Leaky McEliece: Secret Key Recovery From Highly Erroneous Side-Channel Information

Marcus Brinkmann[1], Chitchanok Chuengsatiansup[2], Alexander May[1], Julian Nowakowski[1], and Yuval Yarom[1]

[1] Ruhr University Bochum
[2] The University of Melbourne

Abstract. The McEliece cryptosystem is a strong contender for post-quantum schemes, including key encapsulation for confidentiality of key exchanges in network protocols.

A McEliece secret key is a structured parity check matrix that is transformed via Gaussian elimination into an unstructured public key. We show that this transformation is a highly critical operation with respect to side-channel leakage. We assume leakage of the elementary row operations during Gaussian elimination, motivated by actual implementations of McEliece in real world cryptographic libraries (Classic McEliece and Botan).

We propose a novel algorithm to reconstruct a secret key from its public key with information from a Gaussian transformation leak. Even if the obtained side-channel leakage is extremely noisy, i.e., each bit can be flipped with probability as high as $\tau \approx 0.4$, our algorithm still succeeds to recover the secret key in a matter of minutes for all proposed (Classic) McEliece instantiations. Remarkably, for high-security McEliece parameters, our attack is more powerful in the sense that it can tolerate even larger τ.

Technically, we introduce a novel cryptanalytic decoding technique that exploits the high redundancy exhibited in the McEliece secret key. This allows our decoding routine to succeed in reconstructing each column of the secret key successively. Our result stresses the necessity to well protect highly structured code-based schemes such as McEliece against side-channel leakage.

Keywords: McEliece · Gaussian elimination · Side-channel leakage · Key recovery with hints

1 Introduction

The seminal work of Kocher [27] demonstrated that implementations of cryptographic schemes may leak intermediate states of the algorithm, compromising the security of the schemes. Since then, many so called *side-channel attacks* have been demonstrated, exploiting various side channels [18, 27, 30, 33] and breaking a large number of implementations [10, 11, 26, 33, 34, 43].

Because side channels are usually accidental, i.e., they are not designed to leak, information obtained through side channels is often noisy and incomplete. Over the years several methods have been devised to recover the full secret, exploiting the inherent redundancy in some cryptographic schemes [15]. Most of the research in this area focuses on traditional schemes, such as AES [36], RSA [12,13,21–23,35,45], and variations of the Digital Signature Algorithm [24, 31,32].

Within the context of post-quantum schemes, secret recovery from noisy or partial information has also seen significant interest, e.g., with regard to side channels in HQC [25,37], Kyber and New Hope [7], BIKE, Rainbow and NTRU [17], and the Fujisaki-Okamoto transform [20,42].

Specifically for McEliece, Strenzke et al. [41] propose exploiting potential power side channels in the polynomial multiplication and polynomial evaluation during key generation, particularly during the generation of the parity check matrix. A side-channel attack on the McEliece secret key presented by Strenzke [40] uses a timing side channel in the decryption routine. A recent attack on the Classic McEliece implementation reveals the secret key using a power side channel in a decryption oracle [19]. Yet another attack shows the use of an electromagnetic side channel for revealing the plaintext of a message [28].

Practical Erasure and Error Rates. In the classical setting, RSA secret keys can be recovered from partial information with regard to different leakage models. Heninger and Shacham [22] recover the secret key even if bits are cleared with an *erasure rate* of 0.73, as long as the remaining known bits are evenly distributed at known positions. A more challenging setting is when there are no known bits, but bits can be flipped with a certain *error rate* $\tau < \frac{1}{2}$. This model was analysed by Henecka et al. [21], allowing for error rates from $\tau \approx 0.08$ for factorization recovery up to $\tau \approx 0.24$ for recovery of RSA CRT keys. These error rate were further improved by Paterson et al. [35], who also considered the asymmetric case where bit flips in either direction (1 to 0, vs. 0 to 1) can have different probabilities.

While RSA is known to be vulnerable to partial information leakage, post-quantum schemes are believed to be more leakage-resistant. This view was challenged by Esser et al. [17], who found recovery attacks on BIKE, Rainbow and NTRU under erasure and bit-flip error models. Allowing for an attack complexity of 80 bits, they achieved secret key recovery with erasure rates up to 0.730 for BIKE, 0.890 for rainbow, and 0.422 for NTRU, as well as with error rates up to 0.200 for BIKE, 0.270 for Rainbow, and 0.019 for NTRU. Earlier, Albrecht et al. [7] reported secret key recovery under a cold-boot attack with error rate 0.017 for Kyber and 0.032 for New Hope.

McEliece Secret Key Recovery From Public Key Generation. In this work, we turn our attention to partial key recovery in code-based cryptography. We investigate the McEliece cryptosystem, focusing on the key generation step, and in particular on the creation of the public key from the private key. We

note that due to the relatively large size of the keys, many implementations do not store them. Instead, implementations tend to store only the random seed originally used when the key was first generated, and reconstruct the keys whenever these are required. Thus, we can expect this step to be repeated multiple times. Moreover, depending on the implementation, the adversary may be able to initiate key reconstruction, for example by querying a server for the public key.

One of the main steps of reconstructing the public key of McEliece is to perform Gaussian elimination on a binary matrix, i.e., over $\mathbb{F}_2$. The algorithm scans the matrix and performs addition of rows in a pattern that depends on the value of the scanned bit. Consequently, finding out when the Gaussian elimination adds rows is equivalent to recovering the bits of the matrix.

1.1 Our Contributions

Secret Key Recovery From Noisy Leak Gauss Elimination. We present a novel attacker model on McEliece public key generation, where the attacker can observe the internal state of Gaussian elimination, with an error rate $\tau < \frac{1}{2}$.For this attacker model, we propose a novel algorithm that can correct bit-flip errors as high as $\tau \approx 0.4$, vastly exceeding error rates reported in related work for attacks on RSA and a variety of post quantum schemes. This demonstrates that McEliece key generation is highly sensitive to leakage in the Gaussian elimination step. We also show that the achievable error correction increases with higher security parameters for McEliece. Intuitively, this is caused by the fact that the redundancy of McEliece keys grows with the security level.

Using Codes to Break Codes. In our attack, we introduce a novel cryptanalytic decoding technique, where we construct a code that contains all possible candidate columns of the McEliece secret key, and use this code to correct errors in the leaked execution matrix of Gaussian successively column by column. Because of the high redundancy in the parity check matrix of the Goppa code, the candidate code is extremely sparse. This results in a very large decoding radius, and thus a large theoretical upper bound of correctable errors ($\tau \approx 0.427$ for high-security McEliece parameters). We analyse the success probability of our algorithm with only few heuristic assumptions, and show that based on our leakage model, the correctable error is quite close to the theoretical upper bound of our approach (0.398 vs. 0.427 for high-security McEliece parameters).

Implementing and Evaluating Sendrier's Support Splitting Algorithm. A McEliece key is defined by a so-called *Goppa polynomial* and a list of *Goppa points*. It is well-known that if the Goppa polynomial and the *set* of Goppa points, but not their order, is known to the attacker, the Support Splitting Algorithm (SSA) [38] can (heuristically) be used to efficiently find the secret key. We can make good use of this in our attack for some parameter sets and implementations. However, to our knowledge, there is no implementation of the

algorithm publicly available. So we contribute a fully working implementation of SSA in SageMath, and verify heuristically that it is efficient for the specific case of McEliece parameter sets.

Practical Evaluation of Leakage Potential. We investigate two real-world implementations of McEliece, the one in the Botan cryptographic library [1], which has been recognized by the German Federal Institute for Information Security [2], and the reference implementation of the Classic McEliece submission to the NIST Post-Quantum Cryptography Standardization Project [8]. The former implementation does not use constant-time coding practices and is therefore vulnerable to side-channel attacks that leak control flow [4,5,44,45] and memory access patterns [29,33]. The latter implementation is designed to be resistant to microarchitectural side-channel attacks. However, the programming pattern it uses for constant-time conditional operations amplifies power and electromagnetic leakage [6,16,30]. Thus, we conclude that both implementations may leak the locations of row additions during Gaussian elimination. For our experiments, we patch both implementations to introduce synthetic leaks based on our analysis of their leakage potential.

Implementation and Experimental Verification. We implement our attack against both libraries, Botan and Classic McEliece. The source code for our attack, the implementation of the Support Splitting Algorithm, as well as the artifacts used for the evaluation, are available at:

`https://github.com/lambdafu/Leaky-McEliece`

In a large-scale experiment, we measure the success rate and runtime of our attack on a variety of proposed security parameters. We find that the success rates are in agreement with our analysis, thus verifying our heuristic assumptions. The algorithm is also very fast. Full secret key recovery for the high-security parameter set takes only 2 minutes and 11 seconds.

1.2 Outline

This work is organized as follows: In Section 2, we introduce notations for matrices, Hamming spaces and codes. We also describe McEliece keys, and briefly recall how to attack McEliece via the Support Splitting Algorithm. In Section 3, we describe how McEliece key generation uses Gaussian elimination, and define our leakage model. After that, we introduce our attack in Section 4. In Section 5, we give a theoretical upper bound on the error rate and analyze the success probability of our attack. In Section 6, we apply our attack on two implementations, Botan and Classic McEliece, patched with a synthetic leak. Our experiments confirm the correctness of our heuristic analysis.

2 Preliminaries

This section describes mathematical notations we use throughout the paper. It also presents some background on the McEliece cryptosystem and on coding theory.

2.1 Notations

Matrices. Let $\mathbf{A} \in \mathbb{Z}^{k \times n}$ be a $(k \times n)$-matrix. We use computer algebra notations to address the entries of $\mathbf{A}$. We write $\mathbf{A}[i,j]$ for the entry of $\mathbf{A}$ in row i and column j. More generally, $\mathbf{A}[i_1{:}i_2, j_1{:}j_2]$ denotes the submatrix of $\mathbf{A}$ formed by the i_1-th to i_2-th rows and j_1-th to j_2-th columns (inclusively). In particular, the i-th row is denoted by $\mathbf{A}[i, 1{:}n]$, and the j-th column by $\mathbf{A}[1{:}k, j]$. If the dimensions are clear from the context, we frequently use the short hand notations $\mathbf{A}[i,:]$ and $\mathbf{A}[:,j]$ for the i-th row and j-th column, respectively. The j-th unit vector is denoted by $\mathbf{e}_j$.

Hamming Space. For $\mathbf{x}, \mathbf{y} \in \mathbb{F}_2^n$ we denote their *Hamming distance* by $\Delta(\mathbf{x}, \mathbf{y})$. The n-dimensional *Hamming ball* around $\mathbf{x} \in \mathbb{F}_2^n$ with radius $r \geq 0$ is defined as $B(\mathbf{x}, r) := \{\mathbf{y} \in \mathbb{F}_2^n \mid \Delta(\mathbf{x}, \mathbf{y}) \leq r\}$. We denote the volume of an n-dimensional Hamming ball with radius $r \in \mathbb{N}$ as

$$V^n(r) = \sum_{i=0}^{r} \binom{n}{i}.$$

For $r \leq \frac{n}{2}$, we have

$$V^n(r) \approx \binom{n}{r} \approx 2^{\mathrm{H}(r/n)n}, \tag{1}$$

where $\mathrm{H} : [0,1] \to [0,1]$ denotes the binary entropy function. Notice that the approximations made in Equation (1) suppress only small polynomial factors in n, see, e.g., [14, Lemma 17.5.1].

We define the inverse binary entropy function H^{-1} as the inverse of H restricted to the interval $[0, \frac{1}{2}]$. That is, for $y \in [0,1]$, we define $x = \mathrm{H}^{-1}(y)$ as the unique real number $x \in [0, \frac{1}{2}]$ satisfying $\mathrm{H}(x) = \mathrm{H}(1-x) = y$.

Codes. A (binary) *code* C of length n is a subset of $\mathbb{F}_2^n$. The *minimum distance* d of a code C is defined as

$$d := \min_{\substack{\mathbf{c}, \mathbf{c}' \in C, \\ \mathbf{c} \neq \mathbf{c}'}} \Delta(\mathbf{c}, \mathbf{c}').$$

The *decoding radius* of a code C with minimum distance d is defined as $\lfloor \frac{d-1}{2} \rfloor$. Equivalently, the decoding radius is defined as the largest radius $r \in \mathbb{N}$, for which no Hamming balls $B(\mathbf{c}, r)$ around codewords $\mathbf{c} \in C$ overlap. Let $\mathbf{x} = \mathbf{c} + \mathbf{e} \in \mathbb{F}_2^n$ be an erroneous codeword, where $\mathbf{c} \in C$. If $\Delta(\mathbf{x}, \mathbf{c}) \leq \lfloor \frac{d-1}{2} \rfloor$, then $\mathbf{c}$ is the unique codeword closest to $\mathbf{x}$.

The *Hamming bound* states that for every code C with distance d it holds that

$$V^n\left(\left\lfloor \frac{d-1}{2} \right\rfloor\right) \le \frac{2^n}{|C|}.$$

For codes with $\lfloor \frac{d-1}{2} \rfloor < \frac{n}{2}$, this yields (together with the approximations from Equation (1)) the asymptotic bound

$$\left\lfloor \frac{d-1}{2} \right\rfloor \le \mathrm{H}^{-1}\left(1 - \frac{\log|C|}{n}\right) \cdot n, \tag{2}$$

where $\log(\cdot)$ denotes the base-2 logarithm.

A code is called linear if it is a linear subspace of $\mathbb{F}_2^n$. Every linear code of dimension k can be defined via a parity check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$, satisfying

$$C = \left\{\mathbf{c} \in \mathbb{F}_2^n \mid \mathbf{H} \cdot \mathbf{c}^T = 0\right\}.$$

2.2 McEliece Keys

McEliece Secret to Public Key Transformation. Let us fix a finite field $\mathbb{F}_{2^m}$. A McEliece secret key is defined via

(1) a list L of $n \le 2^m$ distinct *Goppa points* $L = (\alpha_1, \ldots, \alpha_n) \in \mathbb{F}_{2^m}^n$, and
(2) an irreducible *Goppa polynomial* $g \in \mathbb{F}_{2^m}[x]$ of degree t.

From L and g, we obtain the parity check matrix

$$\mathbf{H}(L, g) := \begin{pmatrix} \frac{1}{g(\alpha_1)} & \frac{1}{g(\alpha_2)} & \cdots & \frac{1}{g(\alpha_n)} \\ \frac{\alpha_1}{g(\alpha_1)} & \frac{\alpha_2}{g(\alpha_2)} & \cdots & \frac{\alpha_n}{g(\alpha_n)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\alpha_1^{t-1}}{g(\alpha_1)} & \frac{\alpha_2^{t-1}}{g(\alpha_2)} & \cdots & \frac{\alpha_n^{t-1}}{g(\alpha_n)} \end{pmatrix} \in \mathbb{F}_{2^m}^{t\times n}. \tag{3}$$

Let us fix an $\mathbb{F}_2$-basis $(1, \gamma, \ldots, \gamma^{m-1})$ for $\mathbb{F}_{2^m}$, i.e., we write every $\mathbb{F}_{2^m}$-element as $a_0 + a_1\gamma + \ldots + a_{m-1}\gamma^{m-1}$ with $a_i \in \mathbb{F}_2$. Let

$$\sigma : \mathbb{F}_{2^m} \to \mathbb{F}_2^m,\ a_0 + a_1\gamma + \ldots, a_{m-1}\gamma^{m-1} \mapsto (a_0, \ldots, a_{m-1})^T \tag{4}$$

denote the canonical vector space embedding into column vectors. We extend σ to vectors and matrices over $\mathbb{F}_{2^m}$ by applying σ coordinate-wise. Applying σ on $\mathbf{H}$ yields a secret *binary* parity check matrix

$$\mathbf{H} \in \mathbb{F}_{2^m}^{t\times n} \overset{\sigma}{\mapsto} \mathbf{H}_{\mathsf{sk}} \in \mathbb{F}_2^{tm\times n},$$

where $\mathbf{H}_{\mathsf{sk}}$ defines our $(n - tm)$-dimensional linear code $C \subseteq \mathbb{F}_2^n$.

The secret parity check matrix $\mathbf{H}_{\mathsf{sk}}$ is now turned into a public parity check matrix by transforming the matrix to systematic form $\mathbf{H}_{\mathsf{pk}} = (\mathbf{I}_{tm}|\mathbf{A})$ via Gaussian elimination. For ease of notation, we assume that $\mathrm{rank}(\mathbf{H}_{\mathsf{sk}}[1{:}tm, 1{:}tm]) = tm$. We detail the Gaussian elimination in the Section 3. The matrices $\mathbf{H}_{\mathsf{sk}}$ and $\mathbf{H}_{\mathsf{pk}}$ form the McEliece secret and public keys, respectively.

Parameter Sets. The suggested Classic McEliece parameter sets from the NIST submission [9] are displayed in Table 1. We also include two test parameter sets for faster calculation in experiments. Botan supports any choice of parameters with $m \leq 15$.

Table 1: Classic McEliece parameter sets and two test parameter sets.

Name	(n,t,m)	Name	(n,t,m)
`mceliece348864`	$(3488, 64, 12)$	`toyeliece51220`	$(512, 20, 9)$
`mceliece460896`	$(4608, 96, 13)$	`toyeliece102450`	$(1024, 50, 10)$
`mceliece6960119`	$(6960, 119, 13)$		
`mceliece6688128`	$(6688, 128, 13)$		
`mceliece8192128`	$(8192, 128, 13)$		

2.3 Support Splitting

Two codes $C, C' \subseteq \mathbb{F}_2^n$ are called *permutation equivalent* if there exists a permutation matrix $\mathbf{P} \in \mathbb{F}_2^{n \times n}$ such that

$$C' = \{\mathbf{c} \cdot \mathbf{P} \mid \mathbf{c} \in C\}.$$

In other words, C and C' are permutation equivalent if C' can be derived by permuting the coordinates of the codewords $\mathbf{c} \in C$. Two parity check matrices $\mathbf{H}, \mathbf{H}'$ define permutation equivalent codes if and only if there exists an invertible matrix $\mathbf{G} \in \mathbb{F}_2^{n \times n}$ and a permutation matrix $\mathbf{P} \in \mathbb{F}_2^{n \times n}$ such that

$$\mathbf{H}' = \mathbf{G} \cdot \mathbf{H} \cdot \mathbf{P}.$$

Given two parity check matrices $\mathbf{H}, \mathbf{H}'$ of two permutation equivalent linear codes C, C', Sendrier's Support Splitting Algorithm (SSA) [38] recovers the corresponding permutation matrix $\mathbf{P}$. While nothing is proven about the complexity of SSA, it is conjectured that for random codes, the algorithm runs in time roughly $\mathcal{O}(n^3)$, i.e., for random codes, SSA appears to be highly efficient.[3]

Support Splitting in McEliece. It is well-known that SSA can be used to attack the McEliece cryptosystem, in a scenario where the attacker obtains the Goppa polynomial $g(x)$ along with the *set* of Goppa points $\{\alpha_1, \dots, \alpha_n\}$ (but without their correct *order* $L = (\alpha_1, \dots, \alpha_n)$). Given $g(x)$ and the set of Goppa

[3] More precisely, it is conjectured that SSA has runtime $\mathcal{O}(n^3 + 2^h n^2 \log n)$, where h is the dimension of the so-called *hull* of C. For random codes, h is with high probability a small constant. Typically, $h \in \{0, 1, 2\}$.

points, the attacker can construct a parity check matrix $\mathbf{H}' \in \mathbb{F}_{2^m}^{t \times n}$, which, up to the order of columns, is identical to the matrix $\mathbf{H}(L, g)$ from Equation (3). In particular,

$$\mathbf{H}' = \mathbf{H}(L, g) \cdot \mathbf{P},$$

for some permutation matrix $\mathbf{P} \in \mathbb{F}_2^{n \times n}$.

Let $\mathbf{G} \in \mathbb{F}_2^{n \times n}$ be the invertible matrix that corresponds to the Gaussian elimination, which transforms $\mathbf{H}_{\mathsf{sk}} = \sigma(\mathbf{H}(L, g))$ into $\mathbf{H}_{\mathsf{pk}}$, i.e.,

$$\mathbf{H}_{\mathsf{pk}} = \mathbf{G} \cdot \mathbf{H}_{\mathsf{sk}} = \mathbf{G} \cdot \sigma(\mathbf{H}(L, g)).$$

Then it holds that

$$\sigma(\mathbf{H}') = \sigma(\mathbf{H}(L, g)) \cdot \mathbf{P} = \mathbf{G}^{-1} \cdot \mathbf{H}_{\mathsf{pk}} \cdot \mathbf{P}.$$

Hence, the *known* matrices $\sigma(\mathbf{H}')$ and $\mathbf{H}_{\mathsf{pk}}$ generate permutation equivalent codes. Thus, by running SSA on $\sigma(\mathbf{H}')$ and $\mathbf{H}_{\mathsf{pk}}$, the attacker can efficiently recover $\mathbf{P}$. Knowledge of $\mathbf{P}$ then reveals the secret key via $\mathbf{H}_{\mathsf{sk}} = \sigma(\mathbf{H}') \cdot \mathbf{P}^{-1}$.

3 Our Attack Model: Monitoring Gaussian Elimination

Gaussian Elimination. Let us look at a simplified high-level version of Gaussian elimination to illustrate our attack model. On input of a matrix

$$\mathbf{H}_{\mathsf{sk}} \in \mathbb{F}_2^{tm \times n}, \quad \text{with } \operatorname{rank}(\mathbf{H}_{\mathsf{sk}}[1{:}tm, 1{:}tm]) = tm,$$

Gaussian elimination transforms $\mathbf{H}_{\mathsf{sk}}$ via elementary row operations into a matrix in systematic form, i.e., into a matrix

$$\mathbf{H}_{\mathsf{pk}} = (\mathbf{I}_{tm} | \mathbf{A}) \in \mathbb{F}_2^{tm \times n}.$$

The core component of Gaussian elimination is a subroutine ELIMINATE-COLUMN that on input of a matrix $\mathbf{H}_j \in \mathbb{F}_2^{tm \times n}$ and an index $j \in \{1, \ldots, tm\}$, transforms the j-th column of $\mathbf{H}_j$ into the j-th unit vector $\mathbf{e}_j^T$. A straight-forward implementation of ELIMINATE-COLUMN is given in Algorithm 1.

To bring $\mathbf{H}_{\mathsf{sk}}$ into systematic form $\mathbf{H}_{\mathsf{pk}}$, Gaussian elimination simply sets

$$\mathbf{H}_1 := \mathbf{H}_{\mathsf{sk}}, \tag{5}$$

$$\mathbf{H}_{j+1} := \text{ELIMINATE-COLUMN}(\mathbf{H}_j, j) \quad \text{for } j = 1, \ldots, tm, \tag{6}$$

$$\mathbf{H}_{\mathsf{pk}} := \mathbf{H}_{tm+1},$$

as depicted in Algorithm 2.[4]

[4] Notice that the call to ELIMINATE-COLUMN in Line 3 of GAUSSIAN-ELIMINATION never returns FAIL since we require $\operatorname{rank}(\mathbf{H}_{\mathsf{sk}}[1 : tm, 1 : tm]) = tm$.

Algorithm 1 ELIMINATE-COLUMN

Input: $\mathbf{H}_j \in \mathbb{F}_2^{tm \times n}$, $j \in \{1, \ldots, tm\}$
Output: $\mathbf{H}_{j+1} \in \mathbb{F}_2^{tm \times n}$ with $\mathbf{H}_{j+1}[:, j] = \mathbf{e}_j^T$, and $\mathbf{H}_{j+1} = \mathbf{G}_j \cdot \mathbf{H}_j$ for some invertible $\mathbf{G}_j \in \mathbb{F}_2^{tm \times tm}$, or FAIL.

1: $\mathbf{H}_{j+1} := \mathbf{H}_j$
2: **if** $\mathbf{H}_{j+1}[j, j] \neq 1$ **then** ▷ ensure $\mathbf{H}_{j+1}[j, j] = 1$
3: Find minimal $k \in \{j + 1, \ldots, tm\}$ with $\mathbf{H}_{j+1}[k][j] = 1$.
4: **if** no such k exists **then**
5: **return** FAIL
6: **end if**
7: $\mathbf{H}_{j+1}[j, :] := \mathbf{H}_{j+1}[j, :] + \mathbf{H}_{j+1}[k, :]$ ▷ add k-th to j-th row
8: **end if**
9: **for** $i = 1, \ldots, tm$, $j \neq i$ **do**
10: **if** $\mathbf{H}_{j+1}[i, j] = 1$ **then** ▷ ensure $\mathbf{H}_{j+1}[i, j] = 0$ for $i \neq j$
11: $\mathbf{H}_{j+1}[i, :] := \mathbf{H}_{j+1}[i, :] + \mathbf{H}_{j+1}[j, :]$ ▷ add j-th to i-th row
12: **end if**
13: **end for**
14: **return** $\mathbf{H}_{j+1}$

Algorithm 2 GAUSSIAN-ELIMINATION

Input: $\mathbf{H}_{\mathsf{sk}} \in \mathbb{F}_2^{tm \times n}$ with $\mathrm{rank}(\mathbf{H}_{\mathsf{sk}}[1 : tm, 1 : tm]) = tm$
Output: systematic form $\mathbf{H}_{\mathsf{pk}} = (\mathbf{I}_{tm}|\mathbf{A}) \in \mathbb{F}_2^{tm \times n}$ of $\mathbf{H}_{\mathsf{sk}}$,

1: $\mathbf{H}_1 := \mathbf{H}_{\mathsf{sk}}$
2: **for** $j = 1, \ldots, tm$ **do**
3: $\mathbf{H}_{j+1} :=$ ELIMINATE-COLUMN$(\mathbf{H}_j, j)$
4: **end for**
5: **return** $\mathbf{H}_{\mathsf{pk}} := \mathbf{H}_{tm+1}$

Our Attack Vector. ELIMINATE-COLUMN's row addition in Line 11 of Algorithm 1 is triggered by the if-statement in Line 10. This if-statement is our attack vector. We assume that we have access to a noisy side channel, which allows us to monitor whether Line 11 gets executed. By that, the side channel reveals noisy variants of the matrix entries $\mathbf{H}_j[i, j]$ where $i \neq j$, i.e., the non-diagonal entries of the j-th column of each $\mathbf{H}_j$.

The j-th columns of all $\mathbf{H}_j$'s form our so-called *execution matrix.* Since we assume that our monitoring process is noisy, we define our *leak matrix* as an erroneous version of the execution matrix (see Definition 1 below). Notice that since our leak does not reveal the diagonal entries $\mathbf{H}_j[j, j]$, the diagonal entries of our leak matrix $\mathbf{L}$ are drawn uniformly at random.

Definition 1 (Execution Matrix and Leak Matrix). *We define the* execution matrix *as*

$$\mathbf{E} := \Big(\mathbf{H}_1[:, 1] \mid \mathbf{H}_2[:, 2] \mid \ldots \mid \mathbf{H}_{tm}[:, tm]\Big) \in \mathbb{F}_2^{tm \times tm}.$$

The leak matrix *is an error-prone version of the execution matrix. More precisely, for all* $1 \leq i, j \leq tm$ *we have*

$$\mathbf{L}[i,j] := \mathbf{E}[i,j] + \mathbf{e}[i,j] = \mathbf{H}_j[i,j] + \mathbf{e}[i,j], \tag{7}$$

where $\mathbf{e}[i,j] \in \mathbb{F}_2$ *and for some error probability* $\tau \leq \frac{1}{2}$*:*

$$\Pr[\mathbf{e}[i,j] = 1] = \begin{cases} \tau & \text{for } i \neq j \\ \frac{1}{2} & \text{for } i = j \end{cases}.$$

In other words, we have a Bernoulli-distributed error $\mathbf{e}[i,j] \sim \mathcal{B}(\tau)$ *for all but the diagonal entries. For simplicity of exposition, we call the leak matrix a* $\mathcal{B}(\tau)$-disturbed *version of the execution matrix (thereby ignoring the diagonal issue).*

Note that for $\tau = \frac{1}{2}$, the matrix $\mathbf{L}$ is uniformly random and does not provide any information. As our leak matrix is crucial for understanding our attack, let us make some simple, but important structural observations. To this end, let us introduce one more definition.

Definition 2 (Transformation Matrix). *For all* $1 \leq j \leq tm$*, we define the* j-th transformation matrix $\mathbf{G}_j \in \mathbb{F}_2^{tm \times tm}$ *as the unique, invertible matrix, satisfying*

$$\mathbf{H}_{j+1} = \mathbf{G}_j \cdot \mathbf{H}_j,$$

corresponding to the elementary row operations of Eliminate-column$(\mathbf{H}_j, j)$.

Lemma 1. *For all* $1 \leq j \leq tm$*, the* j*-th column* $\mathbf{L}[:,j]$ *contains a* $\mathcal{B}(\tau)$*-disturbed version of*

$$\mathbf{E}[:,j] = \mathbf{H}_j[:,j] = \mathbf{G}_{j-1} \cdot \ldots \cdot \mathbf{G}_1 \cdot \mathbf{H}_{\mathsf{sk}}[:,j].$$

Proof. By Equations (5) and (6), we obtain

$$\mathbf{H}_j = \mathbf{G}_{j-1} \cdot \ldots \cdot \mathbf{G}_1 \cdot \mathbf{H}_{\mathsf{sk}}, \tag{8}$$

for every $1 \leq j \leq tm$. Plugging in Equation (8) into Equation (7), the statement follows. □

Remark 1. Given the j-th column $\mathbf{E}[:,j] = \mathbf{H}_j[:,j]$ of the execution matrix $\mathbf{E}$, we can efficiently compute the j-th transformation matrix $\mathbf{G}_j$ as follows:

1. We pick an arbitrary matrix $\mathbf{M} \in \mathbb{F}_2^{tm \times n}$, whose j-th column is identical to $\mathbf{H}_j[:,j]$, e.g., $\mathbf{M} = (\mathbf{0}^{j-1} | \mathbf{H}_j[:,j] | \mathbf{0}^{tm-j})$.
2. We run Eliminate-column$(\mathbf{M}, j)$, and monitor all its elementary row operations.
3. We apply the exact same row operations to $\mathbf{I}_{tm}$.

The resulting matrix is identical to $\mathbf{G}_j$.

Of course, our simplified Gaussian-Elimination is not protected at all against leakage of Eliminate-Column's operation in Line 11. However, we show in Section 6 that more involved state-of-the-art implementations also do not provide sufficient leakage resistance.

4 Our Attack: Decoding the Leak Matrix

On a high level, our algorithm for recovering the secret key $\mathbf{H}_{\mathsf{sk}}$ from the leak matrix $\mathbf{L}$ works as follows: Given $\mathbf{L}$, we successively recover all columns of the execution matrix $\mathbf{E} \in \mathbb{F}_2^{tm \times tm}$. As shown in Remark 1, the columns $\mathbf{E}[:,1], \ldots, \mathbf{E}[:,tm]$ then reveal the invertible transformation matrices $\mathbf{G}_1, \ldots, \mathbf{G}_{tm}$, satisfying

$$\mathbf{H}_{\mathsf{pk}} = \mathbf{G}_{tm} \cdot \ldots \cdot \mathbf{G}_2 \cdot \mathbf{G}_1 \cdot \mathbf{H}_{\mathsf{sk}}.$$

These in turn allow us to easily recover the secret key via

$$\mathbf{H}_{\mathsf{sk}} = \mathbf{G}_1^{-1} \cdot \mathbf{G}_2^{-1} \cdot \ldots \cdot \mathbf{G}_{tm}^{-1} \cdot \mathbf{H}_{\mathsf{pk}}.$$

In more detail, we proceed as follows.

Recovery of E[:, 1]. We begin with recovery of the first column $\mathbf{E}[:,1]$. To this end, we compute a (non-linear) code[5]

$$C_1 := \left\{ \sigma\left(\frac{1}{b}(1, a, a^2, \ldots, a^{t-1})\right) \mid a, b \in \mathbb{F}_{2^m}, b \neq 0 \right\} \subseteq \mathbb{F}_2^{tm}. \tag{9}$$

By Equation (3), our code C_1 contains all potential candidates for the columns of the secret key $\mathbf{H}_{\mathsf{sk}}$. In particular, C_1 contains all candidates for the first column $\mathbf{E}[:,1]$ of our execution matrix $\mathbf{E}$, since by definition $\mathbf{E}[:,1] = \mathbf{H}_{\mathsf{sk}}[:,1]$ (see Equation (5)).

Interestingly, C_1 is a very small subset of $\mathbb{F}_2^{tm}$: By Table 1, we obtain for all Classic McEliece parameter sets

$$|C_1| < 2^{2m} \leq 2^{26}, \quad \text{but} \quad |\mathbb{F}_2^{tm}| = 2^{tm} \geq 2^{768}.$$

Hence, we can easily construct and store C_1 in practice. Moreover, when making the mild assumption that the codewords $\mathbf{c} \in C_1$ are distributed somewhat uniformly in $\mathbb{F}_2^{tm}$, we can expect C_1 to have a rather large decoding radius.

By Lemma 1, the first leak matrix column $\mathbf{L}[:,1]$ is a $\mathcal{B}(\tau)$-disturbed version of $\mathbf{E}[:,1]$. Therefore, as long as our error rate τ is not too large, $\mathbf{E}[:,1] \in C_1$ is likely the codeword closest to $\mathbf{L}[:,1]$. Thus, to recover $\mathbf{E}[:,1]$ with high probability, we simply decode $\mathbf{L}[:,1]$ to the closest codeword $\mathbf{c} \in C_1$.

In Section 5, we thoroughly analyze the success probability of this approach. We experimentally verify in Section 6 that it performs well in practice.

Code Update and Recovery of E[:, 2]. After recovering $\mathbf{E}[:,1]$, we obtain the second execution matrix column $\mathbf{E}[:,2]$ as follows:

By Lemma 1, the second leak matrix column $\mathbf{L}[:,2]$ contains a $\mathcal{B}(\tau)$-disturbed version of

$$\mathbf{E}[:,2] = \mathbf{G}_1 \cdot \mathbf{H}_{\mathsf{sk}}[:,2]. \tag{10}$$

[5] Recall that $\sigma : \mathbb{F}_{2^m} \to \mathbb{F}_2^m$ is the canonical vector space embedding from Equation (4), which we apply here coordinate-wise to the vector $\frac{1}{b}(1, a, a^2, \ldots, a^{t-1})$.

We recover $\mathbf{G}_1$ from $\mathbf{E}[:,1]$ via ELIMINATE-COLUMN, as shown in Remark 1, and *update* our code C_1 by multiplying it with $\mathbf{G}_1$. That is, we construct the code

$$C_2 := \mathbf{G_1} \cdot C_1 = \{\mathbf{G}_1 \cdot \mathbf{c} \mid \mathbf{c} \in C_1\}\,. \tag{11}$$

Since C_1 contains all potential candidates for the columns of the secret key $\mathbf{H}_{\mathsf{sk}}$, the updated code C_2 then contains (by Equation (10)) all candidates for $\mathbf{E}[:,2]$. Analogously to the recovery of the first column $\mathbf{E}[:,1]$, we recover the second column $\mathbf{E}[:,2] \in C_2$ with high probability by simply decoding $\mathbf{L}[:,2]$ to the closest codeword $\mathbf{c} \in C_2$.

Inductively Recovering $\mathbf{E}[:,j]$ for $j > 2$. We proceed inductively with reconstruction of the remaining $\mathbf{E}[:,j]$'s:

Suppose we have already recovered $\mathbf{E}[:,1], \ldots \mathbf{E}[:,j-1]$ and constructed the corresponding transformation matrices $\mathbf{G}_1, \ldots, \mathbf{G}_{j-1}$ along with the codes $C_1, \ldots, C_{j-1}$ where

$$C_i := \mathbf{G}_{i-1} \cdot C_{i-1} = \{\mathbf{G}_{i-1} \cdot \mathbf{c} \mid \mathbf{c} \in C_{i-1}\}\,, \quad i \geq 2.$$

Using Remark 1, we recover the transformation matrix $\mathbf{G}_{j-1}$ from $\mathbf{E}[:,j-1]$. We multiply C_{j-1} with $\mathbf{G}_{j-1}$ to obtain the code

$$C_j = \mathbf{G}_{j-1} \cdot C_{j-1} = \mathbf{G}_{j-1} \cdot \ldots \cdot \mathbf{G}_2 \cdot \mathbf{G}_1 \cdot C_1.$$

The resulting code C_j then contains all candidates for the j-th column

$$\mathbf{E}[:,j] = \mathbf{G}_{j-1} \cdot \ldots \cdot \mathbf{G}_2 \cdot \mathbf{G}_1 \cdot \mathbf{H}_{\mathsf{sk}}[:,j].$$

Since $\mathbf{L}[:,j]$ is a $\mathcal{B}(\tau)$-disturbed version of $\mathbf{E}[:,j]$, we then recover $\mathbf{E}[:,j] \in C_j$ with high probability by decoding $\mathbf{L}[:,j]$ to the closest codeword $\mathbf{c} \in C_j$.

Codebook Reduction. Recall that the first execution matrix column $\mathbf{E}[:,1]$ is identical to the first secret key column $\mathbf{H}_{\mathsf{sk}}[:,1]$. Thus, after recovering $\mathbf{E}[:,1]$ from our leak matrix $\mathbf{L}$, we can easily read off the first Goppa point α_1 from $\mathbf{E}[:,1]$ (see Equation (3)).

Since the n Goppa points $\alpha_1, \ldots, \alpha_n$ are distinct, this allows us to slightly reduce the size of the code C_2, and thereby improve the runtime of our attack: Instead of using the code $C_2 := \mathbf{G}_1 \cdot C_1$ from Equation (11), we first remove all $2^m - 1$ codewords

$$\sigma\left(\frac{1}{\beta}(1, a, a^2, \ldots, a^{t-1})\right) \quad \text{with} \quad a = \alpha_1$$

from C_1, and then multiply the resulting code by $\mathbf{G}_1$. Clearly, our smaller code of size $|C_1| - (2^m - 1)$ still contains all candidates for the second column $\mathbf{H}[:,2]$.

By doing some additional bookkeeping, we can also recover the Goppa points α_i, for $i = 2, \ldots, tm$, and thereby further decrease the size of our codes per each recovered column. To this end, we simply define a family of so-called *codebooks*

$$\mathsf{CB}_1 := \left\{ \left(\sigma\left(\frac{1}{b}(1, a, a^2, \ldots, a^{t-1})\right), a, b \right) \mid a, b \in \mathbb{F}_{2^m}, b \neq 0 \right\},$$
$$\mathsf{CB}_j := \{(\mathbf{G}_{j-1} \cdot \mathbf{c}, a, b) \mid (\mathbf{c}, a, b) \in \mathsf{CB}_{j-1}, a \neq \alpha_{j-1}\} \quad j > 1,$$

in which we

1. keep track of the a's and b's that define our codewords $\mathbf{c} \in C_j$, and
2. remove all codewords that are defined via already recovered Goppa points.

Notice that $\mathsf{CB}_j \subseteq C_j \times \mathbb{F}_{2^m} \times \mathbb{F}_{2^m}$, i.e., the first component of each codebook CB_j forms a subcode of C_j. Furthermore, the first component of each CB_j still contains all candidates for the column $\mathbf{E}[:, j]$.

Algorithm 3 MaxLikelihood-Decode

Input: j-th column $\mathbf{L}[:, j] \in \mathbb{F}_2^{tm}$ of leak matrix,
codebook $\mathsf{CB} = \{(\mathbf{c}_i, a_i, b_i)\}_{i=1,\ldots,|\mathsf{CB}|} \subset \mathbb{F}_2^{tm} \times \mathbb{F}_{2^m} \times \mathbb{F}_{2^m}$
Output: $(\mathbf{c}, a, b) \in \mathsf{CB}$ with codeword $\mathbf{c}$ closest to $\mathbf{L}[:, j]$

1: $\mathbf{c}_{\min} := (\mathbf{c}_1, a_1, b_1)$
2: $d_{\min} := \Delta(\mathbf{c}_1, \mathbf{L}[:, j])$
3: **for** $i = 2, \ldots, |\mathsf{CB}|$ **do**
4:     **if** $(\Delta(\mathbf{c}_i, \mathbf{L}[:, j]) < d_{\min})$ **then**
5:         $\mathbf{c}_{\min} := (\mathbf{c}_i, a_i, b_i)$
6:         $d_{\min} := \Delta(\mathbf{c}_i, \mathbf{L}[:, j])$
7:     **end if**
8: **end for**
9: **return** $\mathbf{c}_{\min}$

To efficiently decode the columns $\mathbf{L}[:, j]$ via our codebooks to $\mathbf{E}[:, j]$, we use our algorithm MaxLikelihood-Decode as shown in Algorithm 3.

Our codebook-based approach slightly reduces the size of our codes by $2^m - 1$ per recovered column $\mathbf{E}[:, j]$. A more significant size reduction, however, can be achieved after recovering the first $t+1$ columns: When recovering a column $\mathbf{E}[:, j]$ via MaxLikelihood-Decode, the algorithm's output $(\mathbf{c}, a, b)$ reveals not only the j-th Goppa point $a = \alpha_j$, but also $b = g(\alpha_j)$, i.e., the evaluation of the Goppa polynomial $g(x)$ at α_j. After recovering the first $t+1$ columns $\mathbf{E}[:, 1], \ldots, \mathbf{E}[:, t+1]$, we thus obtain the pairs

$$(\alpha_1, g(\alpha_1)), \ldots, (\alpha_{t+1}, g(\alpha_{t+1})).$$

These $t + 1$ pairs uniquely determine the degree-t Goppa polynomial $g(x)$.

Given the tuples $(\alpha_i, g(\alpha_i))$, $i = 1, \ldots, t + 1$, we efficiently compute $g(x)$ via Lagrange interpolation. Knowledge of $g(x)$ then allows us to filter out all tuples

$$(\mathbf{c}, a, b) \quad \text{with} \quad b \neq g(a)$$

from our codebooks. In other words, we define

$$\mathsf{CB}_j := \{(\mathbf{G}_{j-1} \cdot \mathbf{c}, a, b) \mid (\mathbf{c}, a, b) \in \mathsf{CB}_{j-1}, a \neq \alpha_{j-1}, b = g(a)\}, \quad j \geq t+2. \tag{12}$$

The resulting codebooks are of size $|\mathsf{CB}_j| < 2^m$, i.e., less than the square root of our initial codebook CB_1 with $|\mathsf{CB}_1| = 2^m(2^m - 1)$.

Special Case of Known Goppa Points. As shown in Table 1, the parameter set `mceliece8192128` has $(n, m) = (8192, 13)$. Therefore $2^m = n$, which implies that $\{\alpha_1, \dots, \alpha_n\} = \mathbb{F}_{2^m}$, i.e., the *set* of Goppa points is the whole field. If we succeed to compute $g(x)$, we may apply the Support Splitting algorithm to efficiently recover the secret key. This in turn implies that in the case of $n = 2^m$, we only have to correctly decode $t + 1$ columns via MaxLikelihood-Decode before successfully recovering $g(x)$ via Lagrange interpolation and L via Support Splitting — and thus the whole secret key. In particular, in the case of $n = 2^m$, our side channel has to leak only the first $t+1$ iterations of Eliminate-Column's for-loop.

More generally, we can always apply Support Splitting when we know the *set* of Goppa points. This occurs, for instance, when a McEliece implementation generates the Goppa points deterministically. As we show in Section 6.2, this is the case for the cryptographic library Botan.

Putting Everything Together. Our complete attack Secret-Key-Recovery that recovers the secret key $\mathbf{H}_{\mathsf{sk}}$ from our leak matrix $\mathbf{L}$ is given in Algorithm 4.

5 Analysis

In this section, we analyze for which size of the error τ our algorithm Secret-Key-Recovery succeeds to recover the secret key $\mathbf{H}_{\mathsf{sk}}$ with good success probability. We start by giving a simple asymptotic upper bound on the error rate τ that Secret-Key-Recovery can tolerate. Interestingly, this bound depends only on the McEliece parameter t, and, surprisingly, increases with t. In other words, the higher the McEliece security level, the more errors we can allow in our leak matrix.

After explaining why that is the case, we proceed with a thorough analysis of the success probability of Secret-Key-Recovery. We end this section by showing that our simple asymptotic upper bound quite accurately matches the actual error rates that we obtain from our more thorough probability analysis.

5.1 A Simple Asymptotic Upper Bound on τ

In a nutshell, our algorithm Secret-Key-Recovery successively recovers each column $\mathbf{E}[:, j]$ of our execution matrix by decoding the corresponding leak matrix column $\mathbf{L}[:, j]$ to the closest candidate in some codebook CB_j.

Algorithm 4 SECRET-KEY-RECOVERY

Input: public key $\mathbf{H}_{\mathsf{pk}} \in \mathbb{F}_2^{tm \times n}$,
leak matrix $\mathbf{L} \in \mathbb{F}_2^{tm \times tm}$ (see Definition 1)
Output: secret key $\mathbf{H}_{\mathsf{sk}} \in \mathbb{F}_2^{tm \times n}$

1: $\mathsf{CB} := \left\{ \left(\sigma \left(\frac{1}{b}(1, a, a^2, \dots, a^{t-1}) \right), a, b \right) \mid a, b \in \mathbb{F}_{2^m}, b \neq 0 \right\} \subseteq \mathbb{F}_2^{tm} \times \mathbb{F}_{2^m} \times \mathbb{F}_{2^m}$.
2: **for** $j = 1, \dots, tm$ **do**
3: $(\mathbf{H}_j[:, j], \alpha_j, \beta_j) :=$ MAXLIKELIHOOD-DECODE$(\mathbf{L}[:, j], \mathsf{CB})$
4: Recover $\mathbf{G}_j$ from $\mathbf{H}_j[:, j]$. ▷ See Remark 1.
5: $\mathsf{CB} := \{(\mathbf{G}_j \cdot \mathbf{c}, a, b) \mid (\mathbf{c}, a, b) \in \mathsf{CB}, a \neq \alpha_j\}$
6: **if** $j = t + 1$ **then**
7: Interpolate $g(x) \in \mathbb{F}_{2^m}[x]$ from $(\alpha_1, \beta_1) \dots, (\alpha_{t+1}, \beta_{t+1})$, where $\beta_i = g(\alpha_i)$.
8: **if** set of Goppa points $\{\alpha_1, \dots, \alpha_n\}$ known **then**
9: Recover $\mathbf{H}_{\mathsf{sk}}$ via Support Splitting.
10: **return** $\mathbf{H}_{\mathsf{sk}}$.
11: **else**
12: $\mathsf{CB} := \{(\mathbf{c}, a, b) \in \mathsf{CB} \mid b = g(a)\}$
13: **end if**
14: **end if**
15: **end for**
16: **return** $\mathbf{H}_{\mathsf{sk}} := \mathbf{G}_1^{-1} \cdot \mathbf{G}_2^{-1} \cdot \ldots \cdot \mathbf{G}_{tm}^{-1} \cdot \mathbf{H}_{\mathsf{pk}}$

Let $d_j := \Delta(\mathbf{L}[:, j], \mathbf{E}[:, j])$, and let r_j be the decoding radius of the code defined by codebook CB_j. We decode correctly with probability 1 if and only if

$$d_j \leq r_j, \quad \text{for every } j = 1, \dots, tm.$$

Together with Equation (2), this yields the following asymptotic necessary condition for the correctness of SECRET-KEY-REOCVERY:

$$d_j \leq \mathrm{H}^{-1} \left(1 - \frac{\log |\mathsf{CB}_j|}{tm} \right) \cdot tm, \quad \text{for every } j = 1, \dots, tm.$$

Using $|\mathsf{CB}_1| \approx 2^{2m}$, we can also use the simpler necessary condition

$$d_1 \leq \mathrm{H}^{-1} \left(1 - \frac{2}{t} \right) \cdot tm. \tag{13}$$

By Lemma 1, $\mathbf{L}[:, 1]$ is a $\mathcal{B}(\tau)$-disturbed version of $\mathbf{E}[:, 1]$. Therefore, d_1 is $\mathcal{B}(\tau)$-distributed. For simplicity, let us assume that d_1 achieves its expected value $\mathbb{E}[d_1] = \tau tm$. Then Equation (13) translates to

$$\tau < \mathrm{H}^{-1} \left(1 - \frac{2}{t} \right). \tag{14}$$

We visualize the upper bound from Equation (14) in Figure 1 as a function of t. Figure 1 shows that for typical McEliece with $t \in [64, 128]$, as in Table 1, we obtain an upper bound for τ between roughly 0.39 and 0.42.

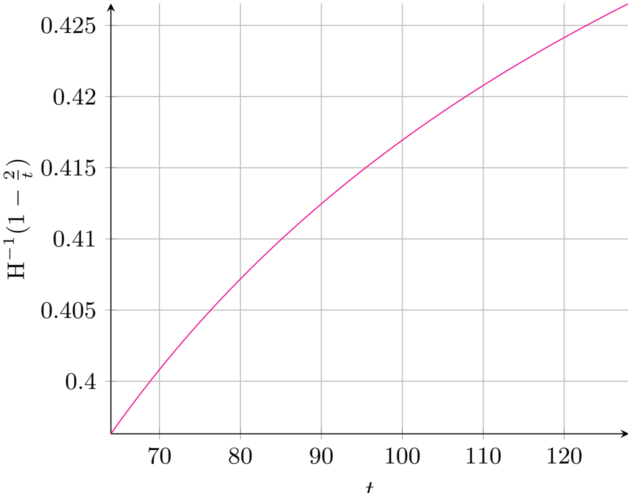


Fig. 1: Upper bound from Equation (14) for $64 \leq t \leq 128$.

Growth of τ. From Figure 1, we see that, quite remarkably, our upper bound for the error rate τ actually increases with t. Hence, the bound suggests that for larger security levels of McEliece we can tolerate larger errors τ in our leak matrix. Let us briefly explain this phenomenon.

Each secret key column $\mathbf{H}_{\mathsf{sk}}[:,j]$ is uniquely defined by some Goppa point $\alpha_j \in \mathbb{F}_{2^m}$ and the corresponding Goppa polynomial evaluation $g(\alpha_j) \in \mathbb{F}_{2^m}$ (see Equation (3)). Since any field element from $\mathbb{F}_{2^m}$ can be represented by m bits, this shows that each secret key column contains only $2m$ bits of information. In other words, we have redundancy of

$$\frac{(t-2)m}{tm} = 1 - \frac{2}{t}$$

per bit of $\mathbf{H}_{\mathsf{sk}}[:,j] \in \mathbb{F}_2^{tm}$.

Recall that the columns $\mathbf{E}[:,j]$ of our execution matrix are of the form

$$\mathbf{E}[:,j] = \mathbf{G}_{j-1} \cdot \ldots \cdot \mathbf{G}_2 \cdot \mathbf{G}_1 \cdot \mathbf{H}_{\mathsf{sk}}[:,j].$$

Since the transformation matrices $\mathbf{G}_i$ are invertible, it follows that each $\mathbf{E}[:,j]$ contains exactly as much information as the corresponding secret key column $\mathbf{H}_{\mathsf{sk}}[:,j]$. Hence, also in every execution matrix column $\mathbf{E}[:,j]$, a $(1-\frac{2}{t})$-fraction of the bits is redundant. Therefore, the larger t gets, the more redundant gets our execution matrix — making it easier to decode the leak matrix $\mathbf{L}$.

Remark 2. Our discussion on the growth of τ yields a nice interpretation of our upper bound from Equation (14): Since $\mathbf{L} \in \mathbb{F}_2^{tm \times tm}$ is a $\mathcal{B}(\tau)$-disturbed version of $\mathbf{E}$, we obtain uncertainty $\mathrm{H}(\tau)$ per bit of $\mathbf{L}$. In contrast, we have redundancy $\left(1-\frac{2}{t}\right)$ per bit of $\mathbf{E}$. Hence, our bound from Equation (14) simply requires that the uncertainty of $\mathbf{L}$ does not exceed the redundancy of $\mathbf{E}$.

5.2 Analysis of Success Probability

What mainly prevents our attack from reaching the upper bound from Equation (14) in practice is that the error does not always match its expected value. This variance has to be taken into account. Let us now precisely determine the success probability of our algorithm Secret-Key-Recovery.

Decoding a Single Column. We start by analyzing the success probability of correctly decoding a single leak column $\mathbf{L}[:,j]$ to the corresponding execution matrix column $\mathbf{E}[:,j]$.

Let $d_j := \Delta(\mathbf{L}[:,j], \mathbf{E}[:,j])$. We decode correctly if $\mathbf{L}[:,j]$ has distance at least $d_j + 1$ to any other codeword $\mathbf{c} \in \mathsf{CB}_j \setminus \{\mathbf{E}[:,j]\}$.[6] Conversely, we may decode incorrectly only if any $\mathbf{L}[:,j]$ hits a point inside a Hamming ball $B(\mathbf{c}, d_j)$. To ease the analysis of the probability of this event, let us introduce the following heuristic assumption.

Assumption 1 *We assume that the points $\mathbf{c} \in \mathbb{F}_2^{tm}$ in our codebook $\mathsf{CB}_j \subseteq \mathbb{F}_2^{tm} \times \mathbb{F}_{2^m} \times \mathbb{F}_{2^m}$ are independent and uniformly at random distributed in $\mathbb{F}_2^{tm}$.*

Lemma 2. *Let $p(d_j)$ denote the probability*

$p(d_j) := \Pr[\mathbf{L}[:,j]$ *decodes correctly to* $\mathbf{E}[:,j] | \mathbf{L}[:,1], \ldots \mathbf{L}[:,j-1]$ *decoded correctly*$]$,

where $d_j := \Delta(\mathbf{L}[:,j], \mathbf{E}[:,j])$. Under Assumption 1 we obtain

$$p(d_j) = \left(1 - \frac{V^{tm}(d_j)}{2^{tm}}\right)^{|\mathsf{CB}_j \setminus \{\mathbf{E}[:,j]\}|}. \tag{15}$$

Proof. Let $\mathbf{c} \in \mathsf{CB}_j \setminus \{\mathbf{E}[:,j]\}$ be arbitrary. Let $E_\mathbf{c}$ denote the event that $\mathbf{L}[:,j]$ decodes (incorrectly) to $\mathbf{c}$. Event $E_\mathbf{c}$ implies that $\mathbf{L}[:,j]$ hits one of the $V^{tm}(d_j)$ points inside the Hamming ball of radius d_j around $\mathbf{c}$.

By Assumption 1, $\mathbf{E}[:,j] \in \mathsf{CB}_j$ is uniformly distributed and therefore $\mathbf{L}[:,j]$ as well (since, by Lemma 1, $\mathbf{L}[:,j]$ is a $\mathcal{B}(\tau)$-disturbed version of $\mathbf{E}[:,j]$). Thus, $E_\mathbf{c}$ happens with probability $V^{tm}(d_j) \cdot 2^{-tm}$. Conversely, $\mathbf{L}[:,j]$ does not decode to $\mathbf{c}$ with probability

$$1 - \frac{V^{tm}(d_j)}{2^{tm}}.$$

The column $\mathbf{L}[:,j]$ decodes correctly if and only if it does not decode to any incorrect $\mathbf{c} \in \mathsf{CB}_j \setminus \{\mathbf{H}[:,j]\}$. By Assumption 1, the events $E_\mathbf{c}$ are independent for all $\mathbf{c}$. Thus,

$$p(d_j) = \prod_{\mathbf{c} \in \mathsf{CB}_j \setminus \mathbf{E}[:,j]} 1 - \frac{V^{tm}(d_j)}{2^{tm}} = \left(1 - \frac{V^{tm}(d_j)}{2^{tm}}\right)^{|\mathsf{CB}_j \setminus \{\mathbf{E}[:,j]\}|}. \qquad \square$$

We verify the validity of Assumption 1 experimentally in Section 6 by showing that the actual success probability closely matches Lemma 2.

[6] For simplicity, we take some notational liberty throughout this section by identifying codebook elements $(\mathbf{c}, \alpha, \beta) \in \mathsf{CB}_j$ with their first component $\mathbf{c}$.

Decoding All Columns. Secret-Key-Recovery succeeds to output the secret key $\mathbf{H}_{\mathsf{sk}}$ if it correctly decodes $\mathbf{L}[:,1],\ldots,\mathbf{L}[:,tm]$. For the j-th column, this happens with probability $p(d_j)$. Hence, Secret-Key-Recovery's success probability is given by

$$\Pr[\textsc{Success}] := \prod_{j=1}^{tm} \sum_{d_j=0}^{tm} p(d_j) \cdot \Pr[\Delta(\mathbf{L}[:,j],\mathbf{E}[:,j]) = d_j]. \tag{16}$$

It remains to determine the distribution of the random variable $\Delta(\mathbf{L}[:,j],\mathbf{E}[:,j])$.

Since, $\mathbf{L}[:,j]$ is a $\mathcal{B}(\tau)$-disturbed version of $\mathbf{E}[:,j]$, d_j is $\mathcal{B}(\tau)$-distributed. As a consequence, we have

$$\Pr[\Delta(\mathbf{L}[:,j],\mathbf{E}[:,j]) = d] = \binom{tm}{d} \cdot \tau^d \cdot (1-\tau)^{tm-d} \tag{17}$$

for all $1 \leq j \leq tm$.

Using Lemma 2 together with Equations (16) and (17), we obtain Secret-Key-Recovery's success probability as

$$\Pr[\textsc{Success}] = \prod_{j=1}^{tm} \sum_{d_j=0}^{tm} \left(1 - \frac{V^{tm}(d_j)}{2^{tm}}\right)^{|\mathsf{CB}_j \setminus \{\mathbf{E}[:,j]\}|} \cdot \binom{tm}{d_j} \cdot \tau^{d_j} \cdot (1-\tau)^{tm-d_j}.$$

Recall that for each codebook CB_j we have $|\mathsf{CB}_j| < 2^{2m}$. Furthermore, for $j \geq t+2$, we have $|\mathsf{CB}_j| < 2^m$, since after $t+1$ recovered columns we interpolate the Goppa polynomial and then reduce the codebook size (see Equation (12)). Therefore, we obtain

$$\begin{aligned}\Pr[\textsc{Success}] > &\left(\sum_{d_j=0}^{tm} \left(1 - \frac{V^{tm}(d_j)}{2^{tm}}\right)^{2^{2m}} \cdot \binom{tm}{d_j} \cdot \tau^{d_j} \cdot (1-\tau)^{tm-d_j}\right)^{t+1} \cdot \\ &\left(\sum_{d_j=0}^{tm} \left(1 - \frac{V^{tm}(d_j)}{2^{tm}}\right)^{2^{m}} \cdot \binom{tm}{d_j} \cdot \tau^{d_j} \cdot (1-\tau)^{tm-d_j}\right)^{tm-(t+1)}.\end{aligned} \tag{18}$$

For the special case of known Goppa points, we stop decoding after $t+1$ iterations in Secret-Key-Recovery. Therefore, we require only the first factor from Equation (18), i.e.,

$$\Pr[\textsc{Success}] \geq \left(\sum_{d_j=0}^{tm} \left(1 - \frac{V^{tm}(d_j)}{2^{tm}}\right)^{2^{2m}} \cdot \binom{tm}{d_j} \cdot \tau^{d_j} \cdot (1-\tau)^{tm-d_j}\right)^{t+1}.$$

Table 2 shows the largest error rate τ for which Equation (18) is at least $\frac{1}{2}$. Interestingly, the error rates match the information theoretical upper bound from Equation (14) quite accurately. In fact, the larger the McEliece parameters get, the more accurate Equation (14) becomes.

Table 2: Largest error rate τ for which Equation (18) gives success probability at least $\frac{1}{2}$ compared to the upper bound from Equation (14).

Name	Equation (18)	Equation (14)	Difference
`toyeliece51220`	0.261	0.316	0.055
`toyeliece102450`	0.340	0.383	0.043
`mceliece348864`	0.360	0.396	0.036
`mceliece460896`	0.384	0.415	0.031
`mceliece6960119`	0.395	0.424	0.029
`mceliece6688128`	0.398	0.427	0.029
`mceliece8192128`	0.398	0.427	0.029

6 Experimental Verification

To validate our attack model and the practical efficiency of our attack we investigate two concrete implementations of McEliece:

1. The reference implementation of *Classic McEliece* [8], a 4th-round submission to the NIST's Post-Quantum Cryptography Standardization Project [3].
2. The cryptographic library Botan [1], which has been recognized by the German Federal Institute for Information Security as a secure implementation [2].

In both libraries, the details of Gaussian elimination during public-key generation are considerably different from the naive description in Algorithm 2.

For Classic McEliece, these differences require us to slightly modify the recovery of the transformation matrices $\mathbf{G}_j$ in our algorithm Secret-Key-Recovery. The necessary modifications are, however, straight-forward. For Botan, the differences do not require any changes at all. Hence, even though real-world implementations of Gaussian elimination are quite different from our naive Algorithm 2, our attack also applies to them.

In the following sections, we describe the substantial differences between our naive Algorithm 2, Classic McEliece and Botan, and discuss the potential for leakages in our attacker model (Sections 6.1 and 6.2). Additionally, we explain the details of our attack implementation (Section 6.3). Furthermore, we show our experimental results to validate the practical efficiency of our approach, and the correctness of our heuristic Assumption 1 (Sections 6.4 and 6.5).

6.1 Classic McEliece

Algorithm 5 outlines the operation of Gaussian elimination in the reference implementation of Classic McEliece [8].

Algorithm 5 Gaussian Elimination from Classic McEliece Key Generation

Input: $\mathbf{H} \in \mathbb{F}_2^{tm \times n}$
Output: $(\mathbf{I}_{tm}|\mathbf{A})$ or FAIL

```
 1: for j = 1, ..., tm do
 2:     for i = j + 1, ..., tm do                    ▷ Begin of diagonal stage
 3:         mask := H[i, j] + H[j, j]
 4:         H[j, :] := H[j, :] + mask · H[i, :]
 5:     end for                                       ▷ End of diagonal stage
 6:     if H[j, j] ≠ 1 then
 7:         return FAIL                ▷ can not bring H into systematic form
 8:     end if
 9:     for i = 1, ..., tm, i ≠ j do                 ▷ Begin of zero stage
10:         mask := H[i, j]                            ▷ Potential leak: mask
11:         H[i, :] := H[i, :] + mask · H[j, :]
12:     end for                                           ▷ End of zero stage
13: end for
```

Like the naive Gaussian elimination in Algorithm 2, Classic McEliece iterates over the first tm columns to bring the matrix into a systematic form. For each column, the algorithm ensures two conditions. The first one, the *diagonal stage* (Lines 2–5), is to ensure $\mathbf{H}[j, j] = 1$. The second one, the *zero stage* (Lines 9–12), ensures that $\mathbf{H}[i, j] = 0$ for $i \neq j$.

To minimize timing side-channel leaks in the first stage (the diagonal stage), the implementation adds in the j-th iteration a fixed number of $tm - j$ rows to the j-th row, i.e., the number of row additions only depends on the index j, but not on secret data. Similarly, the second stage of column elimination (the zero stage) also performs a constant number of row additions in each iteration to minimize timing side-channel leaks.

Thus, the Classic McEliece implementation does more row additions than our naive Algorithm 2. In our algorithm SECRET-KEY-RECOVERY, we have to account for these additional row additions when recovering a transformation matrix $\mathbf{G}_j$ from the corresponding execution matrix column $\mathbf{E}[:, j]$.

Potential Leak Analysis. While the implementation contains neither secret-dependent branches nor secret-dependent memory access patterns, and is therefore protected against traditional timing-based side-channel attacks, leaks through other side channels remain possible. Specifically, the row addition operations conditionally add either a zero row or arbitrary values to a row, depending on the value of $\mathsf{mask} = \mathbf{H}[i, j]$. These two cases have very different Hamming weights. In a zero row all bits are zero, whereas in arbitrary values roughly half the bits are one.[7] Moreover, when adding a zero row, the algorithm does not change the contents of memory, whereas in the other case the contents changes. Thus,

[7] More accurately, the $j - 1$ bits left of the diagonal of the non-zero row added when processing column j are all zero, but the $n - j$ bits to the right are random. Hence the Hamming weight of this row is expected to be $(n - j)/2$.

the Hamming distance between the values in memory and the values written depends on whether the algorithm adds a zero row or not.

The power consumption and the electromagnetic emanations observed during program execution correlate with the Hamming weight of the data that the program processes and with the Hamming distance between old and new values when writing to memory. Consequently, constant-time implementations tend to leak through these channels [6, 30].

6.2 Botan

Algorithm 6 outlines the operation of Gaussian elimination for McEliece key generation in the cryptographic library Botan version 3.1.1 [1].

Botan's implementation of Gaussian elimination is considerably different from the naive one in Algorithm 2. Although we can identify the two stages, diagonal stage (Lines 5–12) and zero stage (Lines 24–28), there is also a *swap stage* (Lines 13–22). The swap stage allows Botan to avoid failing key generation, when the first tm columns of $\mathbf{H}$ do not have full rank. It does so by considering *all* columns of $\mathbf{H}$ (as opposed to only the first tm columns in our Algorithm 2). Suitable columns for the systematic form are swapped to indices $n-tm+1, n-tm+2, \ldots, n$, while columns that are linearly dependent to already chosen columns are swapped to indices $n-tm, n-tm-1, \ldots, n-tm-\mathsf{failcount}$, where failcount is the number of unsuitable columns. As a result, key generation can succeed more often compared to the naive approach.

The final result is a slightly different systematic form $\mathbf{H}_{\mathsf{pk}} = (\mathbf{A}|\mathbf{I}_{tm})$, and a permutation π that is applied to the list of Goppa points L to adjust the secret key according to the column swaps made by Botan's Gaussian elimination. As our leakage model only depends on the zero stage of Gaussian elimination, which is run after the final position of a column has been decided, the order of the leaked data, however, perfectly matches the order of the entries in the (adjusted) secret key. In other words, the column swaps do not require any changes in our algorithm SECRET-KEY-ALGORITHM.

As another difference, Botan's zero-stage iterates over all rows $i \neq j$ in an unusual order. This needs to be taken into account when constructing the leak matrix $\mathbf{L}$ from the leak data, but otherwise has no effect on our attack.

Potential Leak Analysis. Our investigation of Gaussian elimination in Botan reveals that the implementation contains conditional branches, depending on secret data. This indicates that Botan implementation has potential to be vulnerable to side-channel attacks.

Unlike the Classic McEliece implementation that ensures the same number of iterations even when selectively add rows containing certain values, Botan simply uses branch statements to select specific rows to perform addition. Furthermore, when the row addition is performed, there is also an associated memory access pattern. Therefore, through side channels, it is possible to determine whether the branch condition to perform row operation is evaluated to true. To be more

Algorithm 6 Gaussian Elimination from Botan Key Generation

Input: $\mathbf{H} \in \mathbb{F}_2^{tm \times n}$
Output: $((\mathbf{A}|\mathbf{I}_{tm}), \pi)$ or FAIL

```
1:  failcount := 0
2:  c := n                                    ▷ Take columns one-by-one from right.
3:  π := [1, ..., n]                          ▷ Remember secret key adjustments, see text.
4:  for j = 1, ..., tm do
5:      for i = j, ..., tm do                 ▷ Begin of diagonal stage.
6:          if H[i, c] = 1 then
7:              if i ≠ j then
8:                  H[j, :] := H[j, :] + H[i, :]
9:              end if
10:             break
11:         end if
12:     end for                               ▷ End of diagonal stage.
13:     if H[j, j] ≠ 1 then                   ▷ Begin of swap stage.
14:         failcount := failcount + 1
15:         if failcount = n − tm then
16:             return FAIL                   ▷ can not bring H into systematic form
17:         else
18:             π[n − tm + 1 − failcount] := c   ▷ c unsuitable, move to A part of H.
19:             c := c − 1
20:             goto line 5
21:         end if
22:     end if                                ▷ End of swap stage.
23:     π[n − tm + j] := c                    ▷ c suitable, move to I_tm part of H.
24:     for i = j + 1, j + 2, ..., tm, j − 1, j − 2, ..., 1 do   ▷ Begin of zero stage.
25:         if H[i, c] = 1 then               ▷ Potential leak: whether H[i, c] = 1
26:             H[i, :] := H[i, :] + H[j, :]
27:         end if
28:     end for                               ▷ End of zero stage.
29:     c := c − 1
30: end for
31: return (H, π)
```

precise, there is a potential leak of whether $\mathbf{H}[i, c] = 1$ in Line 25 of Algorithm 6 — completely analogous to our simplified leak model from Section 3.

Botan's Choice of Goppa Points. We finish our description of Botan with an observation about its choice of Goppa points. Instead of choosing those values at random, Botan chooses those Goppa points from a predictable set depending on n. The procedure is as follows. First, it chooses a random permutation of the numbers $0, 1, 2, \ldots, n-1$. Then, it maps each number to a corresponding entry in a Gray code using a deterministic function. Finally, Botan interprets these numbers as elements in $\mathbb{F}_{2^m}$ to form the list of Goppa points L. As a consequence, the set of Goppa points is known up to the order of its elements,

and the Support Splitting algorithm can be used to optimize the attack for any value of n, not only for $n = 2^m$. Therefore, only the first $t + 1$ columns have to be leaked and recovered. As such, the success probability only depends on the first $t + 1$ columns.

Thanks to the predictable set of Goppa points, the initial codebook can also be optimized by removing all those values for α_j that do not occur in the first n entries in the Gray code. Note that the initial code has a size of $|\mathsf{CB}_1| = n(2^m - 1)$ as opposed to $2^m(2^m - 1)$. This reduces the attack complexity up to a factor of 2 in the common case of $2^{m-1} < n < 2^m$.

6.3 Implementation

We implement our attack and provide the source code that is used to generate all artefacts in a Git repository:

https://github.com/lambdafu/Leaky-McEliece

To generate leak data, we patch the Botan and Classic McEliece reference implementations to artificially leak the entries $\mathbf{E}[i, j]$, where $i \neq j$, of the execution matrix that are not on the diagonal, without error. Then, we simulate the error by flipping the leaked bits with adjustable probability τ before running our attack.

While we would have preferred a SageMath implementation, we opted for C++, because some of the required operations, like the generation and update of the size-2^{2m} codebook, do not have efficient support in SageMath. Our implementation uses bit-packing for columns to group packs of 64 rows into a single machine word. This reduces memory usage and allows for more efficient codebook updates. All operations over columns, including decoding and row additions, are parallelized (using [39]). The generated candidate matrices are cached, allowing for fast startup times when attacking multiple keys from the same parameter set.

6.4 Experiments

A single experiment is defined by a target implementation (Classic McEliece or Botan), a McEliece parameter set (n, t, m), an error probability τ, a random seed s_1 for the (leaky) key generation: $s_1 \mapsto (L, g) \mapsto \mathbf{H}_{\mathsf{sk}} \mapsto (\mathbf{H}_{\mathsf{pk}}, \mathbf{E})$, and a second random seed s_2 for the error perturbation: $(s_2, \tau) \mapsto \mathbf{e}$. The experiment targets recovering $\mathbf{H}_{\mathsf{sk}}$ from $\mathbf{L} := \mathbf{E} + \mathbf{e}$ (see Definition 1, in particular, Equation (7)).

We perform all computations on a Dual AMD Epyc 7763 with 2 TB memory and 128 cores. In our experiments, we set the error rate τ to be in the range of 0.00–0.50 with an incremental step of 0.01. (For larger parameter sets, we only considered error rates in the range 0.30–0.50.) For each error rate, we repeat the experiment 100 times with independent random seeds. The average success rate is plotted as a function of the error rate τ in Figure 2.

From these measurements, we used linear interpolation to determine the maximum error probability for which the success probability of the attack is larger than $\frac{1}{2}$. These threshold values are given in Table 3. The same table also presents the average execution time.

Table 3: Experiment execution time and the threshold error rate $\tau_{\mathsf{Threshold}}$, where the success probability crosses $\frac{1}{2}$, based on linear interpolation of neighbouring measurements.

	Classic McEliece		Botan	
Name	Wall Time	$\tau_{\mathsf{Threshold}}$	Wall Time	$\tau_{\mathsf{Threshold}}$
`toyeliece51220`	4sec	0.260	3sec	0.261
`toyeliece102450`	5sec	0.340	5sec	0.340
`mceliece348864`	19sec	0.361	15sec	0.360
`mceliece460896`	1min 30sec	0.384	46sec	0.386
`mceliece6960119`	2min 1sec	0.395	1min 20sec	0.395
`mceliece6688128`	2min 10sec	0.397	1min 22sec	0.400
`mceliece8192128`	2min 11sec	0.398	1min 41sec	0.398

Results. Our experimental verification shows that the success probability of our attack on Classic McEliece and Botan is very well aligned with our model, as shown in Figure 2.

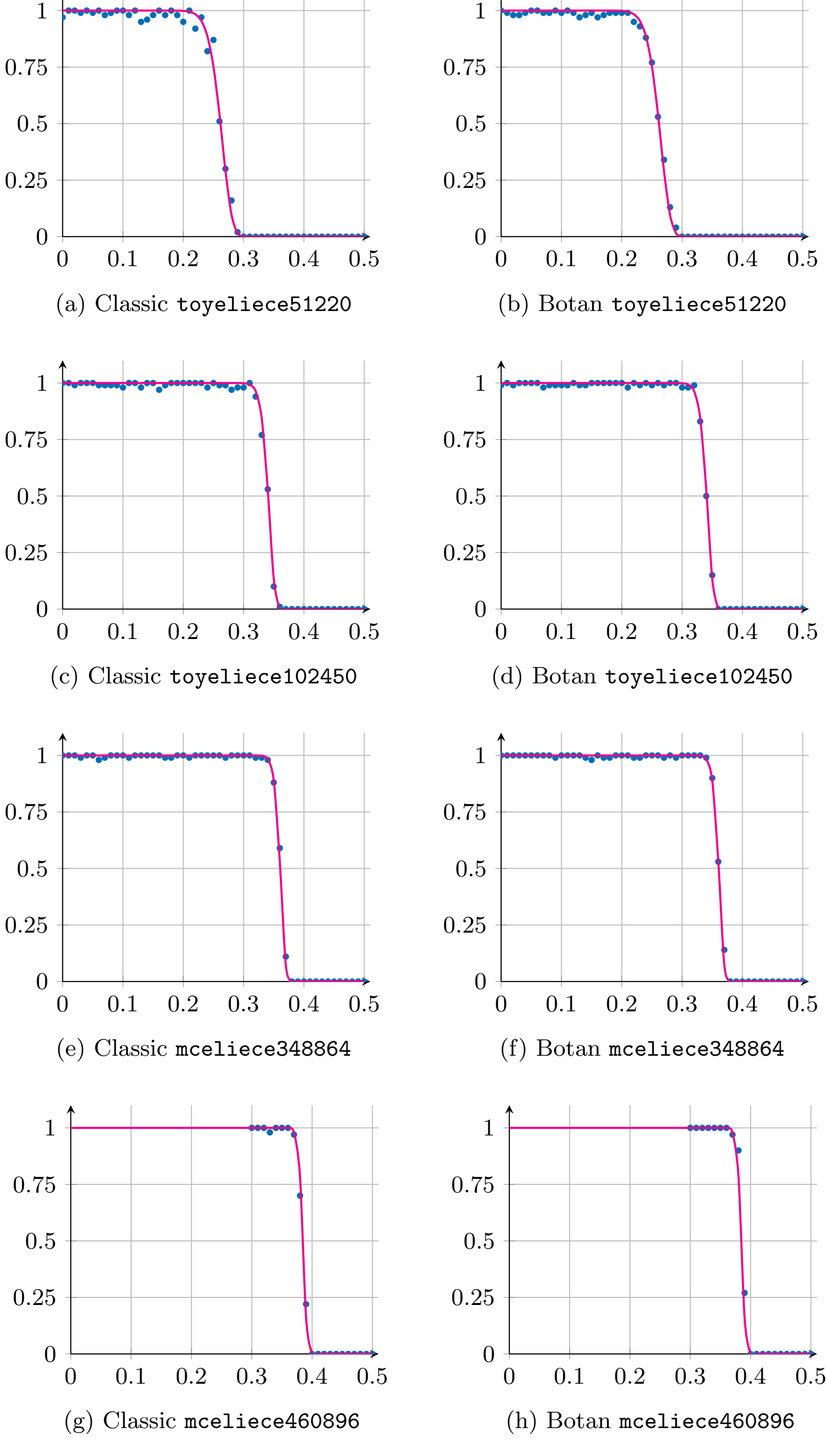
We note that the execution time of the attack on Classic McEliece is slightly slower than attacking Botan (see Table 3). This can be easily explained by the extra row additions performed by Classic McEliece, and the smaller codebook size (due to the deterministic Goppa point generation). In practice, this makes our attack on Botan 25%–50% faster compared to that of Classic McEliece.

The applicability of support splitting and the reduced codebook size nominally improves the success probability for our attack on Botan by a small amount. However, the effect is marginal and not visible in our experimental data.

6.5 Support Splitting

In a setting where the Goppa points $\alpha_1, \ldots, \alpha_n$ are known (e.g., when $n = 2^m$ or when using an implementation with deterministic Goppa point generation, such as Botan), our algorithm Secret-Key-Recovery uses the Support Splitting algorithm (SSA) as a subroutine.

To the best of our knowledge, there is no open source implementation of SSA available. In particular, there seems to be no publicly available data showing how SSA performs on Classic McEliece parameter sets.



(a) Classic `toyeliece51220`

(b) Botan `toyeliece51220`

(c) Classic `toyeliece102450`

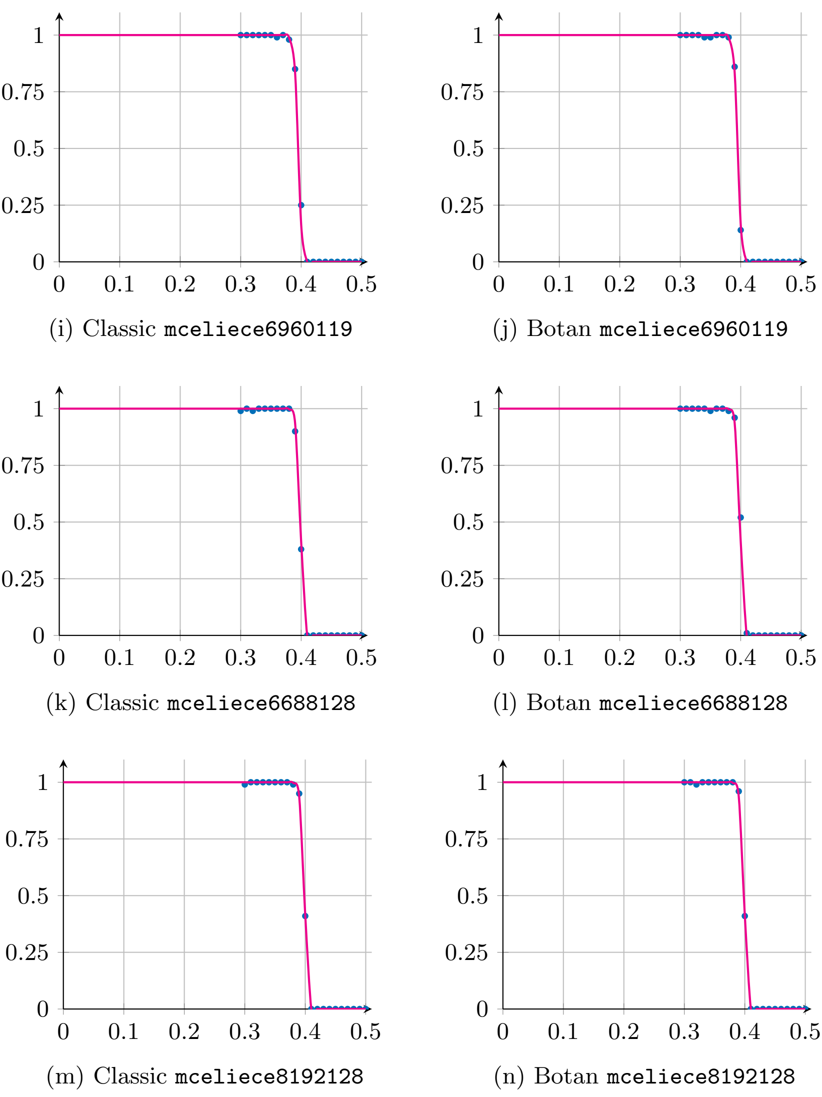
(d) Botan `toyeliece102450`

(e) Classic `mceliece348864`

(f) Botan `mceliece348864`

(g) Classic `mceliece460896`

(h) Botan `mceliece460896`



(i) Classic `mceliece6960119` (j) Botan `mceliece6960119`

(k) Classic `mceliece6688128` (l) Botan `mceliece6688128`

(m) Classic `mceliece8192128` (n) Botan `mceliece8192128`

Fig. 2: Comparison between experimental results (dots) and estimates from Equation (18) (curves) for our attack on Classic McEliece (left) and Botan (right) parameter sets. Horizontal axes show the error probability τ of the leak. Vertical axes show the success probability of our attack.

Since the runtime analysis in Sendrier's original paper is only heuristic, it is important to verify that SSA is indeed efficient for McEliece parameters. To this end, we implement the SSA in SageMath and run it on various parameter sets. The results are depicted in Table 4. As the table shows, the algorithm is highly efficient. Even for the high-security parameter sets, the algorithm terminates in less than 5 minutes.

Table 4: Average, minimal and maximal runtime of our Support Splitting implementation on 10 random instances per parameter set.

Name	t_{avg}	t_{min}	t_{max}
`toyeliece51220`	$<$ 1sec	$<$ 1sec	2sec
`toyeliece102450`	3sec	1sec	4sec
`mceliece348864`	32sec	13sec	45sec
`mceliece460896`	52sec	22sec	1min 40sec
`mceliece6960119`	2min 30sec	51sec	4min 10sec
`mceliece6688128`	2min 2sec	47sec	3min 9sec
`mceliece8192128`	3min 1sec	1min 12sec	4min 50sec

Acknowledgements

This work has been supported by: an ARC Discovery Project number DP210102670; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972; and DFG grant 465120249.

References

1. Botan 3.1.1: code_based_key_gen.cpp. https://botan.randombit.net/doxygen/code__based__key__gen_8cpp_source.html, accessed: 5 October 2023
2. Bsi-projekt: Entwicklung einer sicheren kryptobibliothek, https://www.bsi.bund.de/dok/9060550
3. NIST's post-quantum cryptography standardization project. https://csrc.nist.gov/projects/post-quantum-cryptography
4. Aciiçmez, O., Brumley, B.B., Grabher, P.: New results on instruction cache attacks. In: CHES. Lecture Notes in Computer Science, vol. 6225, pp. 110–124. Springer (2010)
5. Aciiçmez, O., Gueron, S., Seifert, J.: New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In: IMACC. Lecture Notes in Computer Science, vol. 4887, pp. 185–203. Springer (2007)

6. Alam, M., Yilmaz, B.B., Werner, F., Samwel, N., Zajic, A.G., Genkin, D., Yarom, Y., Prvulovic, M.: Nonce@Once: A single-trace EM side channel attack on several constant-time elliptic curve implementations in mobile platforms. In: EuroS&P. pp. 507–522. IEEE (2021)
7. Albrecht, M.R., Deo, A., Paterson, K.G.: Cold boot attacks on ring and module LWE keys under the NTT. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(3), 173–213 (2018)
8. Bernstein, D.J., Chou, T., Cid, C., Gilcher, J., Lange, T., Maram, V., von Maurich, I., Misoczki, R., Niederhagen, R., Persichetti, E., Peters, C., Sendrier, N., Szefer, J., Tjhai, C.J., Tomlinson, M., Wang, W.: Classic McEliece. https://classic.mceliece.org/mceliece-sage-20221023.tar.gz (2022)
9. Bernstein, D.J., Chou, T., Cid, C., Gilcher, J., Lange, T., Maram, V., von Maurich, I., Misoczki, R., Niederhagen, R., Persichetti, E., Peters, C., Sendrier, N., Szefer, J., Tjhai, C.J., Tomlinson, M., Wang, W.: Classic McEliece: conservative code-based cryptography: design rationale (2022), https://classic.mceliece.org/mceliece-rationale-20221023.pdf
10. Bruinderink, L.G., Hülsing, A., Lange, T., Yarom, Y.: Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In: Gierlichs, B., Poschmann, A.Y. (eds.) CHES 2016. LNCS, vol. 9813, pp. 323–345. Springer, Heidelberg (Aug 2016). https://doi.org/10.1007/978-3-662-53140-2_16
11. Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In: ESORICS. Lecture Notes in Computer Science, vol. 6879, pp. 355–371. Springer (2011)
12. Chuengsatiansup, C., Feutrill, A., Sim, R.Q., Yarom, Y.: RSA key recovery from digit equivalence information. In: ACNS. Lecture Notes in Computer Science, vol. 13269, pp. 193–211. Springer (2022)
13. Coppersmith, D.: Finding a small root of a univariate modular equation. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 1070, pp. 155–165. Springer (1996)
14. Cover, T.M.: Elements of information theory. John Wiley & Sons (1999)
15. De Micheli, G., Heninger, N.: Recovering cryptographic keys from partial information, by example. IACR ePrint 2020/1506 (2020)
16. Erata, F., Piskac, R., Mateu, V., Szefer, J.: Towards automated detection of single-trace side-channel vulnerabilities in constant-time cryptographic code. In: EuroS&P. pp. 687–706. IEEE (2023)
17. Esser, A., May, A., Verbel, J.A., Wen, W.: Partial key exposure attacks on BIKE, rainbow and NTRU. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part III. LNCS, vol. 13509, pp. 346–375. Springer, Heidelberg (Aug 2022). https://doi.org/10.1007/978-3-031-15982-4_12
18. Genkin, D., Shamir, A., Tromer, E.: RSA key extraction via low-bandwidth acoustic cryptanalysis. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 444–461. Springer, Heidelberg (Aug 2014). https://doi.org/10.1007/978-3-662-44371-2_25
19. Guo, Q., Johansson, A., Johansson, T.: A key-recovery side-channel attack on classic mceliece implementations. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(4), 800–827 (2022)
20. Guo, Q., Nabokov, D., Nilsson, A., Johansson, T.: SCA-LDPC: A code-based framework for key-recovery side-channel attacks on post-quantum encryption schemes. Cryptology ePrint Archive, Report 2023/294 (2023), https://eprint.iacr.org/2023/294

21. Henecka, W., May, A., Meurer, A.: Correcting errors in RSA private keys. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 351–369. Springer, Heidelberg (Aug 2010). https://doi.org/10.1007/978-3-642-14623-7_19
22. Heninger, N., Shacham, H.: Reconstructing RSA private keys from random key bits. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 1–17. Springer, Heidelberg (Aug 2009). https://doi.org/10.1007/978-3-642-03356-8_1
23. Howgrave-Graham, N.: Finding small roots of univariate modular equations revisited. In: IMACC. Lecture Notes in Computer Science, vol. 1355, pp. 131–142. Springer (1997)
24. Howgrave-Graham, N., Smart, N.P.: Lattice attacks on digital signature schemes. Des. Codes Cryptogr. **23**(3), 283–290 (2001)
25. Huang, S., Sim, R.Q., Chuengsatiansup, C., Guo, Q., Johansson, T.: Cache-timing attack against HQC. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2023**(3), 136–163 (2023)
26. Kannwischer, M.J., Pessl, P., Primas, R.: Single-trace attacks on keccak. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(3), 243–268 (2020)
27. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO'96. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (Aug 1996). https://doi.org/10.1007/3-540-68697-5_9
28. Lahr, N., Niederhagen, R., Petri, R., Samardjiska, S.: Side channel information set decoding using iterative chunking - plaintext recovery from the "classic McEliece" hardware reference implementation. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part I. LNCS, vol. 12491, pp. 881–910. Springer, Heidelberg (Dec 2020). https://doi.org/10.1007/978-3-030-64837-4_29
29. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: IEEE Symposium on Security and Privacy. pp. 605–622. IEEE Computer Society (2015)
30. Nascimento, E., Chmielewski, L., Oswald, D.F., Schwabe, P.: Attacking embedded ECC implementations through cmov side channels. In: SAC. Lecture Notes in Computer Science, vol. 10532, pp. 99–119. Springer (2016)
31. Nguyen, P.Q., Shparlinski, I.E.: The insecurity of the digital signature algorithm with partially known nonces. J. Cryptol. **15**(3), 151–176 (2002)
32. Nguyen, P.Q., Shparlinski, I.E.: The insecurity of the elliptic curve digital signature algorithm with partially known nonces. Des. Codes Cryptogr. **30**(2), 201–217 (2003)
33. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: CT-RSA. Lecture Notes in Computer Science, vol. 3860, pp. 1–20. Springer (2006)
34. Park, A., Shim, K., Koo, N., Han, D.: Side-channel attacks on post-quantum signature schemes based on multivariate quadratic equations - Rainbow and UOV -. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(3), 500–523 (2018)
35. Paterson, K.G., Polychroniadou, A., Sibborn, D.L.: A coding-theoretic approach to recovering noisy RSA keys. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 386–403. Springer, Heidelberg (Dec 2012). https://doi.org/10.1007/978-3-642-34961-4_24
36. Riebler, H., Kenter, T., Plessl, C., Sorge, C.: Reconstructing AES key schedules from decayed memory with FPGAs. In: FCCM. pp. 222–229. IEEE Computer Society (2014)
37. Schamberger, T., Holzbaur, L., Renner, J., Wachter-Zeh, A., Sigl, G.: A power side-channel attack on the reed-muller reed-solomon version of the hqc cryptosystem. In: Post-Quantum Cryptography. pp. 327–352 (2022)

38. Sendrier, N.: Finding the permutation between equivalent linear codes: The support splitting algorithm. IEEE Trans. Inf. Theory **46**(4), 1193–1203 (2000)
39. Shoshany, B.: A C++17 thread pool for high-performance scientific computing. arXiv e-prints arXiv:2105.00613 (May 2021). https://doi.org/10.5281/zenodo.4742687
40. Strenzke, F.: A timing attack against the secret permutation in the McEliece PKC. In: Sendrier, N. (ed.) The Third International Workshop on Post-Quantum Cryptography, PQCRYPTO 2010. pp. 95–107. Springer, Heidelberg (May 2010). https://doi.org/10.1007/978-3-642-12929-2_8
41. Strenzke, F., Tews, E., Molter, H.G., Overbeck, R., Shoufan, A.: Side channels in the McEliece PKC. In: Buchmann, J., Ding, J. (eds.) Post-quantum cryptography, second international workshop, PQCRYPTO 2008. pp. 216–229. Springer, Heidelberg (Oct 2008). https://doi.org/10.1007/978-3-540-88403-3_15
42. Ueno, R., Xagawa, K., Tanaka, Y., Ito, A., Takahashi, J., Homma, N.: Curse of re-encryption: A generic power/EM analysis on post-quantum KEMs. IACR TCHES **2022**(1), 296–322 (2022). https://doi.org/10.46586/tches.v2022.i1.296-322
43. Walter, C.D.: Sliding windows succumbs to big mac attack. In: Koç, Çetin Kaya., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 286–299. Springer, Heidelberg (May 2001). https://doi.org/10.1007/3-540-44709-1_24
44. Yarom, Y., Falkner, K.: Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In: USENIX Security Symposium. pp. 719–732. USENIX Association (2014)
45. Zhang, Z., Tao, M., O'Connell, S., Chuengsatiansup, C., Genkin, D., Yarom, Y.: BunnyHop: Exploiting the instruction prefetcher. In: USENIX Security Symposium. USENIX Association (2023)