IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 2, pp. 94–125. DOI:10.46586/tches.v2025.i2.94-125

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, Bochum, Germany
[2]The University of Klagenfurt, Klagenfurt, Austria

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 highly critical with respect to side-channel leakage. We assume leakage of the elementary row operations during Gaussian elimination, motivated by McEliece implementations in the cryptographic libraries Classic McEliece and Botan.
We propose a novel decoding 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 is flipped with probability as high as $\tau \approx 0.4$, we succeed 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.
We demonstrate our attack on the constant-time reference implementation of Classic McEliece in a single-trace setting, using an STM32L592 ARM processor.
Our result stresses the necessity of properly protecting 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

In 2024, NIST released the first post-quantum cryptography standard for key encapsulation methods, ML-KEM [Nat24]. As the accepted algorithm is lattice-based, NIST continued the competition in a fourth round and encouraged researchers to contribute KEM schemes that are based on other security guarantees. One such contender is Classic McEliece [BCC+22a], which relies on assumptions in code-based cryptography. In this work we analyze how robust McEliece is against certain side-channel attacks during public key generation.

Related work. The seminal paper of Kocher [Koc96] 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 [Koc96, OST06, NCOS16, GST14] and breaking a large number of implementations [OST06, Wal01, BT11, GBHLY16, KPP20, PSKH18].

Since side channels are unintended byproducts of physical phenomena, the information obtained through them is often noisy and incomplete. Over the years several methods have been devised to recover the full secret, exploiting the inherent redundancy in some

Received: 2024-10-15 Accepted: 2024-12-15 Published: 2025-03-04

cryptographic schemes [DMH20]. Most of the research in this area focuses on traditional schemes, such as AES [RKPS14], RSA [HS09, Cop96, How97, HMM10, PPS12, ZTO+23, CFSY22], and variations of DSA [HS01, NS02, NS03].

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 [SHR+22, HSC+23], Kyber and New Hope [ADP18], BIKE, Rainbow and NTRU [EMVW22], and the Fujisaki-Okamoto transform [UXT+22, GNNJ23].

Specifically for McEliece, Strenzke et al. [STM+08] 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. Strenzke [Str10] presents an attack on the McEliece secret key, which uses a timing side channel in the decryption routine. A recent attack on Classic McEliece reveals the secret key using a power side channel in a decryption oracle [GJJ22]. Yet another attack shows the use of an electromagnetic side channel for revealing the plaintext of a message [LNPS20].

Attacker Model: Erasures and Error Rate. Our attacker model originates from so-called cold boot attacks [HSH+08]. In a cold boot attack, one uses the physical effect that volatile memory for a short period of time still retains its content. This effect is sufficient in practice to read out an erroneous version of a secret key, even after switching off power supply. The task is now to recover the key from its erroneous version. In this sense, a cold boot attack is just a special case of a more general side-channel secret key leakage measurement. Depending on the quality of a side-channel/cold boot measurement, one classifies the leaked erroneous secret key bits.

In the simplest *erasure setting*, one keeps only those bits that are correct with (almost) certainty, all other bits are modeled as erasures and have to be reconstructed. The ratio $\sigma < 1$ of erased bits to all bits is called the *erasure rate*. Notice that for $\sigma = 1$ we erase all bits. Thus, achieving an error rate close to 1 indicates a powerful secret key reconstruction algorithm in the erasure setting.

In the more realistic *error setting*, one keeps *all erroneous bits* of a measured secret key, but assumes that each bit has independently been flipped from its correct to a faulty value with *error rate* $\tau < \frac{1}{2}$. The goal is to correct all *bit-flip errors* in the measured secret key for an error rate τ as large as possible. Notice that an erroneous key with error rate $\tau = \frac{1}{2}$ does not provide any information on the underlying key. Thus, achieving an error rate close to $\frac{1}{2}$ indicates a powerful key reconstruction algorithm in the error setting.

For our McEliece results, we consider only the more realistic *error setting*.

Related Classical Results within our Attacker Model. In the *erasure setting*, Heninger and Shacham [HS09] showed how to recover RSA secret keys with an *erasure rate* of $\sigma = 0.73$, as long as the remaining known bits are evenly distributed at known positions. In the more challenging *error setting*, the recovery algorithm of Henecka et al. [HMM10] allows 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 rates were further improved by Paterson et al. [PPS12].

While RSA was known to be vulnerable to partial information leakage, post-quantum schemes were believed to be more leakage-resistant. This view was challenged by Esser et al. [EMVW22], who found recovery attacks on BIKE, Rainbow and NTRU in the erasure as well as in the error setting. Allowing for an attack complexity of 80 bits, they achieved secret key recovery with erasure rates σ up to 0.730 for BIKE, 0.890 for Rainbow, and 0.422 for NTRU, as well as with error rates τ up to 0.200 for BIKE, 0.270 for Rainbow, and 0.019 for NTRU. Earlier, Albrecht et al. [ADP18] 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. 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. In this work, we investigate key recovery from a *single-trace* side-channel observation of the public key generation of McEliece, which runs Gaussian elimination on a binary matrix, i.e., over $\mathbb{F}_2$. Gaussian elimination performs addition of rows in a pattern that depends on the value of the bits in the matrix. We show that an attacker with (noisy) oracle access to row additions of Gaussian elimination is able to recover the bits of the secret key matrix.

Attacks on key generation with a noisy measurement present a significant challenge. Unlike attacks on e.g. decryption, that can be repeated multiple times to average out the noise, key generation is usually executed only once. Thus, an attacker needs to be able to handle noisy data with a single trace only.

We would like to stress that we attack the generation of the public key, not the generation of the secret key. In April 2024, Schmieg, Connolly and Westerbaan [SCW24] proposed in an official comment on the NIST PQC Forum that when a (ML-KEM) secret key is transferred from one system to another, only a *seed* of the key should be transferred, and the key should be re-generated at the receiver's side following a deterministic key generation algorithm. This proposal does not affect our work either way, as we are concerned with public (not secret) key generation, and Classic McEliece does not require the public key for decapsulation. However, if a similar approach was to be used for public key re-generation, then this would provide more trace opportunities for our attack, and thus allow for an improved error correction. In this work we do not assume such an approach, and instead focus on single-trace attacks.

1.1 Our Contributions

Secret Key Recovery From Noisy Gauss Elimination Leak. We present an attacker model for McEliece public key generation, where an attacker observes 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 $\tau \approx 0.4$. 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.

Practical Evaluation of Leakage Potential. We investigate two real-world implementations of McEliece, the one in the Botan cryptographic library [cod], which has been recognized by the German Federal Institute for Information Security [bot], and the reference implementation of the Classic McEliece submission to the NIST Post-Quantum Cryptography Standardization Project [BCC+22a].

The Botan implementation does not use constant-time coding practices and is therefore vulnerable to side-channel attacks that leak control flow [YF14, ZTO+23, ABG10, AGS07] and memory access patterns [LYG+15, OST06]. These may leak the location of row additions. In our experiments, we therefore focus on the more challenging constant-time Classic McEliece implementation, which however is not hardened against differential power analysis attacks. We experimentally verify that the Gaussian elimination code in this implementation leaks the locations of row additions. Specifically, we show that by observing the power consumption of the CPU during a single Gaussian elimination, an attacker can recover the locations of row addition operations with an error rate of $\tau \approx 0.11$, which is well within the bounds of our recovery algorithm. We demonstrate a full end-to-end attack on the Gaussian eliminiation in `mceliece8192128`, running on an STM32L592 ARM processor.

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 elimination successively column by column. Because of the high redundancy in the parity check matrix of a 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).

Experimental Verification. In a large-scale experiment, we measure the success probability and runtime of our attack on a variety of proposed security parameters against both libraries, Botan and Classic McEliece. We find that the success probabilities are in agreement with our analysis, thus verifying our heuristic assumptions. Our algorithm is highly practical, full secret key recovery for the largest security McEliece parameter set takes only 131 seconds.

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) [Sen00] can (heuristically) be used to 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 publicly available implementation of SSA. We contribute a fully working implementation of SSA in SageMath, and verify experimentally that it is efficient for the specific case of McEliece parameter sets.

Artifacts Availability. The source code for our attack, the implementation of the Support Splitting Algorithm, and the artifacts for the evaluation, are available at:

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

1.2 Outline

In Section 2, we introduce notations for matrices, Hamming spaces and codes. We also describe McEliece keys, how their generation uses Gaussian elimination, and briefly recall how to attack McEliece via the Support Splitting Algorithm. In Section 3, we define our leakage model. After that, we introduce our attack in Section 4, and analyze its success probability in Section 5. In Section 6, we analyze two implementations, Botan and Classic McEliece, identify their potential leakage, and verify the leakage of Classic McEliece on real hardware. For the Classic McEliece parameter set with the highest security level, we successfully run an end-to-end attack. In Section 7, we report on experiments for additional parameter sets to measure the success rate and runtime of the attack. In contrast to the end–to-end attack of Section 6, we use simulated leakage in Fig. 8, confirming the correctness of our heuristic analysis. In Section 8, we discuss potential countermeasures.

2 Preliminaries

2.1 Notations

Matrices. Let $\mathbf{A}$ be a $(k \times n)$-matrix. 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 $\Delta(\mathbf{x}, \mathbf{y})$, i.e, $\Delta(\mathbf{x}, \mathbf{y})$ counts the number of coordinates on which $\mathbf{x}$ and $\mathbf{y}$ differ. The *Hamming weight* of $\mathbf{x}$, denoted $\omega(\mathbf{x})$, is defined as the Hamming distance between $\mathbf{x}$ and the all-zero word 0^n. The n-dimensional *Hamming ball* around $\mathbf{x} \in \mathbb{F}_2^n$ with radius $r \geq 0$ is defined as $B(\mathbf{x}, r) := \{\mathbf{y} \in \mathbb{F}_2^n \mid \Delta(\mathbf{x}, \mathbf{y}) \leq r\}$. For the volume $V^n(r)$ of an n-dimensional Hamming ball with radius $r \in \mathbb{N}$ and $r \leq \frac{n}{2}$, we have

$$V^n(r) = \sum_{i=0}^{r} \binom{n}{i} \approx \binom{n}{r} \approx 2^{\mathrm{H}(r/n)n}, \tag{1}$$

where $\mathrm{H}(\cdot)$ denotes the binary entropy function. Recall that for $0 < x < 1$ the binary entropy is defined as $\mathrm{H}(x) := -x \log(x) - (1-x)\log(1-x)$, whereas for $x \in \{0, 1\}$ it is defined as $\mathrm{H}(x) := 0$. (Here, and throughout the paper, $\log(\cdot)$ denotes the base-2 logarithm.) The approximations in Eq. (1) suppress only small polynomial factors in n, see, e.g., [Cov99, 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

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

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

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

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

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

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 $n \leq 2^m$ 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. We detail the Gaussian elimination in 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 [BCC+22b] 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 $\mathbf{S} \in \mathbb{F}_2^{n \times n}$ and a permutation matrix $\mathbf{P} \in \mathbb{F}_2^{n \times n}$ such that $\mathbf{H}' = \mathbf{S} \cdot \mathbf{H} \cdot \mathbf{P}$. Given two parity check matrices $\mathbf{H}, \mathbf{H}'$ of two permutation equivalent linear codes C, C', Sendrier's Support Splitting Algorithm (SSA) [Sen00] 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.[1]

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 $\{\alpha_1, \ldots, \alpha_n\}$ (but without their correct *order* $L = (\alpha_1, \ldots, \alpha_n)$). Given $g(x)$ and the set of Goppa 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 Eq. (3). In particular,

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

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

Let $\mathbf{S} \in \mathbb{F}_2^{tm \times tm}$ 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.,

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

Then it holds that

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

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.[2]

[1] 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 *hull* of C. For random codes, h is with high probability a small constant. Typically, $h \in \{0, 1, 2\}$.

[2] Notice that the call to Eliminate-Column in Line 3 of Gaussian-Elimination never returns Fail since we require $\operatorname{rank}(\mathbf{H}_{\mathsf{sk}}[1 : tm, 1 : tm]) = tm$. In the Classic McEliece implementation, this is ensured by repeatedly sampling random $\mathbf{H}_{\mathsf{sk}}$'s, until one obtains $\mathbf{H}_{\mathsf{sk}}$ with $\operatorname{rank}(\mathbf{H}_{\mathsf{sk}}[1 : tm, 1 : tm]) = tm$.

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** ▷ $\text{rank}(\mathbf{H}_j) < tm$
5: **return** FAIL
6: $\mathbf{H}_{j+1}[j, :] := \mathbf{H}_{j+1}[j, :] + \mathbf{H}_{j+1}[k, :]$ ▷ add k-th to j-th row
7: **for** $i = 1, \ldots, tm$, $j \neq i$ **do**
8: **if** $\mathbf{H}_{j+1}[i, j] = 1$ **then** ▷ ensure $\mathbf{H}_{j+1}[i, j] = 0$ for $i \neq j$
9: $\mathbf{H}_{j+1}[i, :] := \mathbf{H}_{j+1}[i, :] + \mathbf{H}_{j+1}[j, :]$ ▷ add j-th to i-th row
10: **return** $\mathbf{H}_{j+1}$

Algorithm 2 GAUSSIAN-ELIMINATION

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

Our Attack Vector. ELIMINATE-COLUMN's row addition in Line 9 of Algorithm 1 is triggered by the if-statement in Line 8. This if-statement is our attack vector.[3] We assume that we have access to a noisy side channel, which allows us to monitor whether Line 9 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 our side channel is noisy, we define our *leak matrix* as an erroneous version of the execution matrix (see Definition 1 below). Notice 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 erroneous 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 $\tau \leq \frac{1}{2}$:

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

[3] In better protected real-world implementations, Line 8 is of course not a simple if-statement. However, as we show in Section 6, our attack also applies to such implementations.

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 $\mathbf{L}$ a $\mathcal{B}(\tau)$-*disturbed* version of the execution matrix $\mathbf{E}$ (thereby ignoring the diagonal issue).

Note that for $\tau = \frac{1}{2}$, $\mathbf{L}$ is uniformly random and does not provide any information. Of course, our simplified Gaussian-Elimination is not protected at all against leakage of Eliminate-Column's operation in Line 9. However, as we show in Section 6, state-of-the-art implementations also do not provide sufficient leakage resistance. In particular, we show that they also reveal a leak matrix $\mathbf{L}$ via side channels.

4 Our Attack: Decoding the Leak Matrix

For ease of notation, we first describe our new attack in terms of our simplified Gaussian elimination algorithm (Algorithm 2). Importantly, as we show in Section 6, the attack, also applies to real world implementations of Gaussian elimination, as in Classic McEliece and in Botan.

High-level Idea. Let us first sketch the high-level idea behind our attack.

Suppose we obtain a leak matrix $\mathbf{L}$, corresponding to Gaussian elimination on a McEliece secret key $\mathbf{H}_{\mathsf{sk}}$. To recover the secret key $\mathbf{H}_{\mathsf{sk}}$, we start by successively recovering all columns of the execution matrix $\mathbf{E} \in \mathbb{F}_2^{tm \times tm}$ (Definition 1). Along the way, we also recover the *transformation matrices* $\mathbf{S}_j$, as defined below.

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

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

corresponding to the elementary row operations of Eliminate-column$(\mathbf{H}_j, j)$, where the $\mathbf{H}_j$'s are the matrices produced by Gaussian-Elimination (Algorithm 2).

Given the j-th column $\mathbf{E}[:,j] = \mathbf{H}_j[:,j]$ (see Definition 1) of the execution matrix $\mathbf{E}$, we efficiently obtain the j-th transformation matrix $\mathbf{S}_j$ as depicted in Algorithm 3.

Algorithm 3 Recover-Transformation-Matrix

Input: $\mathbf{H}_j \in \mathbb{F}_2^{tm \times n}$
Output: Corresponding transformation matrix $\mathbf{S}_j \in \mathbb{F}_2^{tm \times tm}$,

1: Pick an arbitrary matrix $\mathbf{M} \in \mathbb{F}_2^{tm \times n}$, whose j-th column is identical to $\mathbf{H}_j[:,j]$.
2: Run Eliminate-column$(\mathbf{M}, j)$, and monitor all its elementary row operations.
3: Apply the exact same row operations to $\mathbf{I}_{tm}$ and output the resulting matrix.

Given all $\mathbf{S}_j$'s, we finally recover the secret key from the transformation matrices using the following lemma.

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

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

Proof. The lemma follows immediately from Definitions 1 and 2. □

Since $\mathbf{H}_{tm+1} = \mathbf{H}_{\mathsf{pk}}$ (see Algorithm 2), Lemma 1 shows that knowledge of the transformation matrices then allows us to easily recover the secret key via

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

Recovery of $\mathbf{E}[:,1]$. Let us now detail how to recover the execution matrix columns $\mathbf{E}[:,j]$ from the leak matrix $\mathbf{L}$. We begin with recovery of the first column $\mathbf{E}[:,1]$. To this end, we compute a code

$$C_1 := \left\{ \sigma\left(\frac{1}{b}(1, a, a^2, \ldots, a^{t-1})\right) \mid a, b \in \mathbb{F}_{2^m}, b \neq 0 \right\} \subseteq \mathbb{F}_2^{tm}. \tag{8}$$

Recall that $\sigma : \mathbb{F}_{2^m} \to \mathbb{F}_2^m$ is the canonical vector space embedding from Eq. (4), which we apply here coordinate-wise to the vector $\frac{1}{b}(1, a, a^2, \ldots, a^{t-1})$. Furthermore, notice that the code C_1 is not linear, i.e., it is simply a subset of $\mathbb{F}_2^{tm}$.

By Eq. (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 also Lemma 1).

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 Sections 6 and 7 that it performs well in practice.

Code Update and Recovery of $\mathbf{E}[:,2]$. 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

$$\mathbf{E}[:,2] = \mathbf{S}_1 \cdot \mathbf{H}_{\mathsf{sk}}[:,2]. \tag{9}$$

We recover $\mathbf{S}_1$ from $\mathbf{E}[:,1]$ via Algorithm 3, and *update* our code C_1 by multiplying it with $\mathbf{S}_1$. That is, we construct the code

$$C_2 := \mathbf{S}_1 \cdot C_1 = \{\mathbf{S}_1 \cdot \mathbf{c} \mid \mathbf{c} \in C_1\}. \tag{10}$$

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 Eq. (9)) 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 $\mathbf{S}_1, \ldots, \mathbf{S}_{j-1}$ along with the codes $C_1, \ldots, C_{j-1}$, where

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

Using Algorithm 3, we recover the transformation matrix $\mathbf{S}_{j-1}$ from $\mathbf{E}[:,j-1]$. We multiply C_{j-1} with $\mathbf{G}_{j-1}$ to obtain the code

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

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

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

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 Eq. (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 Eq. (10), we first remove all $2^m - 1$ codewords

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

from C_1, and then multiply the resulting code by $\mathbf{S}_1$. 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 *codebooks*

$$\mathsf{CB}_1 := \left\{ \left(\sigma\left(\frac{1}{b}(1, a, a^2, \ldots, a^{t-1})\right), a, b \right) \mid a, b \in \mathbb{F}_{2^m}, b \neq 0 \right\}, \tag{11}$$

$$\mathsf{CB}_j := \{(\mathbf{S}_{j-1} \cdot \mathbf{c}, a, b) \mid (\mathbf{c}, a, b) \in \mathsf{CB}_{j-1}, a \neq \alpha_{j-1}\} \quad j > 1, \tag{12}$$

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 4 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 $\mathbf{L}[:, j]$

```
c_min := (c_1, a_1, b_1)
d_min := Δ(c_1, L[:, j])
for i = 2, ..., |CB| do
    if (Δ(c_i, L[:, j]) < d_min) then
        c_min := (c_i, a_i, b_i)
        d_min := Δ(c_i, L[:, j])
return 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 4.

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 $(\mathbf{c}, a, b)$ with $b \neq g(a)$ from our codebooks. In other words, we define

$$\mathsf{CB}_j := \{(\mathbf{S}_{j-1} \cdot \mathbf{c}, a, b) \mid (\mathbf{c}, a, b) \in \mathsf{CB}_{j-1}, a \neq \alpha_{j-1}, b = g(a)\}, \quad j \geq t+2. \tag{13}$$

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 $|\mathsf{CB}_1| = 2^m(2^m - 1) \approx 2^{2m}$.

Special Case of Known Goppa Points. As shown in Table 1, the parameter set `mceliece 8192128` has $(n, m) = (8192, 13)$. Therefore $2^m = n$, which implies that $\{\alpha_1, \ldots, \alpha_n\} = \mathbb{F}_{2^m}$, 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. We will show in Section 6.3 that 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 5.

Algorithm 5 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 $\mathbf{H}_{\mathsf{sk}} \in \mathbb{F}_2^{tm \times n}$

$\mathsf{CB} := \left\{ \left(\sigma\left(\frac{1}{b}(1, a, a^2, \ldots, a^{t-1}) \right), a, b \right) \mid a, b \in \mathbb{F}_{2^m}, b \neq 0 \right\} \subseteq \mathbb{F}_2^{tm} \times \mathbb{F}_{2^m} \times \mathbb{F}_{2^m}$.
for $j = 1, \ldots, tm$ **do**
  $(\mathbf{H}_j[:, j], \alpha_j, \beta_j) :=$ MaxLikelihood-Decode$(\mathbf{L}[:, j], \mathsf{CB})$
  $\mathbf{S}_j :=$ Recover-Transformation-Matrix$(\mathbf{H}_j[:, j])$.
  $\mathsf{CB} := \{(\mathbf{S}_j \cdot \mathbf{c}, a, b) \mid (\mathbf{c}, a, b) \in \mathsf{CB}, a \neq \alpha_j\}$
  if $j = t+1$ **then**
    Interpolate $g(x) \in \mathbb{F}_{2^m}[x]$ from $(\alpha_1, \beta_1) \ldots, (\alpha_{t+1}, \beta_{t+1})$, where $\beta_i = g(\alpha_i)$.
    if set of Goppa points $\{\alpha_1, \ldots, \alpha_n\}$ known **then**
      Recover $\mathbf{H}_{\mathsf{sk}}$ via Support Splitting. ▷ See Section 2.3.
      return $\mathbf{H}_{\mathsf{sk}}$.
    else
      $\mathsf{CB} := \{(\mathbf{c}, a, b) \in \mathsf{CB} \mid b = g(a)\}$
return $\mathbf{H}_{\mathsf{sk}} := \mathbf{S}_1^{-1} \cdot \mathbf{S}_2^{-1} \cdot \ldots \cdot \mathbf{S}_{tm}^{-1} \cdot \mathbf{H}_{\mathsf{pk}}$

List Decoding. Instead of using MaxLikelihood-Decode, one may also use list-decoding in our attack, i.e., instead of computing only *the* closest codebook element

$(\mathbf{H}_j[:,j], \alpha_j, \beta_j)$ in Line 3 of Algorithm 5, we may instead compute a list L of close codebook elements, that are within in some well-chosen radius r from the leak matrix column $\mathbf{L}[:,j]$.

Importantly, since each element in L gives rise to a different transformation matrix and codebook in Lines 4 and 5, we then have to split the algorithm into $|L|$ branches. While, in the *worst* case, this may lead to exponentially many branches after tm iterations, in the *average* case, list decoding would not increase the runtime much: If we decode to an incorrect $\mathbf{c} \neq (\mathbf{H}_j[:,j], \alpha_j, \beta_j)$ in the j-th iteration, then the resulting *wrong* codebook in the $(j+1)$-th iteration will likely be considerably different from the correct codebook CB_{j+1}. In that case, we can expect the leak matrix column $\mathbf{L}[:\, j+1]$ to be very far from *all* elements in the wrong codebook. Hence, if we decode incorrectly in the j-th iteration, then (list)-decoding in the $(j+1)$-th iteration is expected to fail. In particular, out of $|L|$ branches, only the branch with the correctly decoded $(\mathbf{H}_j[:,j], \alpha_j, \beta_j)$ survives.

However, since we already achieve high success probability for Algorithm 5 with the simple MaxLikelihood-Decode, we chose to not unnecessarily complicate the algorithm and its analysis.

5 Analysis of Success Probability

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 at most. 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.

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

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

Together with Eq. (2), this yields the following asymptotic necessary condition for the correctness of Secret-Key-Recovery:

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

Using $|\mathsf{CB}_1| \approx 2^{2m}$ and $|\mathsf{CB}_1| > |\mathsf{CB}_j|$ for $j \neq 1$, we can also use the simpler necessary condition

$$d_1 \leq \mathrm{H}^{-1}\left(1 - \frac{2}{t}\right) \cdot tm. \tag{14}$$

By Lemma 1, $\mathbf{L}[:,1]$ is a $\mathcal{B}(\tau)$-disturbed version of $\mathbf{E}[:,1]$. Therefore, d_1 essentially follows the Binomial distribution with parameters tm and τ, which has expected value τtm. For

simplicity, let us assume that d_1 achieves its expected value $\mathbb{E}[d_1] = \tau tm$.[4] Then Eq. (14) translates to

$$\tau < \mathrm{H}^{-1}\left(1 - \frac{2}{t}\right). \tag{15}$$

We visualize the upper bound from Eq. (15) in Fig. 1 as a function of t. Fig. 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.

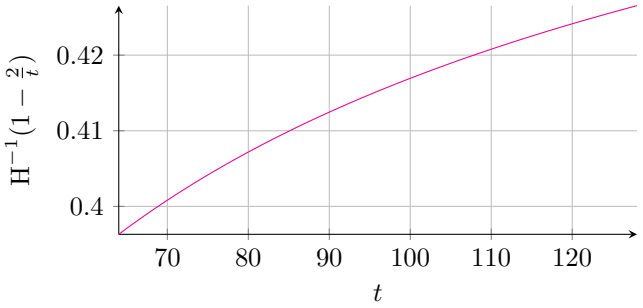


Figure 1: Upper bound from Equation (15) for $64 \leq t \leq 128$.

Growth of τ. From Fig. 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 Eq. (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

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

Since the transformation matrices $\mathbf{S}_i$ 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 $\mathbf{L}$.

5.2 Fine-Grained Analysis

What mainly prevents our attack from reaching the asymptotic upper bound from Eq. (15) 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.

[4] By the Chernoff bound, d_1 is asymptotically very close to its expected value.

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 $\mathbf{c} \in \mathsf{CB}_j \setminus \{\mathbf{E}[:,j]\}$.[5] Conversely, we decode incorrectly only if any $\mathbf{L}[:,j]$ hits a point inside a Hamming ball $B(\mathbf{c}, d_j)$. To analyze the probability of this event, we have to study the distribution of the codebook elements.

While our first codebook CB_1 is somewhat structured (see Eq. (11)), we expect the remaining codebooks CB_j with $j > 1$ to behave quite randomly due to the transformation matrices $\mathbf{S}_{j-1}$ (see Eq. (12)). In particular, as j grows, we expect the distribution of codebook elements to converge to the uniform distribution over $\mathbb{F}_2^{tm}$.

While we cannot hope to *formally* verify this behavior of our codebooks,[6] let us instead, as a sanity check, show that the Hamming weight distribution of our codebook elements is close to that of uniformly random vectors from $\mathbb{F}_2^{tm}$. Let $w \in \mathbb{N}$ with $0 \leq w \leq tm$, and let $X_{j,w}$ denote the number of codebook elements $\mathbf{c} \in \mathsf{CB}_j$ of Hamming weight $\omega(\mathbf{c}) = w$. Since a uniformly random vector $\mathbf{v} \in \mathbb{F}_2^{tm}$ has Hamming weight w with probability $2^{-tm}\binom{tm}{w}$, we have to show that, for every w, it holds that $X_{j,w} \approx 2^{-tm}\binom{tm}{w}|\mathsf{CB}_j|$.

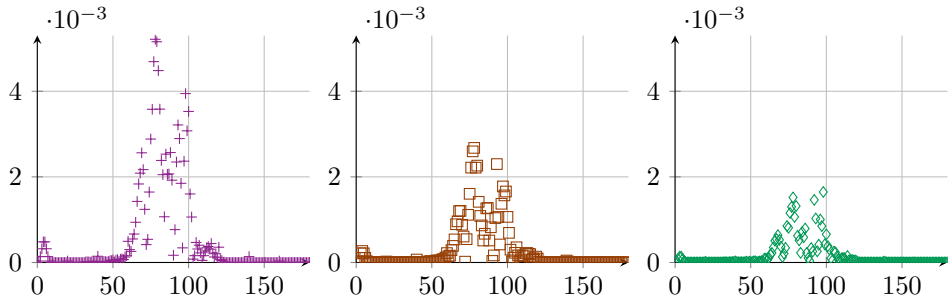


Figure 2: Distribution of $\frac{1}{|\mathsf{CB}_j|}\left|X_{j,w} - 2^{-tm}\binom{tm}{w}|\mathsf{CB}_j|\right|$ as a function in w, for $j = 1, 2, 3$ (from left to right) for one run of the attack.

In Fig. 3, we plot the normalized distances $\frac{1}{|\mathsf{CB}_j|}\left|X_{j,w} - 2^{-tm}\binom{tm}{w}|\mathsf{CB}_j|\right|$ for $j = 1, 2, 3$ and $0 \leq w \leq tm = 180$ in the `toyeliece51220` parameter set $(n = 512, t = 20, m = 9)$. As Fig. 2 shows, the distances quickly converge to zero, showing that the Hamming weight distribution of our codebook elements quickly converges to that of uniformly random vectors from $\mathbb{F}_2^{tm}$. Thus, we may safely model the distribution of our codebook elements using the following 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 distributed uniformly at random in $\mathbb{F}_2^{tm}$.*

Using Assumption 1, we now prove the following lemma.

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

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

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

$$p(d_j) = \left(1 - \frac{V^{tm}(d_j)}{2^{tm}}\right)^{|\mathsf{CB}_j \setminus \{\mathbf{E}[:,j]\}|}. \tag{16}$$

[5] 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 $\mathbf{c}$.

[6] If we wanted to show that empirical distribution of codebook elements is close to the uniform distribution over $\mathbb{F}_2^{tm}$, we would have to sample significantly more than 2^{tm} codebook elements. By Table 1, all parameter sets have $2^{tm} \geq 2^{180}$.

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 further verify the validity of Assumption 1 experimentally in Section 7 by showing that the actual success probability closely matches Lemma 2.

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

$$\Pr[\textsc{Success}] := \prod_{j=1}^{tm} \sum_{d_j=0}^{tm} p(d_j) \cdot \Pr[\Delta(\mathbf{L}[:,j], \mathbf{E}[:,j]) = d_j]. \tag{17}$$

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

$$\Pr[\Delta(\mathbf{L}[:,j], \mathbf{E}[:,j]) = d] = \binom{tm}{d} \cdot \tau^d \cdot (1-\tau)^{tm-d} \tag{18}$$

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

Using Lemma 2 together with Eqs. (17) and (18), 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 Eq. (13)). Therefore, we obtain

$$\begin{aligned}\Pr[\textsc{Success}] > &\left(\sum_{d_j=0}^{tm} \left(1 - \frac{V^{tm}(d_j)}{2^{tm}}\right)^{2^{2m}} \cdot \binom{tm}{d_j} \cdot \tau^{d_j} \cdot (1-\tau)^{tm-d_j}\right)^{t+1} \cdot \\ &\left(\sum_{d_j=0}^{tm} \left(1 - \frac{V^{tm}(d_j)}{2^{tm}}\right)^{2^{m}} \cdot \binom{tm}{d_j} \cdot \tau^{d_j} \cdot (1-\tau)^{tm-d_j}\right)^{tm-(t+1)}.\end{aligned} \tag{19}$$

For the special case of known Goppa points, we stop decoding after $t+1$ iterations in Secret-Key-Recovery. Therefore, we only need the first factor from Eq. (19), i.e.,

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

Fig. 3 shows Pr[Success] as a function of τ for concrete parameters. As the figure shows, if τ is sufficiently small, the success probability is very close to 1. Conversely, if τ is too large, the success probability is very close to 0. Notably, the transition between these two regimes is rather abrupt. This phenomenon can be explained as follows: Let τ^* be the error rate τ for which Pr[Success] equals $\frac{1}{2}$. Then τ^* can be seen as the crossover point between our two regimes. As Table 2 shows, this crossover point matches the asymptotic upper bound $\tau < \mathrm{H}^{-1}(1 - \frac{2}{t})$ from Eq. (15) quite accurately. If τ exceeds this upper bound, then decoding becomes information-theoretically impossible. However, as soon as τ is slightly below the bound, then the high redundancy of $1 - \frac{2}{t}$ per bit makes decoding easy, and we quickly achieve success probability 1.

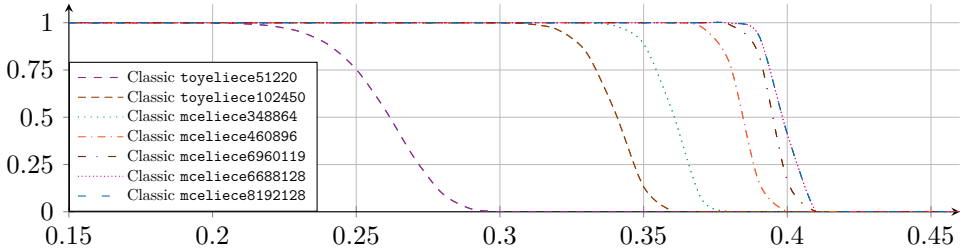


Figure 3: Success probability predicted by Eq. (19) for various parameter sets. The horizontal axis shows the error probability τ of the leak. The vertical axis shows the success probability of our attack.

Table 2: Largest error rate τ for which Eq. (19) gives success probability at least $\frac{1}{2}$ compared to the upper bound from Eq. (15).

Name	Eq. (19)	Eq. (15)	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 Attacking Real World Implementations

We investigate two concrete implementations of McEliece:

1. The reference implementation of *Classic McEliece* [BCC+22a], a 4th-round submission to the NIST's Post-Quantum Cryptography Standardization Project [nis].

2. The cryptographic library Botan [cod], which has been recognized by the German Federal Institute for Information Security as a secure implementation [bot].

Since the exact operations of Gaussian elimination during public-key generation in both implementations slightly differ from our high-level description in Algorithm 2, we describe both of them in the following in detail. We discuss the potential for leakages in our attacker model. As a proof of concept, we use differential power analysis on the Gaussian Elimination of Classic McEliece running on an STM32L592 ARM processor implementation, recovering the secret key in an end-to-end attack.

6.1 Classic McEliece

Algorithm 6 outlines the operation of Gaussian elimination in the reference implementation of Classic McEliece [BCC+22a].

Algorithm 6 Gaussian Elimination from Classic McEliece Key Generation

Input: $\mathbf{H}_p \in \mathbb{F}_{2^8}^{\lceil tm/8 \rceil \times n}$
Output: $(\mathbf{I}_{tm} | \mathbf{A})$ or FAIL

1: **for** $p = 1, \ldots, \lceil tm/8 \rceil$ **do**
2: **for** $b = 0, \ldots, 7$ **do**
3: $j := 8(p-1) + b + 1$
4: **for** $i = j+1, \ldots, tm$ **do** ▷ diagonal stage
5: mask $:= (\mathbf{H}_p[i,p] + \mathbf{H}_p[j,p]) \gg b$
6: mask $:=$ mask $\& 1$
7: mask $:= -$mask
8: **for** $c = 1, \ldots, n/8$ **do**
9: $\mathbf{H}_p[j,c] := \mathbf{H}_p[j,c] +$ mask $\& \mathbf{H}_p[i,c]$
10: **if** $\mathbf{H}[j,j] \neq 1$ **then**
11: **return** FAIL ▷ cannot bring $\mathbf{H}$ into systematic form
12: **for** $i = 1, \ldots, tm$ **do** ▷ zero stage
13: **if** $i = j$ **then**
14: **continue**
15: mask $:= \mathbf{H}[i,j] \gg b$
16: mask $:=$ mask $\& 1$
17: mask $= -$mask ▷ potential leak: mask is `0x00` or `0xff`
18: **for** $c = 1, \ldots, n/8$ **do**
19: $\mathbf{H}_p[i,c] := \mathbf{H}_p[i,c] +$ mask $\& \mathbf{H}_p[j,c]$

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 4–9), is to ensure $\mathbf{H}[j,j] = 1$. The second *zero stage* (Lines 12–19), 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, Classic McEliece does more row additions than our naive Algorithm 2.

Potential Leak Analysis. The Classic McEliece implementation uses programming techniques to defend against side-channel leakage based on observing execution times or memory cache access patterns. In particular, it contains no secret-dependent branches

and no secret-dependent memory access patterns. This is achieved by always executing the exact same sequence of row additions, irrespective of the values of $\mathbf{H}$. To correctly implement the Gaussian elimination algorithm, these row additions involve a mask value that effectively neutralizes some of the row additions, turning them into no-operations (no-op). For the diagonal stage, there are two modes: If there is a zero on the diagonal of the current column, row additions are a no-op until the row contains a one in the current column. After that, there definitely is a one in the diagonal entry, and the mask is inverted, so row addition becomes a no-op for all subsequent rows that contain a one in the current column. For the zero stage, there is only one mode. Here, the row addition is a no-op if the target row already contains a zero, otherwise the current row (which definitely has a one in the current column due to the diagonal stage) is added to ensure a zero value in the current column.

Although important side-channel leakages are mitigated by this approach, some leakage through side channels not based on timing or cache access measurements remains possible. The Classic McEliece implementation speeds up the row operations by applying them at a byte resolution. To this end, it extends the single-bit mask to eight bits by computing its inverse modulo 2^8 (Line 17). Consequently, the resulting values of the mask have very different Hamming weights. The power consumption and the electromagnetic emanations observed during program execution correlate with the Hamming weight of the data that the program processes. Thus, computing the mask and using it has observable impact on the circuit's power consumption. In the zero stage, the algorithm calculates a mask value based on $\mathbf{H}[i, j]$. Hence, (erroneous) leaks of mask directly correspond to our simplified leak matrix $\mathbf{L}$ from Section 3.

Leaking the Zero Stage is Sufficient. In this work, we only consider leaks from the mask in the zero stage, i.e. Line 17, which corresponds to all bits of $\mathbf{H}$ except those on the diagonal. An observation of a potential leak in the diagonal stage in Line 5 could also leak information on the diagonal of $\mathbf{H}$. However, as the mask value does not directly correspond to a single bit of $\mathbf{H}$, using this information in the attack introduces significant complications. In any case, our definition of the leak matrix $\mathbf{L}$ (see Definition 1) does not require knowledge of the diagonal entries.

6.2 Classic McEliece Leakage Verification and Attack

We verify that we can indeed detect power leakage during the Gaussian elimination of Classic McEliece, and thereby obtain erroneous leaks of mask from Algorithm 6. In our experiment, we use ChipWhisperer-Husky connecting to CW308 UFO with CW308T-STM32L5HWC as a target board. We perform a full secret key recovery for the strongest parameter set `mceliece8192`, so $t = 128, m = 13, n = 8192$. We only ran the Gaussian Elimination on the first $n' := t + 1 = 129$ columns, since this is sufficient for a full key recovery using the Support Splitting algorithm in the case of `mceliece8192`, see Section 2.3.[7]

As Classic McEliece is constant-time, all traces of power consumption during Gaussian Elimination are of the same length and aligned automatically by the capturing device. One execution takes 69 seconds and results in $S = 287002932$ 12-bit samples per trace (which is also the number of CPU clock cycles) resulting in 400 MB of raw data per trace. We collected 3000 traces in 2.5 days, generating 1.2 TB of data.

Points of Interest. We assume that the attacker has access to a device identical to the victim device to identify the points of interest (PoIs) using the *Fixed-vs-Random Data*

[7] To this end, we slightly altered the keygen code such that it halts after $t + 1$ columns, by changing a single constant in the code. In practice, an attacker would simply stop measurements after $t + 1$ columns.

test of a test vector leakage assessment (TVLA) procedure [SM15]. (Rather than using Signal-to-Noise ratio analysis for PoI identification [MOP07], we use TVLA, because TVLA corresponds to the difference of the means, which we later use for classifying the PoIs.) We expect a set P_d of PoIs corresponding to the calculation of mask values in Line 5 and a set P_z corresponding to Line 17 in Algorithm 6. We observe that the outer loop is over the n' columns, and that for column j we have two inner loops. The first calculates $tm - j$ masks for the diagonal stage, and the second calculates $tm - 1$ masks for the zero stage. Furthermore, the zero stage loop has a single short iteration when $i = j$ (cf. Line 14). Thus, for each column j, we expect a sequence of $tm - j$ of equally spaced PoIs with distance T_d, followed by a sequence of $i - 1$ equally spaced PoIs with distance T_z, followed by a short gap of length T_{skip}, followed again by a sequence of $tm - j$ equally spaced PoIs with distance T_z. In total, we expect $|P_d| = \sum_{i=tm-n'}^{tm-1} i = 206271$ PoIs in the diagonal stages and $|P_z| = (tm - 1) \cdot n' = 214527$ in the zero stages, resulting in $P = |P_d| + |P_z| = 420798$ mask-related PoIs. Of those, we are mainly interested in the PoIs P_z of the zero stages, as these form the non-diagonal elements of the leak matrix L.

Candidates for Points of Interest. To identify the PoIs, we collect $N = 1000$ traces $R_1, \ldots, R_{1000}$ of power consumption during the Gaussian elimination of random matrices and compute the per-point average $\bar{R}$ and sample standard deviation σ_R. We then collect N traces $A_1, \ldots, A_{1000}$ while performing Gaussian elimination of a fixed matrix A and compute the per-point average $\bar{A}$ and sample standard deviation σ_A of these. To compare these values, we calculate (point-wise) the t-statistic trace $t = (\bar{R} - \bar{A})/\sqrt{(\sigma_A^2 + \sigma_B^2)/N}$. Points with high t-value indicate PoIs, where power consumption differs significantly depending on the bits of the secret key. Due to our large sample size, we have to carefully control false positives. We observe a maximal t-value of 29.24. To allow for up to 10% false positives, we choose a threshold value of 13.34 that results in exactly $1.1 \cdot P$ PoI candidates, according to the empirical distribution. We take the strongest PoIs with a clearance of 10 samples to either side, resulting in 457273 PoIs.

Cycle Count of Loop Iterations. To identify the cycle times T_d and T_z of the diagonal and zero stage loop bodies, we use Fast Fourier Transformation on the initial 1/16th samples of t (roughly corresponding to processing of eight columns). Using a band-pass filter around the expected period $T_{approx} = S/P \approx 682$, plus/minus 25%, we identify the shorter loop with $T_d = 630$ cycles and the longer loop with $T_z = 732$ cycles (see Fig. 4).

Alignment with Regression-Based Model of P_z. To find the PoIs of the zero stage, we look for sequences of PoIs with distance T_z and minimum length $l := 7$, allowing for single missing PoIs within the sequence (see Fig. 5). This identifies almost all iterations of the zero stage, except for 13 missing points due to false negatives, and $(l + 1) \cdot l/2 = 21$ missing points with $j < l + 1$ and $i < j$, which do not form sufficiently long sequences to be identified by this method.

We can use two sequences of PoIs corresponding to the zero stage of a single column operation to easily identify the gap T_{skip} for the loop iteration $i = j$ in Line 14. To arrive at T_{skip}, we take one sequence before the gap (i.e., for some $i < j$), another sequence after the gap ($i > j$), calculate the differences of the means modulo T_z of these sequences, and round the result to the nearest integer. We find that $T_{skip} = 33$ for our experiments.

Now we can fit a model for *all* PoIs of the zero stage, that describes the position $P_z(i, j)$ of a PoI in P_z corresponding to $\mathbf{L}[i, j], i \neq j$. This model allows us to fill in the PoIs that are missing due to gaps or short sequences. Using a quadratic regression on the start of each column (because in each column, the diagonal stage has one less iteration), corrected by a linear regression on the start of each column within an eight-bit group packed in a

single byte, we get for $i \neq j$

$$\begin{aligned} P_z(i,j) &= \text{Column}(j) + \text{Bitshift}(j \bmod 8) + \text{Row}(i) + \text{Skip}(i,j) \\ &= (-315j^2 + 2264764.75j + 1047823) - 5.75 \cdot (j \bmod 8) + T_z \cdot i + \text{Skip}(i,j), \end{aligned}$$

where $\text{Skip}(i,j)$ is 0 if $i < j$ and else it is $-T_z + T_{skip}$. We note that in our experiments, all identified PoIs in P_z correspond perfectly with this model. From here on out, we always consider the full set P_z, with gaps filled in as predicted by the model, discarding P_d and any false positives.

Bit Classification. To identify if a PoI corresponds to a bit 0 or 1, we use a simple difference-of-means classifier using the point-wise averages of the random matrices as a threshold value. Using the random matrices as ground truth, we verify that for each PoI the classes for bits 0 and 1 are well isolated. The results are shown in Fig. 6 For each PoI, the class for bit 0 (resp. bit 1) is equally sized with 499.9555 ± 15.8433 (resp. 500.0445 ± 15.8433), with mean -0.0847 ± 0.0015 (resp. -0.1176 ± 0.0014) and standard deviation 0.0133 ± 0.0007. The difference of means is 0.0329, which is smaller than the sum of the standard deviations 0.0266, resulting in a z-value of 1.2342, which indicates an estimated error rate of 0.1085. We can see that all PoIs are independent and uniform. The signal-to-noise ratio for the measurement at our PoIs is 17.7394 ± 0.4332 dB.

Validation. We then repeat the complete analysis with $N = 1,000$ measurements of a different fixed matrix B, getting similar results. In this case, the maximum t-value is 30.47, the threshold value is 13.71, identifying 452143 PoIs, and resulting in 16 gaps within the sequences. The fitted model is identical, and all identified PoIs correspond perfectly with that model. This validates our results.

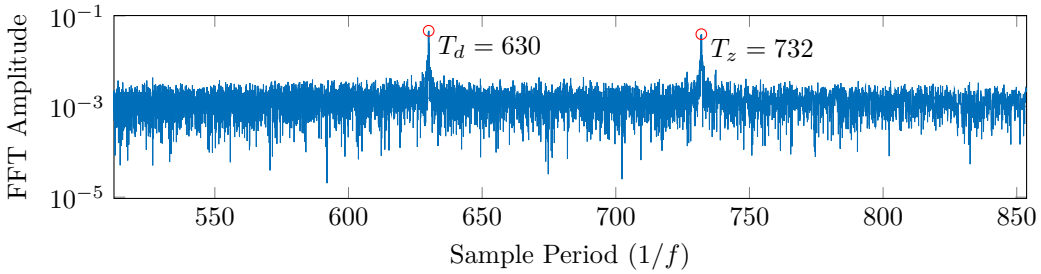


Figure 4: FFT amplitude over $1/f$ for the Welsh t-statistic, showing long periods of PoIs close to the approximate period $S/(|P_d| + |P_z|)$. The two peaks are located at the cycle counts T_d and T_z for the loop iterations of the diagonal and zero stage.

Single Trace Attack and Experimental Error τ. To perform the attack, we capture a single trace from a victim running Gaussian elimination on the first n' columns. We then compare the collected trace against the average value at each PoI, and guess the mask based on whether the value is larger or smaller than the average. Fig. 7 shows two examples of power traces around PoI $P_z(1,0)$, one for each value of mask. Comparing the guesses to the ground truth, we find that we can correctly determine 89.35% of the 214527 mask values on average, corresponding to an error value $\tau \approx 0.1065$ – which is well within our recovery bounds determined in the previous Section 5. (Using simulated leakage, we show in the following Section 7 that our attack also works well with error rates close to the theoretical maximum.) The signal-to-noise ratio for this victim trace across all PoIs is 17.8218 dB. Filling in the diagonal with random values, we get a leak matrix $\mathbf{L}$ as input

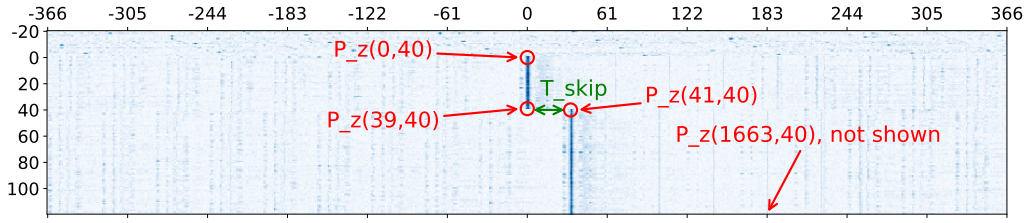


Figure 5: The t-statistics around the PoI $P_z(0, 40)$, given here at the origin. Values are plotted in reading order (left-to-right, top-to-bottom). Darker colors correspond to higher t-values, with T_z values in each row. Only the first 120 of all $tm - 1 = 1663$ PoIs of the zero stage of column 40 are shown.

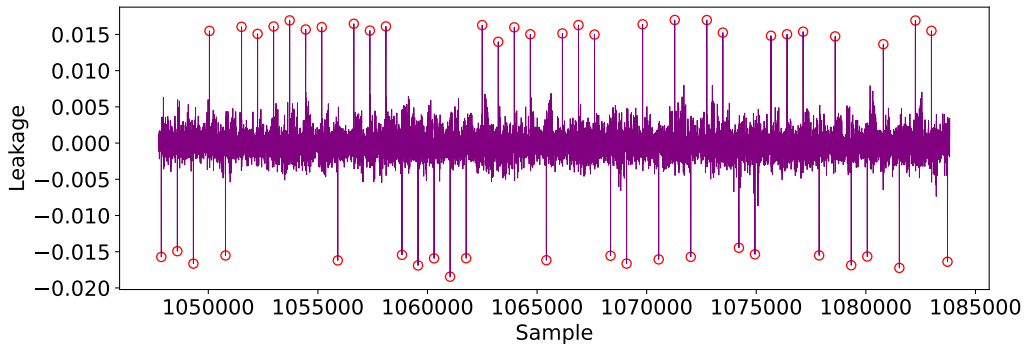


Figure 6: Difference between the point-wise averages of the fixed and random traces, highlighting the first 50 PoIs of the zero stage of the first column.

exactly as in our simplified model. Applying our attack from Section 4, we subsequently recover from $\mathbf{L}$ the complete secret key.

We repeated this attack ten times, with a success rate of 1. The mean error rate yields an *experimental error rate* of

$$\tau = 0.1080 \pm 0.0011,$$

and a mean SNR $17.7684 \pm 0.0323\,\mathrm{dB}$.

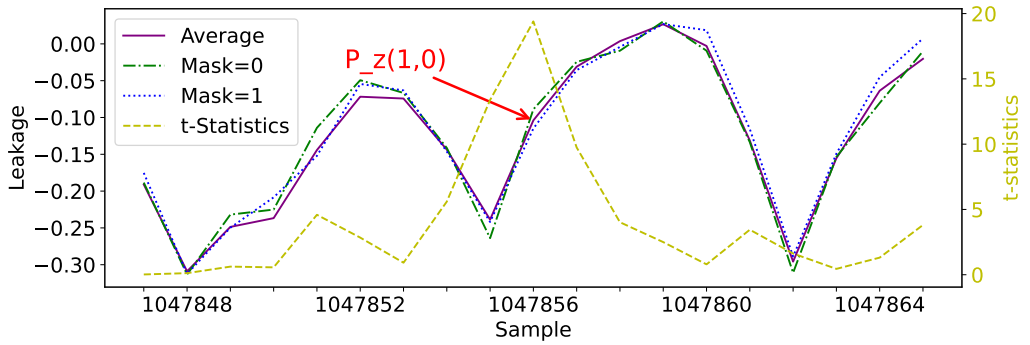


Figure 7: Close up around the PoI $P_z(1, 0)$, showing that the average of random traces separates two single traces, one with $\mathsf{mask} = 0$ and one with $\mathsf{mask} = 1$ at that PoI. The t-values are also shown for comparison.

Possible Leakage Vector. To investigate possible sources for the leak, we compiled the Gaussian elimination from Classic McEliece on the evaluated target CPU[8] with the GNU C Compiler (GCC), version 10.3.1, and get the following assembler code (regardless of McEliece parameter choices) corresponding to lines 15–17 of Algorithm 6:

```
ldrb r1, [r6, r3]        ▷ Load H[i, j] into r1.
asrs r1, r0                    ▷ r1 := r1 ≫ b
sbfx r1, r1, #0, #1          ▷ See main text.
uxtb r1, r1                ▷ r1 := r1 & 0xff
```

The critical instruction is `sbfx`, which stands for *Signed Bitfield Extension*. The way it is used here has the effect that the least significant bit of `r1` is copied into the remaining 31 bits of the register, i.e. if bit b of $\mathbf{H}[i, j]$ is 0, then `r1` will be `0x00000000`, else `r1` will be `0xffffffff` after executing the `sbfx` instruction. This leads to a big difference between the Hamming weights of the two possible states of the register, and is known to be susceptible to differential power analysis [EPMS23]. We verified that the `sbfx` instruction is emitted regardless of the optimization level (`s`, `fast`, `0`, `1`, `2`, `3`). The `sbfx` instruction is specific to the ARM A32 and T32 architectures.

6.3 Botan

Algorithm 7 outlines the operation of Gaussian elimination for McEliece key generation in the cryptographic library Botan version 3.1.1 [cod].

Algorithm 7 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                                        ▷ diagonal stage
6:         if H[i, c] = 1 then
7:             if i ≠ j then
8:                 H[j, :] := H[j, :] + H[i, :]
9:             break
10:    if H[j, j] ≠ 1 then                                          ▷ swap stage
11:        failcount := failcount + 1
12:        if failcount = n − tm then
13:            return Fail                  ▷ can not bring H into systematic form
14:        else
15:            π[n − tm + 1 − failcount] := c      ▷ c unsuitable, move to A part of H
16:            c := c − 1
17:            goto line 5
18:    π[n − tm + j] := c                  ▷ c suitable, move to I_tm part of H
19:    for i = j + 1, j + 2, ..., tm, j − 1, j − 2, ..., 1 do       ▷ zero stage
20:        if H[i, c] = 1 then                          ▷ potential leak: H[i, c] = 1
21:            H[i, :] := H[i, :] + H[j, :]
22:    c := c − 1
23: return (H, π)
```

[8] Relevant compiler options: -mcpu=cortex-m33 -mthumb -Os -funsigned-char -funsigned-bitfields -fshort-enums

Botan's implementation of Gaussian elimination is also slightly different from the high-level description in Algorithm 2. Although we can identify the two stages, diagonal stage (Lines 5–9) and zero stage (Lines 19–21), there is also a *swap stage* (Lines 10–17). 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 $\mathsf{failcount}$ 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 model.

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.

Importantly, the most crucial step – the secret-dependent conditional branch, deciding whether two rows get added – is identical to our simplified leak matrix model from Section 3.

Potential Leak Analysis. Our investigation of Gaussian elimination in Botan reveals that the implementation contains conditional branches, depending on secret data. This shows that the Botan implementation is vulnerable to side-chanel attacks that leak control flow and memory access patterns.

Unlike the Classic McEliece implementation that ensures the same number of iterations for different row additions, 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 met. To be more precise, there is a leak whether $\mathbf{H}[i,c] = 1$ in Line 20 of Algorithm 7 — completely analogous to our simplified leak matrix $\mathbf{L}$ 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 these values at random, Botan chooses the 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. As we have shown in Section 4, this additional information can be used to improve our attack via the Support Splitting Algorithm.

7 Performance for Other Parameter Sets with Simulated Leakage

7.1 Key Recovery

In addition to the experimental validation in Section 6.2, we use simulated leakage experiments to verify our analysis for a broader range of McEliece parameter sets.

Implementation. 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.

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: $(s_2, \tau) \mapsto \mathbf{e}$. An 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 success probability is plotted as a function of the error rate τ in Fig. 8.

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 table also presents the average execution time.

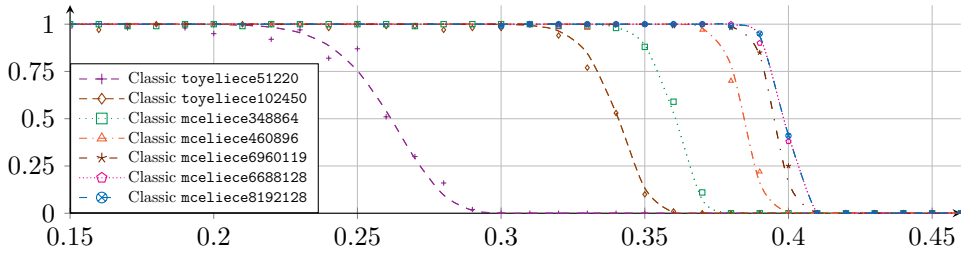
Table 3: Experiment execution time and the threshold error rate $\tau_{\mathsf{Threshold}}$, where the success probability crosses $\frac{1}{2}$, based on linear interpolation of neighbouring measurements.

	Classic McEliece		Botan	
Name	Elapsed Real Time	$\tau_{\mathsf{Threshold}}$	Elapsed Real 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

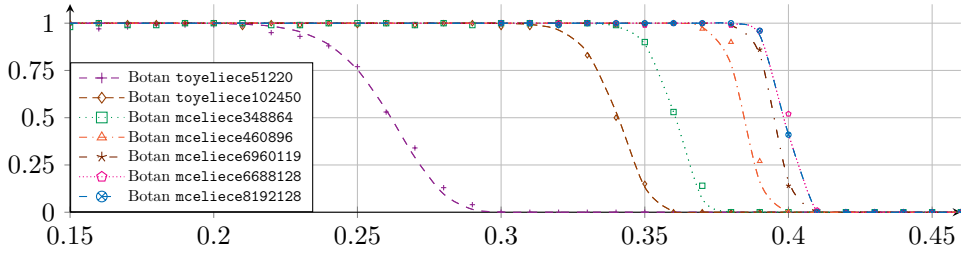
Results. Our experimental verification shows that the success probability of our attack on Classic McEliece and Botan is very well aligned with our model, as shown in Fig. 8.

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 in Botan (due to the deterministic Goppa point generation). In practice, this makes our attack on Botan 25%–50% faster compared to that of Classic McEliece.

Effect of Codebook Reduction on Decoding Time. We observed that the decoding time for each column decreases only marginally for the first $t + 1$ columns, and becomes much quicker after that due to the big codebook reduction based on the Goppy polynomial. Recovering the remaining columns is, in comparison, almost instantaneous. This shows that the codebook reduction after $t + 1$ columns is a significant optimization, while the other reductions provide only a marginal advantage.



(a) Classic McEliece



(b) Botan

Figure 8: Comparison between experimental results (dots) and estimates from Eq. (19) (curves) for our attack on Classic McEliece and Botan parameter sets. Horizontal axes show the error probability τ of the leak. Vertical axes show the success probability.

7.2 Support Splitting

In a setting where the Goppa points 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.

Since the runtime analysis in Sendrier's original paper [Sen00] is only heuristic, it is important to verify that SSA is indeed efficient for McEliece parameters. To this end, we implement SSA in SageMath and run it on various parameter sets. Our experiments show that the algorithm is highly efficient: Given all Goppa points, Support Splitting recovers `toyeliece51220` keys in less than a second, `mceliece348864` keys in roughly half a minute, and even high-security `mceliece8192128` keys in less than 5 minutes.

8 Countermeasures

We now discuss potential countermeasures for our attack. Recall that the running time of our attack is dominated by the size of the initial codebook, which is approximately 2^{2m}. Even though this is exponential in m, our attack is highly efficient for the suggested McEliece parameter sets, since they use small $m \leq 13$. To defend McEliece against our attack, one might consider instantiating the crypto system with larger m. However, altering the system parameters of McEliece requires great caution. Indeed, if one increases only m, but keeps the parameters n and t constant, then the overall security of the cryptographic system decreases significantly. Thus, if one wants to increase m, then n and t have to be increased as well. This results, however, in significantly increased public keys. Since key sizes are already a major concern for McEliece's practicality, this means that increasing m is not a viable option in practice.

As a more efficient counter measure, one may instead apply a random change of basis to

the secret key *before* computing the public key via Gaussian elimination.[9] More concretely, one may pick a random invertible matrix $\mathbf{S}$ and then compute the public key by applying Gaussian elimination to $\mathbf{S} \cdot \mathbf{H}_{\mathsf{sk}}$. Since Gaussian elimination is invariant under invertible transformations from the left, the public key remains the same. The benefit of incorporating this change of basis is that $\mathbf{S}$ then *masks* the Goppa structure of $\mathbf{H}_{\mathsf{sk}}$. This prevents the attacker from computing the codebooks CB_j, thereby effectively protecting against our attack. It is, however, crucial that neither the generation of $\mathbf{S}$ nor the multiplication $\mathbf{S} \cdot \mathbf{H}_{\mathsf{sk}}$ leak, as otheriwse the attacker could mount a similar codebook-based attack.

A generic approach to protect against side-channel attacks is masking [CJRR99], which combines secret values with random masks, such that secrets are never explicitly stored or processed at any given time. Masking can be applied at the hardware implementations [NRR06,BGN+14] or to software [ISW03,RP10]. The added computation required for ensuring secure masking incurs a significant overhead. For example, Chen et al. [CEvMS16] present a threshold implementation [NRR06] of McEliece, which incurs an overhead of almost an order of magnitude in the size of the circuit, and a decrease of a factor of four in the maximum circuit speed. Similarly, reported slowdowns for software implementations of cryptography are by almost an order of magnitude [Wea21]. Although we are not aware of masked implementations of McEliece in software, we anticipate such implementations will suffer similar performance hits. Another limitation of masking is that its effectiveness is somewhat limited. In particular, first-order masking, where each secret value is protected with a single random mask, offers only a limited protection. Recent works have demonstrated effective attacks using machine learning [PPM+23]. The effectiveness of the defense may also be limited due to unexpected interactions between values within the processor's microarchitecture [SSB+21,BGG+14].

Acknowledgements

We would like to thank Colin O'Flynn and NewAE Technology Inc. for help in sourcing the hardware for the experiments.

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

References

[ABG10] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2010.

[ADP18] Martin R. Albrecht, Amit Deo, and Kenneth G. Paterson. Cold boot attacks on ring and module LWE keys under the NTT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):173–213, 2018.

[AGS07] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMACC*, volume 4887 of *Lecture Notes in Computer Science*, pages 185–203. Springer, 2007.

[BCC+22a] Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai,

[9] We thank the anonymous TCHES reviewers for this suggestion.

Martin Tomlinson, and Wen Wang. Classic McEliece. https://classic.mceliece.org/mceliece-sage-20221023.tar.gz, 2022.

[BCC+22b] Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece: conservative code-based cryptography: design rationale, 2022.

[BGG+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *CARDIS*, pages 64–81, 2014.

[BGN+14] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Higher-order threshold implementations. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 326–343. Springer, Berlin, Heidelberg, December 2014.

[bot] BSI-Projekt: Entwicklung einer sicheren Kryptobibliothek. https://www.bsi.bund.de/dok/9060550.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *ESORICS*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371. Springer, 2011.

[CEvMS16] Cong Chen, Thomas Eisenbarth, Ingo von Maurich, and Rainer Steinwandt. Masking large keys in hardware: A masked implementation of McEliece. In Orr Dunkelman and Liam Keliher, editors, *SAC 2015*, volume 9566 of *LNCS*, pages 293–309. Springer, Cham, August 2016.

[CFSY22] Chitchanok Chuengsatiansup, Andrew Feutrill, Rui Qi Sim, and Yuval Yarom. RSA key recovery from digit equivalence information. In *ACNS*, volume 13269 of *Lecture Notes in Computer Science*, pages 193–211. Springer, 2022.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Berlin, Heidelberg, August 1999.

[cod] 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.

[Cop96] Don Coppersmith. Finding a small root of a univariate modular equation. In *EUROCRYPT*, volume 1070 of *Lecture Notes in Computer Science*, pages 155–165. Springer, 1996.

[Cov99] Thomas M Cover. *Elements of information theory*. John Wiley & Sons, 1999.

[DMH20] Gabrielle De Micheli and Nadia Heninger. Recovering cryptographic keys from partial information, by example. IACR ePrint 2020/1506, 2020.

[EMVW22] Andre Esser, Alexander May, Javier A. Verbel, and Weiqiang Wen. Partial key exposure attacks on BIKE, rainbow and NTRU. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part III*, volume 13509 of *LNCS*, pages 346–375. Springer, Cham, August 2022.

[EPMS23] Ferhat Erata, Ruzica Piskac, Víctor Mateu, and Jakub Szefer. Towards automated detection of single-trace side-channel vulnerabilities in constant-time cryptographic code. In *EuroS&P*, pages 687–706, 2023.

[GBHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 323–345. Springer, Berlin, Heidelberg, August 2016.

[GJJ22] Qian Guo, Andreas Johansson, and Thomas Johansson. A key-recovery side-channel attack on classic mceliece implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):800–827, 2022.

[GNNJ23] Qian Guo, Denis Nabokov, Alexander Nilsson, and Thomas Johansson. SCA-LDPC: A code-based framework for key-recovery side-channel attacks on post-quantum encryption schemes. Cryptology ePrint Archive, Report 2023/294, 2023.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 444–461. Springer, Berlin, Heidelberg, August 2014.

[HMM10] Wilko Henecka, Alexander May, and Alexander Meurer. Correcting errors in RSA private keys. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 351–369. Springer, Berlin, Heidelberg, August 2010.

[How97] Nick Howgrave-Graham. Finding small roots of univariate modular equations revisited. In *IMACC*, volume 1355 of *Lecture Notes in Computer Science*, pages 131–142. Springer, 1997.

[HS01] Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Des. Codes Cryptogr.*, 23(3):283–290, 2001.

[HS09] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 1–17. Springer, Berlin, Heidelberg, August 2009.

[HSC+23] Senyang Huang, Rui Qi Sim, Chitchanok Chuengsatiansup, Qian Guo, and Thomas Johansson. Cache-timing attack against HQC. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):136–163, 2023.

[HSH+08] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In Paul C. van Oorschot, editor, *USENIX Security 2008*, pages 45–60. USENIX Association, July / August 2008.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Berlin, Heidelberg, August 2003.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, Berlin, Heidelberg, August 1996.

[KPP20] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. Single-trace attacks on keccak. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):243–268, 2020.

[LNPS20] Norman Lahr, Ruben Niederhagen, Richard Petri, and Simona Samardjiska. Side channel information set decoding using iterative chunking - plaintext recovery from the "classic McEliece" hardware reference implementation. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 881–910. Springer, Cham, December 2020.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622. IEEE Computer Society, 2015.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[Nat24] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard, 2024.

[NCOS16] Erick Nascimento, Lukasz Chmielewski, David F. Oswald, and Peter Schwabe. Attacking embedded ECC implementations through cmov side channels. In *SAC*, volume 10532 of *Lecture Notes in Computer Science*, pages 99–119. Springer, 2016.

[nis] NIST's post-quantum cryptography standardization project. https://csrc.nist.gov/projects/post-quantum-cryptography.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *ICICS*, pages 529–545, 2006.

[NS02] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *J. Cryptol.*, 15(3):151–176, 2002.

[NS03] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Des. Codes Cryptogr.*, 30(2):201–217, 2003.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[PPM+23] Stjepan Picek, Guilherme Perin, Luca Mariot, Lichao Wu, and Lejla Batina. Sok: Deep learning-based physical side-channel analysis. *ACM Comput. Surv.*, 55(11):227:1–227:35, 2023.

[PPS12] Kenneth G. Paterson, Antigoni Polychroniadou, and Dale L. Sibborn. A coding-theoretic approach to recovering noisy RSA keys. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 386–403. Springer, Berlin, Heidelberg, December 2012.

[PSKH18] Aesun Park, Kyung-Ah Shim, Namhun Koo, and Dong-Guk Han. 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.

[RKPS14] Heinrich Riebler, Tobias Kenter, Christian Plessl, and Christoph Sorge. Reconstructing AES key schedules from decayed memory with FPGAs. In *FCCM*, pages 222–229. IEEE Computer Society, 2014.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 413–427. Springer, Berlin, Heidelberg, August 2010.

[SCW24] Sophie Schmieg, Deirdre Connolly, and Bas Westerbaan. Official comment on FIPS 203 ipd: seed as decapsulation key, 2024. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/5CT4NC_6zRI/m/lpifFrpWAwAJ.

[Sen00] Nicolas Sendrier. Finding the permutation between equivalent linear codes: The support splitting algorithm. *IEEE Trans. Inf. Theory*, 46(4):1193–1203, 2000.

[SHR+22] Thomas Schamberger, Lukas Holzbaur, Julian Renner, Antonia Wachter-Zeh, and Georg Sigl. A Power Side-Channel Attack on the Reed-Muller Reed-Solomon Version of the HQC Cryptosystem. In *Post-Quantum Cryptography*, pages 327–352, 2022.

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In Tim Güneysu and Helena Handschuh, editors, *CHES 2015*, volume 9293 of *LNCS*, pages 495–513. Springer, Berlin, Heidelberg, September 2015.

[SSB+21] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In *NDSS 2021*. The Internet Society, February 2021.

[STM+08] Falko Strenzke, Erik Tews, H. Gregor Molter, Raphael Overbeck, and Abdulhadi Shoufan. Side channels in the McEliece PKC. In Johannes Buchmann and Jintai Ding, editors, *Post-quantum cryptography, second international workshop, PQCRYPTO 2008*, pages 216–229. Springer, Berlin, Heidelberg, October 2008.

[Str10] Falko Strenzke. A timing attack against the secret permutation in the McEliece PKC. In Nicolas Sendrier, editor, *The Third International Workshop on Post-Quantum Cryptography, PQCRYPTO 2010*, pages 95–107. Springer, Berlin, Heidelberg, May 2010.

[UXT+22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: A generic power/EM analysis on post-quantum KEMs. *IACR TCHES*, 2022(1):296–322, 2022.

[Wal01] Colin D. Walter. Sliding windows succumbs to big mac attack. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES 2001*, volume 2162 of *LNCS*, pages 286–299. Springer, Berlin, Heidelberg, May 2001.

[Wea21] Rhys Weatherley. Performance of masked algorithms. https://rweather.github.io/lightweight-crypto/performance_masking.html, 2021.

[YF14] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732. USENIX Association, 2014.

[ZTO+23] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. BunnyHop: Exploiting the instruction prefetcher. In *USENIX Security Symposium*. USENIX Association, 2023.