Manuscript submitted to AIMS Journal

doi:10.3934/amc.xx.xx.xx

ON THE COMPUTATIONAL HARDNESS OF THE CODE EQUIVALENCE PROBLEM IN CRYPTOGRAPHY

Alessandro Barenghi

Department of Electronics and Information
Politecnico di Milano

Jean-François Biasse

Department of Mathematics and Statistics
University of South Florida

Edoardo Persichetti

Department of Mathematical Sciences
Florida Atlantic University

Paolo Santini

Department of Information Engineering
Università Politecnica delle Marche

(Communicated by the associate editor name)

Abstract. Code equivalence is a well-known concept in coding theory. Recently, literature saw an increased interest in this notion, due to the introduction of protocols based on the hardness of finding the equivalence between two linear codes. In this paper, we analyze the security of code equivalence, with a special focus on the hardest instances, in the interest of cryptographic usage. Our work stems from a thorough review of existing literature, identifies the various types of solvers for the problem, and provides a precise complexity analysis, where previously absent. Furthermore, we are able to improve on the state of the art, providing more efficient algorithm variations, for which we include numerical simulation data. In the end, the goal of this paper is to provide a complete, single point of access, which can be used as a tool for designing schemes that rely on the code equivalence problem.

1. **Introduction.**

Code-based cryptography is one of the main areas of research aiming to provide security in a post-quantum scenario. The area is largely based on the well-known and understood Syndrome Decoding Problem (SDP), which leads to very good solutions for key establishment [1, 2, 13], but has shown to be far from optimal when designing signature schemes. With this in mind, a recent approach was presented in 2020, leveraging the code equivalence problem as the main hardness assumption; the result is the scheme known as LESS [8], a zero-knowledge protocol that can be converted to signature scheme via the Fiat-Shamir transformation [11].

2020 *Mathematics Subject Classification.* 11T71, 94A60.

Key words and phrases. Code Equivalence, Codes, Post-Quantum, Cryptography, Isometry.

Jean-Francois Biasse is supported by NIST grant 60NANB17D184 and NSF grant 183980; Edoardo Persichetti is supported by NSF grant 1906360.

The publication of LESS stirred the community into giving a deeper look at the hardness of code equivalence in practice; for example, shortly after the appearance of LESS, Beullens introduced an improved algorithm to solve the code equivalence problem [7] for certain specific instances. This had an immediate effect on LESS; new parameter choices were published in [6], alongside a variety of computational optimizations aimed at improving the protocol's efficiency. Such optimizations are possible, in the first place, as code equivalence can be seen as a particular type of cryptographic group action, thus drawing another line in the sand when compared with previous solutions from code-based cryptography. In the end, it is clear that the practical hardness of solving code equivalence is worthy of further investigation.

Our Contribution. In this work, we analyze and improve on the computational methods to solve instances of the Code Equivalence problem (CE) for which a solution exists, with a particular emphasis on the instances that are relevant to cryptography. This contribution is of fundamental interest, and it will have an important impact on future developments regarding schemes based on code equivalence, such as LESS. More specifically, we focus on the most efficient methods to solve the Linear Equivalence Problem (LEP) for codes over fields of cardinality $q \geq 5$, i.e. *the overwhelming majority* of the LEP instances. Note that CE efficiently reduces to LEP, which means that the techniques described in this paper apply to the resolution of almost all instances of CE. For the instances of LEP that we focus on, no efficient method applies, and the best known technique is due to Beullens [7].

- In Section 5 we study the costs of Leon's and Beullens' algorithms for LEP. We provide new arguments to measure the performance, that were not available previously, and that are essential to assess any further improvement.
- In Section 6, we describe a new technique for solving LEP, and we demonstrate that it is an improvement over the state of the art (Leon and Beullens)[1].

In Section 4, we also analyze Leon's and Beullens' algorithms for computationally hard instances of the Permutation Equivalence Problem (PEP). Instances of PEP are a narrow subset of the set of instances of LEP (almost all LEP instances are not a PEP instance), but there are efficient methods for generating hard PEP instances (i.e. when codes have large hull) which can be used in cryptography. Therefore, it is essential to have a precise analysis of the best computational methods for solving PEP in these special instances.

Organization of the paper. We begin in Section 2 by recalling some background notions about coding theory, as well as quantum search algorithms. In Section 3, we describe the code equivalence problem and give a high level overview of its hardness, and what are the main approaches for solvers. The permutation equivalence case is treated first, in Section 4; we then describe solvers for linear equivalence separately, in Section 5. Our improved technique is presented in Section 6. Finally, we conclude in Section 7.

2. Background.

We will use the conventions of Table 1 throughout the rest of the paper.

[1]When possible, we validate our analysis with numerical simulations; the employed Sage scripts are available at `https://github.com/paolo-santini/LESS_project`

a	a scalar
A	a set
$\boldsymbol{a}$	a vector
$\boldsymbol{A}$	a matrix
a	a function or relation
$\mathcal{A}$	an algorithm
$\boldsymbol{I}_n$	the $n \times n$ identity matrix
$[a;b]$	the set of integers $\{a, a+1, \ldots, b\}$
$\mathbb{U}(A)$	the uniform distribution over the set A
$\xleftarrow{\$} A$	sampling uniformly at random from A

Table 1. Notation used in this document.

We denote with $\mathbb{Z}_q$ the ring of integers modulo q, and with $\mathbb{F}_q$ the finite field of order q, as is customary; obviously, we have $\mathbb{Z}_q = \mathbb{F}_q$ when q is a prime. The multiplicative group of $\mathbb{F}_q$ is indicated as $\mathbb{F}_q^*$. Given a vector $\boldsymbol{a} \in \mathbb{F}_q^n$, we denote by $\mathsf{Values}(\boldsymbol{a})$ the ordered multiset formed by its entries. We denote with $\mathsf{Aut}(\mathbb{F}_q)$ the group of automorphisms of the field $\mathbb{F}_q$. The sets of vectors and matrices with elements in $\mathbb{Z}_q$ (resp. $\mathbb{F}_q$) are denoted by $\mathbb{Z}_q^n$ and $\mathbb{Z}_q^{m\times n}$ (resp. $\mathbb{F}_q^n$ and $\mathbb{F}_q^{m\times n}$). We write $\mathsf{GL}_k(q)$ for the set of invertible $k \times k$ matrices with elements in $\mathbb{F}_q$, or simply GL_k when the finite field is implicit. Let S_n be the set of permutations over n elements. Given a vector $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{F}_q^n$ and a permutation $\pi \in \mathsf{S}_n$, we write the action of π on $\boldsymbol{x}$ as $\pi(\boldsymbol{x}) = (x_{\pi(1)}, \ldots, x_{\pi(n)})$. Note that a permutation can equivalently be described as an $n \times n$ matrix with exactly one 1 per row and column. Analogously, for *linear isometries*, i.e. transformations $\tau = (\boldsymbol{v}; \pi) \in \mathbb{F}_q^{*n} \rtimes \mathsf{S}_n$, we write the action on a vector $\boldsymbol{x}$ as $\tau(\boldsymbol{x}) = (v_1 x_{\pi(1)}, \ldots, v_n x_{\pi(n)})$. Then, we can also describe these in matrix form as a product $\boldsymbol{Q} = \boldsymbol{D}\boldsymbol{P}$ where $\boldsymbol{P}$ is an $n \times n$ permutation matrix and $\boldsymbol{D} = \{d_{ij}\}$ is an $n \times n$ diagonal matrix with entries in $\mathbb{F}_q^*$. We denote with M_n the set of such matrices, usually known as *monomial* matrices.

2.1. **Coding Theory.** An $[n, k]$-*linear code* $\mathfrak{C}$ of length n and dimension $k \leq n$ over $\mathbb{F}_q$ is a k-dimensional vector subspace of $\mathbb{F}_q^n$. It can be represented by a full-rank matrix $\boldsymbol{G} \in \mathbb{F}_q^{k\times n}$ with rank k, called *generator matrix*, whose rows form a basis for the vector space, i.e. $\mathfrak{C} = \{\boldsymbol{u}\boldsymbol{G},\ \ \boldsymbol{u} \in \mathbb{F}_q^k\}$. Alternatively, a linear code can be represented as the kernel of a full-rank matrix $\boldsymbol{H} \in \mathbb{F}_q^{(n-k)\times n}$, known as *parity-check matrix*, i.e. $\mathfrak{C} = \{\boldsymbol{x} \in \mathbb{F}_q^n : \boldsymbol{H}\boldsymbol{x}^T = 0\}$. For both representations, there may exist a standard choice, called *systematic form*, which corresponds, respectively, to $\boldsymbol{G} = (\boldsymbol{I}_k \mid \boldsymbol{M})$ and $\boldsymbol{H} = (-\boldsymbol{M}^T \mid \boldsymbol{I}_{n-k})$. Generator (resp. parity-check) matrices in systematic form can be obtained very simply by calculating the row-reduced echelon formstarting from any other generator (resp. parity-check) matrix. We denote such a procedure by sf. The parity-check matrix is important also as it is a generator for the *dual code*, defined as the set of words that are orthogonal to the code, i.e. $\mathfrak{C}^\perp = \{\boldsymbol{y} \in \mathbb{F}_q^n\ :\ \forall \boldsymbol{x} \in \mathfrak{C},\ \ \boldsymbol{x} \cdot \boldsymbol{y}^T = 0\}$. Codes that are contained in their dual, i.e. $\mathfrak{C} \subseteq \mathfrak{C}^\perp$, are called *self-orthogonal* or *weakly self-dual*, and codes that are equal to their dual, i.e. $\mathfrak{C} = \mathfrak{C}^\perp$, are called simply *self-dual*.

We now proceed by recalling some well known definitions and results, which we will frequently use in the rest of the paper.

Definition 2.1. Let $\mathfrak{C} \subseteq \mathbb{F}_q^n$ be a code with dimension k. We define the permutation automorphism group of $\mathfrak{C}$ as

$$\mathsf{Aut}_{\mathsf{S}_n}(\mathfrak{C}) = \{\pi \in \mathsf{S}_n \mid \pi(\mathfrak{C}) = \mathfrak{C}\}.$$

Analogously, we define the monomial automorphism group of $\mathfrak{C}$ as

$$\mathsf{Aut}_{\mathsf{M}_n}(\mathfrak{C}) = \{\mu \in \mathsf{M}_n \mid \mu(\mathfrak{C}) = \mathfrak{C}\}.$$

Note that, if $\pi \in \mathsf{Aut}_{\mathsf{S}_n}$, then for any $\boldsymbol{G}$ that generates $\mathfrak{C}$, there must exist $\boldsymbol{S} \in \mathsf{GL}_k$ such that $\boldsymbol{G} = \boldsymbol{S}\pi(\mathfrak{C})$. Clearly, analogous relation applies to the monomial automophism group.

Definition 2.2 (Code support)**.** For a linear code $\mathfrak{C} \subseteq \mathbb{F}_q^n$, we define the support $\mathsf{Supp}(\mathfrak{C}) \subset \{1, \ldots, n\}$ as the set of indexes i for which there is at least one codeword $\boldsymbol{c} \in \mathfrak{C}$ such that $c_i \neq 0$.

We now introduce another concept which will be fundamental for the analysis we develop in this paper.

Definition 2.3. Let $\mathfrak{C} \subseteq \mathbb{F}_q^n$ be a linear code with dimension k. A k'-dimensional subcode $\mathfrak{C}'$ of $\mathfrak{C}$ is a k'-dimensional vector space that can be generated by k' codewords of $\mathfrak{C}$. The set of all k'-dimensional subcodes with support size w is indicated as $A_w^{(k')}(\mathfrak{C})$. We refer to such a set as the *k'-dimensional Hamming sphere with radius w*.

The concept of code support can be deemed as a direct generalization of the notion of support for a vector (i.e. the set of indexes pointing at non null coordinates). In particular, for a vector, the cardinality of its support is referred to as *Hamming weight*:

$$\mathsf{wt}(\boldsymbol{a}) : \mathbb{F}_q^n \mapsto \mathbb{N} \quad := \quad \mathsf{wt}(\boldsymbol{a}) = |\mathsf{Supp}(\boldsymbol{a})|.$$

Remark 1. For $k' = 1$, the set $A_w^{(k')}(\mathfrak{C})$ contains all the codewords that have Hamming weight w and are distinct, even when considering multiple scalars. To ease the notation, we will refer to such a set as $A_w(\mathfrak{C})$.

We now continue with some properties of linear codes, in terms of subcodes having a desired support size.

Lemma 1. Let $\mathfrak{C} \subseteq \mathbb{F}_q^n$ be a k-dimensional linear code. Then, the number of subcodes of $\mathfrak{C}$ with dimension $k' \leq k$ is given by

$$\begin{bmatrix} k \\ k' \end{bmatrix}_q = \frac{(q^k - 1)\ldots(q^k - q^{k'-1})}{(q^{k'} - 1)\ldots(q^{k'} - q^{k'-1})} = \prod_{i=0}^{k'-1} \frac{q^k - q^i}{q^{k'} - q^i}.$$

When a code is picked at random, it is safe to assume that the contained k'-dimensional subcodes are random as well, that is, uniformly distributed over the set of all possible k'-dimensional vector subspaces of $\mathbb{F}_q^n$. Starting from this consideration (which is a standard assumption in coding theory), we can count the number of subcodes having a certain support size.

Proposition 1. Let $\mathfrak{C} \subseteq \mathbb{F}_q^n$ be a random linear code with dimension k. Then, the average number of subcodes with dimension k' and support size w is bounded from above by

$$\frac{(q^{k'} - 1)^w \binom{n}{w}}{\prod_{i=0}^{k'-1} (q^{k'} - q^i)} \frac{\left[\begin{smallmatrix} k \\ k' \end{smallmatrix}\right]_q}{\left[\begin{smallmatrix} n \\ k' \end{smallmatrix}\right]_q}.$$

Proof. Let $J \subseteq \{1, \cdots, n\}$ of size w, and consider the codes with dimension k' and whose support is exactly J. We can upper bound the number of such codes by $\frac{(q^{k'}-1)^w}{\prod_{i=0}^{k'-1} q^{k'}-q^i}$. Indeed, $(q^{k'}-1)^w$ counts the number of matrices that have no null column among those indexed by J, while all the other ones are null; we divide this number by $\prod_{i=0}^{k'-1} q^{k'}-q^i$ to take into account all possible bases. Note that this is an upper bound since not all the considered matrices will have full rank k'. Since we have $\binom{n}{w}$ choices for J, we obtain $\frac{\binom{n}{w}(q^{k'}-1)^w}{\prod_{i=0}^{k'-1}(q^{k'}-q^i)}$ as an upper bound for the number of k'-dimensional subcodes of $\mathbb{F}_q^n$ with support size w. We now assume that all of the $\left[{k \atop k'} \right]_q$ subcodes of $\mathfrak{C}$ with dimension k' are randomly and uniformly picked among the set of $\left[{n \atop k'} \right]_q$ subspaces of $\mathbb{F}_q^n$ with dimension k'. Then, the probability that a specific k'-dimensional subcode has support size w is $\frac{(q^{k'}-1)^w\binom{n}{w}}{\prod_{i=0}^{k'-1}(q^{k'}-q^i)}\frac{1}{\left[{n \atop k'} \right]_q}$. Multiplying the above probability by $\left[{k \atop k'} \right]_q$ (that is, the number of subcodes in $\mathfrak{C}$ with dimension k') we obtain the estimate. $\square$

Remark 2. When $k' = 1$, the number of subcodes is equal to that of codewords with Hamming weight w (without counting scalar multiples). For simplicity, we will denote this quantity as N_w, and have

$$N_w = N_w^{(1)} = \binom{n}{w}(q-1)^{w-1}\frac{q^k-1}{q^n-1}.$$

Remark 3. When $k' = 2$, we can improve upon the upper bound of Proposition 1 and obtain the average number of subcodes with support size w. To do this, it is enough to subtract from $(q^2-1)^w$ (that is, the number of matrices with w non null columns) the number of matrices that generate a one-dimensional space. Notice that these matrices are such that both rows have weight w and are identical up to a scalar multiplication; hence, they can be counted as $(q-1)^w(q-1) = (q-1)^{w+1}$. Consequently, we can set $\frac{(q^2-1)^w-(q-1)^{w+1}}{(q^2-1)(q^2-1)}$ as the number of subcodes of $\mathbb{F}_q^n$ with dimension 2 and support J of size w. Plugging this estimate into the proof of the above Proposition, we can estimate the average number of support size w subcodes of a random code as

$$N_w^{(2)} = \binom{n}{w}\frac{(q^2-1)^w-(q-1)^{w+1}}{(q^2-1)(q^2-q)}\frac{\left[{k \atop 2} \right]_q}{\left[{n \atop 2} \right]_q}.$$

2.2. **ISD algorithms.**

Information Set Decoding (ISD) is the best technique to produce low weight codewords in a given code. There is a vast literature on ISD algorithms, most of which apply to the binary case; an extensive review can be found for example in [4]. For the more general, q-ary case (which is of interest to us), the work of Peters [14] is usually considered the go-to reference. In this paper, we will denote as $C_{ISD}(q,n,k,w)$ the cost of finding a specific codeword with weight w, in a code with length n and dimension k, defined over $\mathbb{F}_q$. In other words, if $\boldsymbol{c}$ is a codeword with weight w, then $C_{ISD}(q,n,k,w)$ is the cost to have an ISD routine return exactly $\boldsymbol{c}$. To assess $C_{ISD}(q,n,k,w)$, we rely on the analysis in [14]. Note that ISD is a randomized algorithm and, in case a code contains $N_w > 1$ codewords with weight w, then ISD will randomly return one of these codewords. In such a case, the complexity to find a codeword with weight w can be assessed as $\frac{C_{ISD}(q,n,k,w)}{N_w}$.

3. **The Code Equivalence Problem.** The concept of *equivalence* between two codes, in its most general formulation, is defined as follows.

Definition 3.1 (Code Equivalence)**.** We say that two linear codes $\mathfrak{C}_1$ and $\mathfrak{C}_2$ are *equivalent*, and write $\mathfrak{C}_1 \sim \mathfrak{C}_2$, if there exist a field automorphism $\alpha \in \mathsf{Aut}(\mathbb{F}_q)$ and a linear isometry $\tau = (\boldsymbol{v};\pi) \in \mathbb{F}_q^{*n} \rtimes \mathsf{S}_n$ that map $\mathfrak{C}_1$ into $\mathfrak{C}_2$, i.e. such that $\mathfrak{C}_2 = \tau(\alpha(\mathfrak{C}_1)) = \{\boldsymbol{y} \in \mathbb{F}_q^n \ : \boldsymbol{y} = \tau(\alpha(\boldsymbol{x})),\ \boldsymbol{x} \in \mathfrak{C}_1\}$.

Clearly, if $\mathfrak{C}_1$ and $\mathfrak{C}_2$ are two codes with generator matrices $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$, respectively, it holds that

$$\mathfrak{C}_1 \sim \mathfrak{C}_2 \iff \exists(\boldsymbol{S};(\alpha,\boldsymbol{Q})) \in \mathsf{GL}_k \rtimes (\mathsf{Aut}(\mathbb{F}_q) \times \mathsf{M}_n) \text{ s.t. } \boldsymbol{G}' = \boldsymbol{S}\alpha(\boldsymbol{G}\boldsymbol{Q}).$$

The notion we just presented is usually known as *semilinear equivalence* and it is the most generic. If the field automorphism is the trivial one (i.e. $\alpha = id$), then the notion is simply known as *linear equivalence.* If, furthermore, the monomial matrix is a permutation (i.e. $\boldsymbol{Q} = \boldsymbol{D}\boldsymbol{P}$ with $\boldsymbol{D} = \boldsymbol{I}_n$), then the notion is known as *permutation equivalence.* Note that, in cryptographic applications (e.g. [8, 6]), the fields considered are always prime, and therefore the last two notions are the only ones of interest to us. Finally, we state the following computational[2] problem.

Problem 1 (Code Equivalence)**.** *Let $\boldsymbol{G}_1, \boldsymbol{G}_2 \in \mathbb{F}_q^{k\times n}$ be two generator matrices for two linearly equivalent codes $\mathfrak{C}_1$ and $\mathfrak{C}_2$. Find two matrices $\boldsymbol{S} \in \mathsf{GL}_k$ and $\boldsymbol{Q} \in \mathsf{M}_n$ such that $\boldsymbol{G}_2 = \boldsymbol{S}\boldsymbol{G}_1\boldsymbol{Q}$.*

We normally refer, respectively, to *permutation equivalence problem (PEP)* or *linear equivalence problem (LEP)*, according to the notion of code equivalence considered, or simply to the *code equivalence problem* where such distinction is not important.

3.1. **High Level Hardness Overview.** As proven in [15], the permutation equivalence problem is unlikely to be NP-complete, since this property would imply a collapse of the polynomial hierarchy. Yet, while the problem can be efficiently solved for some families of codes, there are many instances that, after almost 40 years of study, are still intractable. In the remainder of the paper, we analyze the best known solvers for the code equivalence problem. We first deal with the case of permutation equivalence, and report the complexity of all techniques to solve this problem. Then, we show how these techniques adapt to the case of linear equivalences.

We begin by recalling a trivial property of code equivalence.

Proposition 2. Let $\mathfrak{C}_1, \mathfrak{C}_2 \subseteq \mathbb{F}_q^n$ be two linear codes with dimension k, and let $\mathfrak{C}_1^\perp$, $\mathfrak{C}_2^\perp$ be their duals. Then

i. if $\pi \in \mathsf{S}_n$ is such that $\pi(\mathfrak{C}_1^\perp) = \mathfrak{C}_2^\perp$, then also $\pi(\mathfrak{C}_1) = \mathfrak{C}_2$;
ii. if $\tau \in \mathsf{M}_n$ is such that $\tau(\mathfrak{C}_1^\perp) = \mathfrak{C}_2^\perp$, then also $\tau'(\mathfrak{C}_1) = \mathfrak{C}_2$, where τ' is derived from τ by taking the inverses of the scaling factors.

The above proposition is crucial to understand the hardness of solving the code equivalence problem. Indeed, the problem can equivalently be solved by looking at the given codes, or at their duals. For the sake of simplicity, in this work, we will describe all the algorithms and procedures by considering solely the codes initially

[2]Note that this problem is traditionally formulated as a decisional problem in literature, yet for our purposes it is more natural to present here the search version.

given; to derive the corresponding complexity for the attack on the duals, it is enough to replace k with $n-k$ in all the provided formulas.

To avoid studying vacuously hard instances (i.e., those represented by codes that are not equivalent), we will always consider the case in which at least a solution is guaranteed to exist. Namely, we consider that:

- the code $\mathfrak{C}_1$ is chosen at random;
- for PEP, we have $\mathfrak{C}_2 \overset{\$}{\leftarrow} \{\pi(\mathfrak{C}_1) \mid \pi \in \mathsf{S}_n\}$;
- for LEP, we have $\mathfrak{C}_2 \overset{\$}{\leftarrow} \{\tau(\mathfrak{C}_1) \mid \tau \in \mathsf{M}_n\}$.

Note that the number of solutions to PEP is equal to the size of the automorphism group. Indeed, if π solves PEP and σ is such that $\sigma(\mathfrak{C}_1) = \mathfrak{C}_1$, then we have another solution to PEP by combining π and σ. Clearly, the same considerations hold for LEP. To the best of our knowledge, the behaviour of the autormorphism groups of random codes under this perspective has never been formally studied. However, it is essentially folklore that these groups is trivial. Consequently, in our study we are going to make use of the following structural assumption.

Assumption 1. *We assume that the permutation and monomial automorphism groups of the considered codes are trivial.*

As a result of the above assumption, all the code equivalence instances we consider admit only one solution.

3.1.1. *The easy cases.* We begin our analysis by discussing algorithms that treat special cases, leading to very efficient solvers. The first such algorithm is the Support Splitting Algorithm (SSA), introduced by Sendrier [17]. This solver is based on the idea of *signature function*, i.e. a function $\mathcal{S}$ that fixes the action of the permutation on each position in the code. A signature function is said to be *fully discriminant* if it returns a different value in each position, and this allows to reveal the permutation linking the two codes. The signature function proposed by Sendrier in [17] is based on the *hull space* of a code, that is, the intersection between a code and its dual, for which the *weight enumerator* is computed. In particular, to create a dependence between the signature value and the code positions, one can *puncture* the code, i.e. remove coordinates from the codewords. Putting these considerations together, in [17, Section 5.2] Sendrier proposes to build a signature as

$$\mathcal{S}(\mathfrak{C}_i) := \left\{ \mathsf{Wef}\Big(\mathfrak{H}(\mathfrak{C}_{\setminus i})\Big) \, , \, \mathsf{Wef}\Big(\mathfrak{H}(\mathfrak{C}^{\perp}_{\setminus i})\Big) \right\},$$

where $\mathfrak{C}_{\setminus i}$ is the code obtained from $\mathfrak{C}$ punctured in position i, $\mathfrak{H}$ denotes the hull and Wef denotes the Weight Enumerator Function. The hull computation requires simple linear algebra, and comes with a cost of $O(n^3)$ operations in the finite field. To compute the weight enumerator of a code, one usually needs to enumerate all of its codewords: assuming that the hull has dimension h, we can use $O(nq^h)$ as an estimate for the cost of each Wef computation. On the other hand, heuristically, we observe that using $\ln(n)$ refinements is enough to obtain a fully discriminant signature. In the end, the complexity of SSA can be estimated as $O\left(n^3 + n^2 q^h \ln(n)\right)$. Thus, the hull dimension plays a crucial role in the analysis of the performance of SSA. For random codes, this dimension is with high probability equal to a small constant [18], de facto making SSA a polynomial-time solver for PEP. On the other hand, SSA is very inefficient for codes that have a large hull.

This is, for instance, the case of (weakly) self-dual codes, for which SSA can be made arbitrarily hard by choosing codes with a sufficiently large dimension. SSA can be extended to solve the linear equivalence problem as well; however, in this case, the algorithm is less efficient. In fact, such an adaptation requires applying SSA to the *closure* of the code, i.e. the linear code defined as $\{\boldsymbol{c}\otimes\boldsymbol{a},\ \ \boldsymbol{c}\in\mathfrak{C}\}$, where $\boldsymbol{a}=(a_1,\cdots,a_{q-1})$ is any ordering of the non-zero elements of $\mathbb{F}_q$. A fundamental point is that, for $q\geq 5$, the closure of a code is always weakly-self dual, and thus has a hull of maximum dimension, leading to exactly the hardest instances for SSA to solve. These results are corroborated by the analysis in [16].

Note that SSA trivially fails in the case of codes with an empty hull. In this case, however, another approach is possible. In 2019, Bardet et al. [5] proposed a new method to solve the permutation equivalence problem, which fully exploits the connection between the permutation equivalence problem and the notion of *graph isomorphism*. The core idea of [5] is to reduce code equivalence to an instance of the *Weighted Graph Isomorphism (WGI) problem*. This is done by building matrices of the form $\boldsymbol{A}_{\mathfrak{C}_i}=\boldsymbol{G}_i^\top\left(\boldsymbol{G}_i\boldsymbol{G}_i\right)^{-1}\boldsymbol{G}_i$ from the codes considered, and observing that $\boldsymbol{A}_{\mathfrak{C}_1}=\boldsymbol{P}^\top\boldsymbol{A}_{\mathfrak{C}_1}\boldsymbol{P}$ allows to recover the permutation $\boldsymbol{P}$ that connects the two codes. Indeed, $\boldsymbol{A}_{\mathfrak{C}_1}$ and $\boldsymbol{A}_{\mathfrak{C}_2}$ are interpreted as the adjacency matrices of two graphs, and hence can be given as input to some routine which solves the WGI problem. Given that, to compute $\boldsymbol{A}_{\mathfrak{C}_1}$ and $\boldsymbol{A}_{\mathfrak{C}_2}$, only $O(n^{2.373})$ operations in the finite field are required (this is essentially the cost of matrix inversion), we have that this approach gives a complexity of

$$O\left(n^{2.373}C_{WGI}(n)\right),$$

where $C_{WGI}(n)$ denotes the complexity of a solver for the weighted graph isomorphism problem. Note that this problem can be solved, for many classes of graphs, with very efficient algorithms. Furthermore, Babai's recent breakthrough paper [3] shows that the problem can be solved, in the worst case, with quasi-polynomial complexity. Hence, even in the worst case scenario, this solver runs in a time that is quasi-polynomial in the code length, on codes that have a trivial hull. For the more general case of codes with a non-trivial hull, the reduction from graph isomorphism works in a different way. In this case the complexity scales heavily with the dimension of the hull and thus the solver is, in practice, much less efficient; a proof of this fact can be found in [5, Theorem 10].

Finally, an algebraic approach was investigated in [16], where the author shows how it is possible to solve permutation equivalence by modeling it as a quadratic system. When the hull is trivial, it is possible to add several linear equations (through a technique called *block linearization*), which makes the system very easy to solve. However, in the general case of a non-trivial hull, the methods proposed by the author (using shortened codes or searching for the closest vector in the code) always end up in exponential complexity; for example, the latter scales proportionally to q^k. It follows that, as mentioned by the author himself, this approach can be deemed efficient only for the case of trivial hulls, once again.

To conclude this first section, we clarify the main takeaway to the reader. All the methods described above provide efficient solvers for very specific cases (small or trivial hulls); however, for codes with large hulls, these methods become quickly impractical. More to the point: when considering code equivalence in cryptography, it is easy to avoid these attacks. In fact, for the linear equivalence problem, it is enough to consider random codes defined over a large enough alphabet ($q \geq 5$), and then the value $q^h = q^k$ is already large enough for any realistic choice of code parameters. On the other hand, if one wants to use permutation equivalence, choosing a weakly-self dual code is sufficient to guarantee maximum hull dimension. All these considerations are already taken into account in the original LESS work, and constitute essentially just a set of "best practices", to be considered when designing a cryptosystem based on code equivalence. We now move on to summarizing algorithms that are relevant to the analysis of such systems.

3.1.2. *Solvers for hard instances.* There are other algorithms that are able to solve the hard instances described above, for which the previous solvers are ineffective. This is because the complexity of such algorithms does not depends on the size of the hull. Instead, the algorithms are based on a different observation, namely, that both permutation and monomial transformations preserve the Hamming weight distribution of the codewords. In particular, if two codes $\mathfrak{C}_1$ and $\mathfrak{C}_2$ are linked by some permutation or monomial transformation, say τ, then we have that for any subset of weight-w codewords $A_1 \subseteq \mathfrak{C}_1$, there must exist some subset of weight-w codewords $A_2 \subseteq \mathfrak{C}_2$ such that $\tau(A_1) = A_2$. Starting from this basic reasoning, the goal becomes that of finding sets of codewords that i) can efficiently been computed, and ii) have enough structure to allow for the reconstruction of τ.

Leon's algorithm [12], which dates as the first technique to solve code equivalence, chooses A_1 and A_2 as the set of all codewords having some low Hamming weight w. The choice of w is crucial to determine the algorithm effectiveness. On the one hand, in fact, if w is too low then A_1 and A_2 may have not enough structure (i.e., they contain very few codewords) so that reconstructing τ may not be possible. Yet, low-weight codewords can be found with ISD algorithms, with a cost that is significantly smaller than that of enumerating the whole code. On the other hand, if w is too high, the number of codewords in A_1 and A_2 may become too high, so that determining the sets becomes too time-consuming.

In a separate work [10], Feulner describes a method for computing the automorphims group of a linear code, looking for canonical representatives via search trees with several refinements. The author gives no concrete analysis about the algorithm's complexity, or even a heuristic formula. Yet, he mentions that, as the consequence of practical experiments, the algorithm has essentially the same cost as Leon's algorithm (the difference is within a factor of two). Technically, this may be due to the fact that the algorithm still uses codeword enumeration, which requires exponential time. As the author himself recognizes, the algorithm may be improved (in general, and not only for some specific codes) by using easier-to-compute signature functions, such as considering all codewords with a certain weight. But then, the complexity of the algorithm would be the same as the one we are considering for Leon's algorithm. In the end, we deem that the algorithm does not constitute a crucial improvement upon Leon's algorithm. This is consistent with literature, where the algorithm is mentioned several times in the same regard [16, 9].

Recently, Beullens [7] proposed an algorithm which is able, in some cases, to improve over Leon's algorithm. For the permutation equivalence case (i.e., when $\tau \in \mathsf{S}_n$), one observes that the multisets formed by the entries of the codewords are preserved as well. Hence, one can construct the sets A_1 and A_2 by considering pairs of codewords (one in A_1, one in A_2) having the same multisets of entries. To avoid too many collisions (which would make the algorithm perform worse than Leon's), one can consider only the codewords having some moderately low weight. As a key observation, Beullens has shown how it is not necessary to find all of these matching codewords (differently from what one does in Leon's algorithm).

For the remainder of this work, we will focus our analysis on algorithms of this second type, as they constitute the most efficient attack avenue for cryptographic schemes based on code equivalence.

4. **Solvers for Hard Permutation Equivalence Instances.** In this section we recall the algorithms for the permutation equivalence problem, whose complexity does not depend on the hull size, that we anticipated in the previous section.

Leon's Algorithm. Chronologically, the first method capable of solving the code equivalence problem is due to Leon [12], and is based on the following reasoning. Let $\mathfrak{C}_1$ and $\mathfrak{C}_2$ be two linear codes with length n and dimension k, and $\pi \in \mathsf{S}_n$ such that $\pi(\mathfrak{C}_1) = \mathfrak{C}_2$. Let X be a set of codewords picked from $\mathfrak{C}_1$. Then, there must exist a set Y formed by codewords of $\mathfrak{C}_2$ and such that $\pi(X) = Y$: among all the maps from X to Y, there must necessarily also be those mapping $\mathfrak{C}_1$ into $\mathfrak{C}_2$. In [12], Leon proposes an algorithm that constructs the ensemble of permutations between two sets, with a running time that is polynomial in the cardinality of the sets. Starting from the observation that permutations preserve the Hamming weight, Leon suggests to form X and Y using the codewords with a properly low weight w. Let $A_w(\mathfrak{C}_1)$ and $A_w(\mathfrak{C}_2)$ denote such sets, and $\mathsf{Mor}_{\mathsf{S}_n}\big(A_w(\mathfrak{C}_1), A_w(\mathfrak{C}_2)\big)$ be the set of all permutations $\pi \in \mathsf{S}_n$ such that $\pi\big(A_w(\mathfrak{C}_1)\big) = A_w(\mathfrak{C}_2)$. In a nutshell, Leon's algorithm operates as follows:

1. compute $A_w(\mathfrak{C}_1)$ and $A_w(\mathfrak{C}_2)$;
2. construct $\mathsf{Mor}_{\mathsf{S}_n}\big(A_w(\mathfrak{C}_1), A_w(\mathfrak{C}_2)\big)$;
3. check if there exists $\pi \in \mathsf{Mor}_{\mathsf{S}_n}\big(A_w(\mathfrak{C}_1), A_w(\mathfrak{C}_2)\big)$ such that $\pi(\mathfrak{C}_1) = \mathfrak{C}_2$.

As Leon proves in the original paper, the complexity of the second and third steps is polynomial in the cardinality of $A_w(\mathfrak{C}_1)$ and $A_w(\mathfrak{C}_2)$, which we estimate with N_w as in Proposition 1. This also allows us to properly choose the value of w. Indeed, N_w grows (exponentially) with w: when w is too high, N_w may become so large that the first and second steps of the algorithm become too time-consuming. On the other hand, if w is too low, then the sets $A_w(\mathfrak{C}_1)$ and $A_w(\mathfrak{C}_2)$ are rather small and do not possess enough structure, in the sense that there may exist a very large number of maps from $A_w(\mathfrak{C}_1)$ to $A_w(\mathfrak{C}_2)$.

Heuristically, optimal values of w are those that are slightly larger than the minimum distance of the codes (which can be estimated with the Gilbert-Varshamov distance). Indeed, this normally guarantees that the sets $A_w(\mathfrak{C}_1)$ and $A_w(\mathfrak{C}_2)$ are moderately small and, at the same time, contain a sufficient number of codewords. A lower bound on the complexity of Leon's algorithm can be estimated as follows.

Proposition 3 ([7]). Let $\mathfrak{C}_1 \subseteq \mathbb{F}_q^n$ be a random code with dimension k, $\pi \xleftarrow{\$} \mathsf{S}_n$ and $\mathfrak{C}_2 = \pi(\mathfrak{C}_1)$. The cost of Leon's algorithm, running with parameter $w \in \mathbb{N}$, $w \leq n$, can be estimated[3] as

$$O\Big(\ln(N_w) C_{ISD}(q,n,k,w)\Big).$$

For the sake of completeness, the proof of Proposition 3 is reported in Appendix A, where we additionally (as a new result) derive a theoretical bound on the required value for w. In practice, the attack is normally optimized when w is slightly larger than the minimum distance (say, by 1 or 2).

Beullens' Algorithm. In a recent work [7], Beullens introduced a novel approach to solve the code equivalence problem. The algorithm can be thought of as a refinement of Leon's algorithm, in which one tries to reduce the computational complexity by avoiding to compute the whole set of codewords with some fixed weight. The algorithm is based on the simple, but effective, intuition that permutations preserve also the multiset entries. Exploiting this observation, one can easily see how Leon's algorithm can be improved, by reducing the size of X and Y. In a nutshell, Beullens' algorithm works by first finding a subset of codewords with weight w from each code, and then searches for collisions among codewords having the same entries multiset. Each found collision is then used to piece-wise reconstruct the action of the permutation: if $\boldsymbol{x} \in \mathfrak{C}_1$ and $\boldsymbol{y} \in \mathfrak{C}_2$ have the same entries multiset and $x_i \neq y_j$, then we guess $\pi(i) \neq j$. When the number of collisions is sufficiently high, one has enough information to fully retrieve the permutation π. As done in [7], we can consider that the algorithm is successful whenever the number of collisions is approximately $2n\ln(n)$.

Note that, differently from Leon, Beullens' algorithm is probabilistic, since it fails in case i) bad collisions are found (i.e., codewords $\boldsymbol{x}$ and $\boldsymbol{y}$ that have the same entries multiset but $\boldsymbol{y} \neq \pi(\boldsymbol{x})$), and ii) the number of collisions is too low. The analysis of these cases and a precise cost estimate (which is missing in Beullens' original paper) are based on several technical aspects, which we detail in Appendix B. A compact and simple analysis of Beullens' algorithm is encapsulated in the following Proposition.

Proposition 4. The time complexity of Beullens' algorithm, running with parameters L and w such that i) $L = \sqrt{2N_w n\ln(n)}$ and ii) $(1-1/N_w)(q-1)L^2\binom{w+q-3}{w-1}^{-1} < 1$ is

$$O\left(\sqrt{\frac{n\ln(n)}{N_w}} C_{ISD}(q,n,k,w)\right).$$

Condition i) implies that we do not find all codewords with weight w, otherwise the algorithm would reduce to Leon. Condition ii) sets a lower bound on the number of codewords we need to find, in order to have enough information to run the permutation recovery algorithm. Finally, condition iii) expresses the fact that bad collisions do not happen.

[3]Here we use the same estimate derived in [7, Section 2.2], which corresponds to a lower bound for the actual complexity since the cost of steps 2 and 3 is neglected. In other words, the proposition takes into account only the cost of the codewords enumeration phase.

5. **Solvers for Hard Linear Equivalence Instances.**

In this section, we recall the procedure of Beullens, which is a starting point for our new algorithm. We provide lower bounds on the complexities of Leon's and Beullens' algorithms to solve LEP, in order to make a comparison with our own estimate. These algorithms have several features which are similar to the ones we have already analyzed in the previous section; yet, using monomial transformations instead of permutations lead, in some cases, to radical differences. Analogously to what we have done for the permutation case, we will study LEP under the hypothesis that a solution always exists and, recalling Assumption 1, that the monomial isomorphism group is trivial. As a consequence, we have that the only solutions for the LEP instance represented by $(\mathfrak{C}_1, \mathfrak{C}_2)$, with $\mathfrak{C}_2 = \tau(\mathfrak{C}_1)$, are the monomial τ and its scalar multiples. Note that all of these transformations use the same permutation, and differ only for the scaling coefficients.

5.1. **Leon's algorithm.**

Leon's algorithm can be used to solve the linear equivalence problem, with an operating procedure that is essentially identical to the one we have already discussed in Section 4. The only difference is in the fact that, after the codewords enumeration, one searches for a monomial matrix instead of a permutation. When the value of w is properly chosen, this can be reconstructed in polynomial time, so that the bottleneck in the computational complexity is (again) in the codewords enumeration. Hence, also in this case, we can rely on Proposition 3 to have an estimate for the cost of the algorithm.

5.2. **Beullens' algorithm.**

In [7], Beullens proposed a second algorithm, to solve the linear equivalence problem. The algorithm principle is analogous to the PEP case, but some modifications are necessary, since monomial transformations do not preserve the multisets of codewords entries. To overcome this issue, Beullens first observes that if $\tau \in \mathsf{M}_n$ is such that $\tau(\mathfrak{C}_1) = \mathfrak{C}_2$, then for any subcode $\mathfrak{B}_1 \subseteq \mathfrak{C}_1$ there must exist a subcode $\mathfrak{B}_2 \subseteq \mathfrak{C}_2$ such that $\tau(\mathfrak{B}_1) = \mathfrak{B}_2$. Considering subcodes of small dimension and small support (we expect that very few such subcodes exist) instead of low weight codewords, we have that the same procedure as the one to solve permutation equivalence (plus some tweaks) can retrieve the secret monomial transformation. In particular, as in [7], we analyze the algorithm when two-dimensional subcodes are employed. Note that, to obtain an algorithm solving linear equivalence, we need the following three tweaks:

1. the codewords matching procedure shown in Algorithm 4 is replaced with an algorithm that produces colliding subcodes. To do this, we need to i) tweak ISD so that it returns subcodes with support size w, and ii) introduce the function $\mathsf{Lex}^{(2)}$ to take into account two-dimensional spaces. For an example of how such functions may be computed, we refer the reader to Appendix C while, for the sake of completeness, we report the full subcodes collisions procedure in Algorithm 1. Note that computing $\mathsf{Lex}^{(2)}$ has a cost of $O\left(n(q^2-1)(q^2-q)\right)$;
2. the list P produced by Algorithm 1 contains pairs $\{\boldsymbol{X}, \boldsymbol{Y}\} \in \mathbb{F}_q^{2\times n} \times \mathbb{F}_q^{2\times n}$ for which $\mathsf{Lex}^{(2)}(\boldsymbol{X}) = \mathsf{Lex}^{(2)}(\boldsymbol{Y})$;
3. in the reconstruction phase, one first finds the permutation, and then recovers the scaling factors. To have an efficient permutation recovery method, we can proceed in a way that is analogous to that of the permutation equivalence case; again, for the sake of completeness, we have reported the procedure in

Algorithm 2. Once the permutation has been recovered, the scaling factors can be found in many efficient ways. For instance, the permutation can be applied to a generator for $\mathfrak{C}_1$, obtaining $\boldsymbol{G}'$. Then, we choose a parity-check matrix for $\mathfrak{C}_2$, and aim to determine a non-singular diagonal matrix $\boldsymbol{D}$ such that $\boldsymbol{G}'\boldsymbol{D}\boldsymbol{H}_2^\top = \boldsymbol{0}$. This linear system has $k(n-k)$ equations for n unknowns, so that in general it is over constrained and can be easily solved. The n non-null entries of $\boldsymbol{D}$ are the unknown scaling coefficients $\boldsymbol{v}$, which are used to retrieve the desired monomial as $\pi \rtimes \boldsymbol{v}$.

Algorithm 1: Algorithm to find and match subcodes

```
Data: Number of subcodes $L \in \mathbb{N}$, support size $w \in \mathbb{N}$, ISD routine
Input: linear codes $\mathfrak{C}_1, \mathfrak{C}_2 \subseteq \mathbb{F}_q^n$ with dimension $k$
Output: list $P$ containing pairs $(\boldsymbol{X}, \boldsymbol{Y}) \in \mathbb{F}_q^{2\times n} \times \mathbb{F}_q^{2\times n}$, such that
        $\mathsf{Lex}^{(2)}(\boldsymbol{X}) = \mathsf{Lex}^{(2)}(\boldsymbol{Y})$

/* Produce a list $X$ of $L$ subcodes from $\mathfrak{C}_1$ with support size $w$ */
1  $X = \varnothing$;
2  while $|X| < L$ do
3      Call ISD to find $\mathfrak{B} \subseteq \mathfrak{C}_1$ with support size $w$;
4      $\boldsymbol{X} \leftarrow$ basis of $\mathfrak{B}$;
5      $X \leftarrow X \cup \{\mathsf{SF}(\boldsymbol{X})\}$;

/* Produce a list $Y$ of $L$ subcodes from $\mathfrak{C}_2$ with support size $w$ */
6  $Y = \varnothing$;
7  while $|Y| < L$ do
8      Call ISD to find $\mathfrak{B} \subseteq \mathfrak{C}_2$ with support size $w$;
9      $\boldsymbol{Y} \leftarrow$ basis of $\mathfrak{B}$;
10     $Y \leftarrow Y \cup \{\mathsf{SF}(\boldsymbol{Y})\}$;

/* Find collisions between the lists $X$ and $Y$ */
11 for $\{\boldsymbol{X}, \boldsymbol{Y}\} \in X \times Y$ do
12     if $\mathsf{Lex}^{(2)}(\boldsymbol{X}) = \mathsf{Lex}^{(2)}(\boldsymbol{Y})$ then
13         $P \leftarrow P \cup \{\boldsymbol{X}, \boldsymbol{Y}\}$;

14 return $P$;
```

We now proceed with the complexity analysis of the algorithm. We first argue that the complexity to find a specific 2-dimensional subcode with support size w is (essentially) the same as finding a specific codeword with weight w. One can indeed apply the same procedure of an ISD algorithm, with only minor tweaks so that the algorithm searches (and returns) a subcode. Namely, the algorithm in [7] can be seen as an adaptation of Lee & Brickell ISD, since it just consists in first applying the typical gaussian elimination and then checking whether couples of rows generate a subcode with support size w. The resulting time complexity is (essentially) the same as Lee & Brickell algorithm to find low weight codewords. For the rest of this work (and coherently with the codewords search version), we will denote by $C_{ISD}(q, n, k, w)$ the corresponding time complexity of finding a solution, in the regime in which a unique solution exists.

Algorithm 2: Fast permutation recovery, for the linear equivalence version of Beullens' algorithm.

```
Input: list $P$, containing $M$ pairs $\{\boldsymbol{X},\boldsymbol{Y}\} \in \mathbb{F}_q^{2\times n} \times \mathbb{F}_q^{2\times n}$ with support size
       $w$ and such that $\mathsf{Values}(\boldsymbol{X}) = \mathsf{Values}(\boldsymbol{Y})$
Output: permutation $\pi$, or report failure

1  $\boldsymbol{U} \leftarrow n \times n$ matrix made of all ones;
2  for $\{\boldsymbol{X},\boldsymbol{Y}\} \in P$ do
3      for $i \in \{1,\cdots,n\}$ do
4          $\boldsymbol{x}_i \leftarrow i$-th column of $\boldsymbol{X}$;
5          for $j \in \{1,\cdots,n\}$ do
6              $\boldsymbol{y}_j \leftarrow j$-th column of $\boldsymbol{Y}$;

               /* Filter $(i,j)$ */
7              if $(\boldsymbol{x}_i == \boldsymbol{0}) \neq (\boldsymbol{y}_j == \boldsymbol{0})$ then
8                  $u_{i,j} = 0$;

   /* Use $\boldsymbol{U}$ to reconstruct the permutation; if not possible, report failure */
9  if $\boldsymbol{U}$ is a permutation matrix then
10     $\pi \leftarrow$ permutation described by $\boldsymbol{U}$;
11     return $\pi$;
12 else
13     report failure;
```

To estimate the number of two-dimensional subcodes with support size w a random code contains, on average, we use $N_w^{(2)}$, as in Remark 3. Taking all of this into account, we have that the cost of each ISD call can be optimistically assessed as $\frac{C_{ISD}(q,n,k,w)}{N_w^{(2)}}$. Then, we can assess the cost of Algorithm 1 as follows.

Proposition 5. Let $\mathfrak{C}_1 \subseteq \mathbb{F}_q^n$ be a random linear code with dimension k, and let $\mathfrak{C}_2 = \tau(\mathfrak{C}_1)$ with $\tau \xleftarrow{\$} \mathsf{M}_n$. Let P be the list obtained by running Algorithm 1 with parameters L and w, with $w \leq n-k+2$. The algorithm runs in time

$$O\left(L\left(\log_2(L) + (q^2-q)(q-1)\right) + M' + M'' + \frac{L}{N_w^{(2)}} C_{ISD}(q,n,k,w)\right),$$

and produces a list P with $M = M' + M''$ elements, where $M' = L^2/N_w^{(2)}$ is the average number of good collisions and $M'' \leq \frac{t_w^{(2)}(L^2-M')}{N_w^{(2)}}$ is that of bad collisions.

Proof. See Appendix E. □

Remark 4. We expect $\frac{t_w^{(2)}(L^2-M')}{N_w^{(2)}}$ to be a loose upper bound on the value of M'', especially when q is not high. This is due to the fact that $t_w^{(2)}$ is a rather loose upper bound on the number of equivalent subcodes with support size w that one code possesses.

Note that, with arguments similar to those of Proposition 12, we can estimate the probability with which Algorithm 2 succeeds in retrieving the correct permutation. Yet, to avoid computations that may be too complicated, we omit these details and skip to the more interesting case in which a much more simpler, slightly optimistic of the Algorithm is derived.

5.3. **Heuristic analysis.** For comparison purposes, it is interesting to provide a crude lower bound on the cost of Beullens' algorithm. Indeed, since we do not have a precise estimate of the number M'' of bad collisions, we need to make sure that heuristic assumptions, made to compare our LEP resolution algorithm described in Section 6 with Beullens' method, are to the advantage of the latter. So, we conservatively neglect M' and assume that bad collisions never happen. Furthermore, we bound from below the number of good collisions we need to reconstruct the permutation. To this end, we consider that we must filter $n(n-1)$ pairs of indexes, and each pair of subcodes gives information about $2w(n-w)$ pairs of indices, so that we need at least $\left\lceil \frac{n(n-1)}{2w(n-w)} \right\rceil$ pairs of subcodes. Since we have that the number of good collisions is heuristically given by $M' = L^2/N_w^{(2)}$, then we can set

$$L = \sqrt{N_w^{(2)} \left\lceil \frac{n(n-1)}{2w(n-w)} \right\rceil}.$$

With this in mind, we can greatly simplify the analysis of the algorithm as follows.

Proposition 6. The time complexity of Beullens' algorithm, running with parameters L and w such that i) $L < N_w^{(2)}$, ii) $L = \sqrt{N_w^{(2)} \left\lceil \frac{n(n-1)}{2w(n-w)} \right\rceil}$ and iii) $\frac{t_w^{(2)} L^2 (1-1/N_w^{(2)})}{N_w^{(2)}} < 1$, is bounded from below by

$$\Omega\left(\frac{L}{N_w^{(2)}} C_{ISD}(q,n,k,w)\right).$$

6. **Improving Beullens' algorithm for LEP.** In this section, we analyze a method to improve Beullens' approach to solve LEP. Namely, we propose a new algorithm to choose the initial two-dimensional subcodes from which the list P is built, which is based on first finding small weight codewords and then combining them to obtain colliding subcodes.

6.1. **Finding subcodes more efficiently.** Our idea consists in constructing two dimensional subcodes by first finding codewords with small Hamming weight, say w', and then considering only the subcodes which are generated by pairs of such codewords and have support size w. Before analyzing our algorithm, we briefly sketch the main intuition behind it. Remember that Beullens' algorithm aims to find pairs of subcodes with support size w and to produce a collision in the first lexicographic basis. Note that no additional condition is required, apart from the one on the support size. Heuristically, we expect any such subcodes to behave like a length-w random code plus $n-w$ coordinates that are always null. Consider a pair of matrices such as those in Figure 1a: to have a matching in the computation of $\mathsf{Lex}^{(2)}$, the two matrices must lead to the same orange sub-matrix (which is expected to contain a number of columns rather close to w). If, instead, we consider subcodes generated by a pair of codewords with weight w' (and, still, with support size w), then the situation is depicted in Figure 1b. To have the same $\mathsf{Lex}^{(2)}$ value, the portions that must collide now contain $2w'-w$ columns.

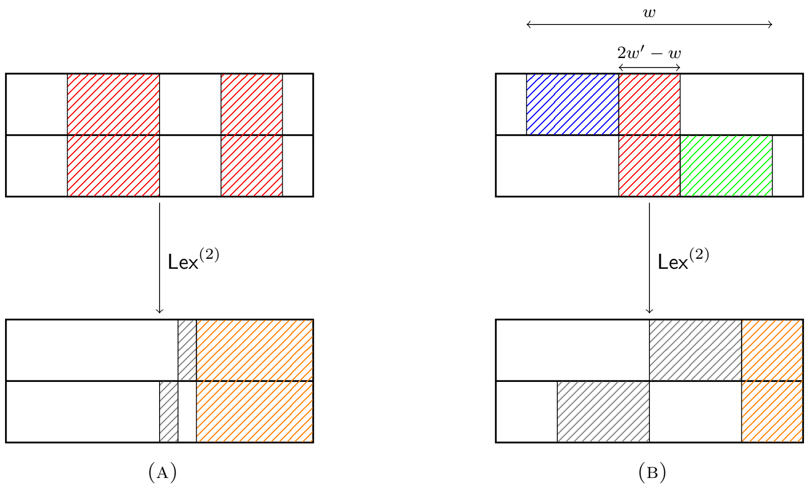


FIGURE 1. Example of computation of $\mathsf{Lex}^{(2)}$ starting from generator matrices of two-dimensional subcodes with support size w. White-filled rectangles denote portions of the matrix that contain only zeros, while grey-filled rectangles indicate all-zeros sub-rows.

If we choose w' so that $2w'$ is only slightly larger than w, then we increase the probability to find collisions (because the number of relevant columns gets lower). Also, given how ISD operates, finding a sufficient number of low weight codewords should be easier than directly finding some subcodes with small support. In the end, this reasoning can be summarized as follows: we consider subcodes with a structure that i) allows to easily find them, and ii) increases the probability to find collisions. Notice that the special structure of the considered subcodes clearly modifies the probability to find bad collisions; in the analysis of our algorithm, we conservatively take this phenomenon into account. Ultimately, we also validate our analysis by means of numerical simulations.

To formalize the above intuition, we consider the following procedure to search for subcodes with small support:

1. use ISD to find L' codewords with weight w';
2. form $2 \times n$ matrices using all $\binom{L'}{2}$ pairs of codewords;
3. keep only the matrices which generate a code with support size w.

Clearly, the values of L' and w' have a strong impact on the complexity of this approach, which we derive in the sequel of this section.

We start our analysis with the following technical Lemma, which describes the distribution probability of the support size of a subcode that is generated by two codewords with known weights.

Lemma 2. Let $\boldsymbol{a} \in \mathbb{F}_q^n$ with Hamming weight w_a. Let $\boldsymbol{b} \in \mathbb{F}_q^n$ be a random vector with Hamming weight w_b. Then, the code generated by $\boldsymbol{a}$, $\boldsymbol{b}$ (i.e., admitting the generator matrix whose rows are $\boldsymbol{a}$ and $\boldsymbol{b}$) has dimension 2 with probability

$$\begin{cases} 1 & \text{if } w_b \neq w_a, \\ 1 - \frac{1}{\binom{n}{w_a}(q-1)^{w_a-1}} & \text{if } w_b = w_a, \end{cases}$$

and support size $w \in [0; n]$ with probability

$$\zeta_{w_a,w_b}(w) = \begin{cases} 0 & \text{if } w < \max\{w_a; w_b\}, \\ 0 & \text{if } w > \min\{n; w_a + w_b\}, \\ \frac{\binom{w_a}{w_a+w_b-w}\binom{n-w_a}{w-w_a}}{\binom{n}{w_b}} & \text{otherwise.} \end{cases}$$

Proof. First, we consider the probability that the two chosen vectors do not generate a space with dimension 2. Note that this can happen only if $\boldsymbol{b} = v\boldsymbol{a}$ for some $v \in \mathbb{F}_q^*$. In such a case, we clearly have $w_a = w_b$. There are $q-1$ distinct values for v (yielding to distinct vectors $v\boldsymbol{a}$), while the number of vectors with weight w_a is given by $\binom{n}{w_a}(q-1)^{w_a}$. Hence, the probability that $\boldsymbol{b}$ is one of them (that is, the probability that $\boldsymbol{a}$ and $\boldsymbol{b}$ generate a space with dimension 1) is given by the following formula:

$$\frac{q-1}{\binom{n}{w_a}(q-1)^{w_a}} = \frac{1}{\binom{n}{w_a}(q-1)^{w_a-1}}.$$

We now derive the probability distribution for the support size of $\mathfrak{C}$, which we denote by w. Note that $w = w_a + w_b - |\mathsf{Supp}(\boldsymbol{a}) \cap \mathsf{Supp}(\boldsymbol{b})|$, from which we obtain $|\mathsf{Supp}(\boldsymbol{a}) \cap \mathsf{Supp}(\boldsymbol{b})| = w_a + w_b - w$. It is immediately seen that it must be $\max\{0; w_a + w_b - n\} \leq |\mathsf{Supp}(\boldsymbol{a}) \cap \mathsf{Supp}(\boldsymbol{b})| \leq \min\{w_a; w_b\}$, from which we find that the support size of $\mathfrak{C}$ is bounded as

$$\max\{w_a; w_b\} \leq w \leq \min\{w_a + w_b; n\}.$$

For all the admitted values, we have that we can have a support size w if and only if the set entries of $\boldsymbol{b}$ overlap with those of $\boldsymbol{a}$ in exactly $w_a + w_b - w$ positions. Since $\boldsymbol{b}$ is random, this happens with probability

$$\frac{\binom{w_a}{w_a+w_b-w}\binom{n-w_a}{w-w_a}}{\binom{n}{w_b}}. \qquad \square$$

Starting from a list containing L' codewords with weight w', the number of subcodes with support size w that we can form using pairs of such codewords can be estimated as

$$\binom{L'}{2}\zeta_{w',w'}(w) \approx \frac{L'^2\zeta_{w',w'}(w)}{2}.$$

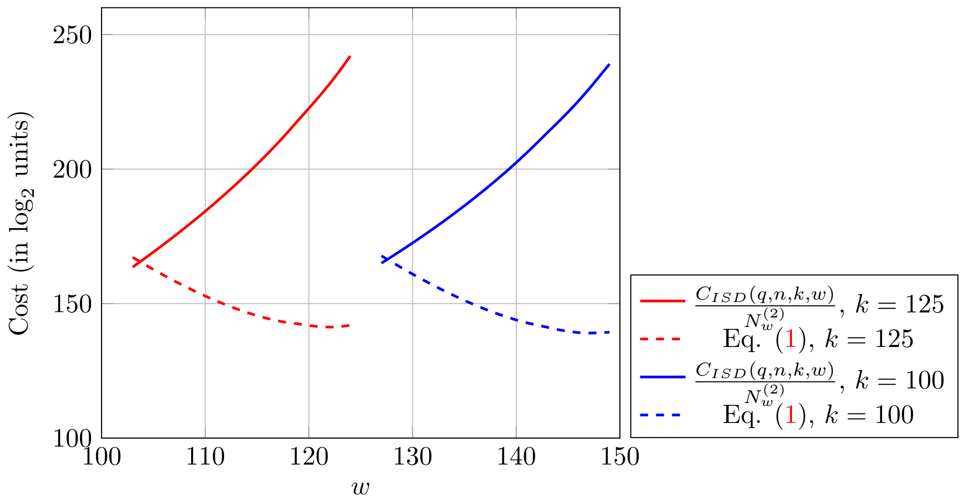


FIGURE 2. Cost of finding a subcode with support size w. All the considered codes have $q = 29$ and $n = 250$.

Setting $\frac{L'^2 \zeta_{w',w'}(w)}{2} \approx 1$, from which $L' \approx \sqrt{\frac{1}{\zeta_{w',w'}(w)}}$, we have that on average the considered approach produces one subcode. Hence, considering the number of ISD calls we need, in order to produce L' distinct codewords with weight w', we have that our proposed approach can find a subcode with support size w with a cost given by

$$\frac{\ln\left(1 - \frac{1}{N_{w'}\sqrt{\zeta_{w',w'}(w)}}\right)}{N_{w'} \ln\left(1 - \frac{1}{N_{w'}}\right)} C_{ISD}(q, n, k, w'). \tag{1}$$

Note that, by using ISD directly, we need to face a cost given by $\frac{C_{ISD}(q,n,k,w)}{N_w^{(2)}}$. To show that this approach is faster than using ISD to directly search for subcodes (as proposed in [7]), we report a comparison between the costs of these two approaches in Figure 2, where we have considered several values of w and, for our proposed approach, we have computed the value of w' which minimizes (1). In the next section we describe how this reasoning affects the complexity of Beullens' algorithm.

6.2. **Improved LEP algorithm.** We now analyze the application of our proposed approach to Beullens' algorithm. Technically, we propose to replace Algorithm 1 with Algorithm 3. Notice that the only difference with Algorithm 1 is in how the lists X and Y are constructed. According to the analysis we have performed in the previous section, we expect this approach to be faster when the values of w' and L' are properly chosen.

As we have already anticipated, as a little technical caveat, we have that the probability to have bad collisions gets modified, because we are considering subcodes having a particular structure. To this end, we consider the following Proposition.

Proposition 7. Consider Algorithm 3, applied on two codes $\mathfrak{C}_1$ and $\mathfrak{C}_2$, where $\mathfrak{C}_1$ is random and $\mathfrak{C}_2 = \tau(\mathfrak{C}_1)$. Then, on average, P contains $M' = \frac{\zeta_{w',w'}(w)}{2}\left(\frac{L'^2}{N_{w'}}\right)^2$ good

Algorithm 3: Our algorithm to find and match subcodes

```
Data: Number of codewords $L' \in \mathbb{N}$, weight $w' \in \mathbb{N}$, support size $w \in \mathbb{N}$,
      ISD routine
Input: linear codes $\mathfrak{C}_1, \mathfrak{C}_2 \subseteq \mathbb{F}_q^n$ with dimension $k$
Output: list $P$ containing pairs $(\boldsymbol{X}, \boldsymbol{Y}) \in \mathbb{F}_q^{2\times n} \times \mathbb{F}_q^{2\times n}$, such that
        $\mathsf{Values}(\mathsf{Lex}(\boldsymbol{X})) = \mathsf{Values}(\mathsf{Lex}(\boldsymbol{Y}))$

/* Produce a list $X'$ of $L'$ codewords from $\mathfrak{C}_1$ with weight $w'$ */
$X' = \varnothing$;
while $|X| < L$ do
    Call ISD to find $\boldsymbol{x} \in \mathfrak{C}_1$ with weight $w'$;
    $X' \leftarrow X' \cup \{\mathsf{Lex}(\boldsymbol{x})\}$;
/* Use pairs of codewords to produce subcodes with support size $w$ */
$X \leftarrow \varnothing$;
for $\boldsymbol{a} \in X'$ do
    for $\boldsymbol{b} \in X' \setminus \{\boldsymbol{a}\}$ do
        $\boldsymbol{X} \leftarrow$ matrix with rows $(\boldsymbol{a}, \boldsymbol{b})$;
        if Support of $\boldsymbol{X}$ has size $w$ then
            $X \leftarrow X \cup (\mathsf{Lex}(\boldsymbol{X}))$;

/* Produce a list $Y'$ of $L'$ codewords from $\mathfrak{C}_2$ with weight $w'$ */
$Y' = \varnothing$;
while $|X| < L$ do
    Call ISD to find $\boldsymbol{y} \in \mathfrak{C}_2$ with weight $w'$;
    $Y' \leftarrow Y' \cup \{\mathsf{Lex}(\boldsymbol{y})\}$;
/* Use pairs of codewords to produce subcodes with support size $w$ */
$Y \leftarrow \varnothing$;
for $\boldsymbol{a} \in Y'$ do
    for $\boldsymbol{b} \in Y' \setminus \{\boldsymbol{a}\}$ do
        $\boldsymbol{Y} \leftarrow$ matrix with rows $(\boldsymbol{a}, \boldsymbol{b})$;
        if Support of $\boldsymbol{Y}$ has size $w$ then
            $Y \leftarrow Y \cup (\mathsf{Lex}(\boldsymbol{Y}))$;

/* Find collisions between the lists $X$ and $Y$ */
for $\{\boldsymbol{X}, \boldsymbol{Y}\} \in X \times Y$ do
    if $\mathsf{Values}(\mathsf{Lex}\boldsymbol{X}) = \mathsf{ValuesLex}(\boldsymbol{Y})$ then
        $P \leftarrow P \cup \{\boldsymbol{X}, \boldsymbol{Y}\}$;

return $P$;
```

collisions and $M'' = p_{w'}(w)\frac{L'^4\zeta_{w',w'}(w)}{4}\left(\zeta_{w',w'}(w) - \frac{2}{N_{w'}^2}\right)$ bad collisions, where

$$p_{w'}(w) = \frac{\binom{n}{w-w'}\binom{n-(w-w')}{w-w'}\binom{n-2(w-w')}{2w'-w}(2w'-w)!(q-1)^{w-2w'+1}}{2\binom{n}{w'}\binom{n-w'}{w-w'}\binom{w'}{2w'-w}}.$$

Proof. We first derive the average number of good collisions. We consider that the number of codewords in X' which are mapped into codewords in Y' (through τ) can be estimated as $\widetilde{M} = \frac{L'^2}{N_{w'}}$. Indeed, for any codeword in X, we have only codeword (among all the $N_{w'}$ ones in $\mathfrak{C}_2$) which is its image through τ. Using any pair of such codewords to construct subcodes, we obtain good collisions. Considering that any of such subcodes will have the desired support size with probability $\zeta_{w',w'}(w)$, we can estimate the number of good collisions as

$$M' = \binom{\widetilde{M}}{2}\zeta_{w',w'}(w) \approx \frac{\widetilde{M}^2}{2}\zeta_{w',w'}(w) = \frac{\zeta_{w',w'}(w)}{2}\left(\frac{L'^2}{N_{w'}}\right)^2.$$

We now comment about the number of bad collisions. For any code, we dispose, on average, of $\binom{L'}{2}\zeta_{w',w'}(w) \approx \frac{L'^2}{2}\zeta_{w',w'}(w)$ subcodes with support size w. Hence, we have a total of $\left(\binom{L'}{2}\zeta_{w',w'}(w)\right)^2 \approx \left(\frac{L'^2}{2}\zeta_{w',w'}(w)\right)^2$ subcode pairs (one from X, one from Y): since M' of these pairs are good collisions, the number of pairs which may arise in bad collisions is $\left(\frac{L'^2}{2}\zeta_{w',w'}(w)\right)^2 - M' = \frac{L'^4\zeta_{w',w'}(w)}{4}\left(\zeta_{w',w'}(w) - \frac{2}{N_{w'}^2}\right)$. If each of these pairs is a bad collision with probability $p_{w'}(w)$, then we can estimate the number of bad collisions as

$$p_{w'}(w)\frac{L'^4\zeta_{w',w'}(w)}{4}\left(\zeta_{w',w'}(w) - \frac{2}{N_{w'}^2}\right).$$

To conclude the proof, we need to estimate $p_{w'}(w)$. Any subcode in X is generated by a $2 \times n$ matrix in which i) the rows have weight w', and ii) overlap in $x = 2w' - w$ positions (since the support has size w). The number of matrices with these properties is obtained as

$$U_{w'}(w) = \binom{n}{w'}\binom{n-w'}{w'-x}\binom{w'}{x}(q-1)^{2w'} = \binom{n}{w'}\binom{n-w'}{w-w'}\binom{w'}{2w'-w}(q-1)^{2w'}.$$

Indeed, the term $\binom{n}{w'}\binom{n-w'}{w'-x}\binom{w'}{x}$ counts all the possible supports for the rows of the generator matrix, while the term $(q-1)^{2w'}$ is due to the fact that, for each row, there are $(q-1)^{w'}$ choices for the set coefficients. We divide $U_{w'}(w)$ by $2(q-1)(q-1)$ to avoid multiple counting of the same matrix (since $(q-1)(q-1)$ is the number of matrices we can obtain by scaling each row, and the factor 2 is due to row swapping). Finally, we multiply $\frac{U_{w'}(w)}{2(q-1)^2}$ by $\frac{\left[{k \atop 2}\right]_q}{\left[{n \atop 2}\right]_q}$ to consider the probability that a matrix generates indeed a subcode of $\mathfrak{C}_1$. Now, we need to consider the number of subcodes of $\mathfrak{C}_1$ we can obtain by applying a monomial transformation to one of the generator matrices in X. This quantity can be set as

$$\begin{aligned}\tilde{t}_w^{(2)} &= \binom{n}{w'-x}\binom{n-w'+x}{w'-x}(q-1)^{2(w'-x)}\binom{n-2(w'-x)}{x}x!(q-1)^{x-1}\frac{\left[{k \atop 2}\right]_q}{\left[{n \atop 2}\right]_q}\\ &= \binom{n}{w-w'}\binom{n-(w-w')}{w-w'}\binom{n-2(w-w')}{2w'-w}(2w'-w)!(q-1)^{w-1}\frac{\left[{k \atop 2}\right]_q}{\left[{n \atop 2}\right]_q}\end{aligned}$$

Indeed, for $\boldsymbol{V} \in X$, we consider all possible matrices $\boldsymbol{V}'$ which can be obtained as $\boldsymbol{V}' = \sigma(\boldsymbol{V})$, where $\sigma \in \mathsf{M}_n$. In each $\boldsymbol{V}'$ we have $w'-x$ columns in which the entry in the first row is set and the one in the second row is null; also, we must have $w'-x$ other columns in which the entry in the first row is null, and the one in the second row is set. The number of such columns is counted as $\binom{n}{w'-x}\binom{n-w'+x}{w'-x}(q-1)^{2*(w'-x)}$. We then consider the number of monomial transformations of the columns containing two set entries, which is equal to x: this number cannot be larger than

$\binom{n-2(w'-x)}{x}x!(q-1)^x$. Indeed, it may happen that distinct transformations produce the same matrix, but we neglect such a possibility to simplify the analysis. Finally, we multiply again by the probability that the subcode generated by such a matrix is contained in $\mathfrak{C}_1$, and divide by $(q-1)$, to avoid multiple counting of matrices that generate the same subcode. Given the above reasoning, we can set

$$p_{w'}(w) = \frac{\tilde{t}_w^{(2)}}{\frac{U_{w'}(w)}{(q-1)}\frac{\left[{k \atop 2}\right]_q}{\left[{n \atop 2}\right]_q}} = \frac{\binom{n}{w-w'}\binom{n-(w-w')}{w-w'}\binom{n-2(w-w')}{2w'-w}(2w'-w)!(q-1)^{w-1}}{2\binom{n}{w'}\binom{n-w'}{w-w'}\binom{w'}{2w'-w}(q-1)^{2w'-2}}$$

$$= \frac{\binom{n}{w-w'}\binom{n-(w-w')}{w-w'}\binom{n-2(w-w')}{2w'-w}(2w'-w)!(q-1)^{w-2w'+1}}{2\binom{n}{w'}\binom{n-w'}{w-w'}\binom{w'}{2w'-w}}.$$

□

We are now ready to evaluate the complexity of our new LEP algorithm.

Proposition 8. The time complexity of our LEP algorithm, using lists of L' codewords of weight w' such that i) $L' = \sqrt[4]{\frac{4N_{w'}^2}{\zeta_{w',w'}(w)}n\ln(n)}$ and ii) $M'' < 1$, is in

$$O\left(\frac{\ln\left(1 - L'/N_{w'}\right)}{N_{w'}\ln\left(1 - 1/N_{w'}\right)}C_{ISD}(q,n,k,w')\right).$$

Conditions i) and ii) sets an estimate on the number of good collisions we need; notice that condition ii) is obtained by setting $M' = 2n\ln(n)$. Finally, condition iii) guarantees that, with high probability, bad collisions do not happen. For large inputs, we have that $L'/N_{w'} = o(1)$, and therefore, the first order approximation of the cost of our LEP algorithm simplifies as

$$O\left(\frac{C_{ISD}(q,n,k,w')}{\sqrt{N_{w'}}}\sqrt[4]{\frac{n\log(n)}{\zeta_{w',w'}(w)}}\right).$$

6.3. **Performance of our new LEP algorithm.**

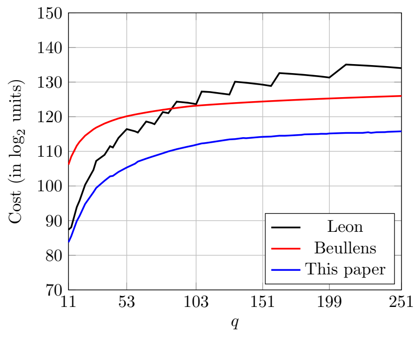
In this section we comment about the effectiveness of our new approach. First, we present the results of numerical simulations, to validate the statement of Proposition 7.

(n,k,q)	(L',w',w)	Num subcodes		M'		M''	
		th.	emp.	th.	emp.	th.	emp.
$(40,20,7)$	$(10,12,19)$	7.55	7.30	1.30	0.43	3.20	2.57
	$(100,13,20)$	626.86	614.00	58.76	59.40	18558.74	15016.60
$(30,10,13)$	$(11,15,21)$	8.88	8.37	10.77	7.98	0.036	0.02
	$(40,16,24)$	207.30	208.32	24.48	24.78	25.42	26.22
$(30,10,19)$	$(25,16,24)$	79.73	82.06	76.20	78.56	0.22	0.44
	$(50,17,24)$	341.36	343.15	5.82	4.70	1.11	1.30

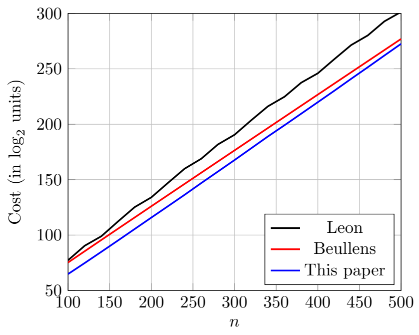
TABLE 2. Comparison between numerical results and theoretical estimates on the composition of the list P obtained with Algorithm 3. For each triplet (n,k,q), the empirical results have been averaged over 100 random codes.

To this end, for each parameter set, we have considered 100 different pairs of codes $\mathfrak{C}_1$ and $\mathfrak{C}_2$. Then, for each pair, we have simulated Algorithm 3; in Table 2 we compare the empirical values of M' and M'' (averaged over all the trials) with the theoretical ones, estimated though Proposition 7. As we can see from the table, the theoretical estimates match the empirical ones; this provides a validation of the heuristic we have employed to assess the performances of Algorithm 3.

In Figure 3 we compare the complexity arising from Proposition 8 with those of Leon's and Beullens' algorithms. For our algorithm we rely on the average complexity estimate resulting from Proposition 8, while for the other algorithms we have considered the lower bounds resulting from Propositions 3 and 6. Remember that for Leon's algorithm we are underestimating the weight value which is necessary to run the attack, so that, in practice, the actual complexity of the algorithm may be much larger. For Beullens' method, the lower bound comes from the fact that Proposition 6 is derived assuming bad collisions never happen (i.e. M'' is set to 0).



(A) $n = 200$, $k = 100$, several q



(B) $q = 251$, $k = \frac{1}{2}n$, several n

FIGURE 3. Comparison between several methods to solve LEP.

This comparison shows that our proposed algorithm performs better than the state-of-the-art solvers. Note that, in Figure 3, the cost of Beullens' method gets closer to ours for fixed q and $n \to \infty$. In this regime, the simplifications made in the cost analysis of Beullens' algorithm might underestimate the real cost by a significant margin. Indeed, the term M'' that we removed is dominated by $t_w^{(2)} \sim n!/(n-w)!$, which is exponential in n. To give more insight on how our algorithm operates, in Table 3 we have reported the optimal setting for the attack, for some of the instances we have considered in Figure 3.

q	L'	w'	w	Cost
11	13	56	97	$2^{83.76}$
53	183	62	111	$2^{105.34}$
103	$8.83 \cdot 10^7$	80	124	$2^{111.90}$
151	$5.57 \cdot 10^9$	83	128	$2^{114.19}$
199	$4.48 \cdot 10^{12}$	86	126	$2^{115.17}$
251	$3.39 \cdot 10^{14}$	88	127	$2^{115.78}$

TABLE 3. Optimal setting and resulting complexity for our algorithm to solve LEP. All the considered codes have $n = 200$, $k = 100$.

7. Conclusions.

The code equivalence problem is seeing an increasing presence in cryptographic literature. Since protocols based on code equivalence have the potential to be very efficient and lead to good solutions for code-based signature schemes (as well as other functionalities), it is important to properly assess the hardness of the problem in practical instances. In this paper, we provided a detailed analysis of the various approaches for solvers, for both permutation and linear code equivalence.

We have briefly explained why solvers that exploit particular properties, such as that of Bardet et al. [5] and Sendrier's support splitting algorithm [17], do not perform well in most instances of LEP, including the ones used in cryptography. In fact, both solvers are only truly efficient for the case of codes with trivial hulls, and it is thus easy to find hard instances. With regards to the latter, for example, it is worth mentioning that, in the linear case, SSA needs to be applied to the *closure* of the considered codes; however, for $q \geq 5$, the closure of a code is always weakly-self dual, and thus has a hull of maximum dimension k, leading to exactly the hardest instances for SSA to solve.

As a consequence of the above considerations, we gave an extensive treatment only to techniques that exploit the Hamming weight as an invariant, and utilize ISD as a subroutine for searching codewords. We have summarized and given a precise cost estimate of the two main algorithms of this type, Leon's [12] and Beullens' [7], that can be originally applied to the case of permutation equivalence. We have then shown how both can be adapted to the linear equivalence case, and produced a concrete technical analysis, which was lacking in the original works. Furthermore, we have presented an improved routine, that can considerably reduce the cost of Beullens' algorithm. We have given accurate complexity formulae, in all cases.

References.

[1] M. R. Albrecht et al. "Classic McEliece: conservative code-based cryptography". In: (). URL: https://classic.mceliece.org/.

[2] N. Aragon et al. "BIKE: Bit Flipping Key Encapsulation". In: *NIST Post-Quantum Standardization, 3rd Round* (2021). URL: https://bikesuite.org/.

[3] L. Babai. "Graph Isomorphism in Quasipolynomial Time". In: *CoRR* abs/1512.03547 (2015). arXiv: 1512.03547. URL: http://arxiv.org/abs/1512.03547.

[4] M. Baldi et al. "A Finite Regime Analysis of Information Set Decoding Algorithms". In: *Algorithms* 12.10 (2019). ISSN: 1999-4893. URL: https://www.mdpi.com/1999-4893/12/10/209.

[5] M. Bardet, A. Otmani, and M. Saeed-Taha. "Permutation Code Equivalence is Not Harder Than Graph Isomorphism When Hulls Are Trivial". In: *IEEE ISIT 2019.* July 2019, pp. 2464–2468.

[6] A. Barenghi et al. "LESS-FM: Fine-tuning Signatures from the Code Equivalence Problem". In: *International Conference on Post-Quantum Cryptography.* Springer. 2021, pp. 23–43.

[7] W. Beullens. "Not Enough LESS: An Improved Algorithm for Solving Code Equivalence Problems over $\mathbb{F}_q$". In: *International Conference on Selected Areas in Cryptography.* Springer. 2020, pp. 387–403.

[8] J.-F. Biasse et al. "LESS is More: Code-Based Signatures Without Syndromes". In: *AFRICACRYPT.* Ed. by A. Nitaj and A. Youssef. Springer, 2020, pp. 45–65.

[9] A. Couvreur, T. Debris-Alazard, and P. Gaborit. "On the hardness of code equivalence problems in rank metric". In: *arXiv preprint arXiv:2011.04611* (2020).

[10] T. Feulner. "The automorphism groups of linear codes and canonical representatives of their semilinear isometry classes." In: *Adv. Math. Commun.* 3.4 (2009), pp. 363–383.

[11] A. Fiat and A. Shamir. "How to prove yourself: Practical solutions to identification and signature problems". In: *CRYPTO.* Springer. 1986, pp. 186–194.

[12] J. Leon. “Computing automorphism groups of error-correcting codes”. In: *IEEE Transactions on Information Theory* 28.3 (May 1982), pp. 496–511.
[13] C. A. Melchor et al. “HQC: Hamming Quasi-Cyclic”. In: *NIST Post-Quantum Standardization, 3rd Round* (2021). URL: http://pqc-hqc.org/.
[14] C. Peters. “Information-set decoding for linear codes over $\mathbb{F}_q$”. In: *International Workshop on Post-Quantum Cryptography*. Springer. 2010, pp. 81–94.
[15] E. Petrank and R. M. Roth. “Is code equivalence easy to decide?” In: *IEEE Transactions on Information Theory* 43.5 (Sept. 1997), pp. 1602–1604.
[16] M. A. Saeed. In: *PhD thesis* (2017).
[17] N. Sendrier. “The Support Splitting Algorithm”. In: *Information Theory, IEEE Transactions on* (Aug. 2000), pp. 1193–1203.
[18] N. Sendrier and P. Symbolique. “On the Dimension of the Hull”. In: *SIAM Journal on Discrete Mathematics* 10 (Nov. 1995). DOI: 10.1137/S0895480195294027.

Appendix A. **Leon's algorithm.** In this Appendix we provide further details about Leon's algorithm. We start by proving Proposition 3.

Proposition 3 ([7])**.** Let $\mathfrak{C}_1 \subseteq \mathbb{F}_q^n$ be a random code with dimension k, $\pi \xleftarrow{\$} \mathsf{S}_n$ and $\mathfrak{C}_2 = \pi(\mathfrak{C}_1)$. The cost of Leon's algorithm, running with parameter $w \in \mathbb{N}$, $w \leq n$, can be estimated[4] as

$$O\big(\ln(N_w)C_{ISD}(q,n,k,w)\big).$$

Proof. (*Heuristic*) We first consider the cost of the codewords enumeration in Step 1. For both codes $\mathfrak{C}_1$, $\mathfrak{C}_2$, we need to find all of the N_w codewords with weight w. To this end, we model an ISD algorithm as an oracle that, in each call, returns a random weight-w codeword. We first focus on $\mathfrak{C}_1$: the first ISD call will take time complexity $C_{ISD}(q,n,k,w)/N_w$. In the second call we desire to find a distinct codeword, so that the time complexity of this second call is $C_{ISD}(q,n,k,w)/(N_w-1)$. If we iterate this reasoning, we get that the codewords enumeration for $\mathfrak{C}_1$ takes time

$$O\left(C_{ISD}(q,n,k,w)\cdot\sum_{i=1}^{N_w}\frac{1}{i}\right).$$

When N_w is large, we consider that $\sum_{i=1}^{N_w}\frac{1}{i} \approx \ln(N_w)$. The codewords enumeration is repeated for $\mathfrak{C}_2$, with analogous cost: this yields a constant factor 2 in the complexity.

Under the assumption that w is properly chosen, $\mathsf{Mor}_{\mathsf{S}_n}\big(A_w(\mathfrak{C}_1), A_w(\mathfrak{C}_2)\big)$ contains a very small number of elements (ideally, only one). So, we can neglect the cost of steps 2 and 3, and consider only the cost of codewords enumeration. □

We now give some insight on how the choice of w is expected to affect the algorithm. In the following Proposition we derive a heuristic lower bound on the size of $\mathsf{Mor}_{\mathsf{S}_n}\big(A_w(\mathfrak{C}_1), A_w(\mathfrak{C}_2)\big)$, which is obtained under the (realistic) assumption that weight-w codewords of random codes have random supports.

Proposition 9. Let $\mathfrak{C}_1 \subseteq \mathbb{F}_q^n$ be a random code with dimension k, $\pi \xleftarrow{\$} \mathsf{S}_n$ and $\mathfrak{C}_2 = \pi(\mathfrak{C}_1)$. Then, the set $\mathsf{Mor}_{\mathsf{S}_n}\big(A_w(\mathfrak{C}_1), A_w(\mathfrak{C}_2)\big)$ contains at least $u!$ elements, where $u = \max\left\{1\,,\ \left\lfloor n\left(1-\frac{w}{n}\right)^{N_w}\right\rfloor\right\}$.

[4]Here we use the same estimate derived in [7, Section 2.2], which corresponds to a lower bound for the actual complexity since the cost of steps 2 and 3 is neglected. In other words, the proposition takes into account only the cost of the codewords enumeration phase.

Proof. (Heuristic) Since $\mathfrak{C}_1$ is random, we use N_w to estimate the number of codewords with weight w. For $i = 1, 2$, let $B_i = \{j \in [1; n] \mid \forall \boldsymbol{x} \in A_w(\mathfrak{C}_i) : c_j = 0\}$ and $\bar{B}_i = \{1, \cdots, n\} \setminus B_i$. Note that, for any index $j \in B_i$, we have that all the codewords in $A_w(\mathfrak{C}_i)$ have a null entry in position j, while for any $j \in \bar{B}_i$ there is at least a codeword in $A_w(\mathfrak{C}_i)$ whose j-th entry is non null. Let us now consider a permutation $\sigma \neq \pi$ such that $\sigma(j) = \pi(j)$, for all $j \in \bar{B}_1$: this implies that $\sigma(j) \in B_2$ for all $j \in \bar{B}_1$. Then, clearly $\sigma \in \mathsf{Mor}_{\mathsf{S}_n}\big(A_w(\mathfrak{C}_1), A_w(\mathfrak{C}_2)\big)$. Notice that the number of valid permutations σ is equal to the number of bijections from B_1 to B_2, which is $|B_1|! = |B_2|!$. Note that this is only a lower bound, since there may exist permutations that map $A_w(\mathfrak{C}_1)$ into $A_w(\mathfrak{C}_2)$ even if $\sigma(j) \neq \pi(j)$ for some $j \in \bar{B}_1$.
To complete the proof, we need to estimate the size of B_1. To this end, we rely on the following estimate $|B_1| = n\left(1 - \frac{w}{n}\right)^{N_w}$. Indeed, since $\mathfrak{C}_1$ is random, we see any of its codewords as a random vector. Consequently, the probability that an index j is in the support of a codeword with weight w is w/n. Since $\mathfrak{C}_1$ has N_w codewords with weight w, the probability that an index never appear in the supports of all N_w weight-w codewords is $\left(1 - \frac{w}{n}\right)^{N_w}$. Multiplying the above probability by n, we obtain an estimate for the average size of B_1. □

The result in the above Proposition de facto sets a theoretical lower bound on the value of w which must be used when running Leon's algorithm.

Appendix B. **Beullens' algorithm for PEP.** In this appendix we provide details about Beullens' algorithm to solve PEP. Given a pair $(\boldsymbol{x}, \boldsymbol{y}) \in \mathfrak{C}_1 \times \mathfrak{C}_2$ such that $\mathsf{Values}(\boldsymbol{x}) = \mathsf{Values}(\boldsymbol{y})$, we will say that $(\boldsymbol{x}, \boldsymbol{y})$ is a *good collision* if $\pi(\boldsymbol{x}) = \boldsymbol{y}$, and a *bad collision* if $\pi(\boldsymbol{x}) \neq \boldsymbol{y}$. Once colliding pairs of codewords have been obtained, one can employ a probabilistic procedure to retrieve the permutation with some probability. In particular, in [7], the author has considered an approach which works only in case bad collisions do not happen.

B.1. **Finding matching codewords.** We start by analyzing the routine which produces pairs of colliding codewords. We briefly recall the approach of [7] and provide a heuristic analysis on the number of bad and good collisions which one expects to have, on average. To begin, we observe that for any pair of vectors such that $\boldsymbol{y} = \pi(\boldsymbol{x})$, it must also be that $v\boldsymbol{y} = \pi(v\boldsymbol{x})$ for all $v \in \mathbb{F}_q^*$. This means that, given a pair $(\boldsymbol{x}, \boldsymbol{y})$ of colliding codewords, we are able to produce additional $q-1$ pairs of colliding codewords which, however, do not bring any new information about π. Considering all of these pairs in the permutation reconstruction algorithm is useless. Hence, we can get rid of such additional collisions with the following approach (proposed by Beullens in [7]). Let Lex denote the function that on input a vector $\boldsymbol{a}$ returns $b\boldsymbol{a}$, with $b \in \mathbb{F}_q^*$ such that $\mathsf{Values}(b\boldsymbol{a})$ comes first, in lexicographical order, among the multiset entries of all scalar multiples of $\boldsymbol{a}$. To understand how this function operates, we have reported an example in Figure 4. Embedding the function Lex into the codewords finding algorithm, one can get rid of all unnecessary codewords.

The full subroutine for finding colliding codewords is shown in Algorithm 4. We observe that including the computation of Lex into the codewords search guarantees that we do not put scalar multiples into the lists X and Y and, consequently, into P. The number of codewords we draw from each code is indicated as L, while w is the Hamming weight of the found codewords.

$$\begin{aligned}\mathsf{Values}(1\boldsymbol{a}) &= \{1,2,2,3,3,3,4\}\\ \mathsf{Values}(2\boldsymbol{a}) &= \{1,1,1,2,3,4,4\}\\ \mathsf{Values}(3\boldsymbol{a}) &= \{1,1,2,3,4,4,4\}\\ \mathsf{Values}(4\boldsymbol{a}) &= \{1,2,2,2,3,3,4\}\end{aligned}$$

(A)

1°	-	$\mathsf{Values}(2\boldsymbol{a}) = \{1,1,1,2,3,4,4\}$
2°	-	$\mathsf{Values}(3\boldsymbol{a}) = \{1,1,2,3,4,4,4\}$
3°	-	$\mathsf{Values}(4\boldsymbol{a}) = \{1,2,2,2,3,3,4\}$
4°	-	$\mathsf{Values}(1\boldsymbol{a}) = \{1,2,2,3,3,3,4\}$

(B)

FIGURE 4. Example of lexicograph ordering, for the finite field with $q = 5$ elements and a vector $\boldsymbol{a} = (0,3,2,0,0,3,3,2,4,1)$, for which $\mathsf{Lex}(\boldsymbol{a}) = 2\boldsymbol{a}$. Figure (A) shows the multisets of entries for all scalar multiples of $\boldsymbol{a}$, while figure (B) reports the lexicographic order of such multisets.

Algorithm 4: Algorithm to find and match codewords

```
Data: Number of codewords $L \in \mathbb{N}$, weight $w \in \mathbb{N}$, ISD routine
Input: linear codes $\mathfrak{C}_1, \mathfrak{C}_2 \subseteq \mathbb{F}_q^n$ with dimension $k$
Output: list $P$ containing pairs $(\boldsymbol{x},\boldsymbol{y}) \in \mathfrak{C}_1 \times \mathfrak{C}_2$, such that
        $\mathsf{Values}(\boldsymbol{x}) = \mathsf{Values}(\boldsymbol{y})$

/* Produce a list $X$ of $L$ codewords from $\mathfrak{C}_1$ with weight $w$ */
1  $X = \varnothing$;
2  while $|X| < L$ do
3      Call ISD to find $\boldsymbol{x} \in \mathfrak{C}_1$ with weight $w$;
4      $X \leftarrow X \cup \{\mathsf{Lex}(\boldsymbol{x})\}$;

/* Produce a list $Y$ of $L$ codewords from $\mathfrak{C}_2$ with weight $w$ */
5  $Y = \varnothing$;
6  while $|Y| < L$ do
7      Call ISD to find $\boldsymbol{y} \in \mathfrak{C}_2$ with weight $w$;
8      $Y \leftarrow Y \cup \{\mathsf{Lex}(\boldsymbol{y})\}$;

/* Find collisions between the lists $X$ and $Y$ */
9  for $\{\boldsymbol{x},\boldsymbol{y}\} \in X \times Y$ do
10     if $\mathsf{Values}(\boldsymbol{x}) = \mathsf{Values}(\boldsymbol{y})$ then
11         $P \leftarrow P \cup \{\boldsymbol{x},\boldsymbol{y}\}$;

12 return $P$;
```

In the next proposition we compute the average size of P, as well as the number of good and bad collisions.

Proposition 10. Let $\mathfrak{C}_1 \subseteq \mathbb{F}_q^n$ be a random linear code with dimension k, and let $\mathfrak{C}_2 = \pi(\mathfrak{C}_1)$ with $\pi \xleftarrow{\$} \mathsf{S}_n$. Then, on average P contains $M = M' + M''$

elements, when $M' = L^2/N_w$ is the average number of good collisions and $M'' = (1-1/N_w)(q-1)L^2\binom{w+q-3}{w-1}^{-1}$ is that of bad collisions.

Proof. We consider that $\mathfrak{C}_1$ contains N_w codewords with weight w and, according to Assumption 1, assume that the automorphism group of the code is trivial. We first determine the number of good collisions. For each $\boldsymbol{x} \in \mathfrak{C}_1$, we have only one codeword $\boldsymbol{y} \in \mathfrak{C}_2$ such that $\pi(\boldsymbol{x}) = \boldsymbol{y}$. Since ISD returns random codewords of weight w, we have that on average the number of good collisions is given by

$$M' = \sum_{i=1}^{L} i \cdot \frac{\binom{L}{i}\binom{N_w-L}{L-i}}{\binom{N_w}{L}} = \frac{L^2}{N_w}.$$

We now count the number of bad collisions; let us first make some preliminary considerations. First, for any vector $\boldsymbol{a}$, we have that $\mathsf{Lex}(\boldsymbol{a})$ contains at least a 1. Hence, for each $\boldsymbol{x} \in X$, we may assume that $\mathsf{Values}(\boldsymbol{x})$ is a random multiset with one entry equal to 1, and the other $w-1$ ones picked at random over $\mathbb{F}_q^*$. The same goes for each $\boldsymbol{y} \in Y$. To have $\{\boldsymbol{x}, \boldsymbol{y}\} \in P$, it must be $\mathsf{Values}(\boldsymbol{x}) = \mathsf{Values}(\boldsymbol{y})$: since there are $\binom{q+w-3}{w-1}$ ways to choose $w-1$ elements from $\mathbb{F}_q^*$ with repetitions, assuming that such elements are drawn at random, we have that a collision between $\mathsf{Values}(\boldsymbol{x})$ and $\mathsf{Values}(\boldsymbol{y})$ is expected to happen with probability $\binom{q+w-3}{w-1}^{-1}$. Hence, the number of bad collisions can be estimated as

$$M'' = (L^2 - M')(q-1)\binom{w+q-3}{w-1}^{-1}.$$

Indeed, we have $L^2 - M'$ possible pairs that do not give rise to a good collision, and a fraction $\binom{w+q-3}{w-1}^{-1}$ of these is expected to yield to a bad collision. Then, the list size of P is given by $M' + M''$. □

A confirmation of the heuristics we have used for the Proposition is shown in Section B.3, where we compare the performances of the algorithm with those of numerical simulations. The complexity of executing Algorithm 4 is computed in the next proposition.

Proposition 11. Let $\mathfrak{C}_1 \subseteq \mathbb{F}_q^n$ be a random code with dimension k, and $\mathfrak{C}_2 = \pi(\mathfrak{C}_1)$, with π being a randomly picked permutation. Then, the complexity of running algorithm 4 with parameter L and w is

$$O\left(L\left(\log_2(L) + (q-1)w\log_2^2(q)\right) + M' + M'' + \frac{C_{ISD}(q,n,k,w)}{N_w} \cdot \frac{\ln\left(1-\frac{L}{N_w}\right)}{\ln\left(1-\frac{1}{N_w}\right)}\right).$$

Proof. (Heuristic) We start by estimating the number of ISD calls to find L distinct codewords. We consider that each ISD call costs $C_{ISD}(n,k,q,w)/N_w$. Now: the average number of distinct codewords we find, after u calls, is $N_w\left(1-\left(1-\frac{1}{N_w}\right)^u\right)$. Since we want this quantity to be equal to L, it must be $u = \frac{\ln\left(1-\frac{L}{N_w}\right)}{\ln\left(1-\frac{1}{N_w}\right)}$. Then, the average cost of calling ISD is $O\left(\frac{C_{ISD}(q,n,k,w)}{N_w} \cdot \frac{\ln\left(1-\frac{L}{N_w}\right)}{\ln\left(1-\frac{1}{N_w}\right)}\right)$. For each found codeword we compute the value of Lex, which comes with a cost of $(q-1)w\log_2^2(q)$ (since we must compute the $(q-1)$ scalar multiple of each found codeword, having weight w). Then, we have to produce the merged list, which can be done efficiently if one firsts hashes the entries of lists X and Y and then uses a binary search algorithm. This comes with a cost that is estimated as $L\log_2(L)$. Finally, we consider that the list P contains $M' + M''$ elements, and consider such a value as the estimate for the complexity to build the list. □

Remark 5. If $L \ll N_w$ (as we expect), then $\ln(1-\frac{L}{N_w}) \approx -\frac{L}{N_w}$ and $\ln(\frac{1}{N_w}) \approx -\frac{1}{N_w}$. Then, the cost of ISD becomes $O\left(\frac{LC_{ISD}(q,n,k,w)}{N_w}\right)$: this means that we call ISD for L times, and that every call costs $\frac{C_{ISD}(q,n,k,w)}{N_w}$. Embedding this simplification into the expression of 11, we obtain a time complexity of

$$O\left(L\log_2(L) + L(q-1)w\log_2^2(q) + M' + M'' + \frac{LC_{ISD}(q,n,k,w)}{N_w}\right).$$

Furthermore, considering that the cost of ISD is expected to be prevalent, with respect to the other terms, we can simply assume that Algorithm 4 costs

$$O\left(\frac{LC_{ISD}(q,n,k,w)}{N_w}\right)$$

B.2. **Probabilistic permutation recovery.** We now move on to assessing the performance of the permutation reconstruction phase described in [7]. The algorithm exploits the following crucial observation: if we know that $\pi(\boldsymbol{x}) = \boldsymbol{y}$ for some permutation π, and we have $x_i \neq y_j$, then we know that π does not map i to j. Considering all pairs of indexes (i,j) for which $x_i \neq y_j$, we gather a significant amount of information about π or, to put it differently, we filter out a wide number of candidate permutations. Exploiting all pairs in P, and putting all the information together, it may become possible to recover the secret π with a procedure as simple as the one in Algorithm 5.

Algorithm 5: Probabilistic permutation recovery, for the permutation equivalence version of Beullens' algorithm

```
Input: list $P$, containing pairs $\{\boldsymbol{x},\boldsymbol{y}\} \in \mathbb{F}_q^n \times \mathbb{F}_q^n$
Output: permutation $\pi \in \mathsf{S}_n$, or report failure

1  $\boldsymbol{U} \leftarrow n \times n$ matrix made of all ones;
2  for $\{\boldsymbol{x},\boldsymbol{y}\} \in P$ do
3      for $i \in \{1,\cdots,n\}$ do
4          for $j \in \{1,\cdots,n\}$ do
5              if $x_i \neq y_j$ then
6                  $u_{i,j} = 0$
   /* Use $\boldsymbol{U}$ to reconstruct the permutation; if not possible, report failure */
7  if $\boldsymbol{U}$ is a permutation matrix then
8      return $\pi$;
9  else
10     report failure
```

We note that, however, such an efficient permutation recovery is characterized by a certain failure probability. Indeed, when P is not populated with a sufficient number of elements, it may happen that the algorithm is not able to return a valid matrix. To estimate the probability that Algorithm 5 is successful, and to additionally derive the minimum number of elements that P should contain, we rely on the following proposition.

Proposition 12. Let $\mathfrak{C}_1 \subset \mathbb{F}_q^n$ be a random linear code with dimension k and $\mathfrak{C}_2 = \pi(\mathfrak{C}_1)$, with $\pi \xleftarrow{\$} \mathsf{S}_n$. Let P be the list produced by Algorithm 4 with input parameters L and w. Let L and w such that $M''\left(1 - \frac{1}{N_w}\frac{L^2}{\binom{w+q-3}{w-1}}\right) \approx 0$. Then, Algorithm 5 runs in time $O\left(n^2\frac{L^2}{N_w}\right)$, and retrieves the correct permutation with probability $(1-\rho^{\frac{L^2}{N_w}})^{n(n-1)}$, where $\rho = \left(1-\frac{w}{n}\right)^2 + \frac{1}{q-1}\left(\frac{w}{n}\right)^2$.

Proof. We assume that L and w are such that P does not contain bad collisions. Hence, we have $|P| = M' = \frac{L^2}{N_w}$. To check each pair in P, the algorithm uses $O(n^2)$ operations (since it goes through all pairs of indexes $(i,j) \in \{1,\dots,n\}^2$). We now proceed to estimate the success probability. We consider a pair of codewords $\{\boldsymbol{x},\boldsymbol{y}\} \in P$ and a pair of indexes (j,ℓ), and consider the probability that we have $x_j = y_\ell$. This probability is given by

$$\rho = \left(1-\frac{w}{n}\right)^2 + \frac{1}{q-1}\left(\frac{w}{n}\right)^2.$$

The algorithm will succeed if, for all possible pairs (j,ℓ), we have at least a couple of codewords $\{\boldsymbol{x},\boldsymbol{y}\}$ for which $x_j \neq y_\ell$. Given that P contains $\frac{L^2}{N_w}$ pairs, and that we have $x_j \neq y_\ell$ must happen for $n(n-1)$ pairs of indexes, we have that the success probability can be estimated as $\left(1-\rho^{\frac{L^2}{N_w}}\right)^{n(n-1)}$. □

Remark 6. The probability in Proposition 12 is an approximation of the actual success probability. Indeed, the proof of the proposition assumes that all pairs of indexes (j,ℓ) for which we have $x_j = y_\ell$ behave as random and uncorrelated variables, which is clearly ideal. Indeed, the actual distribution depends on the supports of the codewords which are present in P and, especially when P is small, the pairs of indexes are heavily correlated. Taking this phenomenon into account in a more accurate way would require a much more involved analysis. In any case, the probability expressed by Proposition 12 offers a crude, but appropriate and simple, approximation of the actual probability.

B.3. **Numerical confirmation.** In this section we present the results of some numerical simulations we have run, in order to validate the analysis of Beullens' algorithm we have performed in the previous sections. For our simulations, we have fully implemented the algorithm using Sage; the code we have used for the experiments is made fully available[5].

We start by considering the codewords finding algorithm; we have run the following experiment:

1. generate random pairs of codes, one being a permutation of the other;
2. for each couple of codes, cal ISD to find L distinct codewords with weight w;
3. for each couple of codes, run Algorithm 4 and have empirically measured the values of M' and M'';
4. average the obtained values of M' and M'', and compare with the theoretical estimates in Proposition 10.

[5]https://github.com/paolo-santini/LESS_project

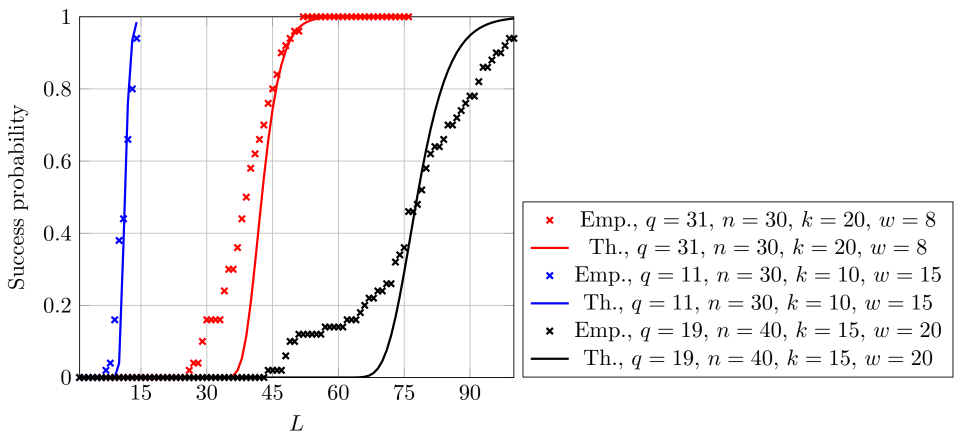


FIGURE 5. Success probability of the probabilistic permutation recovery as a function of L, for codes with different parameters. For every configuration, we have tested the attack on 50 codes. The empirical success probability has been computed by averaging over the trials.

In Table 4 we have reported a comparison between the theoretical values and the empirical ones.

(n, k, q)	w	L	M'		M''		$\lvert P\rvert$	
			th.	emp.	th.	emp.	th.	emp.
$(50, 25, 5)$	13	12	7.2	8.2	1.2	3.3	8.4	11.5
	14	100	47.3	47.6	71.1	268.9	118.4	316.5
$(40, 10, 11)$	23	40	31.5	31.0	$7.78 \cdot 10^{-4}$	0	31.5	31
	24	145	58.4	58.2	$7.4 \cdot 10^{-3}$	0.1	58.3	58.3
$(40, 10, 23)$	26	250	52.7	54.2	$1.9 \cdot 10^{-7}$	0	52.7	54.2
	28	2000	28.9	29.1	$3.9 \cdot 10^{-6}$	0	28.9	29.1
$(30, 10, 31)$	8	90	51.9	51.9	$2.9 \cdot 10^{-2}$	0	51.9	51.9
	9	200	3.5	4.5	$3.1 \cdot 10^{-2}$	0	3.5	4.5

TABLE 4. Comparison between numerical results and theoretical estimates on the composition of list P. For each triplet (n, k, q), the empirical results have been averaged over 10 random codes.

Finally, we have considered also the success probability of the permutation reconstruction algorithm; in Figure 5 we compare the obtained values with the theoretical ones, obtained through Proposition 12. The results show that Proposition 12 offer indeed a realistic approximation of the actual probability.

Appendix C. **Computing First Lexicographic Basis.** In this section we describe how the computation of Lex can be extended to the case of two-dimensional spaces; to avoid confusion with the one-dimensional case, we will refer to the operation as $\mathsf{Lex}^{(2)}$. We define $\mathsf{Lex}^{(2)}$ as the function that, on input $\boldsymbol{V} \in \mathbb{F}_q^{2\times n}$, returns the matrix $\boldsymbol{BV}$, with $\boldsymbol{B} \in \mathsf{GL}_2$ and such that the lexicographic minimum of $\{\tau(\boldsymbol{BV}) \mid \tau \in \mathsf{M}_n\}$ does not come after the lexicographic minimum of each

$\{\tau(\boldsymbol{B}^*\boldsymbol{V}) \mid \tau \in \mathsf{M}_n\}$, with $\boldsymbol{B}^* \in \mathsf{GL}_2 \setminus \boldsymbol{B}$. Note that this definition is a generalization to the two-dimensional case of the Lex function we have already considered in Section B.1.

To compute the first lexicographic basis of all possible monomial transformations of a matrix $\boldsymbol{V}$, we can operate as follows. We multiply each column of $\boldsymbol{V}$ by the inverse of the element in the first row, in order to remain with either zeros or ones in the first row. Now, we permute the columns of the obtained matrix with the goal of placing the zeros in the leftmost part of the first row. To do this, we consider the element in the second row: when two columns have the same element in the first row, we look at the element in the second row, and put on the left the one with the lowest entry. Finally, if we have some non null entry in the second row which corresponds to a null entry in the first row, we can scale the corresponding column to put a one in the second row. For the sake of clarity, in Figure 6 we show an example of this procedure.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 2 & 3 & 2 & 0 & 4 \\ 1 & 0 & 0 & 2 & 0 & 3 & 4 & 0 & 2 \end{pmatrix}$$

(A)

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 2 & 0 & 1 & 2 & 0 & 3 \end{pmatrix}$$

(B)

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 2 & 0 & 0 & 1 & 2 & 3 \end{pmatrix}$$

(C)

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 2 & 3 \end{pmatrix}$$

(D)

FIGURE 6. Example of lexicographic ordering of a basis, for the finite field with $q = 5$ elements. In figure (A) we show the initial basis, while in the other figures we detail the steps we perform to find the corresponding lexicographic minimum. The matrix in figure (B) is obtained by scaling all columns so that the entry in the first row is a 1. To obtain the matrix in figure (C), we sort the columns. Finally, we see that we have some degrees of freedom, since the third and fourth columns have a zero in the first row and a non null entry in the second row. Hence, we scale these columns and finally obtain the minimum lexicograph basis as in figure (D).

Then, given an input matrix $\boldsymbol{V} \in \mathbb{F}_q^{2\times n}$, we compute the matrices $\boldsymbol{BV}$, with $\boldsymbol{B} \in \mathsf{GL}_2$. We then perform the operations shown in Figure 6 and, for each $\boldsymbol{BV}$, keep only the resulting lexicographically minimum matrix. Finally, we compare all of such matrices and pick the one which comes first, in the lexicographically order. Notice that, for each input matrix, we test a total of $(q^2 - 1)(q^2 - q)$ basis, and for each basis we use $O(n)$ operations to find the lexicographic minimum matrix.

Each comparison requires $O(2n)$ operations, so that the computation of $\mathsf{Lex}^{(2)}$ costs $O\left(n(q^2-1)(q^2-q)\right)$ operations.[6]

Appendix D. Considerations about $\mathsf{Lex}^{(2)}$ and two-dimensional equivalent codes. In this appendix we estimate the probability to have bad collisions when considering the computation of $\mathsf{Lex}^{(2)}$ and Values on two-dimensional subcodes. We start with the following technical Lemma.

Lemma 3. Let $\boldsymbol{V}_1, \boldsymbol{V}_2 \in \mathbb{F}_q^{2\times n}$, with $\boldsymbol{A}_1 = \mathsf{Lex}^{(2)}(\boldsymbol{V}_1)$ and $\boldsymbol{A}_2 = \mathsf{Lex}^{(2)}(\boldsymbol{V}_2)$ being such that $\mathsf{Values}(\boldsymbol{A}_1) = \mathsf{Values}(\boldsymbol{A}_2)$. Then, the codes generated by $\boldsymbol{V}_1$ and $\boldsymbol{V}_2$ are equivalent.

Proof. We observe that, if $\mathsf{Values}(\boldsymbol{A}_1) = \mathsf{Values}(\boldsymbol{A}_2)$, then this means that there exist two monomials $\tau_1, \tau_2 \in \mathsf{M}_n$ such that $\tau_1(\boldsymbol{A}_1) = \tau_2(\boldsymbol{A}_2)$. Let $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$ be the associated matrices, and let $\boldsymbol{A}_1 = \boldsymbol{B}_1\boldsymbol{V}_1$, $\boldsymbol{A}_2 = \boldsymbol{B}_2\boldsymbol{V}_2$. Then, we have

$$\boldsymbol{B}_1\boldsymbol{V}_1\boldsymbol{Q}_1 = \boldsymbol{B}_2\boldsymbol{V}_2\boldsymbol{Q}_2 \implies \boldsymbol{V}_1 = \boldsymbol{B}_1^{-1}\boldsymbol{B}_2\boldsymbol{V}_2\boldsymbol{Q}_2\boldsymbol{Q}_1^{-1},$$

which corresponds to the definition of linear equivalence. □

Proposition 1. *Let $\mathfrak{C} \subseteq \mathbb{F}_q^n$ be a random linear code with dimension k. Let $\mathfrak{V} \subseteq \mathfrak{C}$ be a randomly-chosen, two-dimensional subcode with support size w. Then, the average number of two-dimensional subcodes $\mathfrak{B}' \subseteq \mathfrak{C}$ which are linearly equivalent to $\mathfrak{B}$ is upper bounded by*

$$t_w^{(2)} = \binom{n}{w} w!(q-1)^{w-1}\frac{\left[{k \atop 2}\right]_q}{\left[{n \atop 2}\right]_q}.$$

Proof. The bound is trivially obtained by considering all distinct monomial transformations of a code whose support is w, which is given by $\binom{n}{w}w!(q-1)^{w-1}$ (excluding monomial transformations which are identical, up to a scalar multiplication). Then, we consider that for every such code, there is a probability equal to $\frac{\left[{k \atop 2}\right]_q}{\left[{n \atop 2}\right]_q}$ that it is in $\mathfrak{C}$ (since $\mathfrak{C}$ is random). □

Remark 7. With some simple arithmetic manipulations, one can find that

$$t_w^{(2)} \leq n^w q^{-2(n-k)+w} = n^w q^{-2(1-R)n+w},$$

where R is the code rate. The behaviour of $t_w^{(2)}$ depends on the setting that one considers, namely, on the relation between n and q, but also on the desired values of w which, in turns, depend on further several factors (such as existence of subcodes with support size w and the actual cost to find subcodes). Taking all of this into account would require several cumbersome computations and would unnecessarily burden the presentation. Yet, the following simple reasoning is enough to discuss the expected behaviour of $t_w^{(2)}$ and, ultimately, the performance of Leon's algorithm.

- if we fix q and let n grow, then we expect that the optimal w is linear in n, so that the term n^w is super-exponential in n. Hence, at some point, we will have $t_w^{(2)} \geq 1$;

[6]Clearly, one can improve the computation of $\mathsf{Lex}^{(2)}$ using small weight codewords: this avoids to consider all $(q^2-1)(q^2-q)$ changes of basis. However, taking this into account would burden the description. Since this aspect does not affect significantly the complexity of the algorithms we analyze, we chose to consider only the trivial (and, perhaps, naive) computation for $\mathsf{Lex}^{(2)}$.

- when, instead, we fix n and increase q, we have that w may increase as well: the minimum support size we expect to find gets larger and, consequently, Beullens' algorithm is expected to be optimized with a larger value for w. However, at some point, w will saturate: for instance, regardless of q, any code contains two-dimensional subcodes with support size $n-k+2$. Setting $w = n-k+2 \approx n(1-R)$, we have that the term n^w is constant in q while $q^{-2(1-R)n+w} \approx q^{-n(1-r)}$ gets lower as q gets larger. Roughly speaking, the value of $t_w^{(2)}$ gets lower as q increases and this makes the existence of equivalent subcodes less likely. This reasoning describes why we expect Beullens' algorithm to perform better when q becomes larger.

Appendix E. Proof of Proposition 5.

Proposition 5. Let $\mathfrak{C}_1 \subseteq \mathbb{F}_q^n$ be a random linear code with dimension k, and let $\mathfrak{C}_2 = \tau(\mathfrak{C}_1)$ with $\tau \xleftarrow{\$} \mathsf{M}_n$. Let P be the list obtained by running Algorithm 1 with parameters L and w, with $w \leq n-k+2$. The algorithm runs in time

$$O\left(L\left(\log_2(L) + (q^2-q)(q-1)\right) + M' + M'' + \frac{L}{N_w^{(2)}} C_{ISD}(q,n,k,w)\right),$$

and produces a list P with $M = M' + M''$ elements, where $M' = L^2/N_w^{(2)}$ is the average number of good collisions and $M'' \leq \frac{t_w^{(2)}(L^2-M')}{N_w^{(2)}}$ is that of bad collisions.

Proof. For the running time of the algorithm, it is enough to repeat the same computation performed for Proposition 11. Hence, we only show how the number of good and bad collisions can be computed. The value of M' is estimated with the same reasoning we have adopted for the proof of Proposition 10. To estimate M'', we consider the following facts:

1) let $\boldsymbol{V}_1$ be the basis of a subcode $\mathfrak{V}_1 \in \mathfrak{C}_1$, and assume that $\mathfrak{V}_1$ is equivalent to u subcodes in $\mathfrak{C}_1$. According to Lemma 3, this means that there are u subcodes in $\mathfrak{C}_1$ which lead to a collision in the computation of Lex;
2) since $\mathfrak{C}_2$ is a monomial transformation of $\mathfrak{C}_1$, this means that also $\mathfrak{C}_2$ contains u subcodes that are equivalent to $\mathfrak{B}_1$;
2) we can use $t_w^{(2)}$ to upper bound the value of u (to see how $t_w^{(2)}$ is obtained, check Appendix D). Then, for any pair of drawn subcodes $\mathfrak{V}_1 \subseteq \mathfrak{C}_1$, $\mathfrak{V}_2 \subseteq \mathfrak{C}_2$, the probability that they are equivalent is upper bounded as $\frac{t_w^{(2)}}{N_w^{(2)}}$;
4) given that we draw L subcodes from each code, we have a total of L^2 pairs. We know that M' of them are good collisions, while for each remaining one there is a probability $t_w^{(2)}/N_w^{(2)}$ that it is a bad collision. Then, on average, the number of bad collisions is given by $\frac{t_w^{(2)}(L^2-M')}{N_w^{(2)}}$.

□

E-mail address: alessandro.barenghi@polimi.it
E-mail address: biasse@usf.edu
E-mail address: epersichetti@fau.edu
E-mail address: p.santini@staff.univpm.it