A New Sieving-Style Information-Set Decoding Algorithm

Abstract. The problem of decoding random codes is a fundamental problem for code-based cryptography, including recent code-based candidates in the NIST post-quantum standardization process. In this paper, we present a novel sieving-style information-set decoding (ISD) algorithm, addressing the task of solving the syndrome decoding problem. Our approach involves maintaining a list of weight-$2p$ solution vectors to a partial syndrome decoding problem and then creating new vectors by identifying pairs of vectors that collide in p positions. By gradually increasing the parity-check condition by one and repeating this process iteratively, we find the final solution(s). We show that our novel algorithm performs better than other ISDs in the memory-restricted scenario when applied to McEliece. Notably, in the case of problems with very low relative weight, it seems to significantly outperform all previous algorithms. In particular, for code-based candidates BIKE and HQC, the algorithm has lower bit complexity than the previous best results.

Keywords: Code-based cryptography, NIST post-quantum standardization, Information-Set Decoding, Classic-McEliece, BIKE, HQC.

1 Introduction

The recent advancements in the development of quantum computers have greatly impacted cryptography. There is a threat to current standard cryptographic algorithms based on factoring and discrete-log problems, leading to an interest in cryptographic algorithms based on other hardness assumptions. Post-quantum cryptography revolves around primitives that are not known to be broken by a large quantum computer.

One leading and promising field in post-quantum cryptography is code-based cryptography. Being introduced already in the 70s, it has a long history with many proposed primitives that withstand classical as well as quantum attacks. Code-based cryptography relies on the difficulty of the problem of decoding random codes, which has been a very well-studied hardness assumption. The ongoing NIST standardization process for post-quantum cryptography [1] includes in round 4 several code-based proposals (Classic McEliece [11], BIKE [3], and HQC [30]).

One major challenge in these schemes is the selection of secure parameter sets for the proposals, which match the required security levels as decided by NIST. To determine and evaluate parameter sets, the exact cost of the best attacks on the proposed schemes and their corresponding hardness assumption is needed. Improving current ISD algorithms, as well as proposing new ones, are therefore of interest, and their complexity parameters, such as time and space, are important.

Code-based schemes usually rely on the hardness of decoding random codes, or equivalently, the *syndrome decoding problem*, which, given a random matrix $\mathbf{H} \in \mathbb{F}_2^{r \times n}$, a syndrome $\mathbf{s} \in \mathbb{F}_2^r$ and an integer ω asks to find an error vector $\mathbf{e} \in \mathbb{F}_2^n$ with weight ω such that $\mathbf{s} = \mathbf{He}$. The best algorithms to solve this problem belong to a class of algorithms known as information-set decoding (ISD). The first idea of an ISD algorithm was proposed by Prange in 1962 [32], and then a long line of papers have provided subsequent improvements, see [32,26,27,34,12,22,33,28,9] to mention a few. Recently, an approach called *statistical decoding* has been revisited by Carrier et al. [17], and shown to yield improvements in specific code rate regimes.

Most works study the problem for $\omega = cn$, where c is a constant, and investigate the asymptotic runtime exponent. However, for all code-based NIST PQC submissions, as well as other explicit proposals, the asymptotic expressions do not give the estimated complexity as numbers that can be translated to a security level. Some of the asymptotic advantages of improved ISD algorithms have been shown to more or less vanish for certain parameter sets. Therefore, it is not clear which algorithms actually yield practical improvements. We are left to study different expressions for the actual complexity of these algorithms. Another important aspect is that memory requirements are very high in the improved versions of ISD algorithms, and it is likely to be the limiting factor in practice. Hence any algorithm that requires less memory but a similar computational complexity is very relevant. Estimators for concrete complexity of solving the syndrome decoding problem for various algorithms have previously appeared in [24,6] and most recently in [20]. This last work includes an estimator program in Python that computes complexity numbers for many different algorithms and is the source for comparisons in our work.

1.1 Related works

An important ISD algorithm is the Stern algorithm [34] that significantly improved the previous work of Prange. Its slightly improved version using the parity-check matrix as suggested in [22] is used in our work. Other improvements making use of 'representation techniques', as in [28,9], are notable among *enumeration-dominated* ISD. The *nearest-neighbor search* was introduced in [29] and later used in various steps of the algorithms [15,16] .These improved versions of the Stern algorithm share a drawback: they generally require even larger memory, a bottleneck in many situations. Lattice sieving, a method of finding short vectors in a lattice [2,31,8], is an inspiration for our work. In our case, we are working with the Hamming metric. Our sieving method is, therefore, different from the known efficient lattice sieving methods due to the different metrics. A similar idea was also initiated by Bernstein in a cryptanalysis forum.[1]

[1] cryptanalytic-algorithms@list.cr.yp.to

1.2 Contributions

We propose a new ISD-like algorithm for solving the syndrome decoding problem, which we call *Sieving-Style ISD*. From the simple observation that if two weight-p vectors $\mathbf{x}, \mathbf{y}$ collide (i.e., both have a one) in $p/2$ positions (assuming p is even), then their sum is also a weight-p vector. Moreover, if we impose a '*syndrome condition*', being $\mathbf{Hx}, \mathbf{Hy} \in \{\mathbf{0}, \mathbf{s}\}$ for some syndrome $\mathbf{s}$, then again $\mathbf{H}(\mathbf{x}+\mathbf{y}) \in \{\mathbf{0}, \mathbf{s}\}$. Therefore, instead of using birthday-style arguments like in the Stern algorithm and its many subsequent improvements, we can construct new weight-p error vectors by combining in pairs stored weight p vectors, where the two vectors collide in $p/2$ positions. This procedure, together with an iterative increase in the number of considered syndrome positions, gives weight-p vectors fulfilling the syndrome equation.

Given a set $\mathcal{L}$ of small weight p vectors, we derive efficient algorithms for computing the new set of all weight-p vectors of the form $\mathbf{x}+\mathbf{y}$, where $\mathbf{x}, \mathbf{y} \in \mathcal{L}$. This is used as a part of the proposed ISD algorithm. We then analyze the concrete complexity of the proposed algorithm and make comparisons with existing best previous work when considering memory as well as computational complexity. We argue that our proposed algorithm has better time-memory trade-offs, especially when we restrict the memory. Hence, our algorithm can contribute significantly to understanding the concrete security of code-based cryptographic constructions and improve complexity numbers when the memory is limited.

When comparing the complexity to other ISD algorithms, there seems to be an improvement for instances with very low relative weight (in particular, for code-based candidates such as BIKE and HQC). In that case, the new algorithm outperforms all previous algorithms. In summary, we hope that our research enriches as a novel contribution to post-quantum cryptanalysis.

1.3 Organization

We start by giving preliminaries on coding theory and information-set decoding in Section 2. In Section 3, we explain the new ideas and describe all parts of the new algorithm. Section 4 presents the complete complexity analysis for the new algorithm. Section 5 then illustrates the performance by making comparisons with some of the best-known ISD algorithms for parameter choices selected from proposed schemes such as Classic McEliece, BIKE, and HQC. Section 6 gives some details and results from an actual algorithm implementation for small parameters, supporting the theoretical estimations. Section 7 concludes the paper.

2 Preliminaries

Throughout the paper, we use the following notations. We denote by

- bold letters, e.g., $\mathbf{v}$ and $\mathbf{H}$, row vectors and matrices. In particular, $\mathbf{I}_n$ denotes the identity matrix of size $n \times n$.

- $\omega_H(\mathbf{x})$ the Hamming weight of a vector $\mathbf{x}$.
- $\mathbf{x}+\mathbf{y}$ the bit-by-bit XOR between binary vectors $\mathbf{x}$ and $\mathbf{y}$.
- $\mathbb{F}_2$ the binary finite field and $\mathbb{F}_2^{m\times n}$ the vector space over $\mathbb{F}_2$ of dimension $m \times n$.
- log the logarithm base 2.
- $[i] := \{1,\ldots,i\}$ for an integer $i \in \mathbb{N}$.
- $\mathcal{O}(.)$ the usual Landau notation for the asymptotic behavior of algorithms, and $\tilde{\mathcal{O}}(.)$ means we suppress arbitrary polynomial factor.

We should also point out that all complexity expressions consider the actual complexity in the number of bit operations and not the corresponding asymptotic form of complexity expressions.

2.1 Linear codes and related hard problems

Let $\mathbb{F}_2^n$ be the vector space of all n-tuples over the finite field $\mathbb{F}_2$. A linear code, denoted by $\mathcal{C}$, is a vector subspace of $\mathbb{F}_2^n$. An element of the code $\mathbf{c} = (c_1,\ldots,c_n) \in \mathcal{C}$ where $c_i \in \mathbb{F}_2, i = 1,\cdots,n$ is called a *codeword*. If $\mathcal{C}$ is of dimension k, then we say it to be a $[n,k]$-linear code over $\mathbb{F}_2$. The *minimum distance* d of the code is defined as the minimum Hamming weight of nonzero codewords of $\mathcal{C}$.

A code $\mathcal{C}$ is often represented by a *generator* matrix which is a $k \times n$ binary matrix $\mathbf{G}$, where the rows constitute a basis of $\mathcal{C}$. Any set of k independent columns of $\mathbf{G}$ forms an *information set* of $\mathcal{C}$. It is also a common practice to denote the remaining coordinate, called *redundancy* of $\mathcal{C}$, by $r = n-k$. Another representation of a code is with a *parity check matrix*. In particular, there exists an $r \times n$ matrix $\mathbf{H}$ such that $\mathbf{H}\mathbf{c}^T = \mathbf{0}, \forall \mathbf{c} \in \mathcal{C}$. In general, there are many generator and parity check matrices for a code $\mathcal{C}$. When $\mathbf{G} = \begin{pmatrix}\mathbf{I}_k\ \mathbf{A}\end{pmatrix}$ or $\mathbf{H} = \begin{pmatrix}\mathbf{A}^{\mathrm{T}}\ \mathbf{I}_{n-k}\end{pmatrix}$, we say that they are in *systematic form*.

Let $\mathbf{y} \in \mathbb{F}_2^n$ be an arbitrary vector, we call $\mathbf{s} = \mathbf{H}\mathbf{y}^T \in \mathbb{F}_2^r$ the syndrome of $\mathbf{y}$ through $\mathbf{H}$. To ease the notation, we omit the transposition and write $\mathbf{y}$ instead of $\mathbf{y}^T$, and it should be clear from the context unless otherwise mentioned. We observe that if $\mathbf{y}$ is not a codeword of $\mathcal{C}$, i.e., $\mathbf{y} = \mathbf{c}+\mathbf{e}$, for some $\mathbf{c} \in \mathcal{C}$ and an "error vector" $\mathbf{e}$, then the syndrome of $\mathbf{y}$ is nonzero and $\mathbf{s} = \mathbf{H}\mathbf{y} = \mathbf{H}\mathbf{e}$.

Definition 1 *Let $\mathcal{C}$ be a $[n,k]$-linear code with a parity check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$. Given a noisy codeword $\mathbf{y} \in \mathbb{F}_2^n$, its syndrome $\mathbf{s} = \mathbf{H}\mathbf{y}$, and an integer $\omega > 0$, the syndrome decoding problem is to find an error vector $\mathbf{e} \in \mathbb{F}_2^n$ such that $\omega_H(\mathbf{e}) = \omega$, $\mathbf{y}+\mathbf{e} \in \mathcal{C}$, or equivalently $\mathbf{H}\mathbf{e} = \mathbf{s}$. We say that $\mathbf{e}$ solves the $(\mathbf{H},\mathbf{s},\omega)$ instance of the syndrome decoding problem.*

The syndrome decoding problem (SDP) is closely related to the *coset weights problem*, also known as the decisional syndrome decoding problem (DSDP), which has been shown to belong to the NP-complete complexity class by Berlekamp et al. [10].

Definition 2 *Let* $\mathbf{H}$ *be a random* $r \times n$ *matrix,* $\mathbf{s}$ *be a vector in* $\mathbb{F}_2^r$*, and* ω *be a positive integer. The coset weights problem is to determine if there exists a vector* $\mathbf{e} \in \mathbb{F}_2^n$ *such that* $\omega_H(\mathbf{e}) \leq \omega$ *and* $\mathbf{He} = \mathbf{s}$*.*

Although the search version is "harder" than the decisional variant, Arora et al., in [4], showed that they are polynomial-time equivalent, i.e., there exists a polynomial search-to-decision reduction. Therefore, it is common in the literature to say the SDP is NP-hard, despite the fact that the definition of NP applies to decisional problems.

The SDP has been a well-established problem in cryptography and coding theory for more than half a century. Throughout history, the NP-complete class of problems has been building blocks of cryptography. Similarly, the SDP has proven to be useful in constructing many cryptographic primitives. One can find numerous code-based constructions such as public-key cryptography [1,11,3,30], stream ciphers [23], hash functions [5,14], signatures [18], zero-knowledge protocols [35,36], etc., just to name a few. In particular, the current NIST standardization project for post-quantum public-key cryptosystems includes code-based constructions such as McEliece, BIKE, and HQC. With such importance, it is not surprising that extensive efforts have been made in cryptanalysis to gain trust in code-based primitives.

2.2 Information-Set Decoding Algorithms

The most prominent and well-studied approach to solving the Syndrome Decoding Problem is the class of so-called Information-Set Decoding (ISD) algorithms. In a naive attempt, one can search exhaustively through the space of error vectors with weight ω, which is $\binom{n}{\omega}$ and the complexity is $\tilde{\mathcal{O}}\left(\binom{n}{\omega}\right)$. There has been a long line of studies going back to Prange in 1962, who realized we could significantly improve this approach using simple linear algebra. Since then, ISD algorithms have remained an active field of research [32,26,27,34,19,12,22,33,28,9,29,15,16]. In the followings, we describe the general ISD framework and explain some of the technical details of relevant ISD algorithm variants. The essential idea of ISD algorithms is to reduce the search space's dimension with Gaussian elimination. In short, one applies a random permutation $\mathbf{P}$ as

$$\mathbf{He} = \mathbf{HPP^{-1}e} = \bar{\mathbf{H}}\bar{\mathbf{e}} = \mathbf{s}. \tag{1}$$

A Gaussian elimination process with some invertible matrix $\mathbf{Q} \in \mathbb{F}_2^{(n-k)\times(n-k)}$ results in

$$\mathbf{Q}\bar{\mathbf{H}}\bar{\mathbf{e}} = \left(\hat{\mathbf{H}}\ \mathbf{I}_{n-k}\right)\bar{\mathbf{e}} = \mathbf{Qs} = \bar{\mathbf{s}}. \tag{2}$$

Therefore, we can reconstruct a solution of $(\mathbf{H}, \mathbf{s}, \omega)$ by solving a new instance $(\mathbf{Q}\bar{\mathbf{H}}, \bar{\mathbf{s}}, \omega)$. The random permutation $\mathbf{P}$ imposes a particular weight distribution to $\bar{\mathbf{e}} = (\bar{\mathbf{e}}', \bar{\mathbf{e}}'') \in \mathbb{F}_2^k \times \mathbb{F}_2^{n-k}, \omega_H(\mathbf{e}') = p < \omega$. Therefore, equation (2) becomes

$$\hat{\mathbf{H}}\bar{\mathbf{e}}' + \bar{\mathbf{e}}'' = \bar{\mathbf{s}}. \tag{3}$$

In the original Prange's ISD algorithm, one looks for $\mathbf{P}$ that sends all the erroneous bits to the second part, i.e., corresponding to a case of $p = 0$ and $\bar{\mathbf{e}}'' = \bar{\mathbf{s}}$ (equivalently guessing the information-set of $\mathbf{H}$). Therefore, the running time of this algorithm is determined by finding a *correct* permutation, which happens with probability

$$\Pr_{\text{success}} = \frac{\binom{n-k}{\omega}}{\binom{n}{\omega}}. \tag{4}$$

Let $R = k/n$ be the code rate. Asymptotically, the running time of Prange's ISD converges to

$$T = \frac{1}{\Pr_{\text{success}}} \approx \left(\frac{1}{1-R}\right)^{\omega}. \tag{5}$$

Intuitively, Prange's ISD is suitable for the low-weight error regime as it is more likely that a random permutation will yield the desired weight distribution. Hence, the original ISD is still one of many main cryptanalysis tools to estimate the security of many code-based cryptosystems, most notably NIST post-quantum candidates such as McEliece, BIKE, or HQC public-key cryptosystems.

In contrast, many modern variants of ISD allow some error weight $p > 0$ outside the information set. Therefore, one looks for a weight-p vector $\bar{\mathbf{e}}'$ such that

$$\omega_H(\hat{\mathbf{H}}\bar{\mathbf{e}}' + \bar{\mathbf{s}}) = \omega - p. \tag{6}$$

Lee and Brickell [26] solved the above equation by simply *enumerating* $(\hat{\mathbf{H}}\bar{\mathbf{e}}' + \bar{\mathbf{s}})$ until a low weight $\bar{\mathbf{e}}''$ is found via (6). Leon in [27] improved this approach by imposing a ℓ-window of zeroes in $\bar{\mathbf{e}}''$; hence, the contribution from the first ℓ bits of $\bar{\mathbf{s}}$ comes only from $\mathbf{e}'$. In particular, we can write again as $\bar{\mathbf{e}} = \left(\bar{\mathbf{e}}', \mathbf{0}^{\ell}, \bar{\mathbf{e}}''\right) \in \mathbb{F}_2^k \times \mathbb{F}_2^{\ell} \times \mathbb{F}_2^{n-k-\ell}$. Although such a constraint reduces the probability of a good permutation, it offers a check via the equation

$$\hat{\mathbf{H}}_{[\ell]}\bar{\mathbf{e}}' = \bar{\mathbf{s}}_{[\ell]}. \tag{7}$$

It has been shown that such versions of ISD can not gain more than a polynomial factor compared to Prange's ISD.

The first asymptotic improvement came from the Stern ISD algorithm [34] by employing a *Meet-in-the-Middle* strategy to construct the candidates for equation (7). The strategy is to further split up $\bar{\mathbf{e}}' = \mathbf{e}_1 + \mathbf{e}_2$, where $\omega_H(\mathbf{e}_1) = \omega_H(\mathbf{e}_2) = p/2$. Moreover, this approach also mandates that $\mathbf{e}_1$ (and $\mathbf{e}_2$) contributes $p/2$ ones only among the left (right, respectively) $k/2$ coordinates. This is done by storing all $\binom{k/2}{p/2}$ possible values of $(\hat{\mathbf{H}}_{[\ell]}\mathbf{e}_1 + \bar{\mathbf{s}}_{[\ell]})$ in a look-up table and enumerating all possible values for $\hat{\mathbf{H}}_{[\ell]}\mathbf{e}_2$. We also notice that the Stern ISD algorithm was also the first variant to introduce a non-polynomial memory requirement, namely, a look-up table of size $\binom{k/2}{p/2}$.

Later, Finiasz et al. [22], and Dumer [19] argued that one can increase the success probability of each permutation by removing the window of ℓ-zeroes condition and allowing some error bits to that region. More specifically, instead of a full Gaussian elimination, one can apply a *partial* Gaussian elimination to (1) (with an additional parameter ℓ) and obtain the following form

$$\begin{pmatrix} \mathbf{H}' & \mathbf{0} \\ \mathbf{H}'' & \mathbf{I}_{n-k-\ell} \end{pmatrix} \bar{\mathbf{e}} = \begin{pmatrix} \mathbf{H}' & \mathbf{0} \\ \mathbf{H}'' & \mathbf{I}_{n-k-\ell} \end{pmatrix} \left(\bar{\mathbf{e}}' \; \bar{\mathbf{e}}''\right) = \bar{\mathbf{s}} = \left(\bar{\mathbf{s}}' \; \bar{\mathbf{s}}''\right), \tag{8}$$

where $\mathbf{H}' \in \mathbb{F}_2^{\ell \times (k+\ell)}, \mathbf{H}'' \in \mathbb{F}_2^{(n-k-\ell)\times(k+\ell)}, (\bar{\mathbf{e}}', \bar{\mathbf{e}}'') \in \mathbb{F}_2^{k+\ell} \times \mathbb{F}_2^{n-k-\ell}$. Then we proceed to find (almost) all solution for the 'small' syndrome decoding instance $(\mathbf{H}', \bar{\mathbf{s}}', p)$ in the form $\bar{\mathbf{e}}' = \mathbf{e_1} + \mathbf{e_2}$, where $\omega_H(\mathbf{e_1}) = \omega_H(\mathbf{e_2}) = p/2$ (in a similar manner as the Stern algorithm), i.e.,

$$\mathbf{H}'\mathbf{e}_1 + \mathbf{H}'\mathbf{e}_2 = \bar{\mathbf{s}}' \tag{9}$$

and then check for

$$\omega_H(\mathbf{H}''(\mathbf{e_1} + \mathbf{e_2}), \bar{\mathbf{s}}'') = \omega - p. \tag{10}$$

The equations (9) and (10) are sometimes called the *exact matching* and *approximate matching*, respectively, in literature. The state-of-the-art ISD algorithms such as MMT/BJMM [28,9] further speed up the process of constructing $\bar{\mathbf{e}}'$ via a *representation technique*. This practice allows more flexibility on how p error bits are presented in the vector $\bar{\mathbf{e}}'$. We refer the readers to the original works for more details of the representation technique. Subsequently, *Nearest neighbor search* [29] was introduced to amortize the cost of the approximate matching problem, e.g., as in [16].

In comparison with Prange original ISD, whose running time depends on the number of permutations one has to perform (with a polynomial factor for every iteration), enumeration-dominated ISD variants raise the success probability in (4) to

$$\Pr_{\text{success}} = \frac{\binom{n-k-\ell}{\omega-p}\binom{k+\ell}{p}}{\binom{n}{\omega}}. \tag{11}$$

Therefore, modern ISD variants are beneficial in the weight regime where a random permutation is not likely to send all the error weight to $\bar{\mathbf{e}}''$. For concrete security of code-based cryptosystems, enumeration-based ISD remains an essential cryptanalysis tool. However, asymptotically speaking, reducing the complexity of finding a good permutation and spending on enumerating on weight-p vector $\bar{\mathbf{e}}$ does not pay off.[2] Moreover, it comes at the cost of introducing significant memory overheads (and cost of accessing memory) owing to enumeration. Estimates based solely on the algorithmic steps can therefore lead to security underestimation of code-based cryptosystems. Hence, there has been skepticism among cryptographers as to how much modern ISD algorithms can improve code-based cryptanalysis, especially for cryptosystems of interest.

To this end, there have been comprehensive surveys of ISD algorithms such as Baldi et al. [6], Esser-Bellini [20], where concrete bit security estimates for code-based schemes are provided. Importantly, in their works, the memory access cost

[2] When n grows very large, optimal p is $p = 0$.

was taken into consideration to understand better the security of McEliece, HQC, and BIKE. Recently, Esser et al. [21] provided an efficient implementation of the MMT/BJMM algorithm (by deploying multiple techniques and speed-ups such as the *Parity bit trick, Method of the four Russians for Inversions*, and Decoding-one-out-of-Many (DOOM) [33]) with optimized parameters for McEliece and a quasi-cyclic setting. More notably, they also did cryptanalysis with medium-sized instances (60 bits). They showed that the data from their record computations could be used to extrapolate the bit-security of McEliece/HQC parameters in the NIST standardization process.

3 A new heuristic ISD algorithm

In this section, we describe in brevity the main steps of our new ISD algorithm.

3.1 Our ISD framework

We follow the same ISD framework presented in the previous section that derives Equation (8). As in (9), we are now assuming that the first part of the (permuted) error vector, $\bar{\mathbf{e}}'$, is of weight p. So we are looking for all weight-p vectors $\bar{\mathbf{e}}' \in \mathbb{F}_2^{k+\ell}$ that satisfy

$$\mathbf{H}'\bar{\mathbf{e}}' = \bar{\mathbf{s}}'. \tag{12}$$

Once such a vector is found, we can directly compute the corresponding $\bar{\mathbf{e}}''$ giving the desired syndrome and finally check whether the overall weight is ω. When no vector of weight ω is found, we apply a new random permutation, a new partial Gaussian elimination, and the procedure is repeated until success.

Continuing, we assume that the parity check matrix is already in the form of (8), and from now on, we assume that p is even. Hence, the weight $2p$ is used instead. Moreover, to ease the notation, we refer to matrix and vectors in (12) as $\mathbf{H}, \mathbf{e}$ and $\mathbf{s}$ instead. To summarize, we are searching weight $2p$ vectors $\mathbf{e} \in \mathbb{F}_2^{k+\ell}$ fulfilling

$$\mathbf{H}\mathbf{e} = \mathbf{s}, \tag{13}$$

where ℓ is a parameter giving the number of parity check equations used for the first part $\bar{\mathbf{e}}'$.

3.2 New ideas

The new idea behind our approach is to build an algorithm that keeps a list of weight-$2p$ vectors for which a part of the parity check equations are fulfilled. From this list, we create a new list of weight-$2p$ vectors for which an even larger number of the parity check equations are met. Iterating this procedure several times, we end up with a final list of weight-$2p$ vectors for which all considered parity checks are fulfilled.

We need to introduce some further notation for vectors. For any vector $\mathbf{v} \in \mathbb{F}_2^n$ of length n, it is written as $\mathbf{v} = (v_1, v_2, \ldots, v_n)$. The notation $\mathbf{v}_{[i]}$, $1 \leq i \leq n$, is defined as the projection of $\mathbf{v}$ onto the coordinates indexed by $[i]$. So $\mathbf{v}_{[1]}$ is (v_1); $\mathbf{v}_{[2]}$ is (v_1, v_2), and so on. A similar notation is adopted for matrices, where we let $\mathbf{H}_{[i]}$ denote the matrix restricted to the i first rows of $\mathbf{H}$.

We suggest the following algorithm for computing new weight $2p$ vectors from old weight $2p$ vectors based on a modified form of the sieving idea from computing short vectors in lattices:

Assume that $\mathbf{e}, \mathbf{f}$ are two weight-$2p$ vectors. If they collide in p positions, meaning that $e_i = f_i = 1$ for $i = \{i_1, i_2, \ldots, i_p\}$, then their sum is a new weight-$2p$ vector. In addition, we have one more restriction, namely that the new vector should fulfill a parity check equation. Recall that $\mathbf{s}_{[i]}$ is the syndrome $\mathbf{s}$ restricted to its first i positions. The parity check condition used in the first run is

$$\mathbf{H}_{[1]}\mathbf{e} \in \{0, \mathbf{s}_{[1]}\},$$

i.e., only weight-$2p$ vectors fulfilling this condition are kept. In the next run, the parity check will be considered up to the second coordinate, i.e., $\mathbf{H}_{[2]}\mathbf{e} \in \{\mathbf{0}, \mathbf{s}_{[2]}\}$ and so on.

The underlying observation is that if two vectors $\mathbf{e}_1, \mathbf{e}_2$ satisfy $\mathbf{H}_{[i]}\mathbf{e}_j \in \{\mathbf{0}, \mathbf{s}_{[i]}\}$ for $j = 1, 2$, then their sum will also have $\mathbf{H}_{[i]}(\mathbf{e}_1 + \mathbf{e}_2) \in \{\mathbf{0}, \mathbf{s}_{[i]}\}$. Therefore, $\mathbf{H}_{[i+1]}(\mathbf{e}_1 + \mathbf{e}_2) \in \{\mathbf{0}, \mathbf{s}_{[i+1]}\}$ is then fulfilled (when the newly added parity check is applied) with probability roughly one half.

Let us now explain and discuss the algorithmic description that is to be found in Algorithm 1 and Algorithm 2. This describe the inner parts of the full ISD algorithm.

Algorithm 1 Sieve_Syndrome_Dec

Input: Parity check matrix $\mathbf{H}$ with $k + \ell$ columns and ℓ rows, a length ℓ syndrome vector $\mathbf{s}$, the fixed weight $2p$ of the error vectors and an algorithm parameters M.
Output: A set of weight $2p$ vectors $\mathbf{e}$ such that $\mathbf{He} = \mathbf{s}$.

```
1 Initiate a set $\mathcal{L}_0$ with $M$ vectors of weight $2p$;
2 for $i = 1$ to $\ell$ do
3     Create the new set $\mathcal{L}_i \leftarrow \{\mathbf{e} \in \mathcal{L}_{i-1} : \mathbf{H}_{[i]}\mathbf{e} \in \{\mathbf{0}, \mathbf{s}_{[i]}\}\}$;
4     $\mathcal{M}_i \leftarrow$ Merge_Set$(\mathcal{L}_{i-1}, i)$;
5     $\mathcal{L}_i \leftarrow \mathcal{L}_i \cup \mathcal{M}_i$;
6 return $\mathcal{L}_\ell \setminus \{\mathbf{e} : \mathbf{He} = \mathbf{0}\}$;
```

The algorithm is centered around keeping a set $\mathcal{L}$ of M vectors of weight $2p$. In each iteration i, we aim to generate a new set of weight-$2p$ vectors with the same cardinality, where now one additional parity check equation from $\mathbf{He} = \mathbf{s}$ is fulfilled. On the one hand, this new set keeps the existing vectors in the set $\mathcal{L}_{i-1}$ (from the previous iteration), for which one more parity check is still valid (that keeps roughly half of them). On the other hand, we create new weight-$2p$

vectors by considering sums of any two vectors in $\mathcal{L}_{i-1}$, which hold the collision condition and fulfill the aforementioned parity check. This central part of the approach, called the Merge_Set subroutine, is extracted as Algorithm 2 and shall be discussed in detail later.

Algorithm 2 Merge_Set

Input: A parity check matrix $\mathbf{H} \in \mathbb{F}_2^{\ell \times (k+\ell)}$, a syndrome $\mathbf{s} \in \mathbb{F}_2^{\ell}$, an integer i, and a set $\mathcal{L} = \{\mathbf{e} \in \mathbb{F}_2^{k+\ell} : \mathbf{H}_{[i-1]}\mathbf{e} \in \{\mathbf{0}, \mathbf{s}_{[i-1]}\}\}$ of size M.
Output: A set $\mathcal{M}$ of vectors of weight $2p$ such that for $\mathbf{e} \in \mathcal{M}$ we have $\mathbf{H}_{[i]}\mathbf{e} \in \{\mathbf{0}, \mathbf{s}_{[i]}\}$.

```
1 Initiate a set $\mathcal{M} \leftarrow \emptyset$;
2 for $\mathbf{e}, \mathbf{e}' \in \mathcal{L}$ do
3     If $w_H(\mathbf{e} + \mathbf{e}') = 2p$ then $\mathcal{M} \leftarrow \mathcal{M} \cup (\mathbf{e} + \mathbf{e}')$
4 return $\mathcal{M} = \{\mathbf{e} \in \mathcal{M} : \mathbf{H}_{[i]}\mathbf{e} \in \{\mathbf{0}, \mathbf{s}_{[i]}\}\}$;
```

The Merge_Set subroutine is called ℓ times, corresponding to the number of parity-check equations that need to be satisfied. Note that the parity check condition in the previous iteration will also be valid in the next. Therefore, we eventually have a 'candidate' list of weight-$2p$ error vectors that match the ℓ bits of syndrome $\mathbf{s}$. Such candidates are subsequently tested for the approximate matching condition as in (10). Putting everything together, we have a high-level description of our Sieving-style ISD algorithm as in Algorithm 3.

Algorithm 3 Full_ISD

Input: Matrix $\mathbf{H}$ with k rows and n columns, received length n vector $\mathbf{y}$, minimum weight ω and algorithm parameter ℓ.
Output: A weight-ω vector $\mathbf{e}$ such that $\mathbf{Hy} = \mathbf{He}$.

```
1 Compute the syndrome $\mathbf{s} = \mathbf{Hy}$;
2 repeat
3     Pick a random column permutation $\mathbf{P}$;
4     Perform Gaussian elimination on $\mathbf{PH}$ resulting in
      $\hat{\mathbf{H}} = \begin{pmatrix} \mathbf{H}' & 0 \\ \mathbf{H}'' & \mathbf{I}_{n-k-\ell} \end{pmatrix} (\bar{\mathbf{e}}' \ \bar{\mathbf{e}}'') = \bar{\mathbf{s}} = (\bar{\mathbf{s}}' \ \bar{\mathbf{s}}'')$;
5     Let $\mathbf{H}' = \hat{\mathbf{H}}_{[\ell]}$ and $\bar{\mathbf{s}}' = \bar{\mathbf{s}}_{[\ell]}$;
6     $\mathcal{L} \leftarrow$ Sieve_Syndrome_Dec$(\mathbf{H}', \bar{\mathbf{s}}', 2p)$;
7     for $\mathbf{e} \in \mathcal{L}$ do
8         if $\omega_H(\mathbf{H}''\mathbf{e} + \mathbf{s}) = \omega_H(\mathbf{e}'') = \omega - 2p$ then return $\mathbf{P}^{-1}(\mathbf{e}, \mathbf{e}'')$
9 until solution is found
```

3.3 The Merge_Set algorithm

As introduced above, the Merge_Set algorithm operates on a set of weight-$2p$ vectors and should return any weight-$2p$ sum of two such vectors. There is an additional parity check requirement, but since this is valid for half of the vectors, it does not pose a problem. We simply check for each sum vector of weight $2p$.

In short, the problem is to find an efficient way of generating pairs of vectors that sum to a new weight-$2p$ vector. A naive implementation of Algorithm 2 would require checking all pairs of vectors, hence requiring quadratic (in list size) complexity.

Let a (low weight) vector $\mathbf{e}$ be represented by the indices of its ones, i.e., $(i_1, i_2, \ldots, i_{2p})$ in rising order, written $\mathbf{e} \sim (i_1, i_2, \ldots, i_{2p})$. We want to find two vectors that share p indices. In a first attempt to find an efficient solution, we could generate $\binom{2p}{p}$ 'labels' for each vector. A label would be a selection of p out of the $2p$ indices for the vector. With M vectors in total, we would have $\binom{2p}{p} \cdot M$ such labels. They would then be stored in a sorted way so that collisions among them are detected. Labels of the form $(i_1, i_2, \ldots, i_p)$ can be mapped to integers and, with a hash table, one can then get close to complexity $\binom{2p}{p} \cdot M$ and the same memory.

Algorithm 4 Merge_Set_Implementation0

```
Input: A parity check matrix $\mathbf{H} \in \mathbb{F}_2^{\ell \times (k+\ell)}$, a syndrome $\mathbf{s} \in \mathbb{F}_2^{\ell}$, an integer $i$,
and a set $\mathcal{L} = \{\mathbf{e} \in \mathbb{F}_2^{k+\ell} : \mathbf{H}_{[i-1]}\mathbf{e} \in \{\mathbf{0}, \mathbf{s}_{[i-1]}\}\}$ of size $M$. Parameters $p', p''$.
Output: A set $\mathcal{M}$ of vectors of weight $2p$ such that for $\mathbf{e} \in \mathcal{M}$ we have
$\mathbf{H}_{[i]}\mathbf{e} \in \{\mathbf{0}, \mathbf{s}_{[i]}\}$.
1 Declare and initiate parameter (set of vectors) $\mathcal{M} \leftarrow \emptyset$;
2 Find_Collision($\mathcal{L}$, $p$, $p'$,1);
3 return $\mathcal{M} = \{\mathbf{e} \in \mathcal{M} : \mathbf{H}_{[i]}\mathbf{e} \in \{\mathbf{0}, \mathbf{s}_{[i]}\}\}$;
```

However, we propose an even more efficient implementation, where we, in particular, reduce the amount of memory. This approach is described in Algorithm 4 together with Algorithm 5. For the latter, we use recursive calls to ease the description of the procedure. We first describe the basic ideas of the procedure, and later we revisit the exact steps of Algorithm 4 and Algorithm 5.

We split p (and vectors, correspondingly) into two parts as $p = p' + p''$. Each vector given by $(i_1, i_2, \ldots, i_{2p})$ parses into $(i_1, i_2, \ldots, i_{p'})$ as a first part and $(i_{p'+1}, \ldots, i_{2p})$ as a second part. We consider the set $\mathcal{L}$ of length-$(k+\ell)$ and weight-$2p$ vectors to be arranged in a number of 'buckets', where each bucket initially contains the vectors, of which the first part is $(i_1, i_2, \ldots, i_{p'})$.

It means that the number of buckets is $\binom{k+\ell-p''}{p'}$. Note that each vector is only in one bucket. Furthermore, p' should be chosen in such a way that it is likely that there will occur some collision inside each bucket. Now we consider the first bucket, indexed by $(1, 2, \ldots, p')$. The vectors in this bucket already collide in p' positions, and we seek pairs of vectors that collide in an additional p'' positions out of the $2p - p'$ remaining ones. This is done in the following way. We assume we have access to an array $\mathbf{A}$ of size $\binom{k+\ell-p'}{p''}$, indexed by p'' positions. For each vector in the bucket, we create the $\binom{2p-p'}{p''}$ different possible combinations of the remaining p'' positions and write a one in the corresponding position in $\mathbf{A}$. Also, if there was already a one in that position, we have found a collision, and it is recorded. Finally, after all collisions in a bucket are found, the vectors are placed in their 'next bucket', which is the bucket indexed by the next value for the p' positions.

We now elaborate this idea by describing the procedure in a recursive way as in Algorithm 5. To give a brief explanation, it starts with a call to $\mathsf{Find\_Collision}()$, looking for collisions in p positions. It has a bucket (list) of vectors as input. These vectors are now placed in new buckets, depending on the vector's first index value. In bucket $\mathcal{B}_1$, all vectors have a one in position 1, so within $\mathcal{B}_1$, we only need to look for collisions in $p - 1$ additional positions. Therefore, the call to $\mathsf{Find\_Collision}(\mathcal{B}_1, p-1, p'-1, i+1)$. Once this call has returned possible collisions, the vectors in $\mathcal{B}_1$ may still collide in other ways, excluding position 1. This is why we then move the vectors to the next bucket corresponding to the second lowest index in the vector. Since the position 1 was removed from further combinations, the vector now has only $2p - 1$ indices. Let us assign an index x where the vectors are considered to 'start'.

If the remaining depth is not zero (checked in Line 1), we are simply going to put the vectors in different buckets $\mathcal{B}_{x+1}, \ldots, \mathcal{B}_{k+\ell}$ depending on their next index that is greater than x. For instance, if the next index in order is y, the vector is put in bucket $\mathcal{B}_y$ (Line 2). Then we go through all these buckets in order and find all collisions in bucket $\mathcal{B}_i$ by the call $\mathsf{Find\_Collision}(\mathcal{B}_i, p-1, p'-1, i+1)$ (Line 4). Note that since all vectors in bucket $\mathcal{B}_i$ already collide in position i, we decrease the depth and required collision while increasing the index by one.

Once all collisions in $\mathcal{B}_i$ have been found, these vectors may provide further collisions in indices i. Thus, we must move the vectors in $\mathcal{B}_i$ to the next bucket corresponding to the next index that is greater than i. This is done according to Line 5.

When the remaining depth is 0, there are not enough vectors in the input bucket to further motivate a split in smaller buckets. Instead, we now directly find the collisions. For this purpose, we use an array $\mathbf{A}$, indexed by p''-tuples. For each vector, we create all possible p''-tuples of its remaining indices $(i_j, i_{j+1}, \ldots, i_{2p})$ and we write up $\mathbf{A}$ by one in each such position (Line 11). We also keep the address to the vector $\mathbf{v}$ in an array $\mathbf{D}$ where we assume that in each entry, we can store a few elements (Line 14). While updating the array, one may hit an index where $\mathbf{A}$ is already non-zero (meaning collisions). One directly writes them to the global output parameter $\mathcal{M}$.

Example 1. We can visualize the checking step and $\mathsf{Merge\_Set}$ (Lines **3** and **4** in Algorithm 1) by Figure 1. For simplicity, let $p = 3$, $p' = 1$, and $p'' = 2$. In the

Algorithm 5 Find_Collision()

Input: A set $\mathcal{L}$ of vectors of length $k+\ell$; collision weight p; depth sizes p'; first index x.

Output: All vectors of the form $\mathbf{x}+\mathbf{y}$, where $\mathbf{x},\mathbf{y} \in \mathcal{L}$ and they collide in p positions, written to global parameter $\mathcal{M}$.

```
1  if $p' > 0$ then
2      Put the vectors in $\mathcal{L}$ in new buckets $\mathcal{B}_{x+1}, \dots, \mathcal{B}_{k+\ell}$ depending on
       its first index greater than $x$
3      for $i = x+1 \dots k+\ell$ do
4          Find_Collision($\mathcal{B}_i, p-1, p'-1, i+1$)
5          Move the vectors in $\mathcal{B}_i$ to new buckets in $\mathcal{B}_{i+1}, \dots, \mathcal{B}_{k+\ell}$
           depending on its first index greater than $i$
6  else
7      Initiate two arrays $\mathbf{A} \leftarrow 0$, $\mathbf{D} \leftarrow 0$
8      for each vector $\mathbf{v} \sim (i_j, i_{j+1}, \dots, i_{2p})$, $j > x$, in $\mathcal{B}_i$ do
9          create a set $\mathcal{Y}$ of all its $p''$-tuples.
10         for each $p''$-tuple $\mathbf{y} = (y_1, y_2, \dots, y_{p''}) \in \mathcal{Y}$ do
11             $\mathbf{A}[\mathbf{y}] \leftarrow \mathbf{A}[\mathbf{y}] + 1$
12             if $\mathbf{A}[\mathbf{y}] \geq 2$ then
13                 store $\mathbf{v} + \mathbf{D}[\mathbf{y}]$ as collisions in $\mathcal{M}$
14             $\mathbf{D}[\mathbf{y}] \leftarrow \mathbf{D}[\mathbf{y}] \cup \{\mathbf{v}\}$
```

i-th iteration, we have a list $\mathcal{L}_i$ of vectors. First, we put vectors in $\mathcal{L}_i$ in buckets corresponding to their first coordinate. Assume we have $\mathbf{x}_j, \mathbf{x}_k, \mathbf{x}_\ell \in \mathcal{L}_i$ where $\mathbf{x}_j \sim (j_1, \dots, j_{2p})$ (and so forth), and they have the same first coordinate, i.e., they are put in $\mathcal{B}_{j_1}$. Then we only need to proceed with their shortened versions, written as $\mathbf{x}_j^* \sim (j_2, \dots, j_{2p})$, etc., since we have excluded the first coordinate. We then detect collisions in this 'bucket' by producing p'' labels for each vector and marking them on $\mathbf{A}$ correspondingly. For example, if both $\mathbf{x}_j^*, \mathbf{x}_k^*$ include (j_2, j_3), then we potentially have $\mathbf{x}_j + \mathbf{x_k}$ as a 'good' combination to be added in $\mathcal{L}_{i+1}$.

After processing $\mathcal{B}_{i_1}$, we move (dashed red line) vectors in this bucket to their next buckets. For instance, $\mathbf{x}_i$ to $\mathcal{B}_{i_2}$, $\mathbf{x}_j$ to $\mathcal{B}_{j_2}$ and so forth. We now exclude the first two coordinates of $\mathbf{x}_j$ (hence, we use $\mathbf{x}_j^{**} \sim (j_3, \dots, j_{2p})$). Note that in the list $\mathcal{L}_{i+1}$, we also have half the vectors from $\mathcal{L}_i$ that survive the syndrome condition.

4 Analysis of the new ISD algorithm

This section provides estimations on the time complexity, denoted C, and the space complexity. The space is essentially the number of stored vectors M (so it

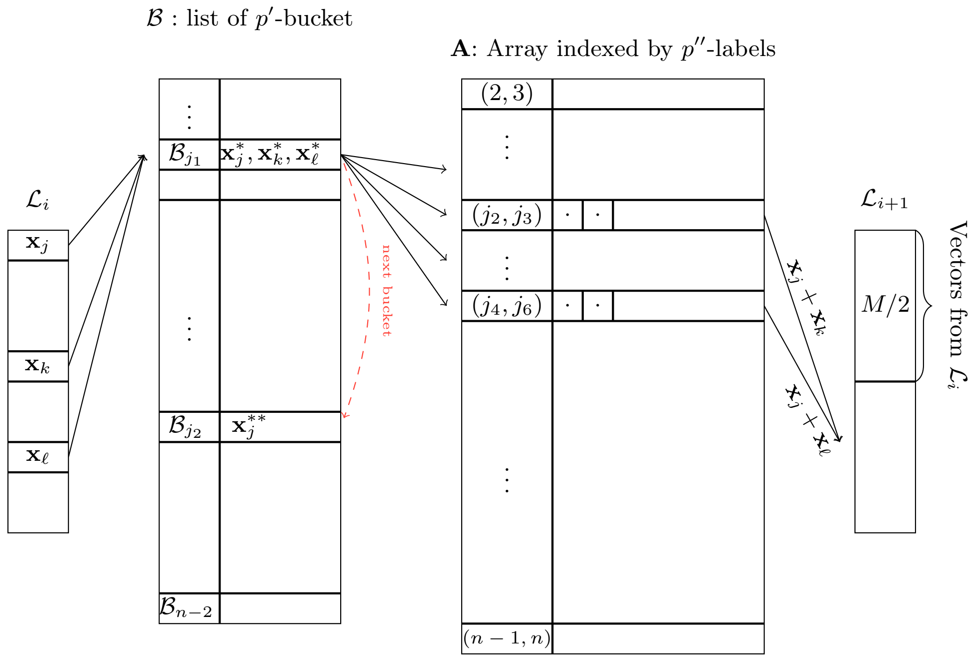


Fig. 1: Checking vector in $\mathcal{L}_i$ and Merge_Set.

is not given in bits). Some smaller additional memory is required for other parts of the algorithm.

4.1 Memory requirements and parameters selection

We first determine the list size M required for the new algorithm to work. Let us recall that the inner iteration of our ISD algorithm, i.e., the Sieve Syndrome Decoding (Algorithm 1), consists of two steps: the Merge_Set subroutine, and verifying the next parity check for vectors in the list that we are processing. Assume that we initiate Algorithm 1 with a list $\mathcal{L}_0$ where $|\mathcal{L}_0| = M$ and we aim to keep the list size constant after every (or the majority of) iteration of the parity check condition. At the i-th iteration, one has for each $\mathbf{e} \in \mathcal{L}_i$ that

$$\omega_H(\mathbf{e}) = 2p, \text{ and } \mathbf{H}_{[i]}\mathbf{e} \in \{\mathbf{0}, \mathbf{s}_{[i]}\}.$$

We observe that, on average, half of them shall satisfy the next parity-check condition, i.e., $\mathbf{H}_{[i+1]}\mathbf{e} \in \{\mathbf{0}, \mathbf{s}_{[i+1]}\}$. Therefore, we choose M that yields another $M/2$ 'good' combinations. We denote the probability of two random weight-$2p$ vectors of length $k+\ell$ colliding in precisely p positions (of the ones) by q, then

$$q = \frac{\binom{2p}{p}\binom{k+\ell-2p}{p}}{\binom{k+\ell}{2p}}.$$

Given a list of M vectors, we can form $\frac{M(M-1)}{2} \approx \frac{M^2}{2}$ combinations. However, as will be explained later, a significant part of our vectors are actually not dependent. Two phenomena arise: 1) We have *more* combinations than the uniformly random case, and 2)Combinations can be already existing vectors (called duplicates). For this purpose, we introduce δ as the fraction of all combinations that give rise to *new* vectors. Continuing, on average, new weight-$2p$ vectors survive the parity check with probability $\frac{1}{2}$. In conclusion, we require

$$\frac{\delta \cdot M^2 \cdot q}{2 \cdot 2} \approx \frac{M}{2}$$

or

$$M \approx \frac{2}{\delta \cdot q}. \tag{14}$$

Let us define

$$N = \{\mathbf{e} \in \mathbb{F}_2^{k+\ell} | \omega_H(\mathbf{e}) = 2p \text{ and } \mathbf{He} = \mathbf{s}\},$$

which is the number of solutions for the exact matching equation (9) (not to be confused with the original syndrome decoding problem). One can expect that the cardinality of N is around

$$\frac{\binom{k+\ell}{2p}}{2^\ell}.$$

The final list of Sieve_Syndrome_Dec contains around $M/2$ solutions of the exact matching equation (the other half yields null syndrome). If ℓ is not too large, there will be many possible solutions, and they all need to be stored in the final list. Therefore, to guarantee that our ISD algorithm is able to retrieve all (or most) solutions of the exact matching problem, we need that

$$M \geq \frac{\binom{k+\ell}{2p}}{2^{\ell-1}}. \tag{15}$$

In conclusion, the list size is first set by (14). Then, to find the optimal parameters for our algorithm, we search for $p \in [0, \omega]$, and ℓ in a 'reasonable' range[3], so that (15) holds, and we select the parameters that yield the lowest complexity.

4.2 Heuristic arguments for duplicated vectors

In this subsection, we investigate the fact that due to dependency in the our list of vectors, there will be some combinations that do not contribute to the new list. In other words, we try to heuristically determine δ in Equation (14), as well as the number of combinations we have to compute in each iteration.

Let us look at three consecutive iterations in Sieve_Syndrome_Dec as in Figure 2. Vectors in $\mathcal{L}_i$ can be split into two sets: $\mathcal{L}_{i,1}$ as the set of vectors from

[3] Similar to Baldi et al. in [6]. We extend the range of ℓ until the optimal value of ℓ is no longer on the edge of the range.

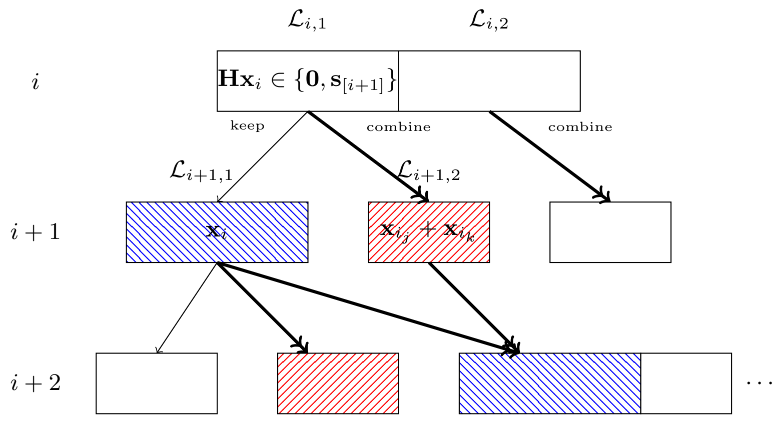


Fig. 2: An example of duplicates

the i-th iteration that also fulfill the syndrome condition up to iteration $i+1$, and $\mathcal{L}_{i,2}$ as the rest. Next, the vectors in iteration $i+1$ are made of three sets $\mathcal{L}_{i+1,j}, j = 1,2,3$. Vectors from $\mathcal{L}_{i+1,1} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots\}$ which is a set of size $\frac{M}{2}$ (directly kept from $\mathcal{L}_{i,1}$); $\mathcal{L}_{i+1,2}$ as sums of two vectors both from $\mathcal{L}_{i,1}$ which is a set of size $\frac{M}{4}$; finally, $\mathcal{L}_{i+1,3}$ as sums of two vectors both from $\mathcal{L}_{i,2}$ which is also a set of size roughly $\frac{M}{4}$. Note that there can be no sum of one vector from $\mathcal{L}_{i,1}$ and one from $\mathcal{L}_{i,2}$, as then the syndrome condition is not fulfilled.

We now look at all the different combinations that give already existing vectors. First, any combination of two vectors from $\mathcal{L}_{i+1,1}$ already exists in $\mathcal{L}_{i+1,2}$. They will account for $(\frac{M}{2})^2 \cdot \frac{q}{2} = \frac{M^2 \cdot q}{8}$ combinations that do not contribute. Then there are also duplicates when combining $\mathcal{L}_{i+1,1}$ and $\mathcal{L}_{i+1,2}$. When a vector $\mathbf{x}_{i_1} + \mathbf{x}_{i_2} \in \mathcal{L}_{i+1,2}$ is added to either $\mathbf{x}_{i_1} \in \mathcal{L}_{i+1,1}$ or $\mathbf{x}_{i_2} \in \mathcal{L}_{i+1,2}$, there will be a duplicate. Since $|\mathcal{L}_{i+1,2}| = \frac{M}{4}$, and for each $\mathbf{x}_{i_1} + \mathbf{x}_{i_2}$, we can have two duplicates which are $\mathbf{x}_{i_1}$ and $\mathbf{x}_{i_2}$. Therefore, the number of generated duplicates is of order $\frac{M}{2}$. Then we may also have additional duplicates from other combinations.

Again, besides the 'useless' combinations from $\mathcal{L}_{i+1,1}$ and $\mathcal{L}_{i+1,2}$, we also have other additions that can give new vectors. In particular, the type $\mathbf{x}_{i_1} + \mathbf{x}_{i_2} + \mathbf{x}_\mathbf{j}$ where $\mathbf{x}_{i_1} + \mathbf{x}_{i_2} \in \mathcal{L}_{i+1,1}$ and $\mathbf{x}_\mathbf{j} \in \mathcal{L}_{i+1,2}, \mathbf{x}_j \neq \mathbf{x}_{i_1}, \mathbf{x}_{i_2}$. Additions of this kind yield approximately $\frac{M}{4} \cdot (\frac{M}{2} - 2) \cdot q \approx \frac{M^2 \cdot q}{8}$ new vectors. To summarize, it is necessary that the total number of combinations is around M, excluding the duplicates. In other words, one has

$$M^2 \cdot q \cdot \left(\underbrace{\frac{1}{8}}_{(1,1)} + \underbrace{\frac{1}{8}}_{(1,3)} + \underbrace{\frac{1}{32}}_{(2,2)} + \underbrace{\frac{1}{16}}_{(2,3)} + \underbrace{\frac{1}{32}}_{(3,3)} \right) + \underbrace{\left(\frac{M}{2} + \frac{M^2 \cdot q}{8} \right)}_{(1,2)} - \underbrace{\left(\frac{M}{2} + \frac{M^2 \cdot q}{8} \right)}_{\text{duplicates}} \approx M,$$

where (j,k) represents the combinations between $\mathcal{L}_{i+1,j}$ and $\mathcal{L}_{i+1,k}$. Simplifying, one obtains

$$M \approx \frac{8}{3 \cdot q} = \frac{4}{3} \cdot \frac{2}{q}. \tag{16}$$

This corresponds to selecting $\delta = \frac{3}{4}$ in Equation (14). However, this is not sufficiently small due to the other (rarer) duplicates. To be conservative, we choose $\delta = \frac{2}{3}$, and it is more than sufficient according to simulations.

The number of duplicates is then *at least* (excluding other rarer cases)

$$\frac{M}{2} + \frac{M^2 \cdot q}{8} = \frac{7 \cdot M}{8}$$

We are motivated by this heuristic estimate and expect (conservatively) to have to create around $2 \cdot M$ combinations for each iteration. Note that this estimate is not necessarily true once the number of possible weight-$2p$ vectors decreases in later iterations (in a sense, this estimate allows us to be on the safe side in (many) early iterations by maintaining the list size).

Example 2. We supported our heuristic arguments of δ with simulations. We test various sets of parameters, and simulations confirm the heuristic arguments. We stop Merge_Set once we observe that the list size is maintained, and we look at the number of total combinations we have done. For example, two (out of many tested) parameter sets are $(k,\ell,p) = (500,30,2)$ and $(k,\ell,p) = (1000,30,2)$. We record the total amount of collisions and duplicates for each iteration.

- For $(k,\ell,p) = (1000,30,2)$, we have $M \approx 2^{15.35}$. For the majority of iterations, we obtain M unique vectors (hence, $M/2$ survive after the check). The ratio between duplicates and M varies around 7/8 and peaks at 0.93 (i.e. we create at most $0.93 \cdot M$ duplicates).
- For $(k,\ell,p) = (500,30,2)$, we have $M \approx 2^{13.43}$. We observe similar behavior, and the ratio between duplicates and M peaks at 1.

4.3 The probability of finding a desired vector

We next provide some heuristic arguments concerning the probability of finding one or several desired vectors, i.e., if the code contains a weight-$2p$ codeword, what is the probability that it is included in the list given as output from Sieve_Syndrome_Dec?

Recall the assumption that, throughout the Sieve_Syndrome_Dec, we have M unique vectors moved from one iteration to the next. However, when i is large enough, this will no longer be true. We now introduce M'_i as the expected number of weight-$2p$ vectors that fulfill up to i parity checks conditions. Then

$$M'_0 = \binom{k+\ell}{2p}$$

and

$$M'_i = \frac{\binom{k+\ell}{2p}}{2^i}.$$

Note that we also have the same amount of weight $2p$ vectors that fulfill the null syndrome $\mathbf{0}_{[i]}$.

Now the heuristic argument is that the set of generated vectors in iteration i is a random selection among all M'_i vectors. So for each created vector in the iteration, we view it as a random pick. Let $M_i = |\mathcal{L}_i|$ denote the list size in iteration i, where $M_i \leq M$. Then the M_i vectors come from the primary check and Merge_Set (Lines 3 and 4 in Algorithm 1). We denote the cardinality of these two sets by $M_i^{(1)}$ and $M_i^{(2)}$, respectively. Then,

$$M_i = \min\left(M, M_i^{(1)} + M_i^{(2)}\right).$$

The primary check contributes, on average, $M_i^{(1)} = M_{i-1}/2$ distinct vectors from the previous iteration.

We now estimate $M_i^{(2)}$ as the expected number of **unique new vectors** from Merge_Set. Intuitively, when $M'_i \gg M$, it is unlikely that we will generate the same vector twice or more, and we have a high chance to reach $M_i = M$ new vectors for the next iteration. However, when M'_i gets closer to M as i grows, we are forced to have more duplicates, and Merge_Set we will not generate $M/2$ new vectors.

As previously discussed, after excluding the obvious obsolete combinations, Merge_Set creates $\frac{3M_{i-1}^2 \cdot q}{8}$ more combinations in iteration i. With the choice of $\delta = 2/3$ to ensure that we expect to generate more new vectors than needed. We have an expected number of $\frac{9 \cdot M_{i-1}^2}{16 \cdot M}$ new vectors.

A vector is unique if it is not among the $M_i^{(1)}$ vectors in the first part and not the same as any previously kept one. Hence, the first vector has a probability $1 - \frac{M_i^{(1)}}{2 \cdot M'_i}$ of being unique, and the second vector has a probability larger than $1 - \frac{M_i^{(1)}+1}{2 \cdot M'_i}$, and so on. In total, the expected number of unique vectors is estimated at around

$$\frac{9 \cdot M_{i-1}^2}{16 \cdot M} - \frac{M_i^{(1)} + (M_i^{(1)} + 1) + \dots}{2 \cdot M'_i} \approx \frac{9 \cdot M_{i-1}^2}{16 \cdot M}\left(1 - \frac{M_i^{(1)} + \frac{9 \cdot M_{i-1}^2}{32 \cdot M}}{2 \cdot M'_i}\right).$$

We expect $M_i^{(2)}$ to be the minimum of $M/2$ and the above expression.

Now assume $\mathbf{e}$ is a desired weight-$2p$ vector that fulfills ℓ parity checks. Then we know that if $\mathbf{e}$ has appeared in an iteration i, it continues to be present in all subsequent iterations $j \geq i$. Recall that we initialize Sieve_Syndrome_Dec with a list of size M. The probability that $\mathbf{e}$ is not randomly selected is

$$\left(1 - \frac{1}{M'_0}\right)^M.$$

For $i = 1, \cdots, \ell$, as the primary check does not produce new vectors, then $\mathbf{e}$ is not present after each iteration if it is not produced from Merge_Set. This routine produces ${M_i}^{(2)}$ more vectors; hence the probability is

$$\left(1 - \frac{1}{2 \cdot M_2'}\right)^{M_i^{(2)}}$$

where the factor 2 can be explained by: the newly created vectors can be those whose syndromes are either $\mathbf{s}_{[i]}$ or $\mathbf{0}_{[i]}$. Therefore, the probability that $\mathbf{e}$ is not found after Sieve_Syndrome_Dec is

$$\left(1 - \frac{1}{M_0'}\right)^{M} \cdot \prod_{i=1}^{\ell} \left(1 - \frac{1}{2 \cdot M_i'}\right)^{M_i^{(2)}} .$$

In other words, our algorithm finds $\mathbf{e}$ with probability

$$1 - \left(1 - \frac{1}{M_0'}\right)^{M} \cdot \prod_{i=1}^{\ell} \left(1 - \frac{1}{2 \cdot M_i'}\right)^{M_i^{(2)}} .$$

We stress that the quantities above are, for a large part, heuristic estimates for the expected number of vectors in Sieve_Syndrome_Dec; hence, the mathematics is not rigorous. In fact, from simulation, we can see that we slightly overestimate $M_i^{(2)}$ and underestimate the probability calculation. However, we can use the expressions to roughly estimate the desired probability for cases where we cannot simulate. If we do that, we observe that the probability typically lies in the range $50 - 100\%$. In section 6, we give some examples from implementations that show that the above heuristic approach is somewhat reasonable.

4.4 Complexity Estimation

We study the complexity in the RAM model, i.e., the cost of reading and writing to one memory address is $\mathcal{O}(1)$ operations, with the memory access cost set to 1. This method is the most traditional way of estimating the complexity, used in many previous papers and also in the complexity estimator given in [20].

Outer iterations Let us recall that the probability that a permutation yields the correct weight distribution, that is, $2p$ in the first $k + \ell$ bits and $\omega - 2p$ in the remaining $n - k - \ell$ bits, is

$$\Pr_{\text{success}} = \frac{\binom{k+\ell}{2p}\binom{n-k-\ell}{w-2p}}{\binom{k+r}{w}}.$$

Therefore, we have to perform, on average, $\frac{1}{\Pr_{\text{success}}}$ iterations. We subsequently examine the cost for each iteration, denoted by C_{iter}.

This probability can be adjusted in two ways. On the one hand, the probability of actually finding a valid vector was argued for in Subsection 4.3. If ℓ is large enough, it was indicated that this probability is mostly larger than 0.5. On the other hand, the parity trick in the Gaussian elimination part, explained in [21], can force the weight of all codewords to be even and then $\Pr_{\text{success}}$ increases by a factor around 2. We adopt the approximation that these two factors cancel each other out.

Gaussian Elimination The following is often referred to as the FS-ISD framework [22]. Firstly, we perform a partial Gaussian elimination on the parity check matrix,

$$\hat{\mathbf{H}} = \begin{pmatrix} \mathbf{H}' & \mathbf{0} \\ \mathbf{H}'' & \mathbf{I}_{n-k-\ell} \end{pmatrix},$$

where $\mathbf{H}'$ is a matrix of dimension $\ell \times (k+\ell)$, $\mathbf{0}$ is an all-zero matrix, and $\mathbf{I}_{n-k-\ell}$ is the identity matrix with dimension $(r-\ell) \times (r-\ell)$, where $r = n-k$.

Similarly to recent ISD analysis works [20,21], we employ the *Method of Four Russian* for Gaussian Elimination, which was proposed in [13,12]. There also exists a theoretical analysis [7], along with open source version of this method, which was later adopted by Esser et. al. [20] for performing the partial Gaussian Elimination that is necessary for our framework. For concrete complexities and fair comparison in our estimate, we excerpt the python script for this step directly from [20].[4]

Sieve_Syndrome_Dec This routine consists of performing Merge_Set ℓ times, corresponding to ℓ parity checks. Let us recap the Merge_Set subroutine of our ISD algorithm. Assume that a list of size M is sufficient, as stated in Section 4.1. In Algorithm 4, we go through the list to check if vectors fulfill the parity check conditions. For every sample of weight $2p$, the checking corresponds to summing $2p$ bits in the parity check matrix and the syndrome bit. Therefore, the cost for this step is about

$$C_{\text{check}} = 2p \cdot M.$$

The next step is Algorithm 5, which combines samples so that we create another $M/2$ vectors for the next iteration. By parsing $p = p' + p''$, we put our vectors in an ordered table of size $\binom{k+\ell-p''}{p'}$ and distribute our vectors according to their first p' coordinates (in the representation form). This way, when we move our vectors, we only need to read the value of the 'next' p' coordinate and move correspondingly; thus, the cost of moving is constant for each vector. Assume we are at the first 'bucket', i.e., examining all the vectors with 1 in their first p' coordinates. We produce all p'' labels for each vector, and we make use of an array $\mathbf{A}$ to keep track of how many times the labels have been produced (recall that we index $\mathbf{A}$ using the p'' labels). We then run through the list of labels that occurred more than once to find the vectors that need to be combined. This routine then ends by moving its content to the next 'bucket'. Note that, for every sample, we do not have to produce labels that include previous coordinates (as

[4] https://github.com/Crypto-TII/syndrome_decoding_estimator.

those labels are already processed in past buckets). The cost of this step can be broken down into the following parts:

- For each vector, we create precisely $\binom{2p}{p}$ markings on the array $\mathbf{A}$. This is the total number of times Lines 11-14 executes for each vector. It consists of two assignments and one comparison. On rare occasions, we additionally get collisions to handle. We also include the cost of creating a p'' label (Line 9-10). Assume that the cost of reading and marking each label in $\mathbf{A}$ is c_{label} operations. Then we need
$$C_{\text{label}} = \binom{2p}{p} \cdot c_{\text{label}} \cdot M,$$
operations for this part.
- The cost of moving vectors (Line 5). Since the remaining number of coordinates of a vector has to be at least p'', we only have to move a vector $\binom{2p-p''}{p'}$ times. Therefore, moving vectors cost
$$C_{\text{move}} = \binom{2p-p''}{p'} \cdot M.$$
- The cost of combining vectors. In the worst case, we have $2 \cdot M$ collisions, but only $M/2$ new unique vectors are kept (as explained in Section 4.2). For each collision, the cost of producing the new vector is the cost of creating the new $2p$ positions. Colliding positions are known, so it reduces to copying the other positions in an ordered form. We also need to compute the other parts of the vector representation and check for duplicates. This last part corresponds to bit-wise adding two values of bit size slightly larger than $\log M$ and then checking in a hash table if it is a duplicate. It may cost $2\log M$ operations.[5] Therefore, this step is estimated to cost
$$C_{\text{combine}} = (2p + 2\log M) \cdot 2 \cdot M.$$

The Merge_Set routine is then repeated ℓ times. Thus, if we introduce $C_{\text{Syndrome\_Dec}}$ as the bit complexity of performing all these steps then
$$C_{\text{Syndrome\_Dec}} = \left(2p + \binom{2p}{p} \cdot c_{\text{label}} + \binom{2p-p''}{p'} + 4(p + \log M)\right) \cdot \ell \cdot M$$

Testing candidates Finally, we have to go through the last list and check for weight-$(\omega - 2p)$ solutions, i.e., via the identity $\omega_H(\mathbf{H}''\mathbf{e} - \mathbf{s}) = \omega - 2p$. This corresponds to adding $2p$ length-$(n-k-\ell)$ columns in $\mathbf{H}''$; moreover, the number of solutions for the exact matching equation (9) is $\frac{\binom{k+\ell}{2p}}{2^\ell}$. Hence
$$C_{\text{solution\_check}} = 2p \cdot (n-k-\ell) \cdot \frac{\binom{k+\ell}{2p}}{2^\ell}.$$

[5] Here, we assume that the vector representation includes a "key" of bit-length larger than $\log M$. We check if the key is already present, which, in such a case, means that we created a duplicate. When we add two vectors, we also add their keys. The keys can be constructed as a syndrome vector for a random code.

Theorem 1 *The bit complexity C of the Sieving-Style ISD algorithm is*

$$C = \frac{1}{\Pr_{\text{success}}} \cdot \left(C_{\text{Gauss}} + C_{\text{Syndrome\_Dec}} + C_{\text{solution\_check}}\right), \tag{17}$$

where C_{Gauss} is the cost of the Gaussian elimination step.

The complexity given here is only an "as good as possible" estimation of the actual complexity. Some observations that decrease the complexity slightly are: For the case of one or a small number of valid solutions, the list size will decrease, and hence the complexity drops in later iterations; In the first few iterations, we can generate weight-$2p$ vectors that can be included in a faster way by exhaustive search.

We can note that if p is not very small, then the dominating part of the complexity expression is $(\binom{2p}{p} \cdot c_{\text{label}} \cdot \ell \cdot M)/\Pr_{\text{success}}$.

5 Numerical results

In this section, we provide the concrete complexity of our described sieving-style ISD algorithm when considering some proposed code-based schemes and also its comparison with other ISD algorithms.

Our analysis focuses first on the Classic McEliece parameter sets, with an extension to HQC and BIKE presented in Subsection 5.2. For reference in comparisons, we use the Syndrome decoding estimator by Esser et al. [20] as it covers most recent developments in this field.

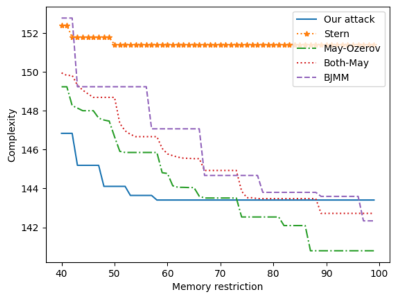
5.1 Numerical results for Classical McEliece

In this section, as well as in Section 5.2, we examine the security estimates with two values of c_{label} (reading and marking a label into $\mathbf{A}$), namely $c_{\text{label}} = 2$ and $c_{\text{label}} = 5$. The first case, corresponding to a value of 2, represents the optimal scenario and is intended to allow for comparisons with previous works, as the constant in the big $\mathcal{O}(\cdot)$ notation corresponding to using a hash table or similar, is typically set to 1. The second case, corresponding to a value of 5, reflects more of the actual computational cost, when calculated step-by-step.

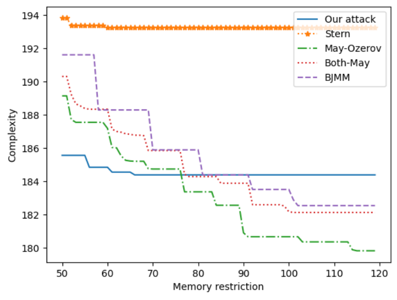
Table B.1 provides the security parameter sets of the Classic McEliece cryptosystem. Five parameter sets are published, including one for Category 1, one for Category 3, and three sets for Category 5.

In Table B.3, we present the bit security estimates of our new sieving-style ISD algorithm on these Classic McEliece parameter sets. The reference values are from the estimator in [20].

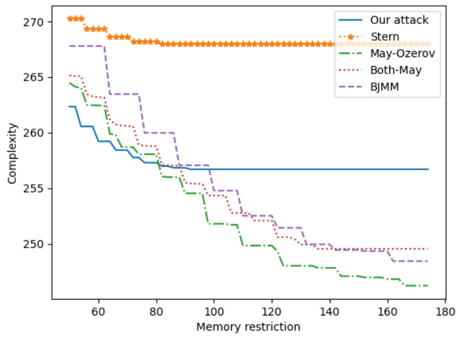
Figure 3 shows that our algorithm is favorable when the memory is restricted to 2^{60} bits compared to other ISD algorithms (details in Table B.3). Although it is not competitive when there is no memory constraint, our algorithm enriches the cryptanalytic arsenal with its different behavior in terms of time-memory trade-off.



(a) Category 1 ($n = 3488$)



(b) Category 3 ($n = 4608$)



(c) Category 5

Fig. 3: Time-memory trade-offs of different ISD (including ours, $c_{\text{label}} = 2$) algorithms.

5.2 Applications to BIKE and HQC

We apply the new algorithm to attacking BIKE and HQC, two round-4 KEM candidates in the NIST PQC project. NIST expects to standardize at most one of these two code-based KEM candidates at the end of the fourth round.

The parameter sets of BIKE and HQC are listed in Table B.2. These two schemes both select low-weight vectors that are sparser than the Classic McEliece scheme. The row weights of BIKE and HQC are of the order of $\mathcal{O}(\sqrt{n})$. In the concrete setting, HQC has an even sparser low-weight vector than BIKE. It is a commonly held belief that there have been limited advancements in the enhancement of modern ISD algorithms for sparse parameters as proposed in BIKE and HQC, as evidenced in [20]. It has been shown that the recent ISD methods of BJMM/MMT, Both-May and May-Ozerov have not made a significant improvement to Stern regarding these sparse parameters.

The bit security estimates on the new sieving-like ISD algorithm are shown in Table 1 and Table 2.[6] The complexity numbers regarding the reference algorithms, i.e., Prange, Stern, Both-May, May-Ozerov, and BJMM are from the recent work [20]. As being described in [20], the quasi-cyclic structure gives us k cyclic shifts of the searched secret key. The bit complexity numbers of a key-recovery attack on BIKE can be reduced by $\log(k)$ bits since BIKE is homogeneous. For key-recovery attacks on HQC and message-recovery attacks on BIKE, the bit complexity numbers of previous ISD algorithms can be reduced by $\log(k)/2$ due to the technique of 'decoding one out of many' (DOOM) [25,33]. The new sieving-ISD algorithm is highly favorable for addressing this DOOM problem, potentially diminishing the bit complexity number by approximately $\log(k) - 1$.

The DOOM problems from the key-recovery attack on HQC and the message-recovery attack on BIKE. We aim to search for a specific $2k$-dimensional vector, namely $\mathbf{e} = (\mathbf{e}_0, \mathbf{e}_1)$, which adheres to the condition $\mathbf{He} = \mathbf{s}$. The parity-check matrix of the code employed in HQC and BIKE is given by $\mathbf{H} = (\mathbf{H}_0, \mathbf{H}_1)$, where the matrices $\mathbf{H}_0$ and $\mathbf{H}_1$ are quasi-cyclic. In this context, $\mathbf{s}_i$ represents the i-th left cyclic shift of the syndrome $\mathbf{s}$. For each instance of $\mathbf{s}_i$, an $\mathbf{e}$ that satisfies $\mathbf{He} = \mathbf{s}_i$ can be found. The crux of the problem lies in outputting a single vector $\mathbf{e}$ amongst the k potential solutions. The problem is easier than the syndrome decoding problem due to the k potential solutions.

Incorporating DOOM into the sieving-ISD algorithm. We make minor modifications to the new sieving-like Information Set Decoding (ISD) algorithm for its application in the DOOM context. The DOOM problem appears to aggregate k sub-problems, i.e., creating k types of vectors and progressing to the subsequent iteration. Yet, the new sieving-like ISD presents a unique characteristic that enables the merging of vectors that have an all-zero syndrome $\mathbf{0}$ across all k categories. By leveraging this feature, we subsequently demonstrate that simply doubling the list size is sufficient to maintain a stable list size.

To be specific, in the i-th iteration, we include an index for each vector to distinguish which parity-check equation is satisfied. In the i-th iteration, one has for each $\mathbf{e} \in \mathcal{L}_i$ that

$$\omega_H(\mathbf{e}) = 2p, \text{ and } \mathbf{H}_{[i]}\mathbf{e} \in \{\mathbf{0}, \mathbf{s}_{j,[i]}\} \text{ for } 0 \leq j \leq k-1.$$

Here $\mathbf{s}_{j,[i]}$ represents the j-th left cyclic shift of the syndrome $\mathbf{s}$ restricted to its first i-th positions. We assign an index j to $\mathbf{e}$ to classify the vector $\mathbf{e} \in \mathcal{L}_i$ to $k+1$ categories, where $j = t$ for $0 \leq t \leq k-1$ represents $\mathbf{H}_{[i]}\mathbf{e} = \mathbf{s}_{t,[i]}$ and $j = k$ represents $\mathbf{H}_{[i]}\mathbf{e} = \mathbf{0}$.

Under the continued assumption that the list size is M per iteration—with $M/2$ samples carried forward from the preceding iteration—it becomes crucial to generate $M/2$ fresh samples via collision detection. In each iteration, we assume that $M/2$ samples have the index k and $M/(2k)$ samples have the index t for

[6] The optimal parameters can be found in the anonymous implementation repository.

$0 \leq t \leq k-1$. Let δ as defined in Section 4.1 be the fraction of all combinations that give rise to new vectors.

We create new samples, specifically $M/(4k)$ of them, carrying indices t where $0 \leq t \leq k-1$. This results in the equation

$$\frac{M}{2} \cdot \frac{M}{2k} \cdot \frac{\delta q}{2} = \frac{M}{4k},$$

yielding the minimum value for M as $\frac{2}{\delta q}$, aligning with the estimate in Equation (14). For the category with index k, we produce new samples, $M/4$ in number, which can be created by merging two vectors with identical indices. Hence, selecting $M = \frac{4}{\delta q}$, we obtain

$$\frac{\delta q \cdot \frac{M^2}{4 \cdot 2}}{2} = \frac{M}{4},$$

where the left side is the new vectors created from merging two vectors with the index k. In this analysis, combinations paired with an index differing from k are ignored due to their less frequent occurrence.

When it comes to complexity, the Gaussian elimination step incurs a marginal increase in cost, as we now work on a $(3k) \times k$ matrix rather than an $(2k) \times k$ matrix. The list size amplification by a factor of 2 means that the cost of inner steps roughly doubles. The cost of checking solutions increases to $k \cdot 2p \cdot (n-k-\ell) \cdot \frac{\binom{k+\ell}{2p}}{2^\ell}$, necessitating an increase in ℓ to balance the cost. Such a moderate increase in ℓ does not significantly impact the overall complexity. The total complexity experiences a significant reduction as the success probability can be amplified by a factor of k.

The bit complexity gap roughly approximates $\log(k) - 1$. For instance, let $c_{\text{label}} = 2$. For the key-recovery attack on HQC parameters aiming at NIST-5, our ISD cost is $2^{275.5}$ ignoring the DOOM gain and $2^{261.2}$ accounting for the DOOM gain. The ratio amounts to $2^{14.3}$, while the value of k stands at $2^{15.8}$. For the key-recovery attack on HQC parameters aimed at NIST-1, our ISD cost is $2^{146.2}$ excluding the DOOM gain and $2^{133.6}$ considering the DOOM gain. The ratio equals $2^{12.6}$, with the value of k being $2^{14.1}$.

Consequently, we accomplish a DOOM gain of $\mathcal{O}(k)$ for k solutions as opposed to the previous $\mathcal{O}\left(\sqrt{k}\right)$ from earlier ISD algorithms, as detailed in [33], which is a noteworthy outcome.

Impact on the security estimates. The newly developed sieving-like ISD algorithm has shown appealing results for the BIKE and HQC parameter sets in Table 1 and Table 2. Compared with the state-of-the-art algorithms (see the estimator in [20]), a gain of up to 12 bits in Category 1 and 15 bits in Category 5 has been observed. It is noteworthy that the complexity of the attacks, in all cases, falls below the NIST requirements, namely 143 bits for Category 1, 207 bits for Category 3, and 272 bits for Category 5. The security degradation may reach a maximum of 10 bits even in Category 1 parameters.

Table 1: Bit security estimates of the key-recovery attacks on BIKE. Here T is the log of the bit complexity and $\hat{M}$ is the log of the memory in bits. The parenthesis notation $(6, 48)$ specifies that $2p = 6$ and $\ell = 48$. We have $p' = 1$.

	Category 1		Category 3		Category 5	
	T	$\hat{M}$	T	$\hat{M}$	T	$\hat{M}$
PRANGE	169	28	234	30	304	32
STERN	147	40	211	43	279	45
BOTH-MAY	148	38	211	60	278	63
MAY-OZEROV	147	55	210	57	278	61
BJMM	147	54	211	59	279	63
Our ISD, $c_{\mathbf{label}} = 2$	140.7	46	203.6	50	270.6	53
	(6,48)		(6,52)		(6, 55)	
Our ISD, $c_{\mathbf{label}} = 5$	141.1	46	203.9	50	271.0	53
	(6,48)		(6,52)		(6, 55)	

We emphasize the novelty of this improvement and demonstrate the superiority of our newly proposed ISD algorithm for sparse parameter sets. An intuitive explanation for this advantage is that our new ISD algorithm is capable of significantly enhancing the STERN algorithm for sparse parameter sets, while other modern ISD algorithms are not.

The value of c_{label} has a minimal effect on the time complexity in contrast to the Classic McEliece scenario. This is primarily due to the fact that the parameter p is set to 3 when solving these highly sparse instances, and thus the cost associated with c_{label} is not the primary contributing factor to the complexity.

Example 3. We present a decomposition of the algorithm complexity into distinct components that pertain to the check, label, move, combine, and final solution check operations and employ as an example the bit estimates from Table 1 for key-recovery attack on the Category 1 BIKE parameter set.

When c_{label} is set to be 2, the attack parameters are $p = 3$, $\ell = 48$ and $p' = 1$, and as presented in Table 1, the list size requirement is 2^{31} and the time complexity amounts to $2^{140.7}$. In this setting, the cost C_{Gauss} of the Gaussian elimination step is $2^{39.54}$, the cost $C_{\text{Syndrome\_Dec}}$ of the Sieve_Syndrome_Dec step is $2^{44.27}$, and the cost $C_{\text{solution\_check}}$ of the final candidate test step is $2^{40.24}$. Moreover, the formulae $C_{\text{check}} \cdot \ell$, $C_{\text{label}} \cdot \ell$, $C_{\text{move}} \cdot \ell$, and $C_{\text{combine}} \cdot \ell$ entail costs of $2^{39.31}$, $2^{42.04}$, $2^{38.72}$, and $2^{43.81}$, respectively.

When c_{label} is set to be 5, the attack time complexity rises to $2^{141.1}$ due to the corresponding increase in the cost $C_{\text{Syndrome\_Dec}}$ of the Sieve_Syndrome_Dec step to $2^{44.67}$. Notably, the cost of $C_{\text{label}} \cdot \ell$ increases from $2^{42.04}$ to $2^{43.37}$, but this cost does not assume a dominant position, and hence the overall impact on the reported complexity is insignificant.

In addition, we have computed the probability of finding the desired vector by numerical means for this particular example, using the method presented in

Table 2: Bit security estimates of the DOOM attacks on BIKE and HQC. Here T is the log of the bit complexity and $\hat{M}$ is the log of the memory in bits. The parenthesis notation $(6, 61)$ specifies that $2p = 6$ and $\ell = 61$. We have $p' = 1$.

	Category 1		Category 3		Category 5	
	T	$\hat{M}$	T	$\hat{M}$	T	$\hat{M}$
BIKE (message)						
Prange	167	28	235	30	301	32
Stern	146	40	211	43	277	45
Both-May	147	38	212	41	276	63
May-Ozerov	146	55	211	57	276	61
BJMM	147	38	211	59	277	63
Our ISD, $c_{\textbf{label}} = 2$	134.7	46	198.3	50	262.4	53
	(6,61)		(6,66)		(6, 70)	
Our ISD, $c_{\textbf{label}} = 5$	135.1	46	198.7	50	262.7	53
	(6,60)		(6,66)		(6, 69)	
HQC (key)						
Prange	166	29	237	31	300	33
Stern	145	41	213	44	276	46
Both-May	146	39	214	42	276	39
May-Ozerov	145	39	214	42	276	44
BJMM	146	39	214	42	276	44
Our ISD, $c_{\textbf{label}} = 2$	133.6	48	200.1	52	261.2	55
	(6,64)		(6,69)		(6, 72)	
Our ISD, $c_{\textbf{label}} = 5$	133.9	48	200.4	52	261.5	55
	(6,63)		(6,68)		(6, 72)	

Section 4.3; our calculation results in an estimated value of 53.5%. The figure confirms that for the reported attack parameters, the success probability are usually larger than 50%. We have verified this observation on other parameter sets as well, thereby affirming the soundness of our complexity analysis in conjunction with the parity bit trick.

6 Simple implementations for Sieve_Syndrome_Dec with smaller parameters

In this section, we discuss our simple implementation with smaller parameters to support arguments and assumptions that we have made throughout the papers.[7] It is valuable to show in simulation that Sive_Syndrome_Dec is capable of producing solutions for the exact matching equation as theory predicts, and the parameters such as list size can be sustained. In the implementation, we generate a random target error vector $\mathbf{e}$ and observe whether this vector can be

[7] Anonymous repository.

found by Sieve_Syndrome_Dec after ℓ iteration of Merge_Set. Moreover, we set $p' = 1$ in our simulations.

Example 4. One of many implemented parameter sets is $k = 300$, $2p = 6$, i.e., $p = 3$. We set $p' = 1$, $p'' = 2$ for the Merge_Set algorithm. For the parameter ℓ, we choose $\ell \approx 28$ (recall Equation (15), we also need $\frac{M}{2} \geq \frac{\binom{k+\ell}{2p}}{2^\ell}$) which corresponds to the exponentially many solution.

The probability that the XOR of two weight-6 vectors results in another weight-6 vector is

$$q = \frac{\binom{2p}{p}\binom{k+\ell-2p}{p}}{\binom{k+\ell}{2p}} \approx 2^{-13.86}.$$

$$M \approx \frac{2}{\delta q} \approx \delta^{-1} \cdot 2^{14.86}.$$

As explained in Section 4.3, we increase M with a factor $\delta^{-1} \approx 3/2$ so that we can keep the list size relatively constant for the majority of iterations (until Merge_Set can not produce $M/2$ new vectors). Hence, we select $M = 2^{15.44}$.

We run the implementation 10^2 times and find $\mathbf{e}$ in 60 runs, i.e., a 60% success rate.

Table 3: Comparison between the heuristic arguments and the actual implementation for $k = 300$, $\ell = 28$, $p = 3$.

Iteration	1	...	14	15	16	...	24	25	26	27	28
$\log(M_i)$ (pred.)	15.46	...	15.46	15.46	15.46	...	15.46	15.44	14.80	13.96	13.09
$\log(M_i)$ (impl.)	15.46	...	15.46	15.46	15.46	...	15.44	15.22	14.71	13.91	12.96
Success Prob. (pred.)	0	...	$22 \cdot 10^{-5}$	$45 \cdot 10^{-5}$	$9 \cdot 10^{-4}$	...	0.204	0.363	0.441	0.477	0.507

We note that any sufficiently large enough ℓ can be chosen. As an example, in the case where $\ell = 50$ (i.e., on average, only one solution), our algorithm still finds the target $\mathbf{e}$ with promising probability ($> 50\%$). It can also be inferred from Table 3 that one can choose ℓ in order to raise the success probability to a desired range as claimed in Section 4.3.

Example 5. It is also of interest to see how our implementation fares with larger instance of k (e.g., close to the medium-sized instance of McEliece). In particular, we proceed with $k = 1000$ and $2p = 4$. We choose a smaller values of p to have a manageable memory requirement for a commercial computer. The following numerical values are derived in the same manner as in Example 1.

For $\ell = 27$, it gives $M \approx 2^{15.42}$. A target vector $\mathbf{e}$ is found in 56 out of 10^2 tests, i.e., a 56% success probability.

Table 4: Comparison between the heuristic arguments and the actual implementation for $k = 1000$, $\ell = 27$, $p = 2$.

Iteration	1	...	19	20	21	22	23	24	25	26	27
$\log(M_i)$ (pred.)	15.42	...	15.42	15.35	14.64	13.82	12.94	12.01	11.05	10.08	9.09
$\log(M_i)$ (impl.)	15.42	...	15.33	14.97	14.32	13.46	12.50	11.51	10.49	9.50	8.41
Success Prob. (pred.)	0	...	0.221	0.380	0.442	0.482	0.511	0.529	0.539	0.545	0.548

Discussions. We have observed that the actual implementation results are comparable to or even surpass the estimation results obtained using the method described in Section 4.3. Moreover, in Table 5, we present the evolution of the estimated list size and estimated success probability over the course of various iterations, utilizing an attack instance on Classic McEliece as reported in Table B.3. In both our theoretical calculations and empirical investigations, we have identified a critical juncture, referred to as a '*breaking point*', which corresponds to the iteration at which the list size of M_i begins to decrease. While the initial decline is gradual, it gains momentum as subsequent iterations progress.

One favorable aspect in this iterative process is that upon reaching the 'breaking point', the success probability becomes non-negligible and quickly rises above 50%. Subsequent iterations will result in a further reduction of the list size, leading to a slower increase in the success probability in finding the targeted vector.

We have observed that the attack instances reported in the previous section all select the parameter ℓ several iterations after the occurrence of the 'breaking point', thereby guaranteeing a success probability exceeding 50%. Additionally, our experiments demonstrate that, for a choice of ℓ close to the 'breaking point', the actual list size is consistent with the theoretical estimation and the observed success probability meets (or even surpasses) the estimated value.

Table 5: The estimated success probability for attacking a Classic McEliece instance with $k = 3360, \ell = 96, p = 8$.

Iteration	1	...	89	90	91	92	93	94	95	96
$\log(M_i)$ (pred.)	53.03	...	53.03	52.93	52.60	51.99	51.18	50.28	49.34	48.37
Success Prob. (pred.)	0	...	0.138	0.242	0.358	0.441	0.485	0.509	0.523	0.529

7 Concluding remarks

We have presented a novel sieving-style information-set-decoding algorithm for solving the syndrome decoding problem and made a heuristic analysis. The algorithm makes advancements of state-of-the-art algorithms when complexity is considered in the RAM model and is characterized by its time-memory trade-off profile. For instance, in McEliece cryptographic scheme, an attack using the algorithm achieves lower complexity when the memory is limited (for example, 2^{60} bits). Interestingly, it was also shown that the low-weight regime (in constructions such as BIKE and HQC) benefits our algorithm compared to the state-of-the-art. This finding is of great interest as the advantage of enumeration-based ISD variants is believed to diminish with sparse parameters. Newly improved complexity results were given for the proposed parameter sets of BIKE and HQC.

Besides the described algorithm, many other versions of the algorithms can be considered. For instance, we can amend the problem of duplicates by only combining vectors in the first few iterations and including the checking routine later. The motivation is that the correct error vector will not likely be created in early iterations, and combining vectors does not result in noticeable dependencies between vectors. Moreover, in specific settings, such as BIKE and HQC, where the optimal value of p is small and the memory requirement is not high, more efficient implementation could be achieved.

Lastly, we note that accelerating the new ISD algorithm using sophisticated instruction sets such as AVX-256 in practical software design seems non-trivial. Further exploration of this intriguing topic and actual, full-scaled implementations of concrete parameters of code-based schemes are left for future endeavors.

References

1. NIST Post-Quantum Cryptography Standardization. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization`, accessed: 2022-11-30
2. Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: Proceedings of the thirty-third annual ACM symposium on Theory of computing. pp. 601–610 (2001)
3. Aragon, N., Barreto, P.S., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Gueron, S., Guneysu, T., Melchor, C.A., et al.: Bike: bit flipping key encapsulation (2017)
4. Arora, S., Barak, B.: Computational Complexity - A Modern Approach. Cambridge University Press (2009), `http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264`
5. Augot, D., Finiasz, M., Sendrier, N.: A family of fast syndrome based cryptographic hash functions. In: Dawson, E., Vaudenay, S. (eds.) Progress in Cryptology - Mycrypt 2005, First International Conference on Cryptology in Malaysia, Kuala Lumpur, Malaysia, September 28-30, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3715, pp. 64–83. Springer (2005), `https://doi.org/10.1007/11554868_6`
6. Baldi, M., Barenghi, A., Chiaraluce, F., Pelosi, G., Santini, P.: A finite regime analysis of information set decoding algorithms. Algorithms 12(10), 209 (2019), `https://doi.org/10.3390/a12100209`
7. Bard, G.: Algorithms for solving linear and polynomial systems of equations over finite fields with application to cryptanalysis. Ph.D. thesis, Faculty of the Graduate School of the University of Maryland, College Park (2007)

8. Becker, A., Gama, N., Joux, A.: Solving shortest and closest vector problems: The decomposition approach. IACR Cryptol. ePrint Arch. p. 685 (2013), http://eprint.iacr.org/2013/685
9. Becker, A., Joux, A., May, A., Meurer, A.: Decoding random binary linear codes in $2^{n/20}$: How $1 + 1 = 0$ improves information set decoding. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology – EUROCRYPT 2012. Lecture Notes in Computer Science, vol. 7237, pp. 520–536. Springer, Heidelberg, Germany, Cambridge, UK (Apr 15–19, 2012)
10. Berlekamp, E.R., McEliece, R.J., van Tilborg, H.C.A.: On the inherent intractability of certain coding problems (corresp.). IEEE Trans. Information Theory 24(3), 384–386 (1978), https://doi.org/10.1109/TIT.1978.1055873
11. Bernstein, D.J., Chou, T., Lange, T., von Maurich, I., Misoczki, R., Niederhagen, R., Persichetti, E., Peters, C., Schwabe, P., Sendrier, N., et al.: Classic mceliece: conservative code-based cryptography. NIST submissions (2017)
12. Bernstein, D.J., Lange, T., Peters, C.: Attacking and defending the McEliece cryptosystem. In: Buchmann, J., Ding, J. (eds.) Post-quantum cryptography, second international workshop, PQCRYPTO 2008. pp. 31–46. Springer, Heidelberg, Germany, Cincinnati, Ohio, United States (Oct 17–19 2008)
13. Bernstein, D.J., Lange, T., Peters, C.: Smaller decoding exponents: Ball-collision decoding. In: Rogaway, P. (ed.) Advances in Cryptology – CRYPTO 2011. Lecture Notes in Computer Science, vol. 6841, pp. 743–760. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 14–18, 2011)
14. Bernstein, D.J., Lange, T., Peters, C., Schwabe, P.: Really fast syndrome-based hashing. In: Nitaj, A., Pointcheval, D. (eds.) AFRICACRYPT 11: 4th International Conference on Cryptology in Africa. Lecture Notes in Computer Science, vol. 6737, pp. 134–152. Springer, Heidelberg, Germany, Dakar, Senegal (Jul 5–7, 2011)
15. Both, L., May, A.: Optimizing bjmm with nearest neighbors: full decoding in 22/21n and mceliece security. In: WCC workshop on coding and cryptography. p. 214 (2017)
16. Both, L., May, A.: Decoding linear codes with high error rate and its impact for LPN security. In: Lange, T., Steinwandt, R. (eds.) Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10786, pp. 25–46. Springer (2018), https://doi.org/10.1007/978-3-319-79063-3_2
17. Carrier, K., Debris-Alazard, T., Meyer-Hilfiger, C., Tillich, J.: Statistical decoding 2.0: Reducing decoding to LPN. In: Agrawal, S., Lin, D. (eds.) Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 13794, pp. 477–507. Springer (2022), https://doi.org/10.1007/978-3-031-22972-5_17
18. Courtois, N., Finiasz, M., Sendrier, N.: How to achieve a McEliece-based digital signature scheme. In: Boyd, C. (ed.) Advances in Cryptology – ASIACRYPT 2001. Lecture Notes in Computer Science, vol. 2248, pp. 157–174. Springer, Heidelberg, Germany, Gold Coast, Australia (Dec 9–13, 2001)
19. Dumer, I.: On minimum distance decoding of linear codes. In: Proc. 5th Joint Soviet-Swedish International Workshop Information Theory. pp. 50–52 (1991)
20. Esser, A., Bellini, E.: Syndrome decoding estimator. In: Hanaoka, G., Shikata, J., Watanabe, Y. (eds.) Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part I. Lecture Notes in Computer

Science, vol. 13177, pp. 112–141. Springer (2022), https://doi.org/10.1007/978-3-030-97121-2_5
21. Esser, A., May, A., Zweydinger, F.: Mceliece needs a break - solving mceliece-1284 and quasi-cyclic-2918 with modern ISD. In: Dunkelman, O., Dziembowski, S. (eds.) Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part III. Lecture Notes in Computer Science, vol. 13277, pp. 433–457. Springer (2022), https://doi.org/10.1007/978-3-031-07082-2_16
22. Finiasz, M., Sendrier, N.: Security bounds for the design of code-based cryptosystems. In: Matsui, M. (ed.) Advances in Cryptology – ASIACRYPT 2009. Lecture Notes in Computer Science, vol. 5912, pp. 88–105. Springer, Heidelberg, Germany, Tokyo, Japan (Dec 6–10, 2009)
23. Fischer, J., Stern, J.: An efficient pseudo-random generator provably as secure as syndrome decoding. In: Maurer, U.M. (ed.) Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding. Lecture Notes in Computer Science, vol. 1070, pp. 245–255. Springer (1996), https://doi.org/10.1007/3-540-68339-9_22
24. Hamdaoui, Y., Sendrier, N.: A non asymptotic analysis of information set decoding. Cryptology ePrint Archive (2013)
25. Johansson, T., Jönsson, F.: On the complexity of some cryptographic problems based on the general decoding problem. IEEE Trans. Inf. Theory 48(10), 2669–2678 (2002), https://doi.org/10.1109/TIT.2002.802608
26. Lee, P.J., Brickell, E.F.: An observation on the security of mceliece's public-key cryptosystem. In: Günther, C.G. (ed.) Advances in Cryptology - EUROCRYPT '88, Workshop on the Theory and Application of of Cryptographic Techniques, Davos, Switzerland, May 25-27, 1988, Proceedings. Lecture Notes in Computer Science, vol. 330, pp. 275–280. Springer (1988), https://doi.org/10.1007/3-540-45961-8_25
27. Leon, J.S.: A probabilistic algorithm for computing minimum weights of large error-correcting codes. IEEE Trans. Inf. Theory 34(5), 1354–1359 (1988), https://doi.org/10.1109/18.21270
28. May, A., Meurer, A., Thomae, E.: Decoding random linear codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In: Lee, D.H., Wang, X. (eds.) Advances in Cryptology – ASIACRYPT 2011. Lecture Notes in Computer Science, vol. 7073, pp. 107–124. Springer, Heidelberg, Germany, Seoul, South Korea (Dec 4–8, 2011)
29. May, A., Ozerov, I.: On computing nearest neighbors with applications to decoding of binary linear codes. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015, Part I. Lecture Notes in Computer Science, vol. 9056, pp. 203–228. Springer, Heidelberg, Germany, Sofia, Bulgaria (Apr 26–30, 2015)
30. Melchor, C.A., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Persichetti, E., Zémor, G., Bourges, I.: Hamming quasi-cyclic (hqc). NIST PQC Round 2(4), 13 (2018)
31. Nguyen, P.Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. Journal of Mathematical Cryptology 2(2), 181–207 (2008)
32. Prange, E.: The use of information sets in decoding cyclic codes. IRE Trans. Information Theory 8(5), 5–9 (1962), https://doi.org/10.1109/TIT.1962.1057777
33. Sendrier, N.: Decoding one out of many. In: Yang, B.Y. (ed.) Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011. pp. 51–67. Springer, Heidelberg, Germany, Tapei, Taiwan (Nov 29 – Dec 2 2011)

34. Stern, J.: A method for finding codewords of small weight. In: Cohen, G.D., Wolfmann, J. (eds.) Coding Theory and Applications, 3rd International Colloquium, Toulon, France, November 2-4, 1988, Proceedings. Lecture Notes in Computer Science, vol. 388, pp. 106–113. Springer (1988), https://doi.org/10.1007/BFb0019850
35. Stern, J.: A new identification scheme based on syndrome decoding. In: Stinson, D.R. (ed.) Advances in Cryptology – CRYPTO'93. Lecture Notes in Computer Science, vol. 773, pp. 13–21. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 22–26, 1994)
36. Véron, P.: Improved identification schemes based on error-correcting codes. Appl. Algebra Eng. Commun. Comput. 8(1), 57–69 (1996), https://doi.org/10.1007/s002000050053

Auxiliary Supporting Material

A The estimation code

We present a small python script to reproduce the complexity numbers reported in this paper.

```python
# import libraries
import mpmath as math
import numpy as np
from math import inf, floor, ceil
from functools import lru_cache, cache
import sys

DOOM = 1 # 0 for the normal attack; 1 for the doom attack.




# parameters
c_label = int(sys.argv[1])
M_UPPER_BOUND = float(sys.argv[2])
M_overhead = 1.5
print("C_label:",c_label)
print("M_upper_bound:",M_UPPER_BOUND)
print("DOOM:", DOOM)
print("  ")
print("===============================================")

@lru_cache(maxsize = 12800000)
def comb(N,k):
    val= math.factorial(N)/(math.factorial(k)*math.factorial(N-k))
    return val

def log2(N):
    return math.log(N,2)


@lru_cache(maxsize = 12800000)
def _gaussian_elimination_complexity(n, k, r):
    """
    Complexity estimate of Gaussian elimination routine

    INPUT:

    - ``n`` -- Row additons are perfomed on ``n`` coordinates
    - ``k`` -- Matrix consists of ``n-k`` rows
    - ``r`` -- Blocksize of method of the four russian for inversion, default is zero

    [Bar07]_ Bard, G.V.: Algorithms for solving linear and polynomial systems of equations over finite fields
    with applications to cryptanalysis. Ph.D. thesis (2007)

    [BLP08] Bernstein, D.J., Lange, T., Peters, C.: Attacking and defending the mceliece cryptosystem.
    In: International Workshop on Post-Quantum Cryptography. pp. 31  46  . Springer (2008)

    EXAMPLES::

        >>> from .estimator import _gaussian_elimination_complexity
        >>> _gaussian_elimination_complexity(n=100,k=20,r=1) # doctest: +SKIP

    """

    if r != 0:
        return (r ** 2 + 2 ** r + (n - k - r)) * int(((n + r - 1) / r))

    return (n - k) ** 2

@lru_cache(maxsize = 12800000)
def _optimize_m4ri(n, k, mem=inf):
    """
    Find optimal blocksize for Gaussian elimination via M4RI

    INPUT:

    - ``n`` -- Row additons are perfomed on ``n`` coordinates
    - ``k`` -- Matrix consists of ``n-k`` rows

    """

    (r, v) = (0, inf)
    for i in range(n - k):
        tmp = log2(_gaussian_elimination_complexity(n, k, i))
        if v > tmp and r < mem:
            r = i
            v = tmp
    return r


def gauss(n,k,w,p,l):
  r = _optimize_m4ri(n,k, 30)
  return  _gaussian_elimination_complexity(n, k, r)*n

delta = 0
@lru_cache(maxsize = 12800000)
def pr_dual_left(k, w, p, l):
  return (comb(floor((float)(k+l)/2)+delta ,p)*comb(k-floor((float)(k+l)/2)-delta, w//2 - p)) / comb(k,w//2)


@lru_cache(maxsize = 12800000)
def pr_dual_right(k, w, p, l):
  return (comb(ceil((float)(k+l)/2)-delta,p)*comb(k-ceil((float)(k+l)/2)+delta, w//2 - p)) / comb(k,w//2)

def pr_success(n,k,w,p,l,dual=0):
  pr = comb(n,w)/(comb(k+l,p)*comb(n-k-l, w - p))
  return log2(pr)
```

```



def C_final_check_doom(n,k,p,l,M, n_0, c_label = c_label):
  sol = n_0*comb(k+l,p)/(2**l) # number of solutions
  return p*(n-k-l)*sol


@cache
def C_final_check(n,k,p,l,M, c_label = c_label):
  sol = comb(k+l,p)/(2**l) # number of solutions
  return p*(n-k-l)*sol

def C_check(n,k,p,l,M, c_label = c_label):
  return p*l*M

def C_label(n,k,p,l,M, c_label = c_label):
  return c_label*comb(p, int(p/2)) *M *l

def C_move(n,k,p,l,M, p_prime, c_label = c_label):
  p_pprime = int(p/2) - p_prime
  return comb(p-p_pprime, p_prime) *M *l

def C_combine(n,k,p,l,M, c_label = c_label):
  return (2*p  + 4* log2(M))*M *l

def C_inner_prime(n,k,p,l,M, p_prime, c_label = c_label):
  p_pprime = int(p/2) - p_prime
  inner = l*M*(p + c_label*comb(p, int(p/2))  + comb(p - p_pprime, p_prime) +
     2*p  + 4* log2(M))
  # p is from the check; c_label is label; C_mov is comb(p-p_pprime, p_prime)
    ; 2*p + 4* log2(M) is from combine.
  final_check = C_final_check(n,k,p,l,M, c_label)
  return final_check + inner

def C_sd(n,k,p,l,M, p_prime, c_label = c_label):
  p_pprime = int(p/2) - p_prime
  inner = l*M*(p + c_label*comb(p, int(p/2))  + comb(p - p_pprime, p_prime) +
     2*p  + 4* log2(M))
  # p is from the check; c_label is label; C_mov is comb(p-p_pprime, p_prime)
    ; 2*p + 4* log2(M) is from combine.
  return inner


def ISD_prime(n,k,w,p,p_prime, l,M,dual):
  # print(k)
  return pr_success(n,k,w,p,l, dual) + log2(float(gauss(n,k,w,p,l) +
    C_inner_prime(n,k,p,l,M,p_prime)))

def ISD_doom(n,k,w,p,p_prime, l,M,dual, n_0):
  '''
  @summary: ISD_doom is the complexity for the ISD for the doom attack. (
    decoding out of the many syndromes)
  @param n_0: n_0 is the number of solutions in the doom attack.
  '''
  def C_final_check_doom(n,k,p,l,M, n_0, c_label = c_label):
    sol = n_0*comb(k+l,p)/(2**l) # number of solutions
    return p*(n-k-l)*sol
  def C_inner_doom(n,k,p,l,M, p_prime, n_0, c_label = c_label):
    p_pprime = int(p/2) - p_prime
    inner = l*M*(p + c_label*comb(p, int(p/2))  + comb(p - p_pprime, p_prime)
     + 2*p  + 4* log2(M))
    # p is from the check; c_label is label; C_mov is comb(p-p_pprime,
    p_prime); 2*p + 4* log2(M) is from combine.
    final_check = C_final_check_doom(n,k,p,l,M,n_0, c_label)
    return final_check + inner
  # print(k)
```

```python
  return pr_success(n,k,w,p,l, dual)- log2(n_0) + log2(float(gauss(n+n_0,k,w,
    p,l) + C_inner_doom(n,k,p,l,M,p_prime,n_0)))

class BIKE_security:
  """
  _summary_
  Estimate the security of the BIKE proposal against the key-recovery attacks
    .
  """
  def __init__(self, level):
    if level == 1:
      self.k = 12323
      self.n = self.k * 2
      self.w = 142
      self.t = 134
      self.lev = 1
    if level == 3:
      self.k = 24659
      self.n = self.k * 2
      self.w = 206
      self.t = 199
      self.lev = 3
    if level == 5:
      self.k = 40973
      self.n = self.k * 2
      self.w = 274
      self.t = 264
      self.lev = 5

  def BIKE_attack_key(self):
    '''
    Attack complexity of BIKE
    '''
    params = [0,0]
    init = 1000.0
    mem = 0
    params.append(init)
    params.append(mem)
    params.append(0)
    for p in range (2, 10+1,2):
      for p_prime in range (0, int(p/2)):
        p_pprime = int(p/2) - p_prime
        for l in range (0,101):
          q = comb(p,int(p/2))*comb(self.k+l-p,int(p/2))/comb(self.k+l,p)
          M = 2/q
          M *= M_overhead
          sol = (comb(self.k+l,p))/(np.longdouble)(2**l)
          temp = ISD_prime(self.n,self.k,self.w,p,p_prime, l,M, dual=0) -
    log2(self.k) # cyclic shift of bike key.
          if sol >= 1 and temp <= init and M/2>= sol and comb(self.k+l,
    p_pprime) <= M and log2(M) <120:
            init = temp
            params[0] = p
            params[1]= l
            params[2] = temp
            params[3] = log2(M)
            params[4] = p_prime

    print(params)
    print("Gauss cost:", log2(gauss(self.n,self.k,self.w,params[0],params[1])
    ))
    print("Merge set cost:", log2(C_sd(self.n,self.k,params[0],params[1],2**
    params[3], params[4])))
    print("check cost:", log2(C_check(self.n,self.k,params[0],params[1],2**
    params[3])))
    print("C_label cost:", log2(C_label(self.n,self.k,params[0],params[1],2**
    params[3])))
    print("C_move cost:", log2(C_move(self.n,self.k,params[0],params[1],2**params[3], params[4])))
    print("C_combine cost:", log2(C_combine(self.n,self.k,params[0],params[1],2**params[3], params[4])))
    print("final check cost:", log2(C_final_check(self.n,self.k,params[0],params[1],2**params[3], params[4])))

  def BIKE_attack_message(self):
    '''
    Attack complexity of BIKE
    '''
    params = [0,0]
    init = 1000.0
    mem = 0
    params.append(init)
    params.append(mem)
    params.append(0)
    for p in range (2, 10+1,2):
      for p_prime in range (0, int(p/2)):
        p_pprime = int(p/2) - p_prime
        for l in range (0,101):
          q = comb(p,int(p/2))*comb(self.k+l-p,int(p/2))/comb(self.k+l,p)
          M = 2/q
          M *= M_overhead
          if DOOM == 1:
            M *= 2 # from the doom attack
            sol = self.k * (comb(self.k+l,p))/(np.longdouble)(2**l)
            temp = ISD_doom(self.n,self.k,self.t,p,p_prime, l,M,0, self.k) # out of many.
          if DOOM == 0:
            sol =  (comb(self.k+l,p))/(np.longdouble)(2**l)
            temp = ISD_prime(self.n,self.k,self.t,p,p_prime, l,M,0)
          if sol >= 1 and temp <= init and M/2>= sol and comb(self.k+l, p_pprime) <= M and log2(M) <M_UPPER_BOUND:
            init = temp
            params[0] = p
            params[1]= l
            params[2] = temp
            params[3] = log2(M)
            params[4] = p_prime

    print(params)
    print("Gauss cost:", log2(gauss(self.n,self.k,self.t,params[0],params[1])))
    print("Merge set cost:", log2(C_sd(self.n,self.k,params[0],params[1],2**params[3], params[4])))
    print("check cost:", log2(C_check(self.n,self.k,params[0],params[1],2**params[3])))
    print("C_label cost:", log2(C_label(self.n,self.k,params[0],params[1],2**params[3])))
    print("C_move cost:", log2(C_move(self.n,self.k,params[0],params[1],2**params[3], params[4])))
    print("C_combine cost:", log2(C_combine(self.n,self.k,params[0],params[1],2**params[3])))
    print("final check cost:", log2(C_final_check_doom(self.n,self.k,params[0],params[1],2**params[3], self.k)))
    print("probability:", pr_success(self.n,self.k,self.t,params[0],params[1], dual=0)-log2(self.k))


class HQC_security:
  """
  _summary_
  Estimate the security of the HQC proposal against the key-recovery attacks.
  """
  def __init__(self, level):
    if level == 1:
      self.k = 17669
```

```python
      self.n = self.k * 2
      self.w = 132
      self.lev = 1
    if level == 3:
      self.k = 35851
      self.n = self.k * 2
      self.w = 200
      self.lev = 3
    if level == 5:
      self.k = 57637
      self.n = self.k * 2
      self.w = 262
      self.lev = 5

  def HQC_attack_key(self):
    '''
    Attack complexity of HQC key.
    '''
    params = [0,0]
    init = 1000.0
    mem = 0
    params.append(init)
    params.append(mem)
    params.append(0)
    for p in range (2, 12+1,2):
      for p_prime in range (0, int(p/2)):
        p_pprime = int(p/2) - p_prime
        for l in range (0,101):
          q = comb(p,int(p/2))*comb(self.k+l-p,int(p/2))/comb(self.k+l,p)
          M = 2/q
          M *= M_overhead
          if DOOM == 1:
            M *= 2 # from the doom attack
            sol = self.k * (comb(self.k+l,p))/(np.longdouble)(2**l)
            temp = ISD_doom(self.n,self.k,self.w,p,p_prime, l,M,0, self.k) # out of many.
          if DOOM == 0:
            sol =  (comb(self.k+l,p))/(np.longdouble)(2**l)
            temp = ISD_prime(self.n,self.k,self.w,p,p_prime, l,M,0)
          if sol >= 1 and temp <= init and M/2>= sol and comb(self.k+l, p_pprime) <= M and log2(M) <M_UPPER_BOUND:
            init = temp
            params[0] = p
            params[1]= l
            params[2] = temp
            params[3] = log2(M)
            params[4] = p_prime

    print(params)
    print("Gauss cost:", log2(gauss(self.n,self.k,self.w,params[0],params[1])))
    print("Merge set cost:", log2(C_sd(self.n,self.k,params[0],params[1],2**params[3], params[4])))
    print("check cost:", log2(C_check(self.n,self.k,params[0],params[1],2**params[3])))
    print("C_label cost:", log2(C_label(self.n,self.k,params[0],params[1],2**params[3])))
    print("C_move cost:", log2(C_move(self.n,self.k,params[0],params[1],2**params[3], params[4])))
    print("C_combine cost:", log2(C_combine(self.n,self.k,params[0],params[1],2**params[3])))
    print("final check cost:", log2(C_final_check_doom(self.n,self.k,params[0],params[1],2**params[3], self.k)))
    print("probability:", pr_success(self.n,self.k,self.w,params[0],params[1], dual=0)-log2(self.k))


class CM_security:
  """
  _summary_
  Estimate the security of the CM proposal against the key-recovery attacks.
  """
  def __init__(self, level):
    if level == 1:
      self.k = 2720
      self.n = 3488
      self.w = 64
      self.lev = 1
    if level == 3:
      self.k = 3360
      self.n = 4608
      self.w = 96
      self.lev = 3
    if level == 5:
      self.k = 5024
      self.n = 6688
      self.w = 128
      self.lev = 5
    if level == 7:
      self.k = 5413
      self.n = 6960
      self.w = 119
      self.lev = 5
    if level == 9:
      self.k = 6528
      self.n = 8192
      self.w = 128
      self.lev = 5

  def CM_attack_key(self):
    '''
    Attack complexity of CM key.
    '''
    params = [0,0]
    init = 1000.0
    mem = 0
    params.append(init)
    params.append(mem)
    params.append(0)
    for p in range (2, 28+1,2):
      for p_prime in range (0, int(p/2)):
        p_pprime = int(p/2) - p_prime
        for l in range (0,151):
          q = comb(p,int(p/2))*comb(self.k+l-p,int(p/2))/comb(self.k+l,p)
          M = 2/q
          M *= M_overhead
          sol = (comb(self.k+l,p))/(np.longdouble)(2**l)
          temp = ISD_prime(self.n,self.k,self.w,p,p_prime, l,M, dual=0)
          if sol >= 1 and temp <= init and M/2**(5)>= sol and comb(self.k+l,
   p_pprime) <= M and log2(M) <M_UPPER_BOUND:
            # The constraint of M/2**(5)>= sol is added to ensure the
   probability is higher than (50%)
            init = temp
            params[0] = p
            params[1]= l
            params[2] = temp
            params[3] = log2(M)
            params[4] = p_prime

    print(params)
    print("Gauss cost:", log2(gauss(self.n,self.k,self.w,params[0],params[1])
    ))
    print("Merge set cost:", log2(C_sd(self.n,self.k,params[0],params[1],2**
    params[3], params[4])))
    print("check cost:", log2(C_check(self.n,self.k,params[0],params[1],2**
    params[3])))
```

```python
    print("C_label cost:", log2(C_label(self.n,self.k,params[0],params[1],2**params[3])))
    print("C_move cost:", log2(C_move(self.n,self.k,params[0],params[1],2**params[3], params[4])))
    print("C_combine cost:", log2(C_combine(self.n,self.k,params[0],params[1],2**params[3], params[4])))
    print("final check cost:", log2(C_final_check(self.n,self.k,params[0],params[1],2**params[3], params[4])))

def outputAttackBikeHqc():
  for level in range(1,6,2):
    print("level:", level)
    print("   ")
    print("BIKE key recovery:")
    bikePara = BIKE_security(level)
    print("n,k,w is:", bikePara.n, bikePara.k, bikePara.w)
    bikePara.BIKE_attack_key()
    print(" ")
    print("BIKE message recovery attacks:")
    print("n,k,w is:", bikePara.n, bikePara.k, bikePara.t)
    bikePara.BIKE_attack_message()
    print(" ")
    print("----------------------------------------------------------------")
    print(" ")
    print("HQC key recovery attacks:")
    hqcPara = HQC_security(level)
    print("n,k,w is:", hqcPara.n, hqcPara.k, hqcPara.w)
    hqcPara.HQC_attack_key()
    print(" ")
    print("=====================================================")

def outputAttackCM():
  for level in range(1,10,2):
    print("level:", level)
    print("------------------")
    print(" ")
    print("CM key recovery attacks:")
    CM_Para = CM_security(level)
    print("n,k,w is:",CM_Para.n, CM_Para.k, CM_Para.w)
    CM_Para.CM_attack_key()
    print(" ")
    print("=====================================================")




if __name__=='__main__':
  math.mp.dps = 20
  outputAttackBikeHqc()
  outputAttackCM()
```

B Supporting Tables.

Table B.1: Security parameters of the Classical McEliece scheme.

Category	n	k	ω
1	3488	2720	64
3	4608	3360	96
5	6688	5024	128
5	6960	5413	119
5	8192	6528	128

Table B.2: BIKE and HQC security parameters.

	Category	n	k	w
	1	24646	12323	134
BIKE (message)	3	49318	24659	199
	5	81946	40973	264
	1	24646	12323	142
BIKE (key)	3	49318	24659	206
	5	81946	40973	274
	1	35338	17669	132
HQC	3	71702	35851	200
	5	115274	57637	262

Table B.3: Bit security estimates of the Classic McEliece scheme. Here T is the log of the bit complexity, and $\hat{M}$ is the log of memory in bits. The parenthesis notation $(14, 83, 3)$ specifies that $2p = 14$ and $\ell = 83$,$p' = 3$.

	Category 1 ($n = 3488$)		Category 3 ($n = 4608$)		Category 5 ($n = 6688$)		Category 5 ($n = 6960$)		Category 5 ($n = 8192$)	
	T	$\hat{M}$	T	$\hat{M}$	T	$\hat{M}$	T	$\hat{M}$	T	$\hat{M}$
Prange	173	22	217	23	296	24	297	24	334	24
Stern	151	50	193	60	268	80	268	90	303	109
Both-May	143	88	182	101	250	136	249	137	281	141
May-Ozerov	141	89	180	113	246	165	246	160	276	194
BJMM	142	97	183	121	248	160	248	163	278	189
$\hat{M} \leq 60$	145	60	187	60	262	58	263	60	298	59
Our ISD, $\hat{M} \leq 60$										
$c_{\text{label}} = 2$	143.4	58	184.8	55	259.2	59	259.8	60	296.6	55
$c_{\text{label}} = 5$	144.6	58	186.0	55	260.4	59	261.0	60	297.6	55
	(14,83,3)		(12,75,2)		(12,78,2)		(12,79,2)		(10,68,2)	
Our ISD, any $\hat{M}$										
$c_{\text{label}} = 2$	143.4	58	184.4	65	256.7	91	256.8	92	290.6	95
$c_{\text{label}} = 5$	144.6	58	185.7	65	258.0	91	258.1	92	291.9	95
	(14,83,3)		(16,96,3)		(24,144,5)		(24,146,5)		(24,149,5)	