Key-Recovery Fault Injection Attack on the Classic McEliece KEM

Sabine Pircher[1,2], Johannes Geier[3], Julian Danner[4], Daniel Mueller-Gritschneder[3], Antonia Wachter-Zeh[1]

[1]Technical University of Munich, Institute for Coding and Cryptography
{sabine.pircher, antonia.wachter-zeh}@tum.de
[2]HENSOLDT Cyber GmbH, Research & Development, Taufkirchen
[3]Technical University of Munich, Chair of Electronic Design Automation
{johannes.geier, daniel.mueller}@tum.de
[4]University of Passau, Chair of Symbolic Computation
julian.danner@uni-passau.de

Abstract. We present a key-recovery fault injection attack on the Classic McEliece Key Encapsulation Mechanism (KEM). The fault injections target the error-locator polynomial of the Goppa code and the validity checks in the decryption algorithm, making a chosen ciphertext attack possible. Faulty decryption outputs are used to generate a system of polynomial equations in the secret support elements of the Goppa code. After solving the equations, we can determine a suitable Goppa polynomial and form an alternative secret key. To demonstrate the feasibility of the attack on hardware, we simulate the fault injections on virtual prototypes of two RISC-V cores at register-transfer level.

Keywords: Post-Quantum Cryptography · Key Recovery · Fault Attack · Laser Fault Injections · Classic McEliece · Key Encapsulation Mechanism

1 Introduction

Post-Quantum Cryptography (PQC) is an important research topic due to the imminent development of large-scale quantum computers. If capable quantum computers become available, cryptographic systems based on the integer factorization problem and the discrete logarithm problem over finite fields and elliptic curves can be attacked in polynomial time due to the work of Shor [23] in 1997. Therefore, the currently employed public-key cryptographic schemes like RSA and ECC are no longer secure. In 2017, the U.S. National Institute of Standards and Technology (NIST) initiated a competition for post-quantum cryptography to replace their current FIPS 186 and SP 800-56A/B recommendations [16]. PQC includes algorithms that run on classical computers but are resistant against attacks from quantum computers. The hardness of PQC is based on computationally hard problems that are expected to be resistant against attacks performed by quantum computers for the next tens or hundreds of years.

Currently proposed algorithms rely on, e.g., lattice problems, the syndrome decoding problem of error-correcting codes, the solving of multivariate equations and isogenies between elliptic curves.

The McEliece cryptosystem is a code-based public-key encryption (PKE) scheme that relies on the syndrome decoding problem (SDP) for decoding error-correcting codes [14]. The McEliece cryptosystem was introduced in 1978 [14] and its dual version, the Niederreiter cryptosystem, in 1986 [18]. In general, every PKE can be transformed into a key encapsulation mechanism (KEM) that encapsulates and decapsulates a symmetric secret session key for a key exchange procedure and is IND-CCA2 secure. The Classic McEliece KEM [3] is a code-based cryptosystem among the finalists of the PQC competition based on the Niederreiter PKE. It needs a large public key size compared to lattice-based KEMs, but is free from decryption failures and has a long history of research. With the efforts towards standardization, the security of implementations is an important issue and fault attacks are an interesting field of research. Fault injections are physical attacks on hardware which lead to computation errors that are exploited to extract secret information from the device. To produce such errors one may use highly focused laser beams that achieve good spatial and temporal precision in order to set and reset single and adjacent bits on a chip [22].

Cayrel et al. [4] present a message-recovery fault attack on Classic McEliece by attacking the syndrome computation that changes the syndrome from $\mathbb{F}_2$ to the integers $\mathbb{N}$. The resulting syndrome decoding problem in $\mathbb{N}$ can be easily solved by integer linear programming. In [5] they present a similar message-recovery attack using only side-channel information on power consumption of the chip. This attack also gathers information on the syndrome in $\mathbb{N}$ but is more tolerant to noise. Very recently, Guo et al. [10] published a key-recovery side-channel attack on Classic McEliece KEM. They use chosen ciphertexts and exploit a side-channel leakage in the additive Fast Fourier Transform (FFT) that evaluates the ELP during decoding. Xagawa et al. [27] demonstrate a single-fault injection attack that works for all NIST PQC Round 3 KEM candidates except Classic McEliece. The single-fault injection attacks presented in [27] are executed by skipping instructions on a chip using glitching of the power supply. The skipping circumvents the IND-CCA2 security of the KEM and enables chosen-chiphertext attacks on the vulnerable PKE. In the course of this work, we found useful relations for completing partially known support sets of a Goppa code, as independently reported by [11].

In this paper, we show that we can obtain an alternative secret key of Classic McEliece by adapting and combining the skipping attacks of Xagawa et al. [27] with the fault injection attack on the PKE by Danner and Kreuzer [6]. We additionally investigated our fault attack on two Open Source RISC-V processors.

The structure of the paper is as follows: Section 2 reviews the Classic McEliece KEM. In Section 3 we define our hardware fault model and give a mathematical description of the key-recovery attack. In Section 4 we present our implementation details and simulation results that validate our attack, together with a

feasibility study for two RISC-V processors using RTL simulations. We conclude in Section 5. All algorithms can be found in the Appendix.
We use the following notation: The finite field of size q is denoted by $\mathbb{F}_q$. Row vectors are denoted by bold lower-case letters (e.g., $\mathbf{e}$) and column vectors by $\mathbf{e}^\top$. Denote $\text{supp}(\mathbf{e}) := \{i \in \{1, \dots, n\} \mid e_i \neq 0\}$ for $\mathbf{e} \in \mathbb{F}_q^n$, and the Hamming weight of $\mathbf{e}$ by $\text{wt}(\mathbf{e})$. We denote matrices by bold capital letters (e.g., $\mathbf{H}$). We consider the Hamming metric for weight and distance. Let $\mathbb{F}_q[x]$ denote the univariate polynomial ring in x with coefficients in $\mathbb{F}_q$. For a polynomial $f \in \mathbb{F}_q[x]$ we denote its degree by $\deg(f)$, and the ideal generated by f with $\langle f \rangle$.

2 Classic McEliece KEM

The Classic McEliece KEM specified in [3] is designed as a quantum-resistant public-key encapsulation mechanism based on the Niederreiter cryptosystem [18]. The security relies on the syndrome-decoding problem (SDP) which is NP-complete for random linear codes [2]. The core idea of the Niederreiter cryptosystem in the KEM is to choose a binary irreducible Goppa code which allows efficient correction of errors when the algebraic structure is known, while a systematic parity check matrix of the code appears to be random. The algebraic structure is part of the secret key and the legitimate user thereby has access to an efficient decoder. For everyone else, the linear code appears to be a random code. It is believed that not only traditional computers, but also quantum computers require an exponential number of operations to correct errors without knowledge of the underlying algebraic structure.
Classic McEliece is extremely efficient in encoding and decoding at the cost of a large public key. For the CAT-5 proposed parameters the public key size is about 1 MB. This large public key makes its generation and storage more expensive compared to other PQC cryptosystems. We work with the Classic McEliece KEM implementation [3] submitted to NIST Round 3 and explain its key functionalities in the remainder of this section.[1] The current proposed parameter sets for Classic McEliece are listed in Table 4 in the appendix.
Classic McEliece consists of three main functions: key generation, encapsulation and decapsulation. They use the public parameters $n, m, t \in \mathbb{N}$ with $n \leq 2^m$, and a monic irreducible polynomial $f(z) \in \mathbb{F}_2[z]$ of degree m. The latter is used to fix a representation of elements in the field $\mathbb{F}_{2^m} \cong \mathbb{F}_2[z]/\langle f(z) \rangle$ as bit-tuples, i.e., elements in $\mathbb{F}_2^m$. The identification is given by the bijective map $\varphi \colon \mathbb{F}_2^m \to \mathbb{F}_2[z]/\langle f(z) \rangle \cong \mathbb{F}_{2^m}$ where $(c_0, \dots, c_{m-1}) \mapsto c_0 + c_1 z + \dots + c_{m-1} z^{m-1}$.
Classic McEliece uses the SHA-3 Keccak SHAKE-256 hash function, defined in [17]. We denote it by $\mathcal{H}$, and its output is always 256 bits long, independent of its input length. In particular, we write $\mathcal{H}(2, \mathbf{v})$ and $\mathcal{H}(i, \mathbf{v}, C)$ for the hash of the concatenation of an initial byte valued $i \in \{0, 1\}$ or 2, vector $\mathbf{v} \in \mathbb{F}_2^n$ and ciphertext C, see also [3, Sec. 2.5.2].

[1] In this paper we do not consider the accelerated variant of the key generation in Classic McEliece that also accepts a semi-systematic form of the parity-check matrix. We expect the attack to work also on this variant after minor modification.

2.1 Key generation

In this paper we understand the secret key[2] as a tuple $(\mathbf{s}, \gamma)$ where $\mathbf{s}$ is a bit-vector in $\mathbb{F}_2^n$ and $\gamma = (\, g, \alpha\,) \in \mathbb{F}_{2^m}[x] \times \mathbb{F}_{2^m}^n$ is a generator tuple of the **binary irreducible Goppa code**

$$\Gamma(\alpha, g) = \left\{ (c_1, \ldots, c_n) \in \mathbb{F}_2^n \mid \sum_{i \in \operatorname{supp}(c)} (x - \alpha_i)^{-1} = 0 \text{ in } \mathbb{F}_{2^m}[x]/\langle g \rangle \right\} \subseteq \mathbb{F}_2^n,$$

with $\deg(g) = t$ and $\alpha = (\alpha_1, \ldots, \alpha_n)$.
Then g is a monic irreducible polynomial and called the **Goppa polynomial** of the code, and $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{F}_{2^m}^n$ satisfies $\alpha_i \neq \alpha_j$ for $i \neq j$ and $g(\alpha_i) \neq 0$ and is called the **support** of the code. Key generation ensures that the linear code $\Gamma(\alpha, g)$ has dimension $k = n - mt$, length n and allows efficient correction of up to t errors. Moreover, one can compute a parity-check matrix in systematic form $\mathbf{H}_{sys} = (\mathbf{I}_{n-k} | \mathbf{T})$, where $\mathbf{I}_{n-k}$ is the identity matrix of size $n - k$. The public key is then given by the matrix $\mathbf{T} \in \mathbb{F}_2^{(n-k)\times k}$. Note that in particular the code $\Gamma(\alpha, g)$ itself is public knowledge since $\mathbf{T}$ is public. However, the algebraic structure, i.e., the Goppa polynomial and the support, are part of the secret key. Algorithm 1 summarizes the construction of the secret and public keys in Classic McEliece.

2.2 Encapsulation

The encapsulation (Algorithm 3) takes a random plaintext and uses the public key to generate a ciphertext from which only the holder of the secret key can extract the random plaintext again. This can be used to establish a common secret session key. In particular, the encapsulation party chooses a vector $\mathbf{e} \in \mathbb{F}_2^n$ of Hamming weight t at random. The ciphertext $C = (\mathbf{c_0}, C_1)$ is generated by encoding the vector $\mathbf{e}$ using the public key such that $\mathbf{c_0} = \mathbf{e}\mathbf{H_{sys}}^\top$ and by calculating the hash $C_1 = \mathcal{H}(2, \mathbf{e})$. The secret session key K is then the hash of $\mathcal{H}(1, \mathbf{e}, C)$. Details can be found in Algorithm 2 and Algorithm 3.

2.3 Decapsulation

The holder of the secret key can compute the same session key using Algorithm 5. This is done by splitting the received ciphertext into the two parts $\mathbf{c_0} \in \mathbb{F}_2^{n-k}$ and the hash C_1, decoding $\mathbf{c_0}$ to a vector $\mathbf{e}' \in \mathbb{F}_2^n$ of weight t using knowledge of the generator tuple (α, g) of the Goppa code. Then the result is checked by calculating $C_1' = \mathcal{H}(2, \mathbf{e}')$ and ensuring that C_1' and C_1 are equal. (If no errors occurred during transmission and the computation is not faulted, this is the case.) Then the output is given by the (reconstructed) session key $K' = \mathcal{H}(1, \mathbf{e}', C)$. In this way both parties conclude with the same session key.

[2] In the actual implementation the secret key does not contain the support α explicitly, but instead the seed of the random function that is used to generate it.

In case the input C_1 or C_2 is no valid ciphertext, the decoding step will fail, or the check of $C_1' = C_1$. In this case a predefined output $K' = \mathcal{H}(0, \mathbf{s}, C)$ is returned, where $\mathbf{s} \in \mathbb{F}_2^n$ is part of the secret key (see line 7 in Algorithm 1).
Our fault injections target the decapsulation algorithm in order to gain polynomial equations in the support α of the Goppa code. To decode a syndrome $\mathbf{c_0} = \mathbf{e}\mathbf{H_{sys}}^\top \in \mathbb{F}_2^{n-k}$ Algorithm 4 first forms the word $\mathbf{v} \in \mathbb{F}_2^n$ by appending zeros to $\mathbf{c_0}$. By construction the syndrome of $\mathbf{v}$ w.r.t. $\mathbf{H_{sys}}$ is exactly $\mathbf{c_0}$, i.e., $\mathbf{v}$ and $\mathbf{e}$ are in the same coset. This means that there is a codeword $\mathbf{c} \in \Gamma(\alpha, g)$ such that $\mathbf{v} = \mathbf{c} + \mathbf{e}$. This word is computed in Line 2. Different algorithms have been suggested in literature for this decoding step: the Sugiyama Algorithm [25], the Berlekamp-Massey Algorithm [1,13], and the Patterson Algorithm [21]. All of them explicitly compute the **error-locator polynomial** (ELP) of $\mathbf{e} \in \mathbb{F}_2^n$ defined as

$$\sigma_e(x) = \prod_{i \in \mathrm{supp}(\mathbf{e})} (x - \alpha_i) \in \mathbb{F}_{2^m}[x].$$

The error $\mathbf{e}$ can then be reconstructed directly from the zeros of $\sigma_e(x)$, since we have for all $i \in \{1, \ldots, n\}$: $\mathbf{e}_i = 1$ if and only if $\sigma_e(\alpha_i) = 0$. For more details on Goppa codes we refer the reader to [12, Ch. 12]. Figure 1a depicts the corresponding steps that are executed in the Classic McEliece implementations, which use the Berlekamp-Massey algorithm to find the ELP.

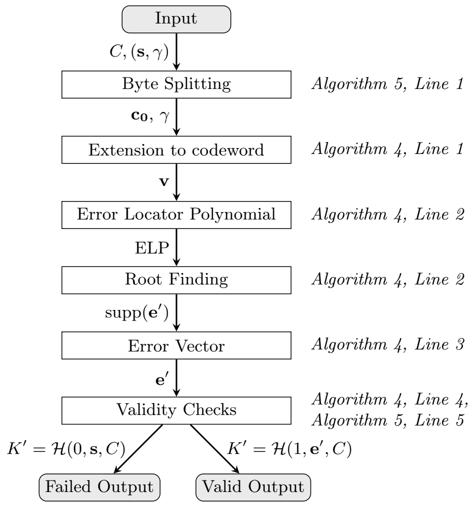
2.4 Implementation

The implementations submitted to NIST [3] contain a reference implementation, as well as several hardware accelerated implementations for x86/AMD64 processors. For our software simulation of the attack, we adapt the hardware accelerated implementation that makes use of vector arithmetics on the processor for faster runtime. To simulate the faut injections on RISC-V cores, we use the reference implementation.
The ELP $\sigma_e(x) \in \mathbb{F}_{2^m}[x]$ is represented differently in the reference and hardware accelerated code. The following remark summarizes how the implementations handle invalid inputs, i.e., syndromes corresponding to errors of smaller weight.

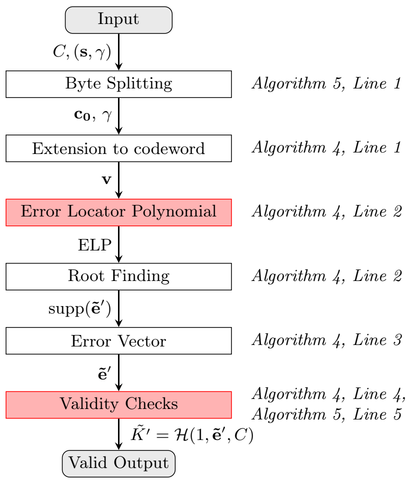
Remark 1 (ELP Implementation Details).

(a) For any valid syndrome $\mathbf{c_0} = \mathbf{e}\mathbf{H_{sys}}^\top$ with $\mathrm{wt}(\mathbf{e}) \leq t$, the coefficients of the corresponding ELP $\sigma_e(x)$ are stored in such a way that it is read as the polynomial $\sigma_e(x) \cdot x^{t - \deg(\sigma_e)}$ of degree t.
(b) This does not affect error correction as long as no α_i is zero, or $\mathrm{wt}(\mathbf{e}) = \deg(\sigma_e(x)) = t$. But, if there is $i \in \{1, \ldots, n\}$ with $\alpha_i = 0$ and $\mathrm{wt}(\mathbf{e}) < t$, then the output $\mathbf{e}' \in \mathbb{F}_2^n$ of line 2 in Algorithm 4 is indeed changed and we get $\mathrm{supp}(\mathbf{e}') = \mathrm{supp}(\mathbf{e}) \cup \{i\}$.
(c) In particular, $\mathbf{e} \neq \mathbf{e}'$ only if there is an $i \in \{1, \ldots, n\}$ with $\alpha_i = 0$ and $\mathbf{e}_i = 0$, and we have $\mathrm{wt}(\mathbf{e}') \leq \mathrm{wt}(\mathbf{e}) + 1$.

Later, this allows us to quickly find the index $i \in \{1, \ldots, n\}$ with $\alpha_i = 0$, if there is such (see Remark 4).



(a) Normal operation



(b) Faulty operation

Fig. 1: Flowchart showing the decapsulation and decoding steps indicating the differences between normal and faulty operation. Fault injections target the steps marked in red.

3 Key-Recovery Attack

This section describes our key-recovery attack. It targets the decapsulation function and can find an alternative secret key. This key can be used in place of the original secret key, i.e., it can be used to find session keys generated with the corresponding public key. Three steps are necessary: First, we inject a fault in the decoding procedure on the ELP coefficients so that it leaks information about the secret key, adapting the work of [6]. Second, we inject a fault to bypass the validity check (VCB) ensuring the faulty decoding result is not rejected. This is done similar to [27]. Third, we demonstrate that under given circumstances the information about the secret key contained in the hashed output can be retrieved. The injections in the decapsulation algorithm necessary for the attack are illustrated in Figure 1b. We describe two kinds of faults in the ELP coefficients: ELP coefficient bit corruption (ELPB) and ELP coefficient zeroing (ELPZ). Both lead to successful key recovery, with ELPZ being more efficient. ELPZ further has synergy with the validity check bypass (VCB), as both can be achieved by fault injections at the same position on the chip (see Section 4.3).

3.1 Fault Model

We consider the setting that the secret key is stored in a Trusted Execution Environment (TEE) so that its memory location is well protected. Only the TEE itself has access to the secret key, i.e., the key cannot be physically accessed or retrieved by any other means.

Assumption 1 *The attacker has access to the input and output of the decapsulation function (Algorithm 5). We can freely choose the input of the decapsulation function (chosen ciphertext attack).*

Assumption 2 *We can inject faults on the physical device during decapsulation changing the transistor states at specific positions and times. To be precise, we assume that single and adjacent bits can be set or reset.*

Such faults are achievable e.g. by a laser fault injection [22].
On the computational level, the following is achieved by fault injections: The validity check in line 4 of Algorithm 4 and line 5 of Algorithm 5 is bypassed (VCB), and the ELP is corrupted either by setting or resetting one or more adjacent bits in a single coefficient (ELPB) or by setting a coefficient to zero (ELPZ).
To simplify the theoretic analysis of fault injections into the ELP, consider the following remark.

Remark 2. We model the faults on a coefficient $a \in \mathbb{F}_{2^m}$ of the ELP as an addition in the field $\mathbb{F}_{2^m}$, i.e., write the faulty coefficient $\tilde{a} \in \mathbb{F}_{2^m}$ as $\tilde{a} = a + \xi$ for some appropriately chosen $\xi \in \mathbb{F}_{2^m}$. Note that for our attack we do not need to know the fault value ξ.

3.2 Fault Attack on the Validity Checks (VCB)

The validity checks confirm whether the decoding function provides a valid output and compares the corresponding hashes. If not, a predefined session key is returned ("Failed Output" in Figure 1a). After fault injection into the ELP, the faulty output $\tilde{\mathbf{e}}'$ in general does not pass the check $\tilde{\mathbf{e}}'\mathbf{H}_{sys}^\top = \mathbf{e}\mathbf{H}_{sys}^\top$ and $\mathrm{wt}(\tilde{\mathbf{e}}') = t$ in Algorithm 4 line 4 and the check $\mathcal{H}(2, \mathbf{e}) = \mathcal{H}(2, \tilde{\mathbf{e}}')$ in Algorithm 5 line 5. Therefore, we need an additional fault injection to bypass these checks such that we are able to retrieve the faulty session key $\tilde{K}' = \mathcal{H}(1, \tilde{\mathbf{e}}', C)$, which contains information about $\tilde{\mathbf{e}}'$, see Figure 1b.
A faulty session key $\tilde{K}' = \mathcal{H}(1, \tilde{\mathbf{e}}', C)$ is a hash of the input ciphertext $C = (\mathbf{c_0}, \mathcal{H}(2, \mathbf{e}))$ and the output $\tilde{\mathbf{e}}' \in \mathbb{F}_2^n$ of the decode algorithm. According to our fault model the attacker has full control over C. It is feasible to extract $\tilde{\mathbf{e}}'$ from $\tilde{K}'$ by exhaustive search if the weight of $\tilde{\mathbf{e}}'$ is small enough.

Remark 3 (De-hash Session Key).

(a) If C and hash $\tilde{K}' = \mathcal{H}(1, \tilde{\mathbf{e}}', C)$ are known for some $\tilde{\mathbf{e}}' \in \mathbb{F}_2^n$ with $\mathrm{wt}(\tilde{\mathbf{e}}') \leq 2$, then one can find $\tilde{\mathbf{e}}'$ with less than $\binom{n}{2} + \binom{n}{1} + \binom{n}{0}$ hash computations and comparisons via exhaustive search.
(b) The statement in (a) is also true if $\mathrm{wt}(\tilde{\mathbf{e}}') \leq 3$ and one index $i \in \mathrm{supp}(\tilde{\mathbf{e}}')$ is known.
(c) For the parameters (Table 4), we have $n \leq 2^{13}$, this means that less than $2^{25} + 2^{12} + 1$ hash computations and comparisons are required to find the output of the decoding algorithm $\tilde{\mathbf{e}}'$ from a faulty session key $\tilde{K}'$, given that $\mathrm{supp}(\tilde{\mathbf{e}}')$ contains at most two unknown indices.

3.3 Fault Attack on the ELP Coefficients

The goal is to inject faults into certain coefficients of the error-locator polynomial (ELP) during the decoding process of chosen words $\mathbf{e} \in \mathbb{F}_2^n$ of Hamming weight 2 such that the evaluation of that polynomial is erroneous. If we have access to the faulty output $\tilde{\mathbf{e}}' \in \mathbb{F}_2^n$ of the decoding step, we can obtain polynomial equations in the secret support α. A set of such equations eventually leads to an alternative support $\tilde{\alpha} \in \mathbb{F}_{2^m}^n$ for which there is an irreducible polynomial $\tilde{g} \in \mathbb{F}_{2^m}[x]$ of degree t with $\Gamma(\alpha, g) = \Gamma(\tilde{\alpha}, \tilde{g})$. This allows efficient correction of up to t errors in the code $\Gamma(\alpha, g)$. Hence, for every $\mathbf{s} \in \mathbb{F}_2^n$ the tuple $(\mathbf{s}, (\tilde{g}, \tilde{\alpha}))$ can be used as an alternative secret key with Algorithm 5.
The fault injections on the ELP are mainly based on the ideas of the fault attack presented in [6], but a handful of adjustments had to be made to accommodate the different fault model and the peculiarities of the implementation. Also the solving process was refined to decrease the number of required fault injections.
Before we discuss how corrupting coefficients of the ELP can lead to polynomial equations in the unknown support $\alpha \in \mathbb{F}_{2^m}^n$ of the Goppa code, we show that one can easily check if zero is one of the support elements and if so, find its index.

Locating zero in the support In the previous section we have seen that we can choose the input of the decoding algorithm as well as read the output, if it is of small weight (Remark 3). So instead of syndromes corresponding to errors $\mathbf{e} \in \mathbb{F}_2$ of weight t, we may select errors of smaller weight, e.g. the all-zero vector. This allows us to decide whether zero is contained in the support, and if it is, find the index $i \in \{1, \dots, n\}$ for which $\alpha_i = 0$.

Remark 4 (Finding Zero). Let $\mathbf{e} = \mathbf{0} \in \mathbb{F}_2^n$, and consider $\mathbf{c_0} = \mathbf{e}\mathbf{H_{sys}}^\top$ as input for the decoding algorithm. If there is a $j \in \{1, \dots, n\}$ with $\alpha_j = 0$, then the decoding algorithm evaluates the polynomial x^t and outputs $\mathbf{e}' \in \mathbb{F}_2^n$ where $\mathrm{supp}(\mathbf{e}') = \{j\}$ by Remark 1; otherwise we have $\mathbf{e}' = \mathbf{e} = \mathbf{0}$. Since $\mathrm{wt}(\mathbf{e}') \leq 1$ we can quickly access $\mathbf{e}'$ from the hash output of the decapsulation function, see Remark 3, and from $\mathbf{e}'$ read off whether there is $j \in \{1, \dots, n\}$ with $\alpha_j = 0$ and in that case deduce this index j.

From now on, we assume that we know $j \in \{1, \dots, n\}$ with $\alpha_j = 0$, if there is such an index; as this information can be gathered with just a single run of the decapsulation algorithm where the validity checks are skipped.

Corrupting bits of coefficients (ELPB) For the fault injections on the ELP that eventually provide the polynomial equations in the support α, we choose syndromes corresponding to vectors $\mathbf{e} \in \mathbb{F}_2^n$ of weight 2 as input to the decoding algorithm. This way the ELP has the form $\sigma_e(x) = (x - \alpha_{i_1})(x - \alpha_{i_2}) \in \mathbb{F}_{2^m}[x]$ for chosen $i_1, i_2 \in \{1, \dots, n\}$ with $i_1 \neq i_2$. The idea is to set or reset single or adjacent bits in one of the two coefficients such that it is replaced by $\tilde{\sigma}_e(x) = \xi x^d + \sigma_e(x)$ for $d \in \{0, 1\}$ and some (unknown) $\xi \in \mathbb{F}_{2^m}$ (see Remark 2). Recall that the output of the decode algorithm, say $\tilde{\mathbf{e}}' \in \mathbb{F}_2^n$, is constructed not from the zeros of $\tilde{\sigma}_e(x)$ but from the zeros of $x^{t-2}\tilde{\sigma}_e(x)$, see Remark 1. The next remark summarizes the information about the zeros of $\tilde{\sigma}_e(x)$ that can be obtained from $\tilde{\mathbf{e}}'$.

Remark 5. Let $\mathbf{e} \in \mathbb{F}_2^n$ with $\mathrm{supp}(\mathbf{e}) = \{i_1, i_2\}$ and $i_1 \neq i_2$. Assume that a fault $\xi \in \mathbb{F}_{2^m}$ is injected into the d-th coefficient of $\sigma_e(x)$ with $d \in \{0,1\}$. Let $\tilde{\mathbf{e}}' \in \mathbb{F}_2^n$ be the output of the decoding algorithm when the polynomial for the root finding is given by $x^{t-2}\tilde{\sigma}_e(x)$ where $\tilde{\sigma}_e(x) = \xi x^d + \sigma_e(x)$.

(a) If $\alpha_j \neq 0$ for all $j \in \{1, \dots, n\}$, then we have $\mathrm{wt}(\tilde{\mathbf{e}}') \leq 2$, and

$$\mathrm{supp}(\tilde{\mathbf{e}}') = \{i \in \{1, \dots, n\} \mid \alpha_i \text{ is a zero of } \tilde{\sigma}_e(x)\}.$$

(b) If there is $j \in \{1, \dots, n\}$ with $\alpha_j = 0$, then $\mathrm{wt}(\tilde{\mathbf{e}}') \leq 3$ and $j \in \mathrm{supp}(\tilde{\mathbf{e}}')$ is known. Moreover, we have

$$\mathrm{supp}(\tilde{\mathbf{e}}') = \{i \in \{1, \dots, n\} \mid \alpha_i \text{ is a zero of } \tilde{\sigma}_e(x)\} \cup \{j\}.$$

By Remark 4 we can distinguish those two cases, and Remark 3 tells us how we can then gain access to $\tilde{\mathbf{e}}' \in \mathbb{F}_2^n$.

Definition 6. *Let $d \in \{0,1\}$, $\mathbf{e} \in \mathbb{F}_2^n$ with $\mathrm{supp}(\mathbf{e}) = \{i_1, i_2\} \subseteq \{1, \dots, n\}$, $i_1 \neq i_2$, and let $\tilde{\mathbf{e}}' \in \mathbb{F}_2^n$ be the output of Algorithm 4 where*

(1) a fault was injected such that the d-th coefficient of $\sigma_e(x)$ is corrupted by $\xi \in \mathbb{F}_{2^m}$, i.e., the ELP is replaced by $\tilde{\sigma}_e(x) = \xi x^d + \sigma_e(x)$,

(2) the output $\tilde{\mathbf{e}}'$ is constructed from the roots of $x^{t-2}\tilde{\sigma}_e(x)$ (see Remark 1), and

(3) a fault injection ensures that the validity checks in line 4 pass.

*Then we call the tuple $(\mathbf{e}, \tilde{\mathbf{e}}')$ a **fault injection**. If $d = 0$ it is also called a **constant injection**, and a **linear injection** for $d = 1$, respectively.*

Our fault model allows to generate arbitrary many such fault injections. Also note that we assume no control over the unknown fault ξ.
Not all fault injections lead to polynomial equations, only those where the faulty ELP has two zeros among the support α. In view of Remark 5, a sufficient condition is given by the following definition.

Definition 7. *A fault injection $(\mathbf{e}, \tilde{\mathbf{e}}')$ is called **successful**, if*

(1) for all $j \in \{1, \dots, n\}$ we have $\alpha_j \neq 0$ and $\mathrm{wt}(\tilde{\mathbf{e}}') = 2$, or

(2) there is $j \in \{1, \dots, n\}$ with $\alpha_j = 0$ and $\mathrm{wt}(\tilde{\mathbf{e}}') = 3$.

For every successful fault injection the set $\{i \in \{1, \dots, n\} \mid \alpha_i \text{ is a zero of } \tilde{\sigma}_e(x)\}$ can be deduced from $\tilde{\mathbf{e}}'$ with Remark 5 and contains exactly two elements.
Next we explain why the term *successful* is adequate.

Proposition 8. *Let $(\mathbf{e}, \tilde{\mathbf{e}}')$ be a successful fault injection with $\mathrm{supp}(\mathbf{e}) = \{i_1, i_2\}$ and $\{i \in \{1, \dots, n\} \mid \alpha_i \text{ is a zero of } \tilde{\sigma}_e(x)\} = \{j_1, j_2\}$.*

(a) If $(\mathbf{e}, \tilde{\mathbf{e}}')$ is a successful constant injection, then $\alpha_{i_1} + \alpha_{i_2} = \alpha_{j_1} + \alpha_{j_2}$, and α is a zero of the linear polynomial $x_{i_1} + x_{i_2} + x_{j_1} + x_{j_2} \in \mathbb{F}_{2^m}[x_1, \dots, x_n]$.

(b) If $(\mathbf{e}, \tilde{\mathbf{e}}')$ *is a successful linear injection, then* $\alpha_{i_1}\alpha_{i_2} = \alpha_{j_1}\alpha_{j_2}$*, and* α *is a zero of the quadratic polynomial* $x_{i_1}x_{i_2} + x_{j_1}x_{j_2} \in \mathbb{F}_{2^m}[x_1, \ldots, x_n]$*.*

Proof. Denote the ELP by $\sigma_e(x) = (x - \alpha_{i_1})(x - \alpha_{i_2})$ and the faulty ELP by $\tilde{\sigma}_e(x) = \xi x^d + \sigma_e(x)$ for $d \in \{0, 1\}$ and $\xi \in \mathbb{F}_{2^m}$. By Remark 5 and Definition 7, we get that α_{j_1} and α_{j_2} are the roots of $\tilde{\sigma}_e(x)$, i.e., $\tilde{\sigma}_e(x) = (x - \alpha_{j_1})(x - \alpha_{j_2})$. Both statements are now shown by comparing the coefficients in $\xi x^d + \sigma_e(x) = \tilde{\sigma}_e(x)$:

$$\xi x^d + (x - \alpha_{i_1})(x - \alpha_{i_2}) = (x - \alpha_{j_1})(x - \alpha_{j_2}).$$

For (a) we get $x^2 + (\alpha_{i_1} + \alpha_{i_2})x + (\alpha_{i_1}\alpha_{i_2} + \xi) = x^2 + (\alpha_{j_1} + \alpha_{j_2})x + \alpha_{j_1}\alpha_{j_2}$, as $d = 0$. Comparing the linear coefficients yields $\alpha_{i_1} + \alpha_{i_2} = \alpha_{j_1} + \alpha_{j_2}$. For (b) we get $x^2 + (\alpha_{i_1} + \alpha_{i_2} + \xi)x + \alpha_{i_1}\alpha_{i_2} = x^2 + (\alpha_{j_1} + \alpha_{j_2})x + \alpha_{j_1}\alpha_{j_2}$, as $d = 1$. So in particular $\alpha_{i_1}\alpha_{i_2} = \alpha_{j_1}\alpha_{j_2}$ follows from the constant coefficients. □

Remark 9. The probability to have a successful fault injection increases with the ratio $\frac{n}{2^m}$. This follows simply from the fact that the number of support elements increases with n and by that also the number of possible roots for the faulty ELP $\tilde{\sigma}_e(x)$ increases.

Zeroing Coefficients (ELPz). Instead of targeting the General Purpose Register (GPR) holding the ELP coefficients directly, one may also aim at the instructions operating on them. For example, by skipping the instruction storing the ELP coefficient to memory, the resulting coefficient will be equal to zero. This is the case because the algorithm sets the ELP vector to zero before calculating its coefficients. Such fault injections also fit well with Definition 6, where the fault value $\xi \in \mathbb{F}_{2^m}$ has the same value as the targeted coefficient of the ELP such that the coefficient cancels out. Using coefficient-zeroing fault injections can also provide polynomial equations as follows.

Proposition 10. *Let* $(\mathbf{e}, \tilde{\mathbf{e}}')$ *be a fault injection on the d-th coefficient of* $\sigma_e(x)$ *s.t. the d-coefficient of* $\tilde{\sigma}_e(x)$ *is zero. Write* $\operatorname{supp}(\mathbf{e}) = \{i_1, i_2\}$*, and let* $j \in \operatorname{supp}(\tilde{\mathbf{e}}')$ *with* $\alpha_j \neq 0$*.*

(a) If $(\mathbf{e}, \tilde{\mathbf{e}}')$ *is a constant fault injection, then we have* $\alpha_{i_1} + \alpha_{i_2} = \alpha_j$*, and* α *is a zero of the linear polynomial* $x_{i_1} + x_{i_2} + x_j \in \mathbb{F}_{2^m}[x_1, \ldots, x_n]$*.*

(b) If $(\mathbf{e}, \tilde{\mathbf{e}}')$ *is a linear fault injection, then we have* $\alpha_{i_1}\alpha_{i_2} = \alpha_j^2$*, and* α *is a zero of the quadratic polynomial* $x_{i_1}x_{i_2} + x_j^2 \in \mathbb{F}_{2^m}[x_1, \ldots, x_n]$*.*

Proof. Note that we have $\sigma_e(x) = (x - \alpha_{i_1})(x - \alpha_{i_2}) = x^2 + (\alpha_{i_1} + \alpha_{i_2})x + \alpha_{i_1}\alpha_{i_2}$. In the situation of (a) we have $\tilde{\sigma}_e(x) = x^2 + (\alpha_{i_1} + \alpha_{i_2})x$. By Remark 1 the implementation constructs $\tilde{\mathbf{e}}'$ from the zeros of $x^{t-2}\tilde{\sigma}_e(x) = x^{t-1}(x + \alpha_{i_1} + \alpha_{i_2})$ which has only one non-zero root, $\alpha_{i_1} + \alpha_{i_2}$. Now $j \in \operatorname{supp}(\tilde{\mathbf{e}}')$ and $\alpha_j \neq 0$ imply that α_j is exactly this non-zero root. Thus we get $\alpha_j = \alpha_{i_1} + \alpha_{i_2}$. For (b) we have $\tilde{\sigma}_e(x) = x^2 + \alpha_{i_1}\alpha_{i_2}$. Now we know that α_j is non-zero and a zero of $x^{t-2}\tilde{\sigma}_e(x)$, i.e., it is a zero of $\tilde{\sigma}_e(x)$. This gives $\alpha_j^2 = \alpha_{i_1}\alpha_{i_2}$. □

Recall that the attacker knows exactly if there is $j \in \operatorname{supp}(\tilde{\mathbf{e}}')$ with $\alpha_j \neq 0$ by virtue of Remark 4. As such the above proposition can be applied directly.

Remark 11. If the support elements $\alpha_1, \dots, \alpha_n$ and $\mathbf{e} \in \mathbb{F}_2^n$ with $\mathrm{wt}(\mathbf{e}) = 2$ are chosen uniformly at random, then the probability that there exists an α_j with $\alpha_j = \alpha_{i_1} + \alpha_{i_2}$ or $\alpha_j^2 = \alpha_{i_1}\alpha_{i_2}$ for $\mathrm{supp}(\mathbf{e}) = \{i_1, i_2\}$ is $\frac{n}{2^m}$. This means that the *success rate* for obtaining a polynomial equation for a zeroing fault injection (ELPz) is about $\frac{n}{2^m}$.

This probability is significantly greater than the success rate of the injections that directly target single or adjacent bits of the coefficients (ELPb), especially if $n \ll 2^m$. Our simulations confirm this observation.
The first step of the attack is now straightforward: generate constant and linear fault injections (where each injection requires two fault injections to (1) corrupt/zero a coefficient in the ELP and (2) skip the validity checks) and deduce linear and quadratic equations which have the common zero α.

3.4 Computing Alternative Secret Keys

Using many fault injections, we collect polynomial equations using Proposition 8 and Proposition 10 in a so-called **fault equation system** $L \subseteq \mathbb{F}_{2^m}[x_1, \dots, x_n]$ with the common zero α. Note that all these polynomials are either linear or quadratic. We require both linear and quadratic equations, as shown in Proposition 18. The first goal is to find a **support candidate set** $S_L \subseteq \mathbb{F}_{2^m}^n$ that is a subset of the set of the common zeros of L and contains a **support candidate** $\tilde{\alpha} \in S_L$ for which an irreducible polynomial $\tilde{g}$ of degree t exists with $\Gamma(\alpha, g) = \Gamma(\tilde{\alpha}, \tilde{g})$. To find such a support candidate set we follow the core solving process of the fault attack in [6, Section 6], summarized in the following proposition. Denote the **set of zeros** of $L \subseteq \mathbb{F}_{2^m}[x_1, \dots, x_n]$ by

$$\mathcal{Z}(L) = \{a \in \mathbb{F}_{2^m}^n \mid f(a) = 0 \text{ for all } f \in L\}.$$

Proposition 12 (Solving Fault Equations). *Let $L \subseteq \mathbb{F}_{2^m}[x_1, \dots, x_n]$ be a fault equation system. Consider the following sequence of instructions.*

(1) Reduce the linear polynomials in L (via Gaussian elimination).
(2) Substitute the leading terms in the quadratic polynomials, call the set of reduced quadratic equations $L_{red} \in \mathbb{F}_{2^m}[x_{i_1}, \dots, x_{i_s}]$.
(3) Fix one of the remaining indeterminates to 1, i.e., for some $i \in \{i_1, \dots, i_s\}$ add $x_i - 1$ to L_{red}.
(4) Find the set of zeros $\mathcal{Z}(L_{red}) \subseteq \mathbb{F}_{2^m}^s$ of L_{red} via Gröbner basis techniques.
(5) Extend the zeros in $\mathcal{Z}(L_{red})$ to elements of $\mathcal{Z}(L \cup \{x_i - 1\}) \subseteq \mathbb{F}_{2^m}^n$ using the linear polynomials, construct and return

$$S_L = \{\tilde{\alpha} \in \mathcal{Z}(L \cup \{x_i - 1\}) \mid \tilde{\alpha}_{j_1} \neq \tilde{\alpha}_{j_2} \text{ for } j_1 \neq j_2 \text{ and } \tilde{\alpha}_i = 1\}.$$

This computes a support candidate set S_L for L.

As soon as support candidates have been found, we check one by one, if they can be extended with an irreducible Goppa polynomial $\tilde{g}$ to generate the Goppa code $\mathit{\Gamma}(\alpha, g)$. One approach to do this, based upon [6, Algorithm 6.7], uses the fact that for every $c \in \mathit{\Gamma}(\alpha, g)$ we have

$$g \mid \sum_{i \in \mathrm{supp}(c)} \prod_{j \in \mathrm{supp}(c) \setminus \{i\}} (x - \alpha_j).$$

Proposition 13 (Finding Goppa Polynomials). *Let $\tilde{\alpha} \in \mathbb{F}_{2^m}^n$ with $\tilde{\alpha}_i \neq \tilde{\alpha}_j$ for $i \neq j$. For $s \geq 1$, consider the following sequence of instructions.*

(1) Let $\tilde{g} = 0$. Choose codewords $c_1, \ldots, c_s \in \mathit{\Gamma}(\alpha, g)$, for $j \in \{1, \ldots, s\}$ set $f_j = \sum_{i \in \mathrm{supp}(c)} \prod_{k \in \mathrm{supp}(c) \setminus \{i\}} (x - \alpha_k)$, and compute $h = \gcd(f_1, \ldots, f_s)$.
(2) Factorize h and collect all irreducible factors of degree t in a set G.
(3) For every $\hat{g} \in G$, check if $\mathit{\Gamma}(\tilde{\alpha}, \hat{g}) = \mathit{\Gamma}(\alpha, g)$ by comparing parity check matrices in systematic form. In that case let $\tilde{g} = \hat{g}$.
(4) Return $\tilde{g}$.

This is an algorithm that returns a non-zero $\tilde{g}$ if and only if there exists an irreducible polynomial $g' \in \mathbb{F}_{2^m}[x]$ with $\mathit{\Gamma}(\tilde{\alpha}, g') = \mathit{\Gamma}(\alpha, g)$. In that case we have $\mathit{\Gamma}(\tilde{\alpha}, \tilde{g}) = \mathit{\Gamma}(\alpha, g)$.

Proof. In case $\tilde{g}$ is non-zero, by step (3), we have $\mathit{\Gamma}(\tilde{\alpha}, \tilde{g}) = \mathit{\Gamma}(\alpha, g)$. Conversely, if there is an irreducible $g' \in \mathbb{F}_{2^m}[x]$ of degree t with $\mathit{\Gamma}(\tilde{\alpha}, g') = \mathit{\Gamma}(\alpha, g)$, then g' is an irreducible factor of h, i.e., $g' \in G$. This g' is processed in step (3) and ensures $\tilde{g} \neq 0$. This proves the claim. $\square$

With $s = 5$ our simulations showed that, in practice, we always have two cases in step (2): either $\deg(h) = 2t$ and G contains exactly one element, or $h = 1$ and $G = \emptyset$. Our implementation is optimized for this observation.

Improvements While the above already summarizes the overall solving procedure, we make a few additional remarks and optimizations.
The first observation is that in the first two steps of the Algorithm in Proposition 13 for finding Goppa polynomials only support elements $\tilde{\alpha}_j$ where $j \in \mathrm{supp}(c_i)$ with $i \in \{1, \ldots, s\}$ are used, i.e., to get a Goppa polynomial candidate $\tilde{g}$ not all elements $\tilde{\alpha}_i$ need to be known.

Remark 14 (Support Candidate Completion). Let $\tilde{\alpha} \in \mathbb{F}_{2^m}^n$ be a support candidate with irreducible Goppa polynomial $\tilde{g} \in \mathbb{F}_{2^m}[x]$ where $\mathit{\Gamma}(\alpha, g) = \mathit{\Gamma}(\tilde{\alpha}, \tilde{g})$. Let $J \subseteq \{1, \ldots, n\}$ be a set where $\tilde{\alpha}_j$ is known for $j \in J$.

(a) Let $c \in \mathit{\Gamma}(\alpha, g)$ be a code-word with $\mathrm{supp}(c) \setminus J = \{i\}$ for some $i \in \{1, \ldots, n\}$. Then one can determine $\tilde{\alpha}_i$ as the unique zero of the linear polynomial

$$1 + (x - y) \cdot \sum_{j \in \mathrm{supp}(c) \setminus \{i\}} (x - \tilde{\alpha}_j)^{-1} \in (\mathbb{F}_{2^m}[x]/\langle \tilde{g}(x) \rangle)[y],$$

since $\sum_{j \in \mathrm{supp}(c)} (x - \tilde{\alpha}_j)^{-1} = 0$ in $\mathbb{F}_{2^m}[x]/\langle \tilde{g} \rangle$ by definition of $\mathit{\Gamma}(\tilde{\alpha}, \tilde{g})$.
(b) In order to find (all) codewords $c \in \mathit{\Gamma}(\alpha, g)$ with $\mathrm{supp}(c) \setminus J = \{i\}$ one can compute an affine $\mathbb{F}_2$-basis of the intersection of the (affine) vector subspaces $\mathit{\Gamma}(\alpha, g)$ and $\{c \in \mathbb{F}_2^n \mid c_i = 1, c_j = 0 \text{ for } j \notin J \cup \{i\}\}$ by linear algebra.

(c) This allows us to exclude all indeterminates x_{i_k} that do not occur in L_{red} in step (2) of the solving procedure (Proposition 12) such that $\mathcal{Z}(L_{red})$ decreases in size by a factor of 2^m for every removed indeterminate. Then we find Goppa polynomial candidates $\tilde{g}$ using the first two steps of Proposition 13, where only known support elements are used (find appropriate code-words $c_1, \ldots, c_s$ similar to (b)). Using part (a), one can now find the missing support elements and construct support candidates $\tilde{\alpha}$. Finally, one can check if $\Gamma(\alpha, g) = \Gamma(\tilde{\alpha}, \tilde{g})$ as in Proposition 13, step (3).

This optimization only works if codewords as in (b) actually exist. Since many linear equations are required to make the solving of the quadratic polynomials in L_{red} feasible in the first place, only a few elements of the support candidates need to be found in the above way, i.e., the set J is rather large in practice. This also makes the existence of the required codewords highly likely. Recently, [11] followed the same approach, and showed that it suffices to know as little as $mt+1$ support elements to find both - the corresponding Goppa polynomial and the remaining support elements - under mild conditions.
Denote the *Frobenius automorphism* by $\psi\colon \mathbb{F}_{2^m} \to \mathbb{F}_{2^m}, a \mapsto a^2$ and consider the related automorphism $\Psi\colon \mathbb{F}_{2^m}[x_1, \ldots, x_n] \to \mathbb{F}_{2^m}[x_1, \ldots, x_n]$ where $\Psi(x_i) = x_i$ for $i \in \{1, \ldots, n\}$ and $\Psi|_{\mathbb{F}_{2^m}} = \psi$. For $\alpha \in \mathbb{F}_{2^m}$ we also write in the following $\varphi(\alpha) := (\varphi(\alpha_1), \ldots, \varphi(\alpha_n))$.

Remark 15. Let $L \subseteq \mathbb{F}_{2^m}[x_1, \ldots, x_n]$ be a fault equation system. Then all polynomials in L are homogeneous, their coefficients are contained in $\mathbb{F}_2$, and $\alpha \in \mathcal{Z}(L)$ is a common zero. In particular $\Psi(f) = f$ for all $f \in L$.
(a) For $a \in \mathbb{F}_{2^m}$ we have $a \cdot \alpha \in \mathcal{Z}(L)$, as for all $f \in L$: $f(a \cdot \alpha) = a^{\deg(f)} f(\alpha) = 0$.
(b) For $i \in \{0, \ldots, m-1\}$ we have $\psi^i(\alpha) \in \mathcal{Z}(L)$, since for all $f \in L$:

$$0 = \psi^i(0) = \psi^i(f(\alpha)) = \Psi^i(f)(\psi^i(\alpha)) = f(\psi^i(\alpha)).$$

This highlights that $\mathcal{Z}(L)$ contains quite many elements derived from α, and in fact all these are as useful to us as the support itself. This is a direct consequence of the following remark, proven in [9]. Let Ψ now operate on $\mathbb{F}_{2^m}[x]$.

Remark 16. Remember that $\Gamma(\alpha, g)$ is a binary irreducible Goppa code with $\deg(g) = t$ as before.
(a) For every $a \in \mathbb{F}_{2^m} \setminus \{0\}$ we have $\Gamma(\alpha, g(x)) = \Gamma(a \cdot \alpha, g(a^{-1}x))$.
(b) For every $i \in \{0, \ldots, m-1\}$ we have $\Gamma(\alpha, g(x)) = \Gamma(\psi^i(\alpha), \Psi^i(g))$
Moreover, $g(a^{-1}x)$ and $\Psi^i(g)$ are both irreducible polynomials of degree t.

Of all these zeros in $\mathcal{Z}(L)$ only a single one of them is sufficient to construct an alternative secret key, as for all of those $\tilde{\alpha}$ there is an irreducible $\tilde{g}$ of degree t with $\Gamma(\alpha, g) = \Gamma(\tilde{\alpha}, \tilde{g})$. In the following we discuss how one can decrease the size of the support candidate set S_L while still ensuring that one of these support candidates is present. Note that step (3) of our solving procedure (Proposition 12) already uses part (a) of the above remarks by fixing the coordinate of x_{i_1} to 1. This shrinks the support candidate set by a factor of 2^m. The next observation allows us to reduce this set by another factor of almost m.

Remark 17. Let $U \subseteq \mathbb{F}_{2^m}$ such that for every $a \in \mathbb{F}_{2^m}$ there exists $u \in U$ and $i \in \{0, \ldots, m-1\}$ with $a = \psi^i(u)$. By Remark 15.(b) and Remark 16.(b) there is $\tilde{\alpha} \in \mathcal{Z}(L)$ with $\tilde{\alpha}_i \neq \tilde{\alpha}_j$ for $i \neq j$ and $\tilde{\alpha}_s \in U$ for every $s \in \{1, \ldots, n\}$.
Instead of $\mathcal{Z}(L_{red})$ we can thus compute

$$\bigcup_{u \in U} \mathcal{Z}(L_{red} \cup \{x_{i_1} - u\}) = \{\tilde{\alpha} \in \mathcal{Z}(L_{red}) \mid \tilde{\alpha}_1 \in U\}$$

in step (4) of Proposition 12.

To find such a set $U \subseteq \mathbb{F}_{2^m}$ consider the following greedy algorithm: Choose $u \in \mathbb{F}_{2^m}$ and add it to U. Then repeat with $\mathbb{F}_{2^m} \setminus \{u, u^2, \ldots, u^{2^{m-1}} \mid u \in U\}$. For the proposed parameter sets, this gave sets U of size very close to $\frac{2^m}{m}$.
Our implementation computes each individual set of zeros with Gröbner basis techniques, to be precise it uses the SageMath function `variety`.
The following proposition indicates the necessity of the quadratic equations in our solving procedure by showing that it is impossible to find a small set of support candidates only from linear fault equations.

Proposition 18. *Assume that $n > 2^{m-1}$. Let $L \subseteq \mathbb{F}_{2^m}[x_1, \ldots, x_n]$ be a fault equation system consisting only of linear polynomials. Then L contains less than $n - m$ linearly independent polynomials.*

Proof. We know from Remark 15 that for all $j \in \{0, ..., m-1\}$ the tuple $z_j = \psi^j(\alpha)$ is a zero of the polynomials in L. We show that $(z_0, ..., z_{m-1})$ is $\mathbb{F}_2$-linear independent in $\mathbb{F}_{2^m}^n$. The rank theorem then shows that L contains less than $n - m$ linearly independent polynomials. Let $b_0, ..., b_{m-1} \in \mathbb{F}_2$ with $b_0 z_0 + ... + b_{m-1} z_{m-1} = 0$. Then we have $b_0 \alpha_i + ... + b_{m-1} \alpha_i^{2^{m-1}} = 0$ for $i \in \{1, \ldots, n\}$. Consider the polynomial $f(x) = b_0 x + b_1 x^2 + ... + b_{m-1} x^{2^{m-1}} \in \mathbb{F}_{2^m}[x]$. Suppose f is non-zero, then it has at most $\deg(f) \leq 2^{m-1}$ roots, but $f(\alpha_i) = 0$ for all $i \in \{1, \ldots, n\}$. With $n > 2^{m-1}$ we get a contradiction. This shows $f = 0$, and thus $b_0 = \cdots = b_{m-1} = 0$ and the linear independence follows by definition. □

Note that the condition $n > 2^{m-1}$ is satisfied by all proposed parameter sets (see Table 4). So for the set of reduced quadratic polynomials $L_{red} \in \mathbb{F}_{2^m}[x_{i_1}, \ldots, x_{i_s}]$ in step (2) of Proposition 12 we have $s \geq m$.

4 Fault Attack Implementation and Simulation

In this section, we demonstrate the viability of the key-recovery attack. We first use a C-implementation to simulate the attack (Section 4.1). For this we inject faulty variable values directly in software. We simulate the inputs and corresponding hashed outputs of the faulty decapsulation procedure. The de-hashing of these is described separately in Section 4.2, leading to the system of polynomial equations. This is solved to obtain an alternative secret key as described in Section 3.4 by a program written in Python3 using SageMath [26].

An attacker cannot directly modify the software execution. Instead, there are different ways to conduct a fault attack, e.g., using a laser to corrupt hardware memory elements in a processor. To investigate whether this allows to inject the specific faults required for the presented attack as were identified at software-level, we execute the cryptosystem as software on a virtual prototype (VP) (Section 4.3). The VP implements the RISC-V Instruction Set Architecture (ISA) and allows us to inject faults into the hardware of the processor in order to study how they impact the executing software. For our software-error-model-based fault injection attack, we first analyze the binary to find the program sections calculating the ELP and processing the validity checks. The disassembly and its required alteration gives us the fault positions necessary to produce the identified faulty variable values. The necessary hardware fault attacks are then simulated on two levels; First, a fast ISA-level simulation assures that the hardware faults produce exploitable output. Second, a Register Transfer Level (RTL) simulation yields practicability of the fault attack w.r.t. a real CPU core's micro-architecture.

4.1 Key-Recovery Simulation

We simulated the attack of Section 3 in C and SageMath code. To speed up that simulation at C-level we work on AMD64 machines with the vector-accelerated AVX-2 implementation. An overview of the software simulation is given in Algorithm 6.
To model the attack, we adapt the implementation of the cryptosystem to include the effects of the ELPB, ELPZ and VCB faults. For the fault injections on the ELP, we have identified the following lines of code as injection points. The fault injection on the ELP happens between the function calls `bm(locator, s)` and `root(images, locator, L)` in `decrypt.c`. Fault injections on the ELP are modelled as bitwise operations on one of its coefficients. This way, the ELPB fault injection that sets two adjacent bits is implemented by setting one coefficient $a \to a \,\mathrm{OR}\, \zeta$, where ζ is an m-bit array containing only zeros except for two adjacent entries. The ELPZ fault injection is implemented by replacing all entries of one ELP coefficient with zeroes. Note that the fault value ξ corresponding to these injections as defined in Remark 2 is unknown, as it depends on the value of a. To skip the validity checks the variable `m` in file `operations.c` and in function `crypto_kem_dec_faulty` in the line `m = ret_decrypt | ret_confirm` are forced to 0. This gives $\mathcal{H}(1, \tilde{\mathbf{e}}', C)$ as output of the C-code for further analysis (see next Section 4.2).[3]
The simulation code is called repeatedly for different chosen ciphertexts and faults ζ to build a system of equations using Propositions 8 and 10, that can be solved with the methods from Section 3.4 to obtain an (alternative) secret key. To obtain linear equations in the support elements, faults are injected into the constant term of the ELP. In ELPB mode, we start with ζ

[3] For the purposes of verifying the fault attack, the simulator also directly gives $\tilde{\mathbf{e}}'$ as output, sparing us the computational effort of de-hashing $\mathcal{H}(1, \tilde{\mathbf{e}}', C) \to \tilde{\mathbf{e}}'$.

having the two least significant bits non-zero. Then we generate faulty session keys from ciphertexts corresponding to plaintext vectors $\mathbf{e}$ with $\mathrm{wt}(\mathbf{e}) = 2$ and $\mathrm{supp}(\mathbf{e}) \in \{\{n-1, 0\}, \{0, 1\}, \{1, 2\}, \dots\}$. This is repeated for faults ζ with non-zero bits in other adjacent positions, until the resulting system of equations contains equations involving all the support elements. In ELPz mode, there is only one way of injecting a fault, so that instead of different fault values ζ, ciphertexts corresponding to plaintext vectors $\mathbf{e}$ with $\mathrm{wt}(\mathbf{e}) = 2$ with increasing distance between the non-zero support elements $\mathrm{supp}(\mathbf{e}) \in \{\{n-1, 1\}, \{0, 2\}, \{1, 3\}, \dots\}$ are used to obtain a sufficiently large system of equations (this is also done in the ELPb-case if the number of possible ζ is exhausted before finding sufficiently many equations). The same procedure is used to inject faults on the linear term of the ELP in order to obtain quadratic equations in the support elements, finishing after an empirically determined fixed number of equations has been obtained. To confirm that the attack is working, we ran simulations of 100 random public/private key pairs, for several sets of parameters where $n \in \{3488, 6688, 8192\}$ (see Table 4). The average number of required fault injections for a successful attack on the different parameter sets are shown in Tables 1 and 2 for the fault modes ELPb and ELPz respectively. The ELPz-mode requires significantly fewer fault injections to complete the attack (compare with Remark 11). Parameter sets with smaller ratio $\frac{n}{2^m}$ also require more injections, as indicated by Remark 9. We find that the SageMath code usually takes only minutes to obtain an alternative secret key from the system of polynomial equations on an office computer.

Table 1: Arithmetic Mean out of 100 simulations for ELPb

	CAT I n=3488, m=12, t=64	CAT V n=6688, m=13, t=128	CAT V n=8192, m=13, t=128
nr of constant injections	31759	70700	56991
nr of linear equations	8627	17649	21343
nr of linear injections	293	564	266
nr of quadratic equations	80	140	100

Table 2: Arithmetic Mean out of 100 simulations for ELPz

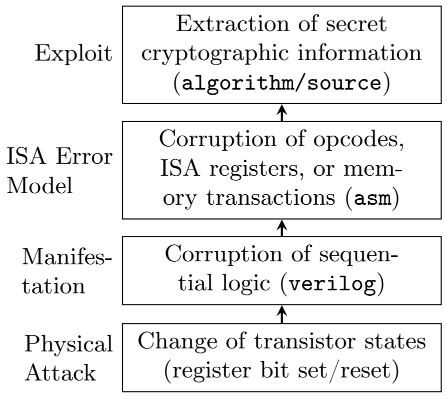
	CAT I n=3488, m=12, t=64	CAT V n=6688, m=13, t=128	CAT V n=8192, m=13, t=128
nr of constant injections	8030	16516	8944
nr of linear equations	6836	13482	8941
nr of linear injections	94	121	100
nr of quadratic equations	80	100	100

4.2 De-hashing: Obtaining the faulty error vector from hash output

As output of the simulation in Section 4.1 we generate two files containing hashes $\tilde{K}' = \mathcal{H}(1, \tilde{\mathbf{e}}', C)$ with $(\mathbf{e}, \tilde{\mathbf{e}}')$ defining linear or quadratic equations in the support elements. Thanks to the small weight of the error vectors $\tilde{\mathbf{e}}'$, we can determine them from the hashes in a brute-force manner as follows.
First, we determine whether the zero element is part of the support set (Remark 1) and determine its index if present, according to Remark 4. This requires only one fault injection, giving the output $\mathcal{H}(1, \mathbf{e}', C)$ for $C = (\mathbf{0}, \mathcal{H}(2, \mathbf{0}))$, with $\mathrm{wt}(\mathbf{e}') \in \{0, 1\}$. The support $\mathrm{supp}(\mathbf{e}')$ specifies the index of the zero element in the support set, if it is present. It is determined from the hash by calculating all $n + 1$ possible hashes until the match with the output is found. Next, for every $\tilde{K}'$ we calculate the hashes $\mathcal{H}(1, \mathbf{v}, C)$ for the chosen ciphertext $C = (\mathbf{c_0}, \mathcal{H}(2, \mathbf{e}))$ and all possible $\tilde{\mathbf{e}}'$, as described in Remark 3. When a match is found, $\tilde{\mathbf{e}}'$ has been determined.
We run the de-hashing on a computer with AMD EPYC 7543P processor running up to 3.3 GHz using 32 cores (64 threads) on Arch Linux with kernel 5.19.7. For the ELPB case we have about $\binom{n}{2}$ different $\tilde{\mathbf{e}}'$ to check for every hash output (number of constant injections plus number of linear injections in Table 1). Depending on the cryptosystem and its parameters, the total runtime on our system spans a few hours up to a few days. For the ELPZ case we have about $\binom{n}{1} = n$ different $\tilde{\mathbf{e}}'$ to check for every hash output (number of constant injections plus number of linear injections in Table 2). The running time on our system is a few seconds.

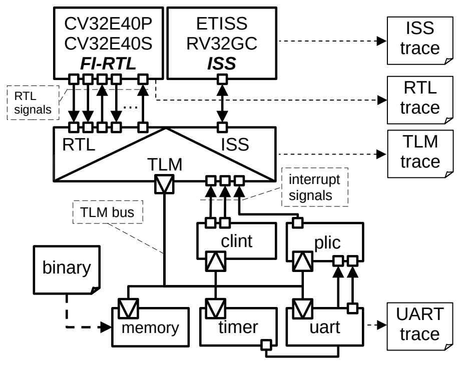
4.3 Simulation at Register Transfer Level

A fault injection simulation campaign and its cost w.r.t. simulation time is dependent on the abstraction level, such that limiting the fault space is crucial. Figure 2a shows our approach to identify the fault injection points to further evaluate the feasibility of an attack by exhaustive search campaign on RTL. Exhaustive search means applying the fault model (single bit set/reset) to all possible bits at all possible simulation steps (clock cycles). To gain more detailed results quicker, we align the simulation abstraction level with the fault abstraction level: We start by formulating a software test for *Exploit*, so that we have a defined faulty reference output of the targeted system. This can be directly manipulated to the source code on C-level. We continue on the *ISA Error Model* level where we run the attacked system on the VP with a fast ISA Level Simulation (RISC-V) to transfer the source code faults into ISA specific errors. This is basically assembly level. From there we gain a more narrow timing information as to where faults in the processor core can result in the wanted exploit, e.g., instruction data manipulations or register value modifications. After that, we take this set of narrow program sections and feed it into an RTL fault simulation, a specific micro architecture, to evaluate the *Manifestation*. This gives us the benefit to do an exhaustive search on bit level to reproduce the exploits while focusing on program sections that were earlier identified as critical. Depending on the *Physical Attack* we want to mimic, we can directly transfer the findings from the RTL simulation (Laser Fault analogue) or make predictions (vulnerable micro architecture) that could become an attack scenario for other physical attacks, e.g., clock or voltage glitches.



Exploit
Extraction of secret cryptographic information (algorithm/source)
ISA Error Model
Corruption of opcodes, ISA registers, or memory transactions (asm)
Manifestation
Corruption of sequential logic (verilog)
Physical Attack
Change of transistor states (register bit set/reset)

(a) Fault Attack Effect Abstraction



(b) Register Transfer or Instruction Set Level Fault Injection Virtual Prototype

Fig. 2: Experimental Setup for ISS/RTL Fault Injection Evaluation

Figure 2b shows the fault injection VP used to evaluate the vulnerability of two open source RISC-V cores, the OpenHWGroup's CV32E40P [19], formerly known as RI5CY [8,7], and its security focused derivative CV32E40S [20]. The RTL model is generated with the open source Verilog hardware description language to C++/SystemC synthesis tool Verilator [24]. The generated RTL is then modified with a LLVM-based automated source code transformation tool. The modified RTL yields a clock cycle accurate simulation with fault injection capability into sequential storage elements, i.e., flip-flop and latch equivalents. The RTL's SystemC ports are connected to the Transaction Level Modeling (TLM) peripherals and memory through an RTL to TLM transactor that implements the required bus protocol. The VP's memory may host any cross-compiled RISC-V binary suitable for the used core and is initialized before simulation start by an binary loader. TLM transactions, certain RTL signals, and the peripheral output are logged in respective trace files. This allows to evaluate the effect an injected fault has on the system's behavior compared to a reference simulation executing the same binary without fault injection. Through the RTL fault injection, we are able to simulate faults on the manifestation level of our fault attack abstraction model as shown in Fig. 2a. We deem a fault attack experiment as successful in a security scenario, if its manifestation results in an undetected exploit defined by the first step of our simulations, see Section 4.1.
Table 3a shows the simulation results for the output mask fault attack to bypass the cryptosytem's validity check. For CV32E40P, in total $4,351$ micro architectural and $1,650$ ISA related bits were faulted over a clock cycle period of

Table 3: Single-Bit RTL Fault Injection Results

(a) Validity Check Fault Scenario

core: CV32E40-	P	S
clock cycles[a]	301	301
micro-arch. bits	4,351	12,938
ISA register bits[b]	1,650	3,648
nr experiments	1,806,301	4,992,386
exploits output mask reset N_{VCB}	114	57
nr unique faulty bits	93	39

[a] identified by ISA simulation
[b] without performance counters

(b) ELP Coefficient Fault Scenario

core: CV32E40-	P	S
clock cycles[a]	521	521
micro-arch. bits	4,351	12,938
ISA register bits[b]	1,650	3,648
nr experiments	3,126,521	8,641,306
nr experiments coeff. bit corruption N_{ELP_B}	69	57
nr unique faulty bits	35	29
nr experiments coeff. zeroing N_{ELP_Z}	508	212
nr unique faulty bits	225	94

301 cycles. The cycle range reflects vulnerable sections of the cryptosystem. 93 unique bits were found to be capable to result in a bypass of the validity check in 114 experiments. None of which resemble ISA registers, e.g. RISC-V GPRs. Although CV3240S contains more sequential logic, thus, injectable bits, the number of successful fault experiments is much lower at 57. Table 3b shows the simulation results for the ELP fault attacks over a clock cycle period of 521 cycles. 69 experiments on 35 unique bits resulted in a faulted ELP coefficient of which a small number was directly injected into GPRs. For the ELP coefficient zeroing exploit, 225 unique bits were the reason for a total number of 508 faulty scenarios in CV32E40P. Most of these fault injections, all of which in the core's micro architecture, manifested as manipulation of the memory store instructions for the respective ELP coefficient. Furthermore, 47 bits and 17 bits, for CV3240P and CV32E40S respectively, are equal for the validity check bypass and the ELP coefficient zeroing attack. This can result in a significantly less complex fault injection setup, e.g., by only requiring one laser injection source.

Figure 3 shows a plot of the three scenarios for both cores. Overall, the more secure CV32E40S is harder to fault. Reasons for this can be found in its hardware-based countermeasures, such as an ECC protected register file eliminating all bit errors up to two by raising a security alert. The remaining faults can be mostly traced back to in-pipeline faults, e.g., modified instruction code or operands for which no error protection is deployed. Furthermore, in both cores faulting the fetch instruction address can result in replacing the original instruction code without modifying the in-pipeline program counter, thus, bypassing the program counter validity checks of CV32E40S. Here, configuring the Memory Protection Unit (MPU) with executable memory ranges can help to mitigate this easy instruction replacement. For our RTL fault analysis, we did not make use of CV32E40S's side-channel countermeasure that randomly inserts dummy in-

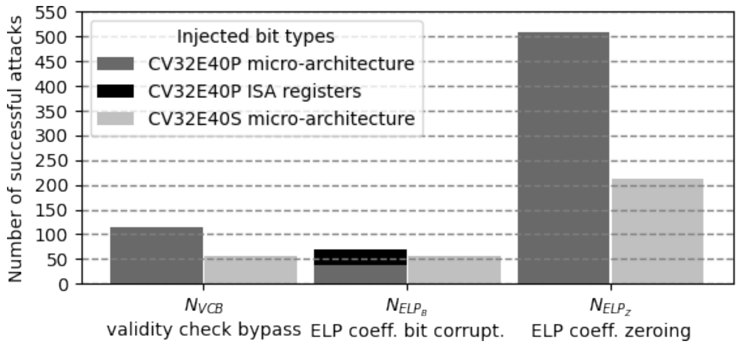


Fig. 3: Single-Bit RTL Fault Injection Results

structions in the executed code, but would consider it as viable countermeasure. Feeding this unit with a true random seed, would make our proposed attack significantly harder. The attack requires delicate timing of the fault injection which would become harder due to the unpredictable execution time.

5 Summary

We have presented a key-recovery fault injection attack on Classic McEliece. Two fault injections are necessary for the attack. CCA2-security is bypassed by one fault injection in combination with brute-force de-hashing of possible faulty session keys. The other fault injection targets the error-locator polynomial (ELP) to find the support set of the Goppa code, which is part of the secret key. We use the faulty output of the decapsulation function to construct a polynomial system of equations whose unknowns are the elements of the support set. Solving these equations, we obtain their values, which can be used with the public key to generate a matching Goppa polynomial. Together, this can be used as an alternative secret key that generates valid session keys. We have verified the attack, simulating it using C and SageMath code. Simulations for several instances of Classic McEliece show that parameters with n close to 2^m are particularly easy to attack. We evaluated the vulnerability of two RISC-V cores, simulating the fault injections on virtual prototypes at RTL. On specific hardware and with knowledge of the core structure, the two required faults may be injected at the same location, simplifying the execution of the attack. The next steps are to deploy our attack on real hardware.
The simulation code in C and SageMath can be found online at GitHub: `https://github.com/sahpir/attackFI-ClassicMcEliece`.

Acknowledgements This research was partly funded by the Bavarian State Ministry for Economic Affairs as part of the funding program *Information and Communication Technology* through the project MITHRIL, grant number IUK623.

A Appendix

A.1 Classic McEliece KEM Algorithms and Parameters

Here we show the parameters and algorithms used in Classic McEliece KEM [3]. In Algortihm 6 we list the steps executed for the attack simulation.

Table 4: Parameter sets of Classic McEliece KEM [3]

Security Category[4]	n	m	t
CAT 1	3488	12	64
CAT 3	4608	13	96
CAT 5	6688	13	128
CAT 5	6960	13	119
CAT 5	8192	13	128

Algorithm 1 Key Generation

```
Input: Parameters $m, t, n \leq 2^m$, $f(z) \in \mathbb{F}_2[z]$ irreducible of degree $m$.
Output: Secret key $(s, \gamma)$, public key $\mathbf{T}$.
1: Generate a uniform random monic irreducible polynomial $g(x) \in \mathbb{F}_{2^m}[x]$ of degree $t$.
2: Select a uniform random sequence $L = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ of $n$ distinct elements in $\mathbb{F}_{2^m}$.
3: Compute the $t \times n$ matrix $\mathbf{H} = \{h_{ij}\}$ over $\mathbb{F}_{2^m}$ where $h_{ij} = \alpha_j^{i-1}/g(\alpha_j)$ for $i \in [t], j \in [n]$, i.e.,
   $$\mathbf{H} = \begin{pmatrix} \frac{1}{g(\alpha_1)} & \frac{1}{g(\alpha_2)} & \cdots & \frac{1}{g(\alpha_n)} \\ \frac{\alpha_1}{g(\alpha_1)} & \frac{\alpha_2}{g(\alpha_2)} & \cdots & \frac{\alpha_n}{g(\alpha_n)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\alpha_1^{t-1}}{g(\alpha_1)} & \frac{\alpha_2^{t-1}}{g(\alpha_2)} & \cdots & \frac{\alpha_n^{t-1}}{g(\alpha_n)} \end{pmatrix}.$$
4: Form matrix $\hat{\mathbf{H}} \in \mathbb{F}_2^{mt \times n}$ by replacing each entry $c_0 + c_1 z + ... + c_{m-1} z^{m-1} \in \mathbb{F}_2[z]/\langle f(z) \rangle \cong \mathbb{F}_{2^m}$ of $\mathbf{H} \in \mathbb{F}_{2^m}^{t \times n}$ with a column of $m$ bits $c_0, c_1, ..., c_{m-1}$.
5: Reduce $\hat{\mathbf{H}}$ to systematic form $(\mathbf{I_{n-k}}|\mathbf{T})$ where $\mathbf{I_{n-k}}$ is an identity matrix of $(n-k) \times (n-k)$ and $k = n - mt$
6: If Step 5 fails, go back to Step 1
7: Generate a uniform random $n$-bit string $s$ (needed if decapsulation fails).
8: Output secret key: $(s, \gamma)$ with $\gamma = (g(x), \alpha_1, \alpha_2, ..., \alpha_n)$
9: Output public key: $\mathbf{T} \in \mathbb{F}_2^{(n-k) \times k}$
```

[4] NIST defines security categories in [15] with the requirement "Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 128-bit key (e.g. AES128)" with 128/192/256-bit key size corresponding to CAT-1/3/5, respectively.

Algorithm 2 Encoding

```
Input: Weight-$t$ row vector $\mathbf{e} \in \mathbb{F}_2^n$, public key $\mathbf{T}$
Output: Syndrome $\mathbf{c_0}$
1: Construct $\mathbf{H_{sys}} = (\mathbf{I_{n-k}}|\mathbf{T})$.
2: Compute and output $\mathbf{c_0} = \mathbf{e}\mathbf{H_{sys}}^\top \in \mathbb{F}_2^{n-k}$
```

Algorithm 3 Encapsulation

```
Input: Public key $\mathbf{T}$
Output: Session key $K$, chiphertext $C$
1: Generate a uniform random vector $\mathbf{e} \in \mathbb{F}_2^n$ of Hamming weight $t$.
2: Use the encoding subroutine defined in Algorithm 2 on $\mathbf{e}$ and public key $\mathbf{T}$ to
   compute $\mathbf{c_0}$
3: Compute $C_1 = \mathcal{H}(2, \mathbf{e})$ (input to hash function is a concatenation of 2 and $\mathbf{e}$ as a
   1-byte and $\lceil n/8 \rceil$-byte string representation).
4: Put $C = (C_0, C_1)$ .
5: Compute $K = \mathcal{H}(1, \mathbf{e}, C)$ (input to hash function is a concatenation of 1, $\mathbf{e}$ and $C$
   as a 1-byte, $\lceil n/8 \rceil$-byte and $\lceil mt/8 \rceil + \lceil \ell/8 \rceil$ string representation).
```

Algorithm 4 Decoding

```
Input: vector $\mathbf{c_0} \in \mathbb{F}_2^{n-k}$, private key $(s, \gamma)$
Output: Weight-$t$ vector $\mathbf{e}' \in \mathbb{F}_2^n$ or Failure
1: Extend $\mathbf{c_0}$ to $\mathbf{v} = (\mathbf{c_0}, 0, ..., 0) \in \mathbb{F}_2^n$ by appending $k$ zeros.
2: Find the unique codeword $\mathbf{c}$ in the Goppa code defined by $\gamma$ that is at distance $\leq t$
   from $\mathbf{v}$. If there is no such codeword, return failure.
3: Set $\mathbf{e}' = \mathbf{v} + \mathbf{c}$.
4: If $w(\mathbf{e}') = t$ and $\mathbf{c_0} = \mathbf{e}'\mathbf{H_{sys}}^\top$, return $\mathbf{e}'$, otherwise return Failure.
```

Algorithm 5 Decapsulation

```
Input: Ciphertext $C$, Private key $(\mathbf{s}, \gamma)$
Output: Session key $K'$
1: Split the ciphertext $C$ as $(\mathbf{c_0}, C_1)$ with $\mathbf{c_0} \in \mathbb{F}_2^{n-k}$ and hash $C_1$.
2: Set $b \leftarrow 1$.
3: Use the decoding subroutine defined in Algorithm 4 on $\mathbf{c_0}$ and private key $\gamma$ to
   compute $\mathbf{e}'$. If the subroutine returns Failure, set $\mathbf{e}' \leftarrow \mathbf{s}$ and $b \leftarrow 0$.
4: Compute $C_1' = \mathcal{H}(2, \mathbf{e}')$ (input to hash function is concatenation of 2 and $\mathbf{e}'$ as a
   1-byte and $\lceil n/8 \rceil$-byte string representation).
5: If $C_1' \neq C_1$, set $\mathbf{e}' \leftarrow \mathbf{s}$ and $b \leftarrow 0$.
6: Compute $K' = \mathcal{H}(b, \mathbf{e}', C)$ (input to hash function is concatenation of $b$, $\mathbf{e}'$ and $C$
   as a 1-byte, $\lceil n/8 \rceil$-byte and $\lceil mt/8 \rceil + \lceil \ell/8 \rceil$ string representation).
7: Output session key $K'$.
```

Algorithm 6 Attack Simulation on C-level

```
1: Specify a ciphertext $C$ as input for the decapsulation function as follows:
     i. Choose a plaintext $\mathbf{e}$ of Hamming weight $\mathrm{wt}(\mathbf{e}) = 2$.
    ii. Calculate the ciphertext $C = (\mathbf{c_0}, C_1) = (\mathbf{e}\mathbf{H_{sys}}^\top, \mathcal{H}(2, \mathbf{e}))$
2: Inject a fault into the error locator polynomial (ELP) as follows:
     i. Fix a fault value of $\zeta \in \mathbb{F}_{2^m}$
    ii. Start the decapsulation process and let the Berlekamp-Massey algorithm calculate the ELP (in file *decrypt.c*).
   iii. Inject a constant or quadratic fault into the ELP (see Definition 6).
3: Inject a fault and reset the variable called $m$ during decapsulation such that the following comparisons are bypassed (in file *operations.c*)
    a) Skip the comparison $\tilde{\mathbf{e}}'\mathbf{H_{sys}}^\top = \mathbf{e}\mathbf{H_{sys}}^\top$ (Alternative: in file *decrypt.c* clear 8-bit variable *ret_ decrypt* during decapsulation).
    b) Skip the comparison $C_1' = \mathcal{H}(2, \tilde{\mathbf{e}}') = \mathcal{H}(2, \mathbf{e}) = C_1$ (Alternative: in file *operations.c* clear 8-bit variable *ret_ confirm*).
4: Reconstruct $\tilde{\mathbf{e}}'$ from the output $\tilde{K'} = \mathcal{H}(1, \tilde{\mathbf{e}}', (\mathbf{e}\mathbf{H_{sys}}^\top, \mathcal{H}(2, \mathbf{e})))$ of the decapsulation function as described in Section 4.2.
5: Calculate an alternative secret key as described in Section 3.4.
```

References

1. Berlekamp, E.R.: Nonbinary BCH decoding (Abstr.). IEEE Transactions on Information Theory **14**(2), 242–242 (1968). `https://doi.org/10.1109/TIT.1968.1054109` (Cited on page 5.)
2. Berlekamp, E.R., McEliece, R.J., van Tilborg, H.C.A.: On the Inherent Intractability of Certain Coding Problems. IEEE Transactions on Information Theory **24**(3), 384–386 (May 1978). `https://doi.org/10.1109/TIT.1978.1055873` (Cited on page 3.)
3. Bernstein, D.J., Chou, T., Lange, T., von Maurich, I., Misoczki, R., Niederhagen, R., Persichetti, E., Peters, C., Schwabe, P., Sendrier, N., Szefer, J., Wang, W.: Classic McEliece: NIST submission (2020), `https://classic.mceliece.org/nist.html`, (accessed Sep. 19, 2022) (Cited on pages 2, 3, 5, and 21.)
4. Cayrel, P.L., Colombier, B., Drăgoi, V.F., Menu, A., Bossuet, L.: Message-Recovery Laser Fault Injection Attack on the Classic McEliece Cryptosystem. In: Advances in Cryptology – EUROCRYPT 2021. pp. 438–467. Lecture Notes in Computer Science, Springer International Publishing, Cham (2021). `https://doi.org/10.1007/978-3-030-77886-6_15` (Cited on page 2.)
5. Colombier, B., Dragoi, V.F., Cayrel, P.L., Grosso, V.: Message-recovery Profiled Side-channel Attack on the Classic McEliece Cryptosystem. Cryptology ePrint Archive, Paper 2022/125 (2022), `https://eprint.iacr.org/2022/125` (Cited on page 2.)
6. Danner, J., Kreuzer, M.: A Fault Attack on the Niederreiter Cryptosystem using Binary Irreducible Goppa Codes. Journal of Groups, Complexity, Cryptology **12**(1), 2:1–2:20 (Mar 2020). `https://doi.org/10.46298/jgcc.2020.12.1.6074`, `https://arxiv.org/abs/2002.01455` (Cited on pages 2, 6, 8, 11, and 12.)
7. Davide Schiavone, P., Conti, F., Rossi, D., Gautschi, M., Pullini, A., Flamand, E., Benini, L.: Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In: International Symposium on Power

and Timing Modeling, Optimization and Simulation (PATMOS). vol. 27, pp. 1–8 (2017). https://doi.org/10.1109/PATMOS.2017.8106976 (Cited on page 18.)
8. Gautschi, M., Schiavone, P.D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gürkaynak, F.K., Benini, L.: Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **25**(10), 2700–2713 (2017). https://doi.org/10.1109/TVLSI.2017.2654506 (Cited on page 18.)
9. Gibson, J.K.: Equivalent Goppa Codes and Trapdoors to McEliece's Public Key Cryptosystem. In: Advances in Cryptology — EUROCRYPT '91. pp. 517–521. Springer, Berlin, Heidelberg (1991) (Cited on page 13.)
10. Guo, Q., Johansson, A., Johansson, T.: A Key-Recovery Side-Channel Attack on Classic McEliece. Cryptology ePrint Archive, Paper 2022/514 (2022), https://eprint.iacr.org/2022/514 (Cited on page 2.)
11. Kirshanova, E., May, A.: Decoding McEliece with a Hint - Secret Goppa Key Parts Reveal Everything. Cryptology ePrint Archive, Paper 2022/525 (2022), https://eprint.iacr.org/2022/525 (Cited on pages 2 and 13.)
12. MacWilliams, F., Sloane, N.: The Theory of Error-Correcting Codes, vol. 16. North-Holland, 1st edn. (1983), isbn: 978-0-444-85193-2 (Cited on page 5.)
13. Massey, J.L.: Shift-register synthesis and BCH decoding. IEEE Transactions on Information Theory **15**(1), 122–127 (1969). https://doi.org/10.1109/TIT.1969.1054260 (Cited on page 5.)
14. McEliece, R.J.: A Public-Key Cryptosystem Based On Algebraic Coding Theory. Deep Space Network Progress Report **44**, 114–116 (Jan 1978) (Cited on page 2.)
15. National Institute for Standards and Technology: Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process (Dec 2016), https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf, (accessed Sep. 19, 2022) (Cited on page 21.)
16. National Institute for Standards and Technology - Computer Security Division, Information Technology Laboratory: Post-Quantum Cryptography Standardization (Jan 2017), https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization, (accessed Sep. 19, 2022) (Cited on page 1.)
17. National Institute of Standards: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Tech. Rep. Federal Information Processing Standard (FIPS) 202, U.S. Department of Commerce (Aug 2015). https://doi.org/10.6028/NIST.FIPS.202, https://csrc.nist.gov/publications/detail/fips/202/final (Cited on page 3.)
18. Niederreiter, H.: Knapsack-type Cryptosystems and Algebraic Coding Theory. Problems Control and Inf. Theory **15(2)**, 159–166 (1986) (Cited on pages 2 and 3.)
19. OpenHW Group: CV32E40P - GitHub, https://github.com/openhwgroup/cv32e40p, (accessed Aug. 25, 2022) (Cited on page 18.)
20. OpenHW Group: CV32E40S - GitHub, https://github.com/openhwgroup/cv32e40s, (accessed Aug. 25, 2022) (Cited on page 18.)
21. Patterson, N.: The algebraic decoding of Goppa codes. IEEE Transactions on Information Theory **21**(2), 203–207 (1975). https://doi.org/10.1109/TIT.1975.1055350 (Cited on page 5.)
22. Selmke, B., Heyszl, J., Sigl, G.: Attack on a DFA Protected AES by Simultaneous Laser Fault Injections. In: Workshop on Fault Diagnosis and Tolerance in Cryptog-

raphy (FDTC). pp. 36–46 (Aug 2016). https://doi.org/10.1109/FDTC.2016.16 (Cited on pages 2 and 7.)

23. Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM J. Comput. **26**(5), 1484–1509 (1997). https://doi.org/https://doi.org/10.1137/S0097539795293172, https://arxiv.org/abs/quant-ph/9508027 (Cited on page 1.)
24. Snyder, W.: Verilator, https://www.veripool.org/verilator/, (accessed Aug. 25, 2022) (Cited on page 18.)
25. Sugiyama, Y., Kasahara, M., Hirasawa, S., Namekawa, T.: A Method for Solving Key Equation for Decoding Goppa Codes. Inform. Control **27**(1), 87–99 (1975). https://doi.org/https://doi.org/10.1016/S0019-9958(75)90090-X (Cited on page 5.)
26. The Sage Developers: SageMath, the Sage Mathematics Software System (Version 9.5) (2022), https://www.sagemath.org (Cited on page 14.)
27. Xagawa, K., Ito, A., Ueno, R., Takahashi, J., Homma, N.: Fault-Injection Attacks Against NIST's Post-Quantum Cryptography Round 3 KEM Candidates. In: Advances in Cryptology – ASIACRYPT 2021. pp. 33–61. Lecture Notes in Computer Science, Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-92075-3_2 (Cited on pages 2 and 6.)