knowledge of the secret permutation (π), this is achieved by splitting it into N shares $\{\pi_0^{(e)}, \ldots, \pi_{N-1}^{(e)}\}$ for each round e such that:

$$\pi = \pi_{N-1}^{(e)} \circ \pi_{N-2}^{(e)} \circ \cdots \circ \pi_0^{(e)}$$

The zero-knowledge property is achieved because any subset of $N-1$ revealed views does not reveal the secret witness. In the signature, the verifier challenges the prover to open $N-1$ out of N parties. If the hidden index is $\alpha^{(e)}$, the prover reveals the seeds for

Table 1: Security vulnerability analysis of PERK implementations against Fault Injection Attacks.

Target Version	Reference	Attack Vector / Component	Status
NIST v2.2.0 (VOLEitH)	[SJDN26]	Root Fixing using Keccak Permutation	Theoretical
NIST v2.1.0 (VOLEitH)	[JD26]	Root Fixing	Theoretical
	[BBK25]	Forgery in the second response	Theoretical
	[BBK25]	GGM tree expansion skip in commitment	Theoretical
	[BBK25]	AES skip in the commitment	Theoretical
pqm4 (MPCitH)	**This Work**	**Root fixing using `randombytes`**	**Practical**
	This Work	**Node Collision in second response**	**Practical**

all leaves $i \neq \alpha^{(e)}$. If $\alpha^{(e)} \neq 0$ the masked share $\pi_0^{(e)}$ is explicitly included in the signature, the remaining shares $\pi_i^{(e)}$ are reconstructed from the revealed seeds.

3 Related Works

MPC-in-the-Head (MPCitH) signature schemes are relatively recent, but their physical security has already been studied in several works. The focus has so far been mostly on active attacks such as fault injection, but most of previous works investigated the schemes vulnerabilities theoretically or with a ChipWhisperer setup. We summarize the most relevant ones in Table 1 and describe the attack targets briefly as follows.

Fault attacks on zero-knowledge-based signatures. Mondal et al. [MAKK24] study fault injection attacks on zero-knowledge-based signature schemes. Their focus is on the seed reveal function used in tree-based share expansion. They show that faults during seed expansion can lead to the recovery of intermediate node values, including hidden leaves. This ultimately enables full secret key recovery. The analysis is applied to code-based schemes such as LESS and CROSS. The attack was demonstrated using fault simulation, and does not justify the practical feasibility.

Fault attacks on PERK targeting SHA-3. Several works consider fault attacks on SHA-3. Jendral et al. [SJDN26] show that a single instruction skip inside the Keccak permutation reduces the entropy of the sponge state. This enables key recovery with a residual uncertainty of 2^{40} in the NIST v2.2.0 implementation of PERK-5.

Fault attacks on MPCitH and VOLEitH schemes. Jendral and Dubrova [JD26] analyze the physical security of a VOLEitH-based scheme using masked FAEST on ARM Cortex-M4 as a case study. They present two side-channel attacks and two fault injection attacks. The fault attacks target GGM tree expansion and VOLE correlation generation. Both attacks enable full secret key recovery from a single signature with probability above 87%. The authors also discuss possible implications for PERK, but do not provide an experimental evaluation on its implementation.

Banda et al. [BBK25] propose generic fault attacks on MPCitH schemes in the second round of the NIST process. Their key-recovery attacks target GGM tree seed generation through instruction skipping, while their forgery attack targets the Fiat–Shamir challenge generation. The attacks are validated on Mirath and RYDE. For PERK, they identify several applicable fault vectors and demonstrate a forgery attack on version 2.1.0 of the reference implementation. However, no end-to-end key-recovery attack is experimentally demonstrated for PERK.

Our position. We present an end-to-end fault injection analysis of the `pqm4` implementation of PERK which is the equivalent of NIST version 2.0.0. We target the MPCitH-based construction and demonstrate two practical fault attacks. As summarized in Table 1, prior works either remain theoretical for PERK or focus on partial attack vectors or reference implementations. In contrast, our work provides the first experimental end-to-end key-recovery attacks on a concrete embedded implementation of PERK. Both attacks require a single fault per signature and are validated on ARM Cortex-M4 platforms.

4 Fault Attacks on PERK

We present two practical fault injection attacks on the `pqm4` implementation of the PERK signature scheme. Both attacks target the randomness generation and GGM tree expansion phases, and enable full recovery of the secret permutation π from a single faulty signature. In this section we outline both attacks in detail and we present our algorithms for key recovery in both cases (Algorithm 1 and Algorithm 2) and we present experimental results in Section 5.

Attack Overview and Fault Injection Model. We consider an adversary capable of injecting precisely timed faults during the execution of the signing algorithm. These faults are transient, enabling an adversary to skip targetted instructions including branch instructions or function calls, such as randomness generation or PRG expansion, and observe the resulting faulty signatures. Instruction-skip fault models are widely used in the physical security evaluation of embedded cryptographic implementations, and have been shown to enable practical attacks on cryptographic implementations [JD25, SJDN26]. Our evaluation follows the same standard fault injection assumptions and demonstrates their applicability to MPC-in-the-Head based signature schemes.

Both attacks exploit the deterministic structure of the GGM tree construction. By injecting a fault at carefully chosen points, the adversary either: (i) forces the tree roots to a known value, or (ii) induces collisions between tree roots across different rounds. In both cases, the faulty execution leaks sufficient information to reconstruct the secret permutation π.

4.1 Attack 1: Fixed Root Seed via Randomness Skipping

Intuition. The attack forces the master seed to a known constant by skipping the randomness generation. As a result, all GGM tree roots become deterministic and known to the attacker, allowing full reconstruction of all leaf seeds and permutation shares. Figure 1 depicts the resultant tree schematic in such a scenario. Here with the information of $\theta^{(0)}$, an adversary can compute all the leaves. The fault is injected in the generate commitment step of the signing algorithm.

In the PERK signing algorithm, the GGM tree construction relies on the unpredictability of the master seed `mseed`. This seed is generated using the `randombytes` function and used to derive the tree roots via a PRG. We refer the reader to Listing 1 that provides a simplified code for the first step of signature generation, which is generating the commitment.

Fault Model and Target. We target the call to `randombytes`. By skipping this instruction, the buffer `rand_bytes` remains in its zero-initialized state. Consequently, `mseed` is fixed to a known constant. This is possible only because in the implementation of the signature algorithm, the `rand_bytes` variable is initialized to zero before being assigned a pseudorandom value. Skipping the zero-initialization prevents this attack.

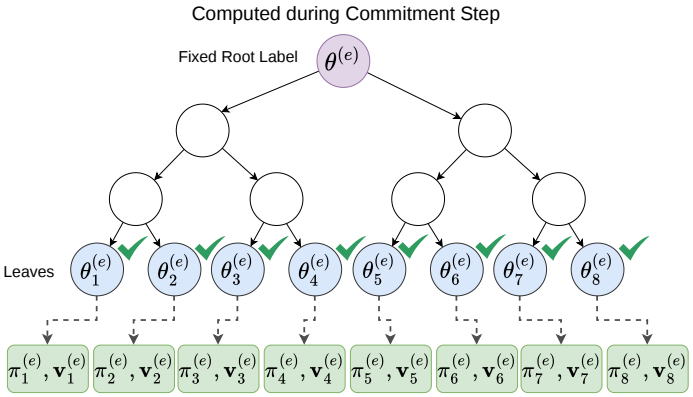

Figure 1: GGM tree schematic after Attack 1. The root seed is fixed to a known value, allowing full reconstruction of all leaf nodes and recovery of the hidden share.

The tree roots are then deterministically derived as:

$$\theta^{(e)} \leftarrow \mathrm{PRG}(\mathtt{salt}, \mathtt{mseed}) \tag{1}$$

Impact on GGM Tree. Since `mseed` is known, all τ tree roots $\theta^{(e)}$ are also known. Each root expands into N leaf seeds via:

$$\{\theta_1^{(e)}, \ldots, \theta_N^{(e)}\} = \mathrm{TreePRG}(\mathtt{salt}, \theta^{(e)}) \tag{2}$$

The corresponding permutation and vector shares are generated as:

$$(\pi_i^{(e)}, v_i^{(e)}) \leftarrow \mathrm{PRG}(\mathtt{salt}, \theta_i^{(e)}) \tag{3}$$

Thus, the entire tree structure becomes deterministic and fully known to the adversary.

Key Recovery. Given a faulty signature, the adversary recomputes all leaf seeds and derives the corresponding permutation shares. Using the revealed share from the response, the secret permutation π is recovered by composing the shares as described in Algorithm 1.

Algorithm 1: Proposed Secret Key Recovery via Fixed Root Seed

Input: Signature: $\sigma = (\mathtt{salt}, h_1, h_2, \mathtt{rsp}^{(\mathtt{e})}{}_{e\in[1,\tau]})$
Output: Secret Permutation: π_{rec}

$\mathtt{mseed} \leftarrow 0$
$\theta_{(fixed)} \leftarrow \mathrm{PRG}(\mathtt{salt}, \mathtt{mseed})$
$(\theta_i^{(1)})_{i\in[1,N]} \leftarrow \mathrm{TreePRG}(\mathtt{salt}, \theta_{(fixed)})$
for $i \leftarrow 1$ **to** N **do**
  $(\pi_i^{(1)}, v_i^{(1)}) \leftarrow \mathrm{PRG}(\mathtt{salt}, \theta_i^{(1)})$
$\pi_1^{(1)} \leftarrow z_2^{(1)}$
$\pi_{rec} \leftarrow \pi_1^{(1)}$
for $i \leftarrow 2$ **to** N **do**
  $\pi_{rec} \leftarrow \pi_i^{(1)} \circ \pi_{rec}$
return π_{rec}

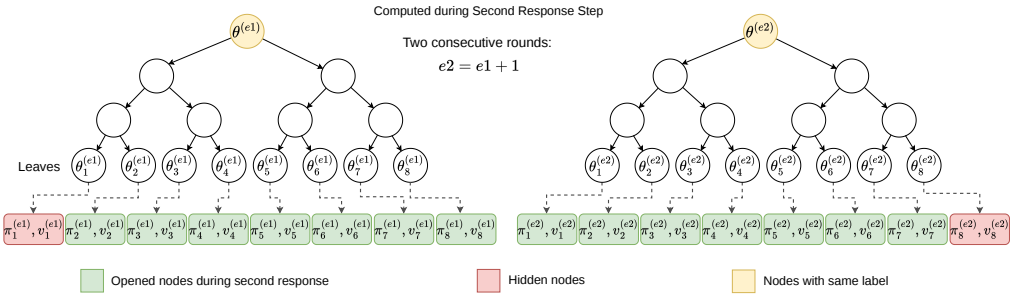

Figure 2: Root collision attack. Two consecutive roots are forced to be identical, allowing reconstruction of all leaf nodes by combining partial openings.

4.2 Attack 2: Node Collision via PRG Skipping

Intuition. This attack induces a collision between two GGM tree roots across different rounds in the second response step of the signing algorithm. As a result, two independent executions produce identical tree expansions. Since each round reveals all but one leaf, combining the partial openings from both rounds allows the adversary to recover all leaf seeds and reconstruct the secret permutation.

Fault Model and Target. In PERK, all tree roots $(\theta^{(e)})_{e\in[1,\tau]}$ are derived sequentially from a PRG initialized with the master seed:

$$(\theta^{(e)})_{e\in[1,\tau]} \leftarrow \mathrm{PRG}(\texttt{salt}, \texttt{mseed})$$

In the `pqm4` implementation, tree roots are recomputed on-the-fly during the second response phase. We target the call to the PRG function responsible for generating $\theta^{(e)}$. By skipping this call at a specific iteration, the previously computed root is reused, resulting in:

$$\theta^{(e)} = \theta^{(e-1)}$$

Impact on GGM Tree. When two consecutive roots collide, the corresponding GGM trees expand identically:

$$\mathrm{TreePRG}(\texttt{salt}, \theta^{(e)}) = \mathrm{TreePRG}(\texttt{salt}, \theta^{(e-1)})$$

Each round reveals all leaf seeds except one, indexed by $\alpha^{(e)}$. If the hidden indices differ, i.e., $\alpha^{(e)} \neq \alpha^{(e-1)}$, the union of the two partial openings reveals all N leaf seeds. This enables full reconstruction of the tree.

Key Recovery. The adversary first reconstructs the partial trees from both faulty responses. One tree is used as a base, and its missing leaf is filled using the corresponding leaf from the second tree. This yields a complete set of leaf seeds:

$$\{\theta_i^*\}_{i\in[1,N]}$$

From these seeds, the permutation shares are recomputed:

$$(\pi_i^*, v_i^*) \leftarrow \mathrm{PRG}(\texttt{salt}, \theta_i^*)$$

Finally, the secret permutation π is recovered by composing the shares, as described in Algorithm 2.

Algorithm 2: Proposed Secret Key Recovery for PERK using two GGM trees with the same root

Input: Signature: $\sigma = (\texttt{salt}, h_1, h_2, \texttt{rsp}^{(\texttt{e})}{}_{e\in[1,\tau]})$,
Indices: $(e_1, e_2), e_2 = e_1 + 1, e_1 \in [1, \tau - 1]$
Output: Secret Permutation: π_{rec}

▷ Extract challenges for all rounds from the signature digests

$(\alpha^{(e)})_{e\in[1,\tau]} \leftarrow \text{ExpandChallenge}(h_1, h_2)$
if $\alpha^{(e_1)} = \alpha^{(e_2)}$ **then**
  return *Null*

▷ Expand both partial trees from their respective responses

$\theta_i^{(e_1)} \leftarrow \text{ExpandPartialTree}(\texttt{salt},\ rsp^{(e_1)}.z_{2,\theta}^{(e_1)},\ \alpha^{(e_1)})$
$\theta_i^{(e_2)} \leftarrow \text{ExpandPartialTree}(\texttt{salt},\ rsp^{(e_2)}.z_{2,\theta}^{(e_2)},\ \alpha^{(e_2)})$

▷ Merge by taking tree e_1 as base and filling its missing leaf from tree e_2

$$\theta_i^* \leftarrow \begin{cases} \theta_i^{(e_1)} & \text{if } i \neq \alpha^{(e_1)} \\ \theta_i^{(e_2)} & \text{if } i = \alpha^{(e_1)} \end{cases} \quad \forall\, i \in [1, N]$$

▷ Regenerate all compositional shares from the merged seed tree

for $i \leftarrow 1$ **to** N **do**
  $(\pi_i^*,\ v_i^*) \leftarrow \text{PRG}(\texttt{salt},\ \theta_i^*)$ from $S_n \times \mathbb{F}_q^n$

▷ Recover π_1 directly from the response

$\pi_1^* \leftarrow z_2^{(e_*)}$

▷ Compose the full permutation chain

$\pi_{rec} \leftarrow \pi_1^*$
for $i \leftarrow 2$ **to** N **do**
  $\pi_{rec} \leftarrow \pi_i^* \circ \pi_{rec}$ ▷ $\pi_{rec} = \pi_N^* \circ \cdots \circ \pi_2^* \circ \pi_1^*$
return π_{rec}

Success Probability. The attack fails only if both rounds hide the same leaf index, i.e., $\alpha^{(e)} = \alpha^{(e-1)}$. Since each $\alpha^{(e)}$ is sampled uniformly from $[N]$, this occurs with probability $1/N$. For PERK parameters, this corresponds to at most $1/32$ (for fast parameter set) and $1/256$ (for short parameter set), making the attack highly reliable in practice.

Remark. The attack can be realized with a single fault by targeting two consecutive iterations. Extending it to non-consecutive rounds would require multiple faults.

5 Experimental Evaluation of Fault Attacks

In this section, we describe the experimental setup, attacker model, and practical evaluation of the proposed attacks. To demonstrate their practicality, we implement the attacks using two fault injection techniques, namely clock glitching and electromagnetic fault injection (EMFI) on ARM Cortex-M4-based platforms. Table 2 summarizes our experimental results for two optimization levels `-O3` and `-Os` for ARM Cortex-M4 microcontrollers.

5.1 Experimental Setup

The attacks target the `Perk-128-fast-3` reference implementation from the `pqm4` repository [1] with commit id `776f31d`. The implementation is written in C and compiled with `arm-none-eabi-gcc` using the `-O3` and `-Os` optimization level for ARM Cortex-M4. Table 3 summarizes the wall-clock execution times of the `KeyGen`, `Sign`, and `Verify` routines on the evaluated platforms. We evaluate the attacks using two independent fault injection setups to demonstrate reproducibility across platforms and techniques.

Table 2: Performance and success rates of key-recovery attacks using EMFI and clock glitching. The number of faults required is shown in parentheses, and success rates are given in square brackets.

Impl.	**Attack 1: Root Seed**			**Attack 2: Node Collision**		
	EMFI	**Clock Glitching**		**EMFI**	**Clock Glitching**	
	`-O3`	`-Os`	`-O3`	`-O3`	`-Os`	`-O3`
`pqm4`	(1) [100%]	(1) [100%]	(1) [10%]	(1) [85%]	(1) [10%]	(1) [75%]
NIST v2.0.0	(1)	(1)	(1)	(2)	(2)	(2)

Table 3: Number of faults in parenthesis. Execution times (in seconds) of `KeyGen`, `Sign`, and `Verify` algorithms on ARM Cortex-M4 platforms, compiled with `-O3` optimization level.

Operation	**STM32F303 (7.37 MHz)**	**Nucleo-L4R5ZI-P (32 MHz)**
KeyGen	0.14s	0.04s
Sign	36.1s	13.5s
Verify	17.4s	5.7s

Clock Glitching Setup. Clock glitching experiments are conducted on an STM32F303 microcontroller using the ChipWhisperer-Lite platform (Figure 3). The STM32F303 can execute the full Perk-128-fast-3 signing algorithm due to its sufficient memory resources. The generated signature (8.18 KB) is buffered on-device and transmitted via the Simple-Serial V2 protocol. The target operates at 7.37 MHz. Thus the signing procedure takes approximately 35 seconds.

The ChipWhisperer glitch module is configured to inject faults by modifying the clock signal supplied to the target. In particular, the `clock_xor` mode produces a faulty clock by XORing the base clock with the generated glitch signal. The glitch is triggered using an external signal (`ext_single`) for precise synchronization with the target execution. The main parameters are glitch width, offset, and external offset. The width and offset are expressed as percentages of the clock period, controlling the duration and position of the glitch within a clock cycle. The external offset is specified in clock cycles and determines the delay between the trigger and the glitch injection.

EMFI Setup. EMFI experiments are performed on a Nucleo-L4R5ZI-P development board (ARM Cortex-M4) operating at 32 MHz. Notably, we use this board as this is a general-purpose development platform and is not specifically designed for physical security evaluation, to demonstrate the practicality of the attacks. The implementation is modified to include a GPIO trigger for precise synchronization with the fault injection setup. Faults are injected using a Keysight VC Glitcher and an electromagnetic transient probe, mounted on an XYZ positioning stage (as depicted in Figure 4). Communication with the target is performed over UART for input transmission and signature retrieval. The target is powered using a 3.3V power supply.

[1] https://github.com/mupq/pqm4/tree/master/crypto_sign/perk-128-fast-3/m4

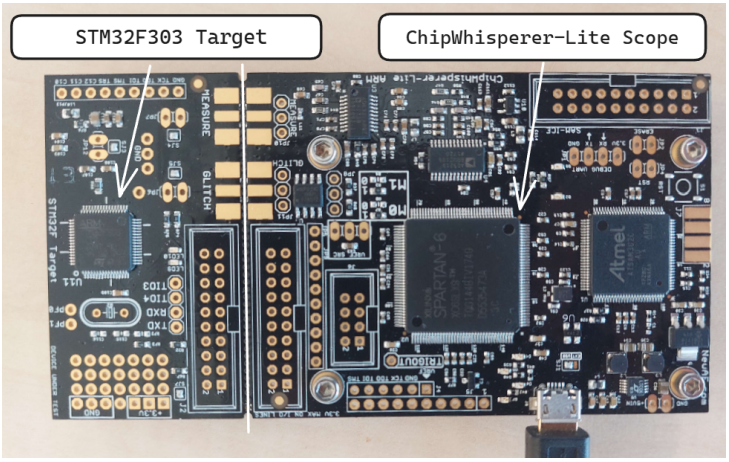

Figure 3: Clock glitching experimental setup using the ChipWhisperer-Lite platform with an STM32F303 target.

Due to the spatial precision required for EMFI, an initial characterization phase is performed to identify effective injection locations. We identified the X range to be 0 to 408 082 units, which defines the horizontal boundaries and the Y range to be 0 to 441 282 units, defining the vertical boundaries. Post characterization, we identified probe position (192 121.61, 244 313.86) to be the point of interest. This is relative to the top-leftmost point, labeled as the origin. Glitch repetition was set to 1, indicating that for each parameter and position, only one glitch was injected.

Attacker Model. We consider a gray-box adversary with full knowledge of the PERK algorithm and implementation, but no access to secret keys or internal randomness. The attacker can query the device for signatures and obtain corresponding faulty outputs.

The adversary is assumed to have physical access to the device and can inject precisely timed faults (e.g., clock glitches or EMFI) during execution. The goal is to exploit faulty signatures to recover sensitive intermediate values and ultimately extract the secret key.

5.2 Attack 1: Fixing Root Seed

The goal of this attack is to fix the root seeds by preventing proper initialization of the `rand_bytes` buffer. Listing 1 shows a simplified version of the commitment generation step in the signature algorithm. In the simplified version , we have removed the code lines

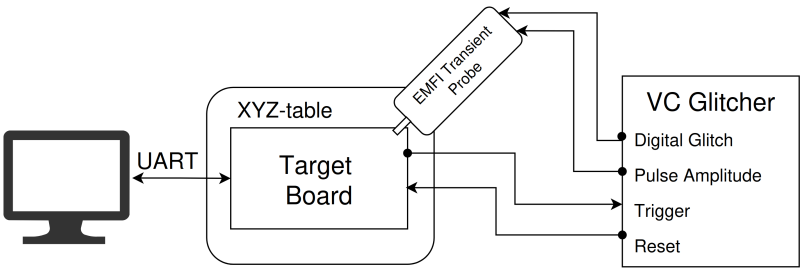

Figure 4: EMFI setup using STM32 Nucleo-144 as the target board

```
1  void sig_perk_gen_commitment(...){
2      uint8_t rand_bytes[SEED_BYTES + SALT_BYTES] = {0};
3      sig_perk_prg_state_t prg_state;
4
5      sig_perk_perm_inverse(pi_inv, pi);
6      randombytes(rand_bytes, sizeof(rand_bytes));
7      memcpy(mseed, rand_bytes, SEED_BYTES);
8      memcpy(signature->salt, rand_bytes + SEED_BYTES, SALT_BYTES);
9
10     sig_perk_prg_init(&prg_state, PRG1, signature->salt, mseed);
11
12     for (uint8_t e = 0; e < PARAM_TAU; ++e) {
13         perk_theta_seeds_tree_t theta_tree;
14         trigger_high();
15         sig_perk_prg(&prg_state, theta_tree[0], sizeof(theta_t));
16         trigger_low();
17     ...
18     }
19 }
```

Listing 1: Core implementation of the `pqm4` commitment function using `randombytes` to generate the master seed (non-essential variables omitted for clarity).

that do not contribute to the execution of the target operation, or the lines that are used later in the signature.

In the reference implementation, `rand_bytes` is first zero-initialized (Line 2) and then overwritten with fresh randomness via the `randombytes` call (Line 6). The buffer is subsequently split into `mseed` and `salt` (Lines 7–8), which are used to initialize the PRG state based on Keccak-128 [BDPA11] (Line 10). This corresponds to the operation defined in Equation (1). The initialized state is then used to generate the GGM trees as described in Equation (2), see Sec. 4.

Fault Target. We target the `randombytes` call (Line 6 in Listing 1). Skipping this call leaves `rand_bytes` in its zero-initialized state, resulting in fixed and deterministic root seeds across executions. Listing 2 shows the corresponding assembly. Register `r0` holds the pointer to `rand_bytes`, `r1` is set to zero, and `r2` specifies the buffer length (48 bytes: 16 bytes for `mseed` and 32 bytes for `salt`). The call to `memset` (Line 4) initializes the buffer to zero. The subsequent call to `PQCLEAN_randombytes` (Line 7) overwrites it with randomness. The attack consists of skipping the function call at Line 7. If the branch is skipped or the function returns prematurely due to a fault, `rand_bytes` remains zeroed and is used unchanged in subsequent steps.

Fault Effect. As a result, the root of the GGM tree becomes fixed across executions. This determinism propagates through the tree expansion, allowing the adversary to recompute hidden nodes once a single seed is revealed. Consequently, all secret shares π_i in the MPC-in-the-head protocol can be reconstructed.

Distinction from the Root-Seed Fixing Attack of [JD25]. In [JD25], describe a theoretical attack on the PERK signing scheme by fixing the root seeds of the GGM tree. They describe this as a potential extension of the attack on Keccak absorption step. In contrast, we show that the pqm4 implementation introduces an alternative fault

```
1    movs    r1, 0x0              ; memset value = 0
2    movs    r2, 0x30             ; size = 48 bytes
3    add     r0, sp, 0x38         ; r0 points to rand_bytes buffer
4    bl      memset               ; memset(rand_bytes, 0, 48)
5
6    add     r0, sp, 0x38         ; r0 = pointer to rand_bytes buffer
7    bl      PQCLEAN_randombytes  ; call randombytes(r0, r1)
```

Listing 2: Assembly code snippet of the random byte generation

surface through the initialization of `rand_bytes`. We perform the attack practically and demonstrate full key recovery. Notably, our Attack-1 and the attack discussed in [JD25] operate on different components of the signing procedure and rely on distinct fault injection vectors.

5.2.1 Fault Injection using Clock Glitching

Results for optimization level `-Os`. To identify a suitable fault injection window, we perform a characterization of glitch parameters over the device execution. Figure 5 shows the resulting heatmap over glitch width and offset. We identify a large contiguous region yields successful faults (for offsets from -48 to -40.5, for widths -12.5 and -12), and a smaller region with smaller width and relatively higher offset of -39 to -38. Offset and width are expressed as a percentage of a period where: $offset, width \in [-49.8, 49.8]$. Negative offset indicate that the glitch is at the end of the previous clock cycle, whereas a negative glitch width of -12 behaves as if they had a positive width on the other half of a clock cycle. [2] The external offset was of 13 as the trigger was set just before the call to the function.

Within this region, the attack achieves near 100% reliability across all tested parameter combinations. Using these parameters, we perform an end-to-end attack by generating signatures on the ChipWhisperer platform and recovering the full permutation from faulty outputs. Over 100 independent signatures, the secret key is successfully recovered in all cases.

Results for optimization level `-O3`. At optimization level -O3, the previously identified fault windows observed under -Os is no longer directly transferable with the same success rates previously identified. In particular, register allocation changes affect the alignment between the injected glitch and the target instruction sequence. This results in a mismatch between previously calibrated glitch parameters (width, offset and ext_offset). As a consequence, the original high-reliability region collapses. Through re-characterization, we identify a reduced but stable operation point at offset=33.9, width=−2 and ext_offset=11 where faults can still be induced reliably despite the altered instruction under -O3. Within this adjusted configuration, successful fault induction is reduced compared to -Os; however, it remains sufficient to recover a faulty signature every 10 signatures.

5.2.2 Fault Injection using EMFI

We complement the clock glitching experiments with electromagnetic fault injection (EMFI). A spatial cartography of the chip is first performed to identify sensitive regions. This phase yields an effective injection point at $(x, y, z) = (192\,121.61, 244\,313.86, 0)$. At this

[2] https://chipwhisperer.readthedocs.io/en/latest/scope-api.html

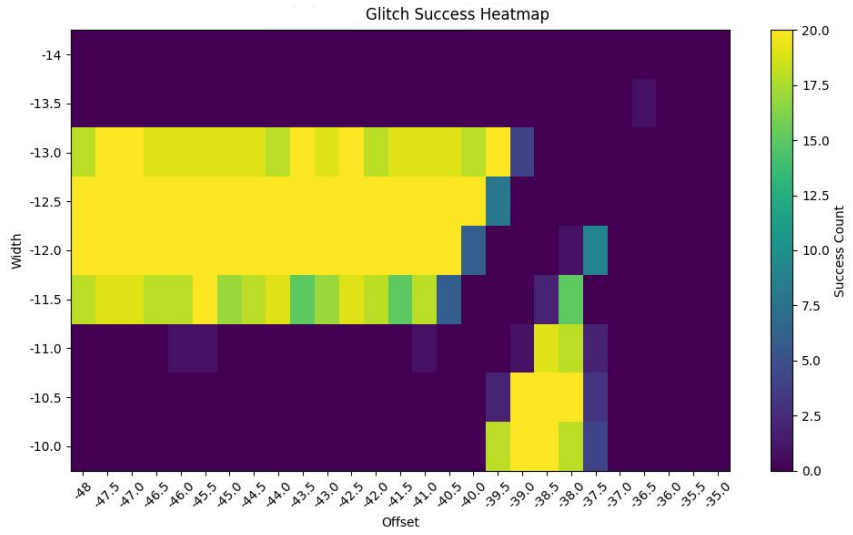

Figure 5: Heatmap of successful fault injections over the explored glitch parameter space. Each cell corresponds to a pair of glitch width and offset values both expressed as a percentage of the clock period. The color intensity indicates the number of successful faults (out of 20 attempts).

location, we identify stable fault conditions with a glitch offset of 45 ns and a source power of 65%, that consistently skip the call to `PQCLEAN_randombytes`. For characterization, the trigger was placed just before the `sig_perk_prg` as shown in Figure 1.

Under these conditions, the attack succeeds with 100% probability computed over 100 independent signature computations, each resulting in successful key recovery. The success rates (probability) can be found in Table 2. The difference in success rates between EMFI and clock glitching can be attributed to the spatial precision of EMFI, which enables more targeted disruption of control flow instructions.

5.3 Attack 2: Node Collision Attack

The goal of this attack is to induce collision of roots of two consecutive GGM trees.

5.3.1 Implementation Deviation and Its Impact on the Fault Surface

The PERK specification describes a clean mathematical scheme in which each GGM tree is generated only once during the commitment phase. The values computed in the commitment phase should not be recomputed. The NIST reference implementation in version 2.0.0 [ABB+25] abides by this assumption, at the cost of higher memory overhead. In the NIST reference implementation version 2.0.0, the prover commits to all party views after expanding the GGM tree from the master seed, and these values are stored in the memory and remain available through the different steps of the protocol. This force this attack, under those assumptions, to require two faults in the same signature. On the other hand, `pqm4` implementation diverge from this ideal model for memory-efficiency reasons, which makes them susceptible to fault attacks.

In the `pqm4` implementation, the GMM tree is not stored after the commitment phase. Instead, the implementation reconstructs the leaf seeds on demand during the second response phase once the verifier has chosen α. This design choice has a direct consequence

```
1 movs        challenges
2 add.w       mseed ,sp
3 add.w       signature ,sp
4 bl          sig_perk_prg
```

Listing 3: Assembly code of the `sig_perk_prg` function to be skipped

on the feasibility of the fault injection. Because the tree state does not persist after glitching the commitment phase only h_1 will be affected as the commitment is effectively overwritten when the prover later recomputes from scratch. In contrast, the second response phase becomes vulnerable, each call to the `sig_perk_prg` function regenerates a fresh root seed, and a single instruction skip at this point can cause two consecutive rounds to reuse the same $\theta^{(e)}$ value. As a result, the attack fault surface on the `pqm4` is shifted from the commitment phase to the second response stage.

5.3.2 Fault Injection on the second response generation

For this attack, the objective is to cause a fault in the second response of the signature creation. In a standard MPCitH scheme, the prover commits to several views in the first round. In the second round, after receiving the verifier's challenge α, the prover reveals all shares π_i and v_i except the one with the index α.

While the theoretical scheme implies that shares are stored after the initial commitment, the `pqm4` version optimizes for efficiency by re-deriving these shares from the master seed during the second response phase. As a result, by targeting the PRG expansion, an attacker can exploit the deterministic nature of the seed derivation. Since shares are reconstructed on-the-fly, a fault injected during this process directly corrupts the revealed information.

The attack specifically targets the `sig_perk_prg` function which expands the master key into θ values. As shown in Listing 3, the assembly implementation provides a clear target for instruction skip, by skipping the branch-link (bl) instruction to `sig_perk_prg`, the memory region remains unchanged with the previous θ value. The glitch can be performed for any (θ_i, θ_{i+1}) which means that the fault triggers can be positioned around any of the 16 values except for $\theta_{i=0}$.

To evaluate the impact of such an attack, the reference `pqm4` implementation was modified to include a synchronization mechanism. Listing 4 illustrates a simplified version of the `sign_perk_gen_gen_second_response` function, where a conditional trigger is inserted. This modification allows for the precise isolation of any e, in this case $e = 1$, during which the GPIO signal `trigger_high` is raised to synchronize the fault injection equipment. This setup bypasses the complexity of finding the exact external offset while confirming that skipping the PRG call at this specific stage results in the propagation of the faulted signature to the verifier.

5.3.3 Fault Injection using Clock Glitching

In order to identify the precise windows for this glitch, we had to perform a full cartography of the device that executes the target instruction. The cartography was conducted in two stages: first, targeting the isolated `sig_perk_prg` function directly, then scaling to the full `crypto_sign` call for an end-to-end validation.

Isolated function cartography: In the first stage, the glitch parameters we swept across the width offset and external offset. Two stable regions emerged consistently,

```
void sig_perk_gen_second_response(){
  for (uint8_t e = 0; e < PARAM_TAU; ++e) {
    vect1_t tmp;  vect1_t v_i;  vect1_t s_0;  vect1_t s_i;
    perk_theta_seeds_tree_t theta_tree;
    const uint16_t alpha = challenges[e].alpha;
    sig_perk_vect1_mult_scalar_vect(s_0, challenges[e].kappa[0], x[0]);
    for (int j = 1; j < PARAM_T; j++) {
      sig_perk_vect1_mult_scalar_vect(tmp, challenges[e].kappa[j], x[j]);
      sig_perk_vect1_add(s_0, s_0, tmp);
    }
    if (e==1) {
      trigger_high();
      sig_perk_prg(&prg_state, theta_tree[0], sizeof(theta_t));
      trigger_low();
    }else{
        sig_perk_prg(&prg_state, theta_tree[0], sizeof(theta_t));
    ...
}
```

Listing 4: Simplified implementation of the `pqm4` `second_response` function. The `if` instruction is added in order to target the second instruction, it could be replaced with the usage of a ext-offset.

centered around both ext offset of 4 and 16. Within these regions, the attack achieved a 100% success rate, which reliably causes the branch `sig_perk_prg` to be skipped and the previous θ value to be reused. Figure 6 illustrates the parameter space, with a distinction between successful skips where θ_{i-1} is reused rather than a glitch that produces an incorrect θ value rather than a crash or a normal execution.

Results for optimization level `-Os`. When doing the end-end characterization, the same parameter windows obtained during the full cartography were applied to the full `crypto_sign` function, the success rate dropped significantly, to approximately 10%. This degradation could be attributed to compiler inlining the `sig_perk_gen_second_response`, causing the targeted instruction to no longer align reliably with the instruction. The results is that the majority of glitch attempts in both ext offset regions produces mostly crashes or yielding a corrupted θ value with only 10% of the glitches causing a faulty signature that reveals the secret permutation.

Results for optimization level `-O3`. When repeating the end-to-end characterization with level `-O3`, very similar parameters to the `-Os` were applied to the full `crypto_sign` function. In this setting, we observe improved and more stable behavior within a very similar region centered around an offset of -3.5 and a width of 2. Within this zone, the attack success rate increases substantially, with the end-to-end optimization yielding 75% recovered signatures.

5.3.4 Fault Injection using EMFI

We repeated the second-response attack using EMFI to evaluate whether those perturbations could corrupt the call to the function during the generation of the revealed shares. As with the previous EMFI experiment, the same spatial cartography was used and the most responsive injection point was again located at $(x, y, z) = (192\,121.61, 244\,313.86, 0)$.

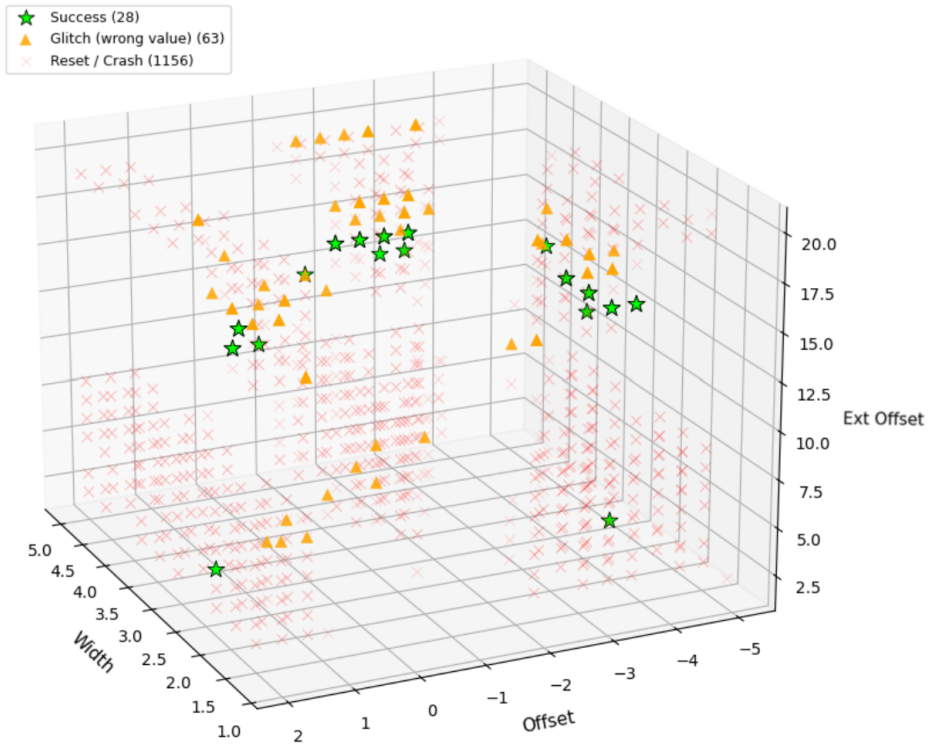

Figure 6: A 3D visualization illustrating the glitch parameter space across width, offset (both expressed as a percentage of the clock period), and external offset (measured in clock cycles). It highlights the transition zone between successful injections with green stars, glitched output using yellow triangles and system resets with red crosses

Using a source of power of 75% and an offset of $64ns$. We conducted 100 independent injections targeting the call associated with the second challenge, and here are the results on the 100 signatures obtained:

Table 4: Outcome of 100 EMFI injections during second-response generation.

Outcome	**Count**
Successful key recovery	85
Target non-responsive	13
Challenge collision ($\alpha_{e_1} = \alpha_{e_2}$)	2

6 Countermeasures

The two attacks presented in this paper exploit distinct vulnerabilities in the `pqm4` implementation of the GGM tree construction in PERK. We discuss countermeasures for each attack separately.

6.1 Countermeasures Against Attack 1

The first attack succeeds because the `randombytes` function can be skipped via a single fault inducing an instruction-skip, causing the master seed to remain in its zero-initialized

state without any detection. Several countermeasures can prevent this behavior.

Output Validation after `randombytes`. With the single instruction fault in mind, the simplest and most targeted fix is to verify that the output buffer is non-zero after the call to `randombytes` before its use to initialize the seed `mseed` or salt. Since a legitimate `mseed` is zero with a negligible probability, an explicit check could detect the fault and can be mitigated with another call to the function. The updated implementation would include the statement `randombytes(rand_bytes, sizeof(rand_bytes));` after line 6 in Listing 1. This prevents deterministic seed reuse under single instruction skip faults, as the attacker must now bypass both the generation and validation logic.

Duplicate sources of Randomness. The attack works because the inputs absorbed by the sponge function are dependent on `salt` and `mseed`, which are sourced from the same `rand_bytes`. Thus, an intuitive fix to this could be to initialize `mseed` and `salt` individually through different instantiations, requiring the attacker to successfully skip two function calls instead of targeting the one point/function call. The updated implementation would include the two instructions `randombytes(mseed, SEED_BYTES);` `randombytes(signature→salt, SALT_BYTES);` in place of line 6-8 in Listing 1.

6.2 Countermeasures Against the Node Collision Attack

The second attack exploits a fault surface introduced by a memory optimization specific to the `pqm4` implementation. The optimization involves clearing the GGM tree values from the memory after the commitment phase and regenerating the trees on-the-fly during the second response phase. Since the trees are generated sequentially, a single skip of the `sig_perk_prg` call will cause two consecutive rounds to share the same root seed $\theta^{(e)}$.

Resetting the θ buffer between rounds. A natural candidate countermeasure against the node collision attack is to reset the buffer holding $\theta^{(e)}$ using the `randombytes` call after each `sig_perk_prg` call, so that a single instruction skip would leave the buffer with a random value rather than retaining the previous round's value. While this fixes the single-fault injection attack, the countermeasure is insufficient for an adversary who can sucessfully glitch both `sig_perk_prg` and `randombytes` calls.

Post-generation Collision Detection. After deriving all τ root seeds from the master seed, the implementation verifies that no two consecutive values are identical (i.e., $\theta^{(e)} \neq \theta^{(e+1)}$, $e \in [\tau - 1]$) before revealing all shares to the verifier. Given that the number of seeds is constrained in the parameters-set for all PERK variants from $18 \leq \tau \leq 60$, this check can be performed in linear time, resulting in a complexity of $O(\tau)$.

7 Conclusion

In this work, we show that implementation-level optimizations in MPC-in-the-Head signature schemes can fundamentally undermine their security under fault injection. We presented two practical fault injection attacks on the PERK signature scheme and demonstrated them end-to-end on the `pqm4` implementation using both clock glitching and electromagnetic fault injection on ARM Cortex-M4 platforms. Our experiments show that a single fault is sufficient to achieve full key recovery, with success rates of 100% and 85% for the two attacks. Our results highlight a critical gap between the theoretical security guarantees of MPC-in-the-Head constructions and their practical implementations. In particular, memory-efficient optimizations such as GGM tree recomputation alter the

execution flow and introduce exploitable fault surfaces. We proposed practical countermeasures and evaluated their overhead. Our findings emphasize the need to incorporate physical attack resilience into the design and implementation of future PQC schemes, especially in constrained embedded environments.

References

[AAB+23] G. Adj, N. Aragon, S. Barbero, M. Bardet, E. Bellini, L. Bidoux, J. J. Chi-Domínguez, V. Dyseryn, A. Esser, T. Feneuil, P. Gaborit, R. Neveu, M. Rivain, L. Rivera-Zamarripa, C. Sanna, J. P. Tillich, J. Verbel, and F. Zweydinger. Mirath signature scheme. https://pqc-mirath.org, 2023.

[ABB+23] N. Aragon, M. Bardet, L. Bidoux, J. J. Chi-Domínguez, V. Dyseryn, T. Feneuil, P. Gaborit, A. Joux, R. Neveu, M. Rivain, J. P. Tillich, and A. Vinçotte. RYDE signature scheme. https://www.pqc-ryde.org/, 2023.

[ABB+25] N. Aaraj, S. Bettaieb, L. Bidoux, A. Budroni, V. Dyseryn, A. Esser, T. Feneuil, P. Gaborit, M. Kulkarni, V. Mateu, M. Palumbi, L. Perin, M. Rivain, J.-P. Tillich, and K. Xagawa. PERK: Post-quantum Emplementation of Relational Kernel, 2025. Version 2.0.

[AMBB+23] C. Aguilar Melchor, S. Bettaieb, L. Bidoux, T. Feneuil, P. Gaborit, N. Gama, S. Gueron, J. Howe, A. Hülsing, D. Joseph, A. Joux, M. Kulkarni, E. Persichetti, T. H. Randrianarisoa, M. Rivain, and D. Yue. The syndrome decoding in the head (SD-in-the-head) signature scheme. https://sdith.org, 2023.

[BBB+23] C. Baum, W. Beullens, L. Braun, C. D. de Saint Guilhem, M. Klooß, C. Majenz, S. Mukherjee, E. Orsini, S. Ramacher, C. Rechberger, L. Roy, and P. Scholl. FAEST v2: Algorithm specifications. https://faest.info/faest-spec-v2.0.pdf, 2023.

[BBFR25] R. Benadjila, C. Bouillaguet, T. Feneuil, and M. Rivain. MQOM: MQ on my mind. https://mqom.org, February 2025. Version 2.0.

[BBK25] Harrison Banda, Jan Brinkmann, and Juliane Krämer. Fault attacks on MPCitH signature schemes. Cryptology ePrint Archive, Report 2025/1745, 2025.

[BDPA11] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The keccak reference, 1 2011. http://keccak.noekeon.org/.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[JD25] Sönke Jendral and Elena Dubrova. Side-channel and fault injection attacks on voleith signature schemes: A case study of masked faest. *Cryptology ePrint Archive*, 2025.

[JD26] Sönke Jendral and Elena Dubrova. Side-channel attacks on VOLEitH signature schemes: Breaking masked FAEST. *CiC*, 2(4):24, 2026.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[MAKK24] Puja Mondal, Supriya Adhikary, Suparna Kundu, and Angshuman Karmakar. Zkfault: Fault attack analysis on zero-knowledge based post-quantum digital signature schemes. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 132–167. Springer, 2024.

[oSN24a] National Institute of Standards and Technology (NIST). Module-lattice-based key-encapsulation mechanism standard. https://doi.org/10.6028/NIST.FIPS.203, 2024-08-13 04:08:00 2024.

[oSN24b] National Institute of Standards and Technology (NIST). National Institute of Standards and Technology: NIST announces additional digital signature candidates for the PQC standardization process. https://csrc.nist.rip/news/2023/additional-pqc-digital-signature-candidates, 2024.

[oSND+24] National Institute of Standards, Technology (NIST), Thinh Dang, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, and Angela Robinson. Module-Lattice-Based Digital Signature Standard. https://doi.org/10.6028/NIST.FIPS.204, 2024-08-13 04:08:00 2024.

[Sha89] Adi Shamir. An efficient identification scheme based on permuted kernels. In *Conference on the Theory and Application of Cryptology*, pages 606–609. Springer, 1989.

[SJDN26] Mona Sobhani, Sönke Jendral, Elena Dubrova, and Mats Näslund. Breaking the one-way property of a SHA-3 implementation via fault injection: Key recovery attacks on post-quantum digital signatures. Cryptology ePrint Archive, Paper 2026/619, 2026.