On breaking McEliece keys using brute force

Lorenz Panny

Technische Universität München, Germany
lorenz@yx7.cc

Abstract. In the McEliece public-key encryption scheme, a private key is almost always not determined uniquely by its associated public key. This paper gives a structural characterization of equivalent private keys, generalizing a result known for the more approachable special case $|L| = q$. These equivalences reduce the cost estimate for a simple private-key search using the support-splitting algorithm (SSA) by a polynomial but practically very substantial factor. We provide an optimized software implementation of the SSA for this kind of key search and demonstrate its capabilities in practice by solving a key-recovery challenge with a naïve a-priori cost estimate of 2^{83} bit operations in just ≈ 1400 core days, testing ≈ 9400 private-key candidates per core and second in the process. We stress that the speedup from those equivalences is merely polynomial and does not indicate any weakness in realistic instantiations of the McEliece cryptosystem, whose parameter choices are primarily constrained by decoding attacks rather than ludicrously more expensive key-recovery attacks.

Keywords: Code-based cryptography · concrete cryptanalysis · Goppa codes

1 Introduction

Code-based cryptography, and McEliece's public-key encryption scheme [McE78] in particular, are one of the primary candidates for post-quantum cryptography. The idea is simple: Decoding is computationally hard for *random* linear codes, whereas there exist efficient decoders for some more structured families of codes. Users can thus generate an efficiently decodable code in private and "scramble" it into an equivalent code which "appears" random and can be published. (Concretely, for the secret code, the standard choice is a Goppa code over the binary finite field $\mathbb{F}_2$, which can be decoded efficiently using algorithms due to Patterson [Pat75] or Bernstein [Ber24].) The secret transformation from the structured private code to the public random-looking code thus forms a trapdoor that allows the private-key holder to decode efficiently, while (presumably) nobody else will be able to do so. As a consequence, as pointed out in McEliece's original paper:

* Date of this document: 2025-04-07.

> It appears that an eavesdropper has two basic attacks to try; first, to try to recover G from G' and so to be able to use Patterson's algorithm. Second, he might attempt to to recover $\mathbf{u}$ from $\mathbf{x}$ without learning G. [McE78]

These two approaches are nowadays known as the "key-recovery attack" and the "decoding attack" against McEliece, respectively. For decoding, the state of the art is *information-set decoding (ISD)*, of which there exist a large number of variants: See [CME22r, § 1] for a long list of references. In comparison, the key-recovery problem has been receiving relatively slim consideration in the literature. The reason for this seems to be that parameter choices for McEliece have always been constrained primarily by the decoding attack, which has a much lower complexity than all known key-recovery attacks. Indeed, as pointed out by McEliece:

> The first attack seems hopeless if n and t are large enough because there are so many possibilities for G, not to mention the possibilities for S and P. [McE78]

However, for independent applications that might rely on the hardness of the McEliece key-recovery problem only, and in the interest of gaining sufficient clarity about the situation in general, there still seems to be value in determining the exact difficulty of this problem.

In this paper, we theoretically investigate and practically optimize a well-known approach for the key-recovery problem, which falls into the category of "smart brute force": Enumerate part of the private-key components (the Goppa polynomial g and *set* of evaluation points $\{\alpha_1, ..., \alpha_n\}$) and check in an efficient manner, using Sendrier's *support-splitting algorithm* [Sen00], whether there exists a valid assignment for the remaining component (permutation of evaluation points) that leads the key-generation routine to produce the targeted public key.

The historical lack of concrete cost estimates for the McEliece key-recovery problem prompted the *Technology Innovation Institute (TII)* in Abu Dhabi to issue a set of challenge instances for this problem, covering estimated difficulties between 22 and 255 bits. The largest challenge instance that was solved during the admissible time span (May 2023 to May 2024) was a McEliece public key with parameters $(m, t, n) = (8, 5, 253)$ and an a-priori difficulty estimate of 83 bits. This estimate matches the result of multiplying the number of private keys up to permutation ($\approx 2^{tm}/t \cdot \binom{2^m}{n}$) by a rough cost estimate for Gauß elimination (n^3), corresponding to a straightforward black-box application of the support-splitting algorithm to the key-recovery problem while setting constant factors in the cost of linear algebra to one. We demonstrate that this naïve estimate for the complexity of a brute-force key search overshoots the actual cost in two orthogonal ways: First, the *number* of guesses required is smaller (by a logarithmic factor) than the number of private keys, since a polynomially large set of private keys corresponds to any given public key. Second, the cost of testing each candidate private key can be *amortized* across multiple guesses, resulting in a lower cost per guess compared to testing each guess completely separately and independently. We further describe and demonstrate some useful implementation techniques which drastically speed up the key search in practice.

Acknowledgements. This work was provoked by the *TII McEliece Challenges*.[1] Thanks to Andre Esser for encouraging me to delve into this topic, and to Violetta Weger for educating me about some aspects of coding theory. I also wish to thank my previous employer, Academia Sinica, whose equipment was used during part of this research.

1.1 Notation

Fix a prime power $q = p^m$ and let t, n be positive integers such that $tm \leq n \leq q$. For a monic squarefree polynomial $g \in \mathbb{F}_q[x]$ of degree t and a length-n sequence $L = (\alpha_1, ..., \alpha_n)$ of distinct elements $\alpha_i \in \mathbb{F}_q$ satisfying $g(\alpha_i) \neq 0$, the *Goppa code* defined by g and L is the $\mathbb{F}_p$-vector space

$$\Gamma(g, L) = \left\{ c \in \mathbb{F}_p^n \,:\, \sum_{i=1}^{n} \frac{c_i}{x - \alpha_i} \;\equiv\; 0 \pmod{g} \right\}.$$

Identifying elements of $\mathbb{F}_{p^m}$ with vectors in $\mathbb{F}_p^m$ using an arbitrary isomorphism of $\mathbb{F}_p$-vector spaces, a parity-check matrix of $\Gamma(g, L)$ is given by

$$\left[\begin{array}{ccc} \alpha_1^0/g(\alpha_1) & \cdots & \alpha_n^0/g(\alpha_n) \\ \hline \alpha_1^1/g(\alpha_1) & \cdots & \alpha_n^1/g(\alpha_n) \\ \hline \vdots & \ddots & \vdots \\ \hline \alpha_1^{t-1}/g(\alpha_1) & \cdots & \alpha_n^{t-1}/g(\alpha_n) \end{array}\right] \in \mathbb{F}_p^{tm \times n}.$$

Two codes (subspaces of $\mathbb{F}_p^n$) are *permutation-equivalent* if one can be obtained from the other by permuting the coordinates of $\mathbb{F}_p^n$. Notice that permuting L evidently produces equivalent Goppa codes. Write $\mathsf{set}(L)$ for the *set* $\{\alpha_1, ..., \alpha_n\}$.

Throughout, let $\mathsf{Irr}_t(q)$ denote the number of monic irreducible polynomials over $\mathbb{F}_q$ of degree t. Note $q^t - \log_2(t) \cdot q^{t/2} \leq t \cdot \mathsf{Irr}_t(q) \leq q^t$, hence $\mathsf{Irr}_t(q) \approx q^t/t$.

1.2 Key search

The most obvious approach to recovering a McEliece private key is to simply guess the pair (g, L), compute the associated public key, and check for equality. Since there are $q!/n!$ choices for L, it is easily seen that $\mathsf{Irr}_t(q) \cdot q!/n!$ attempts suffice on average. (This is under the assumption that g is irreducible.)

One may gain a reduction factor of $(q - n)!$ in the number of attempts by guessing $\mathsf{set}(L)$ instead of L, and consequently checking for permutation equivalence of the resulting codes rather than equality. This test can be performed efficiently in practice using the *support-splitting algorithm* [Sen00]. Using this

[1] https://crowdchallenge.tii.ae/mceliece-challenges/

approach, the expected number of required guesses appears to be $\mathsf{Irr}_t(q) \cdot \binom{q}{n}$, although testing each guess now requires slightly more computation than before.

However, as it turns out, the actual complexity of a brute-force key search is even lower than this: The reason is that every public key has a relatively large number of equivalent private keys associated to it, and recovering any equivalent key suffices; indeed, finding *the* private key is impossible since it is information-theoretically indistinguishable from its equivalent keys. We will analyze these additional equivalences in the upcoming Section 2.

2 Equivalent keys

In this section, we study an important algebraic relationship on the data defining a Goppa code which induces equivalences on the resulting codes. The protagonist is the following group of transformations:

Definition 1. *The* affine semilinear group *of a finite field $\mathbb{F}_q$ is the subgroup*

$$\mathrm{A\Gamma L}(q) := \left\{ (x \mapsto Ax^{\varphi}+B) \,:\, A \in \mathbb{F}_q^{\times},\ B \in \mathbb{F}_q,\ \varphi \in \mathrm{Aut}(\mathbb{F}_q) \right\}.$$

of $\mathrm{Sym}(\mathbb{F}_q)$. *Its elements are called* affine semilinear transformations.

One subtle detail in Definition 1 is that elements of $\mathrm{A\Gamma L}(q)$ might be viewed either as a triple (A, B, φ) or as a permutation in $\mathrm{Sym}(\mathbb{F}_q)$. It is easy to see that these two viewpoints are indeed equivalent:

Lemma 2. *Consider the triples (A, B, φ) and (A', B', φ') with $A, A' \in \mathbb{F}_q^{\times}$, $B, B' \in \mathbb{F}_q$, and $\varphi, \varphi' \in \mathrm{Aut}(\mathbb{F}_q)$.*

If $Ax^{\varphi} + B = A'x^{\varphi'} + B'$ holds for all $x \in \mathbb{F}_q$, then $(A, B, \varphi) = (A', B', \varphi')$.

Proof. Since φ, φ' are field automorphisms, they leave $0, 1 \in \mathbb{F}_q$ invariant. Hence, substituting $x = 0$ into $Ax^{\varphi} + B = A'x^{\varphi'} + B'$ yields $B = B'$, and substituting $x = 1$ then shows $A = A'$. Finally, if $x^{\varphi} = x^{\varphi'}$ for all $x \in \mathbb{F}_q$, then $\varphi = \varphi'$. □

Corollary 3. *The cardinality of* $\mathrm{A\Gamma L}(q)$ *equals $q(q-1)m$ where $m = \log_p(q)$.*

Remark 4. Explictly, in the "triples" viewpoint, formulas for computing in $\mathrm{A\Gamma L}(q)$ are given as follows:

$$(A, B, \varphi)^{-1} = \left((1/A)^{\varphi^{-1}}, (-B/A)^{\varphi^{-1}}, \varphi^{-1}\right);$$
$$(A', B', \varphi') \circ (A, B, \varphi) = (A'A^{\varphi'}, A'B^{\varphi'} + B', \varphi' \circ \varphi).$$

The group $\mathrm{A\Gamma L}(q)$ evidently acts on $\mathbb{F}_q$. We extend this action to subsets of $\mathbb{F}_q$ by element-wise application and to vectors in $\mathbb{F}_q^n$ by coordinate-wise application. Furthermore, we define an action on polynomials in $\mathbb{F}_q[x]$ by acting on the roots:

Definition 5. *Consider a monic polynomial $g \in \mathbb{F}_q[x]$ of degree t and let $\gamma_1, ..., \gamma_t$ denote its (not necessarily distinct) roots in $\overline{\mathbb{F}_q}$. For $\tau = (x \mapsto Ax^\varphi + B) \in \mathrm{A\Gamma L}(q)$, lift φ to an arbitrary automorphism $\Phi \in \mathrm{Aut}(\overline{\mathbb{F}_q})$ which restricts to φ in $\mathrm{Aut}(\mathbb{F}_q)$, and define $\tau * g$ as the polynomial $\prod_{i=1}^{t}(x - A\gamma_i^\Phi - B) \in \mathbb{F}_q[x]$.*

Lemma 6. *The action $*$ of $\mathrm{A\Gamma L}(q)$ on monic polynomials in $\mathbb{F}_q[x]$ is well-defined.*

Proof. Let $\sigma \in \mathrm{Gal}(\overline{\mathbb{F}_q}/\mathbb{F}_q)$. Since g is defined over $\mathbb{F}_q$, there is a permutation π of the set $\{1, ..., t\}$ such that $\gamma_i^\sigma = \gamma_{\pi(i)}$. Utilizing the fact that $\mathrm{Aut}(\overline{\mathbb{F}_q})$ is abelian and hence $\Phi\sigma = \sigma\Phi$, we therefore get

$$(A\gamma_i^\Phi + B)^\sigma = A\gamma_i^{\Phi\sigma} + B = A\gamma_{\pi(i)}^\Phi + B\,.$$

This shows that σ acts as a permutation on the elements $A\gamma_i^\Phi + B$, which by definition make up the roots of $\tau * g$, and therefore $\tau * g$ must be defined over $\mathbb{F}_q$.

It remains to show that the choice of lift Φ does not matter: This is because for all $\Phi, \Phi' \in \mathrm{Aut}(\overline{\mathbb{F}_q})$ which restrict to φ in $\mathrm{Aut}(\mathbb{F}_q)$, we have $\Phi^{-1}\Phi' \in \mathrm{Gal}(\overline{\mathbb{F}_q}/\mathbb{F}_q)$. Let π again denote a permutation such that $\gamma_i^{\Phi^{-1}\Phi'} = \gamma_{\pi(i)}$. Then

$$\gamma_i^{\Phi'} = \gamma_i^{\Phi'\Phi\Phi^{-1}} = \gamma_{\pi(i)}^\Phi\,,$$

thus replacing Φ by Φ' merely induces a permutation on the elements $A\gamma_i^\Phi + B$, which leaves the polynomial $\tau * g$ invariant. □

The following main theorem is a generalization of a result originally due to Moreno [Mor79], which was restricted to the important special case of irreducible g and $\mathsf{set}(L) = \mathbb{F}_q$, in characteristic 2 only. The cryptographic literature usually attributes Moreno's result to Gibson [Gib91], who appears to have rediscovered it later and introduced it to the cryptanalytic context.

Theorem 7. *Recall the setting of Section 1.1 and let $\tau \in \mathrm{A\Gamma L}(q)$. Then*

$$\Gamma(\tau * g,\, \tau * L) = \Gamma(g, L)\,.$$

Proof. Let $c \in \Gamma(g, L)$. For any root $\gamma \in \overline{\mathbb{F}_q}$ of g, we have $\sum_{i=1}^{n} c_i/(\gamma - \alpha_i) = 0$ as evaluation at γ defines a ring homomorphism $\mathbb{F}_q[x]/g \to \overline{\mathbb{F}_q}$. Lift $\varphi \in \mathrm{Aut}(\mathbb{F}_q)$ to $\Phi \in \mathrm{Aut}(\overline{\mathbb{F}_q})$ arbitrarily as before and write $T\colon x \mapsto Ax^\Phi + B$ for shorthand. A simple calculation then reveals

$$\sum_{i=1}^{n} c_i/\big(T(\gamma) - \tau(\alpha_i)\big) = \Big(\sum_{i=1}^{n} c_i/(\gamma - \alpha_i)\Big)^\Phi/A = 0^\Phi/A = 0\,.$$

Since this holds for all roots $T(\gamma)$ of $\tau * g$, the Chinese remainder theorem implies

$$\sum_{i=1}^{n} c_i/\big(x - \tau(\alpha_i)\big) \equiv 0 \pmod{\tau * g}\,,$$

which shows $c \in \Gamma(\tau * g, \tau * L)$. We have thus proved $\Gamma(g, L) \subseteq \Gamma(\tau * g, \tau * L)$, and equality follows from applying the same argument to the code defined by $\tau * g$ and $\tau * L$ with transformation τ^{-1}. □

Regrettably, however, the group action by $\mathrm{A\Gamma L}(q)$ on the set of McEliece private keys does not immediately lead to either lower *or* upper bounds on the number of equivalent private keys per public key. This is due to two separate, counteracting effects:

- It can occasionally happen that some of the equivalences induced by $\mathrm{A\Gamma L}(q)$ as in the theorem are *already* explained by permutation equivalence: This is the case whenever $\mathrm{A\Gamma L}(q)$ stabilizes the polynomial g and acts as a permutation on the list L, i.e., stabilizes $\mathsf{set}(L)$. As a consequence, simply multiplying the number of equivalences coming from $\mathrm{A\Gamma L}(q)$ by the cardinality $n!$ of $\mathcal{S}_n$ leads to overcounting the number of equivalent private keys in general.
- Not all equivalences on public keys are explained by the action of $\mathrm{A\Gamma L}(q)$: We refer to this phenomenon as "spurious equivalences". They, too, are rare, but they may in general lead to undercounting the number of equivalent private keys when estimating it via $\mathrm{A\Gamma L}(q)$.

It seems difficult to strictly prove anything about the prevalance and impact of those effects for a given choice of parameters.

From a practical standpoint, however, since both of these exceptional cases occur only exceedingly rarely when concerned with reasonably large parameter sets, we may for our cryptanalytic purposes assume that we are in the average situation, in which the equivalent keys are indeed in bijection with $\mathrm{A\Gamma L}(q)$ and the count resulting from Heuristic 8 is therefore essentially (or exactly) correct.

Heuristic 8. *Generically, the number of Goppa codes with parameters (p, m, t, n) modulo permutation equivalence is about $\mathsf{Irr}_q(t) \cdot \binom{q}{n} / \big(q(q-1)m\big)$ where $q = p^m$.*

Also note that public keys having more equivalent private keys are more likely to be sampled during key generation, hence the distribution of public keys is biased towards the "weaker" ones with respect to a brute-force key-search attack.

3 The support-splitting algorithm

In the year 2000, Sendrier introduced a practically efficient algorithm for testing whether two linear codes are permutation-equivalent, and if so, to recover the permutation [Sen00]. The approach crucially relies on *permutation invariants* of linear codes: Required are efficiently computable values associated to a linear code which are (1) invariant under permutations, and (2) discriminant, i.e., they tend to take on different values for codes which aren't permutation-equivalent.

We remark that not much has been proven about either the correctness or the complexity of this algorithm. Empirically, however, it appears to work very well for random codes as well as for the family of Goppa codes used in McEliece.

Splitting the support. Provided a sufficiently discriminant invariant $\mathcal{V}$, given two codes $C, C' \subseteq \mathbb{F}_q^n$, it is conceptually easy to recover a permutation $\pi \in \mathcal{S}_n$

connecting the two codes, assuming one exists: The core subroutine takes as input two partitions $\{1, ..., n\} = S_1 \dot\cup \cdots \dot\cup S_\ell = T_1 \dot\cup \cdots \dot\cup T_\ell$ such that $\pi(S_i) = T_i$ for all $i \in \{1, ..., \ell\}$, and outputs another, finer pair of partitions with the same property. Repeating this process will eventually yield a partition with all $|S_i| = |T_i| = 1$, which allows us to read off π. (If the codes are in fact not permutation-equivalent, the subroutine should either detect this and fail, or output an arbitrary finer partition: One may simply check at the end that π is a valid solution.)

Initial partition. The refining step below assumes that a relatively fine partition of the set of positions of C and C' has already been computed. Obtaining such an initial partition can be done easily by puncturing both codes C and C' at each position $i \in \{1, ..., n\}$ and computing the invariants $\mathcal{V}$ associated to each punctured code C_i resp. C'_i; the partition is then given by grouping all C_i with the same $\mathcal{V}(C_i)$ together into one S_k, and similarly for $\mathcal{V}(C'_i)$ into T_k. (Note that $\mathcal{V}(C_i)$ and $\mathcal{V}(C'_i)$ can be used directly as the indices k for the buckets S_k and T_k.)

If $\mathcal{V}(C) \neq \mathcal{V}(C')$, or if the partitions for C and C' resulting from the initial set of singly-punctured codes are not "compatible" in the sense that the cardinalities of the sets S_k resp. T_k of singly-punctured codes C_i and C'_i with the same value under $\mathcal{V}$ match up pairwise, the codes cannot be permutation-equivalent.

Refining step. The refining procedure works as follows: Iterate over nonempty subsets $I \subseteq \{1, ..., \ell\}$, for instance in order of increasing cardinality. For each I, puncture the codes in each set S_k at the positions in $\bigcup_{i \in I} S_i$ and compute the associated invariant $\mathcal{V}$. Now, hopefully, this procedure yields a nonconstant function on S_k, so we may split S_k further into the preimage sets of its values. Applying the same procedure to T_k leads to a partition $T_{k,1} \dot\cup \cdots \dot\cup T_{k,r} = T_k$ for which $\pi(S_{k,i}) = T_{k,i}$. In this case, we have thus successfully refined the partition.

This refining step can fail in two ways: First, it could happen that there are no more subsets $I \subseteq \{1, ..., \ell\}$ for which $\mathcal{V}$ refines the partition after puncturing at the positions indicated by I. In that case, one may resort to brute-forcing parts of the permutation π in order to refine the partition anyway — in a possibly incorrect way, but of course a valid guess must exist if the codes are actually permutation-equivalent. Second, if the preimage sets associated to each value of the punctured codes under $\mathcal{V}$ for S_k resp. T_k do not form compatible partitions, i.e., if the cardinalities of the preimage sets for each value do not match up, then the codes cannot be permutation-equivalent.

Remark 9. The original description of the support-splitting algorithm [Sen00] also applies *shortenings* in addition to puncturings to the input codes. We found that this adds only little discriminatory power while significantly complicating the algorithms, hence we decided not to use shortenings in our implementation.

Hull enumerators. The key insight in [Sen00] was that the *enumerator* of the *hull* of a code forms a permutation invariant which is almost always efficiently computable and sufficiently discriminant, at least in practice, to enable the

support-splitting algorithm to work well. The following definitions are all standard in coding theory: The *enumerator* $\mathcal{W}(C)$ of a linear code $C \subseteq \mathbb{F}_q^n$ is the vector $(w_0, w_1, ..., w_n) \in \mathbb{Z}_{\geq 0}^{n+1}$ where $w_i = \left|\{c \in C : \mathrm{wt}(c) = i\}\right|$. The *dual* of a code C is the code $C^\perp = \left\{w \in \mathbb{F}_q^n : \forall v \in C.\ \langle v, w\rangle = 0\right\}$. The *hull* of C is the code $C \cap C^\perp$.

On average, hulls are random-looking subspaces of relatively small dimension: It was proven in [Sen97, Theorem 4] that the dimension of the hull for a random code roughly obeys a discrete half-normal distribution;[2] hence, the probability of it exceeding a certain $\ell_0 \in \mathbb{Z}_{\geq 0}$ is bounded above by $q^{-\Omega(\ell_0^2)}$, which shrinks superexponentially as ℓ_0 grows.

As a consequence, computing the hull enumerator of a sufficiently random code can almost always be done very quickly: Since the dimension of the hull is overwhelmingly likely to be tiny, one may simply iterate over all vectors contained in the hull and tally their weights. Indeed, the core observation in [Sen00] was that the hull enumerator is (empirically) an efficiently computable and sufficiently discriminant invariant $\mathcal{V}$ to render the support-splitting approach highly effective in practice.

Optimizations. Implementing the support-splitting algorithm in a straightforward manner, using standard linear-algebra routines and hash tables for keeping track of the partitions S_k and T_k, leads to significant overheads from dynamic memory allocations and unpredictable memory-access patterns: See Section 4.

We foreshadow that there are two very useful optimizations for the support-splitting algorithm when applied to brute-force key search:

- First, noticing that almost all candidate codes are actually *not* equivalent to the target code, instead of implementing and running the full support-splitting algorithm inside the brute-force search, it is enough to build a simple and fast *filter* which will reject most inequivalent codes swiftly, but need not be perfectly correct, nor recover the permutation in full. See Section 4.5.
- Second, since one of the input codes remains fixed, one may precompute good puncturing locations by essentially running only one side of the support-splitting algorithm for the target code and noting which puncturings worked well to refine the partition, then performing only those puncturings for the candidate equivalent code at runtime while skipping the rest. See Section 4.6.

4 Efficient key search

In the context of a brute-force key search, in which checking for (permutation) equivalence contributes a multiplicative factor to the overall complexity, it is clearly desirable to minimize the cost per equivalence test.

[2] Asymptotically, the proportion of n-dimensional codes over $\mathbb{F}_q$ with hull dimension ℓ approaches $R_\ell := C/\prod_{i=1}^{\ell}(q^i - 1) \approx C/q^{\ell(\ell+1)/2}$, where C is a constant (dependent on q) which satisfies $0.419 < C < 1$. Note that $\sum_{\ell=\ell_0}^{\infty} R_\ell \in \Theta(R_{\ell_0})$.

To set a baseline, let us first discuss the main limitations of a straightforward implementation: Processing a single guess at a time (per thread), representing codes as generator matrices or parity-check matrices, and using generic linear-algebra libraries for the hull and punctured-hull computations. This approach is prone to suffering from several effects causing slowdown:

- Plenty of conditional branches, which may cause mispredictions and therefore expensive rollbacks and cache invalidations. This includes many conditions whose values are essentially (difficult-to-predict) coinflips, such as during the computation of a reduced row echelon form using Gauß elimination.
- Relatively complex, a priori variably-sized data structures. Depending on implementation specifics, these data structures may incur slowdowns in thread-safe memory management on many-core machines. In addition, for large data structures, another issue is that unpredictable memory-access patterns may cause the processor to stall while waiting for data to be fetched from main memory to a fast cache located closer to the processing core.

By contrast, in the following, we describe a collection of techniques designed to work around these issues on typical contemporary many-core CPUs. The key strategy employed in this work is to rephrase the entire equivalence test as a **binary circuit**, which is much more amenable to fast parallel evaluation, on simpler hardware, than the original high-level algorithm. As a byproduct, the same circuit can easily be ported to other platforms, such as GPUs, FPGAs, or even ASICs, to obtain efficient implementations of the key search.

Remark 10. It seems very much conceivable that generic circuit-optimization techniques could help reduce the number of logical operations in the circuits constructed via our approach, leading to a faster key-recovery attack on any of the aforementioned hardware platforms. We have not explored this possibility.

Remark 11. The main case of interest going forward will be binary codes ($q = 2$). However, whenever the algorithms given in the following are easily expressed for general q, we have refrained from needlessly specializing to $q = 2$. Throughout, we will identify the values $0, 1 \in \mathbb{F}_2$ with the booleans false and true respectively.

Moreover, note that some of the algorithms are formulated using (fixed-length) loops and branching for clarity: These abstractions are not admissible in a circuit to evaluate the algorithm and will have to be unrolled into copies of the loop body and conditional assignments using bitmasks, respectively. See the difference between Algorithm 1 and Algorithm 2 for the latter conversion.

The algorithmic building blocks given in Sections 4.2 and 4.3 are essentially from [Sen00, § 6.1]. We reiterate them here for clarity and completeness, and because some of our optimizations (Section 5.1) interact with those details.

4.1 Preliminaries

Throughout the rest of the paper, we adhere to the following definitions and notational conventions.

– A q-ary *code* of dimension m and length n is an m-dimensional subspace of $\mathbb{F}_q^n$. Codes are represented using a *basis matrix* in $\mathbb{F}_q^{m\times n}$, whose rows form an $\mathbb{F}_q$-basis of the code. Occasionally, basis matrices may be padded (or interleaved) with zero rows.
– The *dual* code $C^\perp$ associated to a code C is its orthogonal complement with respect to the standard inner product. It equals the right kernel of a basis matrix. The *hull* of a code C is the intersection $C \cap C^\perp$.
– *Puncturing* a code $C \subseteq \mathbb{F}_q^n$ at position $i \in \{1, ..., n\}$ means projecting it to the subspace $\mathbb{F}_q^{i-1} \times \{0\} \times \mathbb{F}_q^{n-i}$. On a basis matrix, this can be realized by filling the i^{th} column with zeroes.
– The *weight* $\mathrm{wt}(c) \in \{0, ..., n\}$ of a code word $c \in \mathbb{F}_q^n$ is the number of nonzero coordinates in c. The *(weight) enumerator* $\mathcal{W}(C)$ of a code $C \subseteq \mathbb{F}_q^n$ is the integer vector $(w_0, ..., w_n) \in \mathbb{Z}_{\geq 0}^{n+1}$, where $w_i = \big|\{c \in C : \mathrm{wt}(c) = i\}\big|$.

4.2 Keeping pivots on the diagonal

A useful trick to increase the amount of deterministic code paths in the linear-algebra routines is to ensure all pivots are kept on the main diagonal inside the row echelon form computation, by inserting zero rows between non-zero rows in the matrix as appropriate. (This is a non-standard choice: Most descriptions of Gauß elimination keep "empty" rows at the end.) See Figure 1 for an example.

$$\begin{bmatrix}
\mathbf{1}&0&0&1&0&0&1&1&0&0\\
0&\mathbf{1}&0&0&0&0&0&1&0&0\\
0&0&\mathbf{1}&1&0&0&1&0&0&0\\
0&0&0&0&\mathbf{1}&0&1&0&0&0\\
0&0&0&0&0&\mathbf{1}&0&1&0&0\\
0&0&0&0&0&0&0&0&\mathbf{1}&0\\
0&0&0&0&0&0&0&0&0&\mathbf{1}\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0
\end{bmatrix}
\qquad
\begin{bmatrix}
\mathbf{1}&0&0&1&0&0&1&1&0&0\\
0&\mathbf{1}&0&0&0&0&0&1&0&0\\
0&0&\mathbf{1}&1&0&0&1&0&0&0\\
0&0&0&\mathbf{0}&0&0&0&0&0&0\\
0&0&0&0&\mathbf{1}&0&1&0&0&0\\
0&0&0&0&0&\mathbf{1}&0&1&0&0\\
0&0&0&0&0&0&\mathbf{0}&0&0&0\\
0&0&0&0&0&0&0&\mathbf{0}&0&0\\
0&0&0&0&0&0&0&0&\mathbf{1}&0\\
0&0&0&0&0&0&0&0&0&\mathbf{1}
\end{bmatrix}$$

Figure 1. Generating matrices of the same 7-dimensional subspace of $\mathbb{F}_2^{10}$ given in standard reduced row echelon form (left) and in the "diagonal standard form" (right) that is employed in [Sen00] and in this work.

Definition 12. *A square matrix M is in* diagonal standard form *if its nonzero rows are in reduced row echelon form and all pivots lie on the main diagonal.*

This form of a matrix is very convenient for the linear-algebraic types of computational tasks that appear in coding theory: In particular, having a generator matrix of some code in this form allows us to directly read off a basis of the right kernel of the matrix, i.e., a generator matrix of the dual code. The following result was used without proof in [Sen00, § 6.1.1]:

Lemma 13. *A matrix M in diagonal standard form is idempotent, i.e., $M^2 = M$. In particular, the nonzero columns of $\mathbb{1} - M$ form a basis of the right kernel of M.*

Proof. Let M_{ij} denote the coefficient of M in the i^{th} row and j^{th} column. The entry in the i^{th} row and j^{th} column of M^2 thus equals $c = \sum_{k=1}^{n} M_{ik} M_{kj}$. Since all entries of M below the diagonal are zero, this sum shortens to $c = \sum_{k=i}^{j} M_{ik} M_{kj}$. For each k, if $M_{kk} = 1$, then all M_{ik} with $i < k$ lie above a pivot and therefore must be zero, and if $M_{kk} = 0$, then the entire k^{th} row, thus all M_{kj}, must be zero. Since in both cases, it follows that $M_{ik} M_{kj} = 0$ unless $k = i$, we are left with $c = M_{ii} M_{ij}$. Now, if $M_{ii} = 1$, then $c = M_{ij}$ as claimed. If on the other hand $M_{ii} = 0$, then the entire i^{th} row is zero and it follows that $c = M_{ij} = 0$.

From $M^2 = M$ we get $M \cdot (\mathbb{1} - M) = 0$, hence the columns of $\mathbb{1} - M$ are indeed contained in the right kernel of M. The matrix $\mathbb{1} - M$ is upper triangular with every nonzero column having a 1 on the diagonal, hence the nonzero columns are linearly independent. Finally, from the rank–nullity theorem, the nonzero columns of $\mathbb{1} - M$ must form a basis of the right kernel since they correspond precisely to the zero rows of M. □

Using the diagonal standard form was already suggested in the original description of the support-splitting algorithm; however, the motivation there seems to have been to enable Algorithm 4 for punctured hulls given in the following Section 4.3, rather than to optimize the circuit associated to the linear-algebra computation. Indeed, the diagonal standard form (and a transformation matrix) can be computed using a variant of Gauß elimination that involves fewer case distinctions based on the locations of the pivots; see Algorithms 1 and 2.

Lemma 14. *Algorithm 2 is correct and requires $6n^3 + O(n^2)$ logical operations. If the output L is not needed, omitting its computation reduces this to $2n^3 + O(n^2)$.*

Proof. Several routine verifications. (See also the comments in Algorithm 2.) □

From the given formulation of Algorithm 2, it is straightforward to construct a binary circuit that is equivalent to evaluating the entire algorithm: See Figure 2 for a tiny example. Note that this is just for illustration: We do not actually ever compute or export a graph representation of the circuit for the fast SSA filter at any point; rather, we write software (very similar to Algorithm 2) to evaluate the circuit straight away. See also Section 5.1.

Another important property of Algorithms 1 and 2 is that they are *restartable*: When some of the columns of the input matrix are substituted, the execution of the algorithm can continue from the new columns and thereby reuse all the work done up until the first modified column, resulting in a much lower amortized cost. This is very useful in the context of McEliece key search, which involves a brute-force search over subsets of possible column vectors. See Section 5.2 and also [BLP08, § 4], where a very similar approach was used for the decoding attack.

Remark 15. When applied in the context of Algorithm 3, the input matrix B to Algorithm 2 is symmetric. This causes the transformation matrix T to become

Algorithm 1: Transforming a matrix to diagonal standard form.
– Restartable: If the k^{th} column is replaced, continue the main loop from that k.

Input: $A, B \in \mathbb{F}_q^{n \times n}$.
Output: $L, R \in \mathbb{F}_q^{n \times n}$ such that R is in diagonal standard form, and such that there exists $T \in \mathrm{GL}_n(\mathbb{F}_q)$ with $L = TA$ and $R = TB$.

```
// Initialize output variables.
(L, R) ← (A, B)
// Outer loop: Iterate over columns from left to right.
for k from 1 to n do
    if R_kk = 0 then
        // We don't have a pivot. Try to find one.
        for i from 1 to n do
            if i = k then
                continue
            // Does column k not yet have a pivot in row k, but in row i?
            if R_kk = 0 and (i ≥ k or R_ii = 0) and R_ik ≠ 0 then
                L_k ← L_k + L_i
                R_k ← R_k + R_i
                break
        if R_kk = 0 then
            continue
    // Rescale row k row so that the pivot equals 1.
    L_k ← R_kk^{-1} · L_k
    R_k ← R_kk^{-1} · R_k
    // Inner loop: Zero out the rest of column k.
    for i from 1 to n do
        if i = k then
            continue
        if R_ik = 0 then
            continue
        L_i ← L_i − R_ik · L_k
        R_i ← R_i − R_ik · R_k
return (L, R)
```

Algorithm 2: Transforming a matrix over $\mathbb{F}_2$ to diagonal standard form.

– Constant-time: There are no input-dependent branches or memory accesses.
– Restartable: If the k^{th} column is replaced, continue the main loop from that k.
– Note: The output usually differs from the one produced by Algorithm 1.

Input: $A, B \in \mathbb{F}_2^{n \times n}$.
Output: $L, R \in \mathbb{F}_2^{n \times n}$ such that R is in diagonal standard form, and such that there exists $T \in \mathrm{GL}_n(\mathbb{F}_2)$ with $L = TA$ and $R = TB$.

```
// Initialize output variables.
(L, R) ← (A, B)
// Outer loop: Iterate over columns from left to right.
for k from 1 to n do
    // Inner loop: Try to find a row with a pivot and add it to the correct row.
    for i from 1 to n do
        if i = k then
            continue
        // Does column k not yet have a pivot in row k, but in row i?
        m ← R_kk
        if i < k then
            m ← m ∨ R_ii
        m ← ¬m
        m ← m ∧ R_ik
        // If so, add row i to row k.
        for j from 1 to n do
            L_kj ← L_kj ⊕ (m ∧ L_ij)
        for j from k to n do
            R_kj ← R_kj ⊕ (m ∧ R_ij)
    // Inner loop: Try to zero out the rest of column k assuming we have a pivot.
    for i from 1 to n do
        if i = k then
            continue
        // Does column k contain a one in row i?
        m ← R_ik
        // If so, add row k to row i.
        for j from 1 to n do
            L_ij ← L_ij ⊕ (m ∧ L_kj)
        for j from k to n do
            R_ij ← R_ij ⊕ (m ∧ R_kj)
return (L, R)
```

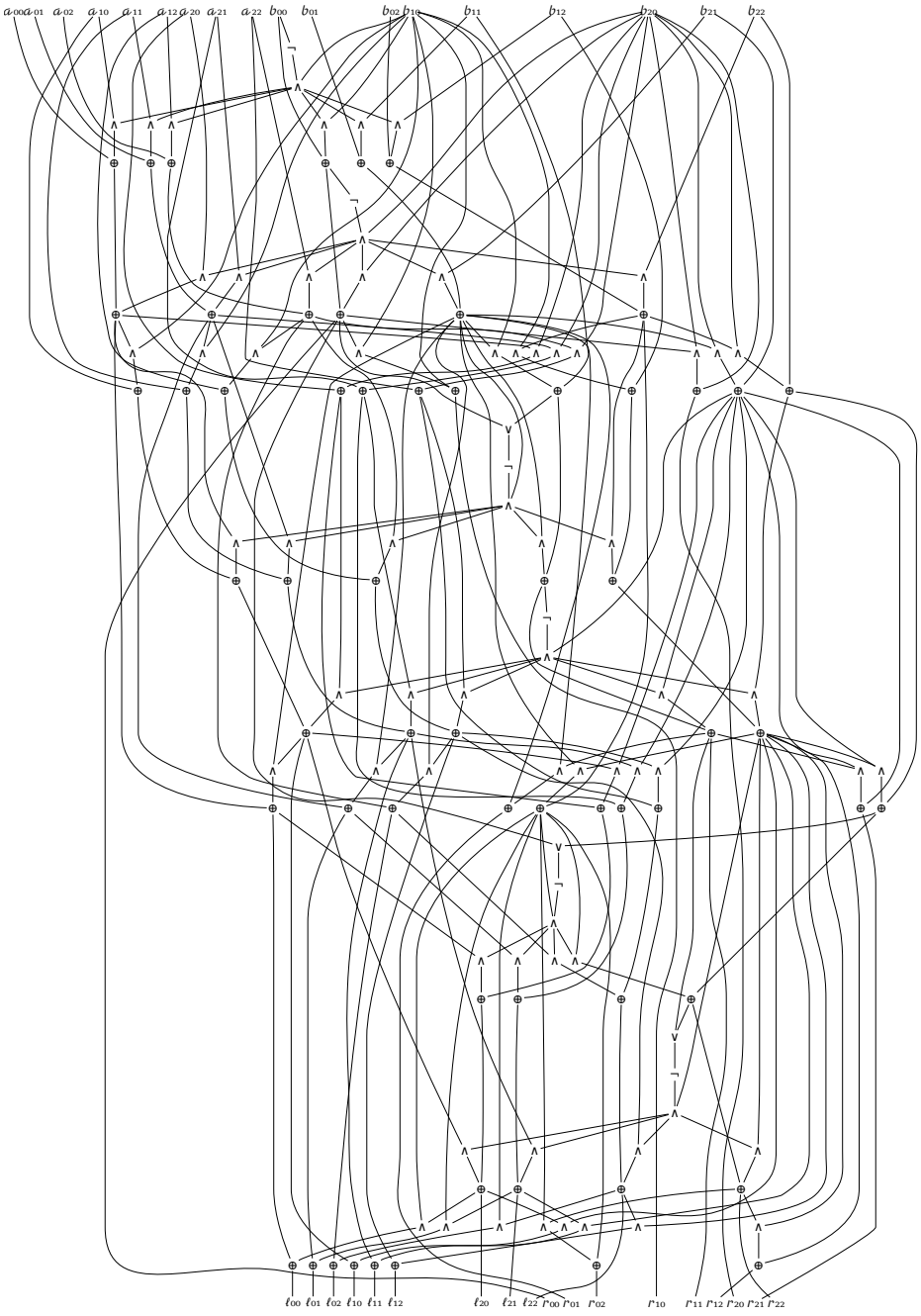

Figure 2. Logic circuit constructed by Algorithm 2 for $n = 3$.

symmetric as well (it equals $(B+D)^{-1}$ where D is the diagonal part of $\mathbb{1}-R$), but unfortunately, we have not succeeded in exploiting this fact to eliminate redundancy and accelerate the algorithm.

Remark 16. Some common optimizations for linear algebra over small finite fields, such as the "Four Russians" method [ADKF70], unfortunately do not apply in our setting: This is because it relies on fast random access to precomputed tables, which becomes expensive in the circuit abstraction.

4.3 Computing everywhere punctured hulls, all at once

All of the following is essentially explained in [Sen00, § 6.1.2]. In this section, we give a simplified and streamlined account for our particular application.

Lemma 17. *Let C be a code given by a square matrix $M \in \mathbb{F}_q^{n\times n}$ in diagonal standard form. Then the hull $C \cap C^\perp$ equals the (right) kernel of $\mathbb{1}+M-M^\mathsf{T}$.*

Proof. This follows from the general fact that $(C_1 \cap C_2)^\perp = C_1^\perp + C_2^\perp$ holds for any two codes C_1, C_2, combined with Lemma 13. □

What this entails is that computing the hull of a linear code can be done entirely *in place*: The nonzero rows of $\mathbb{1}-M$ correspond exactly to the zero rows of M, hence computing the hull via Lemma 17 involves flipping all coefficients lying above the diagonal across the diagonal, negating them, and filling the diagonal with ones, then computing the diagonal standard form again and reading off the kernel. See Algorithm 3.

In addition, Lemma 17 also allows for computing the hulls of *punctured* codes very easily after the hull has been computed: This was detailed for the case $q=2$ in [Sen00, § 5.3.1]. See Algorithm 4.

Remark 18. The algorithms in this paper only require singly-punctured hull enumerators. However, if further puncturings are required, this could easily be achieved using the same methods by keeping track of the matrices L, R computed by Algorithm 1 over the course of multiple sequential applications of Algorithm 4.

4.4 Computing weight enumerators

Computing the weight enumerator of a low-dimensional code over a small finite base field is generally straightforward: Simply enumerate all linear combinations of a basis of the code and record the weight of each code word encountered this way. An optimized method of performing this enumeration (for $q=2$) consists in iterating over the vectors determining linear combinations of a basis according to a *Gray code*, i.e., a way of arranging all bit vectors of a given length in such a way that every pair of two adjacent vectors has Hamming distance one. The details are elaborated in Algorithm 5.

Algorithm 3: Computing the hull of a code in diagonal standard form.

Input: $M \in \mathbb{F}_q^{n \times n}$ in diagonal standard form, defining a code $C \subseteq \mathbb{F}_q^n$.
Output: $H \in \mathbb{F}_q^{\ell \times n}$ a basis matrix of the hull $C \cap C^\perp$.

$A \leftarrow \mathbb{0} \in \mathbb{F}_q^{n \times n}$
$B \leftarrow M$
// Iterate over all rows.
for k *from* 1 *to* n **do**
    // If this row is zero, copy in the corresponding row of of $(\mathbb{1} - M)^\mathsf{T}$.
    if $M_{kk} = 0$ **then**
        $A_{kk}, B_{kk} \leftarrow 1$
        for j *from* 1 *to* k **do**
            $A_{kj}, B_{kj} \leftarrow -M_{jk}$
// Compute the diagonal standard form of B.
Apply Algorithm 1 to the input (A, B), obtaining matrices $L, R \in \mathbb{F}_q^{n \times n}$.
// Read off a basis of the hull from the nonzero columns of $(\mathbb{1} - R)^\mathsf{T}$.
Initialize $\ell \leftarrow 0$ and $H \in \mathbb{F}_q^{0 \times n}$.
for k *from* 1 *to* n **do**
    if $R_{kk} = 0$ **then**
        Append a zero row to H.
        $\ell \leftarrow \ell + 1$
        $H_{\ell k} \leftarrow 1$
        for i *from* 1 *to* $k - 1$ **do**
            $H_{\ell i} \leftarrow -R_{ik}$
return H

Algorithm 4: Computing the hull of a singly-punctured code (over $\mathbb{F}_2$).

Input: $H \in \mathbb{F}_2^{\ell \times n}$ a basis matrix of the hull of a code $C \subseteq \mathbb{F}_2^n$,
matrices $L, R \in \mathbb{F}_2^{n \times n}$ as computed by Algorithm 3 for C,
an index $r \in \{1, ..., n\}$.
Output: $H' \in \mathbb{F}_2^{\ell' \times n}$ a basis matrix of the hull of the code C punctured at r.

$H' \leftarrow H$
// If r is in the support, shorten the code at r.
for k *from* ℓ *down to* 1 **do**
    if H_{kr} **then**
        for i *from* 1 *to* $k - 1$ **do**
            if H_{ir} **then**
                $H'_i \leftarrow H_i \oplus H_k$
        Delete the k^{th} row from H'.
        return H'
// Otherwise, the hull is either augmented by a suitable vector, or unchanged.
if $L_{r,r} = 0$ **then**
    Append the row L_r of L to H'.
return H'

Each individual count inside a weight-enumerator data structure is represented in our binary circuit as a standard ripple-carry incrementer of some suitable (a priori fixed) bit length w. Every such counter also features an overflow bit o to prevent further processing of wrong results after an integer wraparound occured. See Algorithm 6. We note that the counts could alternatively be stored as a unary representation, but this scales much worse: Incrementing a w-bit binary number takes $2w + O(1)$ bit operations, while incrementing a unary number capable of storing w-bit integers takes $2^w + O(1)$ bit operations.

Algorithm 5: Weight enumerator of a low-dimensional code (over $\mathbb{F}_2$).

Input: $H \in \mathbb{F}_2^{\ell \times n}$ a basis matrix of some code in $\mathbb{F}_2^n$.
Output: Weight enumerator $(w_0, ..., w_n) \in \mathbb{Z}_{\geq 0}^{n+1}$ of the code generated by H.

```
Initialize counters w_0, ..., w_n to zero.
Initialize c ← 0 ∈ F_2^n.
Increment w_0.
for s from 1 to 2^ℓ do
    i ← ν_2(s)                                  // The number of trailing zero bits.
    c ← c ⊕ H_i
    // Compute the weight of c.
    Initialize a counter k to zero.
    for j from 1 to n do
        if c_j then
            Increment k.
    // Increment the associated counter.
    for j from 1 to n do
        if k = j then
            Increment w_j.
return (w_0, ..., w_n)
```

Algorithm 6: Incrementing a w-bit integer using bit operations.

Input: Bit array $[b_0, ..., b_{w-1}; o]$ representing $n = b_0 + 2b_1 + \cdots + 2^{w-1}b_{w-1}$ if $o = \mathsf{false}$, else $n = \infty$, and a bit c.
Output: Bit array $[b'_0, ..., b'_{w-1}; o']$ representing $n' = n + c$ if $n + c < 2^w$, else $n' = \infty$.

```
for i from 0 to w − 1 do
    b'_i ← b_i ⊕ c
    c ← b_i ∧ c
o' ← o ∨ c
return [b'_0, ..., b'_{w−1}; o']
```

4.5 Fast filtering using hull enumerators

In the context of a key-recovery attack against McEliece, the main contributing factor of the attack complexity is the cost of repeated permutation-equivalence tests with one of the two inputs (the target public key) remaining fixed. As such, our main goal is to optimize those equivalence tests as much as possible, yielding a "fast filter". Crucially, the construction of the fast filter itself may utilize liberal amounts of precomputation involving the target public-key code.

The key observation is that (1) for codes to possibly be equivalent, the hull enumerators and the multisets of singly-punctured hull enumerators of the two codes must be identical, and (2) it is acceptable for this equivalence test to be imperfect, opening up the potential to achieve much faster execution times by admitting a small chance of false positives.

In the context of our attack, we successfully exploit these two observations to obtain a fast (but imperfect) equivalence check as follows:

- **Puncturings only.** While the original description of the support-splitting code applies both puncturings and shortenings to the input codes, we refrain from using shortenings for simplicity. (See Remark 9.)
- **Single puncturings only.** The full support-splitting algorithm involves a recursive procedure to further discriminate the positions of the two input codes corresponding to each other until the exact permutation inducing the equivalence has been determined. To trade speed for correctness, we instead stop after the first refining step and post-process the resulting stream of candidate equivalent codes using a slower but perfectly correct implementation of the full support-splitting algorithm only afterwards. (See Section 4.7.)
- **Sets instead of multisets.** In theory, we would like to compare the multisets of singly-punctured hull enumerators of an input code to the (precomputed) multiset corresponding to the target public-key code. In reality, this multiset comparison can be approximated by instead simply checking for the presence or absence of certain elements of one multiset in the other multiset: As soon as an expected hull enumerator cannot be found, or an unexpected hull enumerator is encountered, the respective code can immediately be discarded.

The resulting fast filtering method is detailed in Algorithm 7. It relies on the following data structure as input:

Definition 19. *For a code $C \subseteq \mathbb{F}_q^n$, a* fast filter *is the following data structure:*

- *The first component is the hull enumerator of C.*
- *The second component is a list of tuples $(b, v) \in \{\mathsf{true}, \mathsf{false}\} \times \mathbb{Z}_{\geq 0}^{n+1}$ with the property that $b = \mathsf{true}$ if and only if the vector v occurs as one (or multiple) of the singly-punctured hull enumerators of C.*

In a nutshell, this data structure contains information about singly-punctured hull enumerators whose presence or absence quickly establishes (with certainty)

the non-equivalence of "most" random codes to the target code: Check for the presence of enumerators which appear for the target code C but are unlikely to appear in a random code, and check for the absence of enumerators which are likely to appear for a random code but do not appear for C.

Algorithm 7: Evaluating a fast filter for permutation equivalence.

Input: Fast filter $\mathcal{F} = \big(u, \big[(b_1, v_1), ..., (b_\ell, v_\ell)\big]\big)$, some code $C' \subseteq \mathbb{F}_q^n$.
Output: Boolean indicating whether C' passed the filter $\mathcal{F}$ or not.

```
// First make sure the hull enumerators match.
Compute the hull enumerator u' of C'.                    // Algorithms 3 and 5
if u' ≠ u then
    return false

// Verify that all forbidden singly-punctured hull enumerators are absent.
for k from 1 to n do
    Compute the hull enumerator v'_k of C' punctured at k.  // Algorithms 4 and 5
    for i from 1 to ℓ do
        if b_i = false and v'_k = v_k then
            return false

// Verify that all required singly-punctured hull enumerators are present.
for i from 1 to ℓ do
    if b_i = true then
        r ← false
        for k from 1 to n do
            if v'_k = v_k then
                r ← true
                break
        if r = false then
            return false

// All checks passed!
return true
```

The way a "fast filter" is constructed for a given target code C is simple: We sample many random Goppa codes, compute all their singly-punctured hull enumerators, and use a greedy approach to select those enumerators into the filter first whose presence or absence leads to the strongest filtering. See Algorithm 8.

Remark 20. The "fast filtering" approach inherently offers a tradeoff between the complexity (hence the computational effort) of the employed strategy and the quality of the filtering (hence the rate of false positives).

In particular, with the greedy approach used in Algorithm 8, a fast filter can simply be truncated at the end in order to increase the filter's throughput at the expense of some of its distinguishing strength.

Remark 21. One side effect of the particular "fast filtering" technique employed in this work is that the key search proceeds at a much higher rate for some target

Algorithm 8: Constructing a fast filter for permutation equivalence.

Input: Goppa code $C \subseteq \mathbb{F}_p^n$ with $q = p^m$ and $t = \deg(g)$; sample count $K \in \mathbb{Z}_{\geq 1}$.

Output: Fast filter $\mathcal{F}$ which C passes, whereas a random Goppa code with the same parameters is "unlikely" to pass (depending on K).

Compute the hull enumerator $u \in \mathbb{Z}_{\geq 0}^{n+1}$ of C.

Compute the set of all singly-punctured hull enumerators $V_{yes} \subseteq \mathbb{Z}_{\geq 0}^{n+1}$ of C.

for k *from* 1 *to* K **do**

  Sample a random Goppa code C'_k of parameters (p, m, t, n).

  Compute the set of all singly-punctured hull enumerators $V'_k \subseteq \mathbb{Z}_{\geq 0}^{n+1}$ of C'_k.

$\mathcal{V} \leftarrow V_{yes} \cup \bigcup_{k=1}^{K} V'_k$

$\mathcal{C} \leftarrow \{C'_1, ..., C'_K\}$

$\mathcal{F} \leftarrow \big(u, [\,]\big)$

while $\mathcal{V} \neq \emptyset$ **do**

  Filter $\mathcal{C}$ through $\mathcal{F}$ and only keep the codes that pass.

  // At this point one could optionally **break** early once the empirical rate $|\mathcal{C}|/K$ of false positives has subceeded a given target false-positive rate.

  if $\mathcal{C} = \emptyset$ **then**

    break

  for each $v \in \mathcal{V}$ **do**

    $b_v \leftarrow (v \in V_{yes})$

    Compute the proportion ε_v of codes in $\mathcal{C}$ passing the filter $\big(u, \big[(b_v, v)\big]\big)$.

  Find a $v \in \mathcal{V}$ with ε_v minimal among all v and remove v from $\mathcal{V}$.

  Append (b_v, v) to the second component of $\mathcal{F}$.

return $\mathcal{F}$

codes than others: For example, in the most extreme case, the hull enumerator of the public-key code itself would be overwhelmingly unlikely to appear for random Goppa codes, hence a fast filter could consist of simply checking the hull enumerator (without applying any puncturings at all). This effect is visible in Table 1, where smaller "$\#\mathcal{F}$" correlates with higher throughput overall.

The fact that one of the input codes remains fixed implies that bounds on the integers that can appear in weight enumerators, as needed for the algorithms in Section 4.4, are readily available: They can simply be chosen according to the weight enumerators observed for the target public key. While the enumerators computed for a candidate equivalent code may in fact have larger entries, this cannot possibly occur for a valid solution, hence this scenario can be detected (and the offending codes discarded) by simply inspecting the overflow bit.

4.6 Precomputation for the full SSA

For general inputs, the simple filter obtained in the previous Section 4.5, which checks for the presence or absence of certain enumerators within the set of singly-punctured hull enumerators, may not be selective enough. In that case, one could proceed with precomputing the full SSA as described in Section 3: Instead of relying only on singly-punctured hulls, continue down the recursion and puncture out more positions of the code as identified by their punctured hull enumerators.

An important detail is that this can be done as a one-time precomputation for one side of the equivalence test, leaving only the candidate equivalent key to be processed at runtime (following a predictable pattern determined beforehand), which is beneficial in the context of a key-recovery attack: It again enables the whole equivalence test to be phrased as a binary circuit that does not involve any unpredictable control flow or complicated data structures.

We also note that there is a tradeoff between the one-time precomputation cost and the online phase of the attack: Trying (in the notation of Section 3) a larger variety of puncturing sets I at each refining step yields possibly much shorter and therefore more efficient strategies for the resulting equivalence test.

In our implementation of the key-recovery attack (see Section 5), we initially tested a proof-of-concept implementation of this more general approach for the first fast filtering stage, but it quickly became evident that singly-punctured hulls almost always provide strong enough filtering in practice, hence the full "precomputed SSA" approach was abandoned for that purpose and we settled for the simpler strategy described in Section 4.5. We do however use the more general method outlined here when post-processing the codes that passed the fast filter, as described in the following Section 4.7.

4.7 Post-processing codes passing the filter

The fast filter from Section 4.5 is not perfect: It does allow for false positives, i.e., actually inequivalent codes which nevertheless exhibit the correct set of singly-punctured hull enumerators, to be incorrectly identified as candidate equivalent

codes — albeit at a very slow rate, assuming the conditions tested by the fast filter are strong enough, meaning they are satisfied only for a miniscule proportion of random Goppa codes.

Since those false positives are very rare in practice, a second, strictly correct (but more expensive) filtering stage can eliminate them at almost no additional cost per guess: We simply re-run the standard support-splitting algorithm (with a precomputed strategy — see Section 4.6) in order to distinguish the true solution from the false positives that are erroneously passing through the filter. As a byproduct of establishing equivalence to the target code, the SSA at the same time also recovers a permutation that realizes the equivalence between the public-key code and the solution to the key-recovery problem.

5 Implementation

Our parallelized C++ implementation of the McEliece key-recovery attack using the support-splitting algorithm is available for download at the following link:

https://yx7.cc/code/goppify/goppify-latest.tar.xz

In this section, we specialize to $p = 2$ and $q = 2^m$; this is the only case that is currently supported by the implementation. Let w denote the number of bits in a register of the target architecture. (The implementation hardcodes $w = 64$.)

5.1 Bitslicing the fast SSA filter

The most effective optimization employed in our brute-force software to recover McEliece keys is *bitslicing*. Generically, this technique refers to rewriting a given computation as a binary circuit, then evaluating the circuit on w independent inputs *in parallel* using the processor's bitwise logic instructions on registers holding w bits each. This is fundamentally the same concept as batching multiple instances of the same computation using vector instructions, but here it is applied at the level of single bits held in general-purpose CPU registers. See Figure 3.

In the context of linear algebra over $\mathbb{F}_2$, which the support-splitting algorithm heavily relies on, this yields to several important optimizations over the straightforward approach working on one matrix in row-major form at a time:

- **Predictable execution flow:** Avoiding conditional branches and variable-length loops assists the processor in effectively pipelining the execution of the program and helps prevent expensive mispredictions (and the resulting rollbacks) on architectures that rely on speculative execution.
- **Predictable memory-access pattern:** Avoiding random accesses to large memory regions is a crucial consideration in optimizing computations for modern CPUs since fetching data from main memory incurs a potentially very large time penalty. Using data-independent memory-access patterns

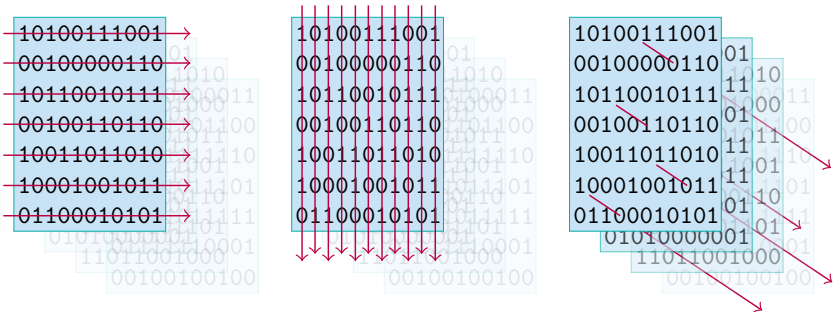

Figure 3. Illustration of the ways a list of matrices over $\mathbb{F}_2$ can be stored in, in (from left to right) row-major, column-major, and bitsliced order. Each arrow indicates the set of bits stored inside a single bit vector (i.e., CPU register) within the target machine.

enables the memory controller on superscalar architectures to prefetch the required information into a lower (faster) level of the cache hierarchy while previous steps of the program are in the process of being executed, resulting in much smaller memory-access latencies and, therefore, higher throughput.

On the opposite side, the primary drawbacks of bitslicing computations are:

- **Random accesses are expensive:** While loading or storing data from or to an array with variable index is expressible in the circuit abstraction (for an example, see Remark 22 and the pivoting step in Algorithm 2), it comes at a linear time cost in the size of the array that is being accessed. In comparison, the RAM model of computation stipulates that a memory access can be performed in constant time, independently of the size of the memory, while more realistic models of computation assign a nonconstant but still sublinear time cost.
- **Integer arithmetic is expensive:** Modern processors typically perform a very large number of bit operations per assembly instruction or clock cycle. Logical bit operations, as employed for bitslicing, offer among the worst throughput in terms of bit operations per unit of time. By comparison, performing integer arithmetic in non-bitsliced representation benefits from specialized silicon inside the CPU, enabling it to perform integer operations with throughput often significantly greater than one instruction per cycle, despite the fact that those operations actually consist of a possibly very large number of bit operations. Therefore, emulating nontrivial computations using relatively primitive operations which could — in non-bitsliced representation — be run through specialized hardware is inherently much slower than simply using the built-in instructions. Nevertheless, when only a small number of simple (say) integer operations is required as part of a larger circuit that can more favorably be expressed in terms of bit operations, it may be beneficial to perform them in the bitsliced representation anyway.

Luckily, neither of those disadvantages has a big impact in our context: The vast majority of the time is spent on linear algebra over $\mathbb{F}_2$, where packing multiple instances into a set of registers in order to exploit the parallelism inherently provided by register-sized bitwise operators offers a clear advantage over using those same hardware units to execute only a single instance at a time.

Remark 22. Algorithms 1 to 7 and 9 are all given in non-bitsliced form for ease of notation. In all cases, converting them into a bitslicable circuit is straightforward: The most important translation step is to replace all conditional expressions by equivalent sequences of bit operations, similar to constant-time cryptographic code: In essence, the conditional assignment "**if** C **then** $R \leftarrow A$ **else** $R \leftarrow B$" turns into the computation of *both* branches $R_{\mathsf{true}} \leftarrow A$ and $R_{\mathsf{false}} \leftarrow B$, followed by the combination of the two possible results using a bitmask, i.e., computing $R \leftarrow (C \wedge R_{\mathsf{true}}) \vee (\neg C \wedge R_{\mathsf{false}})$. Similarly, random accesses to memory can be emulated by iterating over all memory cells that could possibly be accessed and conditionally reading or writing the cell using a bitmask each time.

Shortcuts. Recall that our implementation processes multiple instances of the hull and punctured-hull computations in parallel via bitslicing. For example, when *all* bits in the register holding the value m in Algorithm 2 are unset, the following loops have no effect and can thus be skipped altogether in bitsliced or vectorized software, resulting in noticeable speedups when only few of the instances processed in parallel would normally take that code path. We refer to this as a *shortcut*. Note that this breaks the circuit abstraction.

5.2 Orchestrating the key recovery

On top of the "fast filter" core for rapid batch testing of code equivalences using the bitsliced representation, our key-recovery attack tool consists of several more higher-level procedures that will be detailed below.

Our implementation of the algorithms described in this paper is written in C++23, making extensive use of its efficient and convenient multithreading functionality. We employ the NTL library for computing in $\mathbb{F}_{2^m}$ and $\mathbb{F}_{2^m}[x]$ outside of the bitsliced representation (i.e., for the computations involving $\mathbb{F}_{2^m}$ in Algorithm 10). Note that we exclusively work with parity-check matrices for manipulating Goppa codes, which are slightly easier (or at least more convenient) to construct than generator matrices.

The main loop. Until a solution is found, each processing core repeatedly samples w random (irreducible) polynomials $g_1, ..., g_w \in \mathbb{F}_{2^m}[x]$ of degree t. For

each $\alpha \in \mathbb{F}_{2^m}$, it computes the associated columns

$$v_\alpha^{(k)} := \left[\begin{array}{c} \alpha^0/g_k(\alpha) \\ \hline \alpha^1/g_k(\alpha) \\ \hline \vdots \\ \hline \alpha^{t-1}/g_k(\alpha) \end{array}\right] \in \mathbb{F}_2^{tm} \qquad (*)$$

of a Goppa parity-check matrix (see Section 1.1). The collection of $2^m \cdot w$ columns $\{v_\alpha^{(k)} : k \in \{1, ..., w\},\ \alpha \in \mathbb{F}_{2^m}\}$ is then bitsliced into $2^m \cdot tm$ width-w registers $\{r_i(\alpha) : i \in \{1, ..., tm\},\ \alpha \in \mathbb{F}_{2^m}\}$, with $r_i(\alpha)$ containing the i^{th} coefficient in each of the columns $v_\alpha^{(1)}, ..., v_\alpha^{(w)} \in \mathbb{F}_2^{tm}$.

Finally, the algorithm picks a random ordering for the 2^m elements of $\mathbb{F}_{2^m}$ and thus for the bitsliced columns $(r_1(\alpha), ..., r_{tm}(\alpha))$ that will be combined into candidate parity-check matrices. See Algorithm 10 for details.

Iterating over length-n subsets of $\mathbb{F}_q$. For each choice of the columns $r_i(\alpha)$, the algorithm now enumerates a certain (configurable) number of length-n subsets of those columns, which corresponds to guessing the set of Goppa evaluation points $\mathsf{set}(L)$. These subsets are enumerated in a *depth-first* fashion, systematically constructing all possible combinations of column sets in the candidate Goppa parity-check matrix by filling in the columns from left to right, recursing further to the right whenever possible, and only continuing with the next choice for the "current" column after the algorithm has run out of options for all columns to the right. This enumeration procedure can be implemented using a *stack* data structure; see Algorithm 9 for details.

One major benefit of enumerating subsets of $\mathbb{F}_{2^m}$ in this particular order is that almost all of the time, modifications occur exclusively near the end of the candidate parity-check matrix, which means most of the work involved in the linear-algebra routines can be reused and therefore amortized across multiple guesses for $\mathsf{set}(L)$. This important optimization owes crucially to the fact that the particular variant of Gauß elimination used in Algorithm 2 finalizes the columns of the reduced matrix in diagonal standard form sequentially from left to right, which renders it possible to restart the elimination process midway and continue from whichever column has been swapped out. This is similar to the techniques described in [BLP08, § 4].

Running the filter. Once the bitsliced candidate parity-check matrices have been set up in the way outlined above, all that is left to do is to run the fast filter developed in Section 4.5 on the bitsliced inputs in parallel. This yields a single register containing one bit per candidate parity-check matrix, indicating whether the respective instance has passed the filter. For each parity-check matrix that passed the filter, we reconstruct the matrix in non-bitsliced representation and

Algorithm 9: Searching for a valid subset of column vectors.

Input: Sequence $v_1, .., v_\ell$ of column vectors in $\mathbb{F}_q^d$, dimension $n \in \{1, ..., \ell\}$, and a function $\mathsf{check}\colon \mathbb{F}_q^{d\times n} \to \{0, 1\}$.

Output: Iterator over subsets $\{i_1, ..., i_n\} \subseteq \{1, ..., \ell\}$ satisfying $\mathsf{check}(R) = 1$, where R is the diagonal standard form of the matrix $\big(v_{i_1} \mid \cdots \mid v_{i_n}\big)$.

Initialize an empty stack $\mathcal{S}$ holding values in $\{1, ..., \ell\}^* \times \mathbb{F}_q^{d\times d} \times \mathbb{F}_q^{d\times n}$.
Push $\big(( ), \mathbb{1}, \mathbb{0}\big)$ onto $\mathcal{S}$.
while $\mathcal{S}$ *is not empty* **do**
    Pop the top value $\big((i_1, ..., i_k), L, R\big)$ from $\mathcal{S}$.
    if $k = n$ **then**
        /* At this point, rather than enumerating all $\binom{\ell}{n}$ subsets, one could increment a counter and return early once some limit was reached. */
        if $\mathsf{check}(R) = 1$ **then**
            yield $\{i_1, ..., i_k\}$
    $j \leftarrow i_k + 1$ // convention: $i_0 = 0$
    if $j \le \ell$ **then**
        if $0 < k \le n$ **then**
            // Note on the stack where to continue after the recursive step.
            $(L', R') \leftarrow (L, R)$
            Replace the k^{th} column of R' by the vector $L \cdot v_j$.
            Apply the k^{th} iteration of Algorithm 1 or 2 to the pair (L', R').
            Push $\big((i_1, ..., i_{k-1}, j), L', R'\big)$ onto $\mathcal{S}$.
        if $0 \le k < n$ **then**
            // Set up the stack for entering a recursive depth-first step next.
            Replace the $(k+1)^{\text{th}}$ column of R by the vector $L \cdot v_j$.
            Apply the $(k+1)^{\text{th}}$ iteration of Algorithm 1 or 2 to the pair (L, R).
            Push $\big((i_1, ..., i_k, j), L, R\big)$ onto $\mathcal{S}$.

run a slower non-bitsliced implementation of the full SSA (see Section 4.7) to discover whether the code is truly permutation-equivalent to the target public-key code or a false positive has passed the fast filter.

Algorithm 10: The main key-recovery algorithm.

Input: Fast filter $\mathcal{F}$ for a Goppa code $C \subseteq \mathbb{F}_2^n$ with $q = 2^m$ and $t = \deg(g)$.
Output: Goppa polynomial g and list L of evaluation points defining C.

while true **do**
- Sample w irreducible polynomials $g_1, ..., g_w \in \mathbb{F}_{2^m}[x]$ of degree t.
- Compute the associated columns $v_\alpha^{(1)}, ..., v_\alpha^{(w)}$ according to Equation $(*)$.
- For each $\alpha \in \mathbb{F}_{2^m}$, bitslice the list $\left[v_\alpha^{(1)}, ..., v_\alpha^{(w)}\right] \in (\mathbb{F}_2^{tm})^w$ of w independent columns into a list of length-w registers $r(\alpha) = \left[r_1(\alpha), ..., r_{tm}(\alpha)\right] \in (\mathbb{F}_2^w)^{tm}$.
- Sample a random bijection $\iota\colon \{1, ..., 2^m\} \to \mathbb{F}_{2^m}$ and let $S \leftarrow \left[r(\iota(1)), ..., r(\iota(2^m))\right] \in \left((\mathbb{F}_2^w)^{tm}\right)^{2^m}$.
- Execute the bitsliced version of Algorithm 9 with inputs S and n, using the bitsliced version of Algorithm 7 with input $\mathcal{F}$ for the check function.
- **for each** *index set* $\{i_1, ..., i_n\} \subseteq \{1, ..., 2^m\}$ *returned by Algorithm 9 for the* k^{th} *bit of the bitsliced computation* **do**
  - Rebuild the corresponding Goppa code $C' := \Gamma(g_k, [\iota(i_1), ..., \iota(i_n)])$.
  - Run the standard SSA on input (C, C'), yielding a result $\pi \in \mathcal{S}_n \cup \{\perp\}$.
  - **if** $\pi \neq \perp$ **then**
    - $L \leftarrow \left[\iota(i_{\pi(1)}), ..., \iota(i_{\pi(n)})\right]$
    - **return** (g, L)

Remark 23. Algorithm 10 can easily be adjusted — following essentially the same structure — to work with any other efficient implementation of the fast filter, such as CPU implementations using dedicated vector registers and instructions, GPU implementations, or hardware implementations on FPGAs or ASICs.

5.3 Estimates

An overview of the estimated cost of breaking various (smaller) challenge instances of the McEliece key-recovery problem using our implementation of the attack is displayed in Table 1.

Baseline: Linear algebra using M4RI. The M4RI library [M4RI] is a highly optimized implementation of the "Four Russians" [ADKF70] approach to linear algebra over $\mathbb{F}_2$. Comparing the performance of our bitsliced implementation of Gauß elimination (Algorithm 2) to M4RI reveals that our implementation achieves roughly twice the throughput of MARI when computing reduced echelon forms of random square matrices over $\mathbb{F}_2$ of dimensions $n = 253$. This suggests that the vectorized circuit-based approach employed in this paper is presumably superior when many instances of the same linear-algebra task can be batched, even when the relatively complicated hardware required to run M4RI is available.

Table 1. Overview of the smaller parameter sets from the *TII McEliece Challenges* together with attack complexity estimates using our optimized key-search software. The empirical throughput (private-key guesses per core and second) was measured on an AMD EPYC 9754 processor (Zen 4c microarchitecture) running at 2.25 GHz with hyperthreading disabled.

The required number of guesses was estimated based on Heuristic 8. The columns "$\#\,\mathcal{F}$" and "$\approx \Pr[\mathcal{F}\mapsto\mathsf{true}]$" refer to the length of the used fast filter $\mathcal{F}$, not including the unpunctured hull enumerator itself, and the empirical probability for a random code to pass the filter $\mathcal{F}$, respectively. All filters were constructed by running Algorithm 8 with $K = 100{,}000$. Note that no attempt was made to optimize the estimated total duration of the attack by adjusting the length of the filter.

instance	m	t	n	$\approx \#$ guesses	$\#\,\mathcal{F}$	$\approx \Pr[\mathcal{F}\mapsto\mathsf{true}]$	guesses/(core · s)	$\approx$ core time
39	5	2	28	$2^{11.81}$	5	$2^{-3.68}$	$2^{14.08}$	$\approx$ 0.2 s
41	5	2	27	$2^{14.30}$	5	$2^{-6.32}$	$2^{16.73}$	$\approx$ 0.2 s
43	5	2	26	$2^{16.47}$	6	$2^{-4.25}$	$2^{14.87}$	$2^{1.60}$ s $\approx$ 3.0 s
44	6	2	61	$2^{11.76}$	10	$2^{-10.76}$	$2^{16.61}$	$\approx$ 0.0 s
45	6	3	62	$2^{12.83}$	9	$2^{-5.54}$	$2^{13.61}$	$\approx$ 0.6 s
46	6	4	63	$2^{13.44}$	4	$2^{-13.45}$	$2^{17.31}$	$\approx$ 0.1 s
47	5	2	23	$2^{21.42}$	7	$2^{-6.48}$	$2^{17.35}$	$2^{4.07}$ s $\approx$ 16.8 s
48	6	2	60	$2^{15.69}$	4	$2^{-15.69}$	$2^{18.67}$	$\approx$ 0.1 s
49	5	2	21	$2^{23.62}$	7	$2^{-10.46}$	$2^{20.19}$	$2^{3.43}$ s $\approx$ 10.8 s
50	6	3	61	$2^{17.20}$	14	$2^{-5.41}$	$2^{13.67}$	$2^{3.53}$ s $\approx$ 11.6 s
51	6	4	62	$2^{18.41}$	2	$2^{-18.43}$	$2^{18.64}$	$\approx$ 0.9 s
52	6	5	63	$2^{19.12}$	17	$2^{-7.26}$	$2^{15.16}$	$2^{3.96}$ s $\approx$ 15.5 s
53	6	3	60	$2^{21.13}$	2	$2^{-16.74}$	$2^{18.38}$	$2^{2.75}$ s $\approx$ 6.7 s
55	6	4	61	$2^{22.78}$	10	$2^{-14.13}$	$2^{18.37}$	$2^{4.41}$ s $\approx$ 21.3 s
57	6	3	59	$2^{24.71}$	7	$2^{-16.77}$	$2^{18.23}$	$2^{6.48}$ s $\approx$ 1.5 min
58	6	2	57	$2^{25.63}$	12	$2^{-13.77}$	$2^{18.76}$	$2^{6.87}$ s $\approx$ 1.9 min
59	6	4	60	$2^{26.71}$	9	$2^{-14.28}$	$2^{18.26}$	$2^{8.45}$ s $\approx$ 5.8 min
60	6	3	58	$2^{28.01}$	6	$2^{-15.12}$	$2^{18.68}$	$2^{9.33}$ s $\approx$ 10.7 min
63	6	2	55	$2^{31.10}$	3	$2^{-17.97}$	$2^{19.11}$	$2^{11.99}$ s $\approx$ 1.1 h
65	6	2	54	$2^{33.56}$	9	$2^{-14.64}$	$2^{18.54}$	$2^{15.02}$ s $\approx$ 9.2 h
66	6	3	56	$2^{33.90}$	14	$2^{-7.05}$	$2^{15.11}$	$2^{18.79}$ s $\approx$ 5.2 d
68	6	2	53	$2^{35.85}$	1	$2^{-17.23}$	$2^{19.46}$	$2^{16.39}$ s $\approx$ 1.0 d
69	6	4	57	$2^{36.65}$	9	$2^{-15.23}$	$2^{18.71}$	$2^{17.94}$ s $\approx$ 2.9 d
70	8	5	255	$2^{26.68}$	23	$2^{-9.25}$	$2^{11.79}$	$2^{14.89}$ s $\approx$ 8.4 h
71	6	6	60	$2^{38.13}$	8	$2^{-16.20}$	$2^{18.64}$	$2^{19.49}$ s $\approx$ 8.5 d
72	7	5	125	$2^{34.26}$	1	$2^{-17.19}$	$2^{16.23}$	$2^{18.04}$ s $\approx$ 3.1 d
73	7	6	126	$2^{35.61}$	1	$2^{-23.79}$	$2^{16.07}$	$2^{19.54}$ s $\approx$ 8.8 d
74	7	8	128	$2^{36.20}$	20	$2^{-6.94}$	$2^{10.74}$	$2^{25.47}$ s $\approx$ 1.47 yr
76	6	7	60	$2^{43.91}$	3	$2^{-18.99}$	$2^{18.97}$	$2^{24.93}$ s $\approx$ 1.02 yr
77	7	5	124	$2^{39.23}$	4	$2^{-16.64}$	$2^{15.80}$	$2^{23.43}$ s $\approx$ 4.3 mo
78	6	8	61	$2^{45.78}$	3	$2^{-14.98}$	$2^{18.43}$	$2^{27.35}$ s $\approx$ 5.42 yr
79	7	6	125	$2^{41.00}$	4	$2^{-16.74}$	$2^{15.67}$	$2^{25.33}$ s $\approx$ 1.34 yr
80	7	7	126	$2^{42.39}$	2	$2^{-21.01}$	$2^{16.02}$	$2^{26.37}$ s $\approx$ 2.74 yr
81	7	8	127	$2^{43.20}$	4	$2^{-16.08}$	$2^{15.35}$	$2^{27.86}$ s $\approx$ 7.71 yr
82	6	8	60	$2^{49.72}$	3	$2^{-16.01}$	$2^{18.55}$	$2^{31.16}$ s $\approx$ 76.18 yr
83	8	5	253	$2^{40.08}$	1	$2^{-19.95}$	$2^{13.21}$	$2^{26.87}$ s $\approx$ 3.90 yr
84	8	6	254	$2^{41.42}$	20	$2^{-10.62}$	$2^{12.60}$	$2^{28.82}$ s $\approx$ 14.99 yr
85	8	8	256	$2^{42.01}$	20	$2^{-10.26}$	$2^{9.90}$	$2^{32.10}$ s $\approx$ 146.1 yr
86	7	5	122	$2^{48.22}$	1	$2^{-16.72}$	$2^{16.42}$	$2^{31.79}$ s $\approx$ 118.0 yr
87	7	8	126	$2^{49.19}$	4	$2^{-15.69}$	$2^{15.74}$	$2^{33.45}$ s $\approx$ 371.9 yr
88	7	9	127	$2^{50.03}$	23	$2^{-6.78}$	$2^{12.17}$	$2^{37.86}$ s $\approx$ 7,900 yr
89	8	5	252	$2^{46.06}$	1	$2^{-17.96}$	$2^{13.23}$	$2^{32.83}$ s $\approx$ 242.5 yr
90	7	5	121	$2^{52.34}$	4	$2^{-17.11}$	$2^{16.01}$	$2^{36.33}$ s $\approx$ 2,739 yr
91	8	6	253	$2^{47.82}$	14	$2^{-13.18}$	$2^{13.01}$	$2^{34.81}$ s $\approx$ 954.2 yr
92	8	7	254	$2^{49.19}$	3	$2^{-16.64}$	$2^{13.08}$	$2^{36.11}$ s $\approx$ 2,347 yr
93	8	8	255	$2^{50.01}$	22	$2^{-9.06}$	$2^{11.72}$	$2^{38.29}$ s $\approx$ 10,655 yr
94	7	5	120	$2^{56.26}$	5	$2^{-15.14}$	$2^{16.10}$	$2^{40.16}$ s $\approx$ 39,046 yr
95	7	7	123	$2^{57.38}$	2	$2^{-20.74}$	$2^{16.40}$	$2^{40.98}$ s $\approx$ 68,612 yr
96	9	8	512	$2^{47.83}$	3	$2^{-16.61}$	$2^{8.30}$	$2^{39.53}$ s $\approx$ 25,198 yr
97	7	8	124	$2^{59.55}$	8	$2^{-14.74}$	$2^{16.17}$	$2^{43.38}$ s $\approx$ 363,887 yr
99	7	9	125	$2^{61.42}$	3	$2^{-15.49}$	$2^{15.88}$	$2^{45.54}$ s $\approx$ 1,619,077 yr

5.4 Breaking the "83-bit" instance

In this section we report on details of the successful private-key recovery for the largest solved instance (estimated 83 bits of security) of TII's McEliece Challenges. The target public-key file `pk_McEliece_83.txt`, along with the public-key files for all other difficulty levels, are available at `https://github.com/ElenaKirshanova/tii_decoding_challenge/blob/main/public_keyRec/`.

Solving the challenge was done by simply invoking the script `./solve.sh 83` from the code package without any further modifications. This script first runs the precomputation stage for the target code as described in Sections 4.5 and 4.6, followed by the main key-recovery program (Algorithm 10).

In a stroke of luck, for the particular target public-key code from the challenge, the hull enumerator alone is already quite rare: It only occurs for one in about 3350 random codes. Adding just one additional check for the presence of a certain singly-punctured hull enumerator lowers this proportion below one in a million, resulting in a high rate of filtering for this target key, as indicated in Table 1.

In another stroke of luck, the private key was encountered significantly faster than expected: While Table 1 suggests an estimated $2^{40.08}$ guesses, our software recovered a valid private key for the challenge instance after only $2^{39.04}$ guesses using about 1735 CPU days over the course of about 3.4 wall-clock days.

The hardware platform used to solve the challenge was a server with two AMD EPYC 9754 processors (containing a total of 256 cores implementing the Zen 4c microarchitecture) with dynamic frequency scaling (base clock 2.25 GHz, boost clock up to 3.1 GHz) and hyperthreading (for a total of 512 threads) enabled. Note that these timings are not exactly comparable to the estimates given in Table 1 since dynamic frequency scaling and hyperthreading were enabled at the time of the record run while they are disabled in Table 1.

References

[ADKF70] Владимир Л. Арлазаров, Ефим А. Диниц, Михаил А. Кронрод and Игорь А. Фараджев. “Об экономном построении транзитивного замыкания ориентированного графа”. In: *Доклады Академии Наук СССР* 194 (3 1970), pp. 487–488.

[Ber24] Daniel J. Bernstein. “Understanding binary-Goppa decoding”. In: *IACR Communications in Cryptology* 1.1 (2024). DOI: 10.62056/angy4fe-3.

[BLP08] Daniel J. Bernstein, Tanja Lange and Christiane Peters. “Attacking and Defending the McEliece Cryptosystem”. In: *PQCrypto 2008*. Vol. 5299. Lecture Notes in Computer Science. Springer, 2008, pp. 31–46. URL: https://ia.cr/2018/318.

[CME22r] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson and Wen Wang. *Classic McEliece: conservative code-based cryptography. Design rationale*. Tech. rep. 2022. URL: https://classic.mceliece.org/mceliece-rationale-20221023.pdf.

[Gib91] J. Keith Gibson. “Equivalent Goppa Codes and Trapdoors to McEliece’s Public Key Cryptosystem”. In: *EUROCRYPT 1991*. Vol. 547. Lecture Notes in Computer Science. Springer, 1991, pp. 517–521. DOI: 10.1007/3-540-46416-6_46.

[M4RI] Martin Albrecht and Gregory Bard. *The M4RI Library*. Version 20250128. The M4RI Team. 2025. URL: https://bitbucket.org/malb/m4ri.

[McE78] Robert J. McEliece. *A public-key cryptosystem based on algebraic coding theory*. The Deep Space Network Progress Report 42-44. 1978, pp. 114–116. URL: https://ipnpr.jpl.nasa.gov/progress_report/42-44/44N.PDF.

[Mor79] Oscar Moreno. “Symmetries of binary Goppa codes (Corresp.)” In: *IEEE Transactions on Information Theory* 25.5 (1979), pp. 609–612. DOI: 10.1109/tit.1979.1056089.

[Pat75] Nicholas J. Patterson. “The algebraic decoding of Goppa codes”. In: *IEEE Transactions on Information Theory* 21.2 (1975), pp. 203–207. DOI: 10.1109/TIT.1975.1055350.

[Sen00] Nicolas Sendrier. “Finding the permutation between equivalent linear codes: The support splitting algorithm”. In: *IEEE Transactions on Information Theory* 46.4 (2000), pp. 1193–1203. URL: https://inria.hal.science/inria-00073037/document.

[Sen97] Nicolas Sendrier. “On the Dimension of the Hull”. In: *SIAM Journal on Discrete Mathematics* 10.2 (1997), pp. 282–293. URL: https://inria.hal.science/inria-00074009/document.