UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Elettronica
Curriculum: Smart and Secure Communication Networks

Studio di attacchi al problema dell'equivalenza tra codici lineari

Study of new attacks on the Code Equivalence Problem

Relatore:
Prof. Paolo Santini

Tesi di Laurea di:
Lei Chen

Correlatori:
Prof. Marco Baldi
Prof. Franco Chiaraluce

Anno Accademico 2022-2023

Abstract

Gli schemi crittografici asimmetrici, come le firme digitali, hanno una rilevanza fondamendale nell' ambito della sicurezza degli schemi di comunicazione e della cybersecurity. Essi si basano sulla difficoltà nel risolvere alcuni problemi matematici, che devono essere computazionalmente intrattabili (ovvero, risolubili solamente con algoritmi con complessità temporale esponenziale), altrimenti gli schemi non sarebbero sicuri.

Uno schema recentemente proposto è LESS, la cui sicurezza si basa sul Code Equivalence Problem, ovvero, sul problema di trovare, data una coppia di codici, un'equivalenza lineare che mappa un codice nell'altro. Un caso particolare del code equivalence è quello del Permutation Equivalence Problem (PEP), in cui si richiede che l'equivalenza sia una permutazione. LESS è tra i partecipanti al processo NIST per la standardizzazione di schemi di firma post-quantum e rappresenta un candidato interessante in quanto, grazie a recenti miglioramenti, può arrivare ad avere firme molto compatte.

Questa tesi avanza lo stato dell'arte sulla crittanalisi di PEP, migliorando alcuni degli attacchi esistenti e dimostrando come, in alcuni casi (quando i codici sono definiti su campi finiti non primi), istanze di PEP ritenute difficili possono invece essere risolte in tempo polinomiale. I risultati ottenuti sono giustificati tramite analisi teoriche e sono stati verificati sperimentalmente tramite simulazioni intensive (utilizzando Sagemath e Python).

Come primo contributo, è stato migliorato Support Splitting Algorithm (SSA), trovando una nuova modalità per calcolare la funzione di *signature* (usata come discriminante per ricostruire l'azione della permutazione). Rispetto alla funzione usata originariamente in SSA, la nuova funzione proposta in questa tesi mantiene la stessa complessità computazionale ma è più discriminante. Questo rende SSA globalmente più efficiente, poichè riduce il numero di step necessari per ricostruire la permutazione. I test effettuati confermano l'efficacia della funzione proposta.

Come secondo contributo della tesi, sono state proposte e studiate due modalità per trasformare istanze di PEP ritenute difficili in istanze di PEP che, invece, possono essere risolte in tempo polinomiale grazie a SSA o ad un risolutore per Graph Isomorphism Problem (GIP). Le trasformazioni considerate valgono solamente per codici su campi finiti di dimensione non prima. L' idea è quella di considerare PEP sui sottocodici di sottocampo, che mantengono la struttura della permutazione. Pur partendo da codici difficili da attaccare (chiamati *self-dual*), grazie alle trasformazioni proposte si riescono ad ottenere codici per cui PEP può essere risolto facilmente (con algoritmi il cui costo è polinomiale).

Questi risultati mostrano come istanze di PEP ritenute difficili siano in realtà risolvibili in tempo polinomiale. Questo significa che per LESS, così come per altri schemi crittografici, esistono scelte di parametri che rendono il problema vulnerabile. In particolare, grazie ai risultati presentati in questa tesi, si può concludere che lavorare su campi non primi è una scelta non sicura. Per questo motivo, si raccomanda fortemente di evitare di utilizzare codici di questo tipo per applicazioni crittografiche, poiché sono facili da attaccare.

Abstract

Asymmetric cryptographic schemes, such as digital signatures, have foundational relevance in communication scheme security and cybersecurity. They are based on the difficulty in solving certain mathematical problems, which must be computationally intractable (i.e., solvable only by algorithms with exponential time complexity), otherwise the schemes would not be secure.

A recently proposed scheme is LESS, whose security is based on the Code Equivalence Problem, that is, on the problem of finding, given a pair of codes, a linear equivalence that maps one code into the other. A special case of code equivalence is the Permutation Equivalence Problem (PEP), in which the equivalence is required to be a permutation. LESS is among the participants in the NIST process for standardizing post-quantum signature schemes and is an interesting candidate because, with recent improvements, it can go so far as to have very compact signatures.

This thesis advances the state of the art on cryptanalysis of PEP, improving on some of the existing attacks and demonstrating how, in some cases (when codes are defined over finite nonprimary fields), instances of PEP considered difficult can instead be solved in polynomial time. The results obtained are justified through theoretical analysis and have been verified experimentally through intensive simulations (using Sagemath and Python).

As a first contribution, Support Splitting Algorithm (SSA) was improved by finding a new way to compute the function of *signature* (used as a discriminant to reconstruct the permutation action). Compared with the function originally used in SSA, the new function proposed in this thesis retains the same computational complexity but is more discriminative. This makes SSA more efficient overall, as it reduces the number of steps required to reconstruct the permutation. The tests performed confirm the effectiveness of the proposed function.

As a second contribution of the thesis, two ways of transforming instances of PEP that are considered difficult into instances of PEP that, instead, can be solved in polynomial time using SSA or a Graph Isomorphism Problem (GIP) solver were proposed and studied. The transformations considered apply only to codes over finite fields of dimension not before. The idea is to consider PEP on subfield codes, which maintain the permutation structure. Although starting from codes that are difficult to attack (called *self-dual*), thanks to the proposed transformations we are able to obtain codes for which PEP can be solved easily (with algorithms whose cost is polynomial).

These results show how instances of PEP thought to be difficult are actually solvable in polynomial time. This means that for LESS, as well as for other cryptographic schemes, there are parameter choices that make the problem vulnerable. In particular, thanks to the results

presented in this thesis, it can be concluded that working on non-prime fields is an unsafe choice. For this reason, it is strongly recommended to avoid using such codes for cryptographic applications, as they are easy to attack.

Study of new attacks on the Code Equivalence Problem

Chen Lei

October 9, 2023

Contents

1 Introduction

In the field of telecommunications, security has always been a fundamental component. In fact, in a highly digitized world, the need to protect data and communications is absolutely essential. This is why we talk about cybersecurity, and need ways to protect our communications and devices.

In a digital communication through a public infrastructure, such as the Internet, one of the goals is for the transmitted message to reach its destination without being compromised. In other words, it is essential to ensure that the received content is not altered during transmission or to verify that the sender is indeed the one who produced the message. These needs are especially crucial when it comes to online banking, online financial transactions, transmission of legally valuable documents, electronic payments, cryptocurrency exchanges, and much more. The most powerful solution in the field of cybersecurity that guarantees these properties is *digital signatures*. They are used to verify the authenticity and integrity of a digital document or message.

A digital signature is based on asymmetric cryptography, also known as public-key cryptography. The signature is obtained through a pair of keys, one public and one private. The two keys are linked to each other through advanced mathematical algorithms. The private key, as the name suggests, is secret and is used to generate signatures for messages. Only the owner of the signature possesses this key. On the other hand, the public key is known to everyone and is derived from the private key. Its role is the opposite, namely to validate the signature signed by the corresponding private key. A fundamental property that these two keys must have is that computing the private key from the public key is computationally infeasible. In fact, the security of the signature is based on the difficulty of this calculation, ensuring the secrecy of the private key.

Since the 70s, many digital signature schemes have been invented, each with its own characteristics and complex mathematical problems to solve. Taking the most common schemes as examples, RSA exploits the problem of integer factorization: given two very large prime numbers, it is easy to calculate their product, but it is enormously difficult to factorize the product into its prime factors. Another example is the discrete logarithm problem, on which schemes like Diffie-Hellman, El Gamal, and ECC are based. Specifically, the computational complexity of any algorithm (attack) that solves factorization and discrete logarithm grows approximately as

$$2^{(\alpha \log_2(n))} \tag{1}$$

where n is the number to be factorized and α is some (positive) constant.

However, with the development of quantum computing, solving classical problems that underpin asymmetric cryptography will no longer be difficult. For example, Shor's algorithm allows for the polynomial-time solution of the factorization problem, easily breaking RSA. By applying appropriate modifications to the algorithm, it can also solve the discrete logarithm problem and elliptic curves.

Therefore, in 2017, the National Institute of Standards and Technology (NIST) launched a new competition for standardizing new public-key cryptographic schemes resistant to classical and quantum computers. After three rounds of selection, in 2022, NIST announced the first algorithms to be standardized. The public key encapsulation mechanism (KEM) to be standardized is CRYSTALS-KYBER. The digital signatures to be standardized are CRYSTALS-Dilithium, FALCON, and SPHINCS+. Except for SPHINCS+, all these schemes are based on problems involving structured lattices. However, due to the lack of diversity in computational problems used by signature schemes, at the same time, NIST announced an additional fourth round of selection. The deadline for submissions closed on June 1, 2023.

In this context, the cryptography research group from Università Politecnica delle Marche has submitted to NIST two digital signature schemes, in collaboration with other universities from Europe and USA. One of these is LESS, which stands for Linear Equivalence Signature Scheme [3–5, 7]. LESS is a Zero-Knowledge protocol and a digital signature scheme whose security is based on the Code Equivalence Problem. Thanks to recent developments [6], the scheme can achieve very compact signatures, e.g., less than 2kB for 128 bits of security, making it as one of the post-quantum schemes with the shortest signatures.

1.1 The Code Equivalence problem

A linear code is usually used for error correction, for example, to restore bits that get lost during communication or data saved in storage, or to correct bits affected by errors during a wireless communication. Yet, linear codes are also employed to build cryptographic schemes: very difficult problems can be thought of by exploiting linear codes.

One of these problems is the Code Equivalence Problem (CEP). Briefly, two linear codes are defined as equivalent when they have properties in common, like minimum distance and error correcting capability. Typically, when two codes are equivalent, there exists a (linear) transformation mapping one code into the other. An example of such transformations is that of permutations: two codes are equivalent if one applies a column permutation on one of them. In the context of cryptography, or precisely in the context of the LESS digital signature scheme, the secret key will be the permutation matrix while the public key are the two equivalent codes. Only the owner of the secret key can always demonstrate that the two matrices are equivalent. The security of the scheme is based on the fact that others, anyone unauthorized, cannot prove the equivalence. In other words, the scheme is broken when there are fast methods to find the secret permutation matrix.

1.2 Attacks on PEP

A schoolbook approach to solve PEP is to test all possible permutations and it is certainly impracticable. Yet, we know about more efficient ways to solve PEP. The only known efficient solvers for PEP are the Support Splitting Algorithm (SSA) [10] and the reduction to Graph Isomorphism Problem (GIP) [2], which are effective only when the *hull dimension* is small. By hull, we refer to the intersection between a code and its dual.

Let d denote the hull dimension. Then, SSA works only $d > 0$ and takes time complexity $O\left(n^3 + q^d \ln(n)\right)$. The algorithm is heuristic, i.e., the above expression holds only for random codes. It's not known if the algorithm can effectively solve all PEP instances. Let π be the secret permutation, and C, C' be two codes such that $C' = \pi(C)$. SSA works by applying a *signature function* S, defined as $S(C, i)$: the function takes as inputs the linear code and a position i. Whenever $j = \pi(i)$, then it must be $S(C, i) = S(C', j')$. Using the values of such a function, one can discover the images of each index and, in the end, reconstruct the permutation π.

The reduction to GIP, instead, takes time $O\left(n^{d+\omega} T_{\text{GIP}}(q, n)\right)$, where $\omega \in [2.3, ; 3]$ is the exponent for matrix inversion and $T_{\text{GIP}}(q, n)$ is a solver for GIP. Thanks to the recent breakthrough by Babai [1], we know that GIP can be solved in quasi-polynomial time. This implies that, when d is constant, all instances of PEP can be solved in time which is, in the worst case, quasi-polynomial.

Notice that, on average and especially when q is large, GIP can be solved in $O(n^2)$ time, making de facto PEP solvable in polynomial time given that d is constant. However, when dealing with codes whose hull dimension d grows with n, the cost of the algorithm becomes exponential. The same holds for SSA, since d becomes linear in n so the cost grows as $O(q^{\alpha n})$.

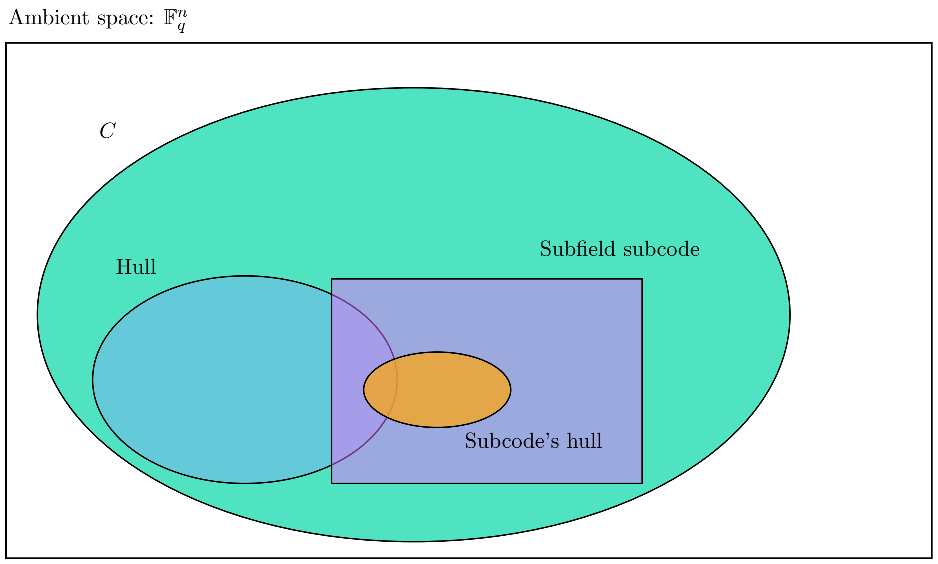


Figure 1: Relation between codes, subfield subcodes and hulls

1.3 Our contribution

In this thesis we give two main contributions. First, we improve the *signature function* employed in the SSA algorithm. Namely, we propose and study a signature function which can be computed in the same time required by the one in [10] but which is more discriminant. This implies that a smaller number of signature evaluations is required: this makes SSA faster. We prove this result by first analyzing theoretically the new signature function, and then confirm it through numerical simulations.

As the second and main contribution of this thesis, we show how to transform a PEP instance with large hull into another PEP instance with much smaller hull dimension. Our technique mainly consists in looking at subfield subcodes of the given codes. Such subcodes preserve the action of the secret permutation and, under certain circumstances (working over a non prime finite field, code rate which is properly low/high), have a hull whose dimension d' is, with very high probability, much smaller than d. To see the relation between codes, hulls and subfield subcodes, see Figure 1.

We show that, with large probability, d' is either 0 or equal to a small constant. This allows us to claim that, looking at subcodes, one can easily solve most of the instances that, prior to this work, were considered hard.

Notice that, even when d' is not necessarily bounded, the reduction turns out to be useful in several cases. First, the subcodes are defined over a subfield, and makes algorithms such as SSA faster. Moreover, if $d' < d$, we have been able to reduce the dimension of the problem, and this makes both SSA and GIP solvers faster.

The reduction we described above can generically be applied when the code rate R is such that $R > 1 - 1/\ell$[1]. However, there are several tweaks that one can apply to broaden the number of instances for which it works. For instance, when $R < 1 - 1/\ell$ (by attacking the dual codes) or, when the code rate does not fall in these cases, one can exploit the Frobenius intersection.

The effectiveness of the transformations we propose has been validated thanks to intensive numerical simulations, using Python and Sagemath. These simulations show that, for the wide majority of codes, we are able to reduce to a PEP instance with small hull. Moreover, we have verified that solving such instances allows to solve the initial PEP instance. That is, the reduction does not create spurious solutions and any solution to the newly obtained instance is indeed a solution for the initial instance. This confirms that our methods work and, in practice, can be effectively employed to solve PEP.

As a consequence, we highly recommend that these instances are not considered when building cryptosystems out of PEP, since the resulting schemes are likely insecure, in light of the attacks we present in this thesis.

[1]The reduction may work even when the rate does not respect such constraints; however, the number of codes for which this holds is very small.

2 Notation & Background

This section talks briefly about some basic concepts of Linear algebra and Code. They are preliminary notions at the basis of algorithm optimizations.

All testing, proof of concept and verification are done through the SageMath platform.

Throughout the thesis, if it is not explicitly specified, the italic q denotes a prime power, i.e., $q = p^m$ for a prime p and positive integer m.

Matrices and vectors are denoted by uppercase bold letters and lowercase bold letters respectively, e.g. $\mathbf{A}$ represents a matrix and $\mathbf{b}$ a vector. $\langle \mathbf{u}, \mathbf{v} \rangle$ denotes the inner product between $\mathbf{u}$ and $\mathbf{v}$, that is, a mathematical operation that takes two vectors and produces a scalar (a single value) as its result. Let n the dimensions of $\mathbf{u}$ and $\mathbf{v}$, the definition is as follow:

$$\langle \mathbf{u}, \mathbf{v} \rangle = \sum_{i=1}^{n} u_i v_i$$

If $\langle \mathbf{u}, \mathbf{v} \rangle = 0$, then $\mathbf{u}$ and $\mathbf{v}$ are *orthogonal* to each other.

The sequence $\{1, ..., n\}$ is denoted by $[n]$.

2.1 Finite fields

A finite field $\mathbb{F}_q$ is a field that contains q distinct elements; finite fields exists whenever q is a prime power, i.e., it is of the form p^m for $p \in \mathbb{N}$ being a prime and $m \in \mathbb{N}$ being a non null integer. The way operations are performed in the finite field depends on whether q is a prime or not.

Whenever q is a prime, then all the calculations are performed by reducing modulus q. For instance, if $q = 7$, then $3 + 4 = 0$ because $(3 + 4) \mod 7 = 7 \mod 7 = 0$. Mathematically, we say that the field is isomorphic to $\mathbb{Z}_q$ (i.e., the ring of all integer reduced modulus q) and express its elements as $\{0, 1, \cdots, q - 1\}$.

Instead, if q is a prime power (i.e., $m > 1$), the corresponding field will be called *Extension field*, then the operations get more involved. First, the field elements are in this case polynomials with coefficients over $\mathbb{F}_p$ and maximum degree $m - 1$. Namely, each element of the field can be expressed as $a(x) = \sum_{i=0}^{m-1} a_i x^i$. To define the field, one must first choose an irreducible polynomial $g(x)$ with degree m and coefficients over the base field

$\mathbb{F}_p$. Then, the finite field is defined by considering all operations $\mod g(x)$. This implies that the sum between two field elements is simply obtained by summing the coefficients referred to the same monomial, i.e.,

$$a(x) + a'(x) = \underbrace{\sum_{i=0}^{m-1} a_i x^i}_{a(x)} + \underbrace{\sum_{i=0}^{m-1} a'_i x^i}_{a'(x)} = \sum_{i=0}^{m-1} (a_i + a'_i) x^i.$$

For what concerns the product of elements, instead, we always have to consider that this is done via reduction $\mod g(x)$. Namely, the product between $a(x)$ and $a'(x)$ is $a(x) \cdot a'(x) \mod g(x)$.

Example

Considering a extension field $\mathbb{F}_q$ with $q = 2^5$, a possible primitive generator polynomial is $p(X) = 1 + X^2 + X^5$. Let α be a root of $p(X)$, then $p(\alpha) = 1 + \alpha^2 + \alpha^5 = 0$ and $\alpha^5 = 1 + \alpha^2$. After that, the elements of this field are polynomials with coefficients over $\mathbb{F}_2$ and maximum degree $4 = 5 - 1$. Now, let $v_1(\alpha) = 1 + \alpha^2$ and $v_2(\alpha) = 1 + \alpha^2 + \alpha^3$ be the two entry of the field, their sum and product are made in the following way:

Their sum is:

$$\begin{aligned} v_1(\alpha) + v_2(\alpha) &= 1 + \alpha^2 + 1 + \alpha^2 + \alpha^3 \\ &= (1 + 1) + (1 + 1)\alpha^2 + \alpha^3 \\ &= \alpha^3 \end{aligned} \tag{2}$$

And the product is:

$$\begin{aligned} v_1(\alpha) \cdot v_2(\alpha) &= (1 + \alpha^2)(1 + \alpha^2 + \alpha^3) \\ &= 1 + \alpha^2 + \alpha^3 + \alpha^2 + \alpha^4 + \alpha^5 \\ &= 1 + \alpha^3 + \alpha^4 + \alpha^5 \\ &= 1 + \alpha^3 + \alpha^4 + 1 + \alpha^2 \\ &= \alpha^2 + \alpha^3 + \alpha^4 \end{aligned} \tag{3}$$

2.2 Linear codes

A Linear code $C \subseteq \mathbb{F}_q^n$ of dimension k and length n is a k-dimensional subspace of $\mathbb{F}_q^n$. The quantity $R = \frac{k}{n}$ is called *code rate*, and any vector

$\mathbf{c} \in C$ is called *codeword.* A canonical representation for a code is through a *generator matrix*, that is, a full-rank matrix $\mathbf{G} \in \mathbb{F}_q^{k\times n}$ such that $C = \big\{\mathbf{uG} \mid \mathbf{u} \in \mathbb{F}_q^k\big\}$. In other words, $\mathbf{G}$ is a basis for the space associated with C: if the code has dimension k, then this space has dimension k and contains q^k codewords. Each codeword is generated as a linear combination of the rows of $\mathbf{G}$. Since $\mathbf{G}$ has full rank k, any two different linear combinations yield two different codewords.

2.2.1 Dual code

The Dual code of C, denoted $C^\perp$, is an $(n-k)$-dimensional linear subspace of $\mathbb{F}_q^n$ that is given by the following set:

$$C^\perp = \{\mathbf{w} \in \mathbb{F}_q^n : \langle \mathbf{w}, \mathbf{v}\rangle \ = 0 \text{ for all } \mathbf{v} \in C\} \tag{4}$$

That is, the set of all vectors that are orthogonal to codewords in C. Denoting $r = n-k$, we have that $C^\perp$ has generator matrix $\mathbf{H} \in \mathbb{F}_q^{r\times n}$ is called *parity-check matrix* and is such that $\mathbf{GH}^\top = 0$.

2.2.2 Change of basis

The general linear group formed by the non singular $k \times k$ matrices over $\mathbb{F}_q$ is indicated as GL_k. In other words, GL_k contains all $k \times k$ matrices with values over $\mathbb{F}_q$ which admit a multiplicative inverse.
Any code admits multiple generator matrices: for any $\mathbf{S} \in \mathrm{GL}_k$, which can be seen as a change of basis, it holds that $\mathbf{SG}$ and $\mathbf{G}$ generate the same code. The systematic generator matrix for a code is the matrix having the form $\big(\mathbf{I}_k, \mathbf{V}\big)$, where $\mathbf{V} \in \mathbb{F}_q^{k\times(n-k)}$. Computing this matrix is rather easy: if $\mathbf{G} = \big(\mathbf{A}, \mathbf{B}\big)$, with $\mathbf{A} \in \mathbb{F}_q^{k\times k}$ and $\mathbf{B} \in \mathbb{F}_q^{k\times(n-k)}$, then $\mathbf{V} = \mathbf{A}^{-1}\mathbf{B}$. Indeed, applying the change of basis $\mathbf{S} = \mathbf{A}^{-1}$, we get

$$\mathbf{SG} = \big(\mathbf{SA}, \mathbf{SB}\big) = \big(\underbrace{\mathbf{A}^{-1}\mathbf{A}}_{\mathbf{I}_k}, \underbrace{\mathbf{A}^{-1}\mathbf{B}}_{\mathbf{V}}\big).$$

Obviously, for any $\mathbf{S} \in \mathrm{GL}_r$, $\mathbf{H}$ and $\mathbf{SH}$ are parity-check matrices for the same code. Starting from a generator matrix in systematic form, one can easily find the parity-check matrix using the following relation

$$\mathbf{G} = \big(\mathbf{I}_k, \mathbf{V}\big) \iff \mathbf{H} = \big(-\mathbf{V}^\top, \mathbf{I}_r\big).$$

2.2.3 Hull of a Linear code

The *Hull* of a linear code C is defined as the intersection between the code and its dual. In mathematical form:

$$\mathcal{H}(C) = C \cap C^{\perp} \tag{5}$$

When the hull $\mathcal{H}(C)$ has dimension zero, we say that it is *trivial*. On the other hand, if the dimension of the hull is not zero, then the code is called *weakly self dual*.
We now recall the most important properties of the hull.

First, any vector in the hull is a codeword of C and is orthogonal to all codewords in C.
The hull is a linear subspace of $\mathbb{F}_q^n$. To see this, consider two vectors $\mathbf{c}, \mathbf{c}' \in \mathcal{H}(C)$, which implies

$$\mathbf{c} = \mathbf{u}\mathbf{G}, \ \ \mathbf{c}' = \mathbf{u}'\mathbf{G},$$
$$\mathbf{c} = \widetilde{\mathbf{u}}\mathbf{H}, \ \ \mathbf{c}' = \widetilde{\mathbf{u}}'\mathbf{H},$$

where $\mathbf{u}, \mathbf{u}' \in \mathbb{F}_q^k$ and $\widetilde{\mathbf{u}}, \widetilde{\mathbf{u}}' \in \mathbb{F}_q^r$. Let us now consider a linear combination of $\mathbf{c}$ and $\mathbf{c}'$ as

$$\mathbf{c}'' = \alpha\mathbf{c} + \beta\mathbf{c}',$$

where $\alpha, \beta \in \mathbb{F}_q^n$. We see that $\mathbf{c}'' \in \mathcal{H}(C)$; indeed

$$\mathbf{c}'' = \alpha\big(\mathbf{u}\mathbf{G}\big) + \beta\big(\mathbf{u}'\mathbf{G}\big) = \underbrace{\big(\alpha\mathbf{u} + \beta\mathbf{u}'\big)}_{\mathbf{u}''}\mathbf{G} = \mathbf{u}''\mathbf{G}, \tag{6}$$

$$\mathbf{c}'' = \alpha\big(\widetilde{\mathbf{u}}\mathbf{H}\big) + \beta\big(\widetilde{\mathbf{u}}'\mathbf{H}\big) = \underbrace{\big(\alpha\widetilde{\mathbf{u}} + \beta\widetilde{\mathbf{u}}'\big)}_{\widetilde{\mathbf{u}}''}\mathbf{H} = \widetilde{\mathbf{u}}''\mathbf{H}. \tag{7}$$

Equation (6) proves that $\mathbf{c}'' \in C$, while equation (7) proves that $\mathbf{c}'' \in C^{\perp}$. So, $\mathbf{c}''$ is also in $C \cap C^{\perp}$ and, consequently, is in $\mathcal{H}(C)$.

Since any vector in the hull is also a codeword of C, the hull of a code is a subcode of C. But since it is also a subcode of $C^{\perp}$, the hull dimension must be in the range $[1; \min\{k, n-k\}]$. As we will see in the next sections, the hull plays a crucial role in the complexity of SSA.

2.2.4 Permutations

We denote by S_n the symmetric group on n elements, and consider its elements as permutations of n objects. We represent a permutation as a bijection of $\{1, \cdots, n\} := [n]$, that is

$$\pi = \begin{pmatrix} 1 & 2 & \cdots & n \\ \pi(1) & \pi(2) & \cdots & \pi(n) \end{pmatrix}.$$

Obviously, the first row can be omitted, obtaining the so-called one-line notation $\pi := (i_1, i_2, \cdots, i_n)$, so that $\pi(j) = i_j$. In other words, π moves the j-th element to position i_j. For a vector $\mathbf{a} = (a_1, \cdots, a_n)$, it holds that

$$\pi(\mathbf{a}) = \big(a_{\pi^{-1}(1)}, \cdots, a_{\pi^{-1}(n)}\big).$$

Extending the action of permutations on matrices $\mathbf{A}$, i.e., $\pi(\mathbf{A})$ indicates the matrix resulting from the action of π on the columns of $\mathbf{A}$. Any permutation can be represented with a $n \times n$ matrix $\mathbf{P} \in \mathbb{F}_q^{n\times n}$, with the property that all rows and all columns contain a unique non null element (with value 1). It is easy to see that $\mathbf{P}^{-1} = \mathbf{P}^\top$ (where $^\top$ denotes transposition), so that $\mathbf{P}\mathbf{P}^\top = \mathbf{I}_n$.

We define $M_n = S_n \ltimes \mathbb{F}_q^{*n}$ as the group of monomial matrices (sometimes also called generalized permutations). For $\tau = (\pi, \mathbf{v}) \in M_n$, one has

$$\tau(\mathbf{a}) = \big(v_1 \cdot a_{\pi^{-1}(1)}, \cdots, v_n \cdot a_{\pi^{-1}(n)}\big).$$

The monomial can be represented as $\mathbf{PD}$, with $\mathbf{P}$ the permutation matrix describing π and $\mathbf{D}$ the diagonal matrix with main diagonal equals $\mathbf{v}$.

Example 1. Let $n = 5$ and $\pi = (4, 3, 2, 5, 1)$; then

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 3 & 2 & 5 & 1 \end{pmatrix}.$$

For $\mathbf{a} = (a_1, a_2, a_3, a_4, a_5)$, we have

$$\pi(\mathbf{a}) = \big(a_{\pi^{-1}(1)}, \cdots, a_{\pi^{-1}(n)}\big) = \big(a_5, a_3, a_2, a_1, a_4\big).$$

Let $q = 11$ and $\mathbf{A} = \begin{pmatrix} 7 & 0 & 3 & 2 & 5 \\ 10 & 2 & 1 & 5 & 8 \end{pmatrix}$. Then

$$\pi(\mathbf{A}) = \big(\mathbf{a}_5, \mathbf{a}_3, \mathbf{a}_2, \mathbf{a}_1, \mathbf{a}_4\big) = \begin{pmatrix} 5 & 3 & 0 & 7 & 2 \\ 8 & 1 & 2 & 10 & 5 \end{pmatrix}.$$

2.2.5 Puncturing

Given a linear code $C \subseteq \mathbb{F}_q^n$ with dimension k, we define its puncturing in positions $J \in [n]$ as the operation that returns the code

$$\mathcal{P}\big(C, J\big) = \Big\{\{c_i\}_{i \notin J} \mid (c_1, \cdots, c_n) \in C\Big\}.$$

For the sake of simplicity, sometimes we will write $C_{\setminus J} = \mathcal{P}(C, J)$; it can be seen that $C_{\setminus J}$ has dimension $\leq k$ and length $n - |J|$.

2.2.6 Shortening

Given a linear code $C \subseteq \mathbb{F}_q^n$ with dimension k, we define its shortening in positions $J \in [n]$ as the operation that returns the code

$$\mathcal{S}\big(C, J\big) = \mathcal{P}\left(C^*, J\right), \text{ where } C^* = \{\mathbf{c} \in C \mid \mathbf{c}_J = (0, \cdots, 0)\}.$$

For the sake of simplicity, sometimes we will write $C_{-J} = \mathcal{S}(C, J)$; it can be seen that C_{-J} has dimension $\leq k - |J|$ and length $n - |J|$.

Proposition 1. *Let C be a linear code with length n and $J \subseteq [n]$. Then*

- *the dual of $\mathcal{P}(C, J)$ contains $\mathcal{S}(C^\perp, J)$;*
- *the dual of $\mathcal{S}(C, J)$ contains in $\mathcal{P}(C^\perp, J)$.*

Proof: Let $\mathbf{v} \in C^\perp$ such that $\mathbf{v}_J = (0, \cdots, 0)$; then, for any $\mathbf{c} \in C$, it holds that

$$0 = \mathbf{c}\mathbf{v}^\top = \underbrace{\mathbf{c}_J\mathbf{v}_J^\top}_{\mathbf{c}_J \cdot (0,\cdots,0)^\top = 0} + \mathbf{c}_{\setminus J}\mathbf{v}_{\setminus J}^\top = \mathbf{c}_{\setminus J}\mathbf{v}_{\setminus J}^\top.$$

The first thesis is proven by considering that $\mathbf{v}_{\setminus J} \in \mathcal{S}(C^\perp, J)$ and $\mathbf{c}_{\setminus J} \in \mathcal{P}(C, J)$. For the second thesis, we consider that for any codeword $\mathbf{c} \in C$ such that $\mathbf{c}_J = (0, \cdots, 0)$, we have $\mathbf{c}_{\setminus J} \in \mathcal{S}(C, J)$; then, for any $\mathbf{v} \in C^\perp$, it holds that

$$0 = \mathbf{c}\mathbf{v}^\top = \mathbf{c}_{\setminus J}\mathbf{v}_{\setminus J}^\top + \underbrace{\mathbf{c}_J\mathbf{v}_J^\top}_{(0,\cdots,0)\mathbf{v}_J^\top = 0} = \mathbf{c}_{\setminus J}\mathbf{v}_{\setminus J}^\top.$$

Since $\mathbf{v}_{\setminus J} \in \mathcal{P}(C^\perp, J)$, this proves the second thesis. ■
A summary of the properties of punctured and shortened codes is shown in Table 1.

	New length	New dimension	New redundancy	New cdal code
$\mathcal{P}(C, J)$: puncturing on z positions	$n - z$	$\leq k$	$\geq n - k - z$	Contains $\mathcal{S}(C^\perp, J)$
$\mathcal{S}(C, J)$: shortening on z positions	$n - z$	$\leq k - z$	$\geq n - k$	Contains $\mathcal{P}(C^\perp, J)$

Table 1: Summary of properties for puncturation and shortening, for a code C with length n and dimension k; the size of the considered set is $|J| = z$.

2.3 Code Equivalence

We consider codes endowed with the Hamming metric, which is defined as

$$\mathrm{wt}(\mathbf{a}) = |\{i \mid a_i \neq 0\}|\,.$$

Since monomials shuffle coordinates and apply scaling factors, the number of zeros is preserved: for this reason, we say that monomials are *isometries* for the Hamming metric. Since permutations are a special case of monomials (monomials with scaling factors all equal to 1), the same holds for permutations.

For a code $C \subseteq \mathbb{F}_q^n$ and $\tau \in M_n$, we denote

$$\tau(C) = \{\tau(\mathbf{c}) \mid \mathbf{c} \in C\}\,.$$

In other words, $\tau(C)$ is the code obtained by applying τ to all codewords of C. It is easy to see that C and $\tau(C)$ have some common properties, such as the same minimum distance and the same weight distribution. For this reason, we say that C and $\tau(C)$ are *equivalent*.

If $\mathbf{G} \in \mathbb{F}_q^{k\times n}$ is a generator matrix for C, then a generator for $C' = \tau(C)$ is of the form

$$\mathbf{G}' = \mathbf{S}\tau(\mathbf{G}) = \mathbf{SGPD},$$

where $\mathbf{S} \in \mathrm{GL}_k$ is any change of basis. Equivalently, if $\mathbf{H}$ is a parity-check matrix for C, then a parity-check matrix for C' is in the form

$$\mathbf{H}' = \mathbf{ZHPD}^{-1},$$

where $\mathbf{Z} \in \mathrm{GL}_{n-k}$. This is very easy to prove: indeed

$$\begin{aligned}\mathbf{H}'\mathbf{G}'^{\top} &= \mathbf{ZHPD}\big(\mathbf{SGPD}^{-1}\big)^{\top} = \mathbf{ZH}\underbrace{\mathbf{PDD}^{-1}\mathbf{P}^{\top}}_{=\mathbf{I}_n}\mathbf{G}^{\top}\mathbf{S}^{\top} \\ &= \mathbf{Z}\underbrace{\mathbf{HG}^{\top}}_{=\mathbf{0}}\mathbf{S}^{\top} = \mathbf{0}.\end{aligned}$$

2.4 Code Equivalence Problem

The problem of finding an isometry between two given codes is called *Code Equivalence Problem*. Normally, two versions of the problem are considered, depending on the type of isometry one wants to determine.

Problem 1. ***Linear Equivalence Problem (LEP)***
Given two linear codes $C, C' \subseteq \mathbb{F}_q^n$ with dimension k, find a monomial $\tau \in M_n$ such that $C' = \tau(C)$. In other words, given $\mathbf{G}, \mathbf{G}' \in \mathbb{F}_q^{k\times n}$, find $\mathbf{S} \in \mathrm{GL}_k$ and $\tau \in M_n$ such that $\mathbf{G}' = \mathbf{S}\tau(\mathbf{G})$.

Problem 2. ***Permutation Equivalence Problem (PEP)***
Given two linear codes $C, C' \subseteq \mathbb{F}_q^n$ with dimension k, find a permutation $\pi \in S_n$ such that $C' = \pi(C)$. In other words, given $\mathbf{G}, \mathbf{G}' \in \mathbb{F}_q^{k\times n}$, find $\mathbf{S} \in \mathrm{GL}_k$ and $\pi \in S_n$ such that $\mathbf{G}' = \mathbf{S}\pi(\mathbf{G})$.

Solvers for the two problems may be significantly different. For the thesis, the focus will be only on PEP.

2.5 Computational complexity, decision problems and search problems

Computational complexity is a way to measure how efficient computer algorithms are. It helps to understand how the time and memory required by an algorithm grow as the size of the problem it's solving gets larger. In the context of asymmetric cryptography, the security of cryptographic schemes is based on the difficulty of solving the mathematical problem behind the scheme. Therefore, from this point of view, the scheme is secure when there is no efficient algorithm for solving the problem.

Typical notation used for representing the complexity is the *Big O notation*. The efficiency of the algorithms can be classified into the following complexities:

- $O(n^a)$ indicates that the complexity grows in polynomial time with n. Cryptographic schemes that are based on a problem that is solvable in polynomial time are considered insecure.
- $O(a^n)$, for $a > 1$, indicates that the complexity is exponential with n. A cryptographic scheme is safe if, in the literature or in the next few decades, only algorithms, for solving the problem behind the scheme, with this complexity exist.

Decision problems and Search problems Decision problems and search problems are two fundamental types of computational problems in computer science and mathematics. The decision problems focus on answering yes or no, while search problems aim to find actual solutions. Taking the example of the code equivalence problem, a decision problem could be asking if the

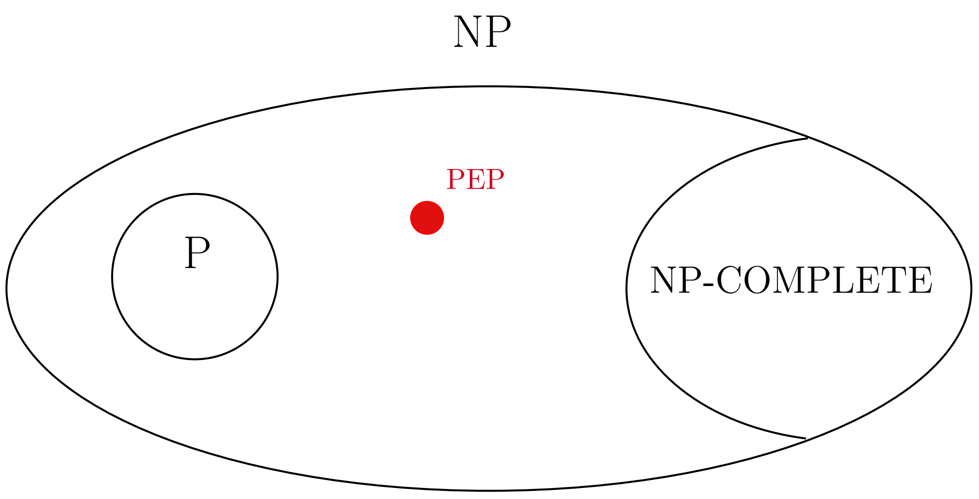


Figure 2: Relation between PEP and complexity classes

two given code are equivalent, while a search problem could be a task to find the permutation matrix that connect these two codes.

A problem lives in NP if its solutions can be verified in polynomial time. Given a problem with large input n, it lives in P if there is at least one algorithm that solves it in time limited by a polynomial in n. NP-complete problems are among the most difficult problems in NP. If a problem can be reduced into an existing NP-complete problem in polynomial time, it implies that this problem is at least as hard as the hardest problems in NP. So, if one could resolve in polynomial time of one of the problem in NP-complete, then all problems in NP-complete are resolvable in polynomial time.

Hardness classification for decisional PEP It has been proven in [8] that NP-completeness of PEP would imply collpase of the polynomial hierarchy, i.e., would imply P = NP. There are many evidences against such a relation, so that the wide majority of researchers believe that P $\neq$ NP. Assuming that P is indeed different from NP, right now we can only place PEP in a class which is in between P and NP-complete as in Figure 2.

Indeed, we do not know about polynomial time solvers for PEP, so we cannot say that PEP is in P. Notice that, for a problem to be in P, we require to know about an algorithm that can solve in polynomial time all instances. For PEP, we only know about algorithms that take polynomial time only when the input PEP instance has some property (e.g., trivial hull). For weakly-self dual codes, the time complexities of these algorithms become exponential. Existence of such hard instances is exactly why we cannot say that PEP is in P.

2.6 Graph Isomorphism Problem

The Graph Isomorphism Problem (GIP) has some connections with Code Equivalence Problem. First, similarly to the Code Equivalence Problem, GIP cannot be NP-complete unless the polynomial hierarchy collapses. For many years, the problem has been deemed hard, so that all known solvers were (worst case) sub-exponential time algorithms. Actually, in recent years, Babai, a professor at the University of Chicago has found a solver algorithm that solves the Graph Isomorphism problem with quasi-polynomial time [1].

2.6.1 Reduction from PEP to GIP

Thanks to Professor Babai's contribution, now everyone knows that graph isomorphism problems can be solved in quasi-polynomial time algorithms. We now describe how solvers for GIP can be used to solve PEP, when the considered codes have trivial hull.

First, we observe that GIP can be formulated as follows:

$$\mathbf{A}' = \mathbf{P}^\top \mathbf{A}\mathbf{P}, \tag{8}$$

where $\mathbf{A}$ is the adjacency matrix of the starting graph, while $\mathbf{P}$ is the permutation matrix finally, $\mathbf{A}'$ is the adjacency matrix of the graph after the isomorphism. As it can be seen from the equation, it has a similarity with the PEP equation $\mathbf{G}' = \mathbf{SGP}$, where instead of the basis change matrix $\mathbf{S}$ there is $\mathbf{P}^\top$. Also in this problem, the unknown is $\mathbf{P}$. Actually, certain instances of PEP can be reduced to the GIP; technically, this is called reduction from PEP to GIP. We proceed by describing how such a reduction works.

Let $\mathcal{G} : \mathbf{G} \mapsto \mathbf{G}^\top(\mathbf{G}\mathbf{G}^\top)^{-1}\mathbf{G} = \mathbf{A}$ be the function that, on input some matrix $\mathbf{G}$, returns the transformed matrix $\mathbf{A}$. This transformation has a constraint on $\mathbf{G}$, i.e. $\mathbf{G}\mathbf{G}^\top$ must not be a singular matrix. Therefore, it works when the hull of $\mathbf{G}$ is trivial.

Applying $\mathcal{G}$ to the PEP's equation, the equation becomes:

$$\begin{aligned}\mathcal{G}(\mathbf{G}') &= \mathbf{G}'^\top(\mathbf{G}'\mathbf{G}'^\top)^{-1}\mathbf{G}' \\ &= \mathbf{A}'.\end{aligned} \tag{9}$$

Instead, for the second part of the equation, we get:

$$\begin{aligned}\mathcal{G}(\mathbf{SGP}) &= \mathbf{P}^\top\mathbf{G}^\top\mathbf{S}^\top(\mathbf{SGPP}^\top\mathbf{G}^\top\mathbf{S}^\top)^{-1}\mathbf{SGP}\\ &= \mathbf{P}^\top\mathbf{G}^\top\mathbf{S}^\top(\mathbf{SGG}^\top\mathbf{S}^\top)^{-1}\mathbf{SGP}\\ &= \mathbf{P}^\top\mathbf{G}^\top\mathbf{S}^\top(\mathbf{S}^{-\top}(\mathbf{GG}^\top)^{-1}\mathbf{S}^{-1})\mathbf{SGP}\\ &= \mathbf{P}^\top\mathbf{G}^\top(\mathbf{GG}^\top)^{-1}\mathbf{GP}\\ &= \mathbf{P}^\top\mathbf{AP},\end{aligned} \tag{10}$$

obtaining a new equation:

$$\mathbf{A}' = \mathbf{P}^\top\mathbf{AP}, \tag{11}$$

where $\mathbf{A}' = \mathcal{G}(\mathbf{G}')$ and $\mathbf{A} = \mathcal{G}(\mathbf{G})$.

This means that, starting from two codes, we found that a permutation mapping one code into the other exists if and only if the corresponding graphs are isomorphic. Now, it is possible to solve PEP using a solver for GIP. If we are interested in solving the search version, then we can get the permutation matrix $\mathbf{P}$ with a GIP solver.

3 The Support Splitting Algorithm (SSA)

In this section we recall the working principle of the Support Splitting Algorithm (SSA).

3.1 SSA structure

The support splitting algorithm has been introduced by Sendrier in 2000. It's an algorithm that solves the permutation equivalence problem by reconstructing the permutation index by index. In other words, given two matrices which, one of these generates the same code of the other up to a column permutation. It finds for each column of one matrix, the corresponding index after the permutation. The algorithm is a composition of three main techniques:

- *Hull*: Since the hull is a subcode, therefore, it may contain fewer codewords. This optimizes the attack.
- *Weight distribution function*: This function, denoted by $WEF(C)$, yields the weight distribution of all codewords of a code. In a standard way, it is done by computing all the codewords so, if the code C has dimension k, computing $WEF(C)$ takes time $O(q^k)$ where q is the finite field's cardinality.
- *Puncturing*: For a code C, puncturing it in a position i means reconstructing the code in such a way that all the words of the new code, compared to the original word, the i-th element is removed. It is denoted by $\mathcal{P}_i(C)$. This operation is very easy to do, just remove the i-th column of main code's generator matrix $\mathbf{G} \in \mathbb{F}_q^{k \times n}$, resulting a matrix of dimension $k \times (n-1)$.

Now is possible to proceed to describe the algorithm. For the sake of simplicity, the following approach finds only the first index. But it can be generalized to find all other indices. To recap, C is the original code and C' is the permuted one.

1. Compute the hull of both codes, so $B = \mathcal{H}(C)$ and $B' = \mathcal{H}(C')$.
2. Puncture the hull B in position 1, obtaining $B_{\setminus 1} = \mathcal{P}_1(C)$.
3. Compute the weight distribution function of the obtained punctured code, i.e., $W_1 = WEF(B_{\setminus 1})$.

4. For each $i \in [n]$, puncture the hull B' in position i, obtaining $B'_{\setminus i} = \mathcal{P}_i(C')$.
5. For each $i \in [n]$, compute the weight distribution function of $B'_{\setminus i}$, obtaining $W'_i = WEF(B'_{\setminus i})$.
6. If there is a unique i such that $W_1 = W'_i$, conclude that 1 is permuted to i.

In the case which there are more indices i that satisfy the equation $W_1 = W'_i$, that is, there are colliding indices, the step six needs to be *refined.* But this case is unlikely to happen if we work on a large cardinality q of finite field so, it can be neglected for the moment.

3.2 Observations

The idea behind the algorithm is actually very simple. The algorithm exploits some fact about codes:

- An *invariant* is a property of a code that does not vary if the code is permuted.
- Two equivalent codes have the same weight distribution of codewords, i.e., $WEF(C) = WEF(C')$. So the weight distribution function is an *invariant* function.
- the previous statement is also valid for the hull of the code. In other words, $WEF(\mathcal{H}(C)) = WEF(\mathcal{H}(C'))$.

The code punctured on the first column has as its generator matrix the composition only of columns $\{\mathbf{g}_2, ..., \mathbf{g}_n\}$ of the original code matrix. If the corresponding i-th column is eliminated from the permuted code matrix, for i which is equivalent to the index of 1 after the permutation. These two codes will have the same columns up to reordering. Therefore, since these two codes differ only by a permutation, their weight distribution of codewords will be the same, that is, $WEF(B_{\setminus 1}) = WEF(B_{\setminus i}),$ for $i = \pi(1)$. On the other side, if the wrong column of second code matrix is deleted, they will differ in some columns (precisely, two columns) so, it is very likely that they have a different weight distribution.

Computational complexity The SSA is a combination of many procedures. Following the steps of the algorithm, first of all, computing the hull of the two codes takes time $O(2n^3)$, since it can be done with linear algebra. Let d denote the dimension of the hull. Then computing the weight distribution of hull $WEF(B_{\setminus 1})$ takes time $O(q^d)$, the same calculation $WEF(B'_{\setminus i})$ is repeated n times on the hull of the second code, i.e. the permuted one. Finally, combining these large O's together gives a complexity of

$$O(2n^3 + (n+1)q^d) \tag{12}$$

to draw the first permuted index.

The role of hull If the calculation of weight distribution were not applied to the code hull, but directly to the code itself, the complexity would have been $O(q^k)$. Since the hull is a subcode, its dimension d must be $d \leq k$. So, using the hull, one could reduce the computational complexity.
Furthermore it has been proven that, for random codes, the hull dimension is extremely small with very high probability [9]. This means that the term q^d becomes a very small constant (i.e., something like q^2 or q^3). Since all the remaining terms in the complexity are polynomials of n, the attack runs in polynomial time.
On the other side, to keep the problem still difficult, normally *self-dual* codes are used. That is, codes that are equal to their dual. This means that $C = C^{\perp}$ or, equivalently, that any generator matrix $\mathbf{G}$ is also a parity-check matrix, which implies $\mathbf{G}\mathbf{G}^{\top} = \mathbf{0}$. In such cases, the hull $\mathcal{H}(C) = C$, so the hull has maximum dimension $d = k$. The time complexity of the algorithm is dominated by $O(q^k)$ so, it is exponential in the code dimension.
The maximum dimension of hull is reached when *self-dual* codes are used. So, for the definition of *self-dual* codes, the value of code dimension is equal to the value of redundancy: $k = r \Rightarrow n = k+r = k+k \Rightarrow n = 2k \Rightarrow k = \frac{1}{2}n$. Then the code rate is $R = \frac{1}{2}$, namely, the time complexity is also exponential in the code length.

If there exist a efficient probabilistic test In the crucial step of the SSA algorithm, that is, after the two equivalent codes C and C' have been punctured, the algorithm compares whether the second is still equivalent with the first. The methods existing so far are based mainly on the calculation of the weight distribution of the code, when the two codes have the same weight distribution they are likely to be equivalent, while when they are not, they are certainly not equivalent.

This operating principle resembles that of the test of Fermat in the prime number determination. The Fermat test interrogates the number several times, if the test fails it is certainly not prime. Otherwise, the number may be a prime. Therefore, by repeating this procedure many times the probability that the number is a prime will tend to one. In fact this test works in almost all cases.
Returning to the case of the PEP, if there is an efficient test that can compare and say "no" when codes are not equivalent, then all cryptographic schemes based on PEP will be immediately broken.

4 SSA Implementation

This section illustrates the steps to implement the SSA algorithm. We did a proof of concept for SSA, using SageMath, a Python-based free open-source mathematics software system. It includes many mathematical packages therefore, speeds up the various tests.

It is evidently very cumbersome to write all the required algorithm code in one go. So, at first time, the algorithm functions had been divided into several files, after the test of functionality they had been reunited in a single script. All essential codes for implementation are attached in the appendix.

4.1 Generation of two equivalent codes

The first thing is to write a script for testing if a linear code, after a permutation and base change, is equivalent to the initial. To generate a totally random code,the representative generator matrix is required. SageMath provides the *random_matrix* function. But it needs some parameters, namely, the Galois field on which it is defined, the number of rows k and the number of columns n. The k and n are equivalent to the dimension and length of the code, respectively. The Galois field is defined using *GF* function, in which the field order is passed. This field will also be used to generate other matrices defined on the same field. After the set up of the random code matrix, the random change of basis matrix with size $k \times k$ is generated. It may happen that the create change of basis is not full rank, in such case the matrix must be regenerated. This is done by inserting a while loop that will not terminate until the matrix's rank is not equal to the dimension k or until the determinant is not different from zero. The last matrix needed is the permutation matrix, it is a matrix that has a 1 for each column and row. SageMath already provides the function to generate such a matrix. Just generate a permutation object through the *Permutations* function with size n, then get a random permutation and convert it to a matrix. At this point is possible to apply the matrix products to obtain the permuted code matrix by

```
G_prime = S*G*P
```

where G is the initial generator matrix, S the change of basis and P the permutation matrix.
Finally, to check whether these two codes are equivalent, the most common method is comparing their weight distribution. So the two matrices

are converted to their respective code objects using *codes.LinearCode* function. and then pull out their weight distribution by calling the method *weight_distribution*. Therefore, both by eye and the comparison of equality operator == have shown that they are the same.

A further test was also done on a codeword of the original code to verify if, after the permutation, it belongs to the permuted code. To generate a codeword, a random vector of length k was chosen. Then its multiplied by the code generator matrix obtaining the codeword $\mathbf{v}$ of length n. Subsequently, $\mathbf{v}$ is permuted with the initially generated permutation matrix. To check if $\mathbf{v}$ is a codeword of the permuted code, it have to be multiplied with the parity matrix of the permuted code to obtain the syndrome. If the syndrome is equal to a zero-vector, then it means that $\mathbf{v}$ is a legal codeword. Doing the calculation, it is actually a codeword.

4.2 Hull calculation

The hull of a code is a subcode obtained from the intersection of code itself with its dual code, the construction of this subcode is rather simple: First step is to define a vector space $\mathbb{F}_q^n$ of dimension n over a finite field of q element, the function that returns a vector space object is called *VectorSpace*. Using it, through the method *subspace* and, passing the code generator matrix as parameter, the two subspaces V_1 and V_2 are created. V_1 and V_2 are respectively the vectorial representation of the code and its dual. Now the intersection space $V_1 \cap V_2$ between the two subspaces V_1 and V_2 can be obtained by executing the method *intersection* of one subspace on the another one. In the end, the basis of intersection space $V_1 \cap V_2$ are exactly the basis of hull. So, the generator matrix of hull is rebuilt by calling the method *basis* on $V_1 \cap V_2$ with a conversion into matrix.

4.3 Generate weakly self dual codes

A totally random code has usually the hull with small dimension (i.e., something like 2 or 3). This makes the attack very easy, therefore it does not have much importance in the study of the attack. The interests are on codes that have a high hull dimension. Those codes are also called weakly self dual code. One method to generate weakly self dual codes is to repeat the random code generation until a code with the desired hull dimension is generated. But evidently, this approach is impracticable for high values of hull dimension. An simple and effective way is the following:

1. First, generate a random vector $\mathbf{v}$ with length n on the finite field $\mathbb{F}_q$, with *random_vector* function.

2. Check if this vector $\mathbf{v}$ is a *self-orthogonal* codeword, if not, repeat the random vector generation. To test $\mathbf{v}$'s self-orthogonality just compute the inner product of $\mathbf{v}$ with itself. If the result is zero, then it is *self-orthogonal*. There are cases that the random vector $\mathbf{v}$ is a null vector or it's linear dependent with other rows of $\mathbf{G}$, in these cases it is not considered a qualified vector.

3. Convert the vector $\mathbf{v}$ into a $1 \times n$ matrix $\mathbf{G}$ with *matrix* function.

4. Convert matrix $\mathbf{G}$ to it's associated code C with *codes.LinearCode* function.

5. Get the dual code $C^{\perp}$ of C with method *dual_code*.

6. Generate a random codeword $\mathbf{w}$ from code $C^{\perp}$ with method *random_element*.

7. Check if the just made codeword $\mathbf{w}$ is a *self-orthogonal* codeword, with the same rule as point 2.

8. Append this codeword $\mathbf{w}$ to matrix $\mathbf{G}$ as a new row.

9. Compute the hull of $\mathbf{G}$, verify if its dimension reaches the required value. If is true, proceed, otherwise return to point 4.

10. Get a random codeword $\mathbf{u}$ from the dual code of $\mathbf{G}$, but this time, proceed if the codeword $\mathbf{u}$ is not a *self-orthogonal* codeword.

11. Append this codeword $\mathbf{u}$ to matrix $\mathbf{G}$ as a new row.

12. Get the rank of $\mathbf{G}$, verify if its dimension reaches the required value. If is true, a desired weakly self dual code is yielded, otherwise return to point 10.

At point 8, the *self-orthogonal* codeword $\mathbf{w}$ appended to matrix $\mathbf{G}$ makes sure that the codewords generated by $\mathbf{G}$ are still *self-orthogonal* so, these basis are also in the dual space. Since $\mathbf{w}$ is caught from the dual code, it may isn't linear dependent with other rows before appending.

4.4 Implementing SSA

After the implementation of the previous functions, at this point, the algorithm is very simple to write.
The value of each parameters are rather small, because values that are too high cause the algorithm to slow down. especially for the length n of the code and the dimension of the hull, The amount of computation required for the latter is exponential.
First of all, for the two generated equivalent codes, their hulls B and B' are considered. Then, the puncturing is applied on the hull B's first column, that is, the first column is removed. This operation can be done by *delete_columns* method of matrix of hull. Consequently, the weight distribution of punctured hull $B_{\setminus 1}$ is saved. Thereafter a for loop of n times is executed, for each loop, the puncturing of i-th index is applied on the hull of the permuted code, then, after puncturing, its weight distribution $WEF(B'_{\setminus i})$ is calculated and compared with $WEF(B_{\setminus 1})$ counted previously. If they are equal, the index i is considered colliding and it is saved.
Several tests were carried out (the corresponding code is in the Appendix), the Tables 2, 3, 4 and Figure 3 contains the results of the colliding indexes obtained by fixing n and k, varying q and *hull_dim*. The *mean_colliding_indices*, (if the value is one, then there are no collisions) is retrieved on about 100 tests. As can be seen from tables, increasing the dimension of hull, the colliding indices tend to be one. And the tendency is faster for higher values of q.

4.5 Code shortening

The use of shortening instead of puncturing has the advantage on the dimension k of the code to which it is applied, that is, while the puncturing does not change the dimension, the shortening reduces the dimension by the number of columns removed. This has a positive impact on the calculation of the weight distribution, considering the cost is exponential in the code dimension k. SageMath provides a method *shortened* for Linear code objects that returns the code shortened in specific columns. But if one wants to realize the shortening on their own, the procedures are very simple. Namely: first consider the parity matrix of the code, and add to it, for each column to be removed, a n-length vector of all zeros except in the index of the column where the value is different from zero. Then convert back to code and consider its dual. Remove the specified columns from the generator matrix of the dual code, if the procedures are performed correctly, these

n	k	q	*hull_dim*	mean_colliding_indices
20	10	2	2	6.6
			3	5.6
			4	4.1
			5	3.5
			6	3.9
		3	2	9.7
			3	5.5
			4	4.5
			5	3.0
			6	2.6
			7	2.1
			8	2.6
			9	4.1
			10	19.1
		5	2	6.2
			3	2.2
			4	1.1
			5	1.0
			7	1.0
			9	1.0
			10	1.7

Table 2: Average number of colliding indices, for SSA and several code parameters. For each set, we have performed 100 experiments. For all the considered instances, we have used $n = 20$.

columns are columns of zeros. The new matrix thus obtained has dimensions $(k-l) \times (n-l)$, for l the number of columns to delete.

4.6 Sendrier's signature

The meaning of the *signature* in this context, is a function that is applied to the code, and has the property that if the two codes are equivalent, they certainly have the same *signature* value. In Sendrier's article, precisely in section 5.2, he introduced a *signature* made this way:

$$S : (C, i) \to (WEF(\mathcal{H}(C_{\setminus i})), WEF(\mathcal{H}(C^{\perp}_{\setminus i}))) \tag{13}$$

n	k	q	*hull_dim*	mean_colliding_indices
15	5	2	1	7.58
			2	6.12
			3	6.02
			4	4.77
			5	4.96
		3	1	10.23
			2	8.97
			3	5.22
			4	4.85
			5	4.27
		5	1	12.11
			2	5.93
			3	2.11
			4	1.25
			5	1.06

Table 3: Average number of colliding indices, for SSA and several code parameters. For each set, we have performed 100 experiments. For all the considered instances, we have used $n = 15$.

Compared to the signature used so far for the tests, which looks like this:

$$F : (C, i) \to (WEF(B_{\setminus i})) \text{ with } B = \mathcal{H}(C) \tag{14}$$

The differences essentially are that, In Sendrier's *signature* first punctures the code then calculates the hull instead of vice versa. Furthermore he considers a pair of values where the second has replaced the code argument with its dual.

4.7 Signatures Comparison

This paragraph illustrates the test to see if the Sendrier's *signature*, or if the replacement of puncturing operation with shortening (with the hull calculation after the shortening) used in equation 14 could improve the attack from the point of view of number of colliding indices. That is, it compares the discriminative capacity of different signatures. Or more clearly, the effectiveness to discriminate two equivalent codes from the others. The more colliding indices are found, the lower is the discriminative capacity.
We did tests (the corresponding code is in the Appendix) to see the quality

n	k	q	$hull\_dim$	mean_colliding_indices
22	7	2	1	8.00
			2	7.45
			3	6.50
			4	4.18
			5	3.35
			6	3.16
			7	2.76
		3	1	11.54
			2	10.10
			3	5.59
			4	4.23
			5	2.56
			6	2.52
			7	2.26
		5	1	14.80
			2	6.64
			3	2.13
			4	1.14
			5	1.02
			6	1.00
			7	1.00
		7	1	16.86
			2	6.76
			3	1.76
			4	1.04
			5	1.00
			6	1.00
			7	1.00

Table 4: Average number of colliding indices, for SSA and several code parameters. For each set, we have performed 100 experiments. For all the considered instances, we have used $n = 22$.

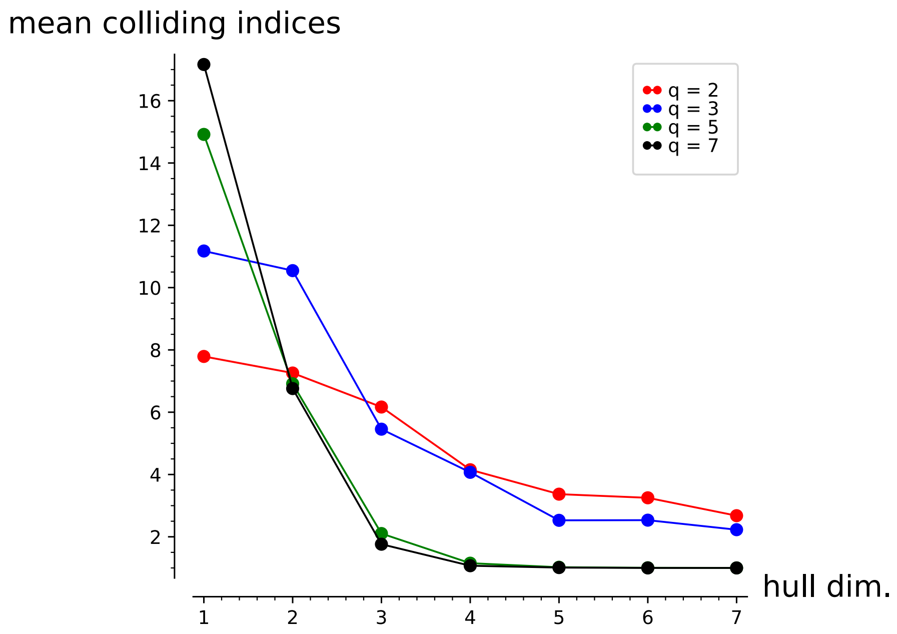


Figure 3: SSA colliding indices for different q as hull dimension increases

of the signatures just mentioned with different values of $hull\_dim$, n, k and q. An example varying the $hull\_dim$ is showed in the Figure 4, ignoring the average number of colliding indices, is recognizable for low hull values (2 and 3) that the difference of mean colliding indices between Sendrier's *signature* and others are very small and, for high values they are just overlapped.

4.8 Shortening multiple columns

The tests done so far have only removed one column at a time from the code. So it is not known whether removing multiple columns at the same time for comparing the weight distribution of shortened codes improves the quantity of colliding indices. That is, if the shortening on codes by removing more columns enhances the discrimination.
Unlike the original algorithm which aimed to find only one permutation index per time, this time it finds directly t indices, so t indicates the number of columns that will be removed. But the number of choices (combinations) of columns in this case is no longer n, but n choose t. That is, the number of combinations grows binomially with t. This is the compromise on computation time.

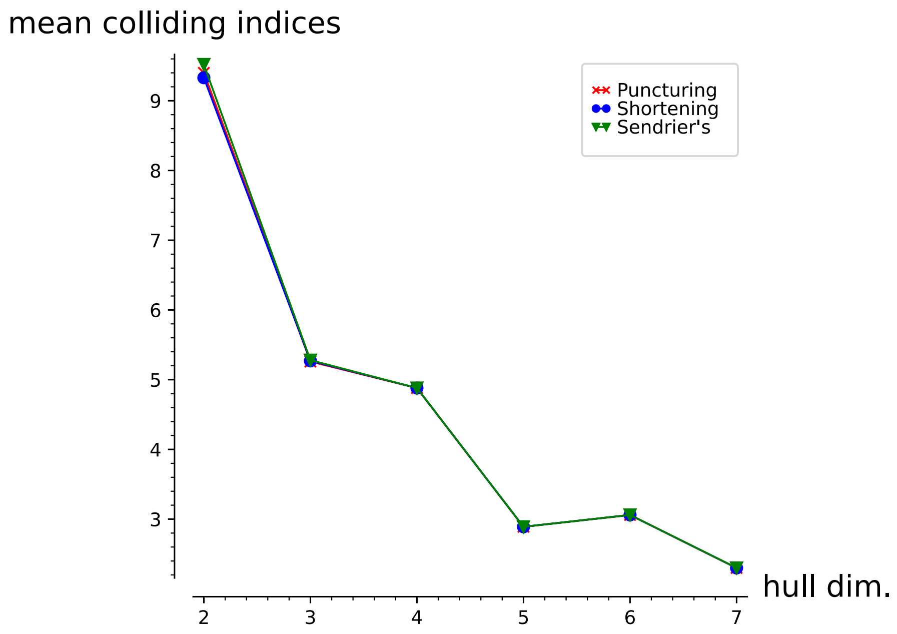


Figure 4: Graphical comparison between Sendrier's signature, Puncturing and Shortening by varying *hull_dim* with $n = 20$, $k = 10$ and $q = 3$

SageMath provides *Combinations* function to enumerate all combinations of t element in a range of n.
The following tables 5 and 6 shows the effect of shortening on 2, 3 or 4 columns at the same time. The tests (the corresponding code is in the Appendix) were done by varying values of n, k, q, *hull_dim* and t. Due to the complexity as the parameters t increase, the tests with $t = 4$ are done 10 times, while the others are 30 times. As one might imagine, the number of colliding indices does not depend on t, but only on $hull\_dim - t$.

5 A New signature function for SSA

As mentioned in section 4.7, the Sendrier's signature performs better than others, its strategy discriminates the codes very well. But all these techniques are based on the calculation of weight distribution of the code which has a time complexity of $O(q^d)$, with d the hull dimension. And when d and finite field cardinality q are small, it happens very often that the codes have same weight distribution. Hence, discrimination performance declines.

n	k	q	t	*hull_dim*	mean_colliding_indices
20	10	5	2	3	40.2
				4	5.1
				5	1.6
				6	1.0
				8	1.0
				9	1.0
				10	1.0
			3	4	216.2
				5	13.3
				6	2.1
				7	1.4
				8	1.0
				10	1.0
			4	5	1001.6
				6	110.3
				7	6.3
				8	1.6
				9	1.0
				10	1.0

Table 5: Colliding indices by shortening multiple columns with $n = 20$ and $k = 10$

Therefore, a new signature is designed, which is not based on the calculation of the weight distribution, but on the codewords, so the complexity is still exponential in d, i.e. $O(q^d)$. Since the weight of a codeword is obtained from counting the non zero element of codeword itself, the information used by Sendrier is only a subset of this new signature.
The idea is first described at a high level, then there is a detail on practical implementation. First consider two codes B and $B' = \pi(C)$ where $\pi \in S_n$ is a permutation. Let $\mathbf{B} \in \mathbb{F}_q^{d \times n}$ for B and $\mathbf{B}' = \mathbf{SGP}$ a generator for B'. Let $\mathcal{L} : \mathbb{F}_q^{d\times n} \mapsto \mathbb{F}_q^{q^d \times n}$ be the function that, on input some matrix $\mathbf{B}$, returns the matrix whose rows are all linear combinations of rows of $\mathbf{B}$. It is easy to see that this matrix possesses some useful invariance properties.

Proposition 2. *Let* $\mathbf{B} \in \mathbb{F}_q^{d\times n}$ *and* $\mathbf{B}' = \mathbf{SBP}$*, with* $\mathbf{S}$ *non singular and* $\mathbf{P}$ *a permutation matrix. Then*

$$\mathcal{L}(\mathbf{B}') = \mathbf{P}'\mathcal{L}(\mathbf{B})\mathbf{P},$$

n	k	q	t	*hull_dim*	mean_colliding_indices
15	7	3	2	3	63.00
				4	26.76
				5	25.56
				6	21.43
				7	74.53
			3	4	232.86
				5	129.76
				6	147.96
				7	135.30
			4	5	605.96
				6	414.56
				7	835.80
		5	2	3	31.76
				4	5.96
				5	1.80
				6	1.30
				7	2.0
			3	4	108.16
				5	26.23
				6	5.10
				7	5.86
			4	5	323.63
				6	80.63
				7	25.66

Table 6: Colliding indices by shortening multiple columns with $n = 15$ and $k = 7$

where $\mathbf{P}'$ *is a* $q^d \times q^d$ *permutation.*

The fact to consider the new permutation $\mathbf{P}'$ is due to the fact that the effect of $\mathbf{S}$ is that of modifying the bijection between $\mathbb{F}_q^d$ and B, i.e., the way information sequences are mapped into codewords. Starting from this consideration, consider a new signature function as follows:

1. compute $\mathcal{L}(\mathbf{B})$;
2. let M_i be the multiset entries of the i-th row of $(\mathbf{B})$;
3. sort the multisets $M_1, \cdots, M_{q^d}$ (e.g., using some lexicograph ordering);

4. let C_i be the i-th column of $\mathcal{L}(\mathbf{B})$;

5. sort the multisets $C_1, \cdots, C_n$;

6. set
$$W_i = \left(M_1, \cdots, M_{q^d}, C_1, \cdots, C_n\right).$$

In practice, the function can be implemented efficiently using hash functions; this way, so it's not necessary to work with multisets, which may be not very easy to handle (especially, when considering an ordering).

5.1 New signature implementation

As mentioned in the previous section, the new signature can be implemented with the help of the hash function. Which drastically reduces the size of the signature. This section illustrates the implementation of the algorithm:

1. In the first place, let $\mathbf{B}$ denotes the matrix on which to calculate the signature.

2. Generate all q^d vectors of finite field vector space $\mathbb{F}_q^d$, where q is finite field cardinality and d the dimension. In coding theory, these vectors represent the set of messages that can be encoded by the code. SageMath provides the *VectorSpace* function to create a vector space object, in which is possible to iterate all vector in the space.

3. Define a new matrix $\mathbf{A}$ of size $q^d \times n$ over $\mathbb{F}_q^{q^d \times n}$ to contain codewords.

4. For each vector $\mathbf{v}$ in $\mathbb{F}_q^d$ encode it with $\mathbf{B}$ obtaining the codeword $\mathbf{c} = \mathbf{vB}$, and insert into the matrix $\mathbf{A}$.

5. After filled up the matrix $\mathbf{A}$ with all q^d codewords, sort every single row and, for each sorted row compute the hash $h_i = hash(\mathbf{c}_i)$.

6. Sort all row hashes and compute the hash of them $h = hash(h_1, ..., h_{q^k})$.

7. Repeat the hash procedure also for transposed matrix $\mathbf{A}^\top$, i.e., sort each row of $\mathbf{A}^\top$ and compute $t_i = hash(\mathbf{m}_i)$, then compute the hash $t = hash(t_1, ..., t_n)$ of sorted hashes t_i

8. The signature is the pair (h, t).

As can be seen from the algorithm, unlike other signatures, it not only calculates all the codewords, it also orders them. The sorting and then hashing facilitates the comparison of the signature with the signature of other code during the execution of the SSA. In other words, sorting and hashing replaces the cumbersome comparing one by one the codewords of the two codes.
Therefore, this algorithm compares directly the codewords rather than just compare the number of weights. This makes the signature more precise.
Indeed, it may is the optimal solution in $O(q^d)$ complexity.

5.2 New signature & Sendrier's signature comparison

In Tables 7 and 8 are shown the mean colliding indices of two signatures under different parameters, the mean is calculated on 100 executions and the code length n is fixed to 20 (the corresponding code for the test is in the Appendix).
Analyzing the table, the value of code dimension k does not influence the average value of colliding indices since during the execution code's hull were considered which, as also happened in previous tests, as the hull dimension increases the colliding indices mean decreases and tends to 1. The same effect is also valid for the finite field cardinality parameter, i.e., as q increases the colliding indices mean decreases.
More clearly, the Figure 5 shows the trend of colliding indices of two signatures with another parameters ($n = 18$, $k = 7$ and $q = 5$), in this case, the improvement of new signature is not particularly big.
In conclusion, the most important thing is that the new signature has a noticeable reduction of average colliding indices compared to Sendrier's signature for $q > 2$, but they have same performance for $q = 2$.

n	k	Hull dimension	Avg. number of collisions	
			New	Old
20	4	2	8.33	8.33
		3	6.40	6.40
		4	4.63	4.63
20	6	2	8.60	8.60
		4	4.23	4.23
		6	3.50	3.50
20	8	2	7.06	7.06
		4	5.03	5.03
		6	3.63	3.63

Table 7: Comparison between old and new signature functions, for $q = 2$ and several values of n, k and hull dimension

n	k	Hull dimension	q	Avg. number of collisions		ratio
				New	Old	
20	4	2	3	7.50	11.10	0.67
			7	2.95	5.85	0.50
		3	3	2.75	5.45	0.50
			7	1.20	1.80	0.66
		4	3	1.60	3.90	0.41
			7	1	1	1
20	6	2	3	6.92	10.44	0.66
			7	3.54	7.18	0.49
		4	3	1.44	4.20	0.34
			7	1.02	1.06	0.96
		6	3	1.06	2.72	0.38
			7	1	1	1
20	8	2	3	6.80	10.52	0.64
			7	3.28	6.64	0.49
		4	3	1.68	4.34	0.38
			7	1.04	1.04	1
		6	3	1.04	3.08	0.33

Table 8: Table Compare New signature with Sendrier's for several values of n, k, hull dimension and q

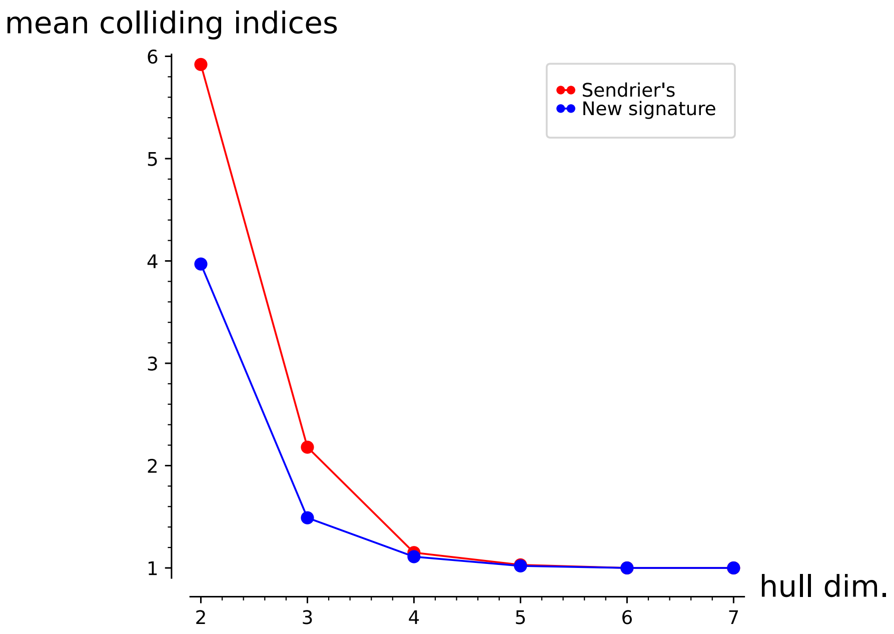


Figure 5: Graphical comparison between new signature and Sendrier's signature with $n = 18$, $k = 7$ and $q = 5$

6 Code Equivalence on non prime field

A finite field $\mathbb{F}_p$ can exist when the number of elements $q = p^m$ is a prime power, i.e., $q = p^m$ for a prime p and positive integer m. The most common and trivial case is when $m = 1$, so $q = m$, in this case, the finite field is composed by the set $\{0, 1, ..., p-1\}$. While $m > 1$, finite fields are built from a irreducible polynomial of degree m with coefficient over base field $\mathbb{F}_p$, and its called *extension field*. Elements of *extension field* are the set of all polynomial of degree $m - 1$ with coefficient on $\mathbb{F}_p$. Naturally, the *extension field* contains the base field $\mathbb{F}_p$ as subfield in fact, polynomials of degree 0, i.e., constants, are entries of base field $\mathbb{F}_p$.
Actually, every finite field $\mathbb{F}_q$ with prime power $q = p^m$ elements contains at least one subfield, and with the same ration, for codes, at least a *subfield subcode*.

6.1 Subfield Subcodes

The finite field $\mathbb{F}_{q=p^m}$ with $m = 1$ has a unique subfield, which corresponds to the field itself. So, it is not a case worth studying. But things get interesting for $m > 1$: for each divisor m' of m except the 1, there exists a smaller subfield $\mathbb{F}_b$ with $b = q^{m'}$. Correspondingly for codes, a subfield subcode whose entries are in a subfield $\mathbb{F}_b \subseteq \mathbb{F}_q$. The subcodes obviously, as the subfields, have fewer codewords. Therefore, attacking the subcodes should come with lower complexity. And also, subfield subcodes are isomorphism of the starting code, hence, they maintain many properties of the source code.
A way to construct subfield subcode is illustrated below:
Considering $\ell = \frac{m}{m'}$, $\mathbb{F}_q$ can be seen as a ℓ-dimensional vector space over the base field $\mathbb{F}_b$: Let $\varphi_b : \mathbb{F}_q \mapsto \mathbb{F}_b^\ell$ denote the map that brings any element of finite field $\mathbb{F}_q$ into a ℓ-length vector over the base field $\mathbb{F}_b$. Let $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ be a parity-check matrix for C, and denote by $\mathbf{H}' \in \mathbb{F}_b^{\ell(n-k)\times n} = \varphi_b(\mathbf{H})$, that is, the parity-check matrix obtained by replacing each element of $\mathbf{H}$ with the ℓ-length vector. Then, B is the code over $\mathbb{F}_b$ whose parity-check matrix is $\mathbf{H}'$. Since $\mathbf{H}'$ has $\ell(n-k)$ rows, the maximum dimension of B is

$$n - \ell(n-k) = n\big(1 - \ell(1-R)\big),$$

so that the maximum achievable rate is $R_\ell = 1 - \ell(1-R)$.
Notice that if the dimension of the starting code is too small, i.e., if $n-k$ is too large, it may happen that $\ell(n-k) > n$. In such a case, $\mathbf{H}'$ has more rows than columns and, with high probability, generates the whole space $\mathbb{F}_b^n$. In such cases, the subcode B would be empty.

6.2 A property of subfield subcodes

The reason to study the subfield subcodes comes from the fact that, by extracting the subcodes from two equivalent codes, i.e. which differ by a permutation, they are also equivalent. In mathematical form:

Theorem 1. *Let $q = p^m$ and $b = p^{m'}$, with $\ell = m/m'$. Let $C \subseteq \mathbb{F}_q^n$ and $C' = \pi(C)$, with π being a permutation. Let $B = \varphi_\ell(C)$ and $B' = \varphi_\ell(C')$. Then, $B' = \pi(B)$.*

Therefore, all algorithms for solving PEP can be applied to the subfield subcode instead of the initial code. With a hope of having a reduction in complexity.

6.3 Rank of a randomly generated matrix

Random matrices are fundamental for algorithms or protocols in a security application. As they provide maximum uncertainty, i.e., entropy, on the keys of cryptographic scheme, making it difficult to discover. Therefore studying random matrices and their properties provides greater consciousness of the limits and potential of the cryptographic scheme on which it is based.
In the case of the LESS digital signature scheme, the random matrix and its permuted form the public keys, and the permutation matrix connecting the two matrices need not be revealed.
This section talks about the probability of having a rank rk of the random matrix lower than its number of rows k, assuming that the matrices taken into consideration has entries over a finite field $\mathbb{F}_q$ and always have the number of rows k smaller than the number of columns n as they are considered as the generator matrix of a code.
The condition of having full rank, i.e. $rk = k$, is that all the k rows of the random matrix are linearly independent. Therefore starting from the first row, since it does not have to compare with any other rows it is certainly a linearly independent unless it is a null row (so the probability that this not happens is $1 - \frac{1}{q^n}$). At this point adding the second row, its condition of not being linearly dependent on the first is that it must not be a multiple of it. Since the entries are based on a finite field $\mathbb{F}_q$, for a codeword (row), there are only q multiples of it. So the probability that the second row is independent from the first is that extracting randomly a n-length vector based on $\mathbb{F}_q$ must not obtain a vector that is a multiple of the first row. In mathematical formula:

$$\Pr\{\mathbf{v}_2 \neq j\mathbf{v_1}; \forall j \in \{1, ..., q-1\}\} = 1 - \frac{q}{q^n} \tag{15}$$

Doing an induction for the i-th row, the constraint on this row is that it must not be a linear combination of the other $i-1$ remaining rows:

$$\Pr\{\mathbf{v}_i \neq \mathbf{w}; \forall \mathbf{w} \in \text{ span}(\mathbf{v}_1, ..., \mathbf{v}_{i-1})\} = 1 - \frac{q^{i-1}}{q^n} \tag{16}$$

Combining the constraints for each row, the probability of having a full-rank random matrix is as follows:

$$\begin{aligned}
\Pr\{rk = k\} &= \prod_{i=1}^{k} 1 - \frac{q^{i-1}}{q^n} \\
&\approx \prod_{i=1}^{k} e^{-\frac{q^{i-1}}{q^n}} \\
&= e^{-\sum_{i=1}^{k} \frac{q^{i-1}}{q^n}} \\
&= e^{-\sum_{i=1}^{k} q^{i-1-n}} \\
&= e^{-\sum_{i=1}^{k} \left(\frac{1}{q}\right)^{n+1-i}} \\
&= e^{-\sum_{i=n-k+1}^{n} \left(\frac{1}{q}\right)^{i}}.
\end{aligned} \tag{17}$$

We now consider the exponent of the above formula. We notice that

$$\sum_{i=n-k+1}^{n} \left(\frac{1}{q}\right)^{i} = \sum_{i=0}^{n} \left(\frac{1}{q}\right)^{i} - \sum_{i=0}^{n-k} \left(\frac{1}{q}\right)^{i}.$$

We now use the properties of the geometric series: for any $\alpha < 1$, it holds that $\sum_{i=0}^{m} \alpha^i = \frac{1-\alpha^{m+1}}{1-\alpha}$. So:

$$\sum_{i=0}^{n} \left(\frac{1}{q}\right)^{i} = \frac{1-(1/q)^{n+1}}{1-1/q},$$

$$\sum_{i=0}^{n-k} \left(\frac{1}{q}\right)^{i} = \frac{1-(1/q)^{n-k+1}}{1-1/q},$$

so that

$$\sum_{i=n-k+1}^{n} \left(\frac{1}{q}\right)^{i} = \sum_{i=0}^{n} \left(\frac{1}{q}\right)^{i} - \sum_{i=0}^{n-k} \left(\frac{1}{q}\right)^{i} = \frac{q^k - 1}{(q-1)q^n}.$$

Since $q^k \gg 1$, we approximate $\frac{q^k-1}{(q-1)q^n} \approx \frac{q^k}{(q-1)q^n} = \frac{1}{(q-1)q^{n-k}}$. We conclude by considering that $e^-x \approx 1 - x$ for small x; for large $n-k$, we have $\frac{1}{(q-1)q^{n-k}} \ll 1$, so that

$$e^{-\sum_{i=n-k+1}^{n} \left(\frac{1}{q}\right)^{i}} \approx e^{-\frac{1}{(q-1)q^{n-k}}} \approx 1 - \frac{1}{(q-1)q^{n-k}}.$$

Thus, the probability of not having a full-rank matrix in a square random generation is approximated as $1/q$. Furthermore, as q increases, or the rate R decreases from 1, the matrices tend to be full-rank (polynomially with q and exponentially with $1 - R$).

6.4 Hull of a randomly generated matrix

The dimension of the hull is very important in solving the PEP. In fact when the hull of the matrix has a high dimension, the problem is difficult to solve. While for the hull with small dimensions the problem is easy especially when the dimension can be considered a constant, In this case the solution of the problem becomes polynomial time.
Assuming the case of a random matrix which the number of rows is less than or equal to the number of columns $k \leq n$. The hull dimension of a matrix $\mathbf{G}$ can be easily obtained, without knowing its construction, by calculating the rank of the matrix $\mathbf{GG}^\top$.
There are two extreme cases: If the dimension of the hull is 0, then the rank of this matrix is $rank(\mathbf{GG}^\top) = k$. On the other hand, if the dimension of the hull is equal to k, then $rank(\mathbf{GG}^\top) = 0$. Hence a general formula of hull dimension d is $d = k - rank(\mathbf{GG}^\top)$. An analytical formula on the probability for the hull dimension had not been found during the study. But a series of experiments (the corresponding code is in the Appendix) are done with the achievement of the fact that the hull dimension is usually zero or unitary and rarely greater than 2. However, after various experiments on the occurrence of the dimension on 100000 random matrices, a expression for the hull dimension probability when it is not 0 can be represented like this:

$$\Pr\{d = x\} = \frac{1}{q^{\binom{x+1}{2}}} \tag{18}$$

Computing the above formula for growing x, we get

x	$\binom{x+1}{2}$
1	1
2	3
3	6
4	10
5	15
6	21
7	28

so that for $d = 0$ we can approximate

$$\Pr\{d = 0\} = 1 - \frac{1}{q} - \frac{1}{q^3} - \frac{1}{q^6} - \cdots \approx 1 - \frac{1}{q}$$

The results proving this are given by the tables 9, 10, 11 and 12, which reports the results of the SageMath simulation on 1000000 random matrices. As can be seen from the tables, when the rate deviates from 1, the formula works, while for rates close to 1, large hull dimensions are more often, but they still extinguish exponentially. This enlargement could be linked to the fact that, with rates near 1, the matrix under test is closest to square, so it has more chance to be not full-rank. Contrary to the intuition of having a high rate gives the code of matrix a small parity-check matrix and their intersection is small.
The Figure 6 shows the theoretical trend of random code's hull dimension for different values of q, as can be seen from it, the greater q is, the lower the probability of having high hull dimension. Except the case for $q = 2$, contrary to the simulation related to table 9, where the probability of having a *trivial* hull is less than the probability of having a hull with dimension 1.

d	occurrency	relative occurency	theorical relative occurency
0	427897	0.427897	0.358398
1	415366	0.415366	0.500000
2	138231	0.138231	0.125000
3	17313	0.017313	0.015625
4	1179	0.001179	0.000976
5	14	0.000014	0.000030

Table 9: Table hull dimension with $q = 2, k = 6, R = \frac{1}{2}$

d	occurrency	relative occurency	theorical relative occurency
0	639404	0.639404	0.628240
1	318626	0.318626	0.333333
2	40453	0.040453	0.037037
3	1499	0.001499	0.001371
4	18	0.000018	0.000016

Table 10: Table hull dimension with $q = 3, k = 6, R = \frac{1}{2}$

d	occurrency	relative occurency	theorical relative occurency
0	793476	0.793476	0.791935
1	198185	0.198185	0.200000
2	8270	0.008270	0.008000
3	69	0.000069	0.000064

Table 11: Table hull dimension with $q = 5, k = 6, R = \frac{1}{2}$

d	occurrency	relative occurency	theorical relative occurency
0	760497	0.760497	0.791935
1	229533	0.229533	0.200000
2	9579	0.009579	0.008000
3	390	0.000390	0.000064
4	1	0.000001	0.0000001

Table 12: Table hull dimension with $q = 5, k = 6, R = \frac{5}{6}$

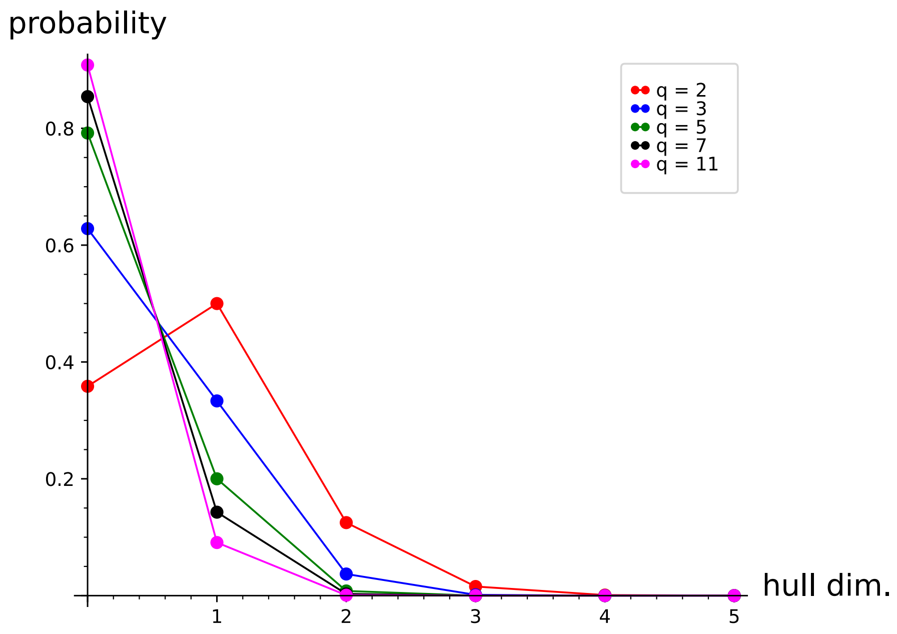


Figure 6: theoretical random code hull dimension for different q

6.5 PEP of randomly generated matrix resolution

PEP on random matrices are easy-to-solve problems. In fact, as seen in the SSA algorithm section, the complexity of this algorithm, if not refined, is exponential in dimension k of the code, however, with a small adjustment on the use of the code, that is, instead of considering the code itself, it considers its hull. At this point, since the matrix is random and, as already seen in section 6.4, these matrices usually have a very small hull dimension, i.e. the occurrences of high hull dimensions drop exponentially. So the SSA algorithm will generally only take q, q^2 or q^3 calculations.
On the other side, in the most frequent case of the zero-dimensional hull, the technique of section 2.6.1 can be applied, i.e. the reduction of the PEP to the GIP.

6.6 Hull of a subfield subcode

When the codes or equivalently the matrices are built on a finite field $q = p^m$ with $m \geq 1$, then there are certainly a subfield subcodes which will have the

codeword space smaller than the starting code, therefore studying properties of subfield subcodes could improve the attack. Since the permutation is inherited by the subfield subcodes then it makes one wonder if the hull of the subfield subcodes inherits the permutation of the starting code, that is, if the permutation is transitive through the calculation of the subfield subcodes and it's hull, and that's actually how it is. At first glance the idea of having to use the hull of the subcode is not thought of since one might think that the hull dimension of the subfield subcode inherits the hull size of the source code. The surprising thing that during the tests done on the subfield subcodes is exalted is that the dimension of their hull has nothing to do with the hull of the starting code. Furthermore it tends to be similar to a completely random code or matrix hull, namely, its dimensions appear with the same probabilities of a random matrix hull.
The figure 7 and 8 (the corresponding code that originate the Figures is in the Appendix) shows both the probabilities of the hull dimension of the subfield subcode of a code with specific hull dimension d and the hull dimension of a random matrix with entries on the same subfield and the same number of rows and columns of subcode's matrix. As can be seen from the figure, the two curves are almost overlapped. Except the case which the source code has a hull dimension equal to the dimension k of the code in which, it has a small peak at 2. But in any case, for high values of hull dimension they have negligible probability.

6.7 Simple GIP solver

To test and solve the PEP in the case of a code with zero hull dimension and to have an auxiliary solver for the new algorithm of the next section, a simple GIP solver with a complexity of $O(n^2)$ has also been written, this algorithm could fail (it might spit out an incorrect permutation matrix) which is more frequent when q is small.
A description of the solver is the following:
As said in section 2.6.1, the GIP solver finds the permutation matrix between two adjacency matrices, therefore given two matrices of the two equivalent codes, they must first be transformed into an adjacency matrix. So once done, consider the two adjacency matrices.
Remind that the two adjacency matrices are square matrices, now looking at the GIP: The transformation or isomorphism of the graph does a very simple thing. Considering a adjacency matrix, the multiplication with the matrix $\mathbf{P}$ at the right permutes the columns. Subsequently the multiplication on the $\mathbf{P}^\top$ at the left permutes the rows, that is, first a permutation of columns

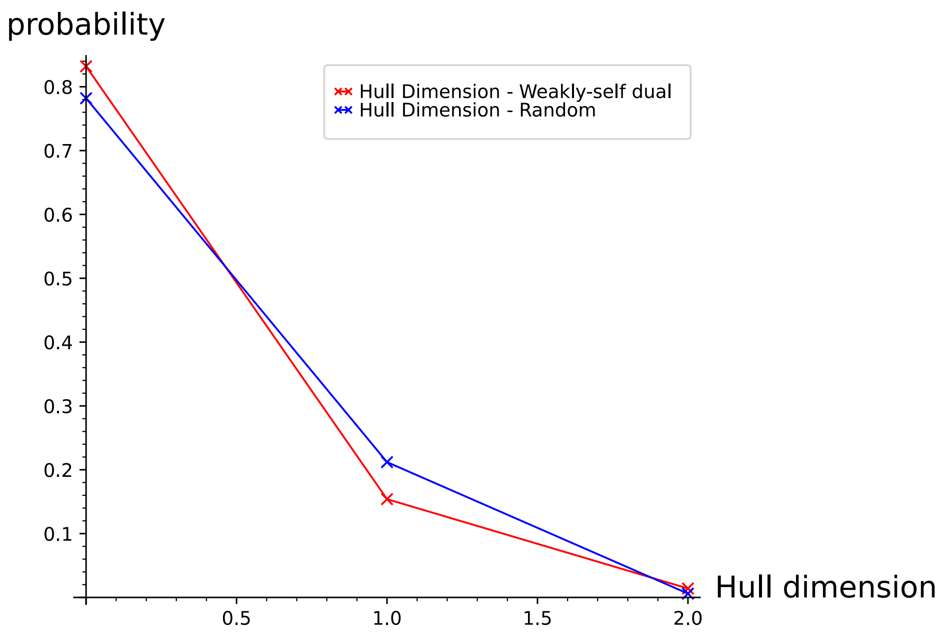


Figure 7: Hull prob. for code with $q = 5$, $\ell = 2$, $n = 20$, $k = 15$ and $d = 4$

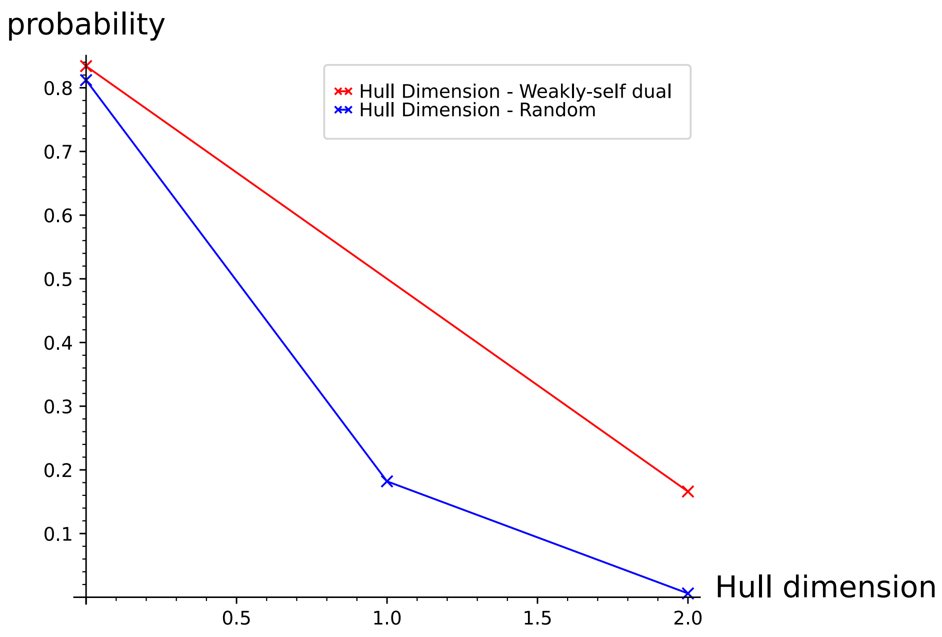


Figure 8: Hull prob. for code with $q = 5$, $\ell = 2$, $n = 20$, $k = 15$ and $d = 5$

and then a permutation of rows is applied to this matrix.
The solver first does a very tricky thing which is to sort all the rows of the two adjacency matrices. Since there is no change of basis in this transformation, the reordering of all the rows of the two matrices causes the permutation of the columns to be elided. The side effect of sorting which causes the failure is that multiple rows with same arrangement and values may come out, and this effect happens often when q is small, e.g. 2, 3 and 5.
After, it is enough to compare the two matrices row by row and the matrix $\mathbf{P}$ can be obtained again.

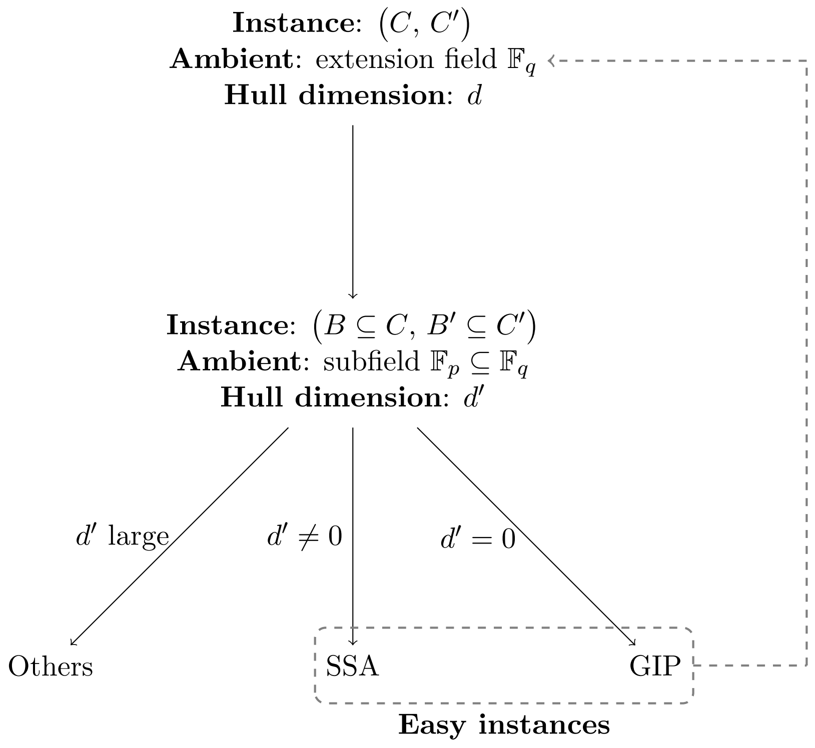
6.8 A New PEP Solver for codes over non prime field

In this section, a new algorithm for solving PEP is proposed, which works in the case of code defined on a finite field $\mathbb{F}_q$ with q an extension field, i.e. $q = p^m$ with $m > 1$, and having a high predefined hull dimension d. By aggregating the techniques and observations discussed above, for example the new signature, the reduction to the GIP with the solver, the use of the code hull and the use of the subfield subcode.
The strategy of this algorithm is to avoid directly calculating the signature of the code with high hull dimension, but instead exploiting its properties. Considering the two codes (matrices) of the PEP, first of all the new algorithm directly calculates the subfield subcodes on codes, there are three cases for the starting codes, the rate $R < \frac{1}{2}$, $R > \frac{1}{2}$ or $R = \frac{1}{2}$. special cases, including $R = \frac{1}{2}$, for them there are methods to get around, an example is to puncture or shorten a column to the code to deviate from $R = \frac{1}{2}$. The details will not be discussed here. When $R < \frac{1}{2}$, the dual code must be considered as the candidate to calculate the subfield subcode. Moreover for same reason, the choice of parameter ℓ must ensure that $\ell(n-k) < n$, otherwise it will very likely to obtain a subfield subcode that has a parity matrix greater than the length n of the code, i.e. a code with a void generator matrix is an empty code. So ultimately, the algorithm works for $R < \frac{1}{\ell}$ or $R > 1 - \frac{1}{\ell}$.

The first step, i.e., considering the subfield subcode, already reduces significantly the complexity of the code to be operated.
But as mentioned in section 6.6, the hull of the subfield subcode has constant dimension therefore for a further improvement on the attack, code's hull is considered. At this point, referring to section 6.5, if the dimension of the hull is zero, the algorithm ends with the transformation into GIP and retrieve directly $\mathbf{P}$ using the solver, otherwise it apply the SSA with the new

signature.
Finally if the case is very unlucky, i.e. the dimension of the hull of subfield subcode is very large then, this new algorithm is absolutely an unsuitable solution to solve it.

The description of the algorithm just mentioned is summarized in the following drawing:


Instance: (C, C')
Ambient: extension field $\mathbb{F}_q$
Hull dimension: d
Instance: $(B \subseteq C, B' \subseteq C')$
Ambient: subfield $\mathbb{F}_p \subseteq \mathbb{F}_q$
Hull dimension: d'
d' large
$d' \neq 0$
$d' = 0$
Others
SSA
GIP
Easy instances

6.9 GIP fullcode

In the literature a method has been proposed for resolving the PEP by reducing it to GIP with non *trivial* code's hull. The technique called *GIP fullcode* is very simple:
Since the reduction to GIP requires that the square matrix $\mathbf{GG}^\top$ must be full rank, otherwise matrix inversion cannot be applied. Since this rank depends on the dimension of the hull, i.e. $rank(\mathbf{GG}^\top) = k - d$, where k is the dimension of the code and d the dimension of it's hull. One can shortening d columns of the matrix to make the hull of the punctuated code, with a

$(k-d) \times (n-d)$ dimensional matrix, become *trivial.* By shortening them randomly, it is very likely that two non-equivalent codes are obtained, so this procedure must be repeated by $\binom{n}{d}$ combinations until a combination, of the d indices shortened that lead to two equivalent codes, is found.
After a series of simplifications on the complexity of this procedure, one obtains that for a constant d, the complexity of GIP fullcode is like $O(n^{d+2.3})$, on the other side, for a hull dimension linear with code length n, that is, $d = \alpha n$, then the complexity will be dominated by $O(n^n)$. For simplicity, we consider the difficult as $O(n^{d+2.3})$.

6.10 New Solver complexity estimation

In this section the performances, both in formula and in numbers, of the different PEP resolution algorithms are presented.
The tables 14 and 15 shows the Big O argument in numerical form. As can be seen from tables, both GIP fullcode and Sendrier's SSA are very sensitive to the dimension d of the code hull, in other words, they are exponential in d. While the new solver, despite some constraints on the parameters, due to the randomness of the subfield subcode's hull dimension, it almost always works with very small hull dimensions. When hull dimension d is zero it uses the GIP solver otherwise it launches the SSA. Hence, this new solver can be considered polynomial time.
The difference in complexity between those algorithms as the hull dimension varies are presented in Table 13 (the corresponding code that compute the complexities is in the Appendix), for large hull dimensions, both SSA and GIP fullcode works in exponential time while the new solver is still polynomial.
Moreover, the Figure 9 shows a visual comparison of the different algorithms. Since the y axis is in the logarithmic scale, an exponential growth is represented as a straight line. So, as can be seen from the Figure, both GIP fullcode and SSA have an exponential trend as hull dimension d increases. While the New solver remains polynomial.

The complexity for GIP Fullcode is:

$$O(n^{d+2.3}) = O(2^{(d+2.3)\log_2 n}) \tag{19}$$

And for Sendrier's SSA:

$$O(n^3 + \log(n)q^d) \tag{20}$$

	Hull dimension	Cost
SSA	Small	Polynomial
SSA	High	Exponential
GIP fullcode	Small	Polynomial
GIP fullcode	High	Exponential
New solver	Small	Polynomial
New solver	High	

Table 13: Table of SSA, GIP fullcode and new solver complexity

Finally, the new solver:

$$O(\mathrm{pr}\{d=0\} \cdot n^2 + \sum_{i=1}^{n/2} \Pr\{d=i\} \cdot (n^3 + \log(n) \cdot (p^{m'})^d)) \tag{21}$$

n	k	q	d	GIP fullcode	SSA Sendrier	ℓ	Our new solver
20	3	2^2	1	14.26	12.96	2	11.24
			3	22.90	12.99		11.24
		2^4		22.90	14.30		9.78
				22.90	14.30	4	9.78
	6			22.90	14.30	2	9.78
			5	31.54	21.58		9.78
		3^2		31.54	17.48		10.29
		3^4		31.54	33.27		8.95
	8	5^2	8	44.51	38.73		9.46
		7^2		44.51	46.49		9.12

Table 14: GIP vs Sendrier SSA vs New solver. All time complexities are expressed in $\log_2$ units

6.11 Frobenius endomorphism

Given a finite field $\mathbb{F}_q$, with $q = p^m$, there is an endomorphism, precisely, an automorphism $\mathcal{F}_p : \mathbb{F}_q \mapsto \mathbb{F}_q$ called Frobenius endomorphism, that is, it maps an entry of finite field into another entry with the following expression:

$$\mathcal{F}_p(x) = x^p \tag{22}$$

n	k	q	d	GIP full code	SSA Sendrier	ℓ	Our new solver
30	6	2^4	5	35.82	21.78	2	11.31
		2^8		35.82	41.77	4	9.98
		3^4		35.82	33.47	2	10.26
		3^8		35.82	65.16	4	9.82
		5^8		35.82	94.64	4	9.81
		7^4		35.82	57.91	2	9.83
		7^4		35.82	57.91	4	9.83
	10	2^4	5	35.82	21.78	2	11.31
		3^4	10	60.35	65.16		10.26
		5^4	10	60.35	94.64		9.88
		7^4	8	50.54	91.60		9.83
40	15	2^8	10	65.46	81.88	2	10.85
		3^4		65.46	65.28		11.21
		3^8		65.46	128.68		10.65
		5^8		65.46	187.64		10.64

Table 15: GIP vs Sendrier SSA vs New solver. All time complexities are expressed in $\log_2$ units

The transformation applied on a code's generator matrix $\mathbf{G}$ is invariant to permutation, i.e., let π a permutation:

$$\mathbf{G}' = \pi(\mathbf{G}) \Rightarrow \mathcal{F}_p(\mathbf{G}') = \pi(\mathcal{F}_p(\mathbf{G})) \tag{23}$$

So, it may be useful for the attack since it is only a remapping of the elements, it certainly does not make the situation worse.

6.12 A More versatile subcode exploiting the Frobenius endomorphism

In section 6.8, a PEP solver for codes defined over a extension field is presented. This solver has constraints on the structure of the code due to the existence of the subfield subcode, i.e. on the maximum rate R that the code can assume for the chosen ℓ, it must be $R < \frac{1}{\ell}$ or $R > 1 - \frac{1}{\ell}$, in other hand, the possible values of ℓ are bound by the power m of the prime number p of the field cardinality $q = p^m$. So, some choices of the rate R and the value of m can prohibit the use of this solver.
This section presents a more versatile subcode that is applicable with any power $m > 1$ and every rate R except $R = \frac{1}{2}$. The idea is as follows:

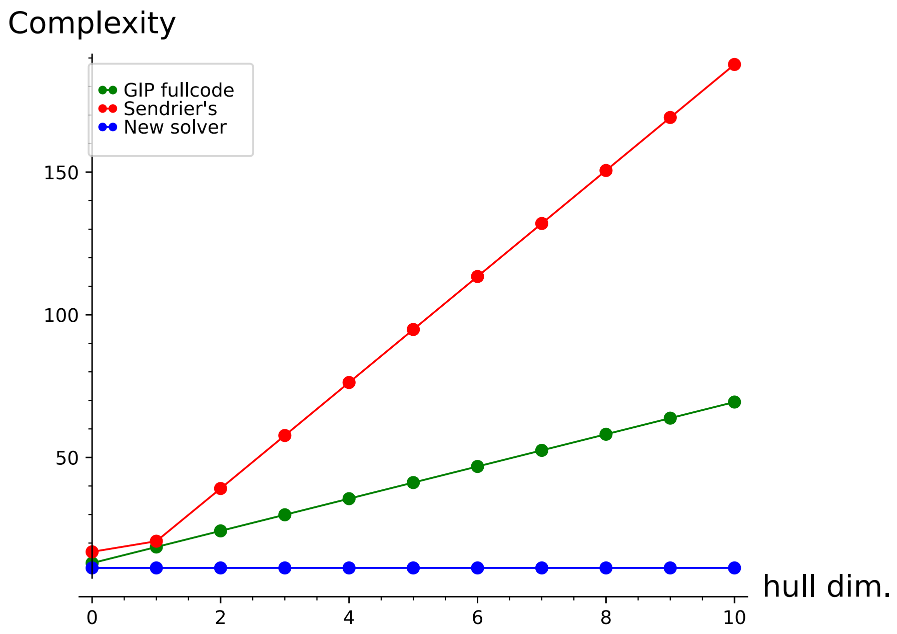


Figure 9: Complexity (log2) comparison between GIP fullcode, Sendrier's SSA and New solver as the code's hull dimension increases. Fixing $n = 50$, $k = 20$ and $q = 5^8$

Assuming that the code rate R is greater than $\frac{1}{2}$. First compute the Frobenius endomorphism $\mathcal{F}_p(\mathbf{G})$ of the code generator matrix $\mathbf{G}$, then intersect $\mathcal{F}_p(\mathbf{G})$ with $\mathbf{G}$. The new subcode to consider is the intersection, it's dimension is at least one, and preserves the permutation.
For the cases of codes with rate $R > \frac{1}{2}$, just consider the dual code, which will have the complementary rate. On the other hand, it is also easy to see that, when the defined finite field is prime, i.e., the finite field is not an extension field, then the Frobenius automorphism will return the code itself, so the intersection does not bring any reduction.
By doing tests with SageMath, it was noticed that the dimension of the intersection is almost 1, while high dimensions rarely appear, that is, without an analytical expression, the probability of having high dimensions decreases exponentially.
Once the such intersection has been calculated, since it already has a very small dimension, to solve the subcode one can use the solver for GIP if the hull is null, otherwise SSA.

6.13 Decision on the equivalence between two codes through reduction in GIP

The solver proposed in section 6.7 was a search-type solver, and could be integrated into the GIP solver of section 6.8. But due to its poor resolution nature it often fails, that is, after a reordering of the row elements on one of the two adjacency matrices, it is likely that multiple rows becomes identical, especially for small q. Therefore it brings an ambiguity in the choice of permutation.

On the other hand, a suitable search-type solver python code that manages to solve the isomorphism has not been found. But a decision-type solver has been found, that is, this type of solver says yes if the two graphs are isomorphic, otherwise it says no. In fact, the *is_isomorphic* function of the *networkx* library performs this type of resolution. With a few settings, the function becomes suitable for solving the decision on GIP with adjacency matrices obtained from the reduction of the PEP.

Even if the *is_isomorphic* function does not return the permutation matrix that links the two adjacency matrices, by doing experiments using this algorithm, the possibility of the resolution of the PEP through the reduction into GIP can be assured.

Below, in the table 16, the discriminating capacity and execution times of the function by giving it either two adjacency matrices obtained from two equivalent codes or from two random codes are presented, obviously the codes are with null hull.

Considering that the two input matrices to the *is_isomorphic* function are obtained by transforming the code's generator matrices with null hull into an adjacency matrix. In this regard, the label equiv false represents the number of cases out of a total of 500 that, the *is_isomorphic* function declared that the two adjacency matrices are not isomorphic when in fact they were obtained from two equivalent codes. While the label random true represents the number of cases that, the function declared that the two adjacency matrices are isomorphic when these matrices were obtained from two random codes. Finally, the labels *euiqv time* and *random time* indicates the average execution time, expressed in seconds, of the function when two adjacency matrices from two equivalent codes and two random codes are passed respectively.

As can be seen from the table, The number of true negatives is always 0 so it gives an indication that the function works, i.e., the function recognizes the equivalent code by having their transformed adjacency matrices. While false positives are exalted only when n is small, i.e. in the case $n = 10$ in

the table, and for a q equal to 2, which are more when the rate is far from a $\frac{1}{2}$ and fewer when the rate is $\frac{1}{2}$. Furthermore, the calculation time of the function both for equivalent code and for random code does not depend on the code rate R, in fact these matrices are transformed into adjacency matrices which has dimension $n \times n$ therefore, given a fixed n, the dimension k does not affect execution time. On the other hand, if n and k are fixed and only the finite field $\mathbb{F}_q$ is varied, the execution time grows as the cardinality q increases.
Looking from afar, one notice that the execution time for equivalent codes grows considerably as n increases but does not grow much for random codes. This situation is understandable since it is enough to find an edge of the graph that cannot be obtained after a permutation, then one can immediately state that they are not isomorphic. While many parts of the adjacency matrix must be tested to be sure that they are isomorphic, a rough estimate is that the complexity grows as n^2 which represents the number of elements in the matrix.

For the complexity of the latter, a further experiment was done by setting $k = \frac{n}{2}$, which means rate $R = \frac{1}{2}$, and finite field $\mathbb{F}_q$ fixed at $q = 2^8$. The results are reported in table 17 and as graphic in Figure 10 (in the Figure, the computational complexities of isometry between random codes are omitted). Looking at the table or chart, as n increases the complexity of checking two isomorphic matrices grows polynomially, while the time spent on checking two random matrices doesn't change much.

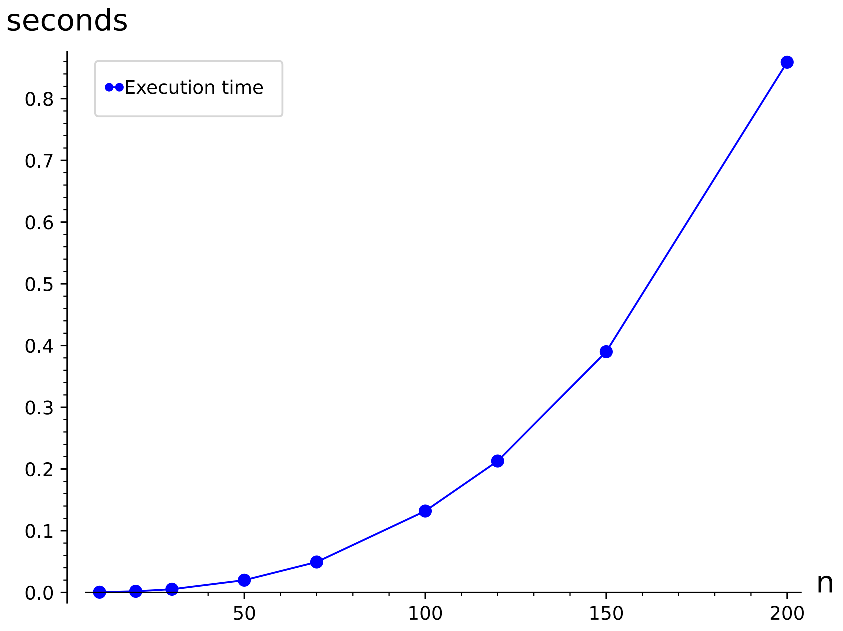


Figure 10: Time complexity of function *is_isomorphic* with growing n

7 Conclusion

With this thesis work, a new method to solve the case of Permutation Equivalence Problem of Code Equivalence Problem has been presented. The technique is polynomial time, and works for a certain instance of the problem. Like all other problems, there are parameter choices that are very vulnerable. Although the new method only breaks a narrow instance of the problem, we have demonstrated that the problem is still difficult if the code structure is chosen carefully.

The discriminators, i.e. the signatures, used in SSA for discriminating a code with another non equivalent one have always exploited the computation of the weight distribution. In this thesis, we have improved the signature while maintaining the computational complexity of $O(q^k)$, i.e., instead of calculating the weight distribution, which requires calculating all codewords, we directly considered this set of codewords as signature. Furthermore, to reduce the signature size, the hash of the sorted set is taken. The improvement about the effectiveness is theoretical, but tests developed in section 5.2 shows that it has really a significant improvement.

Subsequently, in the most substantial part of the thesis, we have studied a new method to transform PEP instances into new PEP instances for codes defined on a extension field, i.e., non-prime field. In particular, when the initial code is a *self dual* code, that is, with a high dimensional hull which in the literature is considered a safe condition for existing algorithms, the final code has a small hull with very high probability, similar to a hull of a completely random code. This permits the use of efficient algorithms.

The idea of the transformation is to consider the subfield subcodes, which maintains the structure of the permutation. Then if the latter code has a null hull we solve it with a GIP solver after a reduction to GIP (these GIP algorithms are polynomial time), otherwise we proceed with SSA which at this point, with very high probability, ends in polynomial time. Our results show that, for these instances of PEP previously thought to be difficult, it is now possible to use an efficient solver, which on average has a polynomial time complexity. The Figure 10 of the 6.10 section has illustrated, theoretically, the estimation of the complexity of the existing algorithms with the new solver. In short, the PEP is easy (polynomial time), when the codes are defined over a non-prime field.

For this reason, we strongly recommend to avoid using codes with the aforementioned structures for cryptographic applications, because they are easy to attack.

To conclude, the case of Linear Equivalence Problem has not yet been studied in this work. But with the research done in the thesis, the attack could also be extended to the case of LEP.

n	k	q	equiv false	random true	equiv time	random time
10	2	2	0	41	2.172E-04	3.008E-05
		7		0	3.670E-04	1.432E-05
		16			4.116E-04	1.772E-05
		256			5.052E-04	5.126E-05
		65536			6.086E-04	7.722E-05
	5	2	0	5	2.699E-04	1.638E-05
		7		0	3.537E-04	1.316E-05
		16			4.064E-04	1.620E-05
		256			4.788E-04	5.528E-05
		65536			6.142E-04	8.068E-05
	8	2	0	52	2.212E-04	3.384E-05
		7		0	3.554E-04	1.298E-05
		16			4.045E-04	1.624E-05
		256			4.754E-04	5.338E-05
		65536			4.863E-04	7.946E-05
20	5	2	0	0	1.291E-03	1.908E-05
		7			1.386E-03	1.808E-05
		16			1.730E-03	1.896E-05
		256			2.378E-03	6.204E-05
		65536			2.516E-03	1.977E-04
	15	2	0	0	1.180E-03	2.116E-05
		7			1.405E-03	2.088E-05
		16			1.747E-03	2.046E-05
		256			2.418E-03	7.838E-05
		65536			2.566E-03	2.002E-04
30	5	2	0	0	2.554E-03	3.188E-05
		7			3.286E-03	2.534E-05
		16			4.237E-03	2.624E-05
		256			6.728E-03	6.228E-05
		65536			7.065E-03	3.064E-04
	15	2	0	0	2.222E-03	2.494E-05
		7			3.278E-03	2.414E-05
		16			4.186E-03	2.432E-05
		256			6.686E-03	7.774E-05
		65536			7.045E-03	3.146E-04
	25	2	0	0	4.055E-03	2.488E-05
		7			3.260E-03	2.534E-05
		16			4.212E-03	2.412E-05
		256			6.688E-03	7.576E-05
		65536			7.059E-03	3.214E-04

Table 16: True negative, false positive and execution time of *is_isomorphic* function

n	k	q	equiv time	random time
10	5	256	4.088E-04	5.452E-05
20	10	256	1.903E-03	6.564E-05
30	15	256	5.222E-03	6.230E-05
50	25	256	1.987E-02	5.986E-05
70	35	256	4.932E-02	6.648E-05
100	50	256	1.319E-01	8.348E-05
120	60	256	2.129E-01	7.776E-05
150	75	256	3.907E-01	8.986E-05
200	100	256	8.591E-01	1.467E-04

Table 17: Time complexity of function *is_isomorphic* with growing n

References

[1] László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697, 2016.

[2] Magali Bardet, Ayoub Otmani, and Mohamed Saeed-Taha. Permutation code equivalence is not harder than graph isomorphism when hulls are trivial. In *2019 IEEE International Symposium on Information Theory (ISIT)*, pages 2464–2468. IEEE, 2019.

[3] Alessandro Barenghi, Jean-François Biasse, Tran Ngo, Edoardo Persichetti, and Paolo Santini. Advanced signature functionalities from the code equivalence problem. *International Journal of Computer Mathematics: Computer Systems Theory*, 7(2):112–128, 2022.

[4] Alessandro Barenghi, Jean-François Biasse, Edoardo Persichetti, and Paolo Santini. LESS-FM: fine-tuning signatures from the code equivalence problem. In *Post-Quantum Cryptography: 12th International Workshop, PQCrypto 2021, Daejeon, South Korea, July 20–22, 2021, Proceedings 12*, pages 23–43. Springer, 2021.

[5] Jean-François Biasse, Giacomo Micheli, Edoardo Persichetti, and Paolo Santini. LESS is more: code-based signatures without syndromes. In *Progress in Cryptology-AFRICACRYPT 2020: 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20–22, 2020, Proceedings 12*, pages 45–65. Springer, 2020.

[6] Tung Chou, Edoardo Persichetti, and Paolo Santini. On Linear Equivalence, Canonical Forms, and Digital Signatures. 2023. `https://eprint.iacr.org/2023/1533`.

[7] Edoardo Persichetti and Paolo Santini. A new formulation of the linear equivalence problem and shorter LESS signatures. *Cryptology ePrint Archive*, 2023.

[8] Erez Petrank and Ron M Roth. Is code equivalence easy to decide? *IEEE Transactions on Information Theory*, 43(5):1602–1604, 1997.

[9] Nicolas Sendrier. On the dimension of the hull. *SIAM Journal on Discrete Mathematics*, 10(2):282–293, 1997.

[10] Nicolas Sendrier. Finding the permutation between equivalent linear codes: the Support Splitting Algorithm. *IEEE Transactions on Information Theory*, 46(4):1193–1203, 2000.

Listings

8 Codes explanation

This appendix reports the code we have used for all the simulations. The meaning of each reported code is reported below.

1. This Code is used as a library, it contains functions useful for the following codes, including the generation of weakly self dual code, the generation of two equivalent code, the hull computation, the Sendrier's signature, the New signature etc..

2. Contains the performance simulation as *hull_dim* increases on SSA using different method to sign the code, as Puncturing, Shortening and Sendrier's signature. After the simulation, it plots a graph showing the mean colliding indices of the signatures.

3. Contains the simulation about the trend of mean colliding indices of SSA as *hull_dim* increases for different values of q. Then it plots a graph showing the curve for each q.

4. Tests if the operation of shortening multiple columns instead of one columns during the SSA increases the performance in terms of mean colliding indices.

5. Compares the performance between the Sendrier's signature and our new signature for SSA, and plots the result for increasing *hull_dim*.

6. Compares the trend of probabilities of having a specific value of *hull_dim* between a subfield-subcode obtained from a weakly self dual code and a

random generated code with same field and dimensions of the subfield-subcode. In the end, the script plots the results.

7. This script records the occurrency of hull dimensions on a large quantity of randomly generated matrices.

8. It confirms the effectiveness of the reduction to GIP from PEP using *is_isomorphic* function powered by Networkx library, that is, to verify if the adjancency matrices obtained by reducing the two equivalent codes are isomorphic and in other hand, if the adjancency matrices obtained by reducing two random codes are not isomorphic.

9. This Code is a *Proof of concept* of PEP resolution for codes defined on a non-prime finite field. It includes the computation of subcodes, the reduction to GIP if the hull is null and the SSA with new signature.

10. This script performs a numerical calculation of the complexities of the different algorithms for solving the PEP such as GIP fullcode, Sendrier's SSA and New solver. Then plots the difference between the New solver and other algorithms for increasing *hull_dim*.

Code Snippet 1

Listing 1: code_utils

```
import time
import numpy
import networkx as nx
import matplotlib.pyplot as plt

''' ### Declarations:
hull(C)
generate_permutation(Fq,n)
generate_change_of_basis(Fq,k)
code_shortening(C,columns)
generate_weakly_self_dual(Fq,n,k,hull_dimension)
Sendrier_signature(Fq,C,i)
get_H_subcode(H, ell, Fq, Fp)
new_signature_function(Fq,B)
build_adjacency(G)
GIP(G,G_prime,P)
generate_weakly_self_dual_code(Fq,n,k,k_hull) #min dim hull 1
genTwoEquiv(Fq,n,k,hull_dim)
frob_intersec(G)
FqToIntMatrix(G)
GIP_is_isomorphic(AdjancencyMatrix1,adjM2)
'''

#Get the intersected code with entry over Fq between C1 and C2.
def code_intersection(C1,C2,Fq):
    #get the generator matrix of two codes
    G1 = C1.generator_matrix();
    G2 = C2.generator_matrix();
    n = C1.length();
    #declare a n-dimensional vector space over Fq.
    Vn = VectorSpace(Fq,n);
    #consider V1 and V2 as subspaces created
    #from basis defined by G1 and G2.
    V1 = Vn.subspace(G1);
    V2 = Vn.subspace(G2);
    #get the intersection
    V12 = V1.intersection(V2);
    #retrieve the basis of intersected space,
    #that is, the generator matrix of intersected code.
    G12 = V12.basis();
    return(G12);

#################################

#Compute the hull of a code; return the generator matrix of hull.
def hull(C):
    #convert the generator matrix to code.
    C = LinearCode(C)
    #retrieve the finite field on which we work
    Fq = C.generator_matrix()[0,0].parent()
    C_dual = C.dual_code(); #get the dual code
    #get the intersection between code and itself,
    #that is, the hull
    G_hull = code_intersection(C,C_dual,Fq);
    return matrix(G_hull) #return a matrix

#################################

#Generate a n * n permutation matrix.
def generate_permutation(Fq,n):
    #consider a random
    P = Permutations(n).random_element();
    #convert the permutation to permutation matrix
    P_matrix = matrix(Fq,P.to_matrix());
    return P_matrix;

#################################

#Generate a k * k change of basis matrix.
def generate_change_of_basis(Fq,k):
    rank_S = 0;
    #regenerate until we get a full rank matrix
    while rank_S < k:
        S = random_matrix(Fq,k,k);
        rank_S = rank(S);

    return S;
#################################

#parameters: C: code to short, columns: list of columns to remove.
def code_shortening(C,columns):
    #convert the generator matrix to code.
    C = LinearCode(C)
    #consider the parity check matrix
    H = C.parity_check_matrix();
    #create a new zero matrix with same number of columns of H
    #and number of rows as many columns to remove,
    #then for each row, add a 1 to the position indicated by parameter columns.
    A = zero_matrix(len(columns),H.ncols());
    for i in range(len(columns)):
        A[i,columns[i]]=1; #Add rows to parity check matrix like [1, 0, 0]; [0, 1, 0]

    #merge the new matrix A at the bottom of H
    Hext = block_matrix([[H],[A]]);
    Hext_code = LinearCode(Hext);
    #get the dual code of the extended H built before.
    Gext = Hext_code.parity_check_matrix();
    #the Gext so obtained has in the columns indicated by parameter'columns'
    #all zeros, so we delete these columns.
    G_shorted = Gext.delete_columns(columns);
    return (G_shorted);

#################################

#Generate a weakly self dual code with desired hull dimension.
def generate_weakly_self_dual(Fq,n,k,hull_dimension):

    #n = 60; #code length
    #k = 30; #code dimension
    #q = 127; #finite field cardinality
    #hull_dimension = 7;  #desired dimension for the hull
    #Fq = GF(q); #define finite field

    ###Start with a self orthogonal codeword
    s = 1; #s is the inner product of the codeword with itself
    #while : repeat until the codeword is not self orthogonal.
    while s !=0:
        a = random_vector(Fq,n); #random vector with length n
        #Compute inner product <a ; a>
        s = 0;
        for i in range(n):
            s+= (a[i])^2;
        #Check if the number of zeros is smaller than n
        nzero = a.list().count(0);
        if nzero == n:
            s = 1;

    #convert to matrix
    nG = matrix(a);

    hd = 1; #the initial Hull dimension is 1
    #construct the code until we get to the desired hull dimension.
    while hd < hull_dimension:
        #Pick codeword from the dual, until you find a self-orthogonal word
        C = codes.LinearCode(nG);
        Cd = C.dual_code();
        s = 1;
        while (s !=0):
            b = Cd.random_element();
            s = 0;
            for i in range(n):
                s += (b[i])^2;
            nzero = b.list().count(0);
            if nzero == n:
                s = 1;

        #Enrich nG with the new codeword
        nG_new = block_matrix([[nG],[matrix(b)]],subdivide=False);
        C_hull = hull(nG_new);
        hd_new = matrix(C_hull).nrows();
        #if hd_new > hd:
        hd = hd_new
        nG = nG_new
        #commented, still work.
        #nG = matrix(C_hull);
        #print("Reached dimension is = "+str(hd));
        #print("done")
    obtained_k = hd;
    #now fill up the matrix with non self-orthogonal rows.
    while obtained_k < k:
        #print(obtained_k, k);
        #Continue picking codewords
        C = codes.LinearCode(nG);
        Cd = C.dual_code();
        s = 0;
        attempts = 20 #after 20 cicles break
        while (s ==0):
            attempts -= 1
            if(attempts < 1):
                return matrix(0)
            b = Cd.random_element();
            for i in range(n):
                s += (b[i])^2;

        #Enrich nG with the new codeword
        nG = block_matrix([[nG],[matrix(b)]],subdivide=False);
        obtained_k = rank(nG);
        #if the final matrix has wrong number of rows return 0.
        if(nG.nrows() > k):
            return matrix(0)

    return (nG)

#################################

#Convert WEF of Sendrier's signature to string of polynomials
def WefToPolyStr(WEF):
    wefLen = len(WEF)
    #if the list of weight distribution has only one element
    #then it can only be the null word.
    if wefLen <= 1:
        return "1"
    s = str(WEF[0])
    #for each element of WEF, give it the corresponding power of X.
    for i in range(1,wefLen):
        if i == 1 and WEF[i] > 0 :
            s = s + "+" + str(WEF[i]) +"X"
        elif WEF[i] > 0:
            s = s + "+" + str(WEF[i]) +"X^"+str(i)
    return s


#S:(C,i) -> (W(H(C i)) , W(H(C^T i)))  <-- first calculate the dual then puncture.
#Compute the Sendrier's signature
def Sendrier_signature(Fq,C,i):
    # get the generator matrix of code C
    C = LinearCode(C)
    G = C.generator_matrix();
    #puncturing it in the position i.
    G = G.delete_columns([i]);
    #then compute the hull, as required by the formula.
    HG = hull(G);
    #if the hull is null, then
    #the weight distribution will have only a null vector.
    if HG == 0:
        WEF1 = [0];
    else:
        HGC = LinearCode(matrix(HG));
        WEF1 = HGC.weight_distribution();

    #now consider the punctured code of the dual code.
    H = C.parity_check_matrix();
    H = H.delete_columns([i]);
    HH = hull(H);
    if HH == 0:
        WEF2 = [0];
    else:
        HHC = LinearCode(matrix(HH));
        WEF2 = HHC.weight_distribution();
    #return a list containing thw two weight distribution
    #in the form of stringfied polynomial
    return [WefToPolyStr(WEF1),WefToPolyStr(WEF2)];

#################################

#Return H of subfield subcode, works only with not extented destination field.
#Params: H original code's parity check matrix, ell = m/m' , Fp is a subfield of Fq.
#For subcodes over subfield with field still an extented, use codes.Subfieldsubcode(C,Fp)
def get_H_subcode(H, ell, Fq, Fp):

    #consider the vector space over Fq with base over Fp.
    #from_V and to_V are two function that maps
    #a polynomial over an extended field to a tuple and viceversa.
    V, from_V, to_V = Fq.vector_space(Fp, map=True);

    r = H.nrows();
    n = H.ncols();
    #H_prime is the parity check matrix of subfield subcode
    #with entry over Fp and number of columns still n
    #but the number of rows are multiplied by ell.
    H_prime = matrix(Fp,ell*r,n);
    '''
    divide the coefficients of each polynomial of the original matrix and
    insert them one by one into the column of the new matrix H_prime.
    '''
    for i in range(r):
        for j in range(n):
            vector_h_ij = to_V(H[i,j]);
            #print(H[i,j])
            #print(vector_h_ij);
            for u in range(ell):
                #print(vector_h_ij[u])
                H_prime[i*ell+u,j] = vector_h_ij[u];

    return H_prime;

#################################

#Compute the new signature
def new_signature_function(Fq,B):
    #q is the cardinality,that is the number of element of
    #the finite field Fq.
    q = Fq.cardinality();
    k = B.nrows();
    n = B.ncols();

    #Generate all vectors over Fq with length k
    #they will be the messages to be encoded.
    V = VectorSpace(Fq,k);

    #Generate all codewords and insert it into the matrix L
    #so the new matrix is composed by all codewords.
    #each row is a codeword.
    L = matrix(Fq,q^k,n);
    i = 0;
    for u in V:
        L[i,:] = matrix(u)*B;
        i += 1;

    #We now hash all rows and columns, after we sort them
    #Let's start with the rows
    hashes_rows = [];
    for i in range(q^k):
        row_i = L[i,:];
        val_i = sorted(row_i.list());
        hash_i = hash(str(val_i));
        hashes_rows.append(hash_i);

    #Now, we sort the vector and hash it again to make it more compact
    hashes_rows = sorted(hashes_rows);
    row_hash = hash(str(hashes_rows));

    #Now, we consider the columns
    hashes_cols = [];
    for i in range(n):
        col_i = L[:,i];
        val_i = sorted(col_i.list());
        hash_i = hash(str(val_i));
        hashes_cols.append(hash_i);

    #Now, we sort the vector like we did to hashes_rows.
    hashes_cols = sorted(hashes_cols);
    col_hash = hash(str(hashes_cols));
    #return the row hash and the column hash.
    return [row_hash, col_hash];

################################

'''
Build the adjacency matrix from code generator matrix.
given G the required matrix, then
the formula is A = G^T (GG^T)^-1 G
it requires that the hull of G is null, otherwise (GG^T)^-1 cannot be performed.
'''
def build_adjacency(G):
    U = G*G.transpose();
    return G.transpose()*U.inverse()*G;

##Simple GIP solver. First compute the adjacency matrices of two input codes,
#then compute the permutation matrix that connects the two isomorphic graphs.
#Very likely to fail with small q
def GIP(G,G_prime,P):
    #the hull must be null.
    G_hull = hull(G)
    if G_hull.nrows() != 0 :
        #print("G's hull is not null")
        return "hullnot0"
    #Take the generator matrices and then reduce them to adjacency matrices.
    G = LinearCode(G).generator_matrix();
    G_prime = LinearCode(G_prime).generator_matrix();
    n = G.ncols();
    #Construct adjacency matrices
    A = build_adjacency(G);
    A_prime = build_adjacency(G_prime);

    #verify that the reduction hold
    #print("A' = P^T*A*P ? ",A_prime == P.transpose()*A*P);

    #we now recover P; we also consider that the algorithm may fail
    #for each row, we consider the multiset given by the entries of the row
    #in vulgar words, we sort all rows of the adjacency matrices so the
    #effect of permutation on rows disappears.
    unique_values = [];
    unique_values_prime = [];
    for i in range(n):
        vals = sorted(A[i,:].list());
        unique_values.append(vals);

        vals_prime = sorted(A_prime[i,:].list());
        unique_values_prime.append(vals_prime);

    #we now search for correspondences; our guess is my_P
    #that is,we look for rows after the permutation.
    my_P = matrix(ZZ,n,n);
    for i in range(n):
        for j in range(n):
            if unique_values_prime[j] == unique_values[i]: #we found a corresponding index
                pos_i = j;

        my_P[i,pos_i] = 1;

    #check whether the found solution is equal to the trush.
    #print("my_P == P ?",my_P == P);
    if my_P == P:
        return "success"
    else:
        return "my_p_wrong"

#################################

#Generate generator matrix for weakly self-dual code
def generate_weakly_self_dual_code(Fq,n,k,k_hull):

    #Parameter for algorithm
    num_new_coordinates = 1; #this is a parameter, don't touch it

    # the code starts here
    # first, we generate a self-orthogonal codeword.
    is_C_self_orto = 1;
    while is_C_self_orto != 0:
        C = random_matrix(Fq,1,n);
        is_C_self_orto = sum([C[0,i]^2 for i in range(n)]);

    #bring the codeword into sys form (eventually, permute columns)
    i = 0;
    #first, we search a non null element of the codeword.
    while C[0,i]==0:
        i+=1;

    #then we swap this element with the first element.
    tmp = C[0,0];
    C[0,0] = C[0,i];
    C[0,i] = tmp;
    #normalization.
    C = C[0,0]^-1*C;

    kp = 1; #this is the number of found linearly independent codewords
    while kp < k:

        #Obtain H matrix of the systematic form of generator matrix C.
        V = C[:,kp:];
        A = -V.transpose();

        #sample new codeword (we compute only the non systematic part of V)
        u = random_matrix(Fq,1,n-kp);
        u_parity = u*A;

        if kp < k_hull: #in this case, search for a self orthogonal codeword

            mu = sum([u[0,i]^2 for i in range(n-kp)]) + sum([u_parity[0,i]^2 for i in range(kp
     )]);

            while mu != 0: #loop until a self-orthogonal codeword is found

                #modify only num_new_coordinates positions in u (and in the corresponding
     codeword)
                delta_u_support = Combinations(n-kp,num_new_coordinates).random_element();
                for i in delta_u_support:
                    val = Fq.random_element();
                    while val == 0:
                        val = Fq.random_element();
                    u[0,i] += val;
                    u_parity += (val*A[i,:]);

                mu = sum([u[0,i]^2 for i in range(n-kp)]) + sum([u_parity[0,i]^2 for i in
     range(kp)]);
        else: #in this case, search for a codeword which is not self-orthogonal

            mu = sum([u[0,i]^2 for i in range(n-kp)]) + sum([u_parity[0,i]^2 for i in range(kp
     )]);

            while mu == 0: #loop until a self-orthogonal codeword is found

                #modify only num_new_coordinates positions in u (and in the corresponding
     codeword)
                delta_u_support = Combinations(n-kp,num_new_coordinates).random_element();
                for i in delta_u_support:
                    val = Fq.random_element();
                    while val == 0:
                        val = Fq.random_element();
                    u[0,i] += val;
                    u_parity += (val*A[i,:]);

                mu = sum([u[0,i]^2 for i in range(n-kp)]) + sum([u_parity[0,i]^2 for i in
     range(kp)]);

        #Add new codeword and see if if enriches dimension
        attempt_new_C = block_matrix(Fq,2,1,[C,block_matrix(Fq,1,2,[u_parity,u])]);
        p = copy(attempt_new_C);

        #Elimination for lower part
        for i in range(kp):
            attempt_new_C[kp,:] += (-attempt_new_C[kp,i]*attempt_new_C[i,:]);

        #Find pivot
        pivot_pos = kp;
        flag_pivot = 0;
        while (pivot_pos < n)&(flag_pivot == 0):
            if attempt_new_C[kp,pivot_pos]==0:
                pivot_pos += 1;
            else:
                flag_pivot = 1;

        #print(flag_pivot);
        if flag_pivot:
            if pivot_pos != kp: #swap columns to bring the pivot in position kp
                tmp = attempt_new_C[:,kp];
                attempt_new_C[:,kp] = attempt_new_C[:,pivot_pos];
                attempt_new_C[:,pivot_pos] = tmp;

            attempt_new_C[kp,:] = attempt_new_C[kp,kp]^-1*attempt_new_C[kp,:];
            #Do elimination for upper part
            for i in range(kp):
                attempt_new_C[i,:] += (-attempt_new_C[i,kp]*attempt_new_C[kp,:]);

            #update C and kp
            kp += 1;
            C = attempt_new_C;

    return C;

#################################

# G_prime,S,G,P = genTwoEquiv(Fq,n,k,hull_dim)
# generate two equivalent code.
def genTwoEquiv(Fq,n,k,hull_dim):
    #if the required hull_dim is zero,
    #then we can't use generate_weakly_self_dual_code
    if hull_dim == 0:
        flag_ok = 0
        #repeat until we get a null hull matrix.
        while flag_ok == 0:
            G = random_matrix(Fq,k,n);
            a = hull(G).nrows()
            if a == 0 :
                flag_ok = 1;
    else:
        G = generate_weakly_self_dual_code(Fq,n,k,hull_dim);
        '''
        G = generate_weakly_self_dual(Fq,n,k,hull_dim)
        while G.nrows() != k:
            G = generate_weakly_self_dual(Fq,n,k,hull_dim)
        '''
    #generate the permutation matrix and the change of basis.
    P = generate_permutation(Fq,n);
    S = generate_change_of_basis(Fq,k);
    #compute the permuted equivalent code.
    G_prime = S*G*P;
    return G_prime,S,G,P;

#################################
##Function to intersect two codes; it returns a basis for the intersection space
def span_intersection(G1,G2,Fq):
    n = G1.ncols();
    Vn = VectorSpace(Fq,n);
    V1 = Vn.subspace(G1);
    V2 = Vn.subspace(G2);
    V12 = V1.intersection(V2);
    G12 = V12.basis();
    return(G12);

#apply Frobenius to matrix G
def frob(G,frob_endo):
    #apply Frobenius endomorphism to every element of G.
    return G.apply_map(lambda x: frob_endo(x))

#works when rate > 1/2, get the intersection code of code itself with it's Frobenius
     automorphism
def frob_intersec(G):
    Fq = G[0,0].parent()
    #get the Frobenius endomorphism function
    fr = Fq.frobenius_endomorphism()
    G1 = G;
    #apply Frobenius to matrix G
    G2 = frob(G,fr);
    #return the intersection between G and his Frobenius endomorphism.
    return matrix(span_intersection(G1,G2,Fq))

################################

#convert matrix over finite field to their integer representation.
def FqToIntMatrix(G):
    n = G.ncols()
    #create a new matrix.
    GADINT = matrix(n,n);
    Fq = G[0,0].parent()
    #check if the field is prime.
    p_is_prime = Fq.is_prime_field()
    for i in range(n):
        for j in range(n):
            #if the field is not prime, then we can use integer_representation function
            if p_is_prime == false:
                GADINT[i,j] =  GAD[i,j].integer_representation()
            else:
                #otherwise we just need to do a convertion.
                GADINT[i,j] =  int(GAD[i,j])
    return GADINT

################################

# determine whether two adjancency matrix are ispmorphic. need import:
#import networkx as nx
#import matplotlib.pyplot as plt
import networkx.algorithms.isomorphism as iso

#enable checks for weights for is_isomorphic funtion.
em = iso.categorical_edge_match('weight', 'weight')

def GIP_is_isomorphic(GADINT,GADINT2):
    #convert matrices to numpy matrices.
    N = numpy.matrix(GADINT)
    N2 = numpy.matrix(GADINT2)

    ### ** change one number, so is_isomorphic will give false ** ###
    '''
    change_one_element = False;
    if change_one_element == True:
        if N[1,1] == 0:
            N[1,1] = 1;
        else:
            N[1,1] = 0
    '''
    ### now convert to networkx graph ###
    X=nx.from_numpy_matrix(N)
    X2=nx.from_numpy_matrix(N2)  #, parallel_edges = False

    #nx.draw(G)   # default spring_
    #plt.show()

    x = nx.is_isomorphic(X, X2, edge_match = em)
    return x
```

Code Snippet 2

Listing 2: compare_puncturing_shortening_sendrier

```
reset()
load("func_chen.sage")

#### define parameters for code ####
n = 20;   #code length
k = 10;   #code dimension
hull_dim = 0;    #desired dimension for the hull
#finite field parametyers
q = 3;
F_source = GF(q)

########################################################
''' # comparison signatures between Sendrier's, Puncturing and Shortening
'''
#a dictionary to put the results
plotlists = {}
#number of tests to compute the mean
numtest = 100
#container to put the results of different signatures
punct = []
short = []
sendr = []
#for cicle that repeats for hulldim from 2 to 7
for hd in range(1,7):
    hull_dim = hd+1
    #counters to save the number of colliding indices.
    punctnum = 0
    shortnum = 0
    sendrnum = 0

    for testth in range(numtest):
        print(testth , " hd ", hull_dim , " q ", q )
        #first we generate two equivalent code for test.
        G2,S,G,P = genTwoEquiv(F_source,n,k,hull_dim)
        #get the hull of G
        GHULL = hull(G)
        #get the shortened/punctured version of G
        GSHORT = LinearCode(G).shortened([0])
        GPUNCT = G.delete_columns([0])
        #compute the hull for shortened/punctured version
        GSHORT = hull(GSHORT)
        GPUNCT = hull(GPUNCT)
        #compute the weight distrbutions
        w_short = LinearCode(GSHORT).weight_distribution()
        w_punct = LinearCode(GPUNCT).weight_distribution()
        w_sendr = Sendrier_signature(F_source,GHULL,0)
        #get the hull of G2
        G2HULL = hull(G2)
        #for each column of the permuted code G2:
        for j in range(n):
            #get the shortened/punctured version of G2 in j-th position
            G2SHORT = LinearCode(G2).shortened([j])
            G2PUNCT = G2.delete_columns([j])
            #compute the hull
            G2SHORT = hull(G2SHORT)
            G2PUNCT = hull(G2PUNCT)
            #compute the weight distrbutions of manipulated codes
            w2_short = LinearCode(G2SHORT).weight_distribution()
            w2_punct = LinearCode(G2PUNCT).weight_distribution()
            w2_sendr = Sendrier_signature(F_source,G2HULL,j)
            #if the signatures are is equal
            #increase the corresponding counter.
            if w_punct == w2_punct:
                punctnum += 1

            if w_short == w2_short:
                shortnum += 1

            if w_sendr == w2_sendr:
                sendrnum += 1
    #get the mean colliding indices by dividing counter by numtest.
    mean_punct = N(punctnum/numtest)
    mean_short = N(shortnum/numtest)
    mean_sendr = N(sendrnum/numtest)
    punct.append((hull_dim , mean_punct));
    short.append((hull_dim , mean_short));
    sendr.append((hull_dim , mean_sendr));

plotlists['punct'] = punct
plotlists['short'] = short
plotlists['sendr'] = sendr

#Plotting
g =  list_plot(plotlists['punct'] , color='red', plotjoined=True, marker = 'x', legend_label=
     " Puncturing");
g += list_plot(plotlists['short'] , color='blue', plotjoined=True, marker = 'o', legend_label=
     ' Shortening');
g += list_plot(plotlists['sendr'], color='green', plotjoined=True, marker = 'v', legend_label=
     " Sendrier's");

g.set_legend_options(borderpad=1,loc=1,shadow=False,fancybox=True)
xlabel = "hull dim."
ylabel = "mean colliding indices"
g.axes_labels( [ xlabel , ylabel])
g.fontsize(10)
g.show();
```

Code Snippet 3

Listing 3: test_mean_colliding_indices

```
reset()
load("code_utils.sage")

#### define parameters for code ####
n = 18;   #code length
k = 7;   #code dimension
hull_dim = 0;    #desired dimension for the hull
#finite field parametyers
q = 3;
Fq = GF(q)

########################################################
'''  test Mean colliding Indices (puncturing) for SSA '''
#this list saves the results for each different value of q.
resultsForQ = [];
numtest = 1000
for q in [2,3,5,7]:
    #this list saves the results for each different value of hull dimension.
    resultsForHull = []
    F_source = GF(q)
    # hull dimension from 1 to k
    for ii in range(k):
        hull_dim = ii+1 #hull_dim from 1 to k, add 1 because range begins from 0
        colliding_indices_count = 0 #counter for colliding indices.
        for iii in range(numtest):
            print(iii , " hd ", hull_dim , " q ", q )
            #generate two equivalent code
            G2,S,G,P = genTwoEquiv(F_source,n,k,hull_dim)
            #calculate the hull of G after puncturing.
            G_h_p = hull(G.delete_columns([0]))
            #now compute the weight_distribution
            #if the hull is null, then consider the weight_distribution as 0
            if G_h_p.nrows() == 0:
                WG = 0
            else:
                G_h_s_c = LinearCode(G_h_p)
                WG = G_h_s_c.weight_distribution()

            #compute the weight_distribution of the hull of permuted code after puncturing.
            for j in range(n):
                G2_h_p = hull(G2.delete_columns([j]))
                if G2_h_p.nrows() == 0:
                    W2G = 0
                else:
                    G2_h_s_c = LinearCode(G2_h_p)
                    W2G = G2_h_s_c.weight_distribution()
                if WG == W2G:
                    colliding_indices_count += 1

        #get the average colliding indices.
        mean_colliding_indices = colliding_indices_count/numtest;
        resultsForHull.append(  (hull_dim , mean_colliding_indices ) );
    #add the result for a specific q.
    resultsForQ.append(resultsForHull)

#plotting
g = list_plot(resultsForQ[0], color='red', plotjoined=True, marker = 'o', legend_label=' q = 2
    ');
g += list_plot(resultsForQ[1], color='blue', plotjoined=True, marker = 'o', legend_label=' q =
     3');
g += list_plot(resultsForQ[2], color='green', plotjoined=True, marker = 'o', legend_label=' q
    = 5');
g += list_plot(resultsForQ[3], color='black', plotjoined=True, marker = 'o', legend_label=' q
    = 7');

g.set_legend_options(borderpad=1,loc=1,shadow=False,fancybox=True)
xlabel = "hull dim."
ylabel = "mean colliding indices"
g.axes_labels( [ xlabel , ylabel])
g.fontsize(10)
g.show();
```

Code Snippet 4

Listing 4: shortening_multiple_columns

```
reset()
load("code_utils.sage")

### this is the script to see how the colliding indices go if we shorten multiple columns.
def colliding_indexes_calculator(q,n,k,hull_dim,t):
    #generate the code with specified dimension of hull and calculated an permutation.
    #n = 20; #code length
    #k = 6; #code dimension
    #q = 3; #finite field cardinality
    #hull_dim = 6;  #desired dimension for the hull
    Fq = GF(q);
    #generate a weakly self dual code.
    G = generate_weakly_self_dual(Fq,n,k,hull_dim);
    while G.nrows() != k:
        G = generate_weakly_self_dual(Fq,n,k,hull_dim);
    #generate the permutation matrix and the change of basis
    P = generate_permutation(Fq,n);
    S = generate_change_of_basis(Fq,k);
    #compute the permuted G
    G_prime = S*G*P;
    #convert the matrices to codes.
    G_code = LinearCode(G)
    G_prime_code = LinearCode(G_prime)

    #t = 4; #number of positions in which we do the shortening
    G_short = code_shortening(G_code,range(t));
    #consider the hull
    H_G_short = hull(G_short);
    #weight_distribution of Hull of shorted G.
    Whgs = LinearCode(H_G_short).weight_distribution();

    #generate all combinations, they are 'n choose t' elements.
    combinations = Combinations(n,t);
    number_of_combinations = len(combinations);
    #a list to save indices that collide.
    colliding_improved_indexes=[];
    counter1 = 0;

    t0= time.time();

    for i in combinations:
        counter1 = counter1 + 1;
        if(counter1 % 800 == 0):
            print(str(counter1) + "/" + str(number_of_combinations));
        #apply the multiple shrotening to the permuted code.
        G_prime_short = code_shortening(G_prime_code,i);
        #consider the hull
        H_G_prime_short = hull(G_prime_short);
        Whgps = LinearCode(H_G_prime_short)
        Whgps = Whgps.weight_distribution();
        #add to list if the weight distribution are equal.
        if Whgps==Whgs:
            colliding_improved_indexes.append(i);


    t1 = time.time();
    #print("time elapsed = " +str( t1-t0) );
    #print("number of combinations = " + str(number_of_combinations) ) ;
    #print("colliding_improved_indexes length = " + str(len(colliding_improved_indexes)));
    return len(colliding_improved_indexes);


#parameters
n = 20
k = 10
hull_dim = 10
q = 5

t = 4

#test params
```

```python
num_test = 100

mean_colliding_indices = 0

for kk in range(num_test):
    mean_colliding_indices += colliding_indexes_calculator(q,n,k,hull_dim,t)

#compute the average colliding indices.
mean_colliding_indices = mean_colliding_indices / num_test
print(N(mean_colliding_indices))
```

Code Snippet 5

Listing 5: comparison_sendr_new

```
reset()
load("code_utils.sage")

#### define parameters for code ####
n = 18;   #code length
k = 7;   #code dimension
hull_dim = 0;    #desired dimension for the hull
#finite field parametyers
q = 3;
Fq = GF(q)

#######################################################
    ## comparison New signature with Sendrier ##
#container to put the results
list_sendr = [];
list_new = [];

numtest = 100 # we do only 100 test because the computation of signatures are very slow.
for q in [5]:
    #container to put the results for each hull dimension.
    list_hull_sendr = []
    list_hull_new = []
    #define the finite field of q element.
    F_source = GF(q)
    for xx in range(1,k):
        hull_dim = xx+1 # hull_dim from 1 to k.
        #counters to save the number of colliding indices.
        sendr_count = 0
        new_count = 0
        #we repeat SSA for 'numtest' times
        for ii in range(numtest):
            print(ii , " hd ", hull_dim , " q ", q )
            #first we generate two equivalent code for test.
            G2,S,G,P = genTwoEquiv(F_source,n,k,hull_dim)
            #get the hull of G
            GHULL = hull(G)
            #shortening the first column.
            GSHORT = LinearCode(GHULL).shortened([0])
            #compute the Sendrier_signature, we pass the hull of G
            WGSEN = Sendrier_signature(F_source,GHULL,0)
            #compute the new_signature, we pass the first column shortened hull.
            WGNEW = new_signature_function(F_source,GSHORT.generator_matrix())

            G2HULL = hull(G2) #get the hull of the permuted G
            #for each column of the permuted code:
            for j in range(n):
                G2SHORT = LinearCode(G2HULL).shortened([j])
                W2SEN = Sendrier_signature(F_source,G2HULL,j)
                W2NEW = new_signature_function(F_source,G2SHORT.generator_matrix())
                #if the signature of Sendrier or new signature is equal
                #increase the corresponding counter.
                if WGSEN == W2SEN:
                    sendr_count += 1

                if WGNEW == W2NEW:
                    new_count += 1
        #get the mean colliding indices by dividing counter by numtest.
        mean_sendr = N(sendr_count/numtest);
        mean_new = N(new_count/numtest);
        print("results check")
        print(mean_sendr)
        print(mean_new)
        list_hull_sendr.append(  (hull_dim , mean_sendr   ) );
        list_hull_new.append(  (hull_dim , mean_new   ) );


    list_sendr.append(list_hull_sendr)
    list_new.append(list_hull_new)

#plotting the trend of two signatures by increasing the hull dimension.
```

```
g = list_plot(list_sendr[0], color='red', plotjoined=True, marker = 'o', legend_label= "
     Sendrier's");
g += list_plot(list_new[0], color='blue', plotjoined=True, marker = 'o', legend_label=' New
     signature');

g.set_legend_options(borderpad=1,loc=1,shadow=False,fancybox=True)
xlabel = "hull dim."
ylabel = "mean good indices"
g.axes_labels( [ xlabel , ylabel])
g.fontsize(10)
g.show();
```

Code Snippet 6

Listing 6: test_hull_subf_random

```
reset();
load('code_utils.sage');

#function that compute the subfield subcode.
def subfield_subcode(C,Fq,Fp,ell):

    H = C.parity_check_matrix();
    H_prime = get_H_subcode(H, ell, Fq, Fp);

    return codes.LinearCode(H_prime).parity_check_matrix();


#Code and finite field parameters
k = 15;
n = 20;

k_hull = 5;

p = 5;
m = 2;
m_prime = 1;

num_test = 500;

#######Finite fields
q = p^m;
b = p^m_prime;
ell = m/m_prime;

Fq = GF(q);
Fb = GF(b);

#these vectors will be used to contain the occurency of different dimension of hull.
hull_dim_self = vector(ZZ,n+1);
hull_dim_random = vector(ZZ,n+1);

#these vectors will be used to contain the occurency of different dimension of subfield
     subcode.
subcode_dimension_self = vector(ZZ,n+1);
subcode_dimension_random = vector(ZZ,n+1);

for i in range(num_test):

    print(i);
    #looking at weakly self dual code
    # if the initial hull is null, then we generate a random matrix.
    if k_hull == 0 :
        G = random_matrix(Fq,k,n)
        while rank(G) != k or hull(Fq,G).nrows() > 0 :
            G = random_matrix(Fq,k,n)
    else:
        #otherwise we generate a weakly self dual code
        G = sample_weakly_self_dual_code(q,n,k,k_hull);


    #print("Code generated! ")#, test_hull_dim, ' ',true_hull_dim);
    #print("---------------");
    G_prime = subfield_subcode(codes.LinearCode(G),Fq,Fb,ell);
    k_prime = G_prime.nrows();
    #save the obtained subcode_dimension
    subcode_dimension_self[k_prime] += 1;

    hull_G = hull(codes.LinearCode(G_prime));
    k_prime = hull_G.nrows();
    #save the obtained hull dimension
    hull_dim_self[k_prime]+=1;

    ######---- now, generate random code with same dimension and length ----#####
    H = random_matrix(Fb, ell*(n-k), n);
    nG = codes.LinearCode(H).parity_check_matrix();

    k_prime = nG.nrows();
    #save the obtained subcode_dimension
    subcode_dimension_random[k_prime] += 1;

    hull_G = hull(codes.LinearCode(nG));
    k_prime = hull_G.nrows();
    #save the obtained hull dimension
    hull_dim_random[k_prime]+=1;


#new list to save the average number of the dimension
vals_self = [];
vals_rnd = [];
dim_self = [];
dim_rnd = [];

'''
we divide the sum of the dimensions by
the number of tests thus obtaining the average number of the dimension.
'''
for i in range(n+1):
    if hull_dim_self[i]>0:
        vals_self.append((i,hull_dim_self[i]/num_test));
    if hull_dim_random[i]>0:
        vals_rnd.append((i,hull_dim_random[i]/num_test));

    if subcode_dimension_self[i]>0:
        dim_self.append((i,subcode_dimension_self[i]/num_test));
    if subcode_dimension_random[i]>0:
        dim_rnd.append((i,subcode_dimension_random[i]/num_test));

#plotting
g = list_plot(vals_self, color='red', plotjoined=True, marker = 'x', legend_label=' Hull
     Dimension - Weakly-self dual');
g += list_plot(vals_rnd, color='blue', plotjoined=True, marker = 'x', legend_label=' Hull
     Dimension - Random');

g.set_legend_options(borderpad=1,loc=1,shadow=False,fancybox=True)
g.axes_labels( ["Hull dimension","probability"])
g.fontsize(10)
g.show();
#g.save("g.svg")
```

Code Snippet 7

Listing 7: random_code_hulldim_occurency

```
reset()
load("code_utils.sage")

# small test on probability of hulldim , result : decreases as 1/q(n^2+n)/2

n = 12; #code length
k = 6; #code dimension
hull_dim = 5;  #desired dimension for the hull
if(hull_dim > min(k,n-k)):
    raise ValueError("hull dim incorrect")

#finite field parametyers
p = 2; #prime number
m = 1; #the power of prime number ,so q is p^m

#define fields
q = p^m;
Fq = GF(q);
#dictionary to save results.
dictHull = {}
dictRank = {}
number_of_tests = 1000000
for i in range(number_of_tests):
    print(i)
    G = random_matrix(Fq,k,n)
    #G2 = random_matrix(Fq,k,k)
    hull_dimens =   hull(G).nrows()
    #rk = k - rank(G2)
    #add the result to the dictionary.
    #dictRank[rk] = dictRank.get(rk,0)+1
    dictHull[hull_dimens] = dictHull.get(hull_dimens,0)+1

#print(dictRank)
print(dictHull)
```

Code Snippet 8

Listing 8: test_is_isom_forgip

```
# in this script, we test the networkx function: is_isomorphic
# it gives yes or no , so it solves the decisional problem
# we apply this function to the adjancency matrices obtained by reducing
# two equivalent code or random code, to see if the equivalence between codes
# is mirrored to the isomorphism between respective weighed graph.

reset()
load("code_utils.sage")
#import the networkx algorithms
import networkx.algorithms.isomorphism as iso
import time

#set checks for weights
em = iso.categorical_edge_match('weight', 'weight')

#see https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.
     algorithms.isomorphism.vf2pp.vf2pp_all_isomorphisms.html#networkx.algorithms.isomorphism
     .vf2pp.vf2pp_all_isomorphisms

#### define parameters for code ####
n = 30;   #code length
k = 5;   #code dimension
hull_dim = 0;    #desired dimension for the hull
if(hull_dim > min(k,n-k)):
    raise ValueError("hull dim incorrect")

#finite field parametyers
p  = 2**12;
F_source = GF(p,'x');
#check if is prime.
p_is_prime = is_prime(p);

#convert matrix over finite field to their integer representation.
def FqToIntMatrix(G):
    n = G.ncols()
    #create a new matrix.
    GADINT = matrix(n,n);
    Fq = G[0,0].parent()
    #check if the field is prime.
    p_is_prime = Fq.is_prime_field()
    for i in range(n):
        for j in range(n):
            #if the field is not prime, then we can use integer_representation function
            if p_is_prime == false:
                GADINT[i,j] =  GAD[i,j].integer_representation()
            else:
                #otherwise we just need to do a convertion.
                GADINT[i,j] =  int(GAD[i,j])
    return GADINT

# we do a test with growing n, we expect the complexity to grow as n squared
for n in [10,20,50,100,200,500,1000,2000,5000]:
    # consider coderate 1/2
    for k in [floor(n/2)]:
        # consider the extended field 2^8
        for p in [2**8]:

            #finite field parametyers
            F_source = GF(p,'x');
            p_is_prime = is_prime(p);

            #dictionaries to save the results, will save true or false of function:
     is_isomorphic
            results = {}
            results['equiv'] = {} #results of two equivalent code case
            results['random'] = {} #results of two completly random code case
            timeresults = {} #save the time spent.
            #variable to switch between equivalent code and random code
            generateEquivOrRand = 'equiv'

            for iii in range(1000):

                if generateEquivOrRand == 'equiv':
                    #### generate two equivalent code ####
                    G2,S,G,P = genTwoEquiv(F_source,n,k,hull_dim)
                    while rank(G) != k :
                        G2,S,G,P = genTwoEquiv(F_source,n,k,hull_dim)

                else :
                    #### generate two completly random code
                    G = random_matrix(F_source,k,n)
                    while rank(G*G.transpose()) != k or rank(G) != k:
                        G = random_matrix(F_source,k,n)

                    G2 = random_matrix(F_source,k,n)
                    while rank(G2*G2.transpose()) != k or rank(G2) != k:
                        G2 = random_matrix(F_source,k,n)


                #### transform into adjancency matrix ####
                GAD= build_adjacency(G)
                GAD2= build_adjacency(G2)

                #print(iii)
                #### move finite field matrix elment into new integer matrix ####
                # because numpy do not accept matrices over finite field.
                GADINT = matrix(n,n);
                for i in range(n):
                    for j in range(n):
                        if p_is_prime == false:
                            GADINT[i,j] =  GAD[i,j].integer_representation()
                        else:
                            GADINT[i,j] =  int(GAD[i,j])

                GADINT2 = matrix(n,n);
                for i in range(n):
                    for j in range(n):
                        if p_is_prime == false:
                            GADINT2[i,j] = GAD2[i,j].integer_representation()
                        else:
                            GADINT2[i,j] = int(GAD2[i,j])

                #### convert to numpy matrix ####
                N = numpy.matrix(GADINT)
                N2 = numpy.matrix(GADINT2)

                # a little test
                ### ** change one number, so is_isomorphic will give false ** ###
                change_one_element = False;
                if change_one_element == True:
                    if N[1,1] == 0:
                        N[1,1] = 1;
                    else:
                        N[1,1] = 0

                ### to nx graph ###
                X=nx.from_numpy_matrix(N)
                X2=nx.from_numpy_matrix(N2)  #, parallel_edges = False

                #nx.draw(G)   # default spring_
                #plt.show()
                #print('before')

                t0 = time.time()
                # edge_match = em  means we also check edge weights
                x = nx.is_isomorphic(X, X2 , edge_match = em)
                t1 = time.time()
                #print('after')
                #add results in the dictionary
                results[generateEquivOrRand][x] = results[generateEquivOrRand].get(x,0)+1
                timeresults[generateEquivOrRand]  = timeresults.get(generateEquivOrRand,0) + (
    t1-t0)
                if generateEquivOrRand == 'equiv':
                    generateEquivOrRand = 'random'
                else :
                    generateEquivOrRand = 'equiv'

```

```
            print("-------------")
            print('n = ', n , ", k = ", k ,', p = ', p)
            print(results)
            print(timeresults)
            print("--------------")
```

Code Snippet 9

Listing 9: new_solver

```
reset()
load("code_utils.sage")

#### define parameters for code ####
n = 30; #code length
k = 7; #code dimension
hull_dim = 5;  #desired dimension for the hull
#finite field parametyers
p = 5 # the prime number
m = 2 # the power

m_prime = 1; #the power of the subfield.


### compute new parameters
coderate = k/n; #coderate
ell = m/m_prime;
q = p**m;
#define the original finite field and the subfield.
F_source = GF(q,'x');
F_subfield = GF(p**m_prime,'x');

#check if the hull_dim is correct
if hull_dim > min(k,n-k) :
    raise ValueError("hull dim incorrect")
#this version do not support coderate 1/2
if coderate == 1/2:
    raise ValueError("Does not work with R = 1/2")

### generate two equivalent code
G2,S,G,P = genTwoEquiv(F_source,n,k,hull_dim)

# if R<1/l or R>1-(1/l) use Subfield subcode, else use Frobenius intersection
if coderate < 1/ell or coderate > 1-(1/ell):
    usefrob = false
    if coderate > 1-(1/ell):
        # use code itself
        G = LinearCode(G)
        G2 = LinearCode(G2)
    else:
        # use the dual code
        G = LinearCode(G).dual_code()
        G2 = LinearCode(G2).dual_code()
    #get the Subfield Subcodes
    subG = codes.SubfieldSubcode(G, F_subfield).generator_matrix()
    subG2 = codes.SubfieldSubcode(G2, F_subfield).generator_matrix()
else:
    #otherwise we exploit the frobenius intersection
    usefrob = true
    #Frobenius intersection works for rate > 1/2
    if coderate > 1/2:
        pass
    else:
        # use the dual code
        G = LinearCode(G).parity_check_matrix()
        G2 = LinearCode(G2).parity_check_matrix()
    #get the intersection with Frobenius endomorphism
    subG = LinearCode(frob_intersec(G)).generator_matrix()
    subG2 = LinearCode(frob_intersec(G2)).generator_matrix()

## considering the subcode.
if usefrob == true:
    print("using Frobenius intersection subcode")
else:
    print("using Subfield subcode")

print("Subcode dims = ",subG.dimensions())
print("Subcode2 dims = ",subG2.dimensions())

hG = hull(subG)
hG2 = hull(subG2)
```

```

print("Subcode’s hull dims = ",hG.dimensions())
print("Subcode2’s hull dims = ",hG2.dimensions())

#check if permutation is still valid
permutation_is_valid = subG * P * LinearCode(subG2).parity_check_matrix().transpose()
print("Permutation is valid = ", permutation_is_valid.is_zero())

#now, if the hull is null, we proceed with GIP, otherwise we proceed with SSA.
if hG.nrows() == 0 and usefrob == false :
    print("[hull=0] Subcode’s hull is zero, proceed with GIP")
    a = GIP(subG,subG2,P)
    print("[hull=0] GIP solve = " , a)
else:
    print("[hull>0] Subcode’s hull is not zero, proceed with SSA")
    if true:
        Fsub = subG[0,0].parent()
        if usefrob == true:
            print(’[hull>0] use directly Frobenius intersection’)
            CG = subG
            CG2 = subG2
        else:
            print(’[hull>0] use subfield subcodes hull’)
            CG = hG
            CG2 = hG2
        #compute the new signature
        sign1 = new_signature_function(Fsub , CG.delete_columns([0]) )
        good_indices_new_sign = [];
        for i in range(n):
            #print(i)
            sign2 = new_signature_function(Fsub , CG2.delete_columns([i]) )
            if sign1 == sign2:
                good_indices_new_sign.append(i+1);
        #print the colliding indices.
        print( ’[hull>0] ’ ,good_indices_new_sign)
```

Code Snippet 10

Listing 10: theorical_complexity_GSN

```
reset()
load("code_utils.sage")

########################################################
#calculates the probability of a code having a hull of a certain dimension.
def prhull(q,x,maxx):
    #if the required dimension is null
    #then the probability will be 1 minus the
    #sum of all the probabilities of not having a non-nll hull.
    if x == 0:
        somm=0
        for i in range(1,maxx+1):
            asd = prhull(q,x+i,0)
            somm += asd

        return N(1-somm)
    else:
        return 1/(q**((x**2+x)/2))

# theorical formula calculation.
# G = GIP fullcode , S = Sendrier's SSA, N = New solver.
def evaluate_GSN(n,k,d,p,m,ell):

    q = p**m
    mp = m/ell;
    #check if the parameter ell respect the constraints.
    if(ell*k > n and ell*(n-k) > n ):
        print("ell too high *******")
        raise ValueError("ell too high")
    #formula of GIP fullcode
    gipfull = n**(2.3+d)
    #formula of Sendrier's SSA
    sendr = n**3 + log(n)*(q**d)
    #formula of New solver, that is, the sum of
    #the sum of all probabilities of having
    #hulls of a certain dimension multiplied by the respective complexity.
    newsolver = prhull(q,0,ceil(n/2))*(n**2)
    for i in range(1,floor(n/2)):
        newsolver += prhull(q,i,0)*(n**3 + log(n)* (p**(mp))**i )

    print('n=',n,', k=',k,', d=',d, ', q=',p,'^',m,', ell=',ell,'.')
    #consider the complexity in log2
    GPFL = N( log( gipfull , 2))
    SDRS = N( log(sendr,2))
    NEWS =  N( log(newsolver,2))
    print('GIP full: ' ,format(GPFL, ".2f") , ', SendrierSSA: ',format(SDRS, ".2f") , ',
     NewSolver: ',format(NEWS, ".2f")  )

evaluate_GSN(n=20,k=3,d=1,p=2,m=2,ell=2)

'''
below is a non-rigorous simulation of the complexities
of the different algorithms as the hull size varies
'''
#code parameters
n = 50
k = 20
d = 10
#list to save the complexity for each hull_dimension.
list_G = []
list_S = []
list_N = []
for d in [0,1,2,3,4,5,6,7,8,9,10]:
    #parameters of finite field
    p = 5
    m = 8
    ell = 2
    q = p**m
    mp = m/ell;
    #check if the parameter ell respect the constraints.
    if(ell*k > n and ell*(n-k) > n ):
        print("ell too high *******")
        raise ValueError("ell too high")

    gipfull = n**(2.3+d) #GIP fullcode formula
    sendr = n**3 + log(n)*(q**d) #Sendrier's formula
    newsolver = prhull(q,0,ceil(n/2))*(n**2) #New solver formula
    for i in range(1,floor(n/2)):
        newsolver += prhull(q,i,0)*(n**3 + log(n)* (p**(mp))**i )

    #consider the complexity in log2
    GPFL = N( log( gipfull, 2))
    SDRS = N( log(sendr,2))
    NEWS =  N( log(newsolver,2))
    list_G.append ((d, GPFL))
    list_S.append((d, SDRS))
    list_N.append((d, NEWS))

#Plotting
g = list_plot(list_G, color='green', plotjoined=True, marker = 'o', legend_label=' GIP
     fullcode');
g += list_plot(list_S, color='red', plotjoined=True, marker = 'o', legend_label= " Sendrier's"
     );
g += list_plot(list_N, color='blue', plotjoined=True, marker = 'o', legend_label=' New
     signature');

g.set_legend_options(borderpad=1,loc=1,shadow=False,fancybox=True)
xlabel = "hull dim."
ylabel = "Complexity"
g.axes_labels( [ xlabel , ylabel])
g.fontsize(10)
g.show();
```