UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

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

PROTOCOLLI A CONOSCENZA ZERO BASATI SULLA DECODIFICA DI VETTORI RISTRETTI PER FIRME DIGITALI POST-QUANTUM

Zero knowledge protocols based on restricted vector decoding for post-quantum digital signatures

Relatore:
Prof. Marco Baldi

Candidato:
Alessio Pavoni
Mat. 1101018

Correlatori:
Ph.D. Paolo Santini
Ph.D. Violetta Weger

Anno Accademico 2021-2022

Contents

Introduction **6**

1 Notation and preliminaries **9**

2 Digital signature schemes **16**

2.1 Digital signature schemes based on PKE schemes 17

2.2 Digital signature schemes based on ZKID schemes 20

2.2.1 Zero-Knowledge Identification Schemes 20

2.2.2 Fiat-Shamir transform 22

3 Security Analysis **26**

3.1 Complexity classes 28

3.2 ISD algorithms for SDP 31

3.2.1 Prange 35

3.2.2 Lee-Brickell 38

3.2.3 Stern 40

3.2.4 MMT 46

3.2.5 BJMM 53

3.2.6 Algorithm comparison 59

4 Restricted SDP **66**

4.1 ISD algorithms for R-SDP 68

4.1.1 Representation technique for R-SDP 70

4.1.2 Exploiting the additive structure for small z 77

5 Signature schemes 86

5.1 CVE 86

5.1.1 Optimization 91

5.1.2 Applying Fiat-Shamir transform 93

5.2 GPS 94

5.2.1 Sigma protocol with helper 94

5.2.2 Removing the Helper 96

5.2.3 Applying Fiat-Shamir transform 100

5.2.4 Optimization 102

5.2.5 Applying Fiat-Shamir transform 112

5.3 R-CVE and R-GPS 113

5.4 R-BG 115

5.5 R-BG(G) 118

5.5.1 R-SDP(G) 118

5.5.2 Using R-SDP(G) 121

5.6 Comparison with other post-quantum signatures schemes 124

6 Implementation of restricted GPS signature scheme 128

6.1 FindIsometry 128

6.2 SeedPath 129

6.3 ReconstructSeeds 130

6.4 Results 130

7 Conclusion and future work 134

Appendix **136**

7.1 opt_test_MPC.sage 136

7.2 opt_MPC_utils.sage 138

7.3 merkletools.sage 166

Acknowledgements **176**

Nella tesi, redatta in lingua inglese, vengono studiati gli schemi di firma digitale, in particolare quelli basati sugli schemi di identificazione a conoscenza zero (ZKID). Sono anche analizzati i cosiddetti Information Set Decoders (ISD), dei risolutori generici per il Syndrome Decoding Problem (SDP). Dal più semplice Prange fino ai più performanti Stern e BJMM, sono approfondite le modalità con cui essi risolvono il problema SDP.

Viene introdotta una variante del problema SDP basata su insiemi di elementi ristretti, R-SDP. Nuovi risolutori ISD vengono inoltre sviluppati, in modo da permettere l'uso della tecnica delle rappresentazioni. Sfruttando la struttura algebrica del set ristretto è infatti possibile aumentare il numero delle rappresentazioni pur mantenendo contenute le dimensioni delle liste. I set di cardinalità particolarmente ridotta possiedono inoltre più struttura, perciò la complessità degli attacchi si riduce ulteriormente in questi casi. Ne consegue che cardinalità pari o molto basse vanno evitate.

Vengono discussi diversi algoritmi di firma già esistenti basati su ZKID, come CVE, GPS e BG. Applicando loro il problema R-SDP, si denotano riduzioni delle dimensioni di firma. Ciò è dovuto alla minore occupazione in bit degli elementi ristretti, come vettori o trasformazioni monomiali, rispetto alle loro controparti non ristrette. In aggiunta, la restrizione implica l'esistenza di un numero minore di soluzioni, perciò si possono raggiungere pesi di valore massimo per cardinalità abbastanza basse. Nel caso di istanze con peso massimo, le trasformazioni si possono rappresentare col solo vettore di scaling, risultando in un ulteriore risparmio sulle dimensioni di firma e in una velocizzazione dei calcoli, dal momento che sono necessarie delle semplici moltiplicazioni component-wise. Oltretutto, pesi più alti rendono gli ISD più difficili, quindi si possono usare codici di dimensioni minori. Questo è il fattore dominante che provoca la riduzione delle dimensioni di firma. Lo schema R-GPS ottiene dimensioni dell'ordine di 12 kB, ovvero da 8 a 10 kB inferiori rispetto a GPS. Questo sottolinea la validità della restrizione applicata.

Viene realizzata una proof-of-concept in Python di R-GPS, che dimostra che lo schema funziona pur richedendo l'implementazione di funzioni complesse per la gestione dei Merkle Tree e degli alberi binari. I tempi di firma e verifica ottenuti sono elevati ma in linea con quanto aspettato da una proof-of-concept in Python. Ciò accade a causa dell'elevato numero di hash che lo schema deve calcolare, numero che cresce con la dimensione del campo finito. Ad ogni modo, una versione completamente ottimizzata e scritta in linguaggio C avrebbe sicuramente dei tempi migliori. Gli schemi considerati verranno implementati in maniera ottimizzata, dal momento che saranno oggetto di una submission NIST nel quarto round del contest di standardizzazione per schemi di firma digitale.

Per instanze full-weight, un nuovo problema può essere impiegato. R-SDP(G) è basato su di un'ulteriore restrizione del set di matrici diagonali con elementi in $\mathbb{E}$ e risulta in firme ancora più contenute quando viene applicato sugli schemi precedentemente discussi. Le risultanti dimensioni di firma nell'ordine di 7 kB sono notevolmente competitive rispetto alle alternative dello stato dell'arte.

Introduction

In recent years, a considerable amount of research on quantum computers has been made. Quantum computers are machines that use complex quantum mechanical phenomena to solve difficult mathematical problems, having much greater computational power than classical computers. If large-scale quantum computers are ever built, they will be able to break many of the public-key cryptosystems currently in use. As almost every digital communication system relies on these cryptosystems, such a breakthrough would seriously compromise the confidentiality and integrity of modern communications. The goal of *post-quantum cryptography* (PQC) is to develop cryptographic systems that are secure against quantum computers (by extension, also against classical computers) and can interoperate with existing communication systems and networks.

As for now, no large-scale quantum computer exists. The question of when it will be built is a complicated one, but many scientists agree on the fact that it is just a matter of time, as they believe it is now a mere engineering challenge. Therefore, regardless of when this scenario will become a reality, we must begin now to prepare our systems to be able to resist quantum computing. For this reason, NIST (*National Institute of Standards and Technology*, part of the U.S. Department of Commerce) announced in 2013 a PQC standardization contest. regarding Public-key Encryption (PKE) schemes, Key-establishment algorithms and Digital Signature schemes. The standardization contest for each category has been carried out in three rounds.

However, the contest for digital signatures is currently in its fourth round. This additional round aims to find alternatives to the lattice-based schemes, as they were the only ones that went through until the third round. Code-based cryptography, i.e. cryptosystems based on coding theory, represents a promising alternative.

Thesis contribution

This thesis is the outcome of a six-month Research Internship period, from September 2022 to March 2023, hosted by the Coding and Cryptography (COD) group of the Technical University of Munich (TUM), under the supervision of the Information Engineering department of the Polytechnic University of Marche. During the internship, I studied signature schemes based on the Restricted Syndrome Decoding Problem (R-SDP) and analyzed Information Set Decoding (ISD) solvers for R-SDP. I realized a SageMath proof-of-concept implementation of one of the studied signature schemes in order to preliminarily assess its performances. I contributed to the realization of two different scientific submissions, that will be presented throughout the thesis. In the accepted IEEE ISIT (*International Symposium on Information Theory*) submission [4], generic decoding based on ISD for R-SDP with small multiplicative orders has been studied. The results showed that, for low orders of the restricted set, security levels decrease with respect to the ones found in literature. In the Crypto submission [5], we analyzed how R-SDP can be applied to build ZKID-based signature schemes, with a focus both on the protocols and on the ISD attacks. We have also introduced a new version of the problem, called R-SDP(G), that leads to much more compact signatures.

Thesis organization

In Chapter 1 we will define the notation and introduce some useful preliminary notions. Digital signatures and how to build a signature scheme will be discussed in Chapter 2. In Chapter 3 we will dive into the complexity classes of the problems we will build our schemes on and we will explore some existing solvers for the well-known SDP problem. In Chapter 4 we will introduce our restricted SDP idea and present solvers for the new problem. In Chapter 5 we will explore some ZKID-based existing signature schemes and apply our problem to them in order to study their performances. An even newer problem will then be proposed and applied to previous schemes and the results will be compared to other existing schemes. The implementation of the R-GPS scheme and its results are shown in Chapter 6. Finally, the conclusion and future work are presented in Chapter 7. The entire source code of the implementation is reported in the Appendix.

Chapter 1

Notation and preliminaries

Throughout the thesis, we use $[a, b]$ to denote the set of all reals $x \in \mathbb{R}$ such that $a \leq x \leq b$. For a finite set A, the expression $a \overset{\$}{\leftarrow} A$ means that a is chosen uniformly at random from A, while $a \xleftarrow{\mathsf{seed}} A$ analogously means the same but with randomness source seed. In addition, we denote by $|A|$ the cardinality of A, by A^C its complement and by $A_0 = A \cup \{0\}$. As usual, for q being a prime number, we denote by $\mathbb{F}_q = \mathbb{Z}_q$ the finite field of order q and by $\mathbb{F}_q^*$ its multiplicative group. For $g \in \mathbb{F}_q^*$, we denote by $\mathrm{ord}(g) = \min\{i \in \mathbb{Z} \setminus \{0\} \mid g^i = 1 \pmod{q}\} \leq q - 1$ its multiplicative order.

We use uppercase and lowercase letters to indicate matrices and vectors, respectively. If J is a set, we use $\mathbf{A}_J$ to denote the matrix formed by the columns of $\mathbf{A}$ that are indexed by J and we refer to the entry of $\mathbf{A}$ in the i-th row and j-th column as $a_{i,j}$. Analogous notation will be used for vectors. The identity matrix of size m is denoted as $\mathbf{I}_m$. We use $\mathbf{0}_{k \times n}$ and $\mathbf{0}_k$ to denote the null matrix of size $k \times n$ and the null vector of length k, respectively.

Let S_n be the symmetric group on n elements, i.e. the group whose elements are all the bijections from the set $[0, n - 1]$ to itself. We call an element σ of S_n a permutation and its action on a vector $\mathbf{a} \in \mathbb{F}_q^n$ as

$$\sigma(\mathbf{a}) = (a_{\sigma(0)}, \dots, a_{\sigma(n-1)}).$$

Let $\mathfrak{M}_n$ be the set of monomial transformations or isometries, i.e. all linear functions that permute the entries of an input vector and multiply them by a non-null element of $\mathbb{F}_q$. In other words, such transformations can be represented through a permutation vector $\pi \in S_n$ and a vector $\mathbf{v} \in (\mathbb{F}_q^*)^n$ of non-zero scaling factors. The action of the monomial transformation τ on an input vector $\mathbf{a} \in \mathbb{F}_q^n$ can then be expressed as

$$\tau(\mathbf{a}) = \left(v_{\pi(0)}a_{\pi(0)}, \dots, v_{\pi(n-1)}a_{\pi(n-1)}\right).$$

We denote by $h_q : [0,1] \to [0,1]$ the q-ary entropy function:

$$h_q(x) = x\log_q(q-1) - x\log_q(x) - (1-x)\log_q(1-x),$$

of which the binary entropy function h_2 is a special case for $q = 2$:

$$h_2(x) = -x\log_2(x) - (1-x)\log_2(x).$$

For $n \geq \sum_{i=1}^m k_i$ we denote by

$$\binom{n}{k_1, \dots, k_m} = \prod_{i=1}^m \binom{\sum_{j=1}^i k_j}{k_i}\binom{n}{n - \sum_{i=1}^m k_i}$$

the multinomial coefficient, for which the binomial coefficient is a special case for $m = 1$. Recall that for large n the multinomial coefficient can be approximated as

$$\lim_{n\to\infty}\binom{f(n)}{f_1(n), \dots, f_m(n)} = 2^{f(n)\cdot g_m\left(\frac{f_1(n)}{f(n)}, \dots, \frac{f_m(n)}{f(n)}\right)} = 2^{n\cdot F\cdot g_m\left(\frac{F_1}{F}, \dots, \frac{F_m}{F}\right)},$$

where

$$g_m(x_1, \ldots, x_m) = -\sum_{i=1}^{m} x_i \log_2(x_i) - \left(1 - \sum_{i=1}^{m} x_i\right) \log_2 \left(1 - \sum_{i=1}^{m} x_i\right)$$

and $F = \lim_{n\to\infty} \frac{f(n)}{n}$, $F_i = \lim_{n\to\infty} \frac{f_i(n)}{n}$ for all $i \in \{1, \ldots, m\}$. Notice that $g_1 = h_2$ corresponds to the binary entropy function, thus for large n the binomial coefficient can be approximated as:

$$\lim_{n\to\infty} \binom{f(n)}{f_1(n)} = 2^{n \cdot F \cdot h_2\left(\frac{F_1}{F}\right)}.$$

A *linear code* $\mathcal{C}$ is a k-dimensional linear subspace of $\mathbb{F}_q^n$. A linear code can be represented either through a *generator matrix* $\mathbf{G} \in \mathbb{F}_q^{k\times n}$ or through a *parity-check matrix* $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, which have the code as image or as kernel, respectively. We say that a linear code has *rate* $R = k/n$. We define $\mathcal{C}_J$ as $\mathcal{C}_J = \{\mathbf{c}_J \mid \mathbf{c} \in \mathcal{C}\}$. For any $\mathbf{x} \in \mathbb{F}_q^n$, we call $\mathbf{s} = \mathbf{x}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$ a *syndrome.* A set $I \subseteq \{1, \ldots, n\}$ of size k is called an *information set* for $\mathcal{C}$, if $|\mathcal{C}| = |\mathcal{C}_I|$. We say that a generator matrix, respectively a parity-check matrix, is in systematic form (with respect to a set J of size ℓ), if the columns of $\mathbf{G}$ indexed by J form Id_ℓ, respectively, if the columns of H not indexed by J form $\mathrm{Id}_{n-\ell}$. We endow the vector space $\mathbb{F}_q^n$ with the *Hamming metric*: the Hamming weight of a vector $\mathbf{x} \in \mathbb{F}_q^n$ is given by the number of its non-zero entries, i.e.,

$$\mathrm{wt}_H(\mathbf{x}) = |\{i \in \{1, \ldots, n\} \mid x_i \neq 0\}|,$$

which then induces a distance, as $d_H(\mathbf{x}, \mathbf{y}) = \mathrm{wt}_H(\mathbf{x} - \mathbf{y})$, for $\mathbf{x}, \mathbf{y} \in \mathbb{F}_q^n$.

Given a q-ary code (not necessarily linear) of block length n and minimum distance d, it is important to estimate which is the largest possible size $A_q(n, d)$, i.e. the maximum number of codewords, that a code can have.

First, we construct a code $\mathcal{C}$ with minimum distance d and maximum size through a greedy procedure: starting from a codeword, we keep on adding codewords that have a distance at least d from the previously chosen ones, until we cannot add more of them. When this happens, the Hamming balls of radius $d-1$ centered in every codeword must cover the whole space. Indeed, otherwise we could pick one more codeword whose distance from the others is at least d, thus the procedure would not have stopped. By defining the volume of a Hamming ball of radius $\ell \in \mathbb{N}$ centered in any of the codewords as

$$\mathrm{Vol}_q(n,\ell) = \sum_{j=0}^{\ell} \binom{n}{j}(q-1)^j,$$

when the greedy procedure stops it holds that

$$A_q(n,d) \cdot \mathrm{Vol}_q(n,d-1) \geq q^n.$$

This inequality represents the so-called Gilbert-Varshamov bound (in short, GV bound), expressed in Lemma 1.

Lemma 1. ***Gilbert-Varshamov bound***
Let $d \leq n$ and q be positive integers and let $\mathcal{C}$ be a code over $\mathbb{F}_q$. It must hold

$$A_q(n,d) \geq \frac{q^n}{\mathrm{Vol}_q(n,d-1)} = \frac{q^n}{\sum_{j=0}^{d-1}\binom{n}{j}(q-1)^j}. \tag{1.0.1}$$

If $\mathcal{C}$ is linear, then $A_q(n,d) = q^k$ for a $k \leq n$ positive integer and it holds

$$k \geq n - \left\lfloor \log_q\left(\mathrm{Vol}_q(n,d-1)\right)\right\rfloor = n - \left\lfloor \log_q\left(\sum_{j=0}^{d-1}\binom{n}{j}(q-1)^j\right)\right\rfloor.$$

While usually other bounds, like the Singleton bound, provide upper bounds on the size of a code, the GV bound represents a lower bound as a function of the field size q, the block length n and the minimum distance d. These results can be obtained in the asymptotic version, i.e. considering exponential approximations with regard to n of the used quantities. First, the following lemma can be proven:

Lemma 2. *Let n and $q \geq 2$ be positive integers and $\delta \in [0, 1-1/q]$ be a real number. It holds*

$$q^{(h_q(\delta)-o(1))n} \leq \mathrm{Vol}_q(n, \delta n) \leq q^{h_q(\delta)n}.$$

In other words, for n big enough, the asymptotic volume of the Hamming ball of radius δn can be approximated to

$$\frac{1}{n}\log_q\left(\mathrm{Vol}_q(n, \delta n)\right) = h_q(\delta). \tag{1.0.2}$$

The rate of a generic code $\mathcal{C}$ of size $|\mathcal{C}|$ is defined as

$$R = \frac{1}{n}\log_q(|\mathcal{C}|), \tag{1.0.3}$$

which becomes the well known

$$R = \frac{k}{n}$$

if the code is linear. We want to take the asymptotic values of (1.0.1). To do so, we use the result in (1.0.2) and the definition (1.0.3) and we obtain the asymptotic formulation of GV bound, expressed in Lemma 3.

Lemma 3. ***Asymptotic Gilbert-Varshamov bound***

Let q be a positive integer and $\delta \in [0, 1-1/q]$ be a real number. For every q and δ there exists an infinite family $\mathcal{C}$ of q-ary codes with rate

$$R \geq 1 - h_q(\delta) \tag{1.0.4}$$

The asymptotic GV bound states the existence of such codes, but it does not say anything about which ones they are.

Consider a random linear code $[n,k]$ over $\mathbb{F}_q$, i.e. a code where each entry of its parity-check matrix (or, equivalently, generator matrix) is picked uniformly at random. The number of vectors of weight t in $\mathbb{F}_q^n$ is $\binom{n}{t}(q-1)^t$. A vector $\mathbf{c}$ is a codeword if it satisfies the parity-check equations, i.e. $\mathbf{c}\mathbf{H}^\top = \mathbf{0}$. Given that $\mathbf{H}$ is a random matrix, that happens with a q^{k-n} probability. Therefore, the average number of codewords of weight t is

$$\binom{n}{t}(q-1)^t q^{k-n}. \tag{1.0.5}$$

If this quantity is greater than 1, then a codeword with weight t exists on average. Then, by definition the minimum distance d is the minimum value for which the (1.0.5) is greater than 1:

$$d = \min\left\{ t \middle| \binom{n}{t}(q-1)^t q^{k-n} \geq 1 \right\}. \tag{1.0.6}$$

By taking the asymptotics of (1.0.6) we obtain the relative distance

$$\delta = \min\left\{ T \middle| h_q(T) - (1-R) \geq 0 \right\},$$

where $T = t/n$ is the relative weight. Considering that $h_q(T)$ grows monotonically from 0 to 1 for $T \in [0, 1-1/q]$, we can consider only this interval of weights. Then, the minimum relative distance is the one for which

$$h_q(\delta) = (1-R) \tag{1.0.7}$$

Therefore, random linear codes attain the asymptotic GV bound with high probability. Then, when using a random linear code we can assume its relative distance as the solution of (1.0.7).

Chapter 2

Digital signature schemes

A signature is a piece of information that guarantees the legitimate origin of an object. Specifically, a digital signature applies to objects intended as digital messages or documents. We refer to the origin of the signature as the *Signer* and to the destination as the *Verifier*. The requirements a (digital) signature must possess are:

- *Authenticity*: the origin must be legit;
- *Integrity*: the message cannot be altered after it has been signed;
- *Non-repudiation*: the origin cannot deny the signing of the message.

Every digital signature scheme consists of three steps:

1. Key generation;
2. Signing;
3. Verification.

The first two steps are carried out by the Signer. After creating a pair of keys, one being secret (sk) and one being public (pk), he proceeds to sign a message m

through a public-key cryptographic primitive, obtaining a signature σ. The last step is performed by the Verifier, which, after receiving the signed message (m, σ), checks its validity through the same primitive.

As with every other cryptographic scheme, also digital signature schemes are subject to attacks. For instance, an impersonator could produce a forged signature to try to convince the Verifier that he is a legitimate Prover, even without knowing its private key.

The performance of a digital signature scheme is measured by these parameters:

- Signature size;
- Public key size;
- Signing and verification times.

The size of an object is intended as the total number of bits that are used to represent such an object. Clearly, small sizes and short times are preferred.

2.1 Digital signature schemes based on PKE schemes

Asymmetric PKE schemes can be used to build digital signature schemes by simply swapping the encryption and decryption phases. That means applying the secret key to the (*ciphertext*) message to obtain the (*plaintext*) signature and then applying the public key to get the message. In practical cases, to attain smaller signature sizes, a *hash-and-sign* approach is much more used, where the ciphertext is the hash digest of the message.

In 1978, McEliece proposed a public-key encoding system [27] based on algebraic coding theory. In McEliece, the private key is the generator matrix $\mathbf{G} \in \mathbb{F}^{k \times n}$ of a

Goppa code with an error correction capacity of t and the public key is a scrambled version $\mathbf{G}' = \mathbf{SGP} \in \mathbb{F}^{k\times n}$ of the private key, where $\mathbf{S} \in \mathbb{F}^{k\times k}$ and $\mathbf{P} \in \mathbb{F}^{n\times n}$ are a secret invertible matrix and a secret permutation matrix, respectively. The aim of such scrambling is to make $\mathbf{G}'$ indistinguishable from a random matrix. The sender uses this public matrix to encode a secret message, or plaintext, $\mathbf{m}$ (or, more likely, its hashed version) and then adds t intentional errors to the codeword, thus obtaining a ciphertext

$$\mathbf{c} = \mathbf{mSGP} + \mathbf{e}.$$

The legitimate receiver inverts the action of the permutation by computing

$$\mathbf{cP}^{-1} = \mathbf{mSG} + \mathbf{eP}^{-1},$$

obtains $\mathbf{mS}$ through the decoding algorithm and then recovers the message $\mathbf{m}$ by post-multiplying it by $\mathbf{s}^{-1}$. In order to recover the message without knowing the secret key, an eavesdropper is forced to decode a seemingly random code, which is a known hard problem even for quantum computers. Therefore, the trapdoor used by McEliece is the t-error correcting procedure for the Goppa code. Anyway, the use of the trapdoor implies the need for a *security assumption*, which is assuming that the scrambled matrix is not distinguishable from a random one. So far, no known polynomial-time distinguisher exists, so the assumption still holds. However, if such an assumption was proven wrong, the security of McEliece would be compromised. In 1986, Niederreiter proposed a variant [28] of McEliece cryptosystem, equivalent but from the parity-check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ point of view. In this case, the plaintext is an error vector $\mathbf{e} \in \mathbb{F}_q^n$ of weight t and the ciphertext is a syndrome $\mathbf{s} = \mathbf{eH}^\top \in \mathbb{F}_q^{(n-k)}$. In [25], it has been proven that McEliece and Niederreiter are equivalent from a security point of view.

In order to get post-quantum signature schemes, it could make sense to use McEliece/Niederreiter encryption schemes. However, in such cases, the hashed message does not lie in the ciphertext space. For example, considering the Niederreiter scheme (same applies to McEliece), the ciphertext is an error vector $\mathbf{e}$ of weight t, where t is the correcting capability of the used Goppa code. Since a hash digest can be considered as a random string, it is very likely that such a random syndrome will result in an error vector of weight greater than t. Such a syndrome is then not correctable and the signature scheme cannot properly work.

For some time, it was widely believed that McEliece/Niederreiter schemes could not be used for signature schemes. However, in 2001 Courtois, Finiasz and Sendrier showed in [14] that is possible to build a signature scheme based on the Niederreiter scheme by accomplishing complete decoding, i.e. decoding syndromes corresponding to errors of weight greater than t, let us say $t + \delta$. To do so, they add δ random columns of the parity-check matrix to the syndrome (they work over $\mathbb{F}_2$). If such columns correspond to one of the $t + \delta$ error positions, the new syndrome will correspond to an error vector of weight t, which is decodable. However, this process is probabilistic. This makes signing very slow, since multiple attempts need to be made, and furthermore leads to parameter choices for the underlying linear codes that yield very large public keys (approximately 1 MB for less than 128 security bits). Moreover, the use of trapdoors forces the scheme to rely on a hard problem through a security assumption.

2.2 Digital signature schemes based on ZKID schemes

A valid code-based alternative to PKE schemes can be found in *Zero-Knowledge Identification Schemes* (ZKID). An identification scheme aims to guarantee an user's knowledge of a secret object. We talk about zero-knowledge if the user is able to do so while not revealing any piece of information about the secret.

2.2.1 Zero-Knowledge Identification Schemes

The properties of an ideal ZKID scheme are:

- **Completeness**: an honest user must always be correctly identified;
- **Soundness**: a dishonest user must never be accepted;
- **Zero Knowledge (ZK)**: the Prover must not leak information about the secret key.

An example, however general, of an identification scheme is the one briefly shown below. The user that wants to be identified, i.e. the *Prover*, creates a pair of keys (sk, pk), respectively private and public. The identifier, i.e. the *Verifier*, wants to check that the public key has indeed been created by the Prover. Then, the Prover has to provide to the Verifier the so-called *Proof of Knowledge*, which is the proof of the Prover's knowledge of the secret key sk linked to pk. Such proof is provided following the generic protocol shown in Figure 2.1:

- The Prover chooses a random value s and keeps it secret;
- The Prover computes the *commitment* c as a function of the random value s and other additional data: $c = f(s, \text{Data})$;

PROVER		VERIFIER
Choose a random value s		
Compute $c = f(s, \text{Data})$		
	$\xrightarrow{c}$	
		Choose a random value b
	$\xleftarrow{b}$	
Compute $r = g(s, sk, b)$		
	$\xrightarrow{r}$	
		Check r using c and pk

Figure 2.1: General ID scheme

- The Verifier chooses a random *challenge* b;
- The Prover computes the *response* r as a function of s, b and the secret key sk: $r = g(s, sk, b)$;
- The Verifier checks r using c and pk.

The function $f(\cdot)$ must be *one-way* in order to hide the random value s, while $g(\cdot)$ must not leak information about sk.

The property that is evaluated to estimate the security level of a scheme is soundness. In practice, it is highly difficult to attain perfect soundness, so it is expected to have a non-zero probability that a dishonest user could be accepted by the identifier. Such probability is called *soundness error* (ε). In order to get an arbitrarily low *cheating probability*, i.e. the success probability of the attacker, the identifier could require to perform the single protocol multiple times. For instance, to get a *security level* of at least λ, that is a cheating probability of $2^{-\lambda}$, N repetitions (rounds) are necessary, where $N > \frac{\lambda}{log_2(1/\varepsilon)}$. Indeed, it holds that $\varepsilon^N < 2^{-\lambda}$. Therefore, it is clear that for a required cheating probability, the lower the soundness error ε, the smaller the number of necessary rounds, thus the communication cost of the protocol.

The value c is called commitment because it represents the commitment that the Prover makes to identify himself using that value, regardless of the challenge chosen by the Verifier. In other words, by communicating a (supposedly honest) $c = f(s, \text{Data})$ before b has been chosen, the only way the Prover has to get identified is by knowing the secret key, since no other input parameter of $g(\cdot)$ is left unchosen. Then, it is clear that the order we use to communicate the parameters c and b is crucial. Indeed, assuming that b is communicated to the attacker before he sends c, he could choose an appropriate c for which he is able to compute a valid answer, even without knowing the secret key.

If the correct exchange order of parameters is enforced, an attacker could try to guess the *challenge* that will be chosen and then send a "prepared" *commitment*. Such a strategy could lead to a victory only in the case the Verifier will choose the b that the attacker foresaw. If, as usually happens, b is chosen uniformly at random in the {0,1} set, the soundness error is likely to be $\varepsilon = 1/2$.

2.2.2 Fiat-Shamir transform

An identification scheme is an interactive protocol between two actors, the Prover and the Verifier, at the end of which the Verifier outputs the answer about the validity of the Prover identity. Starting from a ZKID scheme it is possible to get a digital signature scheme by applying the so-called ***Fiat-Shamir trasform***. This method allows adapting the protocol used in ZKID schemes such that it is executable by a single actor (*signer*), in order to get the digital signature of a message. In other words, the transform allows the signer to get rid of the interactivity by simulating both the Prover and the Verifier of a ZKID scheme, thus obtaining a digital signature from the transcripts of their simulated communication.

Figure 2.2 describes how the Fiat-Shamir transform is applied to a generic ZKID

SIGNER		VERIFIER
Choose N random values $s_1, ...s_N$		
For $i = 1, .., N$:		
1. Compute $c_i = f(s_i, \text{Data})$		
Compute $B = h(m\|\|c_1\|\|...\|\|c_N)$		
For $i = 1, .., N$:		
1. Compute $r_i = g(s_i, sk, b_i)$		
	$\xrightarrow{\{(c_1,...,c_N),(r_1,...,r_N)\}}$	
		Compute $B = h(m\|\|c_1\|\|...\|\|c_N)$
		For $i = 1, .., N$:
		1. Check r_i using c_i, b_i and pk

Figure 2.2: Fiat-Shamir transform for a generic ZKID scheme

scheme. The signer is the actor that has to sign a message m and the Verifier is the actor that has to check the validity of such a signature. Notice that the signature Verifier is a different actor from the ZKID Verifier, as the latter is just simulated from the signer execution in this scenario. The signer generates the N commitments c_i, one for each round, then he computes the hash digest $h(\cdot)$ of the message m concatenated with the N commitments. The first N bits of the resulting digest are then used as challenge bits B (ZKID Verifier simulation). Then, the N responses r_i are computed according to the underlying ZKID scheme. The digital signature of the message m consists of the list of commitments $(c_1, ..., c_N)$ and responses $(r_1, ..., r_N)$. Therefore, once he receives the signature, the Verifier is able to recompute the challenge bits as the hash digest of the message and the commitments and, then, perform the verification of each r_i as established from the ZKID scheme.

In ZKID protocols the challenge bits are randomly generated. For this reason, the Fiat-Shamir transform uses an appropriate hash function to get such bits. Indeed, some hash functions are *random oracles*, i.e. functions that are able to produce (apparently) random bits from varying inputs. The hash function also guarantees that the declared message and commitments will not be modified afterward. Indeed, being a one-way function, the hash univocally ties the string $message||commitments$ to the challenge bits B. In such a way, once the challenge bits are computed, it is not possible to change the message or the commitments, because every small variation of the hash input would lead to an unpredictable change of B. Therefore, if the signer computes its challenge bits B and computes the responses according to B, but sends different commitments or a different message to the Verifier, then its signature would (probably) not be accepted, as the Verifier will use different challenge bits B' than the ones used by the signer to compute the responses. The one-way property of the hash function guarantees the correct creation order of the commitments and challenges. In order to commit cheating, an attacker should guess all N random challenge bits and prepare appropriate commitments, therefore the cheating probability is ε^N.

The use of ZKID protocols in digital signature schemes leads to several advantages. First, the public and private keys are not tied together through a trapdoor. Indeed, ZKID schemes do not perform decryption, then it is not necessary to make a *security assumption* that justifies the hardness of the underlying problem even using a trapdoor. For example, RSA is an asymmetric cryptosystem that, as it has to allow decryption, uses a trapdoor. It is based on the "hard" problem of factoring a large integer number (it is not *NP-hard*, but no efficient non-quantum attack is known). Although, in order to allow decrypting, a trapdoor has to be used, that is limiting the number of possible integers only to the semiprime integers $n = pq$, with p and q primes. It is assumed (security assumption) that also the problem of factoring a large semiprime integer is hard, even if this could not be true. Therefore, the security assumption is a possible vulnerability of a cryptosystem, because its security would be compromised as soon as the assumption is proven wrong.

Finally, the multiple executions of rounds can be parallelized, thus increasing the execution speed of the schemes. However, the necessity of multiple repetitions leads to bigger signature sizes, even if public keys could be very small.

Chapter 3

Security Analysis

The security of every asymmetric cryptographic scheme relies on an underlying problem, which is supposedly hard to solve. This problem is used to hide the private key of the cryptosystem behind its public key (or the message behind the ciphertext). Therefore, an attacker could try to solve the problem in order to find the private key. Considering that there might be different ways of solving the problem, the attacker will choose the most efficient algorithm, i.e. the one that has the lower cost. Let such a cost be 2^{λ_1}. In case of success, he would be able to cheat at every execution of the scheme by using a valid private key. Although, the security of the cryptosystem also depends on the protocol that uses such a problem. For example, an attacker could try to break the protocol by exploiting its structure without finding a valid private key. For example, as we saw for ZKID schemes and the signature schemes obtained from them, the attacker is successful with a certain (cheating) probability, that we call $2^{-\lambda_2}$. If such a probability is high enough, it could be more convenient for the attacker to use this kind of attack, as it would be less costly than solving the underlying problem. Indeed, the average number of protocol executions required to successfully cheat, which is the cost of the attack if we neglect the cost of the single protocol execution, is 2^{λ_2}. The security level of the cryptographic scheme is then computed considering the least costly approach between the two:

$$\lambda = \log_2 \min\left(\left\{2^{\lambda_1}, 2^{\lambda_2}\right\}\right)$$

We will now focus on the security analysis of the underlying problems of the schemes. Security analysis of a cryptographic scheme generally takes into account two different attack approaches:

1. structural attacks;

2. non-structural attacks.

Structural attacks exploit the known algebraic structure of the scheme (i.e. the Goppa code for McEliece cryptosystem), instead of the non-structural attacks that aim to recover the message or secret key regardless of the algebraic structure. As we will use actual random codes, we will only focus in Section 3.2 on non-structural attacks, for which no information about the structure is used.

Code-based cryptography is traditionally based on the hardness of decoding a random linear block code. Being a linear block code defined by a generator matrix $\mathbf{G}$, the problem can be formalized in the following way:

Problem 1. ***Decoding Problem (DP)***
Let $\mathbb{F}_q$ be a finite field of size q and $k \leq n$ be positive integers. Given $\mathbf{G} \in \mathbb{F}_q^{k\times n}$, $\mathbf{c} \in \mathbb{F}_q^n$ and $t \in \mathbb{N}$, are there two vectors $\mathbf{x} \in \mathbb{F}_q^k$ and $\mathbf{e} \in \mathbb{F}_q^n$ with $wt_H(\mathbf{e}) = t$ such that $\mathbf{c} = \mathbf{xG} + \mathbf{e}$?

Analogously, the problem can be expressed using the parity-check matrix $\mathbf{H}$:

Problem 2. ***Syndrome Decoding Problem (SDP)***
Let $\mathbb{F}_q$ be a finite field of size q and $k \leq n$ be positive integers. Given $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_q^{(n-k)}$ and $t \in \mathbb{N}$, is there a vector $\mathbf{e} \in \mathbb{F}_q^n$ with $wt_H(\mathbf{e}) = t$ such that $\mathbf{eH}^\top = \mathbf{s}$?

The two problems are equivalent, thus we will exclusively refer to SDP from now on. The number of solutions to the SDP problem depends on the weight t. Since the code is random, assuming the syndrome is obtained from a vector $\mathbf{e}$ of weight t, i.e. $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$, the average number of solutions is

$$1 + \frac{\binom{n}{k}(q-1)^t - 1}{q^{n-k}} \simeq 1 + \frac{\binom{n}{k}(q-1)^t}{q^{n-k}}.$$

If there is more than one solution, it is intuitive that the SDP problem gets easier to solve. Thus, it is always required to have (on average) a unique solution. That happens when $\binom{n}{k}(q-1)^t q^{k-n} \leq 1$, or, asymptotically

$$h_q(T) - (1 - R) \leq 0. \tag{3.0.1}$$

Therefore, if the target relative weight T is less than the minimum relative distance δ of the random code, i.e., the one that satisfies (1.0.7), we expect to have on average a unique solution.

3.1 Complexity classes

It is important to notice that, as we defined them, DP and SDP ask if a solution to the system of equations exists, but they do not ask to find it. The problems for which the existence of a solution has to be asserted are called *decisional* problems, while the ones that specifically ask to find the solution are called *computational* problems. DP and SDP can be also defined as computational problems. For the computational SDP, the solution is the vector $\mathbf{e}$ that satisfies the parity-check equations, while for the decisional SDP, the solution is ”YES” or ”NO”, according to whether an appropriate vector $\mathbf{e}$ exists or not. For the security analysis, the computational versions of the problems have to be considered, since an attacker that wants to find the private key has to solve a computational problem.

In order to determine if a cryptographic system can be built on a given problem (for example, SDP), its hardness has to be asserted. To do so, we briefly introduce some basics of complexity theory. The following dissertation will not be rigorous, as the fundamental concepts will be presented in an intuitive way.

Definition 1. *P is the class of (decisional) problems that can be solved in polynomial time in the size of the input.*

In other words, the problems that belong to P are the easiest ones.

Definition 2. *NP is the class of (decisional) problems whose solutions can be checked in polynomial time in the size of the input.*

Clearly, P is a subset of NP, because every problem that can be solved in polynomial time can also be checked in polynomial time. In fact, if a problem $\mathcal{P}_1$ lives in P, it exists a polynomial solver that outputs a binary solution s of $\mathcal{P}_1$. Given a candidate solution $c \in \{\text{"YES"}, \text{"NO"}\}$, we can run the polynomial solver and verify if $s = c$, thus checking the solution c in polynomial time.

A way to compare the hardness of two different problems is polynomial-time reductions.

Definition 3. *Let $\mathcal{R}$ and $\mathcal{P}$ be two (decisional) problems. A polynomial-time reduction from $\mathcal{R}$ to $\mathcal{P}$ is composed of the following steps:*

1. *take any instance I of $\mathcal{R}$;*
2. *transform I to an instance I' of $\mathcal{P}$ in polynomial-time;*
3. *assume that $\mathcal{P}$ can be solved in the instance I' and its solution is s';*
4. *transform s' to a solution s of $\mathcal{R}$ in the instance I in polynomial time.*

In other words, the existence of a polynomial-time reduction from $\mathcal{R}$ to $\mathcal{P}$ means that if we solve $\mathcal{P}$ then we also solve $\mathcal{R}$ by appropriately transforming an instance and a solution. Notice that the opposite does not hold in general, therefore this means that in this case $\mathcal{P}$ is at least as hard to solve as $\mathcal{R}$. The definition of polynomial-time reduction allows us to define the class of problems that are interesting from a post-quantum security point of view.

Definition 4. *NP-hard is the class of (decisional) problems to which it exists a polynomial-time reduction from every problem in NP.*

Following the intuitive meaning of polynomial-time reduction, the problems in NP-hard are at least as hard as every problem in NP, even the hardest ones. Thus, NP-hard problems are considered to be the hardest problems to solve and, for this reason, they are studied for security applications. From the definition, to prove that a problem is NP-hard we would have to find a reduction to it from every problem in NP. Luckily this is not necessary, as it is enough to find a reduction to our problem from a problem that is already known to be NP-hard. Notice also that the NP-hard class is only defined for decisional problems. However, the computational version of a problem is clearly at least as hard as its decisional version, so, being unrigorous, it is safe to state that if the decisional problem is NP-hard, also the computational problem is. It is conjectured that NP-hard problems are quantum-resistant, therefore they are used as underlying problems for post-quantum cryptosystems.

An NP-hard problem is a hard problem that, in general, does not live in NP. It would be nice to work with problems that are hard to solve and easily verifiable at the same time, so that when we are given a solution to the problem we are able to quickly check its validity.

Definition 5. *NP-complete is the class which results from the intersection of NP-hard and NP classes.*

Berlekamp, McEliece and Van Tilborg demonstrated in [9] that SDP for binary linear codes embedded with Hamming metric is NP-complete. Successively, Barg in [7] generalized such a demonstration to finite fields of arbitrary size. Therefore, SDP has been proven to be a valid candidate for the underlying problem of a post-quantum cryptographic scheme.

3.2 ISD algorithms for SDP

As already seen, solving SDP means solving the NP-complete decoding problem of a random linear code. Therefore, the security level of the SDP-based cryptographic scheme depends on the best decoding algorithm for random linear codes. If SDP has few solutions, the most efficient strategies are the so-called ***Information Set Decoding (ISD)*** algorithms. Before presenting the main members of this family of algorithms, we notice that the definition of information set can also be expressed from the $\mathbf{G}$ matrix (or, equivalently, $\mathbf{H}$) point of view:

Lemma 4. *Let $k \leq n$ be positive integers and let $\mathcal{C}$ be a $[n, k]$ linear code over $\mathbb{F}_q$, with generator matrix $\mathbf{G} \in \mathbb{F}_q^{k \times n}$ and parity check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$. If I is an information set of $\mathcal{C}$, then it exists simply a generator matrix $\mathbf{G}'$ of $\mathcal{C}$ where $\mathbf{G}'_I = \mathbf{Id}_k$ and it exists another parity check matrix $\mathbf{H}'$ of $\mathcal{C}$ where $\mathbf{H}'_{I^C} = \mathbf{Id}_{n-k}$.*

Proof. Let us consider the $\mathcal{C}_I$ code, obtained through puncturing from $\mathcal{C}$. $\mathcal{C}_I \subseteq \mathbb{F}_q^k$ is a $[k \times k]$ code with generator matrix $\mathbf{G}_I$.
By hypotesis, it holds that $|\mathcal{C}_I| = |\mathcal{C}| = q^k$. That means that for every $\mathbf{u} \in \mathbb{F}_q^k$ it exists one and only one corresponding codeword $\mathbf{c}_I = \mathbf{u}\mathbf{G}_I \in \mathbb{F}_q^k$. In other words, columns of $\mathbf{G}_I$ are linearly independent, so $\mathbf{G}_I$ is invertible.
Let $\mathbf{U} = \mathbf{G}_I^{-1}$, we can obtain an identity matrix from $\mathbf{U}\mathbf{G}_I = \mathbf{Id}_k$. So, defining $\mathbf{G}' = \mathbf{U}\mathbf{G}$ (which is still a valid generator matrix of $\mathcal{C}$ since we obtained it through

linear combinations of $\mathbf{G}$ rows) it holds that $\mathbf{G}'_I = \mathbf{U}\mathbf{G}_I = \mathbf{Id}_k$. For the parity check matrix, we have $\mathbf{G}'\mathbf{H}'^\top = \mathbf{H}'_I + \mathbf{G}'_{I^C}\mathbf{H}'^\top_{I^C} = \mathbf{0}$, where one of the possible solutions is $\mathbf{H}'_I = -\mathbf{G}'_{I^C}$ and $\mathbf{H}'_{I^C} = \mathbf{Id}_{n-k}$. $\square$

Every ISD algorithm aims to solve the SDP problem. Each one of them considers a certain error distribution in relation to an information set, but, in broad terms, all of them follow the same procedure:

1. Choose an information set I;
2. Bring parity check matrix $\mathbf{H}$ in systematic form in relation to I, that is finding an invertible matrix $\mathbf{U} \in \mathbb{F}_q^{(n-k)\times(n-k)}$ such as $(\mathbf{UH})_I = \mathbf{A}$ for some $\mathbf{A} \in \mathbb{F}_q^{(n-k)\times k}$ and $(\mathbf{UH})_{I^C} = \mathbf{Id}_{n-k}$;
3. For every $\mathbf{e} \in \mathbb{F}_q^n$ with weight t and the supposed error distribution:
   (a) Verify if parity check equations (in systematic form in relation to I) $\mathbf{e}\mathbf{H}^\top\mathbf{U}^\top = \mathbf{s}\mathbf{U}^\top$ are satisfied;
   (b) If they are satisfied, return $\mathbf{e}$.
4. If no valid $\mathbf{e}$ has been found, restart with a new information set I.

Here we want to stress out why I has to be an information set. Let us define I as a generic set of size k. The parity check equations are:

$$\mathbf{e}\mathbf{H}^\top = \mathbf{e}_I\mathbf{H}_I^\top + \mathbf{e}_{I^C}\mathbf{H}_{I^C}^\top = \mathbf{s}$$

Given a certain $\mathbf{e}_I$, in order to get the solution $\mathbf{e}_{I^C}$ we have to invert $\mathbf{H}_{I^C}^\top$ and, according to Lemma 4, this is possible if and only if I is an information set. This is how every ISD algorithm carries Step 3.a out: they find ways of enumerating $\mathbf{e}_I$ vectors of a certain weight, they calculate $\mathbf{e}_{I^C}$ as $\left(\mathbf{s} - \mathbf{e}_I\mathbf{H}_I^\top\right)(\mathbf{H}_{I^C}^\top)^{-1}$ and they check that $\mathbf{e}_{I^C}$ has the remaining weight. Therefore it is mandatory for I to be an information set, because, otherwise, it would be more difficult to find $\mathbf{e}_{I^C}$, thus the whole solution $\mathbf{e}$. The cost in binary operations of the algorithm is given by the product of the single iteration cost by the average number of necessary iterations to find the SDP solution. Such a number is the reciprocal of the success probability of one iteration, which is uniquely determined by the supposed error distribution. The more likely the distribution, the higher the success probability. For this reason, the main differences between ISD algorithms are the supposed error distribution and the ways of executing one single iteration. Usually, the main term of the cost expression is the average number of repetitions, so the first proposed improvements on ISD algorithms aimed to reduce the number of iterations.

Before presenting the main ISD algorithms, we briefly introduce some computation techniques that may be used to reduce the cost of operations.

- **Early Abort.** In some of the algorithms a computation is carried out and the result of this computation will determine the execution of the following computations. The condition is usually that the weight of the result cannot exceed a certain value. This condition can be checked either by computing the result and then checking its weight or by computing one result entry at a time and immediately checking the weight of the partial result obtained so far. With this second strategy, as soon as the weight exceeds the maximum value we can stop the computation, thus saving on cost. We refer to this as *early abort* technique.

  For example, let us consider an algorithm that computes $\mathbf{xA}$, for $\mathbf{x} \in \mathbb{F}_q^k$, $wt_H(\mathbf{x}) = t$ and $\mathbf{A} \in \mathbb{F}_q^{k\times n}$, and then proceeds if $wt_H(\mathbf{xA}) = w$. The naive strategy would require

$$nt\left(\lceil\log_2(q)\rceil + \lceil\log_2(q)\rceil^2\right) \tag{3.2.1}$$

binary operations. However, with early abort we compute one entry at a time. Assuming it is uniformly distributed over $\mathbb{F}_q$, the probability that this entry contributes to the weight of the result vector is $\frac{q-1}{q}$. Thus, we expect the partial result vector to have a weight w on average after $\frac{q}{q-1}w$ computed entries. The computation stops when the weight becomes $w+1$, so on average after $\frac{q}{q-1}(w+1)$ entries. Considering that computing one entry costs $t\left(\lceil\log_2(q)\rceil + \lceil\log_2(q)\rceil^2\right)$, using early abort leads to a cost of

$$\frac{q}{q-1}(w+1)t\left(\lceil\log_2(q)\rceil + \lceil\log_2(q)\rceil^2\right), \tag{3.2.2}$$

which is clearly lower than (3.2.1) as soon as $\frac{q}{q-1}(w+1) < n$;

- **Intermediate Sums.** In some algorithms a computation has to be carried out for all vectors in a certain set, namely all the possible vectors of a certain weight. *Intermediate sums* technique does so by starting from performing the computation with lower weight vectors and using the results to carry out the computations for higher weight vectors.

  For example, let us consider an algorithm that has to compute $\mathbf{xA}$, with $\mathbf{A} \in \mathbb{F}_q^{k\times n}$, for all $\mathbf{x} \in \mathbb{F}_q^k$ of weight t. We could normally perform the multiplication for every possible vector, and that costs

$$\binom{k}{t}(q-1)^t nt\left(\lceil\log_2(q)\rceil + \lceil\log_2(q)\rceil^2\right) \tag{3.2.3}$$

  binary operations. However, using intermediate sums we start by multiplying $\mathbf{A}$ by all the possible vectors of weight 1, that is multiplying all the rows of the matrix by every non-zero element of $\mathbb{F}_q$. That costs $(q-1)kn\lceil\log_2(q)\rceil^2$ binary operations. We continue by multiplying $\mathbf{A}$ by all the possible vectors of weight 2, that is computing linear combinations of two multiples of rows of $\mathbf{A}$, obtained from the last step. This costs $\binom{k}{2}(q-1)^2 n\lceil\log_2(q)\rceil$. We then iterate this process until we get to weight t, obtaining a cost of

$$L_q(k,t)n\lceil\log_2(q)\rceil + (q-1)kn\lceil\log_2(q)\rceil^2 \tag{3.2.4}$$

binary operations, where we defined

$$L_q(k,t) = \sum_{i=2}^{t}\binom{k}{i}(q-1)^i.$$

This is clearly lower than (3.2.3) since $L_q(k,t) < \binom{k}{t}(q-1)^t t$.

3.2.1 Prange

Prange's algorithm [29] assumes that errors are outside of a generic information set I, that is $I \cap Supp(\mathbf{e}) = \emptyset$.

For the sake of clarity, we initially assume that the chosen information set is $I = \{1, .., k\}$. Let us bring the parity check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ in systematic form by multiplying it by an appropriate matrix $\mathbf{U} \in \mathbb{F}_q^{(n-k)\times(n-k)}$. We refer to this procedure as *Gaussian Elimination.* The assumed error distribution allows us to separate the $\mathbf{e}$ vector in two parts: $\mathbf{e} = \begin{pmatrix}\mathbf{0}_k & \mathbf{e}_{I^C}\end{pmatrix}$. The first part (where the chosen information set is) has no errors, while the second part has $wt_H(\mathbf{e}_{I^C}) = t$ errors. Parity check equations can be expressed in the following way:

$$\mathbf{e}\mathbf{H}^\top\mathbf{U}^\top = \begin{pmatrix}\mathbf{0}_k & \mathbf{e}_{I^C}\end{pmatrix}\begin{pmatrix}\mathbf{A}^\top \\ \mathbf{Id}_{n-k}\end{pmatrix} = \mathbf{s}\mathbf{U}^\top$$

Therefore it means that $\mathbf{e}_{I^C} = \mathbf{s}\mathbf{U}^\top$. If the supposed error distribution in respect to the information set I is right, it follows that $wt_H(\mathbf{s}\mathbf{U}^\top) = t$. So, to verify the $\mathbf{e}_{I^C}$ we found is valid, it is enough to check that $\mathbf{s}\mathbf{U}^\top$ has weight t.

Algorithm 1 shows Prange's algorithm for a generic information set. In practice, to randomly choose an information set means finding a permutation matrix $\mathbf{P} \in \mathbb{F}_q^{n\times n}$ such as $\mathbf{HP} = \begin{pmatrix}\mathbf{A} & \mathbf{B}\end{pmatrix}$, with $\mathbf{A} \in \mathbb{F}_q^{(n-k)\times k}$ and $\mathbf{B} \in \mathbb{F}_q^{(n-k)\times(n-k)}$, which permutes the columns of $\mathbf{H}$ and, accordingly, the entries of $\mathbf{e}$. The parity check equations do not change due to this procedure, as permutation matrices are orthogonal ($\mathbf{H}^\top = \mathbf{H}^{-1}$). Finally, $\mathbf{U}$ is computed as $\mathbf{B}^{-1}$ and used to bring $\mathbf{H}$ into systematic form, as usual:

$$\mathbf{e}\mathbf{H}^\top\mathbf{U}^\top = (\mathbf{eP})\mathbf{P}^\top\mathbf{H}^\top\mathbf{U}^\top = \begin{pmatrix}\mathbf{0}_k & \mathbf{e}_{I^C}\end{pmatrix}\begin{pmatrix}\mathbf{A}^\top \\ \mathbf{Id}_{n-k}\end{pmatrix} = \mathbf{s}\mathbf{U}^\top.$$

This procedure will be implied throughout the rest of the thesis.

Algorithm 1 Prange's algorithm over $\mathbb{F}_q$ for SDP

```
Input: $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_q^{(n-k)}$, $0 < t \leq n$.
Output: $\mathbf{e} \in \mathbb{F}_q^n$ with $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$ and $wt_H(\mathbf{e}) = t$.
1: Choose an information set $I \subset \{1, ..., n\}$ of size $k$;
2: Compute $\mathbf{U} \in \mathbb{F}_q^{(n-k)\times(n-k)}$ such that
       $(\mathbf{UH})_I = \mathbf{A}$ e $(\mathbf{UH})_{I^C} = \mathbf{Id}_{n-k}$,
   where $\mathbf{A} \in \mathbb{F}_q^{(n-k)\times k}$;
3: Compute $\mathbf{s}' = \mathbf{s}\mathbf{U}^\top$;
4: if $wt_H(\mathbf{s}') = t$ then
5:     Return $\mathbf{e}$, with $\mathbf{e}_I = \mathbf{0}_k$ and $\mathbf{e}_{I^C} = \mathbf{s}'$;
6: end if
7: Restart from Step 1 by choosing another $I$.
```

The cost in binary operations of Prange's algorithm is shown in Theorem 1.

Theorem 1. *Prange's algorithm over $\mathbb{F}_q$ for SDP requires on average*

$$\binom{n-k}{t}^{-1}\binom{n}{t}(n-k)^2(n+1)\left(\lceil\log_2(q)\rceil+\lceil\log_2(q)\rceil^2\right)$$

binary operations.

Proof. Computations for a single iteration basically consist on bringing the $\mathbf{H}$ matrix into systematic form and transforming the $\mathbf{s}$ syndrome accordingly. Therefore, the iteration cost is given by the computation cost of $\mathbf{UH}$ e $\mathbf{Us}^\top$, that is

$$(n-k)^2(n+1)\left(\lceil\log_2(q)\rceil+\lceil\log_2(q)\rceil^2\right)$$

binary operations. As said before, the success probability of an iteration depends on the probability of choosing the correct error distribution, that is the correct distribution of the weight of $\mathbf{e}$. In this case, we assume that no errors occur in the information set, so for a given I this probability is

$$\frac{\#\mathbf{e}\text{: } wt_H(\mathbf{e})=t \text{ without errors in } I}{\#\ \mathbf{e} \text{ used in construction}}=\frac{\#\mathbf{e}\text{: } wt_H(\mathbf{e})=t \text{ without errors in } I}{\#\ \mathbf{e} \text{ with weight } t}=$$
$$=\frac{\binom{n-k}{t}(q-1)^t}{\binom{n}{t}(q-1)^t}=\frac{\binom{n-k}{t}}{\binom{n}{t}},$$

where it has implicitly been implied that SDP solution is unique. The average number of iterations is given by the reciprocal of this probability, that is

$$\binom{n-k}{t}^{-1}\binom{n}{t}.$$

□

In the cost expression, the main term is the average number of required iterations since it is binomial (exponential), while the other terms are polynomial. For this reason, improvements over Prange's algorithm aim to reduce the number of repetitions by assuming more probable error distributions.

3.2.2 Lee-Brickell

Lee-Brickell's algorithm [24] improves the error distribution assumption used in Prange to try to increase the success probability of one iteration, so that less of them are required on average. To do this, the algorithm uses a more likely error distribution assumption. In fact, Lee-Brickell allows for v errors in the information set, that is $|I \cap Supp(\mathbf{e})| = v$.

For the sake of clarity, we initially assume that the chosen information set is $I = \{1, .., k\}$. Let us bring the parity check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ in systematic form by multiplying it by an appropriate matrix $\mathbf{U} \in \mathbb{F}_q^{(n-k)\times(n-k)}$ (Gaussian Elimination). Let us split the error vector $\mathbf{e}$ in two parts: $\mathbf{e} = \begin{pmatrix}\mathbf{e}_1 & \mathbf{e}_2\end{pmatrix}$, with $\mathbf{e}_1 \in \mathbb{F}_q^k$ and $\mathbf{e}_2 \in \mathbb{F}_q^{(n-k)}$. The first part has $wt_H(\mathbf{e}_1) = v$ (where the information set is), while the second part has $wt_H(\mathbf{e}_2) = t$ errors. Parity check equations can be expressed in the following way:

$$\mathbf{e}\mathbf{H}^\top\mathbf{U}^\top = \begin{pmatrix}\mathbf{e}_1 & \mathbf{e}_2\end{pmatrix}\begin{pmatrix}\mathbf{A}^\top \\ \mathbf{Id}_{n-k}\end{pmatrix} = \mathbf{e}_1\mathbf{A}^\top + \mathbf{e}_2 = \mathbf{s}\mathbf{U}^\top$$

Considering the assumed weight distribution, for every possible $\mathbf{e}_1$ of weight v we compute $\mathbf{e}_2 = \mathbf{s}\mathbf{U}^\top - \mathbf{e}_1\mathbf{A}^\top$. If $\mathbf{e}_2$ has weight $t - v$, the SDP solution vector $\mathbf{e} = \begin{pmatrix}\mathbf{e}_1 & \mathbf{e}_2\end{pmatrix}$ of weight t has been found. Algorithm 2 shows Lee-Brickell's algorithm for a generic information set.

The cost in binary operations of Lee-Brickell's algorithm is shown in Theorem 2.

Algorithm 2 Lee-Brickell's algorithm over $\mathbb{F}_q$ for SDP

```
Input: $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_q^{(n-k)}$, $0 < t \leq n$, $\max\{0, k+t-n\} \leq v \leq \min\{k, t\}$.
Output: $\mathbf{e} \in \mathbb{F}_q^n$ with $\mathbf{eH}^\top = \mathbf{s}$ and $wt_H(\mathbf{e}) = t$.
 1: Choose an information set $I \subset \{1, ..., n\}$ of size $k$;
 2: Compute $\mathbf{U} \in \mathbb{F}_q^{(n-k)\times(n-k)}$ such that
        $(\mathbf{UH})_I = \mathbf{A}$ e $(\mathbf{UH})_{I^C} = \mathbf{Id}_{n-k}$,
    where $\mathbf{A} \in \mathbb{F}_q^{(n-k)\times k}$;
 3: Build a list $\mathcal{L} = \{\mathbf{e}_1 | wt_H(\mathbf{e}_1) = v\}$;
 4: Compute $\mathbf{s}' = \mathbf{sU}^\top$;
 5: for $\mathbf{e}_1 \in \mathcal{L}$ do
 6:     Compute $\mathbf{e}_2 = \mathbf{s}' - \mathbf{e}_1\mathbf{A}^\top$;
 7:     if $wt_H(\mathbf{e}_2) = t - v$ then
 8:         Return $\mathbf{e}$, with $\mathbf{e}_I = \mathbf{e}_1$ ed $\mathbf{e}_{I^C} = \mathbf{e}_2$;
 9:     end if
10: end for
11: Restart from Step 1 by choosing another $I$.
```

Theorem 2. *Lee-Brickell's algorithm over $\mathbb{F}_q$ for SDP requires on average*

$$\binom{n-k}{t-v}^{-1}\binom{k}{v}^{-1}\binom{n}{t}\Bigg\{(n-k)^2(n+1)\left(\lceil\log_2(q)\rceil + \lceil\log_2(q)\rceil^2\right) + \\ + \binom{k}{v}(q-1)^v\left[(n-k)k\left(\lceil\log_2(q)\rceil + \lceil\log_2(q)\rceil^2\right) + (n-k)\lceil\log_2(q)\rceil\right]\Bigg\}$$

binary operations.

Proof. One Lee-Brickell iteration is identical to a Prange one, with the addition of the computing of $\mathbf{e}_2 = \mathbf{sU}^\top - \mathbf{e}_1\mathbf{A}^\top$ for every element of the list $\mathcal{L}$. Therefore, the following term has to be added to the cost of one iteration already shown in Theorem 1:

$$\binom{k}{v}(q-1)^v\left[(n-k)\,k\left(\lceil\log_2(q)\rceil + \lceil\log_2(q)\rceil^2\right) + (n-k)\lceil\log_2(q)\rceil\right],$$

where the two sum terms respectively represent the cost of the two computations $\mathbf{e}_1\mathbf{A}^\top$ and $\mathbf{s}\mathbf{U}^\top - \mathbf{e}_1\mathbf{A}^\top$. For a given I and assuming unique SDP solution, the success probability of one iteration is

$$\frac{\#\mathbf{e}\colon\ wt_H(\mathbf{e}) = t \text{ with } v \text{ errors in } I}{\#\ \mathbf{e} \text{ used in construction}} = \frac{\#\mathbf{e}\colon\ wt_H(\mathbf{e}) = t \text{ with } v \text{ errors in } I}{\#\ \mathbf{e} \text{ with weight } t} =$$
$$= \frac{\binom{n-k}{t-v}\binom{k}{v}(q-1)^t}{\binom{n}{t}(q-1)^t} = \frac{\binom{n-k}{t-v}\binom{k}{v}}{\binom{n}{t}},$$

whose reciprocal represents the average number of iterations. □

It is clear that the single Lee-Brickell iteration requires more binary operation than the Prange one, but the average number of iterations is lower. The latter being the main term, in general we get a lower computational cost.

3.2.3 Stern

Stern's algorithm [30] uses the same error distribution of Lee-Brickell, but instead of requiring weight v in the information set, it partitions I and requires weight $v/2$ in each half. Moreover, the information set can be joined with another set Z of size ℓ called *zero window*. Its name derives from its early use, as no error was allowed in it. Anyway, it is more convenient to join Z with the information set and use ℓ as an extra optimization parameter, also because by doing so the assumption on the weight distribution is less strict and, therefore, more likely [19]. In total, we allow for v errors in $I' = I \cup Z$ and $t - v$ errors in $J = (I \cup Z)^C$.

For the sake of clarity, let us assume $I' = \{1, ..., k+\ell\}$ and $J = \{k+\ell+1, ..., n\}$. As usual, we bring the parity check matrix $\mathbf{H}$ in systematic form by multiplying it by an appropriate matrix $\mathbf{U}$:

$$\mathbf{UH} = \begin{pmatrix} \mathbf{A} & \mathbf{0}_{\ell\times(n-k-\ell)} \\ \mathbf{B} & \mathbf{Id}_{n-k-l} \end{pmatrix}$$

with $\mathbf{A} \in \mathbb{F}_q^{\ell\times(k+\ell)}$ and $\mathbf{B} \in \mathbb{F}_q^{(n-k-\ell)\times(k+\ell)}$. We refer to this procedure as *Partial Gaussian Elimination (PGE)*, because the result is not completely systematic. Then we split the error vector in two parts, one corresponding to I' and the other to J: $\mathbf{e} = \begin{pmatrix}\mathbf{e}_{I'} & \mathbf{e}_J\end{pmatrix}$, where $wt_H(\mathbf{e}_{I'}) = v$ and $wt_H(\mathbf{e}_J) = t - v$. Parity check equations can be expressed as follows:

$$\mathbf{eH}^\top\mathbf{U}^\top = \begin{pmatrix}\mathbf{e}_{I'} & \mathbf{e}_J\end{pmatrix}\begin{pmatrix} \mathbf{A}^\top & \mathbf{B}^\top \\ \mathbf{0}_{(n-k-\ell)\times\ell} & \mathbf{Id}_{n-k-\ell} \end{pmatrix} = \begin{pmatrix}\mathbf{s}_1 & \mathbf{s}_2\end{pmatrix} = \mathbf{sU}^\top,$$

with $\mathbf{A} \in \mathbb{F}_q^{\ell\times(k+\ell)}$, $\mathbf{B} \in \mathbb{F}_q^{(n-k-\ell)\times(k+\ell)}$, $\mathbf{s}_1 \in \mathbb{F}_q^{\ell}$ e $\mathbf{s}_2 \in \mathbb{F}_q^{(n-k-\ell)}$. This leads to:

$$\mathbf{e}_{I'}\mathbf{A}^\top = \mathbf{s}_1 \tag{3.2.5}$$

$$\mathbf{e}_{I'}\mathbf{B}^\top + \mathbf{e}_J = \mathbf{s}_2 \tag{3.2.6}$$

(3.2.7)

As a result of the use of the zero window, we get two SDP instances of reduced size. More specifically, the (3.2.5) is referred to as the *small instance*. This is due to the fact that we can literally solve it as a smaller size SDP instance and then use the result to compute the whole solution though the (3.2.6).

To solve the small instance, let us partition I' in two subsets $X = \{1, ..., (k+\ell)/2\}$ and $Y = \{(k+\ell)/2 + 1, ..., k+\ell\}$, where $k+\ell$ is assumed to be even for the sake of clarity. With a little abuse of notation in order to simplify the narrative, we define $\mathbf{e}_X \in \mathbb{F}_q^{k+\ell}(X)$ and $\mathbf{e}_Y \in \mathbb{F}_q^{k+\ell}(Y)$. By doing so, it is possible to write $\mathbf{e}_{I'} = \mathbf{e}_X + \mathbf{e}_Y$, obtaining the (3.2.5) as:

$$\mathbf{e}_X \mathbf{A}^\top = \mathbf{s}_1 - \mathbf{e}_Y \mathbf{A}^\top. \tag{3.2.8}$$

Therefore, it is necessary to find the $\mathbf{e}_X$ and $\mathbf{e}_Y$ that verify the (3.2.8). To do that, we build two lists:

$$\mathcal{S} = \left\{ \left(\mathbf{e}_X, \mathbf{e}_X \mathbf{A}^\top \right) \mid \mathbf{e}_X \in \mathbb{F}_q^{k+\ell}(X), wt_H(\mathbf{e}_X) = v/2 \right\}$$
$$\mathcal{T} = \left\{ \left(\mathbf{e}_Y, \mathbf{s}_1 - \mathbf{e}_Y \mathbf{A}^\top \right) \mid \mathbf{e}_Y \in \mathbb{F}_q^{k+\ell}(Y), wt_H(\mathbf{e}_Y) = v/2 \right\}$$

We then perform a *collision search* between the two lists, that is finding the pair of tuples $\left(\mathbf{e}_X, \mathbf{a}\right) \in \mathcal{S}$ and $\left(\mathbf{e}_Y, \mathbf{b}\right) \in \mathcal{T}$ for which $\mathbf{a} = \mathbf{b}$. By construction of the lists, those $\mathbf{e}_X$ ed $\mathbf{e}_Y$ will satisfy the (3.2.8).

Once a collision is found, we compute $\mathbf{e}_I = \mathbf{e}_X + \mathbf{e}_Y$. Through the (3.2.6) then we compute $\mathbf{e}_J$ and, if the result has weight $t - v$, a valid error vector $\mathbf{e}$ is found. Algorithm 3 shows Stern's algorithm for generic information set and zero window.

Algorithm 3 Stern's algorithm over $\mathbb{F}_q$ for SDP

```
Input: $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_q^{(n-k)}$, $0 < t \leq n$, $0 \leq \ell < n-k$ and
        $\max\{0, k+\ell+t-n\} \leq v \leq \min\{k+\ell, t\}$.
Output: $\mathbf{e} \in \mathbb{F}_q^n$ with $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$ and $wt_H(\mathbf{e}) = t$.
 1: Choose an information set $I \subset \{1, ..., n\}$ of size $k$ and a zero window $Z \subset I^C$ of
    size $\ell$ and define $I' = I \cup Z$ and $J = (I \cup Z)^C$;
 2: Partition $I'$ in $X$ of size $\lceil \frac{k+\ell}{2} \rceil$ and $Y$ of size $\lfloor \frac{k+\ell}{2} \rfloor$;
 3: Compute $\mathbf{U} \in \mathbb{F}_q^{(n-k)\times(n-k)}$ such as
        $$(\mathbf{UH})_{I'} = \begin{pmatrix} \mathbf{A} \\ \mathbf{B} \end{pmatrix}, \text{ and } (\mathbf{UH})_J = \begin{pmatrix} \mathbf{0}_{l\times(n-k-\ell)} \\ \mathbf{Id}_{(n-k-\ell)} \end{pmatrix},$$
    where $\mathbf{A} \in \mathbb{F}_q^{\ell\times(k+\ell)}$ and $\mathbf{B} \in \mathbb{F}_q^{(n-k-\ell)\times(k+\ell)}$;
 4: Compute $\mathbf{s}\mathbf{U}^\top = \begin{pmatrix} \mathbf{s}_1 & \mathbf{s}_2 \end{pmatrix}$, where $\mathbf{s}_1 \in \mathbb{F}_q^\ell$ and $\mathbf{s}_2 \in \mathbb{F}_q^{(n-k-\ell)}$;
 5: Build the list $\mathcal{S} = \{\left(\mathbf{e}_X, \mathbf{e}_X\mathbf{A}^\top\right) \mid \mathbf{e}_X \in \mathbb{F}_q^{(k+\ell)}(X), wt_H(\mathbf{e}_X) = \lceil v/2 \rceil\}$;
 6: Build the list $\mathcal{T} = \{\left(\mathbf{e}_Y, \mathbf{s}_1 - \mathbf{e}_Y\mathbf{A}^\top\right) \mid \mathbf{e}_Y \in \mathbb{F}_q^{(k+\ell)}(Y), wt_H(\mathbf{e}_Y) = \lfloor v/2 \rfloor\}$;
 7: for $\left(\mathbf{e}_X, \mathbf{a}\right) \in \mathcal{S}$ do
 8:     for $\left(\mathbf{e}_Y, \mathbf{a}\right) \in \mathcal{T}$ do
 9:         if $wt_H(\mathbf{s}_2 - (\mathbf{e}_X + \mathbf{e}_Y)\mathbf{B}^\top) = t - v$ then
10:             Return $\mathbf{e} = \begin{pmatrix} \mathbf{e}_{I'} & \mathbf{e}_J \end{pmatrix}$, with $\mathbf{e}_{I'} = \mathbf{e}_X + \mathbf{e}_Y$ and
            $\mathbf{e}_J = \mathbf{s}_2 - (\mathbf{e}_X + \mathbf{e}_Y)\mathbf{B}^\top$;
11:         end if
12:     end for
13: end for
14: Restart from Step 1 by choosing other $I$ and $Z$.
```

Theorem 3. *Stern's algorithm requires on average*

$$\binom{(k+\ell)/2}{v/2}^{-2}\binom{n-k-\ell}{t-v}^{-1}\binom{n}{t}\cdot$$
$$\cdot\Big[(n-k)^2(n+1)\left(\lceil\log_2(q)\rceil+\lceil\log_2(q)\rceil^2\right)+(q-1)(k+\ell)\ell\lceil\log_2(q)\rceil^2+$$
$$+\ell\left(2L_q\left(\frac{k+\ell}{2},\frac{v}{2}\right)+\binom{(k+\ell)/2}{v/2}(q-1)^{v/2}\right)\lceil\log_2(q)\rceil+$$
$$+\frac{\binom{(k+\ell)/2}{v/2}^2(q-1)^v}{q^\ell}\min\left\{n-k-\ell,\frac{q}{q-1}(t-v+1)\right\}\cdot$$
$$\cdot v\left(\lceil\log_2(q)\rceil+\lceil\log_2(q)\rceil^2\right)\Big]$$

binary operations (for the sake of clarity, v and $k+\ell$ are assumed even).

Proof. As usual, bringing $\mathbf{H}$ in systematic form (and consequently transform $\mathbf{s}$) has a cost of

$$(n-k)^2(n+1)\left(\lceil\log_2(q)\rceil+\lceil\log_2(q)\rceil^2\right)$$

binary operations.

To build the list $\mathcal{S}$, in order to reduce the number of operations, *intermediate sums* technique can be used. We have to compute $\mathbf{e}_X\mathbf{A}^\top$ for every $\mathbf{e}_X\in\mathbb{F}_q^{(k+\ell)}(X)$ of weight $v/2$. According to (3.2.4), that costs

$$L_q\left(\frac{k+\ell}{2},\frac{v}{2}\right)\ell\lceil\log_2(q)\rceil+(q-1)\left(\frac{k+\ell}{2}\right)\ell\lceil\log_2(q)\rceil^2$$

binary operations.

Analogously, to find list $\mathcal{T}$, that is calculating $\mathbf{s}_1-\mathbf{e}_Y\mathbf{A}^\top$ for every $\mathbf{e}_Y\in\mathbb{F}_q^{(k+\ell)}(Y)$ of weight $v/2$, using intermediate sums technique the number of needed binary operation is

$$L_q\left(\frac{k+\ell}{2},\frac{v}{2}\right)\ell\left\lceil\log_2(q)\right\rceil+(q-1)\left(\frac{k+\ell}{2}\right)\ell\left\lceil\log_2(q)\right\rceil^2+$$
$$+\binom{(k+\ell)/2}{v/2}(q-1)^{v/2}\ell\left\lceil\log_2(q)\right\rceil,$$

where the first two terms are the cost of $\mathbf{e}_Y\mathbf{A}^\top$ and the third is the cost of $\mathbf{s}_1$ subtraction.

Once $\mathcal{S}$ and $\mathcal{T}$ are built, we need to iterate through the lists. Therefore, the cost of the following operations should be multiplied for $|\mathcal{S}||\mathcal{T}|=\binom{(k+\ell)/2}{v/2}^2(q-1)^v$ but, since before proceeding with the algorithm we look for collisions, we have to multiply by the average number of collisions. Since the vectors we compare ($\mathbf{e}_X\mathbf{A}^\top$ and $\mathbf{s}_1-\mathbf{e}_Y\mathbf{A}^\top$) belong to $\mathbb{F}_q^\ell$ and we assume they are uniformly distributed, such a number is $\frac{\binom{(k+\ell)/2}{v/2}^2(q-1)^v}{q^\ell}$. For every collision we compute $\mathbf{s}_2-(\mathbf{e}_X+\mathbf{e}_Y)\mathbf{B}^\top$ and we check the weight to be $t-v$, so we can use *early abort* technique. For calculating a single entry $v\left(\left\lceil\log_2(q)\right\rceil+\left\lceil\log_2(q)\right\rceil^2\right)$ binary operations are needed, that become

$$min\left\{n-k-\ell,\frac{q}{q-1}(t-v+1)\right\}v\left(\left\lceil\log_2(q)\right\rceil+\left\lceil\log_2(q)\right\rceil^2\right),$$

with the early abort. The minimum is needed for taking into account the possibility that the limit imposed by early abort could be higher than the size of the vector $(n-k-\ell)$.

Finally, the average number of required iterations is the reciprocal of the probability of choosing the correct error distribution of $\mathbf{e}$, that is

$$\binom{(k+\ell)/2}{v/2}^2 \binom{n-k-\ell}{t-v} \binom{n}{t}^{-1}.$$

$\square$

It is worth noticing that the main terms of the cost of Stern are the exponential ones, which are

$$\binom{(k+\ell)/2}{v/2}(q-1)^{v/2} = |\mathcal{S}| = |\mathcal{T}|,$$
$$\frac{\binom{(k+\ell)/2}{v/2}^2 (q-1)^v}{q^\ell} = \frac{|\mathcal{S}| \cdot |\mathcal{T}|}{q^\ell}.$$

Clearly, the list sizes deeply affect the cost of the algorithm. Therefore, finding clever ways to build them to get smaller lists could be beneficial. Finally, notice that also the term $L_q\left(\frac{k+\ell}{2}, \frac{v}{2}\right)$ is exponential, but the main term of the sum is equal to the list size, which has already been considered.

3.2.4 MMT

In Stern, the solution of the small instance $\mathbf{e}_{I'}$ is obtained by concatenating its two halves $\mathbf{e}_X$ and $\mathbf{e}_Y$. Thus, there is only one tuple of vectors $\Big(\mathbf{e}_X, \mathbf{e}_Y\Big)$ that gives the solution. Then, we could think of ways of increasing this number of couples. This would allow us to consider just a portion of the lists so that one representative couple is left with a high probability. This process could even imply having bigger initial lists, but if the list reduction is strong enough this could lead to smaller list sizes than the one used in Stern. This is the basic idea of the so-called *representation technique* [22]. In more detail, this technique considers couples of vectors of weight

$v/2$ of full length, i.e. $k+\ell$, instead of half-length as in Stern. This allows for multiple representative couples (or *representations*) to exist. For example, let us consider $q=2$ for the sake of clarity and $\mathbf{e}_{I'} = \big(10100101\big)$. With Stern, the only representation $\Big(\mathbf{e}_X, \mathbf{e}_Y\Big)$ we have is:

$\mathcal{S}$	$\mathcal{T}$
$\mathbf{e}_X = \big(1010\big)$	$\mathbf{e}_Y = \big(0101\big)$

On the other hand, with the representation technique, we have $r = \binom{v}{v/2}$ representations:

$\mathcal{S}$	$\mathcal{T}$
$\mathbf{e}_X = \big(10100000\big)$	$\mathbf{e}_Y = \big(00000101\big)$
$\mathbf{e}_X = \big(10000100\big)$	$\mathbf{e}_Y = \big(00100001\big)$
$\mathbf{e}_X = \big(10000001\big)$	$\mathbf{e}_Y = \big(00100100\big)$
$\mathbf{e}_X = \big(00100100\big)$	$\mathbf{e}_Y = \big(10000001\big)$
$\mathbf{e}_X = \big(00100001\big)$	$\mathbf{e}_Y = \big(10000100\big)$
$\mathbf{e}_X = \big(00000101\big)$	$\mathbf{e}_Y = \big(10100000\big)$

Clearly, this means that now the list sizes are higher, that is $\binom{k+\ell}{v/2}$ instead of $\binom{(k+\ell)/2}{v/2}$, but as said before we are just considering a portion of the initial lists $\mathcal{S}$ and $\mathcal{T}$, respectively $\mathcal{L}_1$ and $\mathcal{L}_2$. To do that, we first have to consider that we want one representation to survive the reduction with high probability, that is having $P\left\{\left(\mathbf{e}_X, \mathbf{e}_Y\right) \in (\mathcal{L}_1 \times \mathcal{L}_2)\right\} = \frac{1}{r}$ for every representation $\left(\mathbf{e}_X, \mathbf{e}_Y\right)$. Indeed, the probability of having one of the many representations in $(\mathcal{L}_1 \times \mathcal{L}_2)$ would be almost 1 (almost because their probabilities are not statistically independent from one another). Given that the goal is to find a solution for the small instance, a clever way for reducing the lists could be partially enforcing the parity check equations of the small instance, that is:

$$\begin{aligned} \mathcal{L}_1 &= \left\{\left(\mathbf{e}_X, \mathbf{e}_X\mathbf{A}^\top\right) \mid \mathbf{e}_X \in \mathbb{F}_2^{(k+\ell)}, wt_H(\mathbf{e}_X) = v/2, \left(\mathbf{e}_X\mathbf{A}^\top\right)_{[u]} = \mathbf{t}_1\right\} \\ \mathcal{L}_2 &= \left\{\left(\mathbf{e}_Y, \mathbf{e}_Y\mathbf{A}^\top\right) \mid \mathbf{e}_Y \in \mathbb{F}_2^{(k+\ell)}, wt_H(\mathbf{e}_Y) = v/2, \left(\mathbf{e}_Y\mathbf{A}^\top\right)_{[u]} = \mathbf{t}_2\right\}, \end{aligned} \tag{3.2.9}$$

where $\mathbf{t}_1 \in \mathbb{F}_2^u$ is a random target vector for $\mathcal{L}_1$ and $\mathbf{t}_2 = (\mathbf{s}_1)_{[u]} - \mathbf{t}_1$ is a target vector for $\mathcal{L}_2$. Notice that for every $\mathbf{e}_X \in \mathcal{L}_1$ and $\mathbf{e}_Y \in \mathcal{L}_2$ the vector $\mathbf{e}_X + \mathbf{e}_Y$ satisfies the parity-check equations in the first u positions. The probability of the representation $\mathbf{e}_X$ surviving in $\mathcal{L}_1$ is $P\{\mathbf{e}_X \in \mathcal{L}_1\} = 2^{-u}$ and if we choose $u = \lceil \log_2(r) \rceil$ it becomes (approximately) $\frac{1}{r}$. Because of the interdependence between the two list buildings that enforce $(\mathbf{e}_X + \mathbf{e}_Y)\mathbf{A}^\top = \mathbf{s}_1$ in the first u positions, the probability of the representation $\mathbf{e}_Y$ surviving in $\mathcal{L}_2$ is $P\{\mathbf{e}_Y \in \mathcal{L}_2 | \mathbf{e}_X \in \mathcal{L}_1\} = 1$. Thus, $P\left\{\left(\mathbf{e}_X, \mathbf{e}_Y\right) \in (\mathcal{L}_1 \times \mathcal{L}_2)\right\} = P\{\mathbf{e}_X \in \mathcal{L}_1\} P\{\mathbf{e}_Y \in \mathcal{L}_2 | \mathbf{e}_X \in \mathcal{L}_1\} = \frac{1}{r}$ as desired. Moreover, $\mathbf{A}$ is random and, consequently, also $\left(\mathbf{e}_X\mathbf{A}^\top\right)_{[u]}$ and $\left(\mathbf{e}_Y\mathbf{A}^\top\right)_{[u]}$ are random. Thus, this reduction technique, called *Wagner algorithm*, leads to lists of (expected) size

$$\binom{k+\ell}{v/2} r^{-1} = \binom{k+\ell}{v/2}\binom{v}{v/2}^{-1}$$

which is usually less than the list sizes in Stern. Once $\mathcal{L}_1$ and $\mathcal{L}_2$ are built, we have to merge them in order to find a final list $\mathcal{L}$ that only contains solutions of the complete small instance. To do so, Algorithm 4 is used, where $u' = \ell$, $w = v$ and $\mathbf{t} = \mathbf{s}_1$. Obviously, by construction every pair of $\big(\mathbf{x}_i, \mathbf{y}_i\big) \in (\mathcal{L}_1 \times \mathcal{L}_2)$ already satisfies the parity check equations in the first $u = \lceil \log_2(r) \rceil$ positions, then the Merge algorithm actually only checks the remaining $\ell - u$ positions. We call *representation merge* the process of joining two lists using representations and we write that as $\mathcal{L} = \mathcal{L}_1 \bowtie_{[u']} \mathcal{L}_2$, with u' being the length of the target vector.

Algorithm 4 Merge algorithm over $\mathbb{F}_2$

```
Input: Lists $\mathcal{L}_1$ and $\mathcal{L}_2$, $0 < u < \ell$, $0 \leq w \leq k + \ell$, $\mathbf{A} \in \mathbb{F}_2^{\ell \times (k+\ell)}$ and target
        vector $\mathbf{t} \in \mathbb{F}_2^{u'}$.
Output: $\mathcal{L} = \mathcal{L}_1 \bowtie_{[u']} \mathcal{L}_2$, with final weight $w$ and $u'$ enforced positions.
1: Lexicographically sort $\mathcal{L}_1$ and $\mathcal{L}_2$ according to $(\mathbf{x}_i\mathbf{A}^\top)_{[u']}$ for $\mathbf{x}_i \in \mathcal{L}_1$ and
   $(\mathbf{y}_i\mathbf{A}^\top)_{[u']} + \mathbf{t}$ for $\mathbf{y}_i \in \mathcal{L}_2$, respectively;
2: for $\big(\mathbf{x}_i, \mathbf{y}_i\big) \in (\mathcal{L}_1 \times \mathcal{L}_2)$ with $(\mathbf{x}_i\mathbf{A}^\top)_{[u']} = (\mathbf{y}_i\mathbf{A}^\top)_{[u']} + \mathbf{t}$ do
3:     if $wt_H(\mathbf{x}_i + \mathbf{y}_i) = w$ then $\mathcal{L} = \mathcal{L} \cup \{\mathbf{x}_i + \mathbf{y}_i\}$;
4:     end if
5: end for
6: Return $\mathcal{L}$.
```

Lemma 5. *Merge algorithm over $\mathbb{F}_2$ requires on average*

$$(L_1 + L_2)u'(k + \ell) + L_1 \log_2(L_1) + L_2 \log_2(L_2) + (k + \ell)(L_1 L_2 2^{-u'})$$

binary operations, where $L_i = |\mathcal{L}_i|$ for $i = 1, 2$.

Proof. The first term is the cost of $(\mathbf{x}_i\mathbf{A}^\top)_{[u']}$ and $(\mathbf{y}_j\mathbf{A}^\top)_{[u']}$ computations for lists $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively.

The second and third terms are the cost of sorting the two lists.

The last term is the cost of $\mathbf{x}_i + \mathbf{y}_j$ computation multiplied by the average number of collisions $L_1 L_2 2^{-u'}$. $\square$

The final list $\mathcal{L}$ doesn't have to be stored as every vector in it is checked for the full instance on the fly, so it doesn't play a role in the cost expression. Its size is

$$L = |\mathcal{L}| = \min\left\{\frac{L_1 L_2}{2^{\ell-u}}, \binom{k+\ell}{v}2^{-\ell}\right\} = \min\left\{\frac{\binom{k+\ell}{v/2}^2 2^{-2u}}{2^{\ell-u}}, \binom{k+\ell}{v}2^{-\ell}\right\} =$$
$$= \min\left\{\binom{k+\ell}{v/2}^2 2^{-(\ell+u)}, \binom{k+\ell}{v}2^{-\ell}\right\}$$

Still, an open question is left. How do we build the lists $\mathcal{L}_1$ and $\mathcal{L}_2$? The naive way would be to enforce the parity check equations on u positions to every vector with weight $v/2$, but, clearly, this is not optimal from a cost point of view. Instead, we could iterate the process we used to build $\mathcal{L}$, thus obtaining a binary tree structure. By showing the level in the tree with a superscript, let us define $v^{(0)} = v$. Equations (3.2.9) become

$$\mathcal{L}_1^{(1)} = \left\{\left(\mathbf{e}_1^{(1)}, \mathbf{e}_1^{(1)}\mathbf{A}^\top\right) \mid \mathbf{e}_1^{(1)} \in \mathbb{F}_2^{(k+\ell)}, wt_H(\mathbf{e}_1^{(1)}) = v^{(1)}, \left(\mathbf{e}_1^{(1)}\mathbf{A}^\top\right)_{[u^{(1)}]} = \mathbf{t}_1^{(1)}\right\}$$
$$\mathcal{L}_2^{(1)} = \left\{\left(\mathbf{e}_2^{(1)}, \mathbf{e}_2^{(1)}\mathbf{A}^\top\right) \mid \mathbf{e}_2^{(1)} \in \mathbb{F}_2^{(k+\ell)}, wt_H(\mathbf{e}_2^{(1)}) = v^{(1)}, \left(\mathbf{e}_2^{(1)}\mathbf{A}^\top\right)_{[u^{(1)}]} = \mathbf{t}_2^{(1)}\right\},$$

where we defined

$$v^{(1)} = v^{(0)}/2 = v/2$$
$$u^{(1)} = \left\lceil \log_2\left(r^{(1)}\right)\right\rceil = \left\lceil \log_2\left(\binom{v^{(0)}}{v^{(1)}}\right)\right\rceil = \left\lceil \log_2\left(\binom{v}{v/2}\right)\right\rceil$$

and the target vectors as

$$\mathbf{t}_1^{(1)} \xleftarrow{\$} \mathbb{F}_q^{u^{(1)}}$$
$$\mathbf{t}_2^{(1)} = (\mathbf{s}_1)_{[u^{(1)}]} - \mathbf{t}_1^{(1)}.$$

Repeating the process means building $\mathcal{L}_1^{(1)}$ by creating two upper-level lists $\mathcal{L}_1^{(2)}$ and $\mathcal{L}_2^{(2)}$:

$$\mathcal{L}_1^{(2)} = \left\{ \left(\mathbf{e}_1^{(2)}, \mathbf{e}_1^{(2)}\mathbf{A}^\top\right) \mid \mathbf{e}_1^{(2)} \in \mathbb{F}_2^{(k+\ell)}, wt_H(\mathbf{e}_1^{(2)}) = v^{(2)}, \left(\mathbf{e}_1^{(2)}\mathbf{A}^\top\right)_{[u^{(2)}]} = \mathbf{t}_1^{(2)} \right\}$$
$$\mathcal{L}_2^{(2)} = \left\{ \left(\mathbf{e}_2^{(2)}, \mathbf{e}_2^{(2)}\mathbf{A}^\top\right) \mid \mathbf{e}_2^{(2)} \in \mathbb{F}_2^{(k+\ell)}, wt_H(\mathbf{e}_2^{(2)}) = v^{(2)}, \left(\mathbf{e}_2^{(2)}\mathbf{A}^\top\right)_{[u^{(2)}]} = \mathbf{t}_2^{(2)} \right\},$$

where we defined

$$v^{(2)} = v^{(1)}/2 = v/4$$
$$u^{(2)} = \lceil \log_2 (r^{(2)}) \rceil = \left\lceil \log_2 \left(\binom{v^{(1)}}{v^{(2)}} \right) \right\rceil = \left\lceil \log_2 \left(\binom{v/2}{v/4} \right) \right\rceil$$

and the target vectors as

$$\mathbf{t}_1^{(2)} \xleftarrow{\$} \mathbb{F}_q^{u^{(2)}}$$
$$\mathbf{t}_2^{(2)} = (\mathbf{t}_1^{(1)})_{[u^{(2)}]} - \mathbf{t}_1^{(2)}.$$

The new lists are then given as input to the Merge algorithm where $u' = u^{(1)}$, $w = v^{(1)}$ and $\mathbf{t} = \mathbf{t}_1^{(1)}$. The same procedure is applied to $\mathcal{L}_2^{(1)}$. We build two upper-level lists $\mathcal{L}_3^{(2)}$ and $\mathcal{L}_4^{(2)}$:

$$\mathcal{L}_3^{(2)} = \left\{ \left(\mathbf{e}_3^{(2)}, \mathbf{e}_3^{(2)}\mathbf{A}^\top\right) \mid \mathbf{e}_3^{(2)} \in \mathbb{F}_2^{(k+\ell)}, wt_H(\mathbf{e}_3^{(2)}) = v^{(2)}, \left(\mathbf{e}_3^{(2)}\mathbf{A}^\top\right)_{[u^{(2)}]} = \mathbf{t}_3^{(2)} \right\}$$
$$\mathcal{L}_4^{(2)} = \left\{ \left(\mathbf{e}_4^{(2)}, \mathbf{e}_4^{(2)}\mathbf{A}^\top\right) \mid \mathbf{e}_4^{(2)} \in \mathbb{F}_2^{(k+\ell)}, wt_H(\mathbf{e}_4^{(2)}) = v^{(2)}, \left(\mathbf{e}_4^{(2)}\mathbf{A}^\top\right)_{[u^{(2)}]} = \mathbf{t}_4^{(2)} \right\},$$

where the target vectors are

$$\mathbf{t}_3^{(2)} \xleftarrow{\$} \mathbb{F}_q^{u^{(2)}}$$
$$\mathbf{t}_4^{(2)} = (\mathbf{s}_1)_{[u^{(2)}]} - \mathbf{t}_3^{(2)}.$$

The new lists are then given as input to the Merge algorithm where $u' = u^{(1)}$, $w = v^{(1)}$ and $\mathbf{t} = \mathbf{t}_2^{(1)}$.

This process can be potentially iterated infinite times, but the problem of building the top-level lists remains unsolved. Let us assume we stop at the second layer, we have to find an efficient way of building $\mathcal{L}_i^{(2)}$ for $i = 1, 2, 3, 4$. We can use a partition approach as we did with Stern: for each $\mathcal{L}_i^{(2)}$, we build two upper-level lists, called *base lists* as they will be the ones from where the whole tree starts, where each list contains vectors of half the length and half the weight. Then we merge pairs of base lists by concatenating their vectors and enforcing the parity check equation in the first $u^{(2)}$ positions. We refer to this process as *concatenation merge* and we write that as $\mathcal{L}_i^{(2)} = \mathcal{B}_{2i-1} \cup_{[u^{(2)}]} \mathcal{B}_{2i}$ for $i = 1, 2, 3, 4$. In total we need 8 base lists:

$$\mathcal{B}_i = \left\{ \mathbf{e}_i^{(b)} \in \mathbb{F}_2^{k+\ell}(P_i) \mid wt_H(\mathbf{e}_i^{(b)}) = v^{(2)}/2 \right\},$$

for $i = 1, ..., 8$, where $P_{2i} \cup P_{2i+1} = \{1, ..., k + \ell\}$ and partitions P_{2i} and P_{2i+1} have sizes $\left\lceil \frac{k+\ell}{2} \right\rceil$ and $\left\lfloor \frac{k+\ell}{2} \right\rfloor$, respectively. Notice that all even-indexed base lists are the same and all odd-indexed base lists are the same, thus we only need two base lists. This doesn't mean that all the concatenation merges will have the same output, as the randomly picked target vectors used in each merge are supposedly always different. Defining $B = |\mathcal{B}_i|$, $L_2 = |\mathcal{L}_i^{(2)}|$, $L_1 = |\mathcal{L}_i^{(1)}|$ and $L = |\mathcal{L}|$, the list sizes at every level are:

$$\begin{aligned}
B &= \binom{(k+\ell)/2}{v^{(2)}/2} \\
L_2 &= \min\left\{B^2 \cdot 2^{-u^{(2)}}, \binom{k+\ell}{v^{(2)}} 2^{-u^{(2)}}\right\} = B^2 \cdot 2^{-u^{(2)}} \\
L_1 &= \min\left\{L_2^2 \cdot 2^{-\left(u^{(1)}-u^{(2)}\right)}, \binom{k+\ell}{v^{(1)}} 2^{-u^{(1)}}\right\} \\
L &= \min\left\{L_1^2 \cdot 2^{-\left(\ell-u^{(1)}\right)}, \binom{k+\ell}{v^{(0)}} 2^{-\ell}\right\}
\end{aligned}$$

This algorithm has been proposed by May, Meurer and Thomae in [26] and is known as MMT. To be precise, the proposed algorithm uses only two levels, but the number of levels is just one of the parameters for which the algorithm can be optimized. We will not give the cost expression for MMT, because, as we will see, it will be easily obtainable from the cost of the next algorithm.

3.2.5 BJMM

So far, we have been considering only pairs of vectors with disjointed supports, representing 1 entries as $1 = 1+0 = 0+1$ and 0 entries as $0 = 0+0$. In other words, the zeros are "less easily" represented than the ones. To increase the number of ways through which we can get zeros, as we are considering vectors over $\mathbb{F}_2$ we could also represent 0s as $0 = 1+1$, thus considering pair of vectors whose supports overlap in some positions. For instance, the vector $\left(10100101\right)$ from previous examples can be represented with

$$\begin{aligned}&(10111000)+\\&(00011101)=\\&(10100101)\end{aligned}$$

where $\varepsilon = 2$ overlaps occur. Clearly, the weight of the vectors at every level grows to

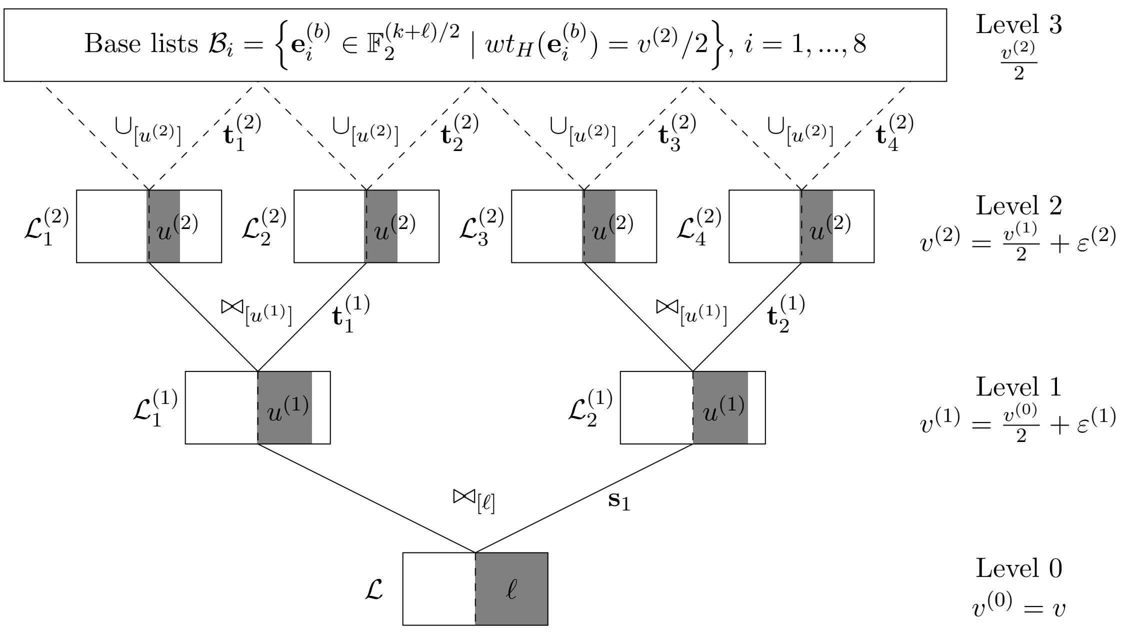
$$v^{(i)} = \frac{v^{(i-1)}}{2} + \varepsilon^{(i)},$$

but the number of representations also increases to

$$r^{(i)} = \binom{v^{(i-1)}}{\frac{v^{(i-1)}}{2}}\binom{k+\ell-v^{(i-1)}}{\varepsilon^{(i)}}.$$

For optimized parameters, this leads to smaller list sizes, which means a smaller cost. The algorithm that allows for overlapping supports is known as BJMM [8] and has been proposed by Becker, Joux, May and Meurer. Clearly, MMT is a special case of BJMM where $\varepsilon^{(i)} = 0$ in all levels. The authors proposed a 3-level tree as no improvements were obtained by using more levels. In Fig.3.1 the tree used in BJMM is illustrated.

Algorithm 5 shows the complete BJMM algorithm for a generic information set.



Base lists $\mathcal{B}_i = \left\{\mathbf{e}_i^{(b)} \in \mathbb{F}_2^{(k+\ell)/2} \mid wt_H(\mathbf{e}_i^{(b)}) = v^{(2)}/2\right\}$, $i = 1, ..., 8$
Level 3
$\frac{v^{(2)}}{2}$
$\cup_{[u^{(2)}]}$ $\mathbf{t}_1^{(2)}$
$\cup_{[u^{(2)}]}$ $\mathbf{t}_2^{(2)}$
$\cup_{[u^{(2)}]}$ $\mathbf{t}_3^{(2)}$
$\cup_{[u^{(2)}]}$ $\mathbf{t}_4^{(2)}$
$\mathcal{L}_1^{(2)}$ $u^{(2)}$
$\mathcal{L}_2^{(2)}$ $u^{(2)}$
$\mathcal{L}_3^{(2)}$ $u^{(2)}$
$\mathcal{L}_4^{(2)}$ $u^{(2)}$
Level 2
$v^{(2)} = \frac{v^{(1)}}{2} + \varepsilon^{(2)}$
$\bowtie_{[u^{(1)}]}$ $\mathbf{t}_1^{(1)}$
$\bowtie_{[u^{(1)}]}$ $\mathbf{t}_2^{(1)}$
$\mathcal{L}_1^{(1)}$ $u^{(1)}$
$\mathcal{L}_2^{(1)}$ $u^{(1)}$
Level 1
$v^{(1)} = \frac{v^{(0)}}{2} + \varepsilon^{(1)}$
$\bowtie_{[\ell]}$ $\mathbf{s}_1$
$\mathcal{L}$ ℓ
Level 0
$v^{(0)} = v$

Figure 3.1: Illustration of the BJMM tree.

Theorem 4. *BJMM algorithm over $\mathbb{F}_2$ requires on average*

$$\binom{n}{t}\binom{n-k-\ell}{t-v}^{-1}\binom{k+\ell}{v}^{-1} \cdot \Big[(n-k)^2(n+1)$$
$$+ 4\left(2Bu^{(2)}(k+\ell) + 2B\log_2(B) + (k+\ell)B^2 2^{-u^{(2)}}\right)$$
$$+ 2\left(2L_2u^{(1)}(k+\ell) + 2L_2\log_2(L_2) + (k+\ell)L_2^2 2^{-(u^{(1)}-u^{(2)})}\right)$$
$$+ \left(2L_1\ell(k+\ell) + 2L_1\log_2(L_1) + (k+\ell)L_1^2 2^{-(\ell-u^{(1)})}\right)$$
$$+ \binom{k+\ell}{v}2^{-\ell}\min\{n-k-\ell, 2(t-v+1)\}\,v\Big]$$

binary operations.

Proof. The multiplicative term is, as usual, the average number of required iterations.

The first term in the brackets is the Partial Gaussian Elimination cost.

The second, third and fourth terms are the cost of merges at the various levels, as shown in Lemma 5. Notice that at the second and third merge, the last sum term takes into account that the lists we are merging already satisfy the parity check equations over the first $u^{(2)}$ and $u^{(1)}$ positions, respectively.

The last term is the cost of $\mathbf{s}_2 - \mathbf{e}_1\mathbf{B}^\top$ computation using the early abort technique, $2(t-v+1)$, multiplied by the average number of collisions in the last merge, $\binom{k+\ell}{v}2^{-\ell}$ and by the cost of one entry of $\mathbf{e}_1\mathbf{B}^\top$ product, v. $\square$

BJMM algorithm can be generalized to the non-binary finite fields $(\mathbb{F}_q, q > 2)$. In this case, every element β of the field can be obtained as $\beta = \alpha + (\beta - \alpha), \forall \alpha \in \mathbb{F}_q$, thus the number of representations at every level generally grows. For example, let us consider the last merge (for the other merges the same observations apply). In Fig.3.2a we assume that all the $\varepsilon^{(1)}$ cancellations occur outside of the support of $\mathbf{e}$. Then, once we chose an appropriate $\mathbf{e}_1^{(1)}$, the elements of $\mathbf{e}_2^{(1)}$ corresponding to the support of $\mathbf{e}$ are fixed, thus we have $\binom{v^{(0)}}{v^{(0)}/2}$ ways of representing the support of $\mathbf{e}$. The elements outside the support that must cancel out can be any element in $\mathbb{F}_q$, thus we have $\binom{k+l-v^{(0)}}{\varepsilon^{(1)}}(q-1)^{\varepsilon^{(1)}}$ representations of the zeros of $\mathbf{e}$. The number of representations of $\mathbf{e}$ with the chosen assumption is

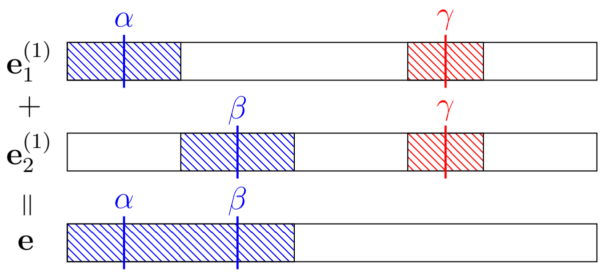
$$\binom{v^{(0)}}{\frac{v^{(0)}}{2}}\binom{k+l-v^{(0)}}{\varepsilon^{(1)}}(q-1)^{\varepsilon^{(1)}}.$$

Clearly, this is not the total number of representations. Considering Fig.3.2b, we can also have overlaps within the support of $\mathbf{e}$. These overlaps must not result in cancellations anymore, but they have to result in the correct entry of $\mathbf{e}$. Thus, considering also these cases, the total number of representations is

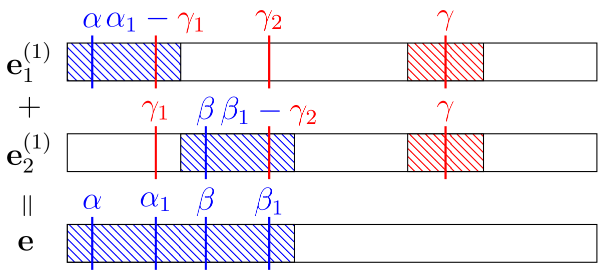
$$r^{(1)} = \sum_{i=0}^{\min(v^{(0)}/2, \varepsilon^{(1)})} \binom{v^{(0)} - 2i}{\frac{v^{(0)}}{2} - i}(q-2)^{2i}\binom{k+\ell-v^{(0)}}{\varepsilon^{(1)} - i}(q-1)^{\varepsilon^{(1)}-i}.$$

Algorithm 5 BJMM algorithm over $\mathbb{F}_2$

```
Input: $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_2^{n-k}$, $0 < t \leq n$, $0 \leq \ell < n-k$,
        $\max\{0, k+\ell+t-n\} \leq v \leq \min\{k+\ell, t\}$,
        $0 \leq \varepsilon^{(1)} \leq k+\ell-v$ and $0 \leq \varepsilon^{(2)} \leq k+\ell-v/2-\varepsilon^{(1)}$.
Output: $\mathbf{e} \in \mathbb{F}_2^n$ with $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$ and $wt_H(\mathbf{e}) = t$.
 1: Choose an information set $I \subset \{1, ..., n\}$ of size $k$ and a zero window $Z \subset I^C$ of
    size $\ell$ and define $I' = I \cup Z$ and $J = (I \cup Z)^C$;
 2: Compute $\mathbf{U} \in \mathbb{F}_q^{(n-k)\times(n-k)}$ such as
        $$(\mathbf{UH})_{I'} = \begin{pmatrix} \mathbf{A} \\ \mathbf{B} \end{pmatrix}, \text{ and } (\mathbf{UH})_J = \begin{pmatrix} \mathbf{0}_{l\times(n-k-\ell)} \\ \mathbf{Id}_{(n-k-\ell)} \end{pmatrix},$$
    where $\mathbf{A} \in \mathbb{F}_q^{\ell\times(k+\ell)}$ and $\mathbf{B} \in \mathbb{F}_q^{(n-k-\ell)\times(k+\ell)}$;
 3: Compute $\mathbf{s}\mathbf{U}^\top = (\mathbf{s}_1 \ \mathbf{s}_2)$, where $\mathbf{s}_1 \in \mathbb{F}_q^\ell$ and $\mathbf{s}_2 \in \mathbb{F}_q^{n-k-\ell}$;
 4: Choose two partitions $P_1, P_2$ of $\{1, ..., k+\ell\}$ of sizes $\lceil \frac{k+l}{2} \rceil$ and $\lfloor \frac{k+l}{2} \rfloor$, respectively;
 5: Define
        $$\mathcal{B}_i = \{\mathbf{x} \in \mathbb{F}_2^{k+\ell}(P_i) | wt_H(\mathbf{x}) = v^{(2)}/2\}$$
    per $i \in \{1, 2\}$;
 6: Choose $\mathbf{t}_1^{(1)} \xleftarrow{\$} \mathbb{F}_2^{u^{(1)}}$, define $\mathbf{t}_2^{(1)} = (\mathbf{s}_1)_{[u^{(1)}]} - \mathbf{t}_1^{(1)}$;
 7: Choose $\mathbf{t}_1^{(2)}, \mathbf{t}_3^{(2)} \xleftarrow{\$} \mathbb{F}_2^{u^{(2)}}$, define $\mathbf{t}_2^{(2)} = (\mathbf{t}_1^{(1)})_{[u^{(2)}]} - \mathbf{t}_1^{(2)}$ and $\mathbf{t}_4^{(2)} = (\mathbf{s}_1)_{[u^{(2)}]} - \mathbf{t}_3^{(2)}$;
 8: for $i \in \{1, ..., 4\}$ do
 9:     Compute $\mathcal{L}_i^{(2)} = \mathcal{B}_1 \bowtie_{[u^{(2)}]} \mathcal{B}_2$ using Merge algorithm to get weight $v^{(2)}$ and
        target vectors $\mathbf{t}_i^{(2)}$;
10: end for
11: for $i \in \{1, 2\}$ do
12:     Compute $\mathcal{L}_i^{(1)} = \mathcal{L}_{2i-1}^{(2)} \bowtie_{[u^{(1)}]} \mathcal{L}_{2i}^{(2)}$ using Merge algorithm to get weight $v^{(1)}$
        and target vectors $\mathbf{t}_i^{(1)}$;
13: end for
14: Compute $\mathcal{L} = \mathcal{L}_1^{(1)} \bowtie_{[\ell]} \mathcal{L}_2^{(1)}$ using Merge algorithm to get weight $v$ and target
    vector $\mathbf{s}_1$;
15: for $\mathbf{e}_1 \in \mathcal{L}$ do
16:     if $wt_H(\mathbf{s}_2 - \mathbf{e}_1\mathbf{B}^\top) = t - v$ then
17:         Return $\mathbf{e} = (\mathbf{e}_1 \ \mathbf{e}_2)$, with $\mathbf{e}_2 = \mathbf{s}_2 - \mathbf{e}_1\mathbf{B}^\top$;
18:     end if
19: end for
20: Restart from Step 1 by choosing other $I$ and $Z$.
```



(a) Overlap only outside of the support of $\mathbf{e}$



(b) Overlap also in the support of $\mathbf{e}$

Figure 3.2: Possible cases for representations of $\mathbf{e}$

Being the $u^{(1)}$ and $u^{(2)}$ parameters the logarithm of the number of representations, the reduction of the list sizes at every merge is bigger than the binary case. Although, this does not lead in general to smaller lists, because the base lists are bigger than the ones in the binary case. List sizes in non-binary BJMM are:

$$\begin{aligned}
B &= \binom{(k+\ell)/2}{v^{(2)}/2}(q-1)^{v^{(2)}/2}\\
L_2 &= \min\left\{B^2\cdot q^{-u^{(2)}}, \binom{k+\ell}{v^{(2)}}q^{-u^{(2)}}(q-1)^{v^{(2)}}\right\} = B^2\cdot q^{-u^{(2)}}\\
L_1 &= \min\left\{L_2^2\cdot q^{-\left(u^{(1)}-u^{(2)}\right)}, \binom{k+\ell}{v^{(1)}}q^{-u^{(1)}}(q-1)^{v^{(1)}}\right\}\\
L &= \min\left\{L_1^2\cdot q^{-\left(\ell-u^{(1)}\right)}, \binom{k+\ell}{v^{(0)}}q^{-\ell}(q-1)^{v^{(0)}}\right\}
\end{aligned}$$

Lemma 6. *Merge algorithm over $\mathbb{F}_q$ requires on average*

$$
\begin{aligned}
&(L_1+L_2)u'(k+\ell)\left(\lceil\log_2(q)\rceil+\lceil\log_2(q)\rceil^2\right)+\\
&+L_1\log(L_1)+L_2\log_2(L_2)+(k+\ell)(L_1L_2q^{-u'})\lceil\log_2(q)\rceil
\end{aligned}
$$

binary operations, where $L_i=|\mathcal{L}_i|$ for $i=1,2$.

Theorem 5. *BJMM algorithm over $\mathbb{F}_q$ requires on average*

$$
\begin{aligned}
&\binom{n}{t}\binom{n-k-\ell}{t-v}^{-1}\binom{k+\ell}{v}^{-1}\cdot\Big[(n-k)^2(n+1)\left(\lceil\log_2(q)\rceil+\lceil\log_2(q)\rceil^2\right)+\\
&+4\Big(2Bu^{(2)}(k+\ell)\left(\lceil\log_2(q)\rceil+\lceil\log_2(q)\rceil^2\right)+2B\log_2(B)+(k+\ell)B^2q^{-u^{(2)}}\lceil\log_2(q)\rceil\Big)+\\
&+2\Big(2L_2u^{(1)}(k+\ell)\left(\lceil\log_2(q)\rceil+\lceil\log_2(q)\rceil^2\right)+2L_2\log_2(L_2)+(k+\ell)L_2^2q^{-(u^{(1)}-u^{(2)})}\lceil\log_2(q)\rceil\Big)+\\
&+\Big(2L_1\ell(k+\ell)\left(\lceil\log_2(q)\rceil+\lceil\log_2(q)\rceil^2\right)+2L_1\log_2(L_1)+(k+\ell)L_1^2q^{-(\ell-u^{(1)})}\lceil\log_2(q)\rceil\Big)+\\
&+\binom{k+\ell}{v}(q-1)^vq^{-\ell}\min\left\{n-k-\ell,\frac{q}{q-1}(t-v+1)\right\}v\lceil\log_2(q)\rceil^2\Big]
\end{aligned}
$$

binary operations.

3.2.6 Algorithm comparison

As already said, the security level of an SDP-based scheme depends on the best, i.e. less costly, decoding algorithm. However, the cost of an algorithm is heavily determined by the parameters chosen for the scheme. For example, the cost in binary operations of Prange's algorithm depends on parameters n, k, t and q, so we can refer to its cost as $c(n,k,t,q)$. The parameters k and t are implicitly functions of n, so let us define their relative values as

$$\begin{aligned} R &= \lim_{n\to\infty} \frac{k(n)}{n} \\ T &= \lim_{n\to\infty} \frac{t(n)}{n}, \end{aligned}$$

which are obviously bounded between 0 and 1. For large lengths n the cost can be approximated by a base-two exponential:

$$c(n,k,t,q) = \lim_{n\to\infty} 2^{nC(R,T,q)+o(n)}.$$

We refer to $c(n,k,t,q)$ as finite regime cost and to $C(R,T,q)$ as asymptotic cost. The latter is obtained from the finite regime cost as

$$C(R,T,q) = \lim_{n\to\infty} \frac{1}{n} \log_2(c(n,k,t,q)).$$

This cost still depends on the relative parameters, but as they are relative values between 0 and 1 they allow us to compare the asymptotic costs of different algorithms regardless of the actual code length n. In order to compare the asymptotic costs of two or more algorithms, one of the relative parameters, usually R or T, is taken as the independent variable. Then, for every possible value of this variable, the rest of the parameters are chosen in order to minimize the asymptotic cost. In this way, $C(R,T,q)$ becomes a one-variable function, and the comparison among the costs of different algorithms is possible for every value of the chosen independent variable.

In the next theorems, asymptotic costs of previously shown algorithms are given. Before doing that, we define some useful notation:

$$Q = \log_2(q) \qquad R = \lim_{n\to\infty} \frac{k(n)}{n} \qquad T = \lim_{n\to\infty} \frac{t(n)}{n}$$

$$L = \lim_{n\to\infty} \frac{\ell(n)}{n} \qquad E^{(i)} = \lim_{n\to\infty} \frac{\varepsilon^{(i)}(n)}{n} \qquad V = \lim_{n\to\infty} \frac{v(n)}{n}$$

$$V^{(i)} = \lim_{n\to\infty} \frac{v^{(i)}(n)}{n} \qquad U^{(i)} = Q \lim_{n\to\infty} \frac{u^{(i)}(n)}{n} = \lim_{n\to\infty} \frac{1}{n} \log_2(r^{(i)})$$

Theorem 6. *The asymptotic cost of Prange's algorithm for SDP is*

$$C(R,T,q) = h_2(T) - (1-R)h_2\left(\frac{T}{1-R}\right).$$

Theorem 7. *The asymptotic cost of Lee-Brickell's algorithm for SDP is*

$$C(R,T,V,q) = h_2(T) - (1-R)h_2\left(\frac{T-V}{1-R}\right) + V \cdot Q.$$

Theorem 8. *The asymptotic cost of Stern's algorithm for SDP is*

$$C(R,T,V,L,q) = N(R,T,V,L,q) + F(R,T,V,L,q),$$

where $N(R,T,V,L,q)$ is the asymptotic number of iterations, which is given by

$$N(R,T,V,L,q) = h_2(T) - (1-R-L)h_2\left(\frac{T-V}{1-R-L}\right) - (R+L)h_2\left(\frac{V}{R+L}\right)$$

and $F(R,T,V,L,q)$ denotes the asymptotic cost of one iteration, which is given by

$$\max\{\Sigma, 2\Sigma - L\cdot Q\}.$$

The asymptotic Stern's list size is

$$\Sigma = \lim_{n\to\infty}\frac{1}{n}\log_2|\mathcal{S}| = \frac{R+L}{2}h_2\left(\frac{V}{R+L}\right) + \frac{V}{2}Q.$$

Theorem 9. *The asymptotic cost of BJMM algorithm for SDP over $\mathbb{F}_2$ is*

$$C(R,T,V,L) = N(R,T,V,L) + F(R,T,V,L),$$

where $N(R,T,V,L)$ denotes the asymptotic number of iterations, which is given by

$$h_2(T) - (1-R-L)h_2\left(\frac{T-V}{1-R-L}\right) - (R+L)h_2\left(\frac{V}{R+L}\right)$$

and $F(R,T,V,L)$ denotes the asymptotic cost of one iteration, which is given by

$$\max\left\{\Lambda_B, 2\Lambda_B - U^{(2)}, 2\Lambda_2 - (U^{(1)} - U^{(2)}), 2\Lambda_1 - (L - U^{(1)})\right\}.$$

The asymptotic BJMM list sizes are

$$\begin{aligned}
\Lambda_B &= \lim_{n\to\infty}\frac{1}{n}\log_2|B| = \frac{R+L}{2}h_2\left(\frac{V^{(2)}}{R+L}\right)\\
\Lambda_2 &= \lim_{n\to\infty}\frac{1}{n}\log_2|L_2| = 2\Lambda_B - U^{(2)}\\
\Lambda_1 &= \lim_{n\to\infty}\frac{1}{n}\log_2|L_1| = (R+L)h_2\left(\frac{V^{(1)}}{R+L}\right) - U^{(1)}
\end{aligned}$$

where, for $i \in \{1,2\}$ *and* $V^{(0)} = V$,

$$U^{(i)} = V^{(i-1)} + \left(R + L - V^{(i-1)}\right) h_2 \left(\frac{E^{(i)}}{R + L - V^{(i-1)}}\right)$$

and relative weights are

$$V^{(i)} = \frac{V^{(i-1)}}{2} + E^{(i)}.$$

Notice that, for the sake of clarity, to compute Λ_1, only the second term of the minimum in L_1 has been considered. This represents an upper bound of L_1, but for high values of n it is likely to be a very close approximation. Moreover, the asymptotic cost for MMT can be easily obtained from Theorem 9 by choosing $E^{(1)} = E^{(2)} = 0$.

Theorem 10. *The asymptotic cost of BJMM algorithm for SDP over* $\mathbb{F}_q$, $q > 2$ *is*

$$C(R,T,V,L,q) = N(R,T,V,L,q) + F(R,T,V,L,q),$$

where $N(R,T,V,L,q)$ *denotes the asymptotic number of iterations, which is given by*

$$h_2(T) - (1 - R - L) h_2 \left(\frac{T - V}{1 - R - L}\right) - (R + L) h_2 \left(\frac{V}{R + L}\right),$$

and $F(R,T,V,L,q)$ *denotes the asymptotic cost of one iteration, which is given by*

$$\max\left\{\Lambda_B, 2\Lambda_B - U^{(2)}, 2\Lambda_2 - (U^{(1)} - U^{(2)}), 2\Lambda_1 - (Q \cdot L - U^{(1)})\right\}.$$

The asymptotic BJMM list sizes are

$$\begin{aligned}
\Lambda_B &= \lim_{n\to\infty}\frac{1}{n}\log_2|B| = \frac{R+L}{2}h_2\left(\frac{V^{(2)}}{R+L}\right) + Q\frac{V^{(2)}}{2}\\
\Lambda_2 &= \lim_{n\to\infty}\frac{1}{n}\log_2|L_2| = 2\Lambda_B - U^{(2)}\\
\Lambda_1 &= \lim_{n\to\infty}\frac{1}{n}\log_2|L_1| = (R+L)h_2\left(\frac{V^{(1)}}{R+L}\right) - (U^{(1)} - Q\cdot V^{(1)})
\end{aligned}$$

where, for $i \in \{1,2\}$ and $V^{(0)} = V$,

$$\begin{aligned}
U^{(i)} &= \max_{J^{(i)}}\left\{Q\left(J^{(i)} + E^{(i)}\right) + 2^{V^{(i-1)}-2J^{(i)}} + (R+L-V^{(i-1)})h_2\left(\frac{E^{(i)} - J^{(i)}}{R+L-V^{(i-1)}}\right)\right\},\\
J^{(i)} &\in \left\{\lim_{n\to\infty}\frac{j}{n} \;\middle|\; j = 0,\dots,\min(v^{(i-1)}/2, \varepsilon^{(i)})\right\}
\end{aligned}$$

and relative weights are

$$V^{(i)} = \frac{V^{(i-1)}}{2} + E^{(i)}.$$

$$r^{(i)} \simeq q^{\frac{v^{(i-1)}}{2}+\varepsilon^{(i)}}\binom{k+\ell-v^{(i-1)}}{\varepsilon^{(i)} - v^{(i-1)}/2}$$

In Figure 3.3, the presented algorithms are compared with a field size of $q = 2$. Their costs are evaluated over different rates. The code is supposed to be random, so we can assume full-distance decoding, i.e. $T = \delta$ where δ is obtained from the GV bound with equality (1.0.7).

As expected, Prange is the worst algorithm. Notice that asymptotically there's no difference between Prange and Lee-Brickell, while every other algorithm exceeds Prange's performance. In particular, we notice that the use of the representation technique brings an improvement over Stern, as MMT outperforms it. Moreover, allowing for overlaps in the representation technique is convenient in $\mathbb{F}_2$, as BJMM

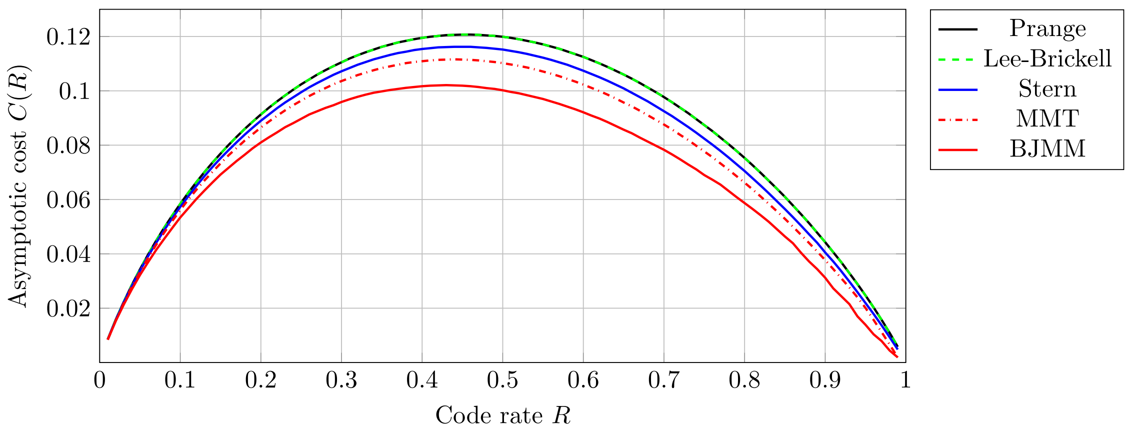


Figure 3.3: Asymptotic costs of ISD algorithms for SDP over $\mathbb{F}_2$.

costs less than MMT. However, in every case, the maximum cost is obtained for $R \simeq 0.45$.

The same analysis can be carried out for $q > 2$. Clearly, the asymptotic cost of each algorithm is higher than the binary case. While Stern still outperforms Prange, BJMM is not better than Stern anymore. The problem lies in the fact that increasing q leads to more representations, but, as a tradeoff, the number of vectors with a certain weight grows as well. While these two aspects play opposite roles in the size of the lists, the former wins. This leads to bigger list sizes and, therefore, bigger costs.

Chapter 4

Restricted SDP

Let us consider some subset $\mathbb{E}$ of $\mathbb{F}_q^\star$, denote by $\mathbb{E}_0 = \mathbb{E} \cup \{0\}$ and by

$$\mathscr{S}_t^{\mathbb{E}} := \{\mathbf{x} \in \mathbb{E}_0^n \mid \mathrm{wt}(\mathbf{x}) = t\}$$

the Hamming sphere with radius t and restriction $\mathbb{E}$. Clearly, for $\mathbb{E}$ of size z, we have $\mid \mathscr{S}_t^{\mathbb{E}} \mid = \binom{n}{t} z^t$. The Restricted Syndrome Decoding Problem (R-SDP), first introduced in [3], reads as follows.

Problem 3. Restricted Syndrome Decoding Problem (R-SDP)
Let $\mathbb{F}_q$ be a finite field of size q and $k \leq n$ be positive integers. Given $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_q^{(n-k)}$ and $t \in \mathbb{N}$, is there a vector $\mathbf{e} \in \mathscr{S}_t^{\mathbb{E}}$ such that $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$?

When $\mathbb{E} = \mathbb{F}_q^*$, the R-SDP corresponds to the canonical SDP. Consequently, it is not surprising that R-SDP is NP-complete for any choice of $\mathbb{E}$. The proof is essentially the same as in [3], where the authors focus on the case $\mathbb{E} = \{\pm x_1, \pm x_2, \cdots, \pm x_a\}$. From now on, we will consider the case

$$\mathbb{E} = \left\{g^j \pmod{q} \mid j \in \{0, 1, \cdots, z-1\}\right\},$$

where $g \in \mathbb{F}_q^*$ has multiplicative order $z = \mathrm{ord}(g) < q - 1$. In other words, we choose $\mathbb{E}$ as the cyclic subgroup of $\mathbb{F}_q^*$ which is generated by g and, to have $\mathbb{E} \neq \mathbb{F}_q^*$, we require that g is not primitive. This choice is made because the result of the multiplication of two restricted elements lies in $\mathbb{E}$ and it can be performed as a sum of exponents.

Analogously as we saw for SDP, we always consider that the R-SDP instance is chosen uniformly at random. We expect to have on average (at most) a unique solution if t is such that

$$\binom{n}{t} z^t q^{k-n} \leq 1. \tag{4.0.1}$$

Considering this asymptotically, we have for $T = t/n$ the condition

$$2^{n(H(T)+T\log_2(z)-(1-R)\log_2(q))} \leq 1,$$

which translates to

$$T\log_2(z) + H(T) - (1-R)\log_2(q) \leq 0.$$

Let T^* be the maximum value of T for which a random instance of R-SDP is expected to have a unique solution, that is

$$T^* = \max\left\{T \in [0;1] \mid T\log_2(z) + H(T) - (1-R)\log_2(q) \leq 0\right\}. \tag{4.0.2}$$

Comparing this to the condition in (3.0.1), we can see that with the R-SDP, we are allowed to choose a much larger weight t and still guarantee the uniqueness of the solution. Notice that if $\log_2(z) \leq (1-R)\log_2(q)$, we even have uniqueness for full-weight vectors. This case is of particular relevance, because as we will see, using full-weight solutions for R-SDP leads to smaller signature sizes with the proposed schemes. Indeed, in this setting permutation vectors are not needed for monomial transformations. Moreover, ISD complexity C grows with the weight, therefore we can obtain the same cost $2^{C\cdot n}$ even with smaller code sizes, thus saving in signature sizes.

4.1 ISD algorithms for R-SDP

We will now show the ISD algorithms that can be used in the restricted setting. As we previously saw for ISD over Hamming metric, the most promising ones are Stern and BJMM (over $\mathbb{F}_2$). For this reason, the study of new solvers for R-SDP will only take into account these two approaches.

To compare the computational complexity of R-SDP with classical SDP, we provide an adaption of the Stern algorithm, which works for any choice for $\mathbb{E}$. As we will see, there can be improvements, which depend specifically on the choice and structure of $\mathbb{E}$. Stern algorithm for R-SDP is basically the same as the one already presented for SDP, with two significant differences:

- The two lists now contain the vectors of weight $v/2$ with restricted entries:

$$\mathcal{S} = \left\{ \left(\mathbf{e}_X, \mathbf{e}_X \mathbf{A}^\top \right) \mid \mathbf{e}_X \in \mathbb{E}_0^{(k+\ell)}(X), wt_H(\mathbf{e}_X) = v/2 \right\},$$
$$\mathcal{T} = \left\{ \left(\mathbf{e}_Y, \mathbf{s}_1 - \mathbf{e}_Y \mathbf{A}^\top \right) \mid \mathbf{e}_Y \in \mathbb{E}_0^{(k+\ell)}(Y), wt_H(\mathbf{e}_Y) = v/2 \right\},$$

  where X and Y are partition subsets of $I' = I \cup Z$;

- After solving the small instance, we have to check that the rest of the solution vector has restricted entries, that is

$$\mathbf{s}_2 - (\mathbf{e}_X + \mathbf{e}_Y)\mathbf{B}^\top \in \mathbb{E}_0^{n-k-l}.$$

For the sake of completeness, Stern's algorithm for R-SDP is shown in Algorithm 6.

Algorithm 6 Stern's algorithm over $\mathbb{F}_q$ for R-SDP

```
Input: $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_q^{(n-k)}$, $0 < t \leq n$, $0 \leq \ell < n-k$ and
        $\max\{0, k+\ell+t-n\} \leq v \leq \min\{k+\ell, t\}$.
Output: $\mathbf{e} \in \mathbb{E}_0^n$ with $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$ and $wt_H(\mathbf{e}) = t$.
 1: Choose an information set $I \subset \{1, ..., n\}$ of size $k$ and a zero window $Z \subset I^C$ of
    size $\ell$ and define $I' = I \cup Z$ and $J = (I \cup Z)^C$;
 2: Partition $I'$ in $X$ of size $\lceil \frac{k+\ell}{2} \rceil$ and $Y$ of size $\lfloor \frac{k+\ell}{2} \rfloor$;
 3: Compute $\mathbf{U} \in \mathbb{F}_q^{(n-k)\times(n-k)}$ such as
        $$(\mathbf{UH})_{I'} = \begin{pmatrix} \mathbf{A} \\ \mathbf{B} \end{pmatrix}, \text{ and } (\mathbf{UH})_J = \begin{pmatrix} \mathbf{0}_{l\times(n-k-\ell)} \\ \mathbf{Id}_{(n-k-\ell)} \end{pmatrix},$$
    where $\mathbf{A} \in \mathbb{F}_q^{\ell\times(k+\ell)}$ and $\mathbf{B} \in \mathbb{F}_q^{(n-k-\ell)\times(k+\ell)}$;
 4: Compute $\mathbf{s}\mathbf{U}^\top = \begin{pmatrix} \mathbf{s}_1 & \mathbf{s}_2 \end{pmatrix}$, where $\mathbf{s}_1 \in \mathbb{F}_q^\ell$ and $\mathbf{s}_2 \in \mathbb{F}_q^{(n-k-\ell)}$;
 5: Build the list $\mathcal{S} = \{\left(\mathbf{e}_X, \mathbf{e}_X\mathbf{A}^\top\right) \mid \mathbf{e}_X \in \mathbb{E}_0^{(k+\ell)}(X), wt_H(\mathbf{e}_X) = \lceil v/2 \rceil\}$;
 6: Build the list $\mathcal{T} = \{\left(\mathbf{e}_Y, \mathbf{s}_1 - \mathbf{e}_Y\mathbf{A}^\top\right) \mid \mathbf{e}_Y \in \mathbb{E}_0^{(k+\ell)}(Y), wt_H(\mathbf{e}_Y) = \lfloor v/2 \rfloor\}$;
 7: for $\left(\mathbf{e}_X, \mathbf{a}\right) \in \mathcal{S}$ do
 8:     for $\left(\mathbf{e}_Y, \mathbf{a}\right) \in \mathcal{T}$ do
 9:         Compute $\widetilde{\mathbf{e}}_J = \mathbf{s}_2 - (\mathbf{e}_X + \mathbf{e}_Y)\mathbf{B}^\top$;
10:         if $wt_H(\widetilde{\mathbf{e}}_J) = t - v$ and $\widetilde{\mathbf{e}}_J \in \mathbb{E}_0^{n-k-l}$ then
11:             Return $\mathbf{e} = \begin{pmatrix} \mathbf{e}_{I'} & \mathbf{e}_J \end{pmatrix}$, with $\mathbf{e}_{I'} = \mathbf{e}_X + \mathbf{e}_Y$ and $\mathbf{e}_J = \widetilde{\mathbf{e}}_J$;
12:         end if
13:     end for
14: end for
15: Restart from Step 1 by choosing other $I$ and $Z$.
```

Since the size of the two lists is now $\binom{(k+\ell)/2}{v/2} z^{v/2}$ and the average number of collisions is $\binom{(k+\ell)/2}{v/2}^2 z^v q^\ell$, the asymptotic cost of Stern for R-SDP is the one stated in Theorem 11, where $Z = \log_2(z)$.

Theorem 11. *The asymptotic cost of Stern's algorithm for R-SDP is*

$$C(R,T,V,L,q,z) = N(R,T,V,L,q) + \max\left\{\Sigma, 2\Sigma - L\cdot Q\right\},$$

where

$$N(R,T,V,L,q) = h_2(T) - (1-R-L)h_2\left(\frac{T-V}{1-R-L}\right) - (R+L)h_2\left(\frac{V}{R+L}\right)$$

is the asymptotic number of iterations and

$$\Sigma = \lim_{n\to\infty}\frac{1}{n}\log_2|\mathcal{S}| = (R+L)h_2\left(\frac{V}{R+L}\right) + \frac{V}{2}Z$$

is the asymptotic Stern's list size.

In Figure 4.1 we give the cost of Stern's algorithm for random R-SDP instances, where we choose $T = T^*$, i.e., the maximal weight that guarantees uniqueness. Note that the cost at the point $z = q-1$ corresponds to the cost of Stern on a random SDP instance and thus, we can see that R-SDP with $z < q-1$ has a much larger cost than the SDP with the same parameters q, n, R.

4.1.1 Representation technique for R-SDP

As we saw in the Hamming metric, the BJMM algorithm is very efficient over $\mathbb{F}_2$, but it is not easily generalizable over bigger field sizes. The challenge of using the representation technique for bigger q lies in picking the entries of vectors to be merged from a convenient *search space*. Convenient in the sense that it has to be

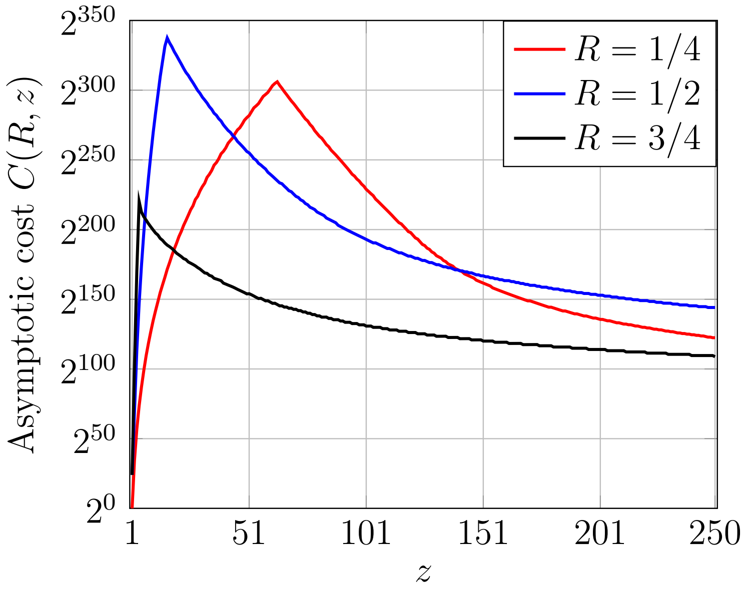


Figure 4.1: Cost of Stern's algorithm for random R-SDP instances with $q = 251$, $n = 256$ and several code rate values.

large enough to gain representations, but small enough to have reasonable list sizes. For the Hamming metric, when the solution $\mathbf{e}$ lives in the whole $\mathbb{F}_q^n$, it is difficult to properly design such search space, because we need every possible entry from $\mathbb{F}_q$. However, in the restricted setting the solution lives in $\mathbb{E}_0^n$, so the entries of the vectors resulting from merging are limited to a certain subset $\mathbb{E}$ of $\mathbb{F}_q$. Therefore, we could try to take a smaller search space than $\mathbb{F}_q$, whose elements are chosen in a smart way. For example, a possible search space could be $X = \mathbb{E}$. However, if $\mathbb{E}$ does not have much additive structure, i.e. there are not many elements y, $y' \in \mathbb{E}$ such that $y + y' \in \mathbb{E}$, the only ways of representing a non-zero entry $x \in \mathbb{E}$ will likely be $x = x + 0 = 0 + x$. With this choice, the number of representations would not be very big.

To get some fixed entry $x \in \mathbb{E}$ as $x = y + y'$, we could choose $y \in \mathbb{E}$ and $y' \in \mathbb{D} := \{a - b \mid a, b \in \mathbb{E}\} \setminus \{\pm\mathbb{E}_0\}$. If $\mathbb{E}$ has already a lot of additive structure, e.g. when z is even, then $\mathbb{D}$ becomes small. Thus, we only need a few additional elements

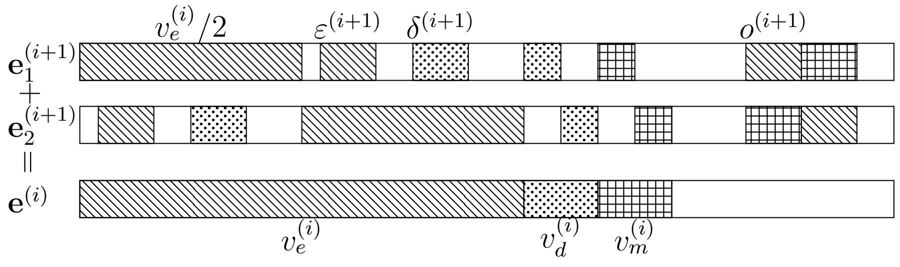


Figure 4.2: Counting the number of representations on level i.

in the search space to gain many representations for elements in $\mathbb{E}$. We propose the following search space $X = \mathbb{E} \cup \mathbb{D} \cup -\mathbb{E}$. On each level i, we are considering vectors $\mathbf{x}$ living in X_0, with $v_e^{(i)}$ entries in $\mathbb{E}$, $v_d^{(i)}$ entries in $\mathbb{D}$ and $v_m^{(i)}$ entries in $-\mathbb{E}$. From now on, we will refer to an a-level BJMM, i.e. 1 concatenation merge and $a-1$ representation merges, as BJMM(a).

To count the number of representations we use Figure 4.2. Note that, for the sake of clarity, some of the following notation letters might be used differently from the exposition of BJMM in Section 3.2. The representation merge at the i-th layer joins two vectors $\mathbf{e}_1^{(i+1)}$ and $\mathbf{e}_2^{(i+1)}$ with weight $v^{(i+1)}$ from the higher layer into a vector $\mathbf{e}^{(i)}$ with weight $v^{(i)}$. We denote by $\varepsilon^{(i+1)}$ the number of entries which are obtained through a $\mathbb{E}+\mathbb{E}$ representation. That is, for a fixed entry x of $\mathbf{e}^{(i)}$, we need to compute the number of possible $y \in \mathbb{E}$ that can reach x through addition with $\mathbb{E}$:

$$n_e(q, z, x) := \left|\{y \in \mathbb{E} \mid \exists y' \in \mathbb{E} : y + y' = x \in \mathbb{E}\}\right|.$$

We denote by $\delta^{(i+1)}$ the number of entries of $\mathbf{e}^{(i)}$ obtained through representations $\mathbb{E}+\mathbb{D}$. Hence, for a fixed entry x of $\mathbf{e}^{(i)}$, we need to compute the number of possible $y \in \mathbb{E}$ that can reach x through addition with $\mathbb{D}$:

$$n_d(q, z, x) := \left|\{y \in \mathbb{E} \mid \exists y' \in \mathbb{D} : y + y' = x \in \mathbb{E}\}\right|.$$

Since $n_e(q,z,x)$ and $n_d(q,z,x)$ are independent of x, we just write $n_e(q,z)$, $n_d(q,z)$. Finally, outside of the support of $\mathbf{e}^{(i)}$, we allow for $o^{(i+1)}$ representations of 0 as $0 = y + (-y) = (-y) + y$, for $y \in \mathbb{E}$. We could also allow for cancellations via $\mathbb{D}$, but as these entries are already only a few, they will be optimized to zero.

The vectors $\mathbf{e}_i^{(i+1)}$ have $v_e^{(i+1)} = v_e^{(i)}/2 + \varepsilon^{(i+1)} + o^{(i+1)}$ entries in $\mathbb{E}$, $v_d^{(i+1)} = v_d^{(i)}/2 + \delta^{(i+1)}$ in $\mathbb{D}$ and $v_m^{(i+1)} = v_m^{(i)}/2 + o^{(i+1)}$ in $-\mathbb{E}$. Hence, we get the number of representations

$$\begin{aligned} r^{(i)} =& \binom{v_e^{(i-1)}}{v_e^{(i-1)}/2} \left(\binom{v_e^{(i-1)}/2}{\delta^{(i)}, \varepsilon^{(i)}} n_d(q,z)^{\delta^{(i)}} n_e(q,z)^{\varepsilon^{(i)}} \right)^2 \\ &\cdot \binom{v_d^{(i-1)}}{v_d^{(i-1)}/2} \binom{v_m^{(i-1)}}{v_m^{(i-1)}/2} \binom{k+\ell - v_e^{(i-1)} - v_d^{(i-1)} - v_m^{(i-1)}}{o^{(i)}, o^{(i)}} z^{2o^{(i)}}. \end{aligned} \tag{4.1.1}$$

The amount of entries where the parity-check equations are enforced is then $u^{(i)} = \lceil \log_q r^{(i)} \rceil$.

After each merge, the obtained lists are filtered to get rid of vectors that are not well-formed. After the filtering, we are considering vectors in $S^{(i)}$ that have $v_e^{(i)}$ entries in $\mathbb{E}$, $v_d^{(i)}$ entries in $\mathbb{D}$ and $v_m^{(i)}$ entries in $-\mathbb{E}$. Hence,

$$\left| S^{(i)} \right| = \binom{k+\ell}{v_e^{(i)}, v_m^{(i)}, v_d^{(i)}} z^{v_e^{(i)} + v_m^{(i)}} \left| \mathbb{D} \right|^{v_d^{(i)}}.$$

To give the asymptotic cost, we need the following notation:

$$
\begin{array}{lll}
Q = \log_2(q) & V_e^{(i)} = \lim\limits_{n\to\infty} \frac{v_e^{(i)}(n)}{n} & N_e = \lim\limits_{n\to\infty} \frac{1}{n}\log_2(n_e(q,z)) \\
Z = \log_2(z) & V_m^{(i)} = \lim\limits_{n\to\infty} \frac{v_m^{(i)}(n)}{n} & N_d = \lim\limits_{n\to\infty} \frac{1}{n}\log_2(n_d(q,z)) \\
L = \lim\limits_{n\to\infty} \frac{\ell(n)}{n} & V_d^{(i)} = \lim\limits_{n\to\infty} \frac{v_d^{(i)}(n)}{n} & \Sigma^{(i)} = \lim\limits_{n\to\infty} \frac{1}{n}\log_2\left(\left|S^{(i)}\right|\right) \\
U^{(i)} = Q\lim\limits_{n\to\infty} \frac{u^{(i)}(n)}{n} & \Delta = \lim\limits_{n\to\infty} \frac{1}{n}\log_2(|\mathbb{D}|) & \\
D^{(i)} = \lim\limits_{n\to\infty} \frac{\delta^{(i)}(n)}{n} & E^{(i)} = \lim\limits_{n\to\infty} \frac{\varepsilon^{(i)}(n)}{n} & O^{(i)} = \lim\limits_{n\to\infty} \frac{o^{(i)}(n)}{n}
\end{array}
$$

Theorem 12. *The asymptotic cost of the presented BJMM*(3) *algorithm for R-SDP is*

$$C(R,T,V,L,q,z) = N(R,T,V,L,q,z) + F(R,T,V,L,q,z)$$

where $N(R,T,V,L,q,z)$ *denotes the asymptotic number of iterations, which is given by*

$$h_2(T) - (R+L)h_2\left(\tfrac{V}{R+L}\right) - (1-R-L)h_2\left(\tfrac{T-V}{1-R-L}\right)$$

and $F(R,T,V,L,q,z)$ *denotes the asymptotic cost of one iteration, which is given by*

$$\max\left\{\Sigma^{(2)}/2, \Sigma^{(2)} - U^{(2)}, 2\Sigma^{(2)} - U^{(2)} - U^{(1)}, 2\Sigma^{(1)} - U^{(1)} - LQ\right\},$$

where for $i \in \{1,2\}$ and $V_e^{(0)} = V$, $V_d^{(0)} = V_m^{(0)} = 0$ we set

$$\begin{aligned}
U^{(i)} &= R + L - R^{(i-1)} + R^{(i-1)} h_2\left(\tfrac{2O^{(i)}}{R^{(i-1)}}\right) + O^{(i)} \\
&\quad + V_e^{(i-1)} g_2\left(\tfrac{2E^{(i)}}{V_e^{(i-1)}}, \tfrac{2D^{(i)}}{V_e^{(i-1)}}\right) + 2\left(D^{(i)}N_d + E^{(i)}N_e + O^{(i)}Z\right), \\
\Sigma^{(i)} &= (R+L) g_3\left(\tfrac{V_e^{(i)}}{R+L}, \tfrac{V_m^{(i)}}{R+L}, \tfrac{V_d^{(i)}}{R+L}\right) + \left(V_e^{(i)} + V_m^{(i)}\right) Z + V_d^{(i)} \Delta, \\
R^{(i)} &= R + L - V_e^{(i)} - V_d^{(i)} - V_m^{(i)}, \\
V_e^{(i)} &= V_e^{(i-1)}/2 + E^{(i)} + O^{(i)},\ V_d^{(i)} = V_d^{(i-1)}/2 + D^{(i)},\ V_m^{(i)} = V_m^{(i-1)}/2 + O^{(i)}.
\end{aligned}$$

For large weight vectors, it makes sense to first shift the considered instance. That is for a fixed $c \in \mathbb{F}_q$, we shift the whole error set $\mathbb{E}$ to $\widetilde{\mathbb{E}} = \{a + c \mid a \in \mathbb{E}\}$. Let us denote by $\mathbf{c}$ the all c vector. Then, such shifting can easily be done by computing the syndrome $\mathbf{s}_c$ of $\mathbf{c}$ and adding it to the original syndrome $\mathbf{s}$: $(\mathbf{e} + \mathbf{c})\mathbf{H}^\top = \mathbf{s} + \mathbf{s}_c$. By choosing $c \in -\mathbb{E}$, one can set the error at all positions with value c to zero. Hence, one obtains $\widetilde{\mathbb{E}} = \{a + c \mid a \in \mathbb{E}\} \setminus \{0\}$ of size $\widetilde{z} = z - 1$. With this error set of reduced size, one can proceed as before. That is we again use the sets

$$\widetilde{\mathbb{D}} = \left\{a - b \mid a, b \in \widetilde{\mathbb{E}}\right\} \setminus \left\{\pm\widetilde{\mathbb{E}}_0\right\} \text{ and } -\widetilde{\mathbb{E}} = \left\{-e \mid e \in \widetilde{\mathbb{E}}\right\} \setminus \widetilde{\mathbb{E}}.$$

Note that for these sets, $n_e(q,z,x)$ and $n_d(q,z,x)$ are indeed dependent on the element x. In order to avoid a more complicated analysis, we resolve this issue by defining the *average* number of representations for an element in $\widetilde{\mathbb{E}}$ as

$$\widetilde{n_e}(q,z,c) = \frac{1}{\widetilde{z}} \sum_{x \in \widetilde{\mathbb{E}}} n_e(q,z,x) \quad \text{and} \quad \widetilde{n_d}(q,z,c) = \frac{1}{\widetilde{z}} \sum_{x \in \widetilde{\mathbb{E}}} n_d(q,z,x),$$

which depends not on the particular element but only on the chosen shift. Hence, $\widetilde{n_e}$ and $\widetilde{n_d}$ can be directly used in Theorem 12.

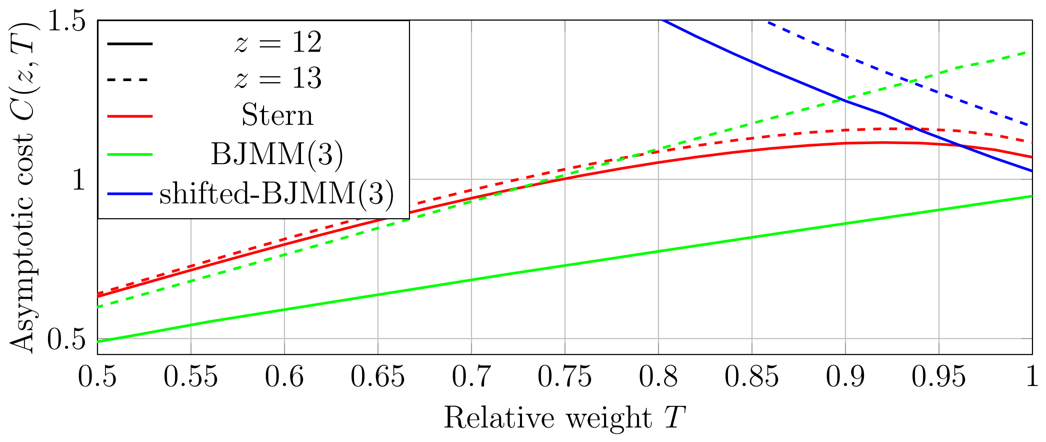


Figure 4.3: Comparison of the asymptotic complexity for the restricted Stern algorithm and the restricted BJMM(3) algorithm using $q = 157$ and $R = 0.45$.

In Figure 4.3, we compare the complexity coefficients of different information set decoders as a function of the relative error weight T. The considered code rate is $R = 0.45$. The field size $q = 157$ allows for $z = 12$ and $z = 13$, which correspond to the solid and dashed lines, respectively. While the performance of Stern depends only on the size of $\mathbb{E}$, the performance of the BJMM algorithms depends on its structure. For $z = 12$, $\mathbb{E}$ possesses a lot of additive structure, which is why BJMM(3) can improve over Stern. In particular, $\mathbb{E} = -\mathbb{E}$ and $n_e(157, 12) = 2$ allow for an increased number of representations. This is not the case for $z = 13$, where we only improve over Stern in the low error weight regime. Finally, we observe that shifting has to be taken into account for high error weights, but becomes quickly impractical as the weight decreases. Considering these observations, we avoid choosing instances for which the BJMM algorithm can achieve a significantly lower complexity than restricted Stern. However, it can be seen that in every case higher weights lead to higher complexities. For Stern, the maximum is not exactly at $T = 1$, but still it occurs at relatively high weights.

4.1.2 Exploiting the additive structure for small z

As already said before, the additive structure of $\mathbb{E}$ facilitates the use of the representation technique. In particular, when z is even, the restricted set has more additive structure. This happens because for every element of $\mathbb{E}$, also its opposite is contained in $\mathbb{E}$, as demonstrated in Lemma 7.

Lemma 7. *Let $\mathbb{E} = \{g^j \mid j \in \{0, 1, \cdots, z-1\}\} \in \mathbb{F}_q$ be the restricted set, $g \in \mathbb{F}_q^*$ not primitive of multiplicative order z. If z is even, then the restricted set can be expressed as*

$$\mathbb{E} = \left\{\pm g^j \mid j \in \left\{0, 1, \cdots, \frac{z}{2} - 1\right\}\right\}.$$

Proof. To prove $-g^j$ is contained in $\mathbb{E}$ it is necessary to prove that -1 is, as the product of any pair of elements in the restricted set also lives in $\mathbb{E}$.

We need to show that there exists an integer i such that $g^i = -1$. By definition of multiplicative order, it holds that $g^z = 1$ and $g^i \neq 1$ for any $i < z$.

Let us define the element $a = g^{z/2}$. Notice that $z/2$ is an integer, as z is even. By squaring a we get

$$a^2 = \left(g^{z/2}\right)^2 = g^z = 1.$$

This means that a is a root of the polynomial $x^2 - 1 = (x+1)(x-1)$, whose solutions are $x = \pm 1$. Since $z/2 < z$, a cannot be 1. Hence, $a = g^{z/2} = -1$. $\square$

Therefore, if the order z of the restricted set is even, the previous presentation of ISD attacks for R-SDP simplifies, as it holds that $-\mathbb{E} = \mathbb{E}$.

In the following pages, we will focus on cases with small and even values of z, such as 2, 4 and 6. We will explicitly show the additive structure and discuss the performances of the proposed algorithms. For larger choices of z, the additive structure of $\mathbb{E}$ has to be assessed, which, as we will see, depends on the factorization of $x^z - 1$ in $\mathbb{F}_q$. As such a factorization cannot be given in general, the actual additive structure for any proposed z should be checked independently.

Case $z = 2$

In this case, we have $\mathbb{E} = \{\pm 1\}$ and $\mathbb{D} = \{\pm 2\}$. Figure 4.4 shows the additive structure of the restricted set for $z = 2$.

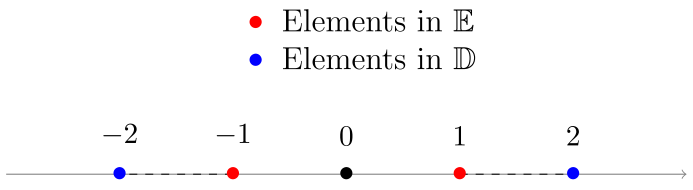


Figure 4.4: Additive structure of $\mathbb{E}$ for $z = 2$.

In order to construct the intermediate lists using representation merge, we have the usual $v_e^{(i)}$ entries in $\{\pm 1\}$ and we also require $v_d^{(i)}$ to denote the number of ± 2's on level i. Then, the number of well-formed vectors on level i is given by $\binom{k+l}{v_e^{(i)}, v_d^{(i)}} 2^{v_e^{(i)} + v_d^{(i)}}$.

The number of representations of $\mathbf{e}^{(i)} = \mathbf{e}_1^{(i+1)} + \mathbf{e}_2^{(i+1)}$ on level i is counted as per Figure 4.5. For this, it is enough to count the number of $\mathbf{e}_1^{(i+1)}$. There are $\binom{v_e^{(i)}}{v_e^{(i)}/2}$ ways of splitting the support of the elements in $\mathbb{E}$, without choosing the entries. Out of the chosen $v_e^{(i)}/2$ we chose $\delta^{(i+1)}$ positions, that overlap with ± 2's in $\mathbf{e}_2^{(i+1)}$ and also in the $v_e^{(i)}/2$ non-chosen positions we choose $\delta^{(i+1)}$ many positions to be ± 2.

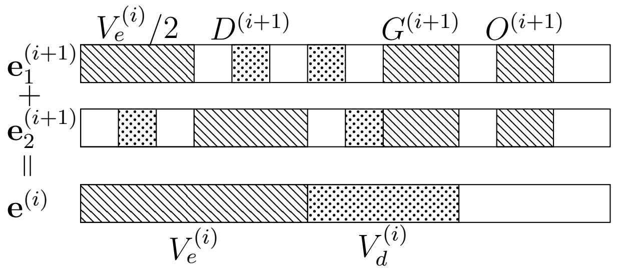


Figure 4.5: Counting the number of representations for level i.

For this we have $\binom{v_e^{(i)}/2}{\delta^{(i+1)}}^2$ possibilities. Out of the $v_d^{(i)}$ many ± 2's on level i, $\gamma^{(i+1)}$ are constructed as $\pm(1+1)$. Notice that this way of obtaining elements of $\mathbb{D}$ was not used for the general case, but can be used when z is even because of Lemma 7. The remaining $(v_d^{(i)} - \gamma^{(i+1)})$ many ± 2's are obtained by support splitting. This results in $\binom{v_d^{(i)}}{(v_d^{(i)}-\gamma^{(i+1)})/2,\gamma^{(i+1)}}$. Finally, one can choose $o^{(i+1)}$ out of the $k + \ell - v_d^{(i)} - v_e^{(i)}$ zero-positions.

Let $V_d^{(i)} = \lim\limits_{n\to\infty} \frac{v_d^{(i)}(n)}{n}$, $D^{(i)} = \lim\limits_{n\to\infty} \frac{\delta^{(i)}(n)}{n}$, $G^{(i)} = \lim\limits_{n\to\infty} \frac{\gamma^{(i)}(n)}{n}$. Then, the following corollary holds.

Corollary 1. *The asymptotic cost of the* BJMM(3) *algorithm for R-SDP with* $z = 2$ *is calculated according to Theorem* 12*, where for* $i \in \{1, 2\}$ *we use*

$$\begin{aligned}
V_e^{(i)} &= \frac{V_e^{(i-1)}}{2} + O^{(i)} + G^{(i)} \\
V_d^{(i)} &= \frac{V_d^{(i-1)} - G^{(i)}}{2} + D^{(i)},
\end{aligned}$$

as relative weights and

$$\begin{aligned}
\Sigma^{(i)} &= (R+L)g_2\left(\frac{V_e^{(i)}}{R+L}, \frac{V_d^{(i)}}{R+L}\right) + Z(V_e^{(i)} + V_d^{(i)}), \\
U^{(i)} &= V_e^{(i)}\left(1 + h_2\left(\frac{2D^{(i+1)}}{V_e^{(i)}}\right)\right) + V_d^{(i)} g_2\left(\frac{G^{(i+1)}}{V_d^{(i)}}, \frac{V_d^{(i)} - G^{(i+1)}}{2V_d^{(i)}}\right) \\
&\quad + (R + L - V_e^{(i)} - V_d^{(i)})h_2\left(\frac{O^{(i+1)}}{R + L - V_e^{(i)} - V_d^{(i)}}\right) + Z \cdot O^{(i+1)}.
\end{aligned}$$

Figure 4.6 shows the curve of the asymptotic cost $C(T)$ for $R = 0.5$, $q = 157$ and $z = 2$. Notice that unique decoding is ensured for every T. A general adaptation of BJMM, i.e. a more classical version that does not use elements in $\mathbb{D}$ (BJMM$_{(\setminus\mathbb{D})}(a)$), has also been considered.

It can be observed that the adapted BJMM algorithm improves significantly

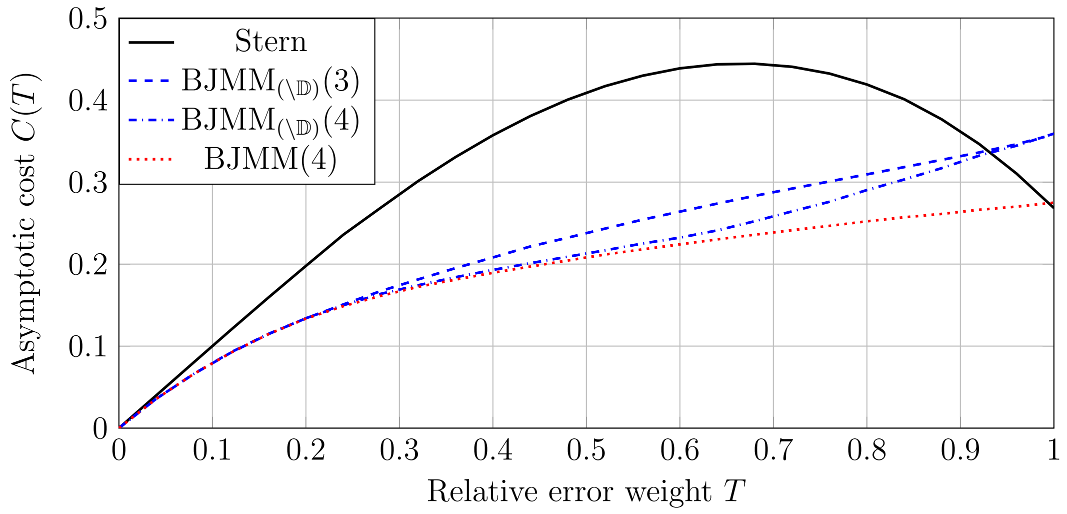


Figure 4.6: Comparison of the asymptotic complexity for restricted Stern's algorithm, the general adaption of BJMM and the generalization given in Corollary 1 using $q = 157$, $z = 2$ and $R = 0.5$.

over restricted Stern for medium error weights. While, as already stated, for the classical SDP two representation levels give the best performance [8], here three representation layers were found to be optimal. The generalization given in Corollary 1 allows for a further improvement for increased error weights. It was observed that the number of elements from $\mathbb{D}$ is optimized to approximately 0 in the base lists. Hence, one can start with restricted base lists and not lose a noticeable amount of performance.

In [3], the case of $z = 2$ is considered with the particular choice of $T = 1$. As can be seen from Figure 4.6, in this weight regime, the approach of Corollary 1 does not offer any improvement over Stern. For such instances, it is advantageous to shift the error vector $\mathbf{e}$, as the resulting relative error weight would approximately become $T = 0.5$.

Case $z = 4$

In this case, we have $\mathbb{E} = \{\pm 1, \pm g\}$ and $\mathbb{D} = \{\pm(g+1), \pm(g-1)\}$. Notice that according to the definition of $\mathbb{D}$, it should also include ± 2 and $\pm 2g$. We do not take such values because they are less useful to representations, considering that they can only represent one value of $\mathbb{E}$ (± 1 and $\pm g$ respectively). Figure 4.7 shows the additive structure of the restricted set for $z = 4$.

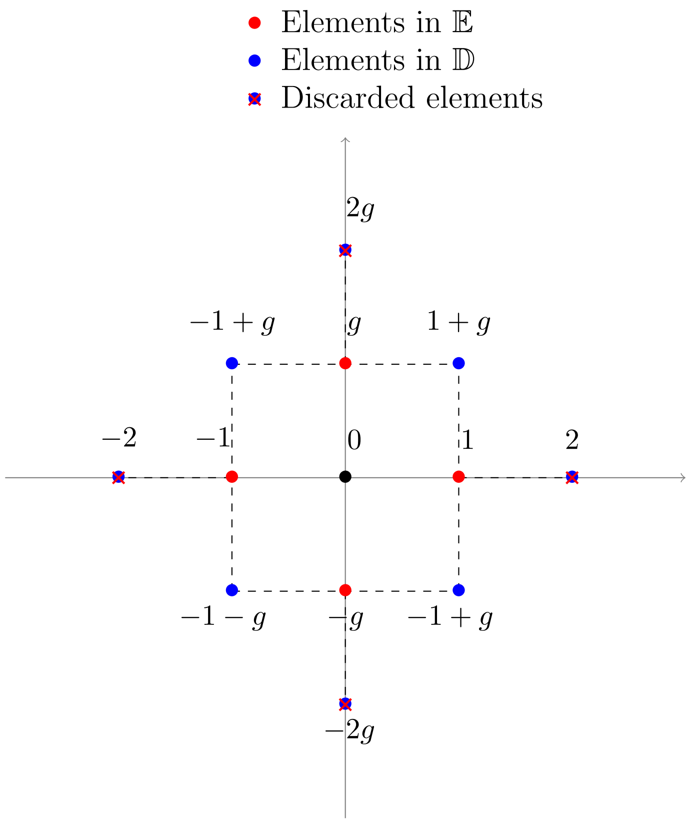


Figure 4.7: Additive structure of $\mathbb{E}$ for $z = 4$.

Following the approach for $z = 2$, we obtain the same number of possibilities for choosing the supports of $\mathbf{e}_1^{(i+1)}$ and $\mathbf{e}_2^{(i+1)}$. There are, however, more possibilities for picking the values in the chosen positions: there are two possibilities for obtaining any $e \in \mathbb{E}$ as the sum of $a \in \mathbb{E}$ and $b \in \mathbb{D}$ and two possibilities for obtaining $e \in \mathbb{D}$ as the sum of two elements in $\mathbb{E}$. This increases the number of representations for level

i overall by factor of $2^{2(\delta^{(i+1)}+\gamma^{(i+1)})}$ compared to the number computed for $z = 2$. Hence, we obtain the same $V_e^{(i)}$, $V_d^{(i)}$, $\Sigma^{(i)}$ as in Corollary 1 and the new number of representations is given by

$$\begin{aligned} U^{(i)} = V_e^{(i)}\left(1 + h_2\left(\frac{2D^{(i+1)}}{V_e^{(i)}}\right)\right) + V_d^{(i)} g_2\left(\frac{G^{(i+1)}}{V_d^{(i)}}, \frac{V_d^{(i)} - G^{(i+1)}}{2V_d^{(i)}}\right) \\ + (R + L - V_e^{(i)} - V_d^{(i)})h_2\left(\frac{O^{(i+1)}}{R + L - V_e^{(i)} - V_d^{(i)}}\right) + Z \cdot O^{(i+1)} \\ + 2(D^{(i+1)} + G^{(i+1)}). \end{aligned}$$

Figure 4.8 shows the curve of the asymptotic complexity $C(T)$ for $R = 0.5$, $q = 157$ and $z = 4$. Again, BJMM improves over Stern for medium error weights. The generalization using $\mathbb{D}$ gives a further speedup for increased weights.

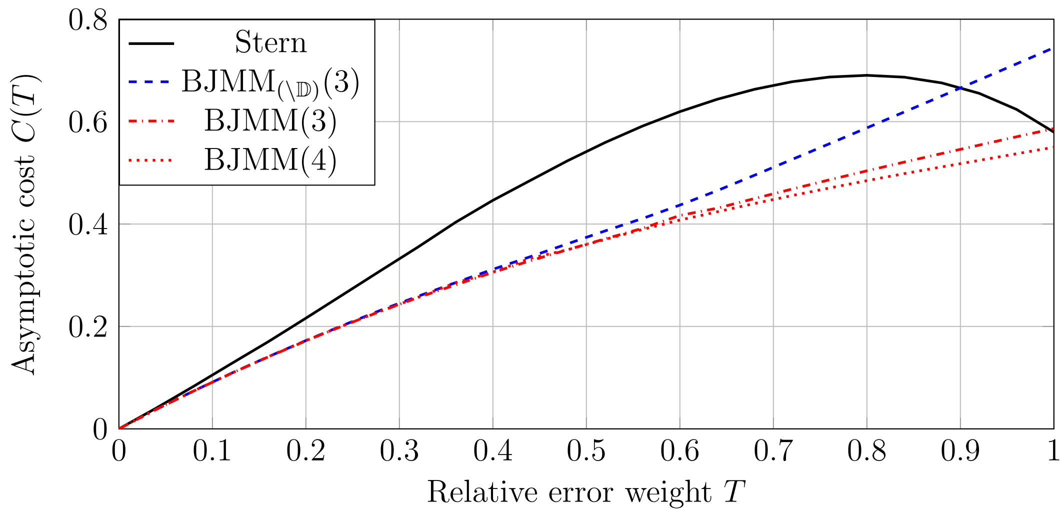


Figure 4.8: Comparison of the asymptotic complexity for restricted Stern's algorithm, the general adaption of BJMM and the proposed generalization using $q = 157$, $z = 4$ and $R = 0.5$.

Case $z = 6$

In this special case, also considered in [31], we have $\mathbb{E} = \{\pm 1, \pm g, \pm(g-1)\}$. In fact, from the definition of multiplicative order z, it holds that g is a root of $x^6 - 1 = (x^3-1)(x+1)(x^2-x+1)$ and that g is not a root of (x^3-1) or $(x+1)$. Therefore, it must be a root of (x^2-x+1), which means that $g^2 = g-1$. Notice that $\mathbb{E}$ has a great additive structure: any element $e \in \mathbb{E}$ can be obtained as $e = e_1 + e_2 = e_2 + e_1$ with $e_1, e_2 \in \mathbb{E}, e_1 \neq e_2$. Thus, by discarding the elements of $\mathbb{D}$ analogously as we did for $z = 4$, for $z = 6$ it holds that $\mathbb{D} = \{\emptyset\}$ and the presentation of representation-based attacks further simplifies. Figure 4.9 shows the additive structure of the restricted set for $z = 6$.

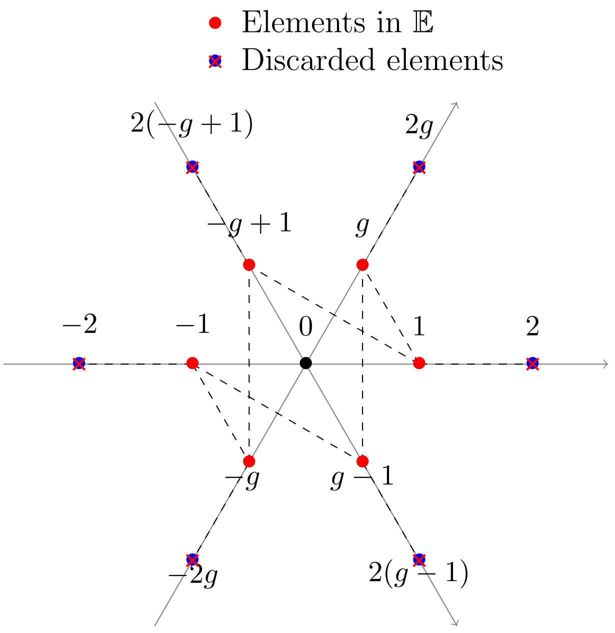


Figure 4.9: Additive structure of $\mathbb{E}$ for $z = 6$.

Therefore, using again Figure 4.5 to explain the number of representations, we have $v_d^{(i)} = \gamma^{(i)} = 0$. Hence, we get exactly the same as in Corollary 1 by setting $V_d^{(i)} = G^{(i)} = 0$ and adding the new representations $2^{2\delta^{(i+1)}}$.

Corollary 2. *The asymptotic cost of the* BJMM(3) *algorithm for R-SDP with* $z = 6$ *is calculated according to Theorem* 12*, where for* $i \in \{1, 2\}$ *we use*

$$V_e^{(i)} = \frac{V_e^{(i-1)}}{2} + O^{(i)} + D^{(i)}$$

as relative weights and

$$\begin{aligned}
\Sigma^{(i)} &= (R+L)h_2\left(\frac{V_e^{(i)}}{R+L}\right) + Z \cdot V_i, \\
U^{(i)} &= V_e^{(i)}\left(1 + h_2\left(\frac{2D^{(i+1)}}{V_e^{(i)}}\right)\right) + 2D^{(i+1)} \\
&\quad + (R+L-V_e^{(i)})h_2\left(\frac{O^{(i+1)}}{R+L-V_e^{(i)}}\right) + Z \cdot O^{(i+1)}.
\end{aligned}$$

Figure 4.10 shows the curve of the asymptotic cost $C(T)$ for $R = 0.5$, $q = 157$ and $z = 6$. Using the additive structure of $\mathbb{E}$ as proposed in Corollary 2 enables a remarkable speedup over Stern and the basic BJMM adaption. Experiments, for which we allowed elements from $\{\pm(g+1), \pm(2g-1), \pm(g-2)\}$ in intermediate lists, did not yield further performance improvements.

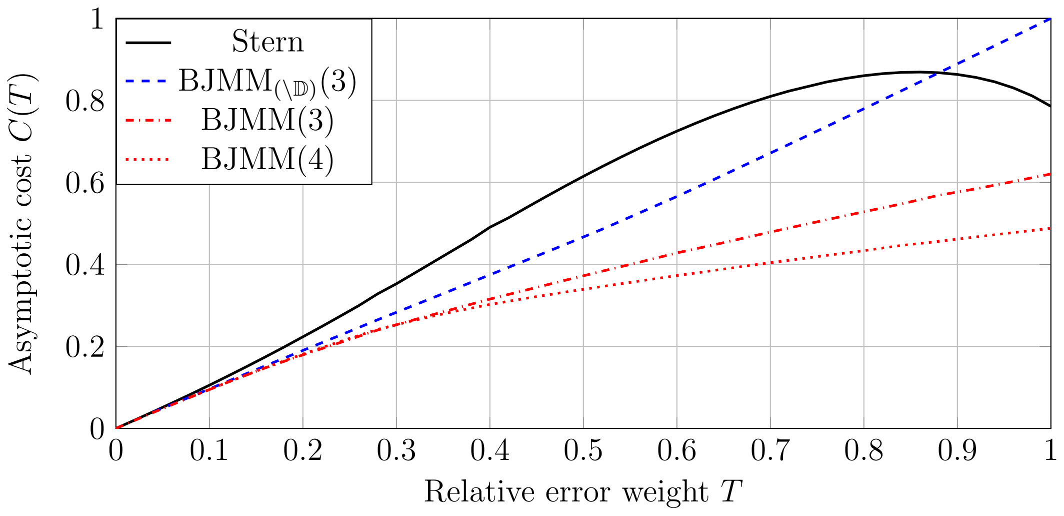


Figure 4.10: Comparison of the complexity coefficients for restricted Stern's algorithm, the general adaption of BJMM and the proposed generalization using $q = 157$, $z = 6$ and $R = 0.5$.

Chapter 5

Signature schemes

As we briefly discussed, there are different ways of building digital signature schemes. This thesis focuses on ZKID-based signature schemes, more specifically on code-based ones. At first, we will present and discuss some existing schemes based on SDP, then we will adapt them to the restricted problem. Finally, we will present a further restricted version of R-SDP and apply it to the schemes.

5.1 CVE

In [13] Cayrel, Véron and El Yousfi Alaoui presented an identification Zero-Knowledge scheme based on Syndrome Decoding Problem in $\mathbb{F}_q$. Such a scheme, which we will refer to as CVE, is shown in Figure 5.1. It is clear how the security of the private key is based on the hardness of SDP. In fact, the only way an attacker has to get a cheating probability of 1 is by knowing the private key $\mathbf{e}$ of weight t. To do so, he must be able to solve the instance $\{\mathbf{s}, \mathbf{H}\}$ of the SDP problem.

The CVE scheme involves the generation by the Prover of two commitments: c_0 e c_1. Only one of them will be used in the verification phase, based on the challenge

Public Data	Parameters $q, n, k, t \in \mathbb{N}$, parity-check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$
Private Key	$\mathbf{e} \in \mathbb{F}_q^n$ with weight t
Public Key	$\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

PROVER		VERIFIER
Choose $\mathbf{u} \xleftarrow{\$} \mathbb{F}_q^n$, $\tau \xleftarrow{\$} \mathfrak{M}_n$ Set $\mathsf{c}_0 := \mathsf{Hash}(\tau, \mathbf{u}\mathbf{H}^\top)$ Set $\mathsf{c}_1 := \mathsf{Hash}(\tau(\mathbf{u}), \tau(\mathbf{e}))$		
	$\xrightarrow{\{\mathsf{c}_0, \mathsf{c}_1\}}$	
		Choose $\beta \xleftarrow{\$} \mathbb{F}_q^*$
	$\xleftarrow{\beta}$	
Set $\mathbf{y} := \tau(\mathbf{u} + \beta\mathbf{e})$		
	$\xrightarrow{\mathbf{y}}$	
		Choose $b \xleftarrow{\$} \{0,1\}$
	$\xleftarrow{b}$	
If $b = 0$, set $f := \tau$ If $b = 1$, set $f := \mathbf{e}' = \tau(\mathbf{e})$		
	$\xrightarrow{f}$	
		If $b = 0$, accept if $\mathsf{c}_0 = \mathsf{Hash}(\tau, \tau^{-1}(\mathbf{y})\mathbf{H}^\top - \beta\mathbf{s})$
		If $b = 1$, accept if $\mathrm{wt}(\mathbf{e}') = t$ and $\mathsf{c}_1 = \mathsf{Hash}(\mathbf{y} - \beta\mathbf{e}', \mathbf{e}')$

Figure 5.1: CVE ZKID protocol.

bit b that will be chosen by the Verifier. Notice also that the responses f of the Prover in the $b = 0$ and $b = 1$ cases, if revealed simultaneously, would compromise the private key. In fact, τ is invertible and thus it would be possible to extract $\mathbf{e}$ from the knowledge of τ and $\tau(\mathbf{e})$. Obviously that is not allowed, as only one of the two possible responses is transmitted.

It can be easily verified that CVE satisfies the *completeness* property. Indeed, the honest user is identified in both cases:

Case $b = 0$: $\tau^{-1}(\mathbf{y})\mathbf{H}^\top - \beta\mathbf{s} = (\mathbf{u} + \beta\mathbf{e})\mathbf{H}^\top - \beta\mathbf{s} = \mathbf{u}\mathbf{H}^\top + \beta\mathbf{e}\mathbf{H}^\top - \beta\mathbf{s} = \mathbf{u}\mathbf{H}^\top$

Case $b = 1$: $\mathbf{y} - \beta\mathbf{e}' = \tau(\mathbf{u} + \beta\mathbf{e}) - \beta\tau(\mathbf{e}) = \tau(\mathbf{u} + \beta\mathbf{e} - \beta\mathbf{e}) = \tau(\mathbf{u})$.

For the *soundness error* computation, let us consider an attacker that wants to identify without having the private key. As already said before, he has to send the commitments before knowing the challenge bit b. Therefore, the only way he has to commit cheating is to foresee the challenge and to accordingly prepare the commitments. So, let us define his two possible strategies, ST_0 and ST_1, where the foreseen challenge is $b = 0$ and $b = 1$, respectively.

- ST_0: the attacker uniformly chooses $\mathbf{u}$ e τ at random and finds a vector $\hat{\mathbf{e}}$, without weight constraints, such that $\hat{\mathbf{e}}\mathbf{H}^\top = \mathbf{s}$ (this can be easily done algebraically, because of the lack of weight constraint). Commitments are then generated choosing $\mathsf{c}_0 = \mathsf{Hash}(\tau, \mathbf{u}\mathbf{H}^\top)$ and c_1 at random (in this case it will not be verified). At this point, the attacker is able to correctly answer regardless of β. Indeed, in this case the Verifier cannot check the weight of $\hat{\mathbf{e}}$, so the value $\mathbf{y} = \tau(\mathbf{u} + \beta\hat{\mathbf{e}})$ and the response $f = \tau$ are enough to pass the verification.

  Actually, this strategy can be improved in order to cheat also when the Verifier chooses $b = 1$. In fact, instead of choosing it at random, the attacker can prepare an appropriate commitment $\mathsf{c}_1 = \mathsf{Hash}(\mathbf{u}^*, \mathbf{e}')$. In this phase, the attacker has to foresee the value β that will be chosen by the Verifier. Such a forecast $\widetilde{\beta}$ is then used to compute the values $\mathbf{u}^*$ and $\mathbf{e}'$. Given that he already set $\mathbf{y} = \tau(\mathbf{u}) + \beta\tau(\hat{\mathbf{e}})$ for the $b = 0$ case, in the commitment phase (when β is not yet available) the attacker uses $\widetilde{\mathbf{y}} = \tau(\mathbf{u} + \widetilde{\beta}\hat{\mathbf{e}})$ to find $\mathbf{e}'$ of weight

t such that $\widetilde{\mathbf{y}} - \widetilde{\beta}\mathbf{e}' = \tau(\mathbf{u}) + \widetilde{\beta}(\tau(\hat{\mathbf{e}}) - \mathbf{e}') = \mathbf{u}^*$. If the Verifier will choose $\beta = \widetilde{\beta}$ (that happens with probability $\frac{1}{q-1}$), then $\widetilde{\mathbf{y}} = \mathbf{y}$ e $\mathsf{Hash}(\mathbf{y} - \beta\mathbf{e}', \mathbf{e}') = \mathsf{Hash}(\widetilde{\mathbf{y}} - \widetilde{\beta}\mathbf{e}', \mathbf{e}') = \mathsf{Hash}(\mathbf{u}^*, \mathbf{e}') = \mathsf{c}_1$ and the check is successful. Otherwise, if $\beta \neq \widetilde{\beta}$ the attacker fails;

- ST_1: the attacker uniformely chooses $\mathbf{u}$ and τ at random and randomly picks a vector $\hat{\mathbf{e}}$ of the correct weight t. Commitments are then generated choosing c_0 at random (it will not be verified) and $\mathsf{c}_1 = \mathsf{Hash}(\tau(\mathbf{u}), \tau(\hat{\mathbf{e}}))$. Even in this situation, the attacker is able to correctly answer regardless of β. Indeed, in this case the Verifier does not check that the vector $\hat{\mathbf{e}}$ is a valid solution of SDP with instance $\{\mathbf{s}, \mathbf{H}\}$, but instead he only checks the correctness of the weight. Therefore, the value $\mathbf{y} = \tau(\mathbf{u} + \beta\hat{\mathbf{e}})$ and the response $f = \tau(\hat{\mathbf{e}})$ are enough to pass the verification.

  Also in this case the strategy can be improved. In fact, in order to cheat even if the Verifier chooses $b = 0$, the attacker can try to foresee the value $\widetilde{\beta}$ and choose $\mathsf{c}_0 = \mathsf{Hash}(\tau, \mathbf{u}\mathbf{H}^\top + \widetilde{\beta}(\hat{\mathbf{e}}\mathbf{H}^\top - \mathbf{s}))$. If $\beta = \widetilde{\beta}$ (with probability $\frac{1}{q-1}$), then $\mathbf{y} = \widetilde{\mathbf{y}}$ and the check would be successful: $\mathsf{Hash}(\tau, \tau^{-1}(\mathbf{y})\mathbf{H}^\top - \beta\mathbf{s}) = \mathsf{Hash}(\tau, \tau^{-1}(\widetilde{\mathbf{y}})\mathbf{H}^\top - \widetilde{\beta}\mathbf{s}) = \mathsf{Hash}(\tau, \mathbf{u}\mathbf{H}^\top + \widetilde{\beta}(\hat{\mathbf{e}}\mathbf{H}^\top - \mathbf{s})) = \mathsf{c}_0$.

Notice that with the term *challenge value* we refer to b and not to β, even if both of them are sent to the Prover after receiving the commitments. This is due to the fact that, according to the strategies we just presented, the knowledge of β is not needed to commit cheating with a soundness error ε of at least 1/2. Considering both strategies and their improvements, the soundness error is

$$\begin{aligned}\varepsilon = \mathsf{Pr}[\mathsf{Cheating}] &= \sum_{i=0}^{1} \mathsf{Pr}[\mathsf{ST} = \mathsf{ST_i}](\mathsf{Pr}[b = i] + \mathsf{Pr}[b = 1-i, \beta = \widetilde{\beta}]) \\ &= \sum_{i=0}^{1} \mathsf{Pr}[\mathsf{ST} = \mathsf{ST_i}](\mathsf{Pr}[b = i] + \mathsf{Pr}[b = 1-i]\mathsf{Pr}[\beta = \widetilde{\beta}]) \\ &= \sum_{i=0}^{1} \frac{1}{2}\left(\frac{1}{2} + \frac{1}{2}\frac{1}{q-1}\right) = \frac{1}{2}\left(1 + \frac{1}{q-1}\right) = \frac{q}{2(q-1)}.\end{aligned}$$

For the sake of brevity, we refer to [13] for the proof of the *zero knowledge* property.

The *communication cost* of the scheme is the cost of a complete interaction between the two parties, measured in number of used bits. The communication cost of a single round can be computed by adding the number of bits that are needed to represent each of the elements transmitted between Prover and Verifier in the *5-way pass* of CVE protocol. Then, supposing that to reach a certain security level N rounds of the protocol in Figure 5.1 are needed (cheating probability ε^N), we have to multiply the cost of a single round by N.

Before doing so, a few tricks have to be considered. In order to represent a vector of length n and weight t, we could represent every entry with a total of $n\lceil\log_2(q)\rceil$ bits or we could represent only its support and its ordered non-null entries with a total of $t\left(\lceil\log_2(n)\rceil + \lceil\log_2(q-1)\rceil\right)$ bits. Therefore, the cost of representing such a vector is

$$\psi(n, q, t) = \min\left\{n\lceil\log_2(q)\rceil, t\left(\lceil\log_2(n)\rceil + \lceil\log_2(q-1)\rceil\right)\right\}.$$

In order to send random elements, it is enough to communicate the seed of the pseudorandom noise generator (PRNG) that has been used to compute such elements. Regarding the seed and hash digest sizes, security aspects have to be considered. The seed has to be at least λ bits long. Indeed, if it was shorter, a man-in-the-middle attacker could obtain through brute force, with a cost lower than 2^λ, every possible τ. Therefore, for the case $b = 1$ he would be able to find the private key $\mathbf{e}$ starting from $\mathbf{e}'$ simply by inverting τ. Because of the possibility of birthday attacks, the hash digest has to be at least 2λ bits long. Then, we get a total communication cost of

$$N\left(\underbrace{2 \cdot 2\lambda}_{\{\mathsf{c}_0,\mathsf{c}_1\}} + \underbrace{\lceil \log_2(q-1) \rceil}_{\beta} + \underbrace{n\lceil \log_2(q) \rceil}_{\mathbf{y}} + \underbrace{1}_{b} + \underbrace{\frac{\lambda + \psi(n,q,t)}{2}}_{f}\right) \tag{5.1.1}$$

bits, where the last term is the average value of the cost of f, which is the arithmetic average of τ and $\tau(\mathbf{e})$ costs, being b equiprobable:

$$\overline{l_f} = \mathsf{Pr}[b=0] \cdot \lambda + \mathsf{Pr}[b=1] \cdot \psi(n,q,t) = \frac{\lambda + \psi(n,q,t)}{2}. \tag{5.1.2}$$

5.1.1 Optimization

As shown in [3], it is possible to further reduce the communication cost through an appropriate compression technique, shown in Figure 5.2.

In order to reduce the transmission cost of the commitment hash digests, the Prover generates them at the beginning of the protocol and sends only the hash digest of both commitments from every round. The hash function is used both for compressing the information to be sent and for making it non-modifiable afterwards, analogously as the Fiat-Shamir transform does. Then, the Verifier will be able to recompute the commitments and to verify that their hash digest is equal to the one received from the Prover. So, in this case the verification is not carried out

Public Data	Parameters $q, n, k, t \in \mathbb{N}$, parity-check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$
Private Key	$\mathbf{e} \in \mathbb{F}_q^n$ with weight t
Public Key	$\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

PROVER		VERIFIER
Generate $\mathsf{c}_0^{(i)}, \mathsf{c}_1^{(i)}$ for $i = 0, ..., N-1$		
Set $\mathsf{c} = \mathsf{Hash}(\mathsf{c}_0^{(0)}, \mathsf{c}_1^{(0)}, ..., \mathsf{c}_0^{(N-1)}, \mathsf{c}_1^{(N-1)})$		
	$\xrightarrow{\mathsf{c}}$	
	$\xleftarrow{\qquad}$ Repeat single round for N times $\xrightarrow{\qquad}$	
		Check validity of c

GENERIC i-th ROUND		
		Choose $\beta^{(i)} \xleftarrow{\$} \mathbb{F}_q^*$
	$\xleftarrow{\beta^{(i)}}$	
Set $\mathbf{y}^{(i)} = \tau^{(i)}(\mathbf{u}^{(i)} + \beta\mathbf{e})$		
	$\xrightarrow{\mathbf{y}^{(i)}}$	
		Choose $b^{(i)} \xleftarrow{\$} \{0,1\}$
	$\xleftarrow{b^{(i)}}$	
If $b^{(i)} = 0$, set $f^{(i)} := \tau^{(i)}$		
If $b^{(i)} = 1$, set $f^{(i)} := (\mathbf{e}')^{(i)} = \tau^{(i)}(\mathbf{e})$		
	$\xrightarrow{f^{(i)},\ \mathsf{c}_{b^{(i)}\oplus 1}^{(i)}}$	
		If $b^{(i)} = 0$, compute
		$\mathsf{c}_0^{(i)} = \mathsf{Hash}\big(\tau^{(i)}, (\tau^{(i)})^{-1}(\mathbf{y}^{(i)})\mathbf{H}^\top - \beta^{(i)}\mathbf{s}\big)$
		If $b^{(i)} = 1$ and $\mathrm{wt}((\mathbf{e}')^{(i)}) = t$, compute
		and $\mathsf{c}_1^{(i)} = \mathsf{Hash}\big(\mathbf{y}^{(i)} - \beta^{(i)}(\mathbf{e}')^{(i)}, (\mathbf{e}')^{(i)}\big)$

Figure 5.2: CVE ZKID protocol with compression technique.

singularly in each round anymore, but at the end of the execution of all N rounds. Considering that in the i-th round the Verifier computes from the response only one of the two commitments ($\mathsf{c}_{b^{(i)}}^{(i)}$, where $b^{(i)} \xleftarrow{\$} \{0,1\}$), the Prover has to send also the unused commitment $\mathsf{c}_{b^{(i)}\oplus 1}^{(i)}$. With this technique, the communication cost is

$$\underbrace{2\lambda}_{\mathsf{c}} + N\left(\underbrace{\lceil \log_2(q-1)\rceil}_{\beta^{(i)}} + \underbrace{n\lceil \log_2(q)\rceil}_{\mathbf{y}^{(i)}} + \underbrace{1}_{b^{(i)}} + \underbrace{2\lambda}_{\mathsf{c}^{(i)}_{b^{(i)}\oplus 1}} + \underbrace{\frac{\lambda+\psi(n,q,w)}{2}}_{f^{(i)}}\right).$$

It is clear that the cost saving with respect to (5.1.1) grows with the number of rounds N.

5.1.2 Applying Fiat-Shamir transform

To get the signature scheme, we apply the Fiat-Shamir transform analogously as we did in Figure 2.2. The challenge bits $b^{(i)}$ and the values $\beta^{(i)}$ are obtained from the hash digest of the message m concatenated to the commitment digest c. The signature σ is composed of c and of the responses from each round obtained as in Figure 5.2:

$$\mathsf{rsp} = \left\{\{\mathbf{y}^{(i)}\}_{i\in\mathbb{Z}_N}, \{f^{(i)}\}_{i\in\mathbb{Z}_N}, \{\mathsf{c}^{(i)}_{b^{(i)}\oplus 1}\}_{i\in\mathbb{Z}_N}\right\}.$$

The terms c and $\mathsf{c}^{(i)}_{b^{(i)}\oplus 1}$ are hash digest of 2λ bit, $\mathbf{y}^{(i)}$ is $n\lceil log_2(q)\rceil$ bits long and the size of $f^{(i)}$ is the one shown in (5.1.2). Therefore, the size of the signature based on CVE is

$$|\sigma|_{\mathsf{CVE}} = \underbrace{2\lambda}_{\mathsf{c}} + N\left(\underbrace{n\lceil \log_2(q)\rceil}_{\mathbf{y}^{(i)}} + \underbrace{2\lambda}_{\mathsf{c}^{(i)}_{b^{(i)}\oplus 1}} + \underbrace{\frac{\lambda+\psi(n,q,t)}{2}}_{f^{(i)}}\right).$$

The public key is the syndrome $\mathbf{s}$ and its bitsize is

$$|PK|_{\mathsf{CVE}} = (n-k)\lceil \log_2(q)\rceil\,.$$

5.2 GPS

In [20], Gueron, Persichetti and Santini present a code-based zero-knowledge signature scheme based on a ZKID protocol we refer to as GPS. Such protocol attains an arbitrarily low soundness error. In such way, less round are needed and consequently, at least in theory, the signature size is reduced. The GPS protocol is build by applying a structure called *sigma protocol with helper* to the CVE protocol. Therefore, also in this case the underlying protocol is SDP.

5.2.1 Sigma protocol with helper

The *sigma protocol with helper* is an interactive algorithm among three parties: the Prover (that wants to get identified), the Verifier (that carries out the identification) and the Helper (a trusted third party).

The protocol applied to the CVE scheme is shown in Figure 5.3.

For the sake of brevity, we refer to [20] for the proof of the *zero knowledge* property.

It can be easily verified that the sigma protocol with helper satisfies the *completeness* property. The first check is $(\mathsf{Hash}(\mathsf{r}, \tau, \mathbf{t}) = \mathsf{c}) \wedge (\tau \text{ is isometry})$. The first part verifies that the private key is a solution of the parity-check equations $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$. In fact:

$$\mathbf{t} = \tau(\mathbf{y})\mathbf{H}^\top - z\mathbf{s} = \tau(\mathbf{u})\mathbf{H}^\top + z\tau(\widetilde{\mathbf{e}})\mathbf{H}^\top - z\mathbf{s} = \tau(\mathbf{u})\mathbf{H}^\top + z\mathbf{e}\mathbf{H}^\top - z\mathbf{s} = \tau(\mathbf{u})\mathbf{H}^\top.$$

The second part guarantees the correctness of the weight of $\mathbf{e} = \tau(\widetilde{\mathbf{e}})$, because checking that τ is an isometry corresponds, by definition, to checking that $wt_H(\mathbf{e}) = wt_H(\widetilde{\mathbf{e}})$ and, in case of honest execution, $\widetilde{\mathbf{e}}$ has the correct weight t. It is important to emphasize that the verification of the weight is carried out for one half by the

Verifier (check of τ isometry) and for the other half it is based on the trust that the Verifier puts in the Helper ($\widetilde{\mathbf{e}}$ of the correct weight). If for some reason the Helper could not be trusted anymore, it would not be possible to check the correctness of the weight of the SDP solution.

For the analysis of the *soundness* property, let us define the strategies an attacker could use. He could choose between two strategies:

- ST_a: An attacker could easily algebraically obtain a solution $\mathbf{e}_f = \tau(\widetilde{\mathbf{e}}_f)$ of any weight of the parity-check equations. Then, instead of the vector $\widetilde{\mathbf{e}}$ given by the Helper through the seed, he could use $\widetilde{\mathbf{e}}_f$ in order to pass the first check. However, the second check would not be passed, because $\mathbf{y}_f = \mathbf{u} + z\widetilde{\mathbf{e}}_f$ is not the one used by the Helper to compute c_z. In other words, the second check binds the attacker to use the same $\mathbf{u}$ and $\widetilde{\mathbf{e}}$ used by the Helper. Then, such strategy is not successful because, by hypothesis, the Verifier can trust the Helper;

- ST_b: As usual, it is possible to cheat by accurately preparing the commitment. Indeed, supposing the attacker expects to receive $\overline{z}$ as challenge value, he could send the commitment $\mathsf{c} = \mathsf{Hash}(\mathsf{r}, \tau, \mathbf{x})$, where

$$\mathbf{x} = \tau(\mathbf{u})\mathbf{H}^\top + \overline{z}\tau(\widetilde{\mathbf{e}})\mathbf{H}^\top - \overline{z}\mathbf{s},$$

  which is equal to the $\mathbf{t}$ computed by the Verifier. In this way, the attacker passes the first check and he is able to pass also the second since he used the same $\mathbf{u}$ and $\widetilde{\mathbf{e}}$ used by the Helper to generate $\mathsf{c}_{\overline{z}}$.

The only possible strategy is ST_b, which has a success probability of the reciprocal of the challenge space size. Therefore, the soundness error is $\varepsilon = 1/q$, which is remarkably lower than the CVE one ($\varepsilon = \frac{q}{2(q-1)}$). This happens because the challenge space, i.e. the set of elements where the challenge is drawn from, has gone from $\{0, 1\}$ to $\mathbb{F}_q$. Moreover, ε can be chosen arbitrarily low by appropriately choosing the prime number q.

Analogously as highlighted for CVE, also in this case the minimum seed length is λ. In fact, if it was shorter a man-in-the-middle attacker could get every possible $\widetilde{\mathbf{e}}$ with a cost lower than 2^{λ}. Being τ published in the response rsp, it would be then immediate for the attacker to find the private key $\mathbf{e} = \tau(\widetilde{\mathbf{e}})$.

5.2.2 Removing the Helper

The big disadvantage of the *sigma protocol with helper* algorithm is the necessity of a trusted Helper. To overcome this obstacle, i.e. removing the Helper from the algorithm, we use the *cut-and-choose* technique. According to this technique, the Prover has to simulate the execution of the Helper. To do so, after uniformly sampling at random the seed that represents the Helper's input, the Prover computes the aux following the Setup phase shown in Figure 5.3. However, the Verifier has to make sure of the simulated Helper's honesty and, to do so, requires multiple executions of such simulation. After this, the Verifier chooses only one of the simulations (challenge instance) that will be used for the authentication as shown in Figure 5.3. For the rest of the unused simulations, the Prover sends the seeds and the aux values to the Verifier. Using such seeds, the Verifier re-simulates the Helper and checks that the aux obtained from its simulation are equal to the ones received from the Prover. The fact that the Prover does not previously know which instance will be used for the authentication does not allow him to commit cheating without trying to guess the instance. In this way, if the check on the aux values is successful, the Verifier can trust the simulated Helper (again, with a certain degree of uncertainty). In Figure 5.4 the protocol with the addition of the cut-and-choose technique is shown.

Public Data Parameters $q, n, k, t \in \mathbb{N}$, parity-check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$
Private Key $\mathbf{e} \in \mathbb{F}_q^n$ with weight t
Public Key $\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

PROVER HELPER VERIFIER

Input: Uniform random $\mathsf{seed} \in \{0,1\}^\lambda$

I. Setup - $\mathsf{H}(\mathsf{seed})$

Choose $\mathbf{u} \xleftarrow{\mathsf{seed}} \mathbb{F}_q^n$ and $\widetilde{\mathbf{e}} \xleftarrow{\mathsf{seed}} \mathbb{F}_q^n$ with weight t

For all $v \in \mathbb{F}_q$:

1. Choose $\mathsf{r}_v \xleftarrow{\mathsf{seed}} \{0,1\}^\lambda$
2. Compute $\mathsf{c}_v = \mathsf{Hash}(\mathsf{r}_v, \mathbf{u} + v\widetilde{\mathbf{e}})$

Set $\mathsf{aux} := \{\mathsf{c}_v\}_{v\in\mathbb{F}_q}$

$\xleftarrow{\mathsf{seed}}$ $\xrightarrow{\mathsf{aux}}$

II. Commitment - $\mathsf{P}_1(\mathbf{H}, \mathbf{e}, \mathsf{seed})$

Choose $\mathbf{u} \xleftarrow{\mathsf{seed}} \mathbb{F}_q^n$ and $\widetilde{\mathbf{e}} \xleftarrow{\mathsf{seed}} \mathbb{F}_q^n$ with weight t

Determine isometry $\tau = \mathsf{FindIsometry}(\mathbf{e}, \widetilde{\mathbf{e}}) : \mathbf{e} = \tau(\widetilde{\mathbf{e}})$

Choose $\mathsf{r} \xleftarrow{\$} \{1,0\}^\lambda$

Compute $\mathsf{c} = \mathsf{Hash}(\mathsf{r}, \tau, \tau(\mathbf{u})\mathbf{H}^\top)$

$\xrightarrow{\mathsf{c}}$

III. Challenge - $\mathsf{V}_1(\cdot)$

Choose $z \xleftarrow{\$} \mathbb{F}_q$

Set $\mathsf{ch} := z$

$\xleftarrow{\mathsf{ch}}$

IV. Response - $\mathsf{P}_2(\mathsf{ch}, \mathsf{seed})$

Regenerate r_z from seed

Compute $\mathbf{y} = \mathbf{u} + z\widetilde{\mathbf{e}}$

Set $\mathsf{rsp} := (\mathsf{r}, \mathsf{r}_z, \tau, \mathbf{y})$

$\xrightarrow{\mathsf{rsp}}$

V. Verification - $\mathsf{V}_2(\mathbf{H}, \mathsf{s}, \mathsf{aux}, \mathsf{c}, \mathsf{rsp})$

Compute $\mathbf{t} = \tau(\mathbf{y})\mathbf{H}^\top - z\mathbf{s}$

Check that $\mathsf{Hash}(\mathsf{r}, \tau, \mathbf{t}) = \mathsf{c}$ and that τ is an isometry

Check that $\mathsf{Hash}(\mathsf{r}_z, \mathbf{y}) = \mathsf{c}_z$

Return 1 (accept) if both checks are successful, or 0 (reject) otherwise

Figure 5.3: Sigma protocol with helper.

It is important to emphasize that the Prover cannot communicate the seed of the instance selected by the Verifier. Otherwise, $\widetilde{\mathbf{e}}_I$ would be obtainable from seed_I and, together with the knowledge of τ_I contained in rsp_I, it would be immediate for an attacker to find the secret key $\mathbf{e} = \tau_I(\widetilde{\mathbf{e}}_I)$.

Public Data	Parameters $q, n, k, t \in \mathbb{N}$, parity-check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$
Private Key	$\mathbf{e} \in \mathbb{F}_q^n$ with weight t
Public Key	$\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

PROVER — VERIFIER

I. Commitment

For all $i \in \mathbb{Z}_M$:

1. Choose $\mathsf{seed}_i \overset{\$}{\leftarrow} \{0,1\}^\lambda$
2. Compute $\mathsf{aux}_i = \mathsf{H}(\mathsf{seed}_i)$
3. Compute $\mathsf{c}_i = \mathsf{P}_1(\mathbf{H}, \mathbf{e}, \mathsf{seed}_i)$

$$\xrightarrow{\{\mathsf{aux}_i\}_{i\in\mathbb{Z}_M}, \{\mathsf{c}_i\}_{i\in\mathbb{Z}_M}}$$

II. Challenge

Choose $I \overset{\$}{\leftarrow} \mathbb{Z}_M$

Choose $z \overset{\$}{\leftarrow} \mathbb{F}_q$

Set $\mathsf{ch} := \{I, z\}$

$$\xleftarrow{\mathsf{ch}}$$

III. Response

Compute $\mathsf{rsp}_I = \mathsf{P}_2(z, \mathsf{seed}_I)$

$$\xrightarrow{\mathsf{rsp}_I, \{\mathsf{seed}_i\}_{i\neq I}}$$

IV. Verification

For all $i \in \mathbb{Z}_M$ such that $i \neq I$:

1. Compute $\overline{\mathsf{aux}}_i = \mathsf{H}(\mathsf{seed}_i)$
2. Check that $\overline{\mathsf{aux}}_i = \mathsf{aux}_i$

Set $b := 1$ if all checks are successful, $b := 0$ otherwise

Compute $b' = \mathsf{V}_2(\mathbf{H}, \mathbf{s}, \mathsf{aux}_I, \mathsf{c}_I, \mathsf{rsp}_I)$

Accept if $b \wedge b' = 1$, reject otherwise

Figure 5.4: Sigma protocol with helper + cut-and-choose technique

In order to compute the soundness error, let us consider the two possible strategies that an attacker could use:

- ST_0: The cut-and-choose technique establishes that the Helper is simulated by the Prover, so the Helper's honesty is not guaranteed anymore. If the attacker could foresee the challenge instance I requested by the Verifier, he could use the previously shown strategy ST_a. In fact, the attacker could forge the Helper by modifying its $\mathsf{aux}_I = \{(\mathsf{c}_v)_I\}_{v\in\mathbb{F}_q}$:

$$(\mathsf{c}_v)_I = \mathsf{Hash}\big((\mathsf{r}_v)_I, \mathbf{u}_I + v(\widetilde{\mathbf{e}}_f)_I)\big) = \mathsf{Hash}\big((\mathsf{r}_v)_I, (\mathbf{y}_f)_I\big).$$

The success probability is equal to the probability of guessing the challenge instance, that is $1/M$;

- ST_1: For each of the M instances, the attacker modifies every commitment c_i as shown for the strategy ST_b. Such strategy is successful (with probability $1/q$ regardless of the challenge instance I) because:
  - check for $i \neq I$: the Verifier only considers the aux_i, that are not modified by ST_b;
  - check for $i = I$: same reason shown in the explanation of ST_b.

The attacker can use the strategy with the higher success probability between ST_0 and ST_1. Therefore, the soundness error is

$$\varepsilon = \max\left\{\frac{1}{M}, \frac{1}{q}\right\}. \tag{5.2.1}$$

As for the CVE, to get the desired cheating probability $2^{-\lambda}$ we can execute N multiple rounds of the just presented protocol. The number of rounds is

$$N = \left\lceil \frac{-\lambda}{\log_2\left(\max\left\{\frac{1}{M}, \frac{1}{q}\right\}\right)} \right\rceil.$$

In order to obtain the communication cost, we have to multiply by N the cost of a single round. It is then necessary to compute the cost of the protocol of Figure 5.4. Recall that the aux_i are composed of q hash digests, therefore each one of them (M

in total) is $2\lambda q$ bits long. Also the c_i are hash digests, each one of them (M in total) is 2λ bits long. Regarding the response of the challenge instance, rsp_I is composed of two random strings (λ bits each), a monomial transformation and a vector over $\mathbb{F}_q^n$ ($n \lceil log_2(q) \rceil$ bits). The transformation is made of a permutation vector of length n with unique entries in $\mathbb{Z}_n$ and a vector of scalar values $\in \left(\mathbb{F}_q^*\right)^n$. If the SDP problem requires a solution of max weigth $t = n$, the transformation can be reduced to the vector of scalar values. Therefore, its size is

$$l_\tau = \begin{cases} n\left(\lceil \log_2(n) \rceil + \lceil \log_2(q-1) \rceil\right), & \text{if } t < n \\ n\lceil \log_2(q-1) \rceil, & \text{if } t = n \end{cases} \tag{5.2.2}$$

bits. In the end, considering the $M-1$ $\{\mathsf{seed}_i\}_{i \neq I}$, each one of them takes λ bits, we get a communication cost of

$$\begin{aligned} N\Big[& \underbrace{2\lambda q M}_{\{\mathsf{aux}_i\}_{i\in\mathbb{Z}_M}} + \underbrace{2\lambda M}_{\{\mathsf{c}_i\}_{i\in\mathbb{Z}_M}} + \underbrace{\lceil \log_2(M) \rceil + \lceil \log_2(q) \rceil}_{\mathsf{ch}} + \\ & + \underbrace{2\lambda + l_\tau + n\lceil log_2(q) \rceil}_{\mathsf{rsp}_I} + \underbrace{\lambda(M-1)}_{\{\mathsf{seed}_i\}_{i\neq I}} \Big] \end{aligned} \tag{5.2.3}$$

5.2.3 Applying Fiat-Shamir transform

By applying the Fiat-Shamir transform to the ZKID protocol we get the signature scheme. The challenge values of the Signer's signature are obtained from the hash digest of the message m concatenated to the aux values and commitments. In this case, for the sake of simplicity, instead of taking the challenge values from the bits of the hash digest string, it is preferred to use the hash digest as seed of the PRNG that is used to randomly sample the challenge. Obviously, at the minimum variation of $\mathsf{m} \mid\mid \mathsf{cmt}^{(j)}$ a whole different challenge is obtained, therefore the principle of the

Fiat-Shamir transform is still valid. In Figure 5.5 the signature scheme based on the GPS ZKID protocol with N round is shown.

Public Data	Parameters $q, n, k, t \in \mathbb{N}$, parity-check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, message m
Private Key	$\mathbf{e} \in \mathbb{F}_q^n$ with weight t
Public Key	$\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

SIGNER VERIFIER

I. Signature

For every round $j \in \mathbb{Z}_N$:

1. Generate commitment $\mathsf{cmt}^{(j)} = \left\{\{\mathsf{aux}_i^{(j)}\}_{i\in\mathbb{Z}_M}, \{\mathsf{c}_i^{(j)}\}_{i\in\mathbb{Z}_M}\right\}$ as in GPS
2. Compute $\mathsf{seed}^{(j)} = \mathsf{Hash}(\mathsf{m}, \mathsf{cmt}^{(j)})$
3. Choose $\mathsf{ch}^{(j)} = \{I^{(j)}, z^{(j)}\} \xleftarrow{\mathsf{seed}} \mathbb{Z}_M \times \mathbb{F}_q$
4. Generate response $\mathsf{rsp}^{(j)} = \{\mathsf{rsp}_{I^{(j)}}^{(j)}, \{\mathsf{seed}_i^{(j)}\}_{i\neq I^{(j)}}\}$ as in GPS

Set signature $\sigma := \left\{\{\mathsf{cmt}^{(j)}\}_{j\in\mathbb{Z}_N}, \{\mathsf{rsp}^{(j)}\}_{j\in\mathbb{Z}_N}\right\}$

$$\xrightarrow{\quad\sigma\quad}$$

II. Verification

Parse σ as $\left\{\{\mathsf{cmt}^{(j)}\}_{j\in\mathbb{Z}_N}, \{\mathsf{rsp}^{(j)}\}_{j\in\mathbb{Z}_N}\right\}$

For every round $j \in \mathbb{Z}_N$:

1. Compute $\mathsf{seed}^{(j)} = \mathsf{Hash}(\mathsf{m}, \mathsf{cmt}^{(j)})$
2. Choose $\mathsf{ch}^{(j)} = \{I^{(j)}, z^{(j)}\} \xleftarrow{\mathsf{seed}} \mathbb{Z}_M \times \mathbb{F}_q$
3. Perform verification as in GPS

Accept or reject the signature accordingly

Figure 5.5: Signature scheme based on GPS ZKID protocol.

In order to obtain the bitsize of the signature σ, it is enough to eliminate from (5.2.3) the cost of the challenge ch:

$$\begin{aligned}|\sigma|_{\mathsf{GPS}} = N\Big[& \underbrace{2\lambda q M}_{\{\mathsf{aux}_i^{(j)}\}_{i\in\mathbb{Z}_M}} + \underbrace{2\lambda M}_{\{\mathsf{c}_i^{(j)}\}_{i\in\mathbb{Z}_M}} + \underbrace{2\lambda + l_\tau + n\lceil\log_2(q)\rceil}_{\mathsf{rsp}_{I^{(j)}}^{(j)}} + \\ & + \underbrace{\lambda(M-1)}_{\{\mathsf{seed}_i^{(j)}\}_{i\neq I^{(j)}}}\Big]\end{aligned} \tag{5.2.4}$$

As for CVE, the public key is the syndrome $\mathbf{s}$ and its bitsize is

Table 5.1: Comparison between optimized CVE and non-optimized GPS-based signature sizes, $\lambda = 128$

	q	n	t	M	N	Sign. Size
CVE (opt.)	991	220	90	–	129	52.5 kB
GPS (no opt.)				991	13	390.2 MB

$$|PK|_{\mathsf{GPS}} = (n-k)\left\lceil \log_2(q) \right\rceil .$$

Because of the smaller soundness error, the GPS signature needs less rounds than the CVE signature. However, this does not lead to a reduction of the signature size, because the cost of the single round is remarkably higher in the GPS. In Table 5.1 the signature sizes of the two schemes for a security level of $\lambda = 128$ are compared. As it is, the performances of the GPS signature scheme in terms of signature size are not even comparable to the ones of the CVE signature scheme.

5.2.4 Optimization

Because of the considerable increase in the signature size, it could seem that using the sigma protocol with helper is counter-productive. The yet unexpressed advantage of this solution lies in the fact that the sigma protocol with helper can be highly optimized. Indeed, it is still possible to apply different techniques in order to drastically shrink the communication cost of the GPS protocol. Such techniques are shown below.

- **Commitments.** In the protocol shown in Figure 5.4, the Prover transmits M commitments, even if just one of them (c_I) is used for the verification. Through a function, called $\mathsf{MerkleTree}$, a Merkle Tree T of depth $d = \lceil \log_2(M) \rceil$ is built.

Its leaves are the hash digests of the commitments $\mathsf{c}_0, ..., \mathsf{c}_{M-1}$:

$$T_{d,\ell} = \mathsf{Hash}(\mathsf{c}_\ell)$$

where $0 \leq \ell \leq 2^d - 1$. If $d \geq \log_2(M)$, the rest of the leaves are obtained by hashing random strings. The internal nodes are then computed, starting from the layer d of the leaves, by hashing two nodes from the previous layer:

$$T_{u,\ell} = \mathsf{Hash}(T_{u+1,2\ell} || T_{u+1,2\ell+1})$$

where $0 \leq u \leq d-1$ and $0 \leq \ell \leq 2^u - 1$. The process iterates, until the root tree $T_{0,0}$ is reached. By doing so, after the Commitment phase, the Prover can just send the Merkle Tree root (2λ bits) and then include the authentication path of c_I once he gets the challenge from the Verifier. By *authentication path* we mean the list of sibling nodes on the path from the leaf to the root (excluded). Such list takes $2\lambda \lceil \log_2(M) \rceil$ bits. In this way, by recomputing c_I as $\mathsf{Hash}(\mathsf{r}, \tau, \mathbf{t})$ (see function V_2 in Figure 5.3) and by using the authentication path, the Verifier can recreate the Merkle Tree root and check that it is equal to the one received from the Prover. By using this technique, it is possible to save on the communication cost, going from $2\lambda M$ bits (M commitments c_i) to $2\lambda(1 + \lceil \log_2(M) \rceil)$ bits (root + authentication path). In Figure 5.6 an example of commitment Merkle Tree is shown. We call $\mathsf{MerkleTree}$ a function that creates a Merkle Tree from a set of leaves and $\mathsf{ReconstructRoot}$ a function that recomputes the root of a Merkle Tree from a leaf and its relative path;

- **Auxiliary Information.** The transmission of aux can be optimized as well. For starters, notice that the function V_2 in Figure 5.3 uses, in the second check, just one of the q commitments that compose the aux. Therefore, for

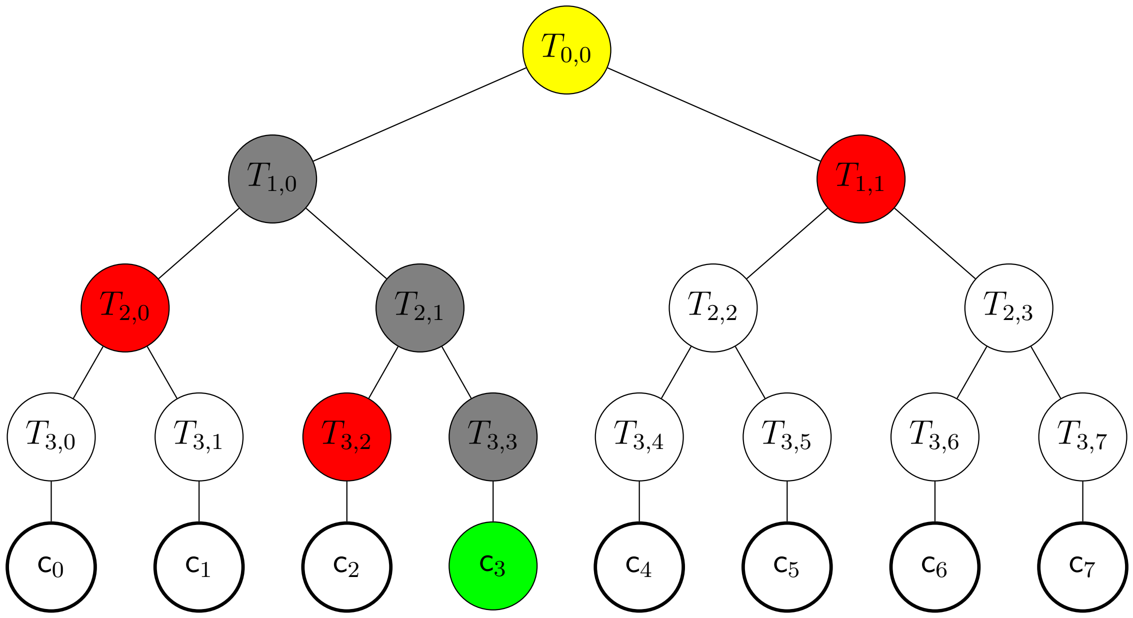


Figure 5.6: Example of commitments Merkle Tree, $M = 8$, $I = 3$. I-th commitment is shown in green, the relative authentication path is shown in red, the root transmitted after the Commitment phase is shown in yellow, the nodes recomputed by the Verifier are shown in grey and the commitments from which the path is calculated by the Prover are shown in thick line.

every $\{\mathsf{aux}_i\}_{i\in\mathbb{Z}_M}$ we can think of applying the same Merkle Tree technique we saw for the commitments, where the leaves are obtained from the hash digests of $\{\mathsf{c}_v\}_{v\in\mathbb{F}_q}$. In total we have M trees, each one of them with its own root and authentication path, so $2\lambda M(1+\lceil\log_2(q)\rceil)$ bits.

Notice also that only one of the M instances is executed, therefore only aux_I is used in the function V_2 and the rest is recomputed from the seeds. Then, the Prover can send the hash digest of the roots of the M Merkle Trees after the Commitment phase and the authentication path of aux_I's tree in the Response phase. The Verifier is then able to recompute the root of T_{aux_I} from the path and the relative c_z recomputed as $\mathsf{Hash}(\mathsf{r}_z, \mathbf{y})$ (see function V_2 in Figure 5.3).

The roots of the remaining trees are recomputed starting from the seeds. Therefore, the Verifier can then recompute the hash digest of the roots and then check that the result is equal to the digest received from the Prover after the Commitment phase. With these optimizations, we go from $2\lambda qM$ bits (M auxiliary information aux_i) to $2\lambda(1+\lceil\log_2(q)\rceil)$ bits (final root + I-th tree's authentication path). In Figure 5.7 an example of aux Merkle Tree is shown.

- **Seeds.** The transmission of the $M-1$ seeds can be optimized as well. In Figure 5.4, the Prover randomly chooses the seeds. For this reason, considering that the hash function can be modeled as a random oracle, it is possible to generate the seeds as hash digests. In particular, it is possible to get all of the M seeds as leaves of a binary hash tree of depth $d=\lceil\log_2(M)\rceil$. The Prover chooses an initial seed as tree root ($T_{0,0}=\mathsf{seed}$), that is given as input to the hash function. The produced digest is then split in half in order to get other two nodes of higher level. The process can then be repeated by hashing the

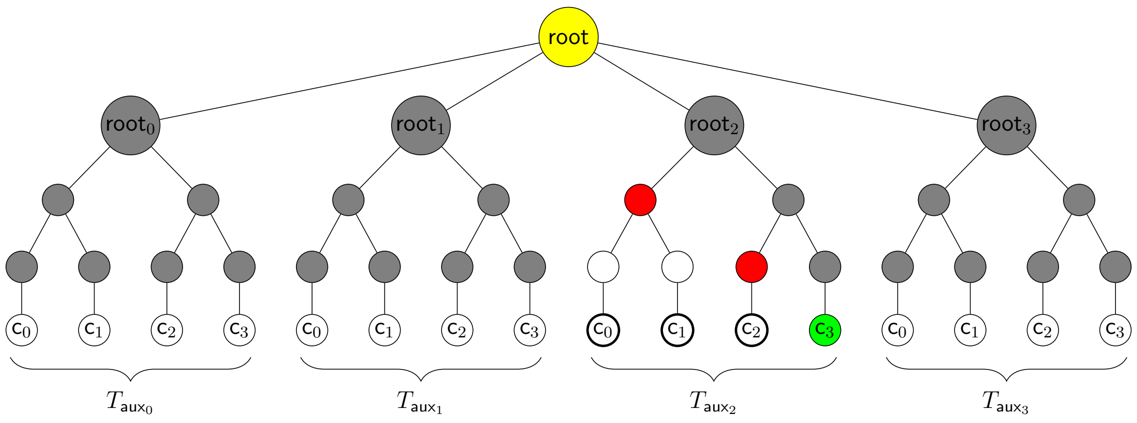


Figure 5.7: Example of auxiliary information Merkle Tree, $M = 4$, $I = 2$, $q = 4$, $z = 3$. The z-th commitment of I-th tree is shown in green, the relative authentication path is shown in red, the final root transmitted after the Commitment phase is shown in yellow, the nodes recomputed by the Verifier are shown in grey and the commitments from which the path is calculated by the Prover are shown in thick line.

obtained nodes until the desired number of leaves is reached:

$$(T_{u+1,2\ell} || T_{u+1,2\ell+1}) = \mathsf{Hash}(T_{u,\ell})$$

for $0 \leq u \leq d-1$ and $0 \leq \ell \leq 2^u - 1$. In order to communicate the $\{\mathsf{seed}_i\}_{i \neq I}$ to the Verifier, the Prover could think about sending the tree root. However, this would also publish seed_I that, as already said before, is not possible due to security reasons. The Prover can then communicate to the Verifier the seed path of seed_I, i.e. the list of sibling nodes on the path from the leaf to the root (excluded). Therefore, the communication cost relative to the seeds goes from $\lambda(M-1)$ to $\lambda \lceil \log_2(M) \rceil$ bits. In Figure 5.8 an example of seed hash tree is shown. We call $\mathsf{SeedTree}$ a function that builds the seed tree from an initial root seed, $\mathsf{SeedPath}$ a function that computes the seed path from the open instances indexes and the initial root seed, $\mathsf{ReconstructSeeds}$ a function that recomputes from the seed path all the seeds but the ones from the open

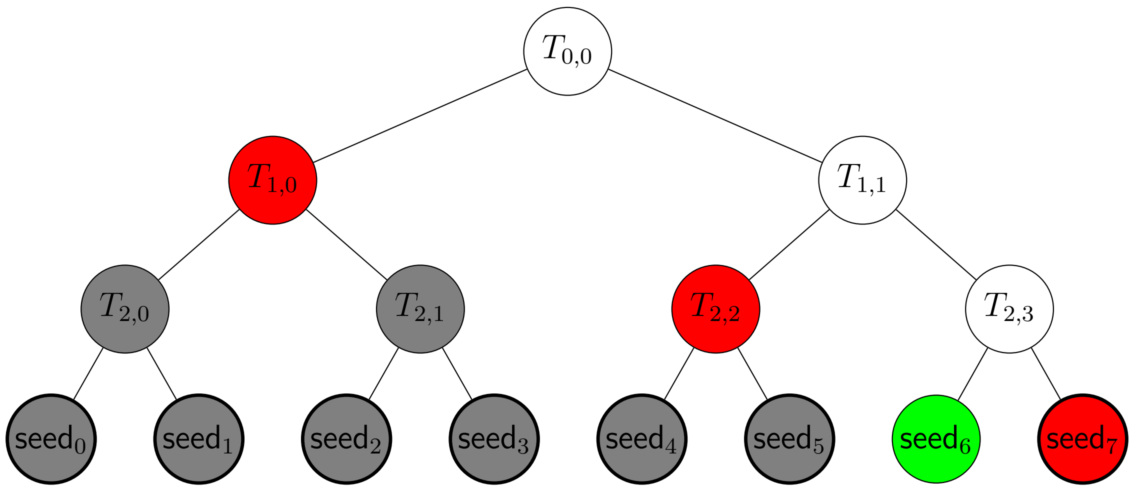


Figure 5.8: Example of seeds hash tree, $M = 8$, $I = 6$. I-th seed is shown in green, the relative seed path is shown in red, the nodes recomputed by the Verifier from the path are shown in gray and the obtained leaves are shown in thick line.

instances.

- **Executions.** In order to achieve the targeted security level λ, as it stands the GPS scheme uses N multiple executions of the protocol shown in Figure 5.4, where M instances are precomputed and only one is used. The same level can be attained with a single execution ($N = 1$) by verifying more than just an instance. In other words, $s = |S|$ instances are verified, where $S \subseteq \mathbb{Z}_M$. Previously, using multiple executions we obtained the total communication cost by multiplying the cost of one execution (also including the just presented optimizations) by the number of rounds N. Instead, in this way we fully exploit the optimizations we just discussed, since by executing only one round we get rid of the multiplicative term N. To be thorough, the protocol of Figure 5.4 with optimized commitments, aux values and seeds has a slightly higher cost with increasing s and, moreover, it depends on S for the same s. In Figure 5.9, a few examples of seed hash trees with $M = 8$ and $s = 2$ are shown for varying S. In Figure 5.9a the number of nodes that compose the seed path grows with respect to Figure 5.8. In the particular cases of Figure 5.9b and Figure5.9c, the number of path nodes drops with respect to Figure 5.9a, but this happens with smaller probability. Therefore, in general we can state that by increasing s the communication cost also grows, even if slightly. To eliminate from the analysis the variability of the cost given by S, we consider the communication cost in its worst case, that is

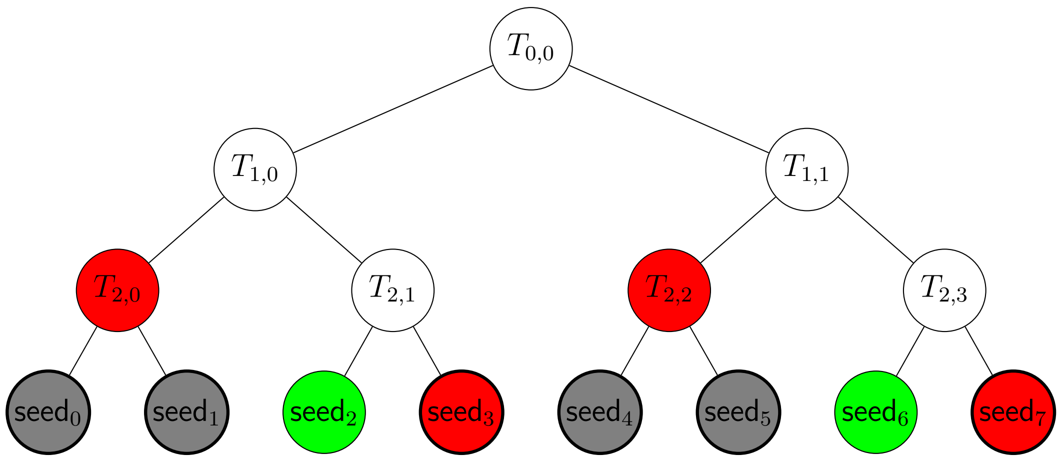
$$\lambda \left[2^{\lceil \log_2(s) \rceil} + s\big(\lceil \log_2(M) \rceil - \lceil \log_2(s) \rceil - 1\big)\right] \tag{5.2.5}$$

Similar considerations apply to the commitments tree (Figure 5.6). Regarding the aux tree (Figure 5.7), raising s means increasing the number of authentication paths sent after the Response phase. Therefore, once again, we recognize a (tolerable) increment of the cost with growing s. In the end, notice that in the extreme case $s \approx M$ we would get very small seed paths, but, at the same time, very numerous authentication paths. This is one of the reasons why cases with several instances to be verified must be avoided.

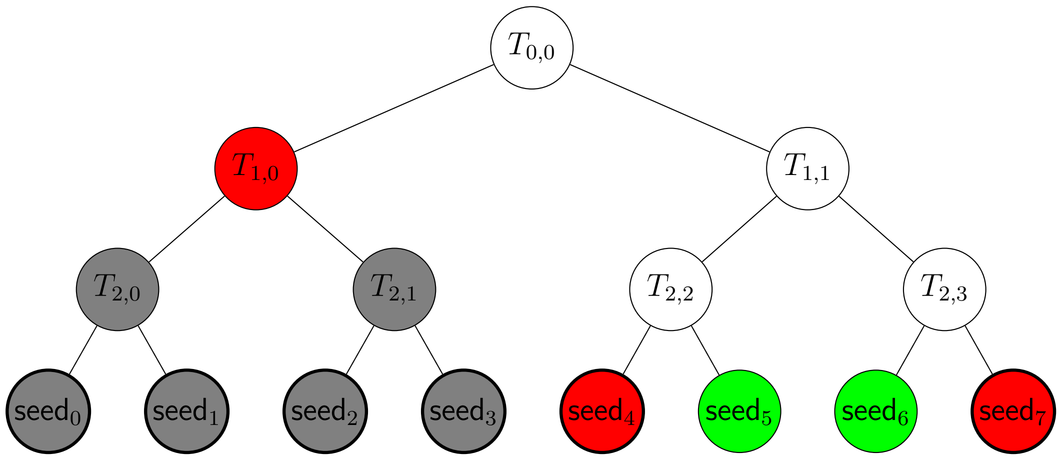
Clearly, the soundness error changes. In this case, by referring as $e \leq s$ to the number of dishonest instances computed with ST_0 from an adversary Prover, the soundness error ε is by

$$\max_{e \in \mathbb{Z}_{s+1}} \frac{\binom{M-e}{s-e}}{\binom{M}{s} q^{s-e}}. \tag{5.2.6}$$

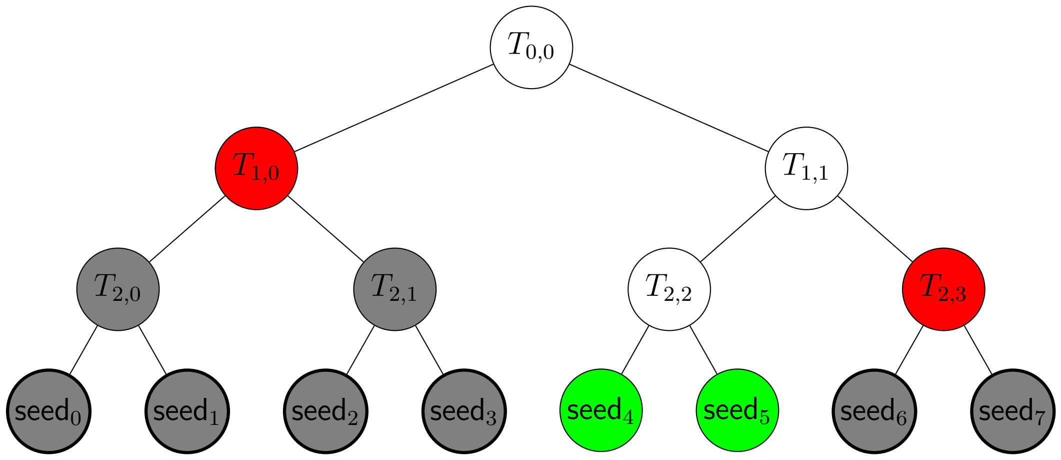
In fact, the attacker wins when all the e dishonest instances are included among the s that are checked by the Verifier and, at the same time, the remaining instances $s-e$ are computed with the correct value of z following the strategy ST_1. The first condition happens with a probability of $\binom{M-e}{s-e}/\binom{M}{s}$, while the second condition, statistically independent from the first, happens with a



(a) $s = 2$, $S = [2, 6]$. The seed path has 4 nodes (worst case).



(b) $s = 2$, $S = [5, 6]$. The seed path has 3 nodes (intermediate case).



(c) $s = 2$, $S = [4, 5]$. The seed path has 2 nodes (best case).

Figure 5.9: Example of seed hash trees, $M = 8$, $s = 2$ and varying S.

probability of $1/q^{s-e}$. Obviously, for $s = 1$ we get the (5.2.1) again:

$$\max_{e\in\mathbb{Z}_{s+1}} \frac{\binom{M-e}{s-e}}{\binom{M}{s}q^{s-e}} = \max_{e\in[0,1]} \frac{\binom{M-e}{s-e}}{\binom{M}{s}q^{s-e}} = \max\left\{\frac{1}{q}, \frac{1}{M}\right\}.$$

For the sake of completeness, notice that in the extreme case $s = M$ (every instance is verified) an attacker could use the ST_a for every instance, thus committing cheating with certain probability. Indeed, in this case the soundness error is

$$\max_{e\in\mathbb{Z}_{s+1}} \frac{\binom{M-e}{s-e}}{\binom{M}{s}q^{s-e}} = \max_{e\in\mathbb{Z}_{M+1}} \frac{\binom{M-e}{M-e}}{\binom{M}{M}q^{M-e}} = \max_{e\in\mathbb{Z}_{M+1}} q^{e-M} = 1.$$

Also for this reason, cases where $s \approx M$ are not considered.

The soundness error of (5.2.6) has two degrees of freedom, s and M. Qualitatively, the soundness error grows when each one of them does. Then, we have to decide if choosing a high s and a low M or vice versa. For example, for $q = 991$ we can attain a security level of at least $\lambda = 128$ both with $s = 19$, $M = 991$ and $s = 63$, $M = 133$. However, in the second case, s is almost half of M and, as we said earlier, this leads to an increase in the communication cost given by the paths. For this reason, the first case is preferable, i.e. a large M and a relatively small s. This is also shown by (5.2.5), which grows linearly with s and only logarithmically with M.

The complete scheme of the optimized GPS ZKID protocol is shown in Figure 5.10. The communication cost of the ZKID scheme can be easily obtained from

Public Data Parameters $q, n, k, t \in \mathbb{N}$, parity-check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$
Private Key $\mathbf{e} \in \mathbb{F}_q^n$ with weight t
Public Key $\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

PROVER VERIFIER

I. Commitment

```
Choose $\mathsf{seed} \xleftarrow{\$} \{0,1\}^\lambda$
Compute $\{\mathsf{seed}_i\}_{i\in\mathbb{Z}_M} = \mathsf{SeedTree}(\mathsf{seed})$
For all $i \in \mathbb{Z}_M$:
  1. Compute $\mathsf{aux}_i = \mathsf{H}(\mathsf{seed}_i)$
  2. Build tree $T_i^{(\mathsf{aux})} = \mathsf{MerkleTree}(\mathsf{aux}_i)$ and call $\mathsf{root}_i^{(\mathsf{aux})}$ its root
  3. Compute $\mathsf{c}_i = \mathsf{P}_1(\mathbf{H}, \mathbf{e}, \mathsf{seed}_i)$
Compute $\mathsf{h} = \mathsf{Hash}\big(\{\mathsf{root}_i^{(\mathsf{aux})}\}_{i\in\mathbb{Z}_M}\big)$
Build tree $T^{(\mathsf{c})} = \mathsf{MerkleTree}\big(\{\mathsf{c}_i\}_{i\in\mathbb{Z}_M}\big)$ and call $\mathsf{root}^{(\mathsf{c})}$ its root
```

$$\xrightarrow{\mathsf{h}, \mathsf{root}^{(\mathsf{c})}}$$

II. Challenge

```
Choose uniformly $S \subseteq \mathbb{Z}_M$ at random, with $|S| = s$
For all $j \in S$:
  1. Choose $z_j \xleftarrow{\$} \mathbb{F}_q$
Set $\mathsf{ch} := \big\{S, \{z_j\}_{j\in S}\big\}$
```

$$\xleftarrow{\mathsf{ch}}$$

III. Response

```
For all $j \in S$:
  1. Compute $\mathsf{rsp}_j = \mathsf{P}_2(z_j, \mathsf{seed}_j)$
  2. Compute $\mathsf{path}_j^{(\mathsf{aux})}$
  3. Compute $\mathsf{path}_j^{(\mathsf{c})}$
Compute $\mathsf{path}^{(\mathsf{seed})} = \mathsf{SeedPath}(S, \mathsf{seed})$
```

$$\xrightarrow{\{\mathsf{rsp}_j\}_{j\in S}, \{\mathsf{path}_j^{(\mathsf{aux})}\}_{j\in S}, \{\mathsf{path}_j^{(\mathsf{c})}\}_{j\in S}, \mathsf{path}^{(\mathsf{seed})}}$$

IV. Verification

```
Set $b := 1$
For all $j \in S$:
  1. Compute $\mathbf{t}_j = \tau_j(\mathbf{y}_j)\mathbf{H}^\top - z_j\mathbf{s}$
  2. Compute $\mathsf{c}_j = \mathsf{Hash}(\mathsf{r}_j, \tau_j, \mathbf{t}_j)$
  3. Compute $\overline{\mathsf{root}}^{(\mathsf{c})} = \mathsf{ReconstructRoot}(\mathsf{path}_j^{(\mathsf{c})}, \mathsf{c}_j)$
  4. Check that $\overline{\mathsf{root}}^{(\mathsf{c})} = \mathsf{root}^{(\mathsf{c})}$
  5. Set $b := 0$ if check is not successful
  6. Compute $(\mathsf{c}_{z_j})_j = \mathsf{Hash}((\mathsf{r}_z)_j, \mathbf{y}_j)$
  7. Compute $\overline{\mathsf{root}}_j^{(\mathsf{aux})} = \mathsf{ReconstructRoot}(\mathsf{path}_j^{(\mathsf{aux})}, (\mathsf{c}_{z_j})_j)$
Recover $\overline{\mathsf{seed}}_{j\notin S} = \mathsf{ReconstructSeeds}(S, \mathsf{path}^{(\mathsf{seed})})$
For all $j \notin S$:
  1. Compute $\overline{\mathsf{aux}}_j = \mathsf{H}(\overline{\mathsf{seed}}_j)$
  2. Build tree $T_{\overline{\mathsf{aux}}_j} = \mathsf{MerkleTree}(\overline{\mathsf{aux}}_j)$ and call $\overline{\mathsf{root}}_j^{(\mathsf{aux})}$ its root
Compute $\overline{\mathsf{h}} = \mathsf{Hash}\big(\{\overline{\mathsf{root}}_j^{(\mathsf{aux})}\}_{j\in\mathbb{Z}_M}\big)$
Set $b' := 1$ if $\overline{\mathsf{h}} = \mathsf{h}$, $b := 0$ otherwise
Accept if $b \wedge b' = 1$, reject otherwise
```

Figure 5.10: Optimized GPS ZKID protocol.

Figure 5.10:

$$\begin{aligned}&\underbrace{2\lambda}_{\mathsf{h}}+\underbrace{2\lambda}_{\mathsf{root}^{(\mathsf{c})}}+\underbrace{s(\lceil\log_2(M)\rceil+\lceil\log_2(q)\rceil)}_{\mathsf{ch}}+\\&+\underbrace{s(2\lambda+l_\tau+n\lceil\log_2(q)\rceil)}_{\{\mathsf{rsp}_j\}_{j\in S}}+\underbrace{s(2\lambda\lceil\log_2(q)\rceil)}_{\{\mathsf{path}_j^{(\mathsf{aux})}\}_{j\in S}}+\underbrace{s(2\lambda\lceil\log_2(M)\rceil)}_{\{\mathsf{path}_j^{(\mathsf{c})}\}_{j\in S}}+\\&+\underbrace{\lambda\left[2^{\lceil\log_2(s)\rceil}+s\big(\lceil\log_2(M)\rceil-\lceil\log_2(s)\rceil-1\big)\right]}_{\mathsf{path}^{(\mathsf{seed})}}\end{aligned} \tag{5.2.7}$$

5.2.5 Applying Fiat-Shamir transform

Once again, by applying the Fiat-Shamir transform to the protocol of Figure 5.10 we obtain the relative signature scheme, represented in Figure 5.11.

The signature size can be obtained from (5.2.7) by eliminating the term that represents the cost of the challenge:

$$\begin{aligned}|\sigma|_{\mathsf{GPS}}=&\underbrace{2\lambda}_{\mathsf{h}}+\underbrace{2\lambda}_{\mathsf{root}^{(\mathsf{c})}}+\underbrace{s(2\lambda+l_\tau+n\lceil\log_2(q)\rceil)}_{\{\mathsf{rsp}_j\}_{j\in S}}+\\&+\underbrace{s(2\lambda\lceil\log_2(q)\rceil)}_{\{\mathsf{path}_j^{(\mathsf{aux})}\}_{j\in S}}+\underbrace{s(2\lambda\lceil\log_2(M)\rceil)}_{\{\mathsf{path}_j^{(\mathsf{c})}\}_{j\in S}}+\\&\underbrace{\lambda\left[2^{\lceil\log_2(s)\rceil}+s\big(\lceil\log_2(M)\rceil-\lceil\log_2(s)\rceil-1\big)\right]}_{\mathsf{path}^{(\mathsf{seed})}}\end{aligned} \tag{5.2.8}$$

In Table 5.2 the signature sizes of the CVE-based and GPS-based schemes are compared for different parameters and security level $\lambda = 128$. The GPS-based signature scheme attains sizes of almost half of the ones obtainable with the CVE-based scheme.

Public Data	Parameters $q, n, k, t \in \mathbb{N}$, parity-check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, message m
Private Key	$\mathbf{e} \in \mathbb{F}_q^n$ with weight t
Public Key	$\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

SIGNER — VERIFIER

I. Signature

Generate commitment $\mathsf{cmt} = \{h, \mathsf{root}^{(\mathsf{c})}\}$ as in opt. GPS

Compute $\mathsf{seed} = \mathsf{Hash}(\mathsf{m}, \mathsf{cmt})$

Choose uniformly $S \subseteq \mathbb{Z}_M$ at random from seed, with $|S| = s$

For all $j \in S$:

1. Choose $z_j \xleftarrow{\mathsf{seed}} \mathbb{F}_q$

Set $\mathsf{ch} := \{S, \{z_j\}_{j\in S}\}$

For all $j \in S$:

Generate $\{\mathsf{rsp}_j\}_{j\in S}, \{\mathsf{path}_j^{(\mathsf{aux})}\}_{j\in S}, \{\mathsf{path}_j^{(\mathsf{c})}\}_{j\in S}, \mathsf{path}^{(\mathsf{seed})}$ as in opt. GPS

Set signature $\sigma := \left\{\mathsf{cmt}, \{\mathsf{rsp}_j\}_{j\in S}, \{\mathsf{path}_j^{(\mathsf{aux})}\}_{j\in S}, \{\mathsf{path}_j^{(\mathsf{c})}\}_{j\in S}, \mathsf{path}^{(\mathsf{seed})}\right\}$

$$\xrightarrow{\quad \sigma \quad}$$

II. Verification

Parse σ as $\left\{\mathsf{cmt}, \{\mathsf{rsp}_j\}_{j\in S}, \{\mathsf{path}_j^{(\mathsf{aux})}\}_{j\in S}, \{\mathsf{path}_j^{(\mathsf{c})}\}_{j\in S}, \mathsf{path}^{(\mathsf{seed})}\right\}$

Compute $\mathsf{seed} = \mathsf{Hash}(\mathsf{m}, \mathsf{cmt})$

Choose uniformly $S \subseteq \mathbb{Z}_M$ at random from seed, with $|S| = s$

For all $j \in S$:

1. Choose $z_j \xleftarrow{\mathsf{seed}} \mathbb{F}_q$

Set $\mathsf{ch} := \{S, \{z_j\}_{j\in S}\}$

Perform verification as in opt. GPS

Accept or reject the signature accordingly

Figure 5.11: Signature scheme based on the optimized GPS ZKID protocol.

5.3 R-CVE and R-GPS

The previously shown CVE and GPS both rely on the SDP problem. We now show that by using R-SDP instead we can save on the signature sizes. From now on, we will refer to the CVE and GPS schemes based on R-SDP as R-CVE and R-GPS, respectively.

For starters, let us define how the problem used in the schemes switches from SDP to R-SDP. While with SDP the private key was a vector $\mathbf{e} \in \mathbb{F}_q^n$ of weight t, with R-SDP the private key is $\mathbf{e} \in \mathbb{E}_0^n$ of weight t. Analogously, the vector $\widetilde{\mathbf{e}}$

Table 5.2: Comparison between optimized CVE-based and optimized GPS-based signature sizes, $\lambda = 128$

	q	n	t	M	s	N	Sign. Size (kB)
CVE (opt.)	991	220	90	–	–	129	52.5
GPS (opt.)				991	19	–	27.2
CVE (opt.)	131	220	90	–	–	130	44.5
GPS (opt.)				512	23	–	28.5
CVE (opt.)	269	207	90	–	–	129	46.5
GPS (opt.)				1024	19	–	25.3

of weight t that is randomly sampled from $\mathbb{F}_q^n$ in GPS is now sampled from $\mathbb{E}_0^n$ in R-GPS. Moreover, the scaling vector of monomial transformations that was taken in $(\mathbb{F}_q^*)^n$ is now taken in $\mathbb{E}^n$. This justifies the choice of taking $\mathbb{E}$ as a cyclic subgroup of $\mathbb{F}_q^*$. In fact, as we recall applying the monomial transformation τ to a vector $\mathbf{a}$ means multiplying each entry of the vector by an element of the scaling vector $\mathbf{v}$. If both $\mathbf{a}$ and $\mathbf{v}$ are restricted, then also the result is, because a product from two elements of $\mathbb{E}$ is still in $\mathbb{E}$. A restricted monomial transformation has a bit-size of

$$l_{z,\tau} = \begin{cases} n\left(\lceil \log_2(n)\rceil + \lceil \log_2(z)\rceil\right), & \text{if } t < n \\ n\left\lceil \log_2(z)\right\rceil, & \text{if } t = n \end{cases} \tag{5.3.1}$$

which is clearly lower than (5.2.2) as $z < q-1$. Also, a restricted vector has a lower bit-size than a regular vector over $\mathbb{F}_q$, as the former takes $n\left\lceil \log_2(z+1)\right\rceil$ bits and the latter takes $n\left\lceil \log_2(q)\right\rceil$ bits to be represented.

Both in CVE and in GPS, part of the communication cost is due to monomial transformations or transformed vectors. Therefore, as using R-SDP reduces their sizes, it is convenient to apply it to such schemes. The signature sizes of R-CVE and R-GPS are

$$|\sigma|_{\textsf{R-CVE}} = \underbrace{2\lambda}_{\textsf{c}} + N \left(\underbrace{n \lceil \log_2(z+1) \rceil}_{\mathbf{y}^{(i)}} + \underbrace{2\lambda}_{\textsf{c}^{(i)}_{b^{(i)} \oplus 1}} + \underbrace{\frac{\lambda + \psi(n,q,t)}{2}}_{f^{(i)}} \right), \tag{5.3.2}$$

$$\begin{aligned} |\sigma|_{\textsf{R-GPS}} = & \underbrace{2\lambda}_{\textsf{h}} + \underbrace{2\lambda}_{\textsf{root}^{(\textsf{c})}} + \underbrace{s(2\lambda + l_{z,\tau} + \psi(n,q,t))}_{\{\textsf{rsp}_j\}_{j \in S}} + \\ & + \underbrace{s(2\lambda \lceil \log_2(q) \rceil)}_{\{\textsf{path}^{(\textsf{aux})}_j\}_{j \in S}} + \underbrace{s(2\lambda \lceil \log_2(M) \rceil)}_{\{\textsf{path}^{(\textsf{c})}_j\}_{j \in S}} + \\ & \underbrace{\lambda \left[2^{\lceil \log_2(s) \rceil} + s\big(\lceil \log_2(M) \rceil - \lceil \log_2(s) \rceil - 1 \big) \right]}_{\textsf{path}^{(\textsf{seed})}}. \end{aligned} \tag{5.3.3}$$

5.4 R-BG

Another scheme that has been considered is the so-called BG, proposed in [12] by Bidoux and Gaborit. BG is not originally based on SDP, but instead on the *Permuted Kernel Problem (PKP)*. Briefly, for PKP the prover first samples a vector $\mathbf{e} \in \mathbb{F}_q^n$, a full rank $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, a permutation $\pi \in S_n$ and computes $\mathbf{s} = \pi(\mathbf{e})\mathbf{H}^\top$. The secret key is the permutation π and the public key is $\{\mathbf{e}, \mathbf{s}\}$. With minor modifications, the scheme can be adapted to the R-SDP setting. The only differences are that we use monomial transformations instead of permutations and that $\mathbf{e}$ and the transformation are sampled from the restricted set. Namely, once $\mathbf{H}$ has been defined, we sample $\mathbf{e}$, $\tau \xleftarrow{\$} S_n \times \mathbb{E}^n$, set the secret key as τ and the public key as $\{\mathbf{e},\ \mathbf{s} = \tau(\mathbf{e})\mathbf{H}^\top\}$. Obviously, to compress the public key size, $\mathbf{e}$ can be generated from a seed $\textsf{seed}^{(pk)}$.

In Figure 5.12 one round of R-BG is shown. SeedTree, SeedPath and Reconstruct-Seeds are the same function used in GPS.

It can be easily verified that the R-BG scheme satisfies the *completeness* property.

The first check verifies that the prover applied the secret key τ. In fact, it computes

$$\begin{aligned}\widetilde{\mathbf{s}} &= \widetilde{\mathbf{e}}_M\mathbf{H}^\top - \beta\mathbf{s} = [\tau_M(\widetilde{\mathbf{e}}_{M-1}) + \mathbf{v}_M]\mathbf{H}^\top - \beta\mathbf{s} = \\ &= [\tau_M \circ \tau_{M-1}(\widetilde{\mathbf{e}}_{M-2}) + \tau_M(\mathbf{v}_{M-1}) + \mathbf{v}_M)]\mathbf{H}^\top - \beta\mathbf{s} = \\ &= [\tau_M \circ \cdots \circ \tau_1(\widetilde{\mathbf{e}}_0) + \sum_{i=1}^{M-1} \tau_M \circ \cdots \circ \tau_{i+1}(\mathbf{v}_i) + \mathbf{v}_M]\mathbf{H}^\top - \beta\mathbf{s} = \\ &= [\tau_M \circ \cdots \circ \tau_1(\beta\mathbf{e}) + \mathbf{v}]\mathbf{H}^\top - \beta\mathbf{s}.\end{aligned}$$

If the secret key has been correctly applied, i.e. $\tau_1 = \tau_2^{-1} \circ \cdots \tau_M^{-1} \circ \tau$, then $\tau_M \circ \cdots \circ \tau_1(\beta\mathbf{e})\mathbf{H}^\top = \beta\tau(\mathbf{e})\mathbf{H}^\top = \beta\mathbf{s}$ and, therefore, $\widetilde{\mathbf{s}} = \mathbf{v}\mathbf{H}^\top$. Notice that a composition of restricted permutation is still restricted, so τ_1 should be restricted. The second check verifies the validity of $\{\widetilde{\mathbf{e}}_i\}_{1\le i\le M}$ vectors and, by extention, the validity of the seeds that produce $\mathbf{e}$, $\{\tau_i\}_{2\le i\le M}$ and $\{\mathbf{v}_i\}_{1\le i\le M}$.

It can be seen that the protocol structure is the same as BG, so it inherits all of its features. As in [12, Theorem 2], the soundness error is

$$\varepsilon = \frac{1}{M} + \frac{M-1}{M(q-1)}.$$

Once again, the security level λ is attained through multiple repetitions of the single round. To set the value of N so that the attack in [23] is mitigated, we rely on the analysis in [12, Section 4.2]. To this end, let

$$\begin{aligned}P(N', N, M) = \sum_{j=N'}^{N} \binom{N}{j} \left(\frac{1}{q-1}\right)^j \left(\frac{M-1}{M}\right)^{N-j}, \\ N^* = \arg\min_{0\le x\le N}\left\{\frac{1}{P(x,N,M)} + M^{N-x}\right\}.\end{aligned}$$

Then, we choose N so that $P(N^*, N, M)^{-1} + M^{N-N^*} > 2^\lambda$.

For the sake of brevity, we refer to [12] for the proof of the *zero knowledge* property.

The communication cost in bits of one round of the R-BG ZKID scheme is

$$\underbrace{2\lambda}_{\mathsf{c}} + \underbrace{\lceil \log_2(q-1) \rceil}_{\beta} + \underbrace{2\lambda}_{\mathsf{h}} + \underbrace{\lceil \log_2(M) \rceil}_{I} + \underbrace{(2\lambda + n \lceil \log_2(q) \rceil + \lambda \lceil \log_2(M) \rceil + l_{z,\tau})}_{\mathsf{rsp}}.$$

Actually, such expression is the upper bound of the real communication cost, as the biggest size of rsp corresponding to $I = 1$ has been used. Moreover, the execution of multiple rounds can be optimized similarly to what we presented for the optimized CVE. This means that the Prover computes for every round the commitments $\mathsf{c}^{(1)}, \dots, \mathsf{c}^{(t)}$ as in Figure 5.12 and sends only their hash digest

$$\mathsf{c} = \mathsf{Hash}\left(\mathsf{c}^{(1)}, \dots, \mathsf{c}^{(t)}, \mathsf{salt}\right),$$

where also an additional salt is inputted into the hash function and is transmitted with c. After he receives the $\beta^{(i)}$ for every round, the Prover computes all the hashes $\mathsf{h}^{(1)}, \dots, \mathsf{h}^{(t)}$ as in Figure 5.12 and sends only their hash digest

$$\mathsf{h} = \mathsf{Hash}\left(\mathsf{h}^{(1)}, \dots, \mathsf{h}^{(t)}\right).$$

The Verifier has then to recompute all the $\mathsf{c}^{(i)}$ and $\mathsf{h}^{(i)}$ of every round and then checks if their hashed values are equal to the ones sent by the Prover. With this technique, the entire communication cost becomes

$$\underbrace{5\lambda}_{\mathsf{c,h,salt}} + t\Bigg(\underbrace{\lceil \log_2(q-1) \rceil}_{\beta^{(i)}} + \underbrace{\lceil \log_2(M) \rceil}_{I^{(i)}} + \underbrace{2\lambda + n \lceil \log_2(q) \rceil + \lambda \lceil \log_2(M) \rceil + l_{z,\tau}}_{\mathsf{rsp}^{(i)}} \Bigg).$$

The signature size obtained by applying the Fiat-Shamir transform is then

$$|\sigma|_{\textsf{R-BG}} = \underbrace{5\lambda}_{\textsf{c,h,salt}} + t\Big(\underbrace{2\lambda + n\lceil\log_2(q)\rceil + \lambda\lceil\log_2(M)\rceil + l_{z,\tau}}_{\textsf{rsp}^{(i)}}\Big). \tag{5.4.1}$$

The public-key size is

$$|PK|_{\textsf{R-BG}} = \psi(n, z+1, t) + (n-k)\lceil\log_2(q)\rceil\,. \tag{5.4.2}$$

5.5 R-BG(G)

When the R-SDP with full Hamming weight is considered, an even more compact representation for restricted objects can be obtained. The idea consists of identifying a set of restricted objects with small cardinality (but not too small, since this may facilitate attacks) and admits a compact representation, preferably fast to compute. We will then use such a set to build a variant of R-BG, called R-BG(G).

5.5.1 R-SDP(G)

We will refer to the set of all diagonal matrices $\mathrm{diag}(g^{i_1}, \ldots, g^{i_n})$, with $i_j \in \{0, \ldots, z-1\}$, as the *restricted diagonal group*, which we denote by $D_n(g) \subseteq \mathbb{F}_q^{n\times n}$. Let us introduce the bijection $\ell : D_n(g) \to \mathbb{Z}_z^n$, which allows for a vector representation of the matrices in $D_n(g)$, as

$$\ell\big(\mathrm{diag}(g^{i_1}, \ldots, g^{i_n})\big) = (i_1, \ldots, i_n).$$

It is easy to see that $(D_n(g), \cdot)$ is isomorphic to $(\mathbb{E}^n, \star)$, where "$\cdot$ " denotes the standard matrix multiplication and "$\star$ " denotes the component-wise multiplication. Additionally, both are abelian (or symmetric) groups and ℓ is a group isomorphism from $(D_n(g), \cdot)$ to $(\mathbb{Z}_z^n, +)$. In other words, every element of $\mathbb{Z}_n^n$ is associated with only one element in $D_n(g)$ through ℓ. More generally, any $\mathbf{A}$ generates a cyclic subgroup $\{\mathbf{A}^i \mid i \in \mathbb{N}\} \subset D_n(g)$. Due to the isomorphism to $\mathbb{Z}_z^n$, the order of $\mathbf{A}$ is the same as the order of $\ell(\mathbf{A})$ in $(\mathbb{Z}_z^n, +)$. Indeed, for $\mathbf{A} = \operatorname{diag}(g^{i_1}, \ldots, g^{i_n})$, $\mathbf{A}^j = \operatorname{diag}\left(g^{j \cdot i_1 \pmod z}, \ldots, g^{j \cdot i_n \pmod z}\right)$. If $j = \bar{z} = ord(\mathbf{A})$, then by definition

$$\mathbf{A}^{\bar{z}} = \operatorname{diag}\left(g^{\bar{z} \cdot i_1 \pmod z}, \ldots, g^{\bar{z} \cdot i_n \pmod z}\right) = \mathbf{I}_n \pmod q.$$

This means that

$$\ell(\mathbf{A}^{\bar{z}}) = \big(\bar{z} \cdot i_1 \pmod z, \ldots, \bar{z} \cdot i_n \pmod z\big) = \big(0 \pmod z, \ldots, 0 \pmod z\big).$$

Recall that $x \in \mathbb{Z}_z$ has order $\frac{z}{\gcd(x,z)}$, where gcd is the greatest common divisor. Thus, by denoting the least common multiple as lcm, we have

$$\operatorname{ord}(\mathbf{A}) = \operatorname{lcm}\big(\operatorname{ord}(i_1), \ldots, \operatorname{ord}(i_n)\big) = \operatorname{lcm}\left(\frac{z}{\gcd(i_1,z)}, \ldots, \frac{z}{\gcd(i_n,z)}\right).$$

Notice that, if one of the i_j is coprime to z, $\mathbf{A}$ has maximum order z.

We now consider the subgroup of $D_n(g)$ whose generating set is a set of m matrices from $D_n(g)$. Namely, we choose m matrices $\mathbf{B}_1, \ldots, \mathbf{B}_m \in D_n(g)$, and define

$$G = \langle \mathbf{B}_1, \cdots, \mathbf{B}_m \rangle = \left\{ \prod_{j=1}^{m} \mathbf{B}_j^{u_j} \,\middle|\, u_i \in \{0, \ldots, z-1\} \right\}.$$

In the following, we will call G the *restricted diagonal subgroup.* To any $\mathbf{A} \in G$, we can associate a vector representation through $\ell_G : G \to \mathbb{Z}_z^m$, as follows

$$\ell_G(\mathbf{A}) = \ell_G\left(\prod_{j=1}^{m} \mathbf{B}_j^{u_j}\right) = (u_1, \ldots, u_m). \tag{5.5.1}$$

Clearly, $(G, \cdot) \subset (D_n(g), \cdot)$ is a subgroup and ℓ_G is a group homomorphism. Thus, for any $\mathbf{A} \in G, x \in \mathbb{N}$ we have $\ell_G(\mathbf{A}^x) = x\ell_G(\mathbf{A}) \pmod{z}$.

Proposition 1. *Let* $\mathbf{M}_G \in \mathbb{Z}_z^{m\times n}$ *be the matrix whose* j*-th row is* $\ell(\mathbf{B}_j)$*, and* $\mathcal{B} = \{\mathbf{u}\mathbf{M}_G \mid \mathbf{u} \in \mathbb{Z}_z^m\}$*. Then, it holds that*

1. $\ell(\mathbf{A}) = \ell_G(\mathbf{A})\mathbf{M}_G \pmod{z}$*, for any* $\mathbf{A} \in G$*;*

2. $|\mathcal{B}| = |G|$.

Proof. Let $\mathbf{B}_j = \operatorname{diag}\left(g^{i_1^{(j)}}, \dots, g^{i_n^{(j)}}\right)$, hence $\ell(\mathbf{B}_j) = \left(i_1^{(j)}, \dots, i_n^{(j)}\right)$, and $\mathbf{A} = \prod_{j=1}^m \mathbf{B}_j^{u_j} \in G$. Then, it holds that

$$\mathbf{A} = \prod_{j=1}^{m} \operatorname{diag}\left(g^{u_j i_1^{(j)}}, \dots, g^{u_j i_n^{(j)}}\right) = \operatorname{diag}\left(g^{\sum_{j=1}^m u_j i_1^{(j)}}, \dots, g^{\sum_{j=1}^m u_j i_n^{(j)}}\right).$$

By construction, the element in the j-th row and v-th column of $\mathbf{M}_G$ is $i_v^{(j)}$. Hence, for $\mathbf{u} = \ell_G(\mathbf{A}) = (u_1, \dots, u_m) \in \mathbb{Z}_z^m$ we get

$$\ell(\mathbf{A}) = \left(\sum_{j=1}^m u_j i_1^{(j)}, \dots, \sum_{j=1}^m u_j i_n^{(j)}\right) = \mathbf{u}\mathbf{M}_G \in \mathbb{Z}_z^n.$$

The second claim follows, since $\ell : D_n(g) \mapsto \mathbb{Z}_z^n$ is a bijection. $\square$

Now that we defined all the needed mathematical tools, let us use them for slightly modifying R-SDP. From now on, we focus only on restrictions $\mathbb{E} = \{g^i \mid i \in \{0, \cdots, z-1\}\}$ such that z is prime. Also, we consider only restricted diagonal subgroups G having maximum order $|G| = z^m$, because, in this case, from Proposition 1 we have the maximum number of obtainable $\mathbf{u}\mathbf{M}_G$ vectors. We now consider R-SDP with the additional constraint that the solution must be associated with an element of G. The corresponding problem is defined as follows.

Problem 4. R-SDP(G): SDP with Restricted Diagonal Subgroup G
Let $G = \langle \mathbf{B}_1, \ldots, \mathbf{B}_m \rangle$*,* $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ *and* $\mathbf{s} \in \mathbb{F}_q^{n-k}$*. Does there exist a vector* $\mathbf{e} \in \mathbb{F}_q^n$ *such that* $\mathrm{diag}(\mathbf{e}) \in G$ *and* $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$*?*

For the sake of simplicity, we will sometimes slightly abuse notation and will say $\mathbf{e} \in G$, obviously implying that $\mathrm{diag}(\mathbf{e}) \in G$. When $G = D_n(g)$, R-SDP(G) corresponds to R-SDP.

Notice that R-SDP(G) admits fewer solutions than the more general R-SDP. Consequently, we can modify the criterion to have a unique solution as

$$|G|q^{-(1-R)n} < 1$$

and since $|G| = z^m$ we get $m\log_2(z) - (1-R)n\log_2(q) \leq 0$.

At first glance, it seems like R-SDP(G) cannot be harder than R-SDP. Indeed, the solution space now is G, with size $|G| \leq z^m$, instead of $\mathbb{E}^n$, which is bigger and has size z^n. So, there may be attacks that exploit this additional constraint. To avoid this, parameters have to be chosen carefully. We refer to [5] for the study of attack strategies that exploit the knowledge of G and the following evaluation of safe parameters.

5.5.2 Using R-SDP(G)

We will now show how to use R-SDP(G) for the R-BG scheme, which we will refer to as R-BG(G). When R-SDP(G) is used, the generating set $\langle \mathbf{B}_1, \ldots, \mathbf{B}_m \rangle$ must be publicly known. In order for $|G|$ to have the maximum order z^m, it follows from Proposition 1 that $\mathbf{M}_G$ must have maximum rank m. The generating set can be easily made part of the public key: the prover samples a seed seed_G to generate a candidate for $\langle \mathbf{B}_1, \ldots, \mathbf{B}_m \rangle$ and checks if the corresponding $\mathbf{M}_G$ has maximum rank.

If this is not true, one discards the seed and restarts. When a valid seed is found, the prover samples $\mathbf{e}$ at random from G and computes the syndrome $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$, where $\mathbf{H}$ can be sampled from the seed. Notice that, as an option, one can even fix G, i.e. fix a generating set $\langle \mathbf{G}_1, \dots, \mathbf{G}_m \rangle$ and use it for every instance of the protocol.

The vector $\mathbf{e}$ is sampled from G. Recall that we are considering the full weight, so the monomial transformations are just composed of the scaling vector $\mathbf{v}$. Moreover, their action on a generic vector $\mathbf{x}$ can be simply expressed as a matrix multiplication $\mathbf{x} \cdot diag(\mathbf{v})$. Since the monomial transformations must maintain the restriction in G, also $\mathbf{v}$ has to be sampled from G. Then, let us show how to randomly draw a vector from G. To uniformly sample at random some $\mathbf{A} \in G$, one can first sample $\mathbf{u} \xleftarrow{\$} \mathbb{F}_z^m$ and then compute $\mathbf{A} = \ell_G^{-1}(\mathbf{u}) = \prod_{j=1}^m \mathbf{B}_j^{u_j}$. In practice, this can be done by first computing $\ell(\mathbf{A}) = (a_1, \dots, a_n) = \mathbf{u}\mathbf{M}_G$, which requires $O(nm)$ operations over $\mathbb{F}_z$, and then using the indices to generate the matrix as $diag\Big(\big(g^{a_1}, \dots, g^{a_m}\big)\Big)$, which requires $O(n)$ operations over $\mathbb{F}_q$. Then, the n values $diag^{-1}(\mathbf{A}) = \big(g^{a_1}, \dots, g^{a_m}\big)$ will either be considered as the elements of a restricted vector or as the diagonal of a matrix defining a restricted isometry. As it is common in ZK protocols, random objects will be communicated using the generating seed, of size λ, which a secure PRNG has been fed with.

Clearly, if a restriction is enforced, it also has to be verified. To verify that a given $\mathbf{a}$ is indeed in G, it is enough to check that $\ell(\mathbf{a})$ is a linear combination of the rows of $\mathbf{M}_G$. This can be done using a basis $\mathbf{C} \in \mathbb{F}_z^{(n-m)\times n}$ for the null space of $\mathbf{M}_G$: $\mathbf{a} \in G$, if and only if $\ell(\mathbf{a})\mathbf{C}^\top = \mathbf{0}$.

For the sake of brevity, the scheme will not be shown, as it is the same as the one in Figure 5.12 with slight modifications: $\mathbf{e}$, τ and τ_i are sampled from G. Similar adjustments can be also applied to R-GPS, thus obtaining R-GPS(G).

Considering that $|G| = z^m$, it takes $m \lceil \log_2(z) \rceil$ bits to represent a vector of

Table 5.3: Performances of the GPS scheme [21] based on different problems, $\lambda = 128$.

	q	z	n	k	w	m	N	M	Sign. Size (kB)
SDP	128		220	101	90		512	23	24.6
	256		207	93	90		1024	19	22.4
	512		196	92	84		2024	16	20.6
	1024		187	90	80		4096	14	19.5
R-SDP	67	11	147	63	147		512	24	14.8
	197	14	105	53	105		1024	19	13.4
	991	33	77	48	77		2048	16	12.9
	991	33	77	38	77		4096	14	12.5
R-SDP(G)	53	13	82	47	82	54	512	25	12.7
	103	17	76	44	76	48	1024	21	12.7
	223	37	56	33	56	34	2048	19	11.8
	1019	509	40	16	40	18	4096	14	11.5

length n from G. Therefore, the signature size of R-BG(G) is

$$|\sigma|_{\textsf{R-BG(G)}} = \underbrace{5\lambda}_{\textsf{c,h,salt}} + t\Big(\underbrace{2\lambda + n\lceil\log_2(q)\rceil + \lambda\lceil\log_2(M)\rceil + m\lceil\log_2(z)\rceil}_{\textsf{rsp}^{(i)}}\Big),$$

which is clearly less than (5.4.1) as $m < n$. The public-key size is

$$|PK|_{\textsf{R-BG(G)}} = m\lceil\log_2(z)\rceil + (n-k)\lceil\log_2(q)\rceil,$$

which is also smaller than (5.4.2). Finally, we provide sets of parameters aiming for a security level of $\lambda = 128$, for both GPS (Table 5.3) and BG in (Table 5.4) based on different problems. We can see that R-BG has signature sizes 3 kB lower than R-GPS. R-GPS(G) sizes are $1 \div 2$ kB smaller than R-GPS and R-BG(G) are $2 \div 3$ kB smaller than R-BG.

Table 5.4: Performances of the BG scheme [12] based on different problems, $\lambda = 128$.

	q	z	n	k	m	N	t	Sign. Size (kB)
PKP	997		61	33		32	42	10.0
						256	31	8.9
R-SDP	991	33	77	38		32	42	10.8
						256	31	9.5
R-SDP(G)	971	97	44	26	26	32	42	8.0
						256	31	7.4
	1019	509	40	16	18	32	42	7.7
						256	31	7.2

5.6 Comparison with other post-quantum signatures schemes

In Table 5.5 we compare the R-GPS, R-BG, R-GPS(G) and R-BG(G) schemes with other post-quantum signatures. As it is common in the literature, we have distinguished between "fast" variants (those with the lowest number of rounds, that is, with a smaller computational cost) and "short" variants (the ones with a larger number of rounds and shorter signatures). Our protocols compare very favorably with the schemes existing in the literature, even when considering the more conservative R-SDP.

For what concerns SDP, we achieve signatures that are smaller than those of all other schemes, apart from some variants of the Ret. of SDitH and WAVE. Notice that WAVE is a Hash&Sign scheme and has large public keys (more than 3MB). Our protocols, instead, use public keys of less than 0.1 kB. Notice that R-GPS achieves signature sizes 10 kB smaller than GPS, thus highlighting the advantage of using R-SDP instead of usual SDP.

Schemes based on the rank metric can achieve smaller signatures when some structure is considered (e.g. ideal codes). An exception is Durandal, which is not obtained from a ZK protocol, but has much larger public keys. An analogous situation holds for LESS-FM. Our R-BG protocol beats all existing schemes based on PKP, and has signatures that are smaller than both variants of SPHINCS$^+$. Finally, we notice that R-BG achieves better signature sizes than R-GPS and both schemes improve when their (G) counterpart is considered.

```
Public Data   Parameters $q, n, k, t \in \mathbb{N}$, parity-check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$
Private Key   $\tau \in S_n \times \mathbb{E}^n$
Public Key    $\mathbf{e} \in \mathbb{E}_0^n$ with weight $t$, $\mathbf{s} = \tau(\mathbf{e})\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

PROVER                                                                  VERIFIER

Choose $\mathsf{seed} \xleftarrow{\$} \{0,1\}^\lambda$
Compute $\{\mathsf{seed}_i\}_{1\le i\le M} = \mathsf{SeedTree}(\mathsf{seed})$
For $i = 2, \cdots, M$:
  1. Choose $\mathsf{seed}_i^*, \mathsf{salt}_i \xleftarrow{\mathsf{seed}_i} \{0;1\}^\lambda$
  2. Choose $\tau_i \xleftarrow{\mathsf{seed}_i^*} S_n \times \mathbb{E}^n$, $\mathbf{v}_i \xleftarrow{\mathsf{seed}_i^*} \mathbb{F}_q^n$
  3. Set $\mathsf{c}_i := \mathsf{Hash}\big(\mathsf{salt}_i, \mathsf{seed}_i^*\big)$
Set $\tau_1 := \tau_2^{-1} \circ \cdots \tau_M^{-1} \circ \tau$
Choose $\mathsf{seed}_1^*, \mathsf{salt}_1 \xleftarrow{\mathsf{seed}_1} \{0;1\}^\lambda$
Choose $\mathbf{v}_1 \xleftarrow{\mathsf{seed}_1^*} \mathbb{F}_q^n$
Set $\mathsf{c}_1 := \mathsf{Hash}\big(\mathsf{salt}_1, \mathsf{seed}_1^*, \tau_1\big)$
Compute $\mathbf{v} = \mathbf{v}_M + \sum_{i=1}^{M-1} \tau_M \circ \cdots \circ \tau_{i+1}(\mathbf{v}_i)$
Set $\mathsf{c} := \mathsf{Hash}\big(\mathbf{v}\mathbf{H}^\top, \{\mathsf{c}_i\}_{1\le i\le M}\big)$
                              $\xrightarrow{\mathsf{c}}$
                                                      Choose $\beta \xleftarrow{\$} \mathbb{F}_q^*$
                              $\xleftarrow{\beta}$
Set $\widetilde{\mathbf{e}}_0 := \beta\mathbf{e}$
For $i = 1, \cdots, M$:
  Set $\widetilde{\mathbf{e}}_i := \tau_i(\widetilde{\mathbf{e}}_{i-1}) + \mathbf{v}_i$
Set $\mathsf{h} := \mathsf{Hash}\big(\{\widetilde{\mathbf{e}}_i\}_{1\le i\le M}\big)$
                              $\xrightarrow{\mathsf{h}}$
                                                      Choose $I \xleftarrow{\$} \{1, \cdots, M\}$
                              $\xleftarrow{I}$
Compute $\mathsf{path} = \mathsf{SeedPath}(I, \mathsf{seed})$
If $I \neq 1$, set $\mathsf{rsp} := \{\mathsf{c}_I, \widetilde{\mathbf{e}}_I, \tau_1, \mathsf{path}\}$
Else, set $\mathsf{rsp} := \{\mathsf{c}_I, \widetilde{\mathbf{e}}_I, \mathsf{path}\}$
                              $\xrightarrow{\mathsf{rsp}}$
                                        If $I \neq 1$, check if $\tau_1 \in S_n \times \mathbb{E}^n$
                              Generate $\{\mathsf{seed}_i\}_{i\neq I} = \mathsf{RecontstructSeeds}(I, \mathsf{path})$
                                For $i \neq I$:
                                  1. Choose $\mathsf{seed}_i^*$, $\mathsf{salt}_i \xleftarrow{\mathsf{seed}_i} \{0;1\}^\lambda$
                                  2. Choose $\tau_i \xleftarrow{\mathsf{seed}_i^*} S_n \times \mathbb{E}^n$, $\mathbf{v}_i \xleftarrow{\mathsf{seed}_i^*} \mathbb{F}_q^n$
                                  3. Set $\mathsf{c}_i := \mathsf{Hash}\big(\mathsf{salt}_i, \mathsf{seed}_i^*\big)$
                                                                Set $\widetilde{\mathbf{e}}_0 = \beta\mathbf{e}$
                               For $i \neq I$:
                                 1. $\widetilde{\mathbf{e}}_i = \tau_i(\widetilde{\mathbf{e}}_{i-1}) + \mathbf{v}_i$
                                 2. If $i \neq 1$: compute $\mathsf{c}_i = \mathsf{Hash}\big(\mathsf{salt}_i, \mathsf{seed}_i^*\big)$
                                 3. Else, compute $\mathsf{c}_1 = \mathsf{Hash}\big(\mathsf{salt}_1, \mathsf{seed}_1^*, \tau_1\big)$
                                            Compute $\widetilde{\mathbf{s}} = \widetilde{\mathbf{e}}_M\mathbf{H}^\top - \beta\mathbf{s}$
                         Set $b := 1$ if $\mathsf{c} = \mathsf{Hash}\big(\widetilde{\mathbf{s}}, \{\mathsf{c}_i\}_{1\le i\le M}\big)$, $b := 0$ otherwise
                          Set $b' := 1$ if $\mathsf{h} = \mathsf{Hash}\big(\{\widetilde{\mathbf{e}}_i\}_{1\le i\le M}\big)$, $b' := 0$ otherwise
                                            Accept if $b \wedge b' = 1$, reject otherwise
```

Figure 5.12: One round of the R-BG protocol.

Table 5.5: Comparison between signature schemes based on different post-quantum problems for $\lambda = 128$. All sizes are expressed in kB.

Problem	**Scheme**	**Pk size**	**Sign. size**	**Pk+Sign. size**	**Variant**
Hamming SDP, low weight	GPS[21]	0.1	24.0	24.1	Fast
		0.1	19.8	19.9	Short
	FJR[17]	0.1	22.6	22.7	Fast
		0.1	16.0	16.1	Short
	SDItH[18]	0.1	11.5	11.6	Fast
		0.1	8.3	8.4	Short
	Ret. of SDitH[1]	0.1	12.1	12.1	Fast, Var.3
		0.1	5.7	5.8	Shortest, Var.3
Hamming SDP, large weight	WAVE[15]	3200	2.1	3202	-
Code Equivalence	LESS-FM[6]	10.4	11.6	23.0	Balanced
		205.7	5.3	211.0	Short sign
Rank Syndrome Decoding	Fen[16]	0.1	11.0	11.1	Fast
		0.1	8.5	8.6	Short
	BG[12]	0.1	17.2	17.3	Fast
		0.1	12.6	12.7	Short
	Durandal[2]	15.2	4.1	19.3	-
Ideal Rank Syndrome Decoding	BG[12]	0.1	12.6	12.7	Fast
		0.1	10.2	10.3	Short
Ideal Rank Support Learning	BG[12]	0.5	8.4	8.9	Fast
		0.5	6.1	6.6	Short
MinRank	Fen[16]	18.2	9.3	27.5	Fast
		18.2	7.1	25.3	Short
MinRank with Linearized Poly	Fen[16]	18.2	7.2	25.4	Fast
		18.2	5.5	23.7	Short
Rank Syndrome Dec. with Linearized Poly	Fen[16]	0.9	7.4	8.3	Fast
		0.9	5.9	6.8	Short
PKP	Beu[11]	0.1	18.4	18.5	Fast
		0.1	12.1	12.2	Short
	Fen[16]	0.1	16.4	16.5	Fast
		0.1	12.8	12.9	Short
	BG[12]	0.1	9.8	9.9	Fast
		0.1	8.8	8.9	Short
Hash collisions	SPHINCS$^+$[10]	<0.1	16.7	16.7	Fast
		<0.1	7.7	7.7	Short
R-SDP	R-GPS	0.1	14.8	14.9	Fast
		0.1	12.5	12.6	Short
	R-BG	0.1	10.8	10.9	Fast
		0.1	9.5	9.6	Short
R-SDP(G)	R-GPS(G)	0.1	12.7	12.8	Fast
		0.1	11.5	11.6	Short
	R-BG(G)	0.1	7.7	7.8	Fast
		0.1	7.2	7.3	Short

Chapter 6

Implementation of restricted GPS signature scheme

In this chapter, the implementation of a proof-of-concept GPS signature scheme (Figure 5.11) is introduced. For the sake of brevity, only the main and most particular functions will be presented. The proof-of-concept does not aim to be efficient, but instead is useful to check if the scheme can be successfully implemented, i.e. if the soundness property is correctly verified in practice. Therefore, coding considerations and strategies will not be discussed in detail. The entire code can be found in the Appendix. The code has been developed with SageMath, a free and open-source mathematical software system based on the Python programming language.

6.1 FindIsometry

Given two vectors $\mathbf{e}$ and $\widetilde{\mathbf{e}}$ as inputs, the function FindIsometry outputs a monomial transformation such that $\mathbf{e} = \tau(\widetilde{\mathbf{e}})$. Briefly recalling the definition of τ, its action can be expressed as

$$\mathbf{e} = (e_0, \dots, e_{n-1}) = \tau(\widetilde{\mathbf{e}}) = \left(v_{\pi(0)}\widetilde{e}_{\pi(0)}, \dots, v_{\pi(n-1)}\widetilde{e}_{\pi(n-1)}\right).$$

We refer to $\mathbf{e}$ and $\widetilde{\mathbf{e}}$ supports as $\mathcal{S}$ and $\widetilde{\mathcal{S}}$, respectively.

After obtaining the indexes of $\mathcal{S}$ and $\widetilde{\mathcal{S}}$, all the possible permutations of the set of $\widetilde{\mathcal{S}}$ indexes are defined and one of them is picked at random. The same is done for $\widetilde{\mathcal{S}}^C$. Such two permutations are then joined to form a single random permutation vector of indexes π, making sure that the permutation of $\widetilde{\mathcal{S}}$ lands on $\mathcal{S}$.

The scaling vector is then computed. For the indexes i outside of $\mathcal{S}$, the scalars $v_{\pi(i)}$ are chosen randomly over $\mathbb{E}$ as they will be multiplied by $\widetilde{e}_{\pi(i)} = 0$. For indexes inside of $\mathcal{S}$, the scalars $v_{\pi(i)}$ are computed as $e_i/\widetilde{e}_{\pi(i)}$.

6.2 SeedPath

Given the list $S \subseteq \mathbb{Z}_M$ of selected instances and the root seed, SeedPath outputs the seed path that does not disclose $\{\mathsf{seed}_i\}_{i\in S}$. Every node in the seed path is defined as a *Node* class object whose attributes are:

- Value;
- Layer index;
- Index within the layer;
- Direction (left or right node).

Let us refer to the situation of Figure 5.9a, with $M = 8$, $s = 2$ and $S = [2, 6]$. The first step consists of creating a temporary seed path for the first challenge instance, which is 2 in this case (Figure 6.1a). Then, from the root layer to the leaves layer, we remove nodes that do not match with the rest of the challenge instances. For example, the node $T_{1,1}$ would reveal seed_6, therefore we substitute it with its child nodes (Figure 6.1b). Analogously, at the next iteration $T_{2,3}$ will be substituted with seed_7 (Figure 6.1c), thus ending the algorithm.

6.3 ReconstructSeeds

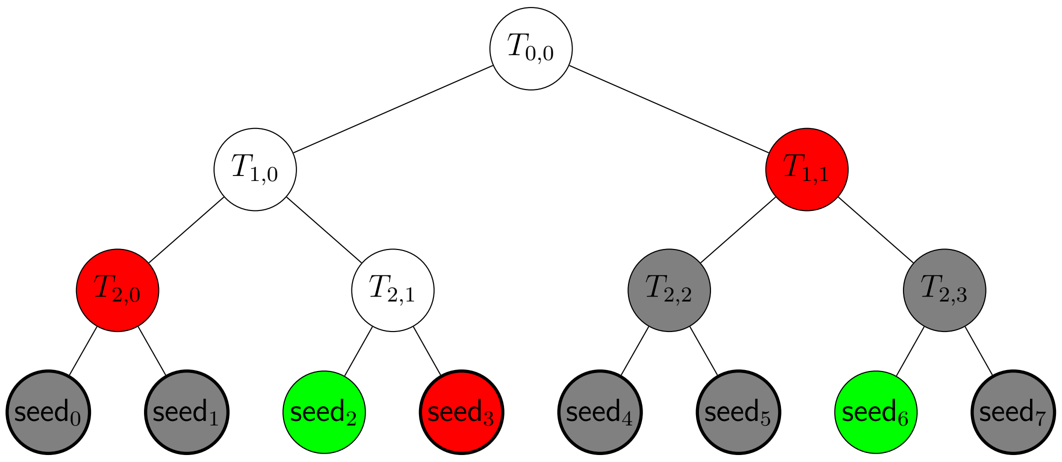
Given the list $S \subseteq \mathbb{Z}_M$ of selected instances and the seed path, ReconstructSeeds outputs $\{\mathsf{seed}_i\}_{i \notin S}$.

Let us refer to the situation of Figure 5.9c. Starting from the first node on the left, $T_{1,0}$, we compute by hashing every layer until we get to the leaves layer (Figure 6.2a). Then, we do the same for the next node to the right, $T_{2,3}$ (Figure 6.2b).

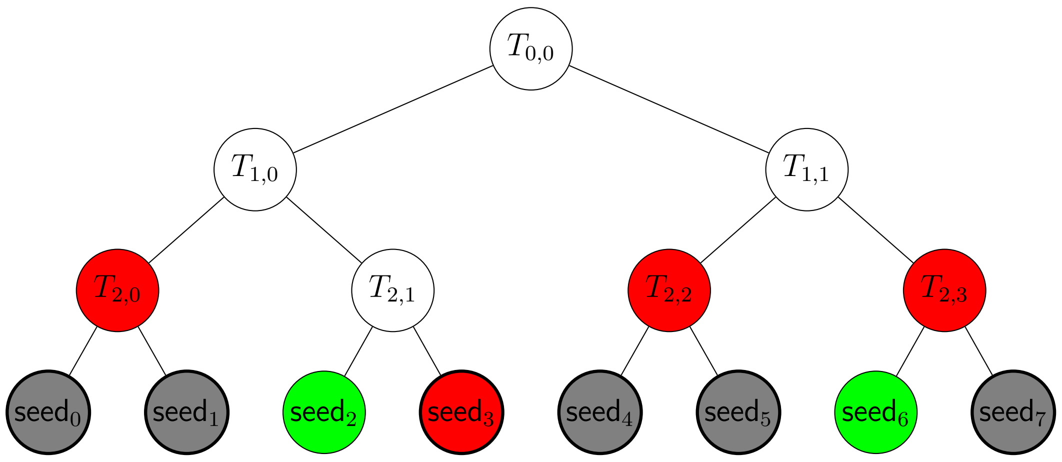
6.4 Results

In this section, the results of the R-GPS scheme implementation are presented. In Figure 6.3 an example of script execution is shown. As we can notice, the completeness property of the scheme is verified, thus the implementation has been successful.

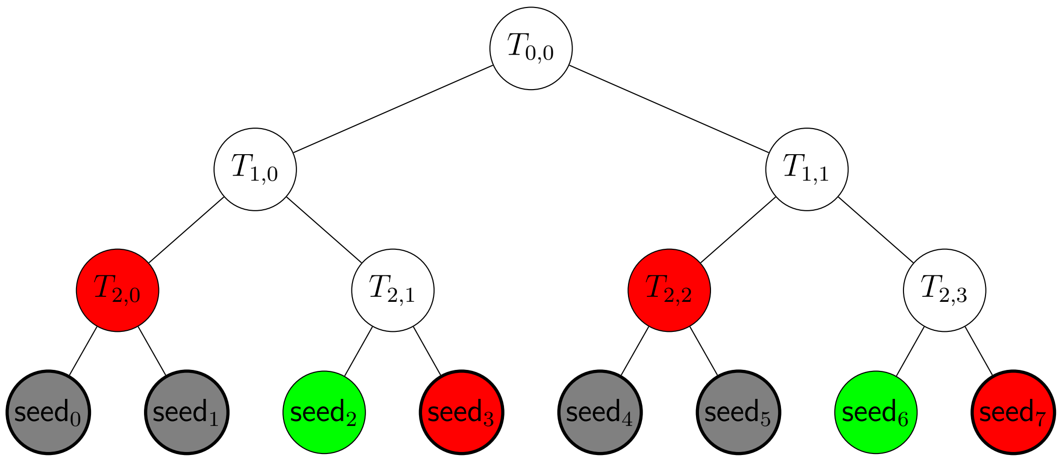
For the signature and verification timings, we get timings ranging from some minutes to tens of minutes, depending on the parameters. Clearly, such timings are large. But again, this is a proof-of-concept implementation, which is not written in a highly-performance programming language, i.e. C, nor optimized. Thus, the obtained times are expected. We also noticed that timings grow with q. This is because the number of aux grows, therefore more hashes have to be computed and hashing is the bottleneck of the scheme. Obviously, verification is always faster than signing as the Verifier performs fewer operations than the Signer.



(a) Temporary seed path of seed_2.

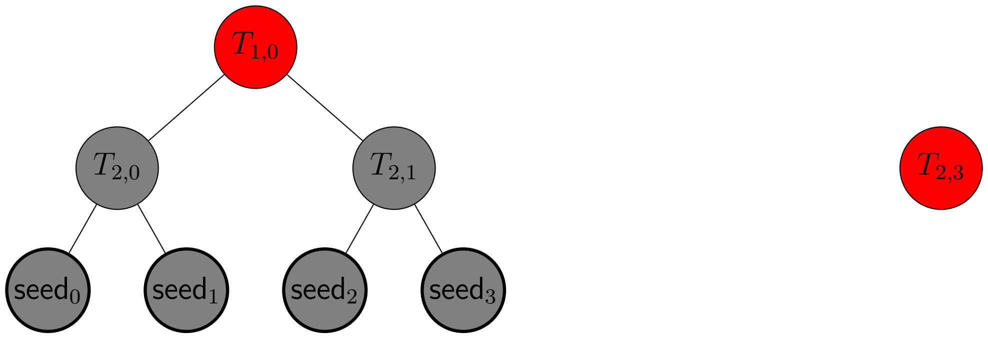


(b) Substituting $T_{1,1}$ with its child nodes.

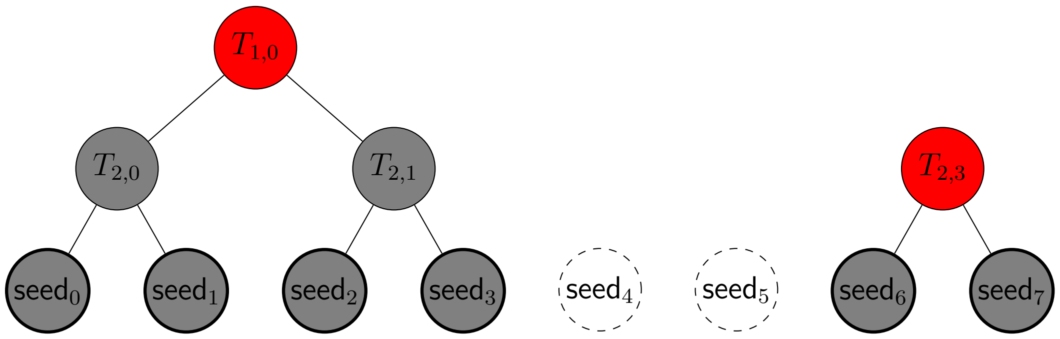


(c) Substituting $T_{2,3}$ with seed_7.

Figure 6.1: Example of $\mathsf{SeedPath}$ execution for $M = 8$, $s = 2$ and $S = [2, 6]$.

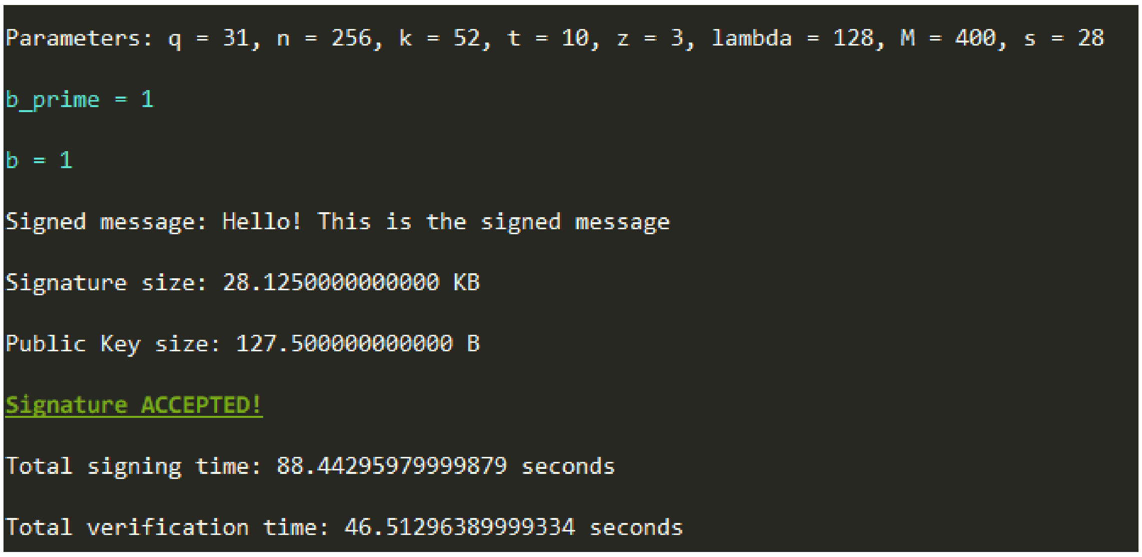


(a) Computing the seeds from the first node on the left.



(b) Continuing with $T_{2,3}$. The seeds that cannot be recomputed are shown in dotted line.

Figure 6.2: Example of $\mathsf{ReconstructSeeds}$ execution for $M = 8$, $s = 2$ and $S = [4, 5]$.



```
Parameters: q = 31, n = 256, k = 52, t = 10, z = 3, lambda = 128, M = 400, s = 28

b_prime = 1

b = 1

Signed message: Hello! This is the signed message

Signature size: 28.1250000000000 KB

Public Key size: 127.500000000000 B

Signature ACCEPTED!

Total signing time: 88.44295979999879 seconds

Total verification time: 46.51296389999334 seconds
```

Figure 6.3: Example of script execution.

Just as a term of comparison, with a proof-of-concept C implementation of R-BG we obtain 19.2 ms for the signing and 18.0 ms for the verification, while a more optimized C implementation gives 2.1 ms for the signing and 2.0 ms for the verification.

Chapter 7

Conclusion and future work

New approaches to code-based digital signatures have been studied in this work. In particular, we focused on zero-knowledge ID schemes and then applied the Fiat-Shamir transform to obtain ZKID-based signature schemes. Information Set Decoder generic solvers for Syndrome Decoding Problem have been analyzed to understand how the best algorithm such as Stern and BJMM perform.

A relatively new problem based on sets of restricted elements, R-SDP, has been introduced. New ISD solvers have been developed for this setting, where the appropriate choice of the elements search space aims to increase the number of representations while keeping the lists as small as possible. For low values of z it is also possible to further exploit the algebraic structure of $\mathbb{E}$, making attacks even more successful. It is then clear that even or small orders of the restricted set have to be avoided.

We explored some of the already existing ZKID-based signature schemes, such as CVE, GPS and BG. By applying R-SDP to these schemes, we highlighted a saving in signature sizes. This happens because the restricted objects, i.e. vectors and monomial transformations, require fewer bits to be represented than their counterparts over $\mathbb{F}_q$. Moreover, the restriction implicates the existence of fewer solutions, so the instance weight can be maximum for z small enough. In this case, transformations can be represented with just the scaling vector, so this results in a further bit saving and faster computations as mere component-wise multiplications are required. More importantly, higher weights make ISD more difficult. Therefore, smaller codes can be used, thus getting even smaller sizes. The R-GPS scheme achieves signature sizes of the order of 12 kB, which is 8 to 10 kB times less than GPS. This underlines the strength of the applied restriction.

The proof-of-concept Python implementation of R-GPS shows that the scheme can correctly work, even if it requires the execution of non-trivial subroutines for the management of Merkle Trees and binary trees. The timings are large but expected for a Python proof-of-concept. This is due to the large number of hashes that the scheme has to perform and this number grows with q. However, a fully optimized version of R-GPS will surely yield much better timings. The considered schemes will be implemented in an optimized way, as they will be the subject of a NIST submission for the fourth round of the standardization contest for signature schemes.

For full-weight instances, an even newer problem has been introduced, R-SDP(G), based on a restriction of the set of diagonal matrices with entries in $\mathbb{E}$, showing that it leads to even smaller signatures when applied to the considered schemes. The obtained signature sizes in the order of 7 kB are highly competitive with respect to other state-of-the-art solutions.

Future work will aim to possibly find better-performing alternatives than R-BG and to choose optimal parameters for R-SDP, both for security and implementation aspects. The schemes have then to be developed in C language in a highly-optimized fashion in order to cut the timings down as much as possible and be competitive even with schemes that show smaller signature and/or public key sizes.

Appendix

7.1 opt_test_MPC.sage

```
reset()

import hashlib
import numpy
import sys
from time import perf_counter

load('/path/to/opt_MPC_utils.sage')
load('/path/to/merkletools.sage')

#Scheme parameters
q = 997           #Select prime integer q (size of the field)
n = 256          #Lenght of the code
r = 204          #Redundancy of the code (n-k)
w = 10           #Private key weight
z = 83           #Restricted set size
_lambda = 128    #Seed bit lenght
M = 400          #Number of instances
```

```
s_num = 20      #Number of chosen rounds (|S|)
mex = "Ciao!"   #Message to be signed
avg = True      #True if average measurements over 5 executions are
↪ needed, False otherwise

#Preparations to execute the protocol
Fq=GF(q) #Finite field with q elements
Fq_set = Set(Fq)
Fq_star = Fq_set.difference([0]) #Multiplicative group of Fq
E = Set(get_E()) #Restricted set

#Key generation
e, Htr_unsys, s, pk_seed = key_gen()
for i in range(len(e)):
    assert e[i] in E or e[i] == 0, "ERROR: e is not restricted!"


#MPC-in-the-head identification
ok, sig_size, tot_time = MPC_id(e,Htr_unsys,s,mex,s_num)
if avg:
    for i in range(4):
        ok_tmp, sig_size_tmp, tot_time_tmp =
        ↪ MPC_id(e,Htr_unsys,s,mex,s_num)
        sig_size += sig_size_tmp
        tot_time += tot_time_tmp
    sig_size = sig_size/5
    tot_time = tot_time/5
```

```

print("\nSigned message: "+mex)
print("\nSignature size: "+str(sig_size/1024.)+" KB")
print("\nPublic Key size (seed): "+str(sys.getsizeof(pk_seed))+" B")
if ok==1:
    print(Bcolors.BOLD+Bcolors.UNDERLINE+Bcolors.OKGREEN+"\nSignature
    ↪  ACCEPTED!\n"+Bcolors.ENDC+Bcolors.ENDC);
else:
    print(Bcolors.BOLD+Bcolors.UNDERLINE+Bcolors.FAIL+"\nSignature
    ↪  REJECTED!\n"+Bcolors.ENDC+Bcolors.ENDC);

print("Total execution time: "+str(tot_time)+" seconds\n")
```

7.2 opt_MPC_utils.sage

```
#MPC functions

from random import seed

class Bcolors:
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKCYAN = '\033[96m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'

#Class that represents a node of the seed tree
class Node:

    def __init__(self,val='',lev=-1,ind=-1,dir=''):
        self.val = val  #Value of the node
        self.lev = lev  #Level within the tree of the node
        self.ind = ind  #Index of the node in that level
        self.dir = dir  #Direction of the node (left or right)

    def level(self):
        return self.lev

    def value(self):
        return self.val

    def index(self):
        return self.ind

    def direction(self):
        return self.dir

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

#Obtaining restricted set
```

```
def get_E():

    assert is_prime(q),"q is not prime!"
    assert (q-1)%z == 0, "z does not divide q-1!"

    alpha = Fq.primitive_element()
    exp = (q-1)/z
    g = alpha**exp
    return [g^l for l in range(z)]

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

#Key generation
def key_gen():

    set_random_seed()
    pk_seed = initial_seed() #Save the seed that generates the public
    ↪  key H
    Htr_unsys = random_matrix(Fq,n-r,r) #public matrix (only
    ↪  non-systematic portion)
    set_random_seed() #Set a random seed that generates the private
    ↪  key e so it can't be reconstructed from pk_seed
    e = FWV()

    #compute syndrome
    s = e[0:r] + e[r:n]*Htr_unsys

    return e,Htr_unsys,s,pk_seed

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

#Generate a random vector with weight w and lenght n
def FWV():

    a = vector(Fq,w)

    for i in range(0,w):
        a[i] = E.random_element()

    b = zero_vector(n) #List of zeros of lenght n
    P = Permutations(range(0,n))
    rnd_supp = P.random_element()

    for i in range(0,w):
        b[rnd_supp[i]] = a[i]

    return b

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

#Find an isometry function tau such that e = tau(e_tilde)
def FindIsometry(e,e_tilde):

    supp_e = []
    supp_e_tilde = []

    for i in range(0,n):
        if e[i]!=0:
            supp_e.append(i)
        if e_tilde[i]!=0:
            supp_e_tilde.append(i)

    P_supp = Permutations(supp_e_tilde)
    tau_perm_supp = P_supp.random_element()

    zeros_e_tilde = [x for x in list(range(0,n)) if x not in
    ↪  supp_e_tilde] #Complement of e_tilde support
    P_zeros = Permutations(zeros_e_tilde)
    tau_perm_zeros = P_zeros.random_element()

    #Joining the permutations of e_tilde support and of the
    ↪  complement of e_tilde support, according to e support
    tau_perm = [] #Total permutation
    j = 0
    k = 0

    for i in range(0,n):
        if i in supp_e:
            tau_perm.append(tau_perm_supp[j])
            j = j + 1
        else:
                tau_perm.append(tau_perm_zeros[k])
                k = k + 1

    tau_values = vector(Fq,n)

    for i in range(0,n):
        #Find value vector through reciprocal calculation
        if e_tilde[tau_perm[i]] == 0:
            tau_values[tau_perm[i]] = E.random_element()
        else:
            tau_values[tau_perm[i]] = Fq(e[i]/e_tilde[tau_perm[i]])

    return tau_perm, tau_values

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

#Apply monomial transformation (isometry)
def apply_rest_monomial(tau_perm,tau_values,a):

    b = vector(Fq,n)

    for i in range(0,n):
        p = tau_perm[i]
        b[i] = tau_values[p]*a[p]

    return b

```

```
####################################################################

#Hash input
def hash(input):

    dig = hashlib.sha256()
    dig.update(input.encode('utf-8'))
    dig = dig.hexdigest()

    return dig

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

#Compute the hash of an input node and separates the result in two
↪  parts
def hash_forking(input):

    dig = hash(input)
    sx = dig[:len(dig)//2]
    dx = dig[len(dig)//2:]

    return sx, dx

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

#Generates the M seeds as leaves of a binary tree obtained by a
↪  starting seed
```

```python
def SeedTree(starting_seed):

    last_level = [str(starting_seed)]

    while len(last_level) < M:

        next_level = []

        for i in last_level:
            sx, dx = hash_forking(i)
            next_level.append(sx) # Sibling node (SX)
            next_level.append(dx) # Sibling node (DX)

        last_level = next_level

    return last_level[0:M] # Return the M required seeds

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

#Returns the seed path of S-th seeds which are the S-th leaves of the
↪ binary tree obtained from starting_seed
def SeedPath(S,starting_seed):

    path = []

    #Get number of leaves of the seeds binary tree
    n_leaves = 1
    while n_leaves < M:
        n_leaves = 2*n_leaves
    indexes = list(range(0,n_leaves)) #List of leaves indexes

    #First, we get the path of just the first seed whose index is in S
    last_level = str(starting_seed)   #Root node

    levels = log(n_leaves,2)

    lv_counter = levels
    last_index = 0
    while not len(indexes) == 1:
        lv_counter -= 1
        sx, dx = hash_forking(last_level)
        if S[0] in indexes[:len(indexes)//2]: #S[0] is on the left part
            path.append(Node(dx,lv_counter,2*last_index+1,'right'))
            last_index = 2*last_index
            last_level = sx     #New parent node
            del indexes[len(indexes)//2:] #Eliminate indexes we don't need anymore
        else: #S[0] is on the right part
            path.append(Node(sx,lv_counter,2*last_index,'left'))
            last_index = 2*last_index + 1
            last_level = dx     #New parent node
            del indexes[:len(indexes)//2] #Eliminate indexes we don't need anymore

    assert indexes[0] == S[0], "Seed Path computation error!"

    for i in S:
        if not i == S[0]:
            indexes = [list(range(0,n_leaves/(2^x))) for x in range(0,levels)] #List of node index lists for every level
            for j in range(levels-1,-1,-1): #j is the level index (from levels-1 to 0 (leaves level) included)
                #i is on the left part
                if i in indexes[0][:len(indexes[0])//2]:
                    #Cut the indexes in half at every level
                    for ind in indexes: #Eliminate indexes we don't need anymore
                        if len(ind) == 1:
                            del ind[0]
                        else:
                            del ind[len(ind)//2:]
                    for x in path:
                        if x.level()==j and x.index()==indexes[j][0]:
                            #Remove the node and add its child (only if not included in S)
                            path.remove(x)
                            #Calculate child nodes
                            sx, dx = hash_forking(x.value())
                            if j-1 == 0 and not indexes[j-1][0] in S or j-1 > 0:
                                #Left child node
                                path.append(Node(sx,j-1, indexes[j-1][0],'left'))
                            if j-1 == 0 and not indexes[j-1][1] in S or j-1 > 0:
                                #Right child node
                                path.append(Node(dx,j-1, indexes[j-1][1],'right'))
                #i is on the right part
                else:
                    #Cut the indexes in half at every level
                    for ind in indexes: #Eliminate indexes we don't need anymore
                        if len(ind) == 1:
                            del ind[0]
                        else:
                            del ind[:len(ind)//2]
                    for x in path:
                        if x.level()==j and x.index()==indexes[j][0]:
                            #Remove the node and add its child (only if not included in S)
                            path.remove(x)
                            #Calculate child nodes
                            sx, dx = hash_forking(x.value())
```

```python
                            if j-1 == 0 and not indexes[j-1][0] in S or j-1 > 0:
                                #Left child node
                                path.append(Node(sx,j-1, indexes[j-1][0],'left'))
                            if j-1 == 0 and not indexes[j-1][1] in S or j-1 > 0:
                                #Right child node
                                path.append(Node(dx,j-1, indexes[j-1][1],'right'))

    return path

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

#Function that takes a list a of elements as input, generates a Merkle Tree from it and returns the call to the constructor
def MerkleTree(input):

    a = input

    #https://stackoverflow.com/questions/18754180/create-multiple-instances-of-a-class
    mt = MerkleTools()

    counter = 0
    while not log(len(a),2)==ceil(log(len(a),2)):
        counter += 1
        #We add 'custom' leaves to get the right number of leaves (power of 2). This is done by hashing the last leaf (we can't add random leaves because this would
        #cause the reconstructed tree (created by Verifier) to be different to the first one (created by Prover) in the custom leaves if the set seed isn't the same)
        a.append(hash(a[-1]))

    d = log(len(a),2)

    for i in range(0,2^(d)):
        mt.add_leaf(a[i]) #Not automatically hashed, the input a is intended to be already hashed

    mt.make_tree()

    #Removing custom leaves now that we no longer need them
    for i in range(0,counter):
        a.pop(-1)

    return mt

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

```

```python
#Outputs the list of reconstructed seeds from the seed path obtained
↪  from SeedPath (obviously, seeds with j in S are not included in
↪  the list)
def ReconstructSeeds(S,path_seed):

    #Get number of leaves of the seeds binary tree
    n_leaves = 1
    while n_leaves < M:
        n_leaves = 2*n_leaves

    seed = ['']*n_leaves #Final list of seeds we are looking for

    for node in path_seed:

        #First, we have to reconstruct the leaves associated to each
        ↪  node in path_seed
        last_level = [node]

        for i in range(0,node.level()): #As said before, i is the
        ↪  level of the tree where the digest is, so is also the
        ↪  number of hash forkings that we have to compute

            next_level = []
            next_level_indexes = []

            for x in last_level:
                sx, dx = hash_forking(x.value())
                next_level +=
                 ↪  [Node(sx,x.index()-1,2*x.index(),'left'),
                 ↪  Node(dx,x.index()-1,2*x.index()+1,'right')]
                next_level_indexes += [2*x.index(), 2*x.index()+1]

            last_level = next_level

        if node.level() == 0:
            next_level_indexes = [node.index()]

        #After recontructing the leaves, we have to place them in the
         ↪  final seed list
        counter = 0
        for i in next_level_indexes:
            seed[i] = last_level[counter].value()
            counter += 1

    test = 1
    for i in S:
        if not seed[i] == '':
            test = 0
    if test and len(seed)==n_leaves:
        return seed
    else:
        raise Exception("Seed recomputation error!")

####################################################################
```

```
#Here the actor (Helper, Prover, Verifier) classes description
↪  begins.
#If the comments above the method definitions are in:
# - lower case letters: the method is one of the steps of the first
↪  MPC-in-the-head scheme (the one with the Helper)
# - upper case letters: the method is one of the steps of the second
↪  MPC-in-the-head scheme (the one without the Helper), which use
↪  the "lower case" methods

#HELPER
class Helper:

    #Setup (H)
    def H(seed):

        set_random_seed(seed)
        u_helper = random_vector(Fq,n)
        e_tilde_helper = FWV()

        r_v = matrix(GF(2),q,_lambda)
        c_v = [""]*len(Fq)

        set_random_seed(seed)
        for v in Fq:
            r_v[v] = random_vector(GF(2),_lambda)
            c_v[v] = hash(str(r_v[v])+str(u_helper+v*e_tilde_helper))

        aux = c_v

        return aux

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

#PROVER
class Signer:

    def __init__(self,hash_leaf=False,s_num=0):

        self.s_num = s_num
        self.hash_leaf = hash_leaf # True to hash the input leaves,
         ↪  False otherwise

        self.u = []
        self.e_tilde = []
        self.tau_perm = []
        self.tau_values = []
        self.r_vec = []

        self.aux = []                               # list of M aux
         ↪  lists of strings
        self.aux_hash = [[] for x in range(M)]      # hashed version
         ↪  of aux (leaves must be obtained by hashing the aux
         ↪  values)
        self.c = []                                 # list of M
        ↪  commitments
        self.c_hash = []                            # hashed version
        ↪  of c
        self.root_aux = [""]*M                      # list of the M
        ↪  roots of each aux tree
        self.T_aux = []                             # list of calls
        ↪  to the T_aux merkle trees constructors

    #Commitment (P1)
    def P1(self,Htr_unsys,e,seed):

        set_random_seed(seed)
        self.u.append(random_vector(Fq,n))
        self.e_tilde.append(FWV())
        tau_p, tau_v = FindIsometry(e,self.e_tilde[-1])
        self.tau_perm.append(tau_p)
        self.tau_values.append(tau_v)
        self.r_vec.append(random_vector(GF(2),_lambda))
        tau_u = apply_rest_monomial(self.tau_perm[-1],
        ↪  self.tau_values[-1],self.u[-1])

        c = hash(str(self.r_vec[-1])+str(self.tau_perm[-1])+
        ↪  str(self.tau_values[-1])+
        ↪  str(tau_u[0:r]+tau_u[r:n]*Htr_unsys))

        return c

    #Response (P2)
    def P2(self,ch,seed,I):

        set_random_seed(seed)
        r_v_prover = matrix(GF(2),q,_lambda)

        for v in Fq:
            r_v_prover[v] = random_vector(GF(2),_lambda)

        r_z = r_v_prover[ch]
        y = self.u[I] + ch*self.e_tilde[I]
        rsp =
        ↪  [self.r_vec[I],r_z,self.tau_perm[I],self.tau_values[I],y]

        return rsp

    #I. COMMITMENT
    def Commitment(self,Htr_unsys,e):

        set_random_seed()
        self.starting_seed = initial_seed()          # Random starting
        ↪  seed, root of the seed tree
        self.seed = SeedTree(self.starting_seed)     # list of M seeds
        ↪  obtained from the seed tree

        for i in range(0,M):
            self.aux.append(Helper.H(int(self.seed[i],16)))
            if self.hash_leaf:
                for j in range(0,len(self.aux[i])):
                    self.aux_hash[i].append(hash(self.aux[i][j]))
            else:
                self.aux_hash[i] = self.aux[i] #We don't do the hash
                 ↪  of the leaves
            mt = MerkleTree(self.aux_hash[i])
            self.T_aux.append(mt)
            self.root_aux[i] = mt.get_merkle_root()
            self.c.append(self.P1(Htr_unsys,e,int(self.seed[i],16)))

        h = hash(''.join(self.root_aux)) # Total hash of aux trees
         ↪  roots
        if self.hash_leaf:
            for i in range(0,len(self.c)):
                self.c_hash.append(hash(self.c[i]))
        else:
            self.c_hash = self.c

        self.T_c = MerkleTree(self.c_hash)
        mt = self.T_c
        root_c = mt.get_merkle_root()

        return h, root_c

    #II. CHALLENGE
    def Challenge(self,mex,h,root_c):

        S = []      # list of s_num selected rounds (from 0 to M-1)
        z = []      # list of s_num selected aux (from 0 to q-1)

        set = Set(Integers(M))
        set_random_seed(Integer(hash(str(mex)+str(h)+
        ↪  str(root_c)),16))

        for i in range(0,self.s_num):
            S.append(int(set.random_element()))
            set = set.difference([S[i]])
            z.append(Fq.random_element())

        return S, z

    #III RESPONSE
    def Response(self,z,S):

        rsp_S = []                # list of s responses
        path_aux_S = []           # list of s aux for the selected
        ↪  rounds
        path_c_S = []             # list of s paths of the commitments
        ↪  (c) for the selected rounds

        for i in S:
            idx = S.index(i)      #Index of i within the list S
```

```
                rsp_S.append(self.P2(z[idx],int(self.seed[i],16),i))

                mt = self.T_aux[i]
                path_aux_S.append(mt.get_proof(int(z[idx])))
                mt = self.T_c
                path_c_S.append(mt.get_proof(int(i)))

        path_seed = SeedPath(S,self.starting_seed)

        return rsp_S, path_aux_S, path_c_S, path_seed

    #Main execution of the Signer
    def main(self,Htr_unsys,e,mex):

        #I. COMMITMENT
        h, root_c = self.Commitment(Htr_unsys,e)

        #II. CHALLENGE
        S, z = self.Challenge(mex,h,root_c)

        #III. RESPONSE
        rsp_S, path_aux_S, path_c_S, path_seed = self.Response(z,S)

        #Signature
        sigma = [h,root_c,rsp_S,path_aux_S,path_c_S,path_seed]

        return sigma
```

```

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

#VERIFIER
class Verifier:

    def __init__(self,hash_leaf,s_num):

        self.hash_leaf = hash_leaf  # True to hash the input leaves,
        ↪  False otherwise
        self.s_num = s_num                    # Number of rounds to
        ↪  select

        self.T_aux_bar = []                                        #
        ↪  list of calls to the T_aux_bar merkle trees constructors
        self.aux_bar_not_S = ['']*M                                #
        ↪  list of the M aux_bar reconstructed by every seed not
        ↪  included in S (each entry is a list of length q)
        self.aux_bar_not_S_hash = [[] for x in range(M)]          #
        ↪  hashed version of aux_bar_not_S (leaves must be obtained
        ↪  by hashing the aux values)
        self.root_aux_bar = ['']*M                                 #
        ↪  list of the roots of all the T_aux_bar merkle trees

    #Verification (V2)
    def V2(self,Htr_unsys,s,aux,c,rsp):

        tau_y = apply_rest_monomial(rsp[2],rsp[3],rsp[4])
        t = tau_y[0:r]+tau_y[r:n]*Htr_unsys - self.z*s

        c_verifier = hash(str(rsp[0])+str(rsp[2])+str(rsp[3])+str(t))

        #The 2nd and 3rd conditions check if tau is an isometry
        if c_verifier == c and numpy.unique(rsp[2]).size == n and 0
        ↪  not in rsp[3]:

            c_z_verifier = hash(str(rsp[1])+str(rsp[4]))
            c_z = aux[self.z]

            if c_z_verifier == c_z:
                return 1
            else:
                return 0
        else:
            return 0

    #II. VERIFICATION
    def Verification(self,Htr_unsys,s,h,root_c,rsp_S,path_aux_S,
    ↪  path_c_S,path_seed):

        seed_bar = ReconstructSeeds(self.S,path_seed)
        b = 1
        for j in range(0,M):
            if j in self.S:
```

```python
        idx = self.S.index(j) #Index of j in the self.S list
        tau_y_S = apply_rest_monomial(rsp_S[idx][2], rsp_S[idx][3],rsp_S[idx][4])
        t_S = tau_y_S[0:r]+tau_y_S[r:n]*Htr_unsys - self.z[idx]*s
        c_S = hash(str(rsp_S[idx][0])+str(rsp_S[idx][2])+ str(rsp_S[idx][3])+str(t_S))

        if self.hash_leaf:
            c_S_hash = hash(c_S)
        else:
            c_S_hash = c_S

        mt = MerkleTools()
        if not int(mt.validate_proof(path_c_S[idx], c_S_hash, root_c)): #validate_proof() is equal to get_merkle_root() (the function that they call "ReconstructRoot" in the paper) + check with target root
            b = 0

        c_z_S = hash(str(rsp_S[idx][1])+str(rsp_S[idx][4]))
        if self.hash_leaf:
            c_z_S_hash = hash(c_z_S)
        else:
            c_z_S_hash = c_z_S

```

```
                self.root_aux_bar[j] =
                ↪  mt.get_root_from_path(path_aux_S[idx],
                ↪  c_z_S_hash)
                self.T_aux_bar.append('') #T_aux_bar must have j
                ↪  entries, even if this one won't be used
            else:
                self.aux_bar_not_S[j] = Helper.H(int(seed_bar[j],16))

                if self.hash_leaf:
                    for i in range(0,len(self.aux_bar_not_S[j])):
                        self.aux_bar_not_S_hash[j].append(hash
                        ↪  (self.aux_bar_not_S[j][i]))
                else:
                    self.aux_bar_not_S_hash[j] =
                    ↪  self.aux_bar_not_S[j]

                self.T_aux_bar.append(MerkleTree
                ↪  (self.aux_bar_not_S_hash[j]))
                mt = self.T_aux_bar[j]
                self.root_aux_bar[j] = mt.get_merkle_root()

        h_bar = hash(''.join(self.root_aux_bar))
        if h_bar == h:
            b_prime = 1
            print(Bcolors.OKCYAN+"\nb_prime = 1"+Bcolors.ENDC)
        else:
            b_prime = 0
            print(Bcolors.WARNING+"\nb_prime = 0"+Bcolors.ENDC)

        if b == 1:
            print(Bcolors.OKCYAN+"\nb = 1"+Bcolors.ENDC)
        else:
            print(Bcolors.WARNING+"\nb = 0"+Bcolors.ENDC)

        return b and b_prime

    #Function the Verifier uses to reconstruct the challenge (is the
    ↪ same as Signer.Challenge())
    def Challenge_verifier(self,mex,h,root_c):

        self.S = []      # list of s_num selected rounds (from 0 to
        ↪ M-1)
        self.z = []      # list of s_num selected aux (from 0 to q-1)

        set = Set(Integers(M))
        set_random_seed(Integer(hash(str(mex)+str(h)+
        ↪ str(root_c)),16))

        for i in range(0,self.s_num):
            self.S.append(int(set.random_element()))
            set = set.difference([self.S[i]])
            self.z.append(Fq.random_element())

        return None
```

```python

    #Main execution of the Verifier
    def main(self,Htr_unsys,s,sigma,mex):

        h, root_c, rsp_S, path_aux_S, path_c_S, path_seed = sigma

        self.Challenge_verifier(mex,h,root_c)

        return
         ↪ self.Verification(Htr_unsys,s,h,root_c,rsp_S,path_aux_S,
         ↪ path_c_S,path_seed)

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

def MPC_id(e,Htr_unsys,s,mex,s_num):

    signer = Signer(True,s_num)       #True for abilitating hashing
     ↪ of input leaves
    verifier = Verifier(True,s_num)   #True for abilitating hashing
     ↪ of input leaves

    start_time = perf_counter()

    #SIGNATURE
    global sigma ################################
    sigma = Signer.main(signer,Htr_unsys,e,mex)

    #VERIFICATION
    ok = Verifier.main(verifier,Htr_unsys,s,sigma,mex)

    stop_time = perf_counter()

    #Get signature byte size
    sig_size = sys.getsizeof(sigma[0]) + sys.getsizeof(sigma[1])
    ↪   #First two element of the signature are digests
    #The signature byte size must not include the bytes used for the
    ↪   Python list or dictionary structure
    for j in range(0,len(sigma[2])):
        sig_size += sys.getsizeof(sigma[2][j])
    for i in [3,4]:
        for j in range(0,len(sigma[i])):
            sig_size += sys.getsizeof(sigma[i][j])
    for j in range(0,len(sigma[5])):
        sig_size += sys.getsizeof(sigma[5][j])

    return ok, sig_size, stop_time-start_time
```

7.3 merkletools.sage

```
import hashlib
import binascii
import sys

if sys.version_info < (3, 6):
    try:
        import sha3
    except:
        from warnings import warn
        warn("sha3 is not working!")

class MerkleTools(object):
    def __init__(self, hash_type="sha256"):
        hash_type = hash_type.lower()
        if hash_type in ['sha256', 'md5', 'sha224', 'sha384',
         ↪  'sha512',
                          'sha3_256', 'sha3_224', 'sha3_384',
                           ↪  'sha3_512']:
            self.hash_function = getattr(hashlib, hash_type)
        else:
            raise Exception('`hash_type` {} nor
             ↪  supported'.format(hash_type))

        self.reset_tree()

    def _to_hex(self, x):
        try:  # python3
            return x.hex()
        except:  # python2
            return binascii.hexlify(x)

    def reset_tree(self):
        self.leaves = list()
        self.levels = None
        self.is_ready = False

    def add_leaf(self, values, do_hash=False):
        self.is_ready = False
        # check if single leaf
        if not isinstance(values, tuple) and not isinstance(values,
         ↪  list):
            values = [values]
        for v in values:
            if do_hash:
                v = v.encode('utf-8')
                v = self.hash_function(v).hexdigest()
            v = bytearray.fromhex(v)
            self.leaves.append(v)

    def get_leaf(self, index):
        return self._to_hex(self.leaves[index])

    def get_leaf_count(self):
        return len(self.leaves)

    def get_tree_ready_state(self):
        return self.is_ready

    def _calculate_next_level(self):
        solo_leave = None
        N = len(self.levels[0])  # number of leaves on the level
        if N % 2 == 1:  # if odd number of leaves on the level
            solo_leave = self.levels[0][-1]
            N -= 1

        new_level = []
        for l, r in zip(self.levels[0][0:N:2],
         ↪  self.levels[0][1:N:2]):
            new_level.append(self.hash_function(l+r).digest())
        if solo_leave is not None:
            new_level.append(solo_leave)
        self.levels = [new_level, ] + self.levels  # prepend new
         ↪  level

    def make_tree(self):
        self.is_ready = False
        if self.get_leaf_count() > 0:
            self.levels = [self.leaves, ]
            while len(self.levels[0]) > 1:
                self._calculate_next_level()
        self.is_ready = True

    def get_merkle_root(self):
        if self.is_ready:
            if self.levels is not None:
                return self._to_hex(self.levels[0][0])
            else:
                return None
        else:
            return None

    def get_proof(self, index):
        if self.levels is None:
            return None
        elif not self.is_ready or index > len(self.leaves)-1 or index
         ↪  < 0:
            return None
        else:
            proof = []
            for x in range(len(self.levels) - 1, 0, -1):
                level_len = len(self.levels[x])
                if (index == level_len - 1) and (level_len % 2 == 1):
                 ↪  # skip if this is an odd end node
                    index = int(index / 2.)
                    continue
                is_right_node = index % 2
                sibling_index = index - 1 if is_right_node else index
                 ↪  + 1
                sibling_pos = "left" if is_right_node else "right"
                sibling_value =
                 ↪  self._to_hex(self.levels[x][sibling_index])
                proof.append({sibling_pos: sibling_value})
                index = int(index / 2.)
```

```python
            return proof

    def validate_proof(self, proof, target_hash, merkle_root):
        merkle_root = bytearray.fromhex(merkle_root)
        target_hash = bytearray.fromhex(target_hash)
        if len(proof) == 0:
            return target_hash == merkle_root
        else:
            proof_hash = target_hash
            for p in proof:
                try:
                    # the sibling is a left node
                    sibling = bytearray.fromhex(p['left'])
                    proof_hash = self.hash_function(sibling +
                    ↪ proof_hash).digest()
                except:
                    # the sibling is a right node
                    sibling = bytearray.fromhex(p['right'])
                    proof_hash = self.hash_function(proof_hash +
                    ↪ sibling).digest()
            return proof_hash == merkle_root

    #I ADDED THIS: it's the same as validate_proof, but doesn't do
    ↪ the final check and only outputs the merkle root
    def get_root_from_path(self, proof, target_hash):
        target_hash = bytearray.fromhex(target_hash)
        if len(proof) == 0:
            return target_hash
        else:
            proof_hash = target_hash
            for p in proof:
                try:
                    # the sibling is a left node
                    sibling = bytearray.fromhex(p['left'])
                    proof_hash = self.hash_function(sibling + proof_hash).digest()
                except:
                    # the sibling is a right node
                    sibling = bytearray.fromhex(p['right'])
                    proof_hash = self.hash_function(proof_hash + sibling).digest()
            return self._to_hex(proof_hash)
```

References

[1] C. Aguilar-Melchor et al. “The Return of the SDitH”. In: *Cryptology ePrint Archive* (2022).

[2] N. Aragon et al. “Durandal: a rank metric based signature scheme”. In: *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part III 38*. Springer. 2019, pp. 728–758.

[3] M. Baldi et al. “A new path to code-based signatures via identification schemes with restricted errors”. In: *arXiv preprint arXiv:2008.06403* (2020).

[4] M. Baldi et al. *Generic Decoding of Restricted Errors*. 2023. arXiv: `2303.08882 [cs.CR]`.

[5] M. Baldi et al. *Zero Knowledge Protocols and Signatures from the Restricted Syndrome Decoding Problem*. Cryptology ePrint Archive, Paper 2023/385. `https://eprint.iacr.org/2023/385`. 2023. URL: `https://eprint.iacr.org/2023/385`.

[6] A. Barenghi et al. “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*. Springer. 2021, pp. 23–43.

[7] A. M. Barg. “Some New NP-Complete Coding Problems”. In: *Probl. Peredachi Inf.* (1994).

[8] A. Becker et al. “Decoding Random Binary Linear Codes in 2n/20: How $1+1=0$ Improves Information Set Decoding”. In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by D. Pointcheval and T. Johansson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 520–536. ISBN: 978-3-642-29011-4.

[9] E. Berlekamp, R. McEliece, and H. van Tilborg. “On the inherent intractability of certain coding problems (Corresp.)” In: *IEEE Transactions on Information Theory* 24.3 (1978), pp. 384–386. DOI: `10.1109/TIT.1978.1055873`.

[10] D. J. Bernstein et al. “The SPHINCS+ signature framework”. In: *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security.* 2019, pp. 2129–2146.

[11] W. Beullens. “Sigma protocols for MQ, PKP and SIS, and fishy signature schemes”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer. 2020, pp. 183–211.

[12] L. Bidoux and P. Gaborit. “Shorter Signatures from Proofs of Knowledge for the SD, MQ, PKP and RSD Problems”. In: *arXiv preprint arXiv:2204.02915* (2022).

[13] P.-L. Cayrel, P. Véron, and S. M. El Yousfi Alaoui. “A Zero-Knowledge Identification Scheme Based on the q-ary Syndrome Decoding Problem”. In: *Selected Areas in Cryptography.* Ed. by A. Biryukov, G. Gong, and D. R. Stinson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–186. ISBN: 978-3-642-19574-7.

[14] N. T. Courtois, M. Finiasz, and N. Sendrier. “How to Achieve a McEliece-Based Digital Signature Scheme”. In: *Advances in Cryptology — ASIACRYPT 2001.* Ed. by C. Boyd. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 157–174. ISBN: 978-3-540-45682-7.

[15] T. Debris-Alazard, N. Sendrier, and J.-P. Tillich. “Wave: A new code-based signature scheme”. In: (2018).

[16] T. Feneuil. “Building MPCitH-based Signatures from MQ, MinRank, Rank SD and PKP”. In: *Cryptology ePrint Archive* (2022).

[17] T. Feneuil, A. Joux, and M. Rivain. “Shared permutation for syndrome decoding: New zero-knowledge protocol and code-based signature”. In: *Designs, Codes and Cryptography* (2022), pp. 1–46.

[18] T. Feneuil, A. Joux, and M. Rivain. “Syndrome decoding in the head: Shorter signatures from zero-knowledge proofs”. In: *Cryptology ePrint Archive* (2022).

[19] M. Finiasz and N. Sendrier. “Security Bounds for the Design of Code-Based Cryptosystems”. In: *Advances in Cryptology – ASIACRYPT 2009.* Ed. by M. Matsui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 88–105. ISBN: 978-3-642-10366-7.

[20] S. Gueron, E. Persichetti, and P. Santini. “Designing a Practical Code-Based Signature Scheme from Zero-Knowledge Proofs with Trusted Setup”. In: *Cryptography* 6.1 (2022). ISSN: 2410-387X. DOI: `10.3390/cryptography6010005`. URL: `https://www.mdpi.com/2410-387X/6/1/5`.

[21] S. Gueron, E. Persichetti, and P. Santini. “Designing a Practical Code-Based Signature Scheme from Zero-Knowledge Proofs with Trusted Setup”. In: *Cryptography* 6.1 (2022), p. 5.

[22] N. Howgrave-Graham and A. Joux. “New Generic Algorithms for Hard Knapsacks”. In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by H. Gilbert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 235–256. ISBN: 978-3-642-13190-5.

[23] D. Kales and G. Zaverucha. “An attack on some signature schemes constructed from five-pass identification schemes”. In: *Cryptology and Network Security: 19th International Conference, CANS 2020, Vienna, Austria, December 14–16, 2020, Proceedings*. Springer. 2020, pp. 3–22.

[24] P. J. Lee and E. F. Brickell. “An Observation on the Security of McEliece’s Public-Key Cryptosystem”. In: *Advances in Cryptology — EUROCRYPT ’88*. Ed. by D. Barstow et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 275–280. ISBN: 978-3-540-45961-3.

[25] Y. X. Li, R. Deng, and X. M. Wang. “On the equivalence of McEliece’s and Niederreiter’s public-key cryptosystems”. In: *IEEE Transactions on Information Theory* 40.1 (1994), pp. 271–273. DOI: `10.1109/18.272496`.

[26] A. May, A. Meurer, and E. Thomae. “Decoding random linear codes in $\mathcal{O}(2^{0.054n})$”. In: *Advances in Cryptology–ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings 17*. Springer. 2011, pp. 107–124.

[27] R. J. McEliece. “A public-key cryptosystem based on algebraic”. In: *Coding Thv* 4244 (1978), pp. 114–116.

[28] H. Niederreiter. “Knapsack-type cryptosystems and algebraic coding theory”. In: *Prob. Contr. Inform. Theory* 15.2 (1986), pp. 157–166.

[29] E. Prange. “The use of information sets in decoding cyclic codes”. In: *IRE Transactions on Information Theory* 8.5 (1962), pp. 5–9. DOI: `10.1109/TIT.1962.1057777`.

[30] J. Stern. “A method for finding codewords of small weight”. In: *Coding Theory and Applications*. Ed. by G. Cohen and J. Wolfmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 106–113. ISBN: 978-3-540-46726-7.

[31] J.-P. Thiers and J. Freudenberger. “Generalized Concatenated Codes over Gaussian and Eisenstein Integers for Code-Based Cryptography”. In: *Cryptography* 5.4 (2021), p. 33.

Acknowledgements

At the end of this thesis, I would like to dedicate some words to all the people this endeavor would not have been possible without.

I would like to express my deepest gratitude to Professor Marco Baldi and Dr. Paolo Santini for their guidance, for their patience and for all the help they provided throughout the realization of the thesis.

Special thanks also to Dr. Violetta Weger and Mr. Sebastian Bitzer, who welcomed me in Munich and made this abroad experience the best I could possibly have.

Ringrazio i miei nonni Rosalba e Mario e tutti i miei parenti per la vicinanza e l'affetto dimostratimi durante tutto il mio percorso.

Ringrazio i miei amici, in particolare Francesco, Lorenzo, Marco, Nicolò e Renato, che mi hanno permesso di trascorrere il periodo universitario con gioia e leggerezza tra serate, meme, giochi da tavolo e battute di dubbio gusto.

Infine, un ringraziamento speciale a mia madre Arianna, mio padre Fabio e mia sorella Alessandra. Vi ringrazio per il sostegno e per tutte le opportunità che mi avete sempre garantito con tanti sacrifici. Questa tesi è dedicata a voi.