LESS:
Linear Equivalence Signature Scheme

<

https://www.less-project.com/

Marco Baldi, Università Politecnica delle Marche
Alessandro Barenghi, Politecnico di Milano
Luke Beckwith, George Mason University
Jean-François Biasse, University of South Florida
Andre Esser, Technology Innovation Institute
Kris Gaj, George Mason University
Kamyar Mohajerani, George Mason University
Gerardo Pelosi, Politecnico di Milano
Edoardo Persichetti, Florida Atlantic University and Università “Sapienza”
Markku-Juhani O. Saarinen, PQShield and Tampere University
Paolo Santini, Università Politecnica delle Marche
Robert Wallace, George Mason University

Submitters: The team listed above is the principal submitter. There are no auxiliary submitters.

Inventors/Developers: Same as the principal submitter. Relevant prior work is credited where appropriate.

Owners: Submitters.

Email Address (preferred): edoardo.persichetti@uniroma1.it

Postal Address and Telephone (if absolutely necessary):
Edoardo Persichetti, Department of Computer Science, Sapienza University, Viale Regina Elena 295, +39 (329) 694 4609.

Signature: ×. See also printed version of “Statement by Each Submitter”.

Version: 2.0

Contents

1 Tools **1**

1.1 Notation and Main Concepts . . . 1

1.2 Signatures from Code Equivalence . . . 2

2 Design Rationale **2**

3 Protocol Description (2.B.1) **3**

3.1 Building Blocks . . . 3

3.2 Auxiliary Functions . . . 7

3.3 LESS Operations . . . 13

4 Security and Known Attacks (2.B.4/2.B.5) **16**

4.1 Known Attacks . . . 16

5 Performance (2.B.2) **20**

5.1 Performance in Software . . . 21

5.2 Performance in Hardware . . . 22

6 Known Answer Tests (2.B.3) **24**

7 Advantages and Limitations (2.B.6) **24**

A Mathematical Background **29**

B Proofs **29**

B.1 Identification . . . 29

B.2 Fiat-Shamir . . . 31

C Known Attacks **32**

C.1 Attacks Reducing to Permutation Equivalence . . . 32

C.2 Attacks based on Low-weight Codeword Finding . . . 32

C.3 Quantum Hardness . . . 34

1 Tools

In this section we introduce the basic tools that we need to describe LESS. More detailed mathematical background will be introduced in the rest of the document.

1.1 Notation and Main Concepts

Notation	Semantics
a	a scalar
A	a set
$\mathbf{a}$	a vector
$\mathbf{a}_J$	vector formed by the entries of $\mathbf{a}$ indexed by J
$\mathbf{A}$	a matrix
$\mathbf{A}_J$	matrix formed by columns of $\mathbf{A}$ indexed by J
$\mathbf{I}_n$	the $n \times n$ identity matrix
$\mathbb{Z}_n$	the ring of integers modulo n
$\mathbb{F}_q$	the finite field of order q
$\mathbb{F}_q^*$	the multiplicative group of $\mathbb{F}_q$
$\mathbb{F}_q^{k\times n}$	the set of matrices of size $k \times n$ over $\mathbb{F}_q$
$\mathbb{S}_{\mathsf{RREF}}$	set of all the $\mathbb{F}_q^{k\times n}$ matrices in RREF
$\mathbb{S}_{t,\omega}$	set of strings of length t and weight ω over $\mathbb{Z}_s$
$a \overset{\$}{\leftarrow} A$	sampling a uniformly at random from A

Table 1: Notation used in this document.

Linear Codes. An $[n, k]$-*linear code* $\mathfrak{C}$ over $\mathbb{F}_q$ is a k-dimensional vector subspace of $\mathbb{F}_q^n$, defined as the row space of a full-rank *generator matrix* $\mathbf{G} \in \mathbb{F}_q^{k\times n}$.

Standard Forms. There exists a standard form of generator matrices, called *systematic form*, corresponding to $\mathbf{G} = (\mathbf{I}_k \mid \mathbf{M})$, which can be obtained by simply calculating the row-reduced echelon form starting from any other generator matrix. In general, it is possible that doing so returns a matrix that does not have full rank. If so, there are simple procedures to obtain a matrix in systematic form by reducing with respect to a different minor (see e.g. [ABC$^+$]); however, for our purpose this is not necessary (and in fact, would produce a mismatch in the checks, since reducing according to a different minor is effectively the same as permuting some columns). Instead, we simply computing the first available Row-Reduced Echelon Form (RREF); this is enough to obtain a unique representation and enable verification. We denote this procedure by RREF.

Isometries. These are maps that preserve the Hamming weight. We focus on the case of linear isometries, that is, maps that consist of a permutation, together with non-zero scaling factors from $\mathbb{F}_q^*$. In LESS, we represent such isometries via generalized permutation or *monomial* matrices, i.e., $n \times n$ matrices $\mathbf{Q} \in \mathsf{M}_n$ having exactly one non-zero element in each row and column.

1.2 Signatures from Code Equivalence

LESS is built on top of a one-round Sigma protocol, which we present in Figure 1. A visual representation of the proof of knowledge is given in Figure 2.

Public Data	System parameters $q, n, k \in \mathbb{N}$, $\mathbf{G}_0 \in \mathbb{F}_q^{k \times n}$ and hash function $\textsc{Hash}$.
Private Key	$\mathbf{Q} \in \mathsf{M}_n$.
Public Key	$\mathbf{G}_1 = \mathrm{RREF}(\mathbf{G}_0\mathbf{Q})$.

PROVER		VERIFIER
$\widetilde{\mathbf{Q}} \overset{\$}{\leftarrow} \mathsf{M}_n$, $\widetilde{\mathbf{G}} \leftarrow \mathrm{RREF}(\mathbf{G}_0\widetilde{\mathbf{Q}})$ $\mathsf{cmt} \leftarrow \textsc{Hash}(\widetilde{\mathbf{G}})$	$\xrightarrow{\mathsf{cmt}}$	
	$\xleftarrow{\mathsf{ch}}$	$\mathsf{ch} \overset{\$}{\leftarrow} \{0,1\}$
$\mathsf{rsp} \leftarrow (1-\mathsf{ch})\widetilde{\mathbf{Q}} + \mathsf{ch} \cdot \mathbf{Q}^{-1}\widetilde{\mathbf{Q}}$	$\xrightarrow{\mathsf{rsp}}$	Accept if $\textsc{Hash}(\mathrm{RREF}(\mathbf{G}_{\mathsf{ch}} \cdot \mathsf{rsp})) = \mathsf{cmt}$

Figure 1: The Sigma protocol.

$$\begin{tikzcd}
\mathbf{G}_0 \arrow[r, "\widetilde{\mathbf{Q}}"] \arrow[rr, dashed, bend right, "\mathbf{Q}"'] & \widetilde{\mathbf{G}} & \mathbf{G}_1 \arrow[l, "\mathbf{Q}^{-1}\widetilde{\mathbf{Q}}"']
\end{tikzcd}$$

Figure 2: Representation of the proof of knowledge.

The Sigma protocol in Figure 1 is based on the Linear Equivalence Problem (LEP), which is introduced and discussed, along with security considerations, in Section 4.

2 Design Rationale

LESS in a Nutshell. LESS stands for Linear Equivalence Signature Scheme. It is constructed by applying the Fiat-Shamir transformation [FS86] to a zero-knowledge identification scheme. Such a scheme is obtained, essentially, by iterating the one-round Sigma protocol reported in Figure 1. The protocol description given in Section 3 includes several subsequent modifications. We will describe these in Section 3.1.

What is the Security of LESS? The Fiat-Shamir transformation guarantees Existential Unforgeability against Chosen Message Attacks (EUF-CMA), provided the underlying zero-knowledge scheme is secure. The transformation operates in the Random Oracle Model (ROM), and plausibly provides security in the Quantum Random Oracle Model (QROM) as well, as argued for instance in [DFMS19, LZ19]. For the case of LESS, security of the underlying zero-knowledge scheme relies on the hardness of the Linear Equivalence Problem (LEP).

What is the Linear Equivalence Problem? LEP is a particular flavor of the *code equivalence problem* in the Hamming metric. This is a traditional problem from coding theory, which asks to determine whether two linear codes are *equivalent*. By this, we mean that there exist an isometry connecting the two codes. We call Linear Equivalence Problem (LEP) the case where such an isometry is a monomial transformation, as defined in Section 1; this allows to distinguish from special cases such as, for example, when isometries consist of simpler objecs like permutations (in which case we talk about *permutation equivalence*) or include additional factors like field automorphisms (*semilinear equivalence*). In the case of LESS, we consider the computational version of the problem; in other words, given two (systematic) generator matrices $\mathbf{G}_0, \mathbf{G}_1 \in \mathbb{F}_q^{k\times n}$, find $\mathbf{Q} \in \mathsf{M}_n$ such that $\mathbf{G}_1 = \mathrm{RREF}(\mathbf{G}_0\mathbf{Q})$. Security of LEP is discussed in Section 4.

How is LESS designed? The structure of the LESS protocol features an inherent flexibility in the choice of its parameters, and lends itself naturally to various use cases. As reported in Section 5, we are able to optimize performance with different criteria in mind. A first option is to provide a balanced set of parameters, which yields similar sizes for public key and signature. Intuitively, such a set results in the smallest size for two different metrics: public key, and public key + signature. With this choice, LESS can be seen as a viable candidate for a general purpose signature scheme, with performance which recalls that of e.g. SPHINCS+ [BHH+15]. Alternatively, one could try to push the design by accepting tradeoffs, in order to obtain a smaller signature. In this case, the size of the signature gets closer to the bar set by current post-quantum schemes, and compares especially well with other code-based protocols.

3 Protocol Description (2.B.1)

The LESS signature parameters are as follows:

λ: target security level (positive integer)

q: finite field size (positive prime integer)

n: code length (positive integer)

k: code dimension (positive integer $< n$)

s: number of matrices $\mathbf{G}_i$ in the public key (positive integer)

t: number of protocol repetitions (positive integer)

ω: number of non-null challenges (positive integer)

More details on how to select parameters will be given in Sections 4 and 5 while discussing, respectively, security and performance aspects. For the remainder of this document, the parameters above are considered to be available to all algorithms.

3.1 Building Blocks

In this section we briefly describe the various stepping stones in the design of LESS, which transform the one-round Sigma protocol of Figure 1 into the full-fledged signature scheme. These steps are all gathered together in the procedures for key generation, signing, and verification which are presented in Section 3.3.

Iterating. The identification protocol in Figure 1 only provides *soundness* 1/2, meaning that a malicious party can successfully impersonate an honest prover half of the times. To achieve the authentication level required, then, it is necessary to iterate the protocol t times, where in the simplest case $t = \lambda$. By "iterate" here we intend repeating the Commit, Challenge and Response phases independently; the verifier will verify each response separately and only accept if verification is passed in all iterations.

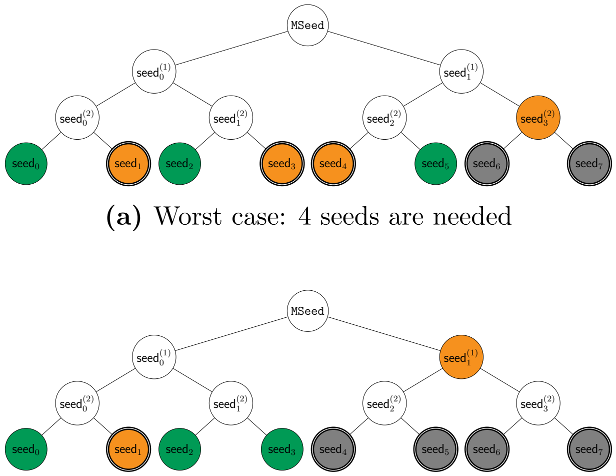
Fiat-Shamir. The Fiat-Shamir transformation [FS86] is applied to the iterated protocol in order to obtain a signature scheme. Informally, this transformation replaces the role of the verifier in the challenge step by producing the string of challenges via a collision-resistant hash function, which is computed on the message to be signed, together with the commitments for each round; in doing so, it turns the protocol from interactive to non-interactive. Note that, if the scheme is designed to be *commitment-recoverable* (as is the case for LESS), it is not necessary to transmit the commitments as part of the signature; this instead includes the hash digest, which can then be used to verify the signature once the commitments are, as the name says, recovered on the verifier side.

Multiple Public Keys. It is possible to greatly reduce the soundness error by expanding the public key [FG19]. In our case, this means including multiple generator matrices of the form $\mathbf{G}_i = \mathrm{RREF}(\mathbf{G}_0\mathbf{Q}_i)$, corresponding to as many secret matrices $\mathbf{Q}_i$, for $1 \le i \le s-1$. The verifier's challenge then asks to complete the diagram in Figure 2 starting from a specific key $\mathbf{G}_j$, to which the prover responds with the matching isometry $\mathbf{Q}_j^{-1}\widetilde{\mathbf{Q}}$. Since the challenge space is larger, fewer rounds are necessary to achieve the same authentication level, which allows to reduce the signature size (at the cost of an increase in the public key).

Compression of Random Elements. Several components of the LESS signature scheme are generated at random, and are obtained by expanding a randomly drawn seed via a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG). This includes the matrix $\mathbf{G}_0$ (which is included in the public key), all the elements of the private key $\mathsf{sk}[i] = \mathbf{Q}_i$ for $1 \le i \le s$ (as per optimization above), and all matrices $\widetilde{\mathbf{Q}}_i$, corresponding to the commitments in each of the rounds $0 \le i \le t-1$ in which the Sigma protocol gets executed. For all such components, the storage requirements can be minimized by storing the corresponding CSPRNG seeds, at the cost of performing the CSPRNG expansion at runtime. We note that the computational cost of expanding the elements is small, as the expansion procedure only involves pseudorandomly sampling the various monomial matrices, as well as the generator matrix $\mathbf{G}_0$. We note that, with respect to the latter, $\mathbf{G}_0$ can be sampled directly in row reduced echelon form by: i) sampling the positions of the k pivot-containing columns, ii) randomly filling the remaining $n-k$ columns, taking care to leave null-entries whenever at the left of a pivot. This is accomplished filling the cells of the $n-k$ columns row-wise, from right to left, skipping the columns containing the pivots in the process.

Fixed-weight Challenges. When the isometry queried is the one "on the left", i.e. the one between $\mathbf{G}_0$ and $\widetilde{\mathbf{G}}$, the response consists of the monomial matrix $\widetilde{\mathbf{Q}}$, which is generated uniformly at random. As per the paragraph above, it is then sufficient to transmit only the seed used to generate it. In other words, the challenge corresponding to 0 is much lighter than the others, and so the communication cost can be improved by adjusting the probability distribution of the challenge string, to make this possibility more likely to happen [BKP20]. This means that individual rounds require less communication on average (at the cost of increasing the number of rounds).

Seed Tree. To efficiently represent the t seeds used in the signing and verification algorithms of LESS, we use a binary "seed tree" [BKP20]. To begin, the root of the tree is set by a randomly chosen master seed $\mathsf{mseed} \in \{0,1\}^{\lambda}$. For every node, we generate its two children by feeding a CSPRNG with the node value and parse the CSPRNG output (with length 2λ) as its two children. This procedure is iterated for $\lceil \log_2(t) \rceil$ times, so that we end up with a layer having $2^{\lceil \log_2(t) \rceil} \geq t$ seeds $\mathsf{seed}_0, \cdots, \mathsf{seed}_{t-1}$. Note that every node in the tree is a binary string with length λ. When one needs to communicate all but a subset of the t seeds, say, for instance, all except those indexed by a set $J \subset \{0, \cdots, t-1\}$ of size ω, it is possible to exploit the tree structure to reduce the number of bits transmitted. The idea to improve efficiency is that of sending parent nodes, whenever possible: the verifier will repeat the procedure to generate the children nodes, and will thus obtain the required seeds, while minimizing the amount of space required in the signature. See Figure 3 for an example of this procedure[1].



(a) Worst case: 4 seeds are needed

(b) Lucky case: only 2 seeds are needed

Figure 3: Example of binary seed tree for $t = 8$ and $\omega = 3$. The chosen seeds (in green) are not revealed. The prover transmits only the orange nodes and the verifier can generate the remaining seeds (but not the chosen ones) by applying the CSPRNG. The nodes generated in this way are colored in gray. The leaves, in the base layer, which are obtained by the verifier are highlighted with the thick double line.

In the worst case, communicating the $t - \omega$ seeds requires the following amount of bits

$$\lambda \omega \log_2(t/\omega).$$

Monomial Inversions. A straightforward rendition of the signing algorithm would require to compute monomial matrix inversions at runtime during the signature. It is possible to avoid this computation by modifying the key generation procedure so that the values of $\mathbf{Q}_i^{-1}$ are obtained via the CSPRNG, and the corresponding seeds are stored as the private key. In this fashion, monomial matrix inversions take place at keypair generation time only.

[1] To protect from collision attacks on the commitments (e.g. [Cha22]), we actually build the tree starting from the root $\mathsf{MSeed}||\mathsf{Salt}$, where Salt is a binary string of length 2λ. The salt is used in every subsequent call to the CSPRNG (i.e. every time a seed gets expanded into two). To further protect against collision attacks, we feed the CSPRNG using also the indices which specify the location of the current leaf.

Performing Verification with Information Sets. Our aim is to reduce the size of the value to be communicated as rsp_i, for $0 \leq i \leq t-1$, whenever the corresponding challenge value is not null. In this case, in fact, rsp_i is a monomial matrix which cannot be compressed with a seed: the full representation of such a transformation would require $n\Big(\lceil \log_2(n) \rceil + \lceil \log_2(q-1) \rceil\Big)$ bits. To reduce this cost, we rely on the Information Set-LEP (IS-LEP) variant introduced in [PS23]. By modifying how commitments are generated, it becomes possible to verify the equivalence "on the right" by just sending $k\Big(\lceil \log_2(n) \rceil + \lceil \log_2(q-1) \rceil\Big)$ bits, instead. Given that we always have $k = n/2$, this technique allows to halve the communication cost for the most expensive executions. We now briefly summarize the details of this variant.

Instead of transmitting the entire monomial matrix, the prover is communicating only a truncated representation of how $\mathbf{Q}^{-1}\widetilde{\mathbf{Q}}$ acts; namely, its action on an information set. In principle, this would prevent verification of the commitment, since the verifier would compute a code which is different from the one generated by the prover. Namely, while the prover used $\widetilde{\mathfrak{C}}$ (i.e., the code generated by $\widetilde{\mathbf{G}}$), the verifier has a code $\widetilde{\mathfrak{C}}'$ which is identical to $\widetilde{\mathfrak{C}}'$ only on an information set. For the coordinates outside of the information set, codewords are modified by an unknown monomial transformation.

To allow for reconciliation, we need the following modification in the commitment generation and verification procedures. Let $\widetilde{\mathbf{G}}$ be the generator matrix obtained by the prover, and $\widetilde{\mathbf{G}}'$ that computed by the verifier. Since the two codes are equal on an information set J (which is known also to the verifier, since it is communicated along with the response), this means that $\widetilde{\mathbf{G}}'_J = \mathbf{S}\widetilde{\mathbf{G}}_J$ for some non-singular $\mathbf{S}$. For the coordinates which are not indexed by the information set, the two matrices satisfy the following relation:

$$\widetilde{\mathbf{G}}'_{\{1,\cdots,n\}\setminus J} = \mathbf{S}\widetilde{\mathbf{G}}_{\{1,\cdots,n\}\setminus J}\mathbf{Z},$$

where $\mathbf{Z} \in \mathsf{M}_{n-k}$. After computing the RREF with respect to J, the non-systematic parts of the two matrices are

$$\begin{aligned}
&\text{Prover: RREF on } \widetilde{\mathbf{G}} \text{ with respect to } J && \mapsto && \mathbf{V} = \widetilde{\mathbf{G}}_J\widetilde{\mathbf{G}}_{\{1,\cdots,n\}\setminus J},\\
&\text{Verifier: RREF on } \widetilde{\mathbf{G}}' \text{ with respect to } J && \mapsto && \mathbf{V}' = \widetilde{\mathbf{G}}'^{-1}_J\widetilde{\mathbf{G}}'_{\{1,\cdots,n\}\setminus J} = \widetilde{\mathbf{G}}_J^{-1}\widetilde{\mathbf{G}}_{\{1,\cdots,n\}\setminus J}\mathbf{Z}.
\end{aligned}$$

Since $\mathbf{V}$ and $\mathbf{V}'$ are equal up to a monomial transformation, it is enough that both the prover and the verifier compute the first lexicographic matrix that one can obtain, considering all possible monomial transformations. This matrix is obtained by first scaling each column so that its elements are in lexicographic ordering, and then by sorting the obtained columns. The resulting matrix is used as a representative of the orbits generated by $\mathbf{V}$ and $\mathbf{V}'$, under the action of monomials: since the two matrices are in the same orbit, the lexicographically ordered matrices will be the same. Computing this matrix takes a much shorter time than a single RREF, so that in practice the impact on computational complexity is not noticeable.

In [PS23], it is proven that this formulation, which we call IS-LEP, is completely equivalent to LEP. By this, we mean that two codes are a "YES" (resp. "NO") instance for IS-LEP if and only if they are a "YES" (resp. "NO") instance for LEP.

3.2 Auxiliary Functions

In our constructions, we require several accessory functions, which we describe here. To begin with, we clarify our choice of cryptographic primitives, to obtain a practical realization of the Hash and CSPRNG functions. We employ the SHAKE family of functions with an appropriate security level (i.e. SHAKE-128 for category 1 and SHAKE-256 for other categories) as a CSPRNG, while we employ SHA-3 as our implementation of the function Hash, selecting a digest of size 2λ.

Algorithm 1: PrepareDigestInput($\mathbf{G}, \mathbf{Q}$)

```
Input: $\mathbf{G}$: a generator matrix
       $\mathbf{Q}$: a monomial matrix
Output: $\mathbf{V}$: non-IS portion of the result of RREF($\mathbf{GQ}$)
        $\overline{\mathbf{Q}}$: Monomial matrix acting as $\mathbf{Q}$ does on the IS of $\mathbf{GQ}$, and packs the IS
          of $\mathbf{G}\overline{\mathbf{Q}}$ on the leftmost $k$ columns
Data: LexMin($\mathbf{a}$): function computing the lexicographic minimum in
      $\{z\mathbf{a} \mid z \in \mathbb{F}_q^*\}$ for $\mathbf{a} \in \mathbb{F}_q^k$
      LexSortColumns($\mathbf{V}$): function sorting the columns of a $k \times (n-k)$
      matrix $\mathbf{V}$ in lexicographic increasing order from left to right.
      RREF-P($\mathbf{G}$): function computing the RREF of the input and a $n$ elements
      Boolean vector isPiv, where the $i$-th cell is true if the $i$-th column of
      RREF($\mathbf{G}$) contains a pivot

PivIdx ← 0
NonPivIdx ← $k$
$\mathbf{V}$ ← [ ]  // matrix with no columns
$\overline{\mathbf{Q}}$ ← $[0_n, 0_n, \dots, 0_{n-1}]$  // $0_n$: null column vector w/ $n$ rows
($\mathbf{G}^\dagger$, isPiv) ← RREF-P($\mathbf{GQ}$)  // $\mathbf{G}^\dagger = [g_0^\dagger, g_1^\dagger, \dots, g_{n-1}^\dagger]$, columns
for col ← 0 to $n-1$ do
    {val, rowIdx} ← Extract($q_{\mathsf{col}}$)  // pick the non-zero value and its position
    if isPiv[col] = false then
        $\mathbf{V}$ ← [$\mathbf{V}$, LexMin($g_{\mathsf{col}}^\dagger$)]
        $\overline{q}_{\mathsf{rowIdx},\mathsf{NonPivIdx}}$ ← val
        NonPivIdx ← NonPivIdx + 1
    else
        $\overline{q}_{\mathsf{rowIdx},\mathsf{PivIdx}}$ ← val
        PivIdx ← PivIdx + 1
// $\overline{\mathbf{Q}}$ is such that $\mathbf{G}\overline{\mathbf{Q}}$ admits systematic form (i.e., RREF($\mathbf{G}\overline{\mathbf{Q}}$) = [$\mathbf{I}$ $\mathbf{V}$]) and the scaling factors
   on the leftmost IS of $\mathbf{G}\overline{\mathbf{Q}}$ are the same as the ones of the leftmost IS of $\mathbf{GQ}$
// In other words, $\overline{\mathbf{Q}}$ acts on $\mathbf{G}$ as the application of $\mathbf{Q}$, followed by a stable sorting of the
   columns, where the IS columns precede the non-IS ones
$\mathbf{V}$ ← LexSortColumns($\mathbf{V}$)
return $\mathbf{V}, \overline{\mathbf{Q}}$
```

```
Algorithm 2: CSPRNG($\mathsf{seed}$, $\mathsf{M}_n$)
  Input: $\mathsf{seed} \in \{0,1\}^\lambda$
  Output: Q : a monomial matrix represented by two lists of length $n$, called $Q.\pi$ and $Q.u$,
          containing the permutation and coefficients.
  Data: XOF_INIT($\mathsf{seed}$): function which initializes the XOF state using the provided seed.
        XOF_SQUEEZEBITS($b$): function which returns the next $b$ bits from the XOF digest.
        $\mathbf{mask_q} = (2^{\lceil \log_2(q) \rceil} - 1)$: bitmask for coefficient samples.
        $\mathbf{mask_n} = (2^{\lceil \log_2(n) \rceil} - 1)$: bitmask for permutation samples.
  $XOF\_init(\mathsf{seed})$
  $bits \leftarrow XOF\_SqueezeBits(64)$
  $Q.\pi \leftarrow [0, 1, .., n-1]$
  $c \leftarrow 0$
  for $i \in [0, n-1]$ do
      do
          if $c == \lfloor 64/\lceil \log_2(q) \rceil \rfloor - 1$ then
              $bits \leftarrow XOF\_SqueezeBits(64)$
              $c \leftarrow 0$
          $x \leftarrow bits \ \& \ mask_q$
          $bits \leftarrow bits >> \lceil \log_2(q) \rceil$
          $c \leftarrow c + 1$
      while $x \geq q - 1$
      $Q.u[i] \leftarrow x + 1$
  $bits \leftarrow XOF\_SqueezeBits(64)$
  $c \leftarrow 0$
  for $i \in [0, n-1]$ do
      do
          if $c == \lfloor 64/\lceil \log_2(n) \rceil \rfloor - 1$ then
              $bits \leftarrow XOF\_SqueezeBits(64)$
              $c \leftarrow 0$
          $x \leftarrow bits \ \& \ mask_n$
          $bits \leftarrow bits >> \lceil \log_2(n) \rceil$
          $c \leftarrow c + 1$
      while $x \geq n$
      $t \leftarrow Q.\pi[i]$
      $Q.\pi[i] \leftarrow Q.\pi[x]$
      $Q.\pi[x] \leftarrow t$
  return Q
```

The approach used for the expansion of a monomial matrix from a seed is described in Algorithm 2. This operation requires sampling within two different ranges: $[1, q-1]$ and $[0, n-1]$. For the former, we reject samples using the range $[0, q-2]$ and then increment by one to shift the samples into the correct range. The instance of SHAKE is initialized with the λ-bit seed. Then the digest of the XOF is parsed in 64-bit blocks using bit-aligned sampling within the blocks. This provides a beneficial trade-off between minimizing the number of wasted digest bits while also limiting the complexity of bit-shifting. When the bit-rate changes from sampling in the range $[1, q-1]$ to $[0, n-1]$, the bits of the current block are discarded.

Algorithm 3: CSPRNG(seed, $\mathbb{S}_{t,\omega}$)

```
Input: seed $\in \{0,1\}^{\lambda}$
Output: ch $\in \mathbb{S}_{t,\omega}$
Data: XOF_init(seed): function which initializes the XOF state using the provided seed.
      XOF_SqueezeBits($b$): function which returns the next $b$ bits from the XOF
      digest.
      $\mathbf{mask_s} = (2^{\lceil \log_2(s) \rceil} - 1)$: bitmask for challenge entries.
      $\mathbf{mask_t} = (2^{\lceil \log_2(t) \rceil} - 1)$: bitmask for shuffling the challenge.
$XOF\_init$(seed)
ch$[0 : t-1] \leftarrow [0, \dots, 0]$
if $s \neq 2$ then
    for $i \in [t-\omega, t-1]$ do
        do
            $val \leftarrow XOF\_SqueezeBits(\lceil \log_2(s)/8 \rceil * 8)$
            $val \leftarrow val \ \& \ mask_s$
        while $val \geq s-1$
        ch$[i] \leftarrow s+1$
else
    ch$[t-\omega, t-1] \leftarrow [1, \dots, 1]$
for $p \in [t-\omega, t-1]$ do
    do
        $pos \leftarrow XOF\_SqueezeBits(\lceil \log_2(t)/8 \rceil * 8)$
        $pos \leftarrow pos \ \& \ mask_t$
    while $pos > p$
    $x \leftarrow$ ch$[p]$
    ch$[p] \leftarrow$ ch$[pos]$
    ch$[pos] \leftarrow x$
return G
```

The approach used for the generation of the fixed-weight challenge is described in Algorithm 3. Similarly to Algorithm 2, two different bit rates are sampled after the initialization of the SHAKE instance. The first set of samples is taken within the range $[1, s-1]$. This is achieved by using rejection sampling in the range $[0, s-2]$ and then incrementing by one. Note that, when s is equal to two, the only possible value is one. Therefore, this stage can be entirely skipped. These samples represent the values of the non-zero challenges. They are stored in the top ω positions of the challenge array. They are then distributed randomly throughout the challenge by randomly filling the array. Unlike monomial sampling, this operation is performed only once during sign and verify. Since the latency of this operation is very low compared to the rest of the algorithm, a simple definition is preferred. Thus, all samples are generated using byte-aligned chunks of the XOF digest.

```
Algorithm 4: CSPRNG(seed, $\mathbb{S}_{\mathsf{RREF}}$)
Input: seed $\in \{0,1\}^{\lambda}$
Output: $\mathbf{G} \in \mathbb{Z}_q^{k \times n}$ in RREF
Data: XOF_INIT(seed): function which initializes the XOF state using the
        provided seed.
      XOF_SQUEEZEBITS($b$): function which returns the next $b$ bits from the
        XOF digest.
      $\mathbf{mask_q} = (2^{\lceil \log_2(q) \rceil} - 1)$: bitmask for coefficient samples.
$\mathbf{G}[0 : k-1, 0 : k-1] \leftarrow \mathbf{I}_k$
$XOF\_init$(seed)
$c \leftarrow 0$
$bits \leftarrow XOF\_SqueezeBits(64)$
for $i \in [0, k-1]$ do
    for $j \in [k, n-1]$ do
        do
            if $c == \lfloor 64/\lceil \log_2(q) \rceil \rfloor - 1$ then
                $bits \leftarrow XOF\_SqueezeBits(64)$
                $c \leftarrow 0$
            $x \leftarrow bits \;\&\; mask_q$
            $bits \leftarrow bits >> \lceil \log_2(q) \rceil$
            $c \leftarrow c + 1$
        while $x \geq q$
        $\mathbf{G}[i, j] \leftarrow x$
return $\mathbf{G}$
```

Algorithm 4 describes the process of expanding a seed into a generator matrix in RREF. This is accomplished in a straightforward manner by initializing the first k columns of $\mathbf{G}$ to $\mathbf{I}_k$, and then sampling random coefficients in the range $[0, q-1]$ row-wise. The samples are again generated using bit-wise parsing of the XOF digest within 64-bit blocks.

At the end of key generation, the generator matrices that comprise the public key are compressed in order to reduce their size. The procedure, illustrated in Algorithm 5, requires compressing the locations of the pivot columns, as well as all the coefficients of the non-pivot columns. This task is accomplished by first prefixing the length-n list of 1-bit flags, which specify if a column of $\mathbf{G}$ is a pivot; note that, for some parameter sets, the value of n is not divisible by 8, so we pad with zeros to make sure that the coefficient encoding begins on a new byte. The coefficients are then serialized row-by-row. The inverse of this operation, which expands the compressed description back into a generator matrix (in RREF) is described in Algorithm 6.

During signing, the non-zero responses are also compressed. The non-zero responses can be represented by two lists of length k representing a set of coefficients in the range $[1, q-1]$ and a set of permutation values in the range $[0, n-1]$. The permutation list is serialized first, and the coefficient list second.

Algorithm 5: CompressRREF($\mathbf{G}$)

```
Input: $\mathbf{G}$ : a generator matrix in RREF.
       $\mathbf{p} \in [0;1]^n$ where $\mathbf{p}[i]$ denotes if columnn $i$ of $\mathbf{G}$ is a pivot.
Output: $\mathbf{b} \in \{0,1\}^{(k*k*\lceil\log_2(q)\rceil+n}$
Data: $\mathrm{LSB}_x(\mathbf{a})$: function which returns the $x$ least significant bits of the input $\mathbf{a}$.

$\mathbf{b} \leftarrow LSB_1(\mathbf{p}[n-1]) \,||\, \ldots \,||\, LSB_1(\mathbf{p}[0])$
$\mathbf{b} \leftarrow 0^{8-(n\ mod\ 8)} || \mathbf{b}$
for $i \in [0, k-1]$ do
    for $j \in [0, n-1]$ do
        if $\mathbf{p}[i] \neq 1$ then
            $\mathbf{b} \leftarrow LSB_{\lceil\log_2(q)\rceil}(\mathbf{G}[i,j]) \,||\, \mathbf{b}$
return $\mathbf{b}$
```

Algorithm 6: ExpandToRREF($\mathbf{G}$)

```
Input: $\mathbf{b} \in \{0,1\}^{(n-k)*n*\lceil\log_2(q)\rceil+k*\lceil\log_2(n)\rceil}$
Output: $\mathbf{G}$ : a generator matrix in RREF.
Data: $\mathrm{LSB}_x(\mathbf{a})$: returns the $x$ least significant bits of the input $\mathbf{a}$.

$\mathbf{p} \leftarrow \{\}$
$\mathbf{p}[n-1] \,||\, \ldots \,||\, \mathbf{p}[0] \leftarrow LSB_n(\mathbf{b})$
$\mathbf{b} \leftarrow \mathbf{b} >> (\lceil n/8 \rceil * 8)$
for $i \in [0, k-1]$ do
    for $j \in [0, n-1]$ do
        if $\mathbf{p}[j] \neq 1$ then
            $\mathbf{G}[i,j] \leftarrow LSB_{\lceil\log_2(q)\rceil}(\mathbf{b})$
            $\mathbf{b} \leftarrow \mathbf{b} >> \lceil\log_2(q)\rceil$
for $i \in [0, k-1]$ do
    $pivot\_idx \leftarrow 0$
    for $j \in [0, n-1]$ do
        if $\mathbf{p}[j] == 1$ then
            if $i == pivot\_idx$ then
                $\mathbf{G}[i,j] \leftarrow 1$
            else
                $\mathbf{G}[i,j] \leftarrow 0$
            $pivot\_idx \leftarrow pivot\_idx + 1$
return $\mathbf{G}$
```

Seed Tree Implementation Strategy The LESS signing verification algorithms involve the hierarchical derivation of a sequence of seeds, one for each one of the t iterations of the underlying ZKID protocol. In order to optimize the representation of the seeds to be sent inside the signature, we employ a binary tree structure, as introduced in Section 3.1.

Algorithm 7: CompressMonomAction($\mathbf{Q}^*$))

```
Input: $\mathbf{Q}^*$ representing the relevant coefficient and permutation values of the
       monomial action
Output: $\mathbf{b} \in \{0,1\}^{k*\lceil \log_2(q) \rceil + k*\lceil \log_2(n) \rceil}$
b ← {}
for $i \in [0, k-1]$ do
    $\mathbf{b} \leftarrow LSB_{\lceil \log_2(n) \rceil}(Q^*.\pi[i]) \mid\mid \mathbf{b}$
for $i \in [0, k-1]$ do
    $\mathbf{b} \leftarrow LSB_{\lceil \log_2(q) \rceil}(Q^*.u[i]) \mid\mid \mathbf{b}$
return b
```

Algorithm 8: ExpandToMonomAction($\mathbf{u}, \pi$)

```
Input: $\mathbf{b} \in \{0,1\}^{k*\lceil \log_2(q) \rceil + k*\lceil \log_2(n) \rceil}$
Output: $\mathbf{Q}^*$ representing the relevant coefficient and permutation values of the
        monomial action
for $i \in [0, k-1]$ do
    $\pi[i] \leftarrow LSB_{\lceil \log_2(n) \rceil}(\mathbf{b})$
    $\mathbf{b} \leftarrow \mathbf{b} >> \lceil \log_2(n) \rceil$
for $i \in [0, k-1]$ do
    $u[i] \leftarrow LSB_{\lceil \log_2(q) \rceil}(\mathbf{b})$
    $\mathbf{b} \leftarrow \mathbf{b} >> \lceil \log_2(q) \rceil$
return $\mathbf{Q}^*$
```

The SeedTreeLeaves procedure computes a binary tree of nodes, each one of which contains a binary string obtained concatenating a random value, a random salt and an integer node index represented in natural binary. For each child node, the (first) random value contained in the node is obtained through a CSPRNG seeded with the entire binary string of its parent. The root node employs, as a first binary string a fresh random bitstring, drawn from the system TRNG.

The SeedTreePaths procedure determines, given a seed tree, and a subset of the leaves to be disclosed, represented as a bitset, and derives which tree paths of nodes should be disclosed so that it is possible for the verifier to rebuild all the leaves which have been marked in the bitset. It does so by determining the highest ancestors in the tree, for which all the descendants are nodes to be revealed, proceeding from the leaves to the root.

Finally, the RebuildSeedTreeLeaves procedure receives the output of the SeedTreePaths one, and the same subset of leaves which should be recoverable from the tree paths contained in it. The procedure starts by determining, on a stencil of the binary tree, which subtrees have been fully disclosed, by inserting their roots within the output of the SeedTreePaths procedure. It then proceeds to recompute, starting from these elements, and proceeding towards the leaves, all the leaves nodes which are indicated in the subset of leaves to be recovered.

3.3 LESS Operations

We now describe in detail the procedures for generating keys, signing, and verifying.

Algorithm 9: LESS-Keygen()

```
Input: None
Output: $\mathsf{sk} = (\mathsf{seed}_1, \ldots, \mathsf{seed}_{s-1})$: private key, where $\mathsf{seed}_i \in \{0,1\}^\lambda$ is employed to
        derive $\mathbf{Q}_i^{-1}$. The first entry of the private key $\mathbf{Q}_0 = \mathbf{I}_n$ is not stored.
        $\mathsf{pk} = (\mathsf{seed}_0, \mathbf{G}_1, \ldots, \mathbf{G}_{s-1})$: public key, where $\mathbf{G}_i \in \mathbb{F}_q^{k \times n}$ is stored as the
        non-pivot columns and their positions via the CompressRREF
        subroutine.
        $\mathsf{seed}_0$ is employed to derive $\mathbf{G}_0$ in RREF at runtime.
for $i \leftarrow 1$ to $s-1$ do
    $\mathsf{sk}[i] \xleftarrow{\$} \{0,1\}^\lambda$
    $\mathbf{Q} \leftarrow \mathrm{CSPRNG}(\mathsf{sk}[i], \mathsf{M}_n)$
    $\mathbf{Q}_i \leftarrow \mathbf{Q}^{-1}$
    $\mathbf{G}_i \leftarrow \mathrm{RREF}(\mathbf{G}_0\mathbf{Q}_i)$
    $\mathsf{pk}[i] \leftarrow$ CompressRREF$(\mathbf{G}_i)$
return $(\mathsf{sk}, \mathsf{pk})$
```

Algorithm 10: LESS-Sign(sk, msg, pk)

```
Input: $\mathsf{sk} = (\mathsf{seed}_1, \ldots, \mathsf{seed}_{s-1})$: private key, where $\mathsf{seed}_i \in \{0,1\}^\lambda$ is employed to
       derive $\mathbf{Q}_i^{-1}$. The first entry of the private key which is the identity matrix
       $\mathbf{Q}_0 = \mathbf{I}$ is not stored.
       $\mathsf{pk}[0] = \mathsf{seed}_0$: first element of the public key employed to derive $\mathbf{G}_0$ in
       RREF at runtime
       $\mathsf{msg}$: message to be signed, as a sequence of bits
Output: $\sigma = (\mathsf{rsp}_1, \ldots, \mathsf{rsp}_t, \mathbf{d})$: signature composed by a salt $\mathsf{salt}$, $\omega$ ZKID protocol
        responses $\mathsf{rsp}_i, 0 \leq i < t$, the seed-tree path $\mathsf{treepath}$ and a digest $\mathbf{d}$
$\mathbf{G}_0 \leftarrow \mathrm{CSPRNG}(\mathsf{pk}[0], \mathbb{S}_{\mathsf{RREF}})$
$\mathsf{rootSeed} \xleftarrow{\$} \{0,1\}^\lambda$
$\mathsf{salt} \xleftarrow{\$} \{0,1\}^\lambda$
$(\mathsf{seed}[0], \ldots, \mathsf{seed}[t-1]) \leftarrow \textsc{SeedTreeLeaves}(\mathsf{rootSeed}, \mathsf{salt})$
for $i \leftarrow 0$ to $t-1$ do
    $\widetilde{\mathbf{Q}}_i \leftarrow \mathrm{CSPRNG}(\mathsf{seed}[i], \mathsf{M}_n)$
    $(\mathbf{V}_i, \overline{\mathbf{Q}_i}) \leftarrow \textsc{PrepareDigestInput}(\mathbf{G}_0, \widetilde{\mathbf{Q}}_i)$
$\mathbf{d} \leftarrow \textsc{Hash}(\mathbf{V}_0 || \ldots || \mathbf{V}_{t-1} || \mathsf{msg} || \mathsf{salt})$
$(x_0, \ldots, x_{t-1}) \leftarrow \mathrm{CSPRNG}(\mathbf{d}, \mathbb{S}_{t,\omega})$
$\mathsf{treepath} \leftarrow \textsc{SeedTreePaths}(\mathsf{rootSeed}, (x_0, \ldots, x_{t-1}))$
$j \leftarrow 0$
for $i \leftarrow 0$ to $t-1$ do
    if $x_i \neq 0$ then
        $\mathbf{Q} \leftarrow \mathsf{sk}[x_i]$
        $\mathsf{rsp}_j \leftarrow \textsc{CompressMonomAction}(\mathbf{Q}\overline{\mathbf{Q}_i})$
        $j \leftarrow j+1$
return $(\mathsf{salt}, \mathsf{treepath}, \mathsf{rsp}_0, \ldots, \mathsf{rsp}_{\omega-1}, \mathbf{d})$
```

```
Algorithm 11: LESS-Verify($\mathsf{pk}, \sigma, \mathsf{msg}$)
  Input: $\mathsf{pk} = (\mathbf{G}_0, \ldots, \mathbf{G}_{s-1})$: public key, where $\mathbf{G}_i \in \mathbb{F}_q^{k \times n}$
         $\sigma = (\mathsf{salt}, \mathsf{treepath}, \mathsf{rsp}_0, \ldots, \mathsf{rsp}_{\omega-1}, \mathbf{d})$: signature composed by a salt $\mathsf{salt}$,
         $omega$ ZKID protocol responses $\mathsf{rsp}_i, 0 \le i < t$, the seed-tree path $\mathsf{treepath}$
         and a digest $\mathbf{d}$
         $\mathsf{msg}$: message to be signed, as a sequence of bits
  Output: Boolean value indicating whether the signature is valid
$\mathbf{G}_0 \leftarrow \mathrm{CSPRNG}(\mathsf{pk}[0], \mathbb{S}_{\mathsf{RREF}})$
$(x_0, \ldots, x_{t-1}) \leftarrow \mathrm{CSPRNG}(\mathbf{d}, \mathbb{S}_{t,\omega})$
$(\mathsf{seed}[0], \ldots, \mathsf{seed}[t-1]) \leftarrow \textsc{RebuildSeedTreeLeaves}(\mathsf{treepath}, (x_0, \ldots, x_{t-1}), \mathsf{salt})$
for $i \leftarrow 0$ to $t-1$ do
    if $x_i = 0$ then
        $\widetilde{\mathbf{Q}}_i \leftarrow \mathrm{CSPRNG}(\mathsf{seed}[i], \mathsf{M}_n)$
        $(\mathbf{V}_i, \overline{\mathbf{Q}_i}) \leftarrow \textsc{PrepareDigestInput}(\mathbf{G}_0, \widetilde{\mathbf{Q}}_i)$
    else
        $\overline{\mathbf{Q}}_i \leftarrow \textsc{ExpandToMonomAction}(\mathsf{rsp}_i)$
        $\mathbf{G}_i \leftarrow \mathsf{pk}[x_i]$
        $\overline{\mathbf{G}}_i \leftarrow \mathbf{G}_i \overline{\mathbf{Q}}_i$
        $[\mathbf{I}\ \mathbf{V}_i] \leftarrow \mathrm{RREF}(\overline{\mathbf{G}}_i)$   // $\mathbf{V}_i = [\mathbf{v}_0\ \mathbf{v}_1\ \cdots\ \mathbf{v}_{n-k-1}]$
        for $j \leftarrow 0$ to $(n-k)-1$ do
            $\mathbf{v}_j \leftarrow \textsc{LexMin}(\mathbf{v}_j)$
        $\mathbf{V}_i \leftarrow \textsc{LexSortColumns}(\mathbf{V}_i)$
$\mathbf{d}' \leftarrow \textsc{Hash}(\mathbf{V}_0 || \ldots || \mathbf{V}_{t-1} || \mathsf{msg} || \mathsf{salt})$
if $(\mathbf{d} = \mathbf{d}')$ then
    return true
return false
```

4 Security and Known Attacks (2.B.4/2.B.5)

The security of LESS formally relies on the following three items:

- The Sigma protocol of Figure 1 is secure.
- The Fiat-Shamir transformation.
- Protocol-level modifications.

The first item was addressed in [BMPS20], where the claim is proved in a very intuitive way; we report these arguments in Appendix B.1.

The second item is a very well-known technique, arguably one of the cornerstones of signature scheme design. Introduced in 1986 by Fiat and Shamir [FS86], it has been extensively studied since. We recall it briefly in Appendix B.2.

Finally, LESS includes some modifications which affect the scheme at a protocol level, but have no impact on security. These techniques are already incorporated in the description given in Section 3.3, and briefly summarized, one by one, in Section 3.1, where we refer to the appropriate literature for further details about their impact on security.

4.1 Known Attacks

Broadly speaking, attacks on LEP split into two classes of algorithms – those using a reduction to the Permutation Equivalence Problem (PEP) and those based on finding codewords of small Hamming weight. The former tend to only work under very specific circumstances, and are easy to bypass; we give here an intuition for the latter, while more details are provided in Appendix C.

Information-Set Decoding. An *Information-Set Decoding (ISD)* algorithm is an iterative procedure which aims at finding a certain low-weight word. This can be, for instance, an error vector to correct a noisy codeword, or directly a low-weight codeword in a certain linear code. In its simplest form, each iteration of the ISD algorithm guesses a set I of k indices, the information set. Provided that no errors occur in I, i.e., the searched word is all-zero with respect to I, one can then recover the target word by simple linear algebra.

Since its introduction by Prange [Pra62] in 1962, several variants have been presented (e.g. [LB88, Ste89, Dum91, BLP11, MMT11, BJMM12, MO15, BM17, BM18, Ess22]), utilizing clever observations such as exploiting collisions [BLP11] or representations [BJMM12] to provide speed-ups. The algorithms can easily be generalized to non-binary fields. However, as noted in [Meu13], the gain from more advanced algorithms of this class deteriorates quickly for increasing values of q. Overall, the complexity of ISD algorithms is exponential in nature, and, generally, increases with the weight of the target word.

Using ISD as Subroutine. When utilizing ISD to solve code equivalence (see Appendix C.2), an attacker is interested in finding a certain number ℓ of distinct codewords with Hamming weight w. Let $C_{\text{ISD}}(q, n, k, w)$ be the cost of each call, that is, the expected time to find a (random) codeword with the desired properties. Then, the cost to find ℓ *distinct* codewords with weight w, out of $N(w)$, is

$$f\big(\ell, N(w)\big) \cdot C_{\mathrm{ISD}}(w),$$

where $f\big(\ell, N(w)\big)$ counts the average number of calls of ISD, and is well approximated as

$$f\big(\ell, N(w)\big) \approx \begin{cases} \ell & \text{, if } \ell \ll N(w), \\ \ell \cdot \ln(\ell) & \text{, else.} \end{cases} \tag{1}$$

The best known algorithm to find those codewords for $q > 2$ is Peters' ISD [Pet10], which is a generalisation of Stern's ISD to the non-binary case. The complexity of this generalization is

$$C_{\mathrm{ISD}}(w) = \min_{p,u} \left\{ \frac{C_{\mathrm{ITER}}(w,p,u)}{P_{\mathrm{SUCC}}(w,p,u)} \right\},$$

where

$$\begin{aligned} C_{\mathrm{ITER}}(w,p,u) = u & \left(\left(\frac{k}{2} - p + 1 \right) + (q-1)^p \left(\binom{\lfloor \frac{k}{2} \rfloor}{p} + \binom{\lceil \frac{k}{2} \rceil}{p} \right) \right) \\ & + \frac{\frac{2pq(w-2p+1)}{q-1} \left(1 + \frac{q-2}{q}\right) \binom{\lfloor \frac{k}{2} \rfloor}{p} \binom{\lceil \frac{k}{2} \rceil}{p} (q-1)^{2p}}{q^u} \\ & + \frac{(n-k)^2(n+k)}{2}, \end{aligned}$$

and

$$P_{\mathrm{SUCC}}(w,p,u) = \min \left(\frac{\binom{\lfloor \frac{k}{2} \rfloor}{p} \binom{\lceil \frac{k}{2} \rceil}{p} \binom{n-k-u}{w-2p} \cdot N(w)}{\binom{n}{w}}, 1 \right).$$

Note that for random linear codes over $\mathbb{F}_q$ the expected number of weight-w codewords is given by

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

Quantum Improvements. In a quantum setting, the basic ISD algorithm by Prange can be accelerated via a Grover search [Ber10]. Overall, this results in a square-root gain on the amount of sets to be tested until an information set is found.

While there exist also quantum versions of more advanced ISD algorithms [KT17, Kir18], those yield only small asymptotic improvements at the cost of significant polynomial overhead and exponential demand for quantum random access memory. In turn, considering quantum circuits for actual parameters, the best quantum attack remains the Grover-search enhanced version of Prange's algorithm.

However, NIST's metrics imply constraints on the maximum depth of the quantum circuits used to launch an attack. Under such metrics, quantum attacks do not improve over classical approaches. We give a more detailed explanation in Appendix C.3.

Relying on random instances. For the related PEP random instances can be solved in polynomial time. However, those attacks do not translate to LEP as long as $q \geq 5$, making random instances of LEP a secure design choice. We give more details on this in Appendix C.1.

Attacks based on Low-weight Codeword Finding. The fastest algorithms for solving LEP as well as PEP are based on the search for codewords with low Hamming weight (or subcodes with small support) [Leo82, Beu21, BBPS23]. All these attacks share the common principle of looking at a small set of codewords (or subcodes) from which the action of μ can be recovered. For instance, Leon's algorithm [Leo82] requires to find, for both codes, all codewords with weight $\leq w$, that is

$$A = \{\mathbf{c} \in \mathfrak{C} \mid \mathrm{wt}(\mathbf{c}) \leq w\} \quad \text{and} \quad A' = \left\{\mathbf{c}' \in \mathfrak{C}' \mid \mathrm{wt}(\mathbf{c}') \leq w\right\}.$$

It then holds that $A\mathbf{Q} = A'$ and, when $w \ll n$, we further have $|A| \ll |\mathfrak{C}| = q^k$. Roughly speaking, since A and A' only contain a few codewords, reconstructing $\mathbf{Q}$ gets easy. Modern algorithms relax the requirements of Leon and, instead, aim at finding a sufficiently large number of *collisions*, i.e., pairs of codewords $\mathbf{c} \in \mathfrak{C}$, $\mathbf{c}' \in \mathfrak{C}'$ such that $\mathbf{c}\mathbf{Q} = \mathbf{c}'$. This idea was first proposed in [Beu21] and later refined in [BBPS23]. We give more details on those approaches in Appendix C.2.

The concrete gain of those attacks over Leon's algorithm depends on exact parameters. However, as a rule of thumb, those approaches outperform Leon's only if q is sufficiently large.

Conservative Design Criteria. In practice, the best attacks against LEP are those based on low-weight codeword finding. All those attacks use the following general attack framework:

i) Produce two lists L_1 and L_2 with short codewords that contain at least X collisions, i.e., X pairs of elements between L_1 and L_2 are mapped under $\mathbf{Q}$.

ii) Find those collisions.

iii) Use the collisions to reconstruct the secret monomial $\mathbf{Q}$.

In our parameter selection we lower bound the complexity of any algorithm following this framework. Therefore, we conservatively assume that a single collision between L_1 and L_2, i.e., a choice of $X = 1$ is enough for the algorithm to succeed and neglect the cost of steps ii) and iii).

Those assumptions lead to a particular conservative design. Usually, known attacks require X to be sufficiently large. Smaller X implies that less low-weight codewords need to be found, which leads to an overall smaller cost of (the usually dominating) step i). The small information on the monomial given by only a single collision would then lead to an expensive reconstruction phase in step iii), which is disregarded.

Furthermore, choosing X minimal also guards against future improvements of the reconstruction phase, i.e., against techniques that require a smaller number of collisions. It is also worth noting, that the most efficient algorithms [Beu21, BBPS23] actually require to find small-support subcodes rather than low-weight codewords; a task that is inherently more difficult. However, we assume that finding a collision between low-weight codewords is sufficient for all algorithms.

Taking into account these conservative design decisions, we use the following criterion to select secure LEP instances.

Criterion 1. *Let q denote the finite field size, n the code length and k the dimension. We consider only $q \geq 5$ and random codes. For a given security parameter λ, we select n, k, q such that for any $w \in \{1, \cdots, n\}$ finding lists $L_1 \subseteq \mathfrak{C}$ and $L_2 \subseteq \mathfrak{C}'$ with weight-w codewords and such that $L_1 \cap L_2\mathbf{Q} := \{(\mathbf{c}, \mathbf{c}') \in L_1 \times L_2 \mid \mathbf{c}' = \mathbf{c}\mathbf{Q}\}$ is non empty, takes time greater than 2^λ.*

This criterion translates into a simple parameter selection methodology. Therefore consider L_1 and L_2 of same size ℓ. The cost to produce these lists is

$$f\big(\ell, N(w)\big) \cdot C_{\mathrm{ISD}}(w).$$

Recall, that $C_{\mathrm{ISD}}(w)$ is the cost to find a random codeword of weight w and the term $f(\ell, N(w))$ accounts for the number of ISD calls to find ℓ *distinct* codewords. We now observe that, on average, we have

$$|L_1 \cap L_2\mathbf{Q}| = \frac{|L_1| \cdot |L_2|}{N(w)} = \frac{\ell^2}{N(w)},$$

collisions between the two lists. This is, because for each codeword $\mathbf{c} \in L_1$, there is only one match among the $N(w)$ codewords in $\mathfrak{C}'$, namely $\mathbf{c}' = \mathbf{c}\mathbf{Q}$, giving a collision. This match is present in L_2 with probability $\frac{\ell}{N(w)}$. In turn, to expect at least one collision, it must hold that

$$\ell^2 \geq N(w) \quad \text{or equivalently} \quad \ell \geq \sqrt{N(w)}.$$

Following our conservative design, we assume that such a single collision is sufficient and let $\ell = \sqrt{N(w)}$. Further, since this implies $\ell \ll N(w)$ we have (compare to Equation (1))

$$f\big(\ell, N(w)\big) \approx \ell = \sqrt{N(w)}.$$

Consequently, Criterion 1 translates into the following criterion, which is the basis for our parameter selection.

Criterion 2. *We consider random codes defined over $\mathbb{F}_q$ with $q \geq 5$, and choose q, n, k so that, for any w, it holds that*

$$\sqrt{N(w)} \cdot C_{\mathrm{ISD}}(w) > 2^\lambda.$$

The above criterion emphasizes the fact that, for appropriate choices of q, in the light of existing attacks, solving LEP reduces essentially to finding low-weight codewords.

5 Performance (2.B.2)

In this section we report performance figures for LESS. We begin by presenting a summary of the byte length for the various protocol objects, and the resulting key and signature sizes, in Table 2. The value of λ is set to $128, 192$ and 256 for security categories 1, 3 and 5, respectively.

		NIST Category 1	**NIST Category 3**	**NIST Category 5**
$\ell_{\mathsf{tree\_seed}}$	(byte)	16	24	32
$\ell_{\mathsf{sec\_seed}}$	(byte)	32	32	32
$\ell_{\mathsf{pub\_seed}}$	(byte)	32	32	32
ℓ_{salt}	(byte)	32	32	32
ℓ_{digest}	(byte)	32	32	32
$\ell_{\mathbb{F}_q}$	(byte)	$\lceil \log_2(q) \rceil / 8$		
$\ell_{\mathbf{G}_i}$	(byte)	$k(n-k)\lceil \log_2(q) \rceil / 8$		
ℓ_{mono}	(byte)	$k\Big(\lceil \log_2(n) \rceil + \lceil \log_2(q-1) \rceil \Big)/8$		
ℓ_{path}	(byte)	$\omega \log_2(t/\omega) \cdot \ell_{\mathsf{tree\_seed}}$		
ℓ_{sk}	(byte)	$\ell_{\mathsf{sec\_seed}}$		
ℓ_{pk}	(byte)	$(s-1)\ell_{\mathbf{G}_i} + \ell_{\mathsf{pub\_seed}}$		
ℓ_{sig}	(byte)	$\omega \cdot \ell_{\mathsf{mono}} + \ell_{\mathsf{path}} + \ell_{\mathsf{salt}}$		

Table 2: Choice of functions, and data sizes (in bytes).

Next, we present our choice of parameters for LESS. In addition to the design criteria and security arguments presented in the previous sections, we include in our thought process some considerations connected to implementation efficiency. For instance, we restrict our attention to the value $q = 127$, which allows for an optimal bit representation. Secondly, we try to avoid large data sizes and thus remain within the psychological threshold of 100 KiB. With this in mind, as anticipated in Section 2, we explore two different directions: a *balanced* configuration, where public key and signature are roughly of the same size, and a *short* configuration, where we shorten the signature size, at the cost of larger public keys. We use the nomenclature LESS-$\alpha\beta$, which recalls simultaneously the security level achieved (via the number $\alpha \in \{1, 3, 5\}$), and the characteristics of the resulting choice (via the letter β). To be precise, we use "b" for the "balanced" set, "s" for the "short" set and "i" for an "intermediate" set, present only for category 1.

Note that Table 3 does not report the size of the private key. This is because it is possible to compress the LESS private keys down to a single CSPRNG seed. Interestingly, while it is computationally convenient to keep the inverses of the monomial matrices stored as the private key (to avoid computing the inverses at signature time), this does not prevent the private key compression. It is in fact sufficient to randomly draw the contents of the private key and, during the key generation procedure, take care to multiply $\mathbf{G}_0$ by the inverse of the randomly generated values. In this fashion, as reported in Table 2, the entire LESS private key can be compressed down to a single CSPRNG seed, regardless of the number of monomials that compose it.

NIST Cat.	**Parameter Set**	**Code Params**			**Prot. Params**			**pk**	**sig**
		n	k	q	t	ω	s	(KiB)	(KiB)
1	LESS-1b	252	126	127	247	30	2	13.7	8.4
	LESS-1i				244	20	4	41.1	6.1
	LESS-1s				198	17	8	95.9	5.2
3	LESS-3b	400	200	127	759	33	2	34.5	18.4
	LESS-3s				895	26	3	68.9	14.1
5	LESS-5b	548	274	127	1352	40	2	64.6	32.5
	LESS-5s				907	37	3	129.0	26.1

Table 3: Parameter sets for LESS, and resulting data sizes.

5.1 Performance in Software

The most demanding operation in the LESS signature and verification algorithms is the computation of the RREF, which has essentially a complexity that is cubic in k. This calls for a careful tuning of the size of the prime field $\mathbb{F}_q$ on which the operations are computed. In LESS, we fix the value of q to 127 so that all the elements of $\mathbb{F}_q$ can be represented within a single byte. Then, we make use of Barrett's reduction technique [Bar87], which requires one triple-precision multiplication, one double-precision one, and an addition, instead of a more expensive division operation. Note that these multiplications can easily fit into a 32-bit multiplier with single-precision output, and that such a multiplication unit is readily available also on embedded platforms (e.g., ARM Cortex-M3 and ARM Cortex-M4), performing typically one multiplication per clock cycle.

Multiplications by monomial matrices can be implemented as simple column permutations of the corresponding generator matrix, combined with a scalar-by-vector multiplication over $\mathbb{F}_q$. Such operations allow for a significant amount of inner parallelism in the latter operation, which can be leveraged for consistent speedups if vector ISA extensions are available on the computing platform. Care should be taken in performing the column-wise permutation, as its value is part of the private key. To this end, the use of any constant-time sorting algorithm with optimal complexity is appropriate.

Below, we report the times obtained for our reference code, measured in Megacycles (Mcycles). The values are collected on an Intel Core i7-12700K, on a P-core, clocked at 4.9 GHz. Clock cycle values collected via `rtdscp`, as averages of 100 primitive runs. The computer is endowed with 64 GiB of PC5-19200 DDR5 and is running Debian 11. The source was compiled with `gcc` 10.2.1-20210110 (version packaged with the distribution), with `-O3 -march=native` compilation options. Due to the highly parallel nature of signing and verification (all t iterations can be computed in parallel), we expect a significant performance improvement from a parallel SIMD implementation.

The most recent version of the reference implementation is available on our website at:

`https://www.less-project.com/implementation.html`

NIST Cat.	**Parameter Set**	**KeyGen** (Mcycles)	**Sign** (Mcycles)	**Verify** (Mcycles)
1	LESS-1b	3.4	878.7	890.8
	LESS-1i	9.8	876.6	883.6
	LESS-1s	23.0	703.6	714.7
3	LESS-3b	9.3	7224.1	7315.8
	LESS-3s	18.3	8527.4	8608.6
5	LESS-5b	24.4	33787.7	34014.0
	LESS-5s	48.0	22621.5	22703.3

Table 4: Timings for the reference implementation of LESS.

To provide a hint at the improved performance that we can obtain by leveraging more advanced tools, we report below the results of an additional implementation. Since, as explained above, the RREF computation is by far the most expensive operation, this implementation is realized by amending the ANSI C reference code with Gaussian Elimination code implemented using AVX2 C intrinsics. The test system was a Dell OptiPlex XE4, a mid-range 2022 desktop system with Intel Core i7-12700 CPU running at 2.1 GHz. The test programs were executed on a single CPU thread with frequency scaling disabled. The system has 64GB of physical RAM and was running Ubuntu 22.04.2 LTS Linux operating system, and the C test code was compiled with gcc 11.3.0 packaged in that operating system. Compilation and optimization flags were \verb|-Wall -Wextra -Ofast -march=native|.

NIST Cat.	**Parameter Set**	**KeyGen** (Mcycles)	**Sign** (Mcycles)	**Verify** (Mcycles)
1	LESS-1b	0.9	263.6	271.4
	LESS-1i	2.3	254.3	263.4
	LESS-1s	5.1	206.6	213.4
3	LESS-3b	2.8	2446.9	2521.4
	LESS-3s	5.2	2984.3	3075.1
5	LESS-5b	6.4	10212.6	10458.8
	LESS-5s	11.7	6763.2	7016.5

Table 5: Timings for the additional implementation of LESS.

5.2 Performance in Hardware

In this section, we report preliminary performance and area results of LESS in hardware. In particular, we report results for Artix-7 FPGAs generated for part XC7A200TFBG484-3 using AMD-Xilinx Vivado 2022.2. Cycle counts were determined using simulation.

RREF in Hardware. As discussed above, the most computationally expensive operation in LESS is the conversion of generator matrices to RREF. In the hardware implementation, the majority of the area and latency come from this module. This architecture aims for high performance, so scaling and row addition operations are performed on an entire row in one clock cycle. The area and latency results of this design are reported in Table 6. The LUT area of this module is primarily dependent on n, as the number of arithmetic units is equal to n, and q, due to the bit width of arithmetic units. The operating frequency at a given parameter set is dependent on the value of k, for the current design. An internal translation table is used to track row swaps. This table grows in size with k, leading to a longer delay. The number of clock cycles required to perform the RREF operation, not including the load and readout time, is uniquely dependent on k.

Parameter Set	**Frequency** (MHz)	**LUTs** ($\times 10^3$)	**FFs** ($\times 10^3$)	**BRAM** (36 Kbit)	**Cycles** ($\times 10^3$)	**Latency** (μs)
LESS-1{b,i,s}	125	29.4	20.4	25.5	16.1	129.3
LESS-3{b,s}	111	46.3	33.8	40	40.4	364.3
LESS-5{b,s}	100	66.6	47.4	55	75.7	756.6

Table 6: RREF Implementation Results on Artix-7

Full Results. The maximum frequency of the LESS module is limited by the critical path of the RREF unit. The RREF module also consumes the majority of resources and takes up the most significant portion of the latency. Due to the comparably low latency and complexity of the other operations, most of the other hardware modules can be optimized for minimal resources since their latency can be hidden behind the RREF operation. Approximately 60% of the LUTs of the design are used in the RREF module depending on the parameter set, and about $70\% - 75\%$ of the cycles are spent in the RREF module.

The LUT consumption scales with the dimension of the matrices due to the impact these parameters have on the RREF unit. The BRAM utilization is also linked with the number of rows of the matrix. Most of the BRAM resources are consumed by the RREF design, which needs to be able to access an entire row of the matrix at once in order to achieve low latency for the operation. Thus, the level 5 parameter sets, which have the largest value for the parameters n and k, have the highest BRAM utilization.

As a reference we compare with the optimized software implementation benchmarked on a Intel Core i7-12700 CPU running at 2.1 GHz, which is $16.8 - 21\times$ faster than the hardware. Despite this significant difference in clock frequency, the hardware outperforms the software for all parameter sets. The key generation operation is $1.9 - 2.8\times$ faster in hardware. Signing is $2.4 - 3.3\times$ faster, and verification is $2.5 - 2.4\times$ faster. The performance could be increased further through the use of a higher-end FPGA, which can enable higher clock frequencies, or through implementation as an ASIC. The operations which benefit the most from hardware acceleration are hashing and the RREF operation. The benefit of accelerating the hardware for RREF operation increases as the size of the matrices increases. This explains why the hardware outperforms the software by a larger margin as the security level increases.

Utilization					Performance						
Parameter Set	LUT ($\times 10^3$)	FF ($\times 10^3$)	DSP	BRAM	Frequency (MHz)	KeyGen		Sign		Verify	
						Cycles ($\times 10^3$)	(μs)	Cycles ($\times 10^3$)	(μs)	Cycles ($\times 10^3$)	(μs)
LESS-1b	56.9	36.5	0	41	125	25.6	204.8	6,492.7	51,941.6	6,435.4	51,483.0
LESS-1i						72.8	582.3	6,398.7	51,189.9	6,357.5	50,859.9
LESS-1s						167.3	1,338.5	5,296.5	42,371.7	5,262.6	42,100.7
LESS-3b	81.4	53.8	0	77	111	62.9	566.4	48,548.7	436,938.5	48,396.3	435,566.3
LESS-3s						120.8	1,087.5	57,199.6	514,796.1	57,066.6	513,599.3
LESS-5b	113.3	75.9	0	120.5	100	116.8	1,051.0	161,827.4	1,456,446.6	161,543.9	1,453,895.1
LESS-5s						224.1	2,017.2	108,579.0	977,211.3	108,382.7	975,443.9

Table 7: Area and Performance of LESS on Artix-7.

6 Known Answer Tests (2.B.3)

The LESS KAT archive is available at the following link:

https://www.less-project.com/implementation.html

7 Advantages and Limitations (2.B.6)

Advantages

Design Flexibility and Scalability. LESS is constructed by converting a zero-knowledge protocol via Fiat-Shamir, with the addition of several optimizations. As a result, we obtain a very flexible scheme which offers, for each security level, several tradeoffs between simplicity, speed, public key size, and signature size. Furthermore, parameters are easy to select and scale gracefully.

Group Action Structure. Unlike its predecessors, LESS is the first code-based signature scheme not directly relying on the hardness of decoding. Instead, LESS exploits the group action structure given by isometries in the Hamming metric. This has several advantages, such as being able to utilize a specific zero-knowledge protocol with soundness 1/2, enabling some of the aforementioned optimizations, and lending itself nicely to several additional formulations.

Advanced Functionalities. Due to its particular structure, the LESS framework can be utilized to effectively build signature schemes with additional properties. For instance, ring signatures, identity-based signatures [BBN+22] and threshold signatures [BBMP23] can all be designed with the same components of LESS and comparable costs, filling a noticeable gap in literature.

Solid Security Foundations. The security of LESS is based on the Linear Equivalence Problem (LEP). This is a traditional problem from coding theory, which is well-known and has been studied for decades. In fact, even simply determining whether two linear codes are equivalent (i.e. the decisional version of LEP) is typically considered a difficult problem by coding theorists. Furthermore, weak instances have been identified and investigated in literature, and they are easy to avoid; this being the case, the best known solvers boil down to codeword searching, and can therefore rely on the established security track of the Syndrome Decoding Problem (SDP).

Limitations

Larger Public Keys. Compared to some of its competitors, LESS features larger public keys, in the order of at least a few kilobytes. This is due to the public key being matrices (in standard form) of moderate size. While, with our parameter choice, our data size is still compact enough to fit in most scenarios (e.g. microcontrollers), such larger keys could be a limitation for applications where many public keys need to be transmitted.

Computational Bottleneck. The performance of LESS depends in overwhelming proportion on the cost of the Gaussian elimination algorithm (to compute the RREF). This makes it so that instances with larger public keys are also the fastest ones, leading to a potentially unpleasant tradeoff. While this cost can be alleviated by various means (e.g. parallelization, precomputation, hardware acceleration), it may still be a problem for applications where speed is the paramount priority.

References

[AABN02] Michel Abdalla, Jee Hea An, Mihir Bellare, and Chanathip Namprempre. From identification to signatures via the Fiat-Shamir transform: Minimizing assumptions for security and forward-security. In *EUROCRYPT*, pages 418–433. Springer, 2002.

[ABC$^+$] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece: conservative code-based cryptography.

[Bar87] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 311–323. Springer, Heidelberg, August 1987.

[BBMP23] Michele Battagliola, Giacomo Borin, Alessio Meneghetti, and Edoardo Persichetti. Cutting the GRASS: Threshold GRoup Action Signature Schemes. *Cryptology ePrint Archive*, 2023.

[BBN$^+$22] Alessandro Barenghi, Jean-François Biasse, Tran Ngo, Edoardo Persichetti, and Paolo Santini. Advanced signature functionalities from the code equivalence problem. *International Journal of Computer Mathematics: Computer Systems Theory*, 0(ja):1–0, 2022.

[BBPS23] Alessandro Barenghi, Jean-François Biasse, Edoardo Persichetti, and Paolo Santini. On the computational hardness of the code equivalence problem in cryptography. *Advances in Mathematics of Communications*, 17(1):23–55, 2023.

[Ber10] Daniel J. Bernstein. Grover vs. McEliece. In Nicolas Sendrier, editor, *The Third International Workshop on Post-Quantum Cryptography, PQCRYPTO 2010*, pages 73–80. Springer, Heidelberg, May 2010.

[Beu21] Ward Beullens. Not enough LESS: An improved algorithm for solving code equivalence problems over $\mathbb{F}_q$. In *Selected Areas in Cryptography: 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers*, pages 387–403. Springer, 2021.

[BHH$^+$15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. Sphincs: Practical stateless hash-based signatures. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 368–397, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[BJMM12] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in $2^{n/20}$: How $1 + 1 = 0$ improves information set decoding. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 520–536. Springer, Heidelberg, April 2012.

[BKP20] Ward Beullens, Shuichi Katsumata, and Federico Pintore. Calamari and Falafl: Logarithmic (linkable) ring signatures from isogenies and lattices. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 464–492. Springer, Heidelberg, December 2020.

[BLP11] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Smaller decoding exponents: Ball-collision decoding. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 743–760. Springer, Heidelberg, August 2011.

[BM17] Leif Both and Alexander May. Optimizing BJMM with nearest neighbors: full decoding in 22/21n and McEliece security. In *WCC workshop on coding and cryptography*, volume 214, 2017.

[BM18] Leif Both and Alexander May. Decoding linear codes with high error rate and its impact for LPN security. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*, pages 25–46. Springer, Heidelberg, 2018.

[BMPS20] Jean-François Biasse, Giacomo Micheli, Edoardo Persichetti, and Paolo Santini. LESS is more: Code-based signatures without syndromes. In Abderrahmane Nitaj and Amr Youssef, editors, *AFRICACRYPT*, pages 45–65. Springer, 2020.

[BN06] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 390–399. ACM Press, October / November 2006.

[Cha22] André Chailloux. On the (In) security of optimized Stern-like signature schemes. In *[Informal] Proceedings of WCC 2022: The Twelfth International Workshop on Coding and Cryptography, March 7 - 11, 2022, Rostock (Germany). URL: https://www.wcc2022.uni-rostock.de/storages/uni-rostock/Tagungen/WCC2022/Papers/WCC_2022_paper_54.pdf*, 2022.

[DFMS19] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. Security of the Fiat-Shamir transformation in the quantum random-oracle model. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part II*, volume 11693 of *LNCS*, pages 356–383. Springer, Heidelberg, August 2019.

[Dum91] Ilya Dumer. On minimum distance decoding of linear codes. In *Proc. 5th Joint Soviet-Swedish Int. Workshop Inform. Theory*, pages 50–52, 1991.

[EB22] Andre Esser and Emanuele Bellini. Syndrome decoding estimator. In *Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography*, volume 13177 of *Lecture Notes in Computer Science*, pages 112–141. Springer, 2022.

[Ess22] Andre Esser. Revisiting nearest-neighbor-based information set decoding. Cryptology ePrint Archive, Report 2022/1328, 2022. https://eprint.iacr.org/2022/1328.

[FG19] Luca De Feo and Steven D. Galbraith. Seasign: Compact isogeny signatures from class group actions. 11478:759–789, 2019.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194. Springer, 1986.

[Kir18] Elena Kirshanova. Improved quantum information set decoding. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*, pages 507–527. Springer, Heidelberg, 2018.

[KT17] Ghazal Kachigar and Jean-Pierre Tillich. Quantum information set decoding algorithms. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, pages 69–89. Springer, Heidelberg, 2017.

[LB88] P. J. Lee and E. F. Brickell. An observation on the security of mceliece's public-key cryptosystem. In D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, and Christoph G. Günther, editors, *Advances in Cryptology — EUROCRYPT '88*, pages 275–280, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[Leo82] J. Leon. Computing automorphism groups of error-correcting codes. *IEEE Transactions on Information Theory*, 28(3):496–511, 5 1982.

[LZ19] Qipeng Liu and Mark Zhandry. Revisiting post-quantum fiat-shamir. In *Advances in Cryptology - CRYPTO 2019*, pages 326–355, 2019.

[Meu13] Alexander Meurer. *A coding-theoretic approach to cryptanalysis.* PhD thesis, Ruhr University Bochum, 2013.

[MMT11] Alexander May, Alexander Meurer, and Enrico Thomae. Decoding random linear codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 107–124. Springer, Heidelberg, December 2011.

[MO15] Alexander May and Ilya Ozerov. On computing nearest neighbors with applications to decoding of binary linear codes. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 203–228. Springer, Heidelberg, April 2015.

[Pet10] Christiane Peters. Information-set decoding for linear codes over f q. In *Post-Quantum Cryptography: Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings 3*, pages 81–94. Springer, 2010.

[Pra62] E. Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.

[PS23] Edoardo Persichetti and Paolo Santini. A New Formulation of the Linear Equivalence Problem and Shorter LESS Signatures. *Cryptology ePrint Archive*, 2023.

[Sae17] Mohamed Ahmed Saeed. Algebraic approach for code equivalence. *PhD thesis*, 2017.

[Sen00] Nicolas Sendrier. The support splitting algorithm. *Information Theory, IEEE Transactions on*, pages 1193 – 1203, 08 2000.

[SS13] Nicolas Sendrier and Dimitris E. Simos. The hardness of code equivalence over $\mathbb{F}_q$ and its application to code-based cryptography. In Philippe Gaborit, editor, *PQCrypto 2013*, pages 203–216, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Ste89] Jacques Stern. A method for finding codewords of small weight. In Gérard Cohen and Jacques Wolfmann, editors, *Coding Theory and Applications*, pages 106–113, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

A Mathematical Background

Linear Codes. As seen in Section 1.1, an $[n,k]$-*linear code* $\mathfrak{C}$ over $\mathbb{F}_q$ is a k-dimensional vector subspace of $\mathbb{F}_q^n$. The value n is called *length* of the code, and the value k is its dimension. The code can be represented intuitively by choosing a basis for the vector space, whose elements are organized as rows of a matrix $\mathbf{G} \in \mathbb{F}_q^{k\times n}$, called *generator matrix*. Then, the generator matrix defines the code as a mapping between vectors $\mathbf{u} \in \mathbb{F}_q^k$ and the corresponding words $\mathbf{uG}$. Obviously, there exist more than one generator matrix for the same code, corresponding to different choices of basis. It follows that all generator matrices are connected via a change-of-basis matrix, i.e. an invertible matrix $\mathbf{S} \in \mathrm{GL}_k(q)$ such that $\mathbf{G}' = \mathbf{SG}$. Alternatively, a linear code can be represented as the kernel of a matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, known as *parity-check matrix*, i.e. $\mathfrak{C} = \{\mathbf{x} \in \mathbb{F}_q^n : \mathbf{Hx}^T = 0\}$. Once again, the parity-check matrix of a code is not unique. For both cases, the standard choice is given by the aforementioned *systematic form*. For the generator matrix, this corresponds to $\mathbf{G} = (\mathbf{I}_k \mid \mathbf{M})$, while the systematic form of the parity-check matrix is given by $\mathbf{H} = (-\mathbf{M}^T \mid \mathbf{I}_{n-k})$.

For every linear code, we can define the *dual code* as the set of words that are orthogonal to the code, i.e. $\mathfrak{C}^\perp = \{\mathbf{y} \in \mathbb{F}_q^n \;:\; \forall \mathbf{x} \in \mathfrak{C},\;\; \mathbf{x}\cdot\mathbf{y}^T = 0\}$. It is then easy to see that a parity-check matrix of a linear code is a generator of its dual, and viceversa. In fact, it must be that $\mathbf{G}\cdot\mathbf{H}^T = \mathbf{0}_{k\times(n-k)}$. Codes that are contained in their dual, i.e. $\mathfrak{C} \subseteq \mathfrak{C}^\perp$, are called *weakly self-dual*, and codes that are equal to their dual, i.e. $\mathfrak{C} = \mathfrak{C}^\perp$, are called simply *self-dual*.

For a random code, the number of codewords with Hamming weight w (excluding scalar multiples) is estimated by

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

The above quantity corresonds to the expected number of weight-w codewords in a random codeword.

B Proofs

B.1 Identification

We begin by recalling the Sigma protocol. In the protocol below, the hash value is the *commitment*, the bit chosen by the verifier is the *challenge*, and the monomial matrix transmitted is the the prover's *response*. A *transcript* consists of the entirety of the communication between prover and verifier; in this case, the triple $(\mathsf{cmt}, \mathsf{ch}, \mathsf{rsp})$. We now show that this protocol satisfies the necessary security requirements for identification schemes.

Completeness.

It is immediate to check that the protocol is correct, and an honest prover always gets accepted. In fact, if $b = 0$ the verifier receives $\tilde{\mathbf{Q}}$ and verification becomes $\textsc{Hash}(\mathrm{RREF}(\mathbf{G}_b\overline{\mathbf{Q}})) = \textsc{Hash}(\mathrm{RREF}(\mathbf{G}_0\tilde{\mathbf{Q}}))$ which yields exactly h. On the other hand, if $b = 1$, then $\overline{\mathbf{Q}} = \mathbf{Q}^{-1}\tilde{\mathbf{Q}}$ and

Public Data	System parameters $q, n, k \in \mathbb{N}$, $\mathbf{G}_0 \in \mathbb{F}_q^{k \times n}$ and hash function $\textsc{Hash}$.
Private Key	$\mathbf{Q} \in \mathsf{M}_n$.
Public Key	$\mathbf{G}_1 = \mathrm{RREF}(\mathbf{G}_0\mathbf{Q})$.

PROVER		VERIFIER
$\widetilde{\mathbf{Q}} \xleftarrow{\$} \mathsf{M}_n$, $\widetilde{\mathbf{G}} \leftarrow \mathrm{RREF}(\mathbf{G}_0\widetilde{\mathbf{Q}})$	$\xrightarrow{\mathsf{cmt}}$	
$\mathsf{cmt} \leftarrow \textsc{Hash}(\widetilde{\mathbf{G}})$		
	$\xleftarrow{\mathsf{ch}}$	$\mathsf{ch} \xleftarrow{\$} \{0,1\}$
$\mathsf{rsp} \leftarrow (1-\mathsf{ch})\tilde{\mathbf{Q}} + \mathsf{ch}\cdot\mathbf{Q}^{-1}\tilde{\mathbf{Q}}$	$\xrightarrow{\mathsf{rsp}}$	Accept if
		$\textsc{Hash}(\mathrm{RREF}(\mathbf{G}_b \cdot \mathsf{rsp})) = \mathsf{cmt}$

Figure 4: The Sigma Protocol.

we have $\textsc{Hash}(\mathrm{RREF}(\mathbf{G}_b\overline{\mathbf{Q}})) = \textsc{Hash}(\mathrm{RREF}(\mathbf{G}_1\mathbf{Q}^{-1}\tilde{\mathbf{Q}})) = \textsc{Hash}(\mathrm{RREF}(\mathrm{RREF}(\mathbf{G}_0\mathbf{Q})\mathbf{Q}^{-1}\tilde{\mathbf{Q}}))$, which is again equal to h.

Honest-Verifier Zero-Knowledge.

The next step is to show that the produced responses do not leak information about the private key. This can be done by proving that there exists a probabilistic polynomial-time simulator algorithm $\mathcal{S}$ that, without the knowledge of the private key, is able to produce a transcript which is indistinguishable from one obtained after an interaction with an honest verifier. To this end, we introduce the following straightforward Lemma (which we will not prove).

Lemma 1. *For any $\mathbf{A} \in \mathsf{M}_n$ and $\mathbf{B} \xleftarrow{\$} \mathsf{M}_n$, the product $\mathbf{A}^{-1}\mathbf{B}$ is uniformly distributed over M_n.*

The simulator works as follows.

- When the challenge is $b = 0$, it can trivially simulate correctly by choosing a matrix $\tilde{\mathbf{Q}}$ uniformly at random. This, in fact, corresponds to a legitimate response for this challenge, and doesn't include the secret.

- When the challenge is $b = 1$, the simulator again chooses a matrix, say $\mathbf{Q}^*$, uniformly at random. By Lemma 1, we have seen that the product $\mathbf{Q}^{-1}\tilde{\mathbf{Q}}$ that would be output by an honest execution of the protocol is uniformly distributed among all monomial matrices. Therefore $\mathcal{S}$ is able to simulate correctly in this case.

This simple argument shows that both responses are actually indistinguishable from randomly generated ones, and thus do not reveal any secret.

Soundness.

Finally, we prove that the protocol is 2-special sound. We do this by describing an extractor algorithm and showing that it is able to find a witness, i.e. solve the code equivalence problem. To this end, let $\mathcal{A}$ be an adversary that is given a LEP instance $\{\mathbf{G}_0, \mathbf{G}_1\}$. The algorithm proceeds as follows.

To begin, set $\mathbf{G}_0$ and $\mathbf{G}_1$ as public data and public key for the identification scheme. The adversary then tries to obtain two transcripts $(\mathsf{cmt}, \mathsf{ch}_0, \mathsf{rsp}_0,)$ and $(\mathsf{cmt}, \mathsf{ch}_1, \mathsf{rsp}_1)$ such that $\mathsf{ch}_0 \neq \mathsf{ch}_1$ and the verifier accepts $(\mathsf{cmt}, \mathsf{ch}_i, \mathsf{rsp}_i)$ for $i \in \{0, 1\}$: in other words, the transcripts are such that both challenges are satisfied for the same commitment. To do this, the adversary can rely on the forking lemma (see next section), rewinding the tape until two such transcripts are found. At this point, the two responses must be two monomial matrices $\overline{\mathbf{Q}}_0$ and $\overline{\mathbf{Q}}_1$ such that

$$\textsc{Hash}(\mathrm{RREF}(\mathbf{G}_0\overline{\mathbf{Q}}_0)) = \textsc{Hash}(\mathrm{RREF}(\mathbf{G}_1\overline{\mathbf{Q}}_1)).$$

Unless a collision for the hash function was found, this implies that

$$\mathrm{RREF}(\mathbf{G}_0\overline{\mathbf{Q}}_0) = \mathrm{RREF}(\mathbf{G}_1\overline{\mathbf{Q}}_1).$$

Now, since two matrices with the same reduced-row echelon form define the same linear code, we have that

$$\mathbf{S}\mathbf{G}_0\overline{\mathbf{Q}}_0 = \mathbf{G}_1\overline{\mathbf{Q}}_1$$

for some invertible matrix $\mathbf{S}$ or, equivalently,

$$\mathbf{S}\mathbf{G}_0\overline{\mathbf{Q}}_0(\overline{\mathbf{Q}}_1)^{-1} = \mathbf{G}_1.$$

In fact, since $\mathbf{G}_1$ is in reduced-row echelon form, which is unique, it must be that

$$\mathbf{G}_1 = \mathrm{RREF}(\mathbf{S}\mathbf{G}_0\overline{\mathbf{Q}}_0(\overline{\mathbf{Q}}_1)^{-1}) = \mathrm{RREF}(\mathbf{G}_0\overline{\mathbf{Q}}_0(\overline{\mathbf{Q}}_1)^{-1}).$$

It is then evident that the monomial $\overline{\mathbf{Q}}_0(\overline{\mathbf{Q}}_1)^{-1}$ is the desired witness; this can be calculated immediately from the two responses.

B.2 Fiat-Shamir

The following theorem was proved in [AABN02] and states the security of the Fiat-Shamir transform in all generality.

Theorem B.1. *Consider a non-trivial canonical identification protocol that is secure against impersonation under passive attacks. Then the signature scheme derived using the Fiat-Shamir transform is secure against chosen-message attacks in the random oracle model.*

The proof utilizes the famous Forking Lemma, which we recall below in the formulation of Bellare-Neven (see [BN06]).

Lemma 2. *Fix an integer $Q \geq 1$ and a set H of size $|H| \geq 2$. Let $\mathcal{A}$ be a randomized algorithm that takes as input elements $h_1, \ldots, h_Q \in H$ and outputs a pair (J, σ) where $1 \leq J \leq Q$ with probability P. Consider the following experiment:*

1. *Choose $h_1, \ldots, h_Q$ uniformly at random from H.*
2. *$\mathcal{A}(h_1, \ldots, h_Q)$ outputs (I, σ) with $I \geq 1$.*
3. *Choose $h'_I, \ldots, h'_Q$ uniformly at random from H.*
4. *$\mathcal{A}(h_1, \ldots, h_{I-1}, h'_I, \ldots, h'_Q)$ outputs (I', σ').*

Then the probability that $I' = I$ and $h'_I \neq h_I$ is at least

$$P\left(\frac{P}{Q} - \frac{1}{|H|}\right).$$

C Known Attacks

C.1 Attacks Reducing to Permutation Equivalence

A linear equivalence between two codes implies a permutation equivalence between the *closure* of both codes. The closure of a code $\mathcal{C}$ is defined as the code $\{\mathbf{c} \otimes \mathbf{a} \mid c \in \mathcal{C}\}$, where $\mathbf{a} = (a_1, \ldots, a_{q-1})$ is an arbitrary ordering of the elements of $\mathbb{F}_q^*$. In turn, the closure of a code of length n and dimension k, is a code of same dimension k and increased length $n(q-1)$.

A possible attack strategy is, hence, to apply known algorithms to solve the permutation equivalence problem to the linear closure of the codes. However, the complexity in this case is exponential in the dimension of the *hull* of the given codes, which is the intersection between the code and its dual. Since, for $q \geq 5$, the closure of any code is weakly self-dual, i.e., its hull dimension is maximal and corresponding to $\min(n-k, k)$, this strategy quickly becomes inefficient. For completeness, we report anyway the two common approaches for this type of attacks.

Support Splitting Algorithm. The Support Splitting Algorithm (SSA) [Sen00] is an algorithm to solve the permutation equivalence problem, i.e., when the monomial matrix $\mathbf{Q}$ in the equivalence is a permutation matrix[2]. The algorithm defines the concept of a signature of a code, which is invariant under permutations. Concretely, the signature used is defined as the weight enumerator of the hull and can therefore be computed in time $\mathcal{O}(q^h)$, where h is the dimension of the hull. Then by puncturing both codes and comparing their signature, information on the permutation can be obtained.

Since in the linear equivalence case the algorithm has to be applied to the linear closure of the code, whose hull has dimension $h = \min(n-k, k)$, for $q \geq 5$ the computation of the signature becomes inefficient [SS13].

Algebraic Approaches. Algebraic approaches model the permutation equivalence between the codes as a system of polynomial equations. The main drawback when applying the technique to the closure of the code is the large amount of variables due to the increased length $n \cdot q$ of the code. In [Sae17] many tricks were applied to reduce the amount of variables. However, they remain efficient only for $q < 5$.

C.2 Attacks based on Low-weight Codeword Finding

The best attacks known for solving the linear equivalence problem between two codes exploit that the Hamming weight of codewords (and, more generally, the support size of subcodes) is invariant under the action of monomial transformations and, hence, leverage techniques for short codewords (and subcodes) finding. The problem of finding short codewords in random codes is known to be NP-hard and is the foundation for some code-based constructions which are considered to be most conservative. Therefore, even if the linear equivalence problem does not enjoy NP-completeness guarantees as low-weight codeword finding, its practical hardness depends on the exact same class of algorithms. Also, the algorithms in this category are basically extensions of each other, always

[2] Notice that, here, signature refers to the name of a mathematical function, and is in no way related to the concept of a cryptographic digital signature.

improving on the running time of their successor. Also, they rely on some common ingredients and coding theory concepts, which we recall in the following.

Let $\mathfrak{C}, \mathfrak{C}' \subseteq \mathbb{F}_q^n$, such that $\mathfrak{C}' = \mathfrak{C}\mathbf{Q}$ for some transformation $\mathbf{Q}$. The most efficient attacks against LEP have all a common operating principle: they first determine subsets $A \subseteq \mathfrak{C}$ and $A' \subseteq \mathfrak{C}'$ such that $A' = A\mathbf{Q} = \{\mathbf{cQ} \mid \mathbf{c} \in A\}$, then use such subsets to extract information about $\mathbf{Q}$. In all existing attacks, A and A' contain short codewords or subcodes.

Leon's Algorithm. To find the linear equivalence $\mathbf{Q}$ between two codes, Leon's algorithm [Leo82] first finds all codewords of minimum weight in both codes. This results in two lists of codewords L_1 and L_2 for which it holds that $L_1 = L_2\mathbf{Q} = \{\mathbf{cQ} \mid \mathbf{c} \in L_1\}$. This guarantees that, setting $A = L_1$ and $A' = L_2$, one has $A' = A\mathbf{Q}$. Usually those lists inherit enough structure so that $\mathbf{Q}$ is uniquely identified. If not, one might extend L_1, L_2 by all codewords of slightly higher weight. Leon shows that the cost of recovering $\mathbf{Q}$ from L_1, L_2 is dominated by the initial construction of L_1, L_2.

Beullens' Algorithm. Beullens proposed an algorithm to retrieve the hidden $\mathbf{Q}$ by exploiting 2-dimensional subcodes with support w [Beu21]. This, in several cases, can provide a non trivial speed-up with respect to Leon's algorithm, since it may be not necessary to find all such subcodes. Indeed, to determine if two such 2-dimensional subcodes are indeed connected through a monomial transformation, Beullens defines a canonical representation of the basis, which can be computed in polynomial time. Then, if two subcodes form a collision, i.e., share the same canonical representation of their bases, they are linearly equivalent and, more importantly, with very high probability they are connected through $\mathbf{Q}$. In other words, Beullens' algorithm first populates two lists L_1 and L_2, each with 2-dimensional subcodes with support size w (L_1 with subcodes from $\mathfrak{C}$, L_2 with subcodes from $\mathfrak{C}'$), and then determines collisions. Unless q is small, we have that, with high probability, collisions correspond to $L_2 \cap L_1\mathbf{Q}$. In such cases, this approach can be more efficient than Leon's algorithm since it does not require to find all subcodes with support size w.

For finding 2-dimensional subcodes with support size w Beullens uses an adaptation of Lee-Brickel's ISD algorithm [Beu21]. Similar to the case of Leon, the running time of Beullens algorithm is dominated by finding a sufficient amount of 2-dimensional subcodes of small support. Indeed, the adversary should set up the attack so that, on average, the number of collisions is approximately $2\ln(n)$.

BBPS Algorithm. In [BBPS23], Barenghi, Biasse, Persichetti and Santini improved on Beullens' algorithm by changing how the 2-dimensional subcodes with support size w are constructed. They first find codewords of weight w', and then search for combinations among them that form 2-dimensional codes of support w. This allows a direct application of advanced ISD techniques to find weight-w' codewords.[3] Also by ensuring that the overlap between the pairs of codewords used in the subcode construction is small, i.e. ensuring that $2w' - w$ is small, the probability for finding linear equivalent subcodes increases. Overall this improves the running time, with respect to Beullens' algorithm. Yet, the attack still requires to first find low weight codewords, and then matching them to find pairs of subcodes that are mapped through the secret transformation.

C.3 Quantum Hardness

NIST's metrics for quantum security restrict the depth of any quantum circuit used for an attack to $2^{\texttt{maxdepth}}$. This limitation accounts for the practical difficulty in constructing large quantum computers. In turn quantum attacks, require to build short quantum circuits which are reapplied several times. In [EB22] it is shown that for Prange's algorithm such an attack has complexity

$$T_{\mathrm{QP}} = \frac{(D_{\mathrm{GE}})^2}{q \cdot 2^{\texttt{maxdepth}}},$$

where D_{GE} describes the depth of a circuit implementing the Gaussian elimination procedure and q is the probability of sampling an information set given by

$$q = N_1(w) \cdot \frac{\binom{n-w}{k}}{\binom{n}{k}}.$$

Note that classical Prange has complexity of about $T_{\mathrm{P}} = \frac{D_{\mathrm{GE}}}{q}$. Moreover, from guarding against classical Stern, which is more efficient than Prange, we know that T_{P} is large enough to at least fulfill the bit security guarantees of the classical security levels given by NIST, which implies

$$T_{\mathrm{P}} \geq 2^{\beta},$$

with $\beta = 143, 207, 272$ for categories 1, 3 and 5 respectively .

Now, NIST specifies the quantum security levels for category 1,3 and 5 as $2^{\alpha-\texttt{maxdepth}}$ with $\alpha = 157, 221, 285$, respectively. Summarizing, we get

$$T_{\mathrm{QP}} = \frac{T_{\mathrm{P}} \cdot D_{\mathrm{GE}}}{2^{\texttt{maxdepth}}} > \frac{2^{\beta} \cdot D_{\mathrm{GE}}}{2^{\texttt{maxdepth}}} \overset{!}{>} 2^{\alpha-\texttt{maxdepth}},$$

which is fulfilled as long as $D_{GE} > 2^{\alpha-\beta}$. For the different security levels we have $\alpha - \beta$ equal to 14, 14 and 13 respectively. Since the dimensions of the involved matrices are at least of order $m \geq 2^7$ even an optimistic estimate of $D_{GE} = m^2$ yields the desired security level. Due to various omitted polynomial factors in the translation from Stern to Prange and our general conservative estimation of the attack costs practical quantum circuits are likely to have even higher complexity. It is therefore reasonable to assume that the classical hardness of our parameter sets implies the hardness against quantum Prange under NIST metrics.

[3] Without adaptations those algorithm can not be used to find subcodes with small support.