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

Tung Chou, Academia Sinica

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

Markku-Juhani O. Saarinen, Tampere University

Paolo Santini, Università Politecnica delle Marche

Robert Wallace, George Mason University

Floyd Zweydinger, Technology Innovation Institute

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): epersichetti@fau.edu

Postal Address and Telephone (if absolutely necessary):
Edoardo Persichetti, Department of Mathematics and Statistics, Florida Atlantic University, 777
Glades Rd, Boca Raton FL 33431 (USA), +1 (561) 322 0647.

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

Version: 2.0

Changelog

- Inclusion of *canonical forms* for response compression in the underlying Sigma protocol, leading to drastic signature size decreases.

- New choice of protocol parameters to optimize the canonical forms setting. In particular, some of the proposed instances now have a much smaller amount of rounds, leading also to improvements on the overall computational complexity. Note that the coding theory parameters remain the same as in Round 1.

- Various implementation improvements, including

  - multiple speedups for the field arithmetic;
  - improved RREF computation by exploiting pivots reusing. For verification, this has been implemented in a non-constant time fashion to fully exploit pivots reusing. For signature generation, to avoid side-channel information leakage, the number of reused pivots is bounded;
  - canonical forms are implemented in a non-constant-time fashion, for both signature generation and verification. To prevent from side-channel information leakage, signature generation now employs a *blinding* technique;
  - the GGM tree construction has been slightly modified. The worst-case size on the number of released nodes, for the new construction, is tighter than the bound used for the Round 1 submission.

- Added Floyd Zweydinger and Tung Chou for substantial contributions to the LESS team.

Contents

Notation	Semantics
$[i, j]$	the set $\{i, \dots, j-1\}$
$[j]$	the set $[0, j]$
a	a scalar
$\mathbf{a}$	a vector
a_i	the i-th entry of a vector
$\mathbf{a}[i]$	the i-th entry of a vector
$\mathbf{a}_J$	vector formed by the entries of $\mathbf{a}$ indexed by J
$\mathbf{A}$	a matrix
$\mathbf{A}[i, j]$	the entry at the i-th row and j-th column
$\mathbf{A}[i :]$	the i-th row
$\mathbf{A}[: j]$	the j-th column
$\mathbf{A}_J$	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 $\mathbb{F}_q$
$\mathbb{S}_{t,w}$	set of strings of length t and weight w over $\mathbb{Z}_s$
$a \overset{\$}{\leftarrow} A$	sampling a uniformly at random from A
λ	a security parameter

Table 1: Notation used in this document.

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

Linear Codes. An $[n, k]$-*linear code* $\mathscr{C}$ 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 for generator matrices, called *systematic form*, which yields a matrix of the form $\mathbf{G} = (\mathbf{I}_k \mid \mathbf{M})$. Such a standard form is typically obtained by calculating the Row-Reduced Echelon Form (RREF), starting from any other generator matrix for the code. In general, doing so may return a matrix which does not have full rank. If so, there is a simple procedure to obtain a matrix in systematic form, consisting of reducing with respect to a different minor. This is effectively the same as permuting some columns, i.e. we obtain a systematic form as via a procedure which we denote by RREF*, so that RREF$^*(\mathbf{G}) = $ RREF$(\pi(\mathbf{G}))$ for some permutation π; if the first k columns of $\mathbf{G}$ form a non-singular matrix, then $\pi = id$. Note that, in practice, the procedure RREF* returns both the systematic form of the input matrix, as the well as the permutation π needed to do that.

Isometries. These are maps that preserve the Hamming weight. In LESS, we need three types of isometries, forming the following three sets:

- S_n: the group of *permutations* of length-n, usually known as *symmetric group.* Using the two-lines notation, a permutation π can be expressed as

$$\pi \coloneqq \begin{pmatrix} 1 & 2 & \cdots & n \\ i_1 & i_2 & \cdots & i_n \end{pmatrix},$$

  such that π moves the j-th element to position i_j. For a matrix $\mathbf{G}$ with n columns, $\mathbf{G} = (\mathbf{g}_1, \cdots, \mathbf{g}_n)$, it holds that $\pi(\mathbf{G}) = \big(\mathbf{g}_{\pi^{-1}(1)}, \cdots, \mathbf{g}_{\pi^{-1}(n)}\big)$. Each permutation can be represented through an $n \times n$ matrix $\mathbf{P}$, called *permutation matrix*, having exactly one 1 entry in each row and column, so that $\pi(\mathbf{G}) = \mathbf{G} \cdot \mathbf{P}$.

- M_n: the group of *monomial maps* of length n, usually known as *monomial group.* Every monomial map is comprised of a pair $\mu \coloneqq (\pi, \mathbf{v})$ with $\pi \in S_n$ and $\mathbf{v} \in \mathbb{F}_q^{*n}$, acting as $\mu(\mathbf{G}) = \big(v_1 \cdot \mathbf{g}_{\pi^{-1}(1)}, \cdots, v_n \cdot \mathbf{g}_{\pi^{-1}(n)}\big)$. Each monomial map can be represented through an $n \times n$ matrix $\mathbf{Q}$, called *monomial matrix*, having exactly one non-zero element in each row and column and such that $\mu(\mathbf{G}) = \mathbf{G} \cdot \mathbf{Q}$;

- $S_{k,n}$: the set of *partial permutations*, i.e. maps such that

$$\pi^{-1}(1) < \pi^{-1}(2) < \cdots < \pi^{-1}(k), \qquad \pi^{-1}(k+1) < \pi^{-1}(k+2) < \cdots < \pi^{-1}(n).$$

  In other words, each map $\pi \in S_{k,n}$ is uniquely associated with a size-k subset $J \subseteq \{1, \cdots, n\}$ such that indices in J are moved to the first k positions, while indices outside of J are moved to the last $n-k$ positions.

1.2 Signatures from Code Equivalence

LESS is based on a Sigma protocol that proves the knowledge of an isometry $\mu \in M_n$ between a pair of linear codes $\mathscr{C}, \mathscr{C}' \subseteq \mathbb{F}_q^n$ with dimension k. The isometry μ, held by the prover, is the witness, while the verifier only knows the pair $(\mathscr{C}, \mathscr{C}')$. The prover samples uniformly at random an isometry $\widetilde{\mu}$ from M_n and commits to $\widetilde{\mathscr{C}} = \widetilde{\mu}(\mathscr{C})$; then, they either reveal $\widetilde{\mu}$ so that the verifier can verify how $\widetilde{\mathscr{C}}$ is obtained from $\mathscr{C}$, or $\mu' = \widetilde{\mu} \circ \mu^{-1}$ so that the verifier can check how $\widetilde{\mathscr{C}}$ is obtained from $\mathscr{C}'$. A visual representation of the proof of knowledge is given in Figure 1.

$$\begin{tikzcd}
\mathscr{C} \arrow[r, "\widetilde{\mu} \xleftarrow{\$} M_n"] \arrow[rr, dashed, bend right, "\mu"'] & \widetilde{\mathscr{C}} & \mathscr{C}' \arrow[l, "\widetilde{\mu} \circ \mu^{-1}"']
\end{tikzcd}$$

Figure 1: The LESS proof of knowledge.

In LESS, linear codes are represented through their generator matrices, in RREF. A procedural description of the LESS proof of knowledge is given in Figure 2. This Sigma protocol satisfies all the required properties, namely *completeness*, *zero-knowledge* and *soundness*, as shown in [BMPS20]. In particular, it is 2-special sound, with soundness error 1/2.

Public Data System parameters $q, n, k \in \mathbb{N}$ and hash function $\textsc{Hash}$.
Private Key $\mu \in M_n$.
Public Key $(\mathbf{G}, \mathbf{G}')$ with $\mathbf{G} \xleftarrow{\$} \mathbb{F}_q^{k \times n}$ and $\mathbf{G}' = \mathrm{RREF}\big(\mu(\mathbf{G})\big)$.

PROVER		VERIFIER
$\widetilde{\mu} \xleftarrow{\$} M_n$, $\widetilde{\mathbf{G}} \leftarrow \mathrm{RREF}\big(\widetilde{\mu}(\mathbf{G})\big)$		
$\mathsf{cmt} \leftarrow \textsc{Hash}(\widetilde{\mathbf{G}})$	$\xrightarrow{\mathsf{cmt}}$	
	$\xleftarrow{\mathsf{ch}}$	$\mathsf{ch} \xleftarrow{\$} \{0, 1\}$
If $\mathsf{ch} = 0$: $\mathsf{rsp} \leftarrow \widetilde{\mu}$.		
$\mathsf{rsp} \leftarrow \widetilde{\mu}$.		
Else: $\mathsf{rsp} \leftarrow \widetilde{\mu} \circ \mu^{-1}$.		
$\mathsf{rsp} \leftarrow \widetilde{\mu} \circ \mu^{-1}$.	$\xrightarrow{\mathsf{rsp}}$	**If** $\mathsf{ch} = 0$:
		$\hat{\mathbf{G}} \leftarrow \mathrm{RREF}\big(\mathsf{rsp}(\mathbf{G})\big)$
		Else:
		$\hat{\mathbf{G}} \leftarrow \mathrm{RREF}\big(\mathsf{rsp}(\mathbf{G}')\big)$
		Accept if $\textsc{Hash}(\hat{\mathbf{G}}) = \mathsf{cmt}$

Figure 2: The LESS Sigma protocol.

2 Design Rationale

LESS in a nutshell. LESS stands for **L**inear **E**quivalence **S**ignature **S**cheme. It is constructed by applying the Fiat-Shamir transformation [FS86] to a zero-knowledge identification scheme. The latter is obtained, essentially, by iterating the one-round Sigma protocol of Figure 2. The final protocol includes several modifications, which we will describe individually in Section 3. These do not impact security nor the goal of the protocol, affecting instead only its modus operandi.

What is the security basis 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 in [DFMS19, LZ19]. For the case of LESS, security of the underlying zero-knowledge scheme reduces to the hardness of the Linear Equivalence Problem (LEP).

What is the Linear Equivalence Problem? The *code equivalence problem* in the Hamming metric is a traditional problem from coding theory, which asks to determine whether two linear codes are *equivalent*. Equivalence here refers to the existence of an isometry connecting the two codes. We call Linear Equivalence Problem (LEP) the case where such an isometry is a monomial map, as defined in Section 1. In the case of LESS, we consider the *computational* version of the problem: given two (systematic) generator matrices $\mathbf{G}, \mathbf{G}' \in \mathbb{F}_q^{k \times n}$, find a monomial $\mu \in M_n$ such that $\mathbf{G}' = \mathrm{RREF}(\mu(\mathbf{G}))$. We discuss the security of LEP in Section 5.

How is LESS designed? The structure of the LESS protocol features an inherent flexibility in the choice of parameters, lending itself naturally to various use cases. As reported in Section 6, we can optimize performance with different criteria in mind. Thanks to various optimizations, and in particular via the use of *canonical forms*, the size of rsp can be minimized, reaching close to the optimal theoretical lower bound of 2λ bits. As a result, LESS is naturally prone to compact signatures. This leads to two main design avenues. A first option is to keep the smallest possible public key, accepting a signature size between 2 and 3 KB. Intuitively, this 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. Alternatively, one can push the design by accepting tradeoffs, in order to obtain an even smaller signature. In this case, the size of the signature drops to a size between 1 and 2 KB, which is smaller than most competitors.

3 Protocol Description (2.B.1)

The LESS protocol 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)

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

More details on how to select parameters will be given in Sections 5 and 6 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 2 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 4.

Reducing Witness Size via Canonical Forms. The first major building block consists of applying the framework introduced in [CPS23], which generalizes the work of [PS23] and significantly reduces the communication cost. The framework generalizes the proof of equivalence in the following sense. Instead of proving equivalence by showing that the two codes admit the same generator matrix (as done in Figure 2), it is shown that both codes lie in the same *equivalence class*. Then, by carefully defining the notion of equivalence classes, the latter check remains computationally efficient.

Let $F \subseteq M_n$ be a subgroup of the monomial group such that any $\varphi \in F$ is associated to a unique pair of isometries $(\varphi_r, \varphi_c) \in M_k \times M_{n-k}$ where φ_r acts only on the first k coordinates and φ_c acts only on the last $n-k$ coordinates. It is easy to see that each isometry in M_n can be obtained from an isometry in F via a partial permutation from $S_{k,n}$, i.e.

$$\forall \mu \in M_n, \quad \exists!(\pi, \varphi) \in S_{k,n} \times F : \quad \mu = \varphi \circ \pi. \tag{1}$$

Verifying that two codes are mapped by an isometry from F is computationally easy, thanks to the use of a *canonical form*. This is a function CF that, on input a matrix $\mathbf{A} \in \mathbb{F}_q^{k \times (n-k)}$, returns either a failure $\perp$ or a matrix $\mathbf{A}' = \mathbf{Q}_r \cdot \mathbf{A} \cdot \mathbf{Q}_c$, where $\mathbf{Q}_r$ and $\mathbf{Q}_c$ are the $k \times k$ and $(n-k) \times (n-k)$ monomial matrices representing φ_r and φ_c, respectively. The function verifies the following property:

$$\mathrm{CF}(\mathbf{A}) = \mathrm{CF}(\mathbf{Q}_r \cdot \mathbf{A} \cdot \mathbf{Q}_c), \quad \forall (\mathbf{Q}_r, \mathbf{Q}_c).$$

Using canonical forms, one can obtain *generator matrices in canonical form* by first bringing the matrix into systematic form and then computing the canonical form of the non-systematic part, i.e.:

$$\mathbf{G} \xrightarrow{\text{Systematic Form}} \big(\mathbf{I}_k \mid \mathbf{A}\big) \xrightarrow{\text{Canonical Form}} \big(\mathbf{I}_k \mid \mathrm{CF}(\mathbf{A})\big).$$

This means that now, the prover is able to use $\mathrm{CF}(\mathbf{A})$ as a commitment in the following way. As before, the prover knows an isometry μ between $\mathscr{C}$ and $\mathscr{C}'$. He samples uniformly at random $\widetilde{\mu}$ from M_n, obtains a systematic generator matrix $\big(\mathbf{I}_k \mid \mathbf{A}\big)$ for $\widetilde{\mathscr{C}} = \widetilde{\mu}(\mathscr{C})$ via RREF*, applies CF and finally commits to $\mathrm{CF}(\mathbf{A})$. As a first challenge, the prover is again asked to reveal $\widetilde{\mu}$, and then the verifier is able to reproduce all steps. Alternatively, the prover will reveal a map χ which is a representative for the coset $F\mu^*$, where $\mu^* = \pi \circ \widetilde{\mu} \circ \mu^{-1}$; in other words, we have

$$\chi = \varphi \circ \underbrace{\pi \circ \widetilde{\mu} \circ \mu^{-1}}_{\mu^*} = \varphi \circ \mu^*, \quad \varphi \in F.$$

The verifier can then apply χ and compute RREF, yielding a systematic generator $\big(\mathbf{I}_k \mid \hat{\mathbf{A}}\big)$, and check the commitment via CF.

The diagram from Figure 1 gets modified as follows.

$$\begin{tikzcd}
\mathscr{C} \arrow[r, "\widetilde{\mu} \overset{\$}{\leftarrow} M_n"] \arrow[rrrrrr, dashed, bend right, "\mu"'] & \widetilde{\mathscr{C}} \arrow[r, "\mathrm{RREF}^*"] & \big(\mathbf{I}_k \mid \mathbf{A}\big) \arrow[r, "\mathrm{CF}"] & \mathrm{CF}(\mathbf{A}) & \big(\mathbf{I}_k \mid \hat{\mathbf{A}}\big) \arrow[l, "\mathrm{CF}"'] & \hat{\mathscr{C}} \arrow[l, "\mathrm{RREF}"'] & \mathscr{C}' \arrow[l, "\chi"']
\end{tikzcd}$$

Figure 3: The LESS proof of knowledge using canonical forms.

Crucially, among all transformations in the coset, we choose one having a special structure, allowing for a convenient encoding. For the remainder of this document, we indicate by CosetRep the function that, on input a monomial map $\mu^* \in M_n$, returns the size-k set J encoding χ, which is the chosen representative for the coset $F\mu^*$. For completeness, the modified Sigma protocol, based on canonical forms, is shown in Figure 4. Note that the protocol preserves special soundness but now the underlying hard problem is, formally, slightly different from LEP. Still, as shown in [CPS23], the two problems are equivalent when random codes are concerned; more details are given in Section 5.

Public Data	System parameters $q, n, k \in \mathbb{N}$, hash function $\textsc{Hash}$, canonical form function CF.
Private Key	$\mu \in M_n$.
Public Key	$(\mathbf{G}, \mathbf{G}')$ with $\mathbf{G} \xleftarrow{\$} \mathbb{F}_q^{k \times n}$ and $\mathbf{G}' = \mathrm{RREF}\big(\mu(\mathbf{G})\big)$.

PROVER		VERIFIER
do:		
$\quad \widetilde{\mu} \xleftarrow{\$} M_n$		
$\quad \Big((\mathbf{I}_k \mid \mathbf{A}), \pi\Big) \leftarrow \mathrm{RREF}^*\big(\widetilde{\mu}(\mathbf{G})\big)$		
while $\mathrm{CF}(\mathbf{A}) = \perp$		
$\mathsf{cmt} \leftarrow \textsc{Hash}\big(\mathrm{CF}(\mathbf{A})\big)$	$\xrightarrow{\mathsf{cmt}}$	
	$\xleftarrow{\mathsf{ch}}$	$\mathsf{ch} \xleftarrow{\$} \{0, 1\}$
If $\mathsf{ch} = 0$:		
$\quad \mathsf{rsp} \leftarrow \widetilde{\mu}$		
Else:		
$\quad \mathsf{rsp} \leftarrow J = \textsc{CosetRep}(\pi \circ \widetilde{\mu} \circ \mu^{-1})$	$\xrightarrow{\mathsf{rsp}}$	**If** $\mathsf{ch} = 0$:
		$\quad \big(\mathbf{I}_k \mid \mathbf{A}\big) \leftarrow \mathrm{RREF}^*\big(\mathsf{rsp}(\mathbf{G})\big)$
		Else:
		$\quad \mathbf{A} \leftarrow \mathbf{G}'^{-1}_{\mathsf{rsp}} \cdot \mathbf{G}'_{\{1,\cdots,n\} \setminus \mathsf{rsp}}$
		Accept if $\textsc{Hash}\big(\mathrm{CF}(\mathbf{A})\big) = \mathsf{cmt}$

Figure 4: Description of the Sigma protocol based on canonical forms.

Iterating. The protocol in Figure 4 still only provides soundness 1/2, meaning that a malicious party can successfully impersonate an honest prover half of the time. To achieve the authentication level required, 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 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. To implement the transform, we follow the recommendations provided in [Cha22], i.e., add a salt and also use the round index when computing commitments/calling a PRNG.

Multiple Public Keys. It is possible to greatly reduce the soundness error by expanding the public key [FG19]. In our case, this means using, as the public key, a tuple $(\mathbf{G}_0, \mathbf{G}_1, \ldots, \mathbf{G}_{s-1})$, where $\mathbf{G}_i = \mathrm{RREF}\big(\mu_i(\mathbf{G}_0)\big)$ for $1 \leq i < s$, corresponding to as many secret monomial maps $\mu_0, \cdots, \mu_{s-1}$. The protocol in Figure 1 is just a special case, for which $s = 2$. The verifier's challenge then asks to complete the diagram in Figure 1 starting from a specific key $\mathbf{G}_j$, to which the prover responds with the matching isometry $\widetilde{\mu} \circ \mu_j^{-1}$. Since the challenge space is larger, fewer rounds are necessary to achieve the same soundness, 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 (PRNG). This includes the matrix $\mathbf{G}_0$ (which is included in the public key), all elements of the private key $\mathsf{sk}[i] = \mu_i$ for $1 \leq i < s$ (as per optimization above), and all maps $\widetilde{\mu}^{(i)}$, corresponding to the commitments in each of the rounds $0 \leq i < t$ in which the Sigma protocol is executed. For all such components, the storage requirements can be minimized by storing the corresponding PRNG seeds, at the cost of performing the PRNG 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 maps, 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 $\mathbf{A} \xleftarrow{\$} \mathbb{F}_q^{k\times(n-k)}$ and set $\mathbf{G}_0 = (\mathbf{I}_k \mid \mathbf{A})$.

Fixed-Weight Challenges. When the isometry queried is the one between $\mathbf{G}_0$ and $\widetilde{\mathbf{G}}$, the response consists of the monomial map $\widetilde{\mu}$, which is generated uniformly at random. As per the paragraph above, it is then sufficient to transmit only the seed used to generate it. Since, for the case of challenge equal to 1, this is not possible, the challenge corresponding to 0 is lighter and the communication cost can be improved by adjusting the probability distribution of the challenge string [BKP20]. This allows to reduce the signature size at the cost of increasing the number of rounds, by fixing the number of non-zero challenges over all rounds in the signing process. As an added benefit, this makes the signature size constant.

When considering fixed-weight challenges, we require a slightly higher amount of repetitions to achieve a soundness error of $2^{-\lambda}$, namely, the amount of rounds t must satisfy

$$\binom{t}{w}(s-1)^w \geq 2^\lambda,$$

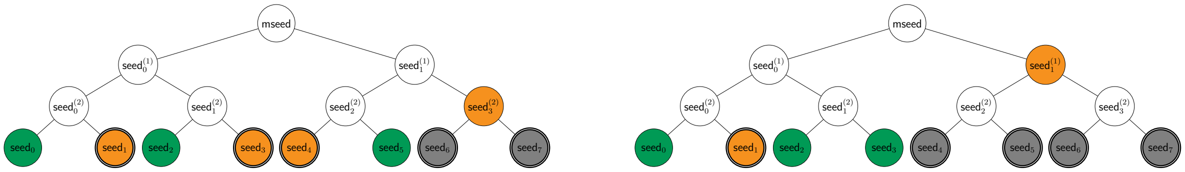
where $w < t$ is the number of non-zero challenges and s the amount of generator matrices in the public key.

GGM Trees. To efficiently represent the seeds used in the signing and verification algorithms of LESS, we use a binary tree structure known as "GGM tree" [GGM86]. The goal is to obtain t seeds $\mathsf{seed}_0, \cdots, \mathsf{seed}_{t-1}$, starting from a master seed $\mathsf{mseed} \in \{0,1\}^\lambda$. For every node, we generate its two children by feeding a PRNG with the node value and parse the PRNG output (with length 2λ) as its two children. This procedure is iterated until we get a layer (the final one) having t seeds, each being a binary string with length λ.

Note that, in order to protect from collision attacks on the commitments (e.g. [Cha22]), we include additional randomness, actually building the tree starting from a root of the form mseed||salt, where salt is a binary string of length 2λ. This salt is used in every subsequent call to the PRNG (i.e. every time a seed gets expanded into two). Furthermore, to improve protection against collision attacks, we additionally feed the PRNG using the indices which specify the location of the current leaf.

Now, when t is a power of 2, we end up with a unique, complete binary tree. When t is not a power of 2, instead, we employ the construction proposed in [BPS+23], according to which the resulting tree is obtained by "merging" m full subtrees, with m being the Hamming weight of t when converted into a binary string.

When one needs to communicate all but a subset of the t seeds, say, for instance, all except those indexed by a set $U \subset \{0, \cdots, t-1\}$ of size w, it is possible to exploit the tree structure to reduce the number of bits transmitted. The idea behind the improved 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 5 for an example of this procedure, for the case where t is a power of 2.



(a) Worst case: 4 seeds are needed

(b) Lucky case: only 2 seeds are needed

Figure 5: Example of binary seed tree for $t = 8$ and $w = 3$. 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 PRNG. 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.

An upper bound for the nodes which must be communicated is

$$\min\left\{\lfloor w \log_2(t/w) + m - 1 \rfloor \ ; \ t - w\right\}.$$

On average, especially when w is small, the number of nodes which actually get communicated may be significantly lower than the bound.

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

3.2 High-Level Description

Putting together the above building blocks, we now provide a complete high-level description of how LESS signatures are obtained and verified. For the sake of clarity, in Table 2 we list the main functions employed by our algorithms. In all algorithms, the parameters $n, k, q, t, w, s, \lambda \in \mathbb{N}$ are considered public data.

Name	**Input**	**Output**
RREF	$k \times n$ matrix	$k \times n$ matrix in RREF
RREF*	$k \times n$ matrix	$k \times n$ matrix in systematic form and permutation $\pi \in S_{k,n}$ that brings RREF to systematic form
SampleGenerator	string of arbitrary length	$k \times n$ in RREF
SampleMonomial	string of arbitrary length	length-n monomial map
SampleChallenge	string of arbitrary length	length-t string with values in $\{0, 1, \cdots, s - 1\}$ and Hamming weight w
CF	$k \times (n - k)$ matrix	$k \times (n - k)$ matrix in canonical form
CosetRep	length-n monomial map	coset representative, encoded as a size-k subset of $\{1, \cdots, n\}$
UnpackCosetRep	coset representative, encoded as a size-k subset of $\{1, \cdots, n\}$	length-n monomial map
BuildGGM	seed $\mathsf{seed}_{\mathsf{tree}} \in \{0; 1\}^{\lambda}$, salt $\mathsf{salt} \in \{0; 1\}^{2\lambda}$	seeds $\{\mathsf{seed}^{(1)}, \cdots, \mathsf{seed}^{(t)}\}$ which constitute the bottom layer of a GGM tree with root $\mathsf{seed}_{\mathsf{tree}} \vert\vert \mathsf{salt}$
GGMPath	seed $\mathsf{seed}_{\mathsf{tree}} \in \{0; 1\}^{\lambda}$, salt $\mathsf{salt} \in \{0; 1\}^{2\lambda}$, subset $U \subseteq \{1, \cdots, t\}$ of size w	subset of nodes path to reconstruct all seeds in bottom layer which are not indexed by U, in a GGM tree with root $\mathsf{seed}_{\mathsf{tree}} \vert\vert \mathsf{salt}$
RebuildGGM	nodes path, salt $\mathsf{salt} \in \{0; 1\}^{2\lambda}$, subset $U \subseteq \{1, \cdots, t\}$ of size w	seeds $\mathsf{seed}^{(i)}$ such that $i \notin U$, for a GGM tree salted with salt

Table 2: Main functions employed in LESS.

For ease of readability, we will leave out of the high-level description of certain auxiliary functions; these include "basic" functionalities such as hashing, or expanding a seed into multiple ones, as well as some more "technical" functions related, for example, to element compression, integrity checks, etc. Regarding the former, we will simply specify, in Section 4.1, our choice of standard functions and procedures; the latter, instead, will be reported in a dedicated section, Section 4.2, for completeness. We also detail, in Section 4.3, various implementation optimizations that we added to improve our code.

Algorithms 1, 2 and 3 show, respectively, the key generation, signature generation, and signature verification algorithms.

Algorithm 1: LESS Key Generation

```
Input: Private seed $\mathsf{seed}_{\mathsf{sk}}$
Output: Private key $\mathsf{sk}$ and public key $\mathsf{pk}$.
/* Generate the seed for the public key */
$\mathsf{state}_{\mathsf{sk}} \leftarrow \mathrm{XOF.INIT}(\mathsf{seed}_{\mathsf{sk}})$
$\mathsf{seed}_{\mathsf{pk}} \leftarrow \mathrm{XOF}(\mathsf{state}_{\mathsf{sk}})$

/* Sample starting generator matrix */
$\mathbf{G}_0 \leftarrow \mathrm{SampleGenerator}(\mathsf{seed}_{\mathsf{pk}})$

/* Sample seeds for private key */
$(\mathsf{seed}_1, \ldots, \mathsf{seed}_{s-1}) \leftarrow \mathrm{XOF}(\mathsf{state}_{\mathsf{sk}})$

/* Generate public key matrices */
for $i = 1, \cdots, s-1$ do
    $\tau_i \leftarrow \mathrm{SampleMonomial}(\mathsf{seed}_i)$ // Sample monomial map from seed
    $\mu_i \leftarrow \tau_i^{-1}$ // Compute inverse
    $\mathbf{G}_i \leftarrow \mathrm{RREF}(\mu_i(\mathbf{G}_0))$ // $i$-th component of public key

/* Set keypair */
$\mathsf{sk} \leftarrow \mathsf{seed}_{\mathsf{sk}}$
$\mathsf{pk} \leftarrow (\mathsf{seed}_{\mathsf{pk}}, \mathbf{G}_1, \cdots, \mathbf{G}_{s-1})$
return $(\mathsf{sk}, \mathsf{pk})$
```

It is worth noting that, as explained in [CPS23], the computation of canonical forms can incur a small probability of failure. If this happens when the function is called (e.g. line 15 of Algorithm 2), we simply increment the seed corresponding to the current round, $\mathsf{seed}^{(i)} + 1$, and restart the loop iteration.

Additionally, we point out that the signature resulting from Algorithm 2 does not have constant size, since the number of nodes in path depends on the support U of the particular challenge vector. Indeed, the binary length of path is always upper bounded by $\lambda \cdot \{w \log_2(t/w) + m - 1\ ;\ t - w\}$, which can be seen as a worst-case size; however, with high probability, the size of path is smaller than the bound. If one wants signatures of constant size, one can pad path with zeros until the worst-case size is reached. If one accepts, instead, signatures of variable size, then the signature size can be further reduced. This, however, requires being able to parse the signature in order to correctly locate the binary string encoding path. We deal with this by appending, at the end of the signature, one extra byte encoding the number of seeds in path. Since all the other terms in the signature have constant size, the location of path can be easily recovered.

Algorithm 2: LESS Sign

```
Input: Private key $\mathsf{sk} = \mathsf{seed_{sk}}$ and message $\mathsf{msg}$.
Output: Signature $\mathsf{sgn}$ on message $\mathsf{msg}$.
/* Generate the seed for the public key */
$\mathsf{state_{sk}} \leftarrow \mathrm{XOF.INIT}(\mathsf{seed_{sk}})$
$\mathsf{seed_{pk}} \leftarrow \mathrm{XOF}(\mathsf{state_{sk}})$

/* Expand starting generator matrix and private key */
$\mathbf{G}_0 \leftarrow \textsc{SampleGenerator}(\mathsf{seed_{pk}})$
$(\mathsf{seed}_1, \ldots, \mathsf{seed}_{s-1}) \leftarrow \mathrm{XOF}(\mathsf{state_{sk}})$
for $i = 1, \cdots, s-1$ do
    $\tau_i \leftarrow \textsc{SampleMonomial}(\mathsf{seed}_i)$// Sample monomial map from seed

/* Sample salt and seeds for the $t$ rounds */
$\mathsf{salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$

/* Take $t$ seeds as the bottom layer of a GMM tree */
$\mathsf{seed_{tree}} \leftarrow \mathrm{XOF}(\mathsf{state_{sk}})$
$\{\mathsf{seed}^{(1)}, \ldots, \mathsf{seed}^{(t)}\} \leftarrow \textsc{BuildGGM}(\mathsf{seed_{tree}}, \mathsf{salt})$

/* Sample seed for Blinding */
$\mathsf{state_{blind}} \leftarrow \mathrm{XOF.INIT}(\mathrm{XOF}(\mathsf{state_{sk}}))$

/* Generate commitments: for round $i$, sample ephemeral monomial map $\widetilde{\mu}^{(i)}$ using $\mathsf{seed}^{(i)}||\mathsf{salt}||\mathsf{msg}||i$ */
for $i = 1, \cdots, t$ do
    $\widetilde{\mu}^{(i)} \leftarrow \textsc{SampleMonomial}(\mathsf{seed}^{(i)}||\mathsf{salt}||i)$// Ephemeral monomial map for round $i$
    $\left(\left(\mathbf{I}_k \mid \mathbf{A}^{(i)}\right) \;,\; \pi^{(i)}\right) \leftarrow \mathrm{RREF}^*\left(\widetilde{\mu}^{(i)}(\mathbf{G}_0)\right)$// Apply $\widetilde{\mu}^{(i)}$ to $\mathbf{G}_0$, compute systematic form
    $\mathbf{A}^{(i)} \leftarrow \textsc{Blind}(\mathsf{state_{blind}}, \mathbf{A}^{(i)})$
    $\mathbf{B}^{(i)} \leftarrow \mathrm{CF}(\mathbf{A}^{(i)})$// Canonical form for round $i$

/* Generate challenge: length-$t$ vector with all zeros apart from $w$ values with values in $\{1, \cdots, s-1\}$ */
$\mathsf{cmt} \leftarrow \textsc{Hash}(\mathbf{B}^{(1)}|| \cdots ||\mathbf{B}^{(t)}||\mathsf{salt}||\mathsf{msg})$
$\left(b^{(1)}, \cdots, b^{(t)}\right) \leftarrow \textsc{SampleChallenge}(\mathsf{cmt})$
$U \leftarrow$ support of $\left(b^{(1)}, \cdots, b^{(t)}\right)$// Set with indices of non zero values

/* Get path, in the GGM tree, to recompute $\{\mathsf{seed}^{(i)}\}_{i \notin U}$ */
$\mathsf{path} \leftarrow \textsc{GGMPath}(\mathsf{seed_{tree}}, \mathsf{salt}, U)$

/* For each round $i$ such that $b^{(i)} \neq 0$, provide coset representative for $\pi^{(i)} \circ \widetilde{\mu}^{(i)} \circ \tau_{b^{(i)}}$ */
for $i \in U$ do
    $\mathsf{rsp}^{(i)} \leftarrow \textsc{CosetRep}\left(\pi^{(i)} \circ \widetilde{\mu}^{(i)} \circ \tau_{b^{(i)}}\right)$// Coset representative

$\mathsf{sgn} \leftarrow \left\{\mathsf{cmt},\ \mathsf{salt},\ \mathsf{path},\ \{\mathsf{rsp}^{(i)}\}_{i \in U}\right\}$
return $\mathsf{sgn}$
```

Algorithm 3: LESS Verify

```
Input: Public key $(\mathsf{seed}_{\mathsf{pk}}, \mathbf{G}_1, \cdots, \mathbf{G}_{s-1})$, message $\mathsf{msg}$ and signature $\mathsf{sgn}$.
Output: $\mathsf{true}$ (accept) or $\mathsf{false}$ (reject).

/* Expand starting generator matrix */
$\mathbf{G}_0 \leftarrow \textsc{SampleGenerator}(\mathsf{seed}_{\mathsf{pk}})$

/* Regenerate challenge */
Read $\mathsf{cmt}$ and $\mathsf{salt}$ from $\mathsf{sgn}$
$\left(b^{(1)}, \cdots, b^{(t)}\right) \leftarrow \textsc{SampleChallenge}(\mathsf{cmt})$
$U \leftarrow$ support of $\left(b^{(1)}, \cdots, b^{(t)}\right)$// Set with indices of non zero values

/* For all rounds with null challenge value, generate the seeds from GGM tree */
Read $\mathsf{path}$ from $\mathsf{sgn}$
For $i \notin U$, $\mathsf{seed}^{(i)} \leftarrow \textsc{RebuildGGM}(\mathsf{path}, U, \mathsf{salt})$

/* Read responses from signature */
For $i \in U$, read $\mathsf{rsp}^{(i)}$ from $\mathsf{sgn}$

/* For round $i$, obtain $\mathbf{A}^{(i)}$ by taking non-systematic part of some RREF (starting from $\mathbf{G}_{b^{(i)}}$), use it as input for
   CF and recompute $\mathsf{cmt}^{(i)}$ */
for $i = 1, \cdots, t$ do
    if $b^{(i)} = 0$ then
        /* Sample ephemeral monomial map, apply it to $\mathbf{G}_0$ and do RREF */
        $\widetilde{\mu}^{(i)} \leftarrow \textsc{SampleMonomial}(\mathsf{seed}^{(i)}||\mathsf{salt}||i)$// Ephemeral monomial map for round $i$
        $\left(\left(\mathbf{I}_k \mid \mathbf{A}^{(i)}\right)\ ,\ \pi^{(i)}\right) \leftarrow \mathrm{RREF}^*\left(\widetilde{\mu}^{(i)}(\mathbf{G}_0)\right)$// Apply $\widetilde{\mu}^{(i)}$ to $\mathbf{G}_0$, compute systematic form
    else
        if $\textsc{CheckCosetRep}(\mathsf{rsp}^{(i)}) = \mathsf{false}$ then
            return $\mathsf{false}$
        /* Compute RREF for $\mathbf{G}_{b^{(i)}}$ by pivoting columns in $\mathsf{rsp}^{(i)}$ */
        Set $\mathbf{G}'_1 :=$ columns of $\mathbf{G}_{b^{(i)}}$ indexed by $\mathsf{rsp}^{(i)}$// Columns that form the information set
        Set $\mathbf{G}'_2 :=$ columns of $\mathbf{G}_{b^{(i)}}$ not indexed by $\mathsf{rsp}^{(i)}$// Columns outside of information set
        $\mathbf{A}^{(i)} \leftarrow \mathbf{G}'^{-1}_1 \cdot \mathbf{G}'_2$// Non-systematic part after RREF
    $\mathbf{B}^{(i)} \leftarrow \mathrm{CF}(\mathbf{A}^{(i)})$// Canonical form for round $i$

/* Accept if $\mathsf{cmt}$ is correctly recomputed, reject otherwise */
$\mathsf{cmt}' \leftarrow \textsc{Hash}(\mathbf{B}^{(1)}||\cdots||\mathbf{B}^{(t)}||\mathsf{salt}||\mathsf{msg})$
if $\mathsf{cmt} = \mathsf{cmt}'$ then
    return $\mathsf{true}$
else
    return $\mathsf{false}$
```

4 Procedural Description

In this section, we describe in full detail how all the required functionalities for LESS are implemented.

4.1 Main Functions

We write here the main functions, i.e., those listed in Table 2. To begin with, we clarify our choice of cryptographic primitives, to obtain a practical realization of a PRNG, as well as the hash function Hash. For the former, we employ the SHAKE family of functions with an appropriate security level, namely SHAKE-128 for Category 1 and SHAKE-256 for other categories. On the other hand, we employ SHA-3 as our implementation of the function Hash, selecting a digest of size 2λ. We will recap these choices in Table 3.

RREF. The RREF algorithm is a straightforward instantiation of the Gaussian elimination. It systematically transforms a given matrix into row echelon form through a series of elementary row operations: row swapping, scalar multiplication of rows, and adding or subtracting multiples of one row to another. This transformation is done by first finding a suitable pivot element. This is then used in subsequent row operations to eliminate the other rows.

Algorithm 4: RREF

```
   Input: $\mathbf{G} \in \mathbb{F}_q^{k\times n}$
   Output: $\mathbf{G} \in \mathbb{F}_q^{k\times n}$ in RREF
   Data:
1  for row_to_reduce $\in [k]$ do
2      $i, j \leftarrow 0, 0$
3      for $j \in$ [row_to_reduce, $n$] do
4          for $i \in$ [row_to_reduce, $k$] do
5              if $\mathbf{G}[i, j] > 0$ then
6                  go to 7
       /* Swap the rows at row_to_reduce with i */
7      $\mathbf{G}$[row_to_reduce :], $\mathbf{G}[i :] \leftarrow \mathbf{G}[i :]$, $\mathbf{G}$[row_to_reduce :]
       /* Reduce the pivot row */
8      $\mathbf{G}$[row_to_reduce :] $\leftarrow \mathbf{G}$[row_to_reduce, $j]^{-1} \cdot \mathbf{G}$[row_to_reduce :]
       /* Reduce the rest */
9      for row $\in [k] \setminus$ row_to_reduce do
10         $\mathbf{G}$[row :] $\leftarrow \mathbf{G}$[row :] $- \mathbf{G}$[row, $j$] $\cdot \mathbf{G}$[row_to_reduce :]
11 return $\mathbf{G}$
```

The only difference between Algorithm 4 and Algorithm 5 is that the latter keeps track of the pivot columns and returns additionally a permutation shifting the pivot columns to the left.

Note that the descriptions of Algorithm 4 as well as Algorithm 5 given here do not run in constant time. For a constant-time implementation, it is additionally important to mention that only the pivot row needs be kept secret as the pivot column is published.

Algorithm 5: RREF*

```
Input: $\mathbf{G} \in \mathbb{F}_q^{k \times n}$
Output: $\mathbf{G} \in \mathbb{F}_q^{k \times n}$ in RREF, $\pi \in S_n$
1  for row_to_reduce $\in [k]$ do
2      $i, j \leftarrow 0, 0$
       /* Initialize the permutation with the identity */
3      $\pi \leftarrow$ id
4      for $j \in$ [row_to_reduce, $n$] do
5          for $i \in$[row_to_reduce, $k$] do
6              if $\mathbf{G}[i, j] > 0$ then
                   /* Keep track of the pivot columns */
7                  $\pi$[row_to_reduce], $\pi[j] \leftarrow \pi[j]$, $\pi$[row_to_reduce]
8                  go to 9
       /* swap the rows at row_to_reduce with i */
9      $\mathbf{G}$[row_to_reduce :], $\mathbf{G}[i :] \leftarrow \mathbf{G}[i :]$, $\mathbf{G}$[row_to_reduce :]
       /* Reduce the pivot row */
10     $\mathbf{G}$[row_to_reduce :] $\leftarrow \mathbf{G}$[row_to_reduce, j]$^{-1} \cdot \mathbf{G}$[row_to_reduce :]
       /* Reduce the rest */
11     for row $\in [k] \setminus$ row_to_reduce do
12         $\mathbf{G}$[row :] $\leftarrow \mathbf{G}$[row :] $- \mathbf{G}[\text{row}, j] \cdot \mathbf{G}$[row_to_reduce :]
13 return $\mathbf{G}, \pi$
```

Sampling Generator Matrices. Algorithm 6 describes the process of expanding a seed into a generator matrix in RREF. This is accomplished by initializing the first k columns of $\mathbf{G}$ to $\mathbf{I}_k$, and then sampling random coefficients in the range $[q]$ row-wise. The samples are again generated using bit-wise parsing of the XOF digest within 64-bit blocks. Note that this algorithm is only used once in each of key generation, signing, and verification to sample the matrix $\mathbf{G}_0$.

Algorithm 6: SampleGenerator

```
Input: seed $\in \{0, 1\}^\lambda$
Output: $\mathbf{G} \in \mathbb{F}_q^{k \times n}$ in RREF
$\mathbf{G} \leftarrow (\mathbf{I}_{n-k} | \mathbf{0}^{n-k})$
state = XOF.init(seed)
for $i \in [k]$ do
    for $j \in [k, n]$ do
        $\mathbf{G}[i, j] \leftarrow$ SampleUniformNumber(state, $q$)
return $\mathbf{G}$
```

Sampling Monomial Maps. The approach used for the expansion of a monomial map from a seed is described in Algorithm 7. This operation requires sampling within two different ranges: $[1, q-1]$ and $[0, n-1]$. For the former, we reject the 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 7: SampleMonomial

```
Input: seed $\in \{0,1\}^\lambda$
Output: $\mu$ : a monomial matrix represented by two lists of length $n$, called $\mu.\pi$ and $\mu.u$,
        containing the permutation and coefficients.
SampleMonomial (state)
    $\mu.\pi \leftarrow [n]$
    for $i \in [n]$ do
        $\mu.u[i] \leftarrow 1 + $ SampleUniformNumber(state, $q - 1$)
        chunks of the XOF digest.
    for $i \in [n]$ do
        $x \leftarrow$ SampleUniformNumber(state, $n$)
        $\mu.\pi[i], \mu.\pi[x] \leftarrow \mu.\pi[x], \mu.\pi[i]$
    return $\mu$
state = XOF.init(seed)
return SampleMonomial(state)
```

Sampling the Challenge String. The approach used for the generation of the fixed-weight challenge is described in Algorithm 8. Similarly to Algorithm 7, 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 w 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 8: SampleChallenge

```
Input: seed $\in \{0,1\}^\lambda$
Output: ch $\in \mathbb{S}_{t,w}$
state = XOF.init(seed)
if $s \neq 2$ then
    for $i \in [w]$ do
        ch$[t - w + i] \leftarrow 1 + $ SampleUniformNumber(state, $s - 1$)
else
    ch $\leftarrow (\mathbf{0}^{t-w}\mathbf{1}^{w})$
for $i \in [t - w, t]$ do
    do
        $p \leftarrow$ XOF(state, 8)
        $p \leftarrow p \mod t$
    while $p > i$
    ch$[i]$, ch$[p] \leftarrow$ ch$[p]$, ch$[i]$
return ch
```

Implementing Canonical Forms. Algorithm 9 computes the canonical form $\mathbf{G}'$ from an input matrix $\mathbf{G}$.

Algorithm 9: CF

```
Input: $\mathbf{G} \in \mathbb{F}_q^{k \times n-k}$ non-IS of Generator Matrix
Output: $\mathbf{G}' \in \mathbb{F}_q^{k \times n-k}$ in canonical form
Data: $\mathbf{A} \in \mathbb{F}_q^{k \times (n-k)}$, $\mathbf{M} \in \mathbb{F}_q^{k \times (n-k)}$: temporary buffer
/* set $\mathbf{M}$ to the largest matrix */
$\mathbf{M} \leftarrow (q-1)^{k \times (n-k)}$
for $i \in [k]$ do
    for $j \in [k]$ do
        $\mathbf{A}[j:] \leftarrow \mathbf{G}[j:] \cdot \mathbf{G}[i:]^{-1}$
    $\mathbf{A} \leftarrow \textsc{ScaleCF}(\mathbf{A})$
    if $\mathbf{A} \prec_{\mathbb{F}_q^{k \times (n-k)}} \mathbf{M}$ then
        $\mathbf{M} \leftarrow \mathbf{A}$
return $\mathbf{M}$
```

Algorithm 10 compresses the permutation μ^* by setting those bit positions within an n bit string, which corresponds to a column, which is permuted into the information set by μ^*.

Algorithm 10: CosetRep

```
Input: Map $\mu^*$ representing the permutation values of the monomial action
Output: $\mathbf{b} \in \{0,1\}^n, \mathsf{wt}(\mathbf{b}) = k$
$\mathbf{b} \leftarrow \mathbf{0}^n$
for $i \in [k]$ do
    /* set the $\mu^*[i]$ bit in $\mathbf{b}$ */
    $\mathbf{b}[\mu^*[i]] = 1$
return $\mathbf{b}$
```

Its inverse operation, Algorithm 11, takes as input an n bit string $\mathbf{b}$ and outputs a permutation μ^*, which shuffles columns, whose corresponding bit is set, to the left. Note that to ensure the correctness of Algorithm 11 it must be that $\mathsf{wt}(\mathbf{b}) = k$, otherwise μ^* will not be a valid permutation.

Algorithm 11: UnpackCosetRep

```
Input: $\mathbf{b} \in \{0,1\}^n$, with hamming weight $k$
Output: Map $\mu^*$ representing the permutation values of the monomial action
$c_1 \leftarrow 0$
$c_2 \leftarrow k$
$\mu^* \leftarrow \mathbf{0}^n$
for $i \in [n]$ do
    if $\mathbf{b}[i] = 1$ then
        $\mu^*[c_1] \leftarrow i$
        $c_1 \leftarrow c_1 + 1$
    else
        $\mu^*[c_2] \leftarrow i$
        $c_2 \leftarrow c_2 + 1$
return $\mathbf{b}$
```

GGM Tree Implementation Strategy. The LESS signing and verification algorithms involve the hierarchical derivation of a sequence of seeds, one for each 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.

The BuildGGM procedure computes a binary tree of nodes, each containing 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 PRNG seeded with the entire binary string of its parent. The root node employs, as a first binary string, a length-λ string. The pseudo-code of the BuildGGM procedure is shown in Algorithm 12. The algorithm takes as input a root seed seed and a salt, materializes internally a binary tree of nodes and computes t round seeds $\mathsf{seed}^{(i)}, 0 \leq i \leq t-1$, from it, each associated to one leaf of the tree. In particular, the root is initialized with the root seed. Then, proceeding from top to bottom and left to right, each node is expanded into two children by appending the salt and the 16-bit index of the node to the seed of the node and feeding it into the PRNG that produces two seeds of λ bits. The 16-bit index is passed in little-endian byte order and helper arrays are used for proper indexing within the truncated tree structure.

Algorithm 12: BuildGGM

```
Input:  $\mathsf{seed}$: the $\lambda$-bit root seed from which the whole tree is generated
        $\mathsf{salt}$: a $2\lambda$-bit string, chosen uniformly at random
Output: $(\mathsf{seed}^{(0)}, \dots, \mathsf{seed}^{(t-1)})$: the $t$ round seeds
Data:   $t$: number of leaves (corresponds to the number of protocol rounds)
        $\lambda$: security parameter (a seed is $\lambda$ bits long)
        $\mathsf{npl}[\dots]$: number of nodes per level
        $\mathsf{lpl}[\dots]$: number of leaves per level
        $\mathsf{off}[\dots]$: offsets required to move between two levels in the unbalanced tree
$\mathcal{T}[0] \leftarrow \mathsf{seed}$
$\mathsf{startNode} \leftarrow 0$
for $\mathsf{level} \in [\lceil \log_2(t) \rceil]$ do
    for $i \in [\mathsf{npl}[\mathsf{level}] - \mathsf{lpl}[\mathsf{level}]]$ do
        $\mathsf{parent} \leftarrow \mathsf{startNode} + i$
        $\mathsf{leftChild} \leftarrow \mathsf{LeftChild}(\mathsf{parent}) - \mathsf{off}[\mathsf{level}]$
        $\mathsf{rightChild} \leftarrow \mathsf{leftChild} + 1$
        // expand parent seed, salt and parent index
        $\mathsf{state} = \mathrm{XOF.INIT}(\mathcal{T}[\mathsf{parent}] \mid \mathsf{salt} \mid \mathsf{parent})$
        $\mathcal{T}[\mathsf{leftChild}],\ \mathcal{T}[\mathsf{rightChild}] \leftarrow \mathrm{XOF}(\mathsf{state}, 2\lambda)$
    $\mathsf{startNode} \leftarrow \mathsf{startNode} + \mathsf{npl}[\mathsf{level}]$
// return the leaves of the tree as round seeds
return $\mathsf{Leaves}(\mathcal{T})$
```

The GGMPath procedure determines, given a seed tree, a subset of the leaves to be disclosed, 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. The pseudo-code of the GGMPath procedure is shown in Algorithm 13. The algorithm takes the challenge b to label a reference tree $\mathcal{T}'$ which indicates which nodes to pack into path. More specifically, it places the challenge bits $b^{(i)}$ on the leaves and updates the reference tree such that a parent node is labeled to be published if both of its children are to be published. Afterwards, GGMPath iterates through the tree from top to bottom and left to right and packs a node into the path if the node itself is to be revealed, while its parent is not to be revealed. In the actual reference implementation, it is not required to re-generate the full seed tree again, but it is given a pointer to the tree/leaves as constructed in BuildGGM.

Algorithm 13: GGMPath

```
Input: seed: the $\lambda$-bit root seed from which the whole tree is generated
       salt: a $2\lambda$-bit string, chosen uniformly at random
       $U$: the size-$w$ subset of $\{0, \cdots, t-1\}$ containing the indices of leaves that must
       not be revealed
Output: path: the subset of nodes that allow re-computing the leaves which are not
        indexed by $U$
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a seed is $\lambda$ bits long)
      npl[...]: number of nodes per level
      off[...]: offsets required to move between two levels in the unbalanced tree
$\mathcal{T} \leftarrow$ BuildGGM(seed, salt)
// Use the tree $\mathcal{T}'$ to indicate which nodes to reveal
// $\mathcal{T}'$: binary tree data structure with stencil matching the one of $\mathcal{T}$, and binary valued nodes. All nodes with 0
// value correspond to nodes of $\mathcal{T}$ meant to be revealed; nodes with value 1 correspond to nodes of $\mathcal{T}$ that
// must not be revealed. ComputeNodesToPublish($\cdot$) builds it knowing the sequence of indices in $U$.
$\mathcal{T}' \leftarrow$ ComputeNodesToPublish($U$)  // $\mathcal{T}'$ includes the roots of subtrees containing only nodes to be
                                                      // published and each of the roots has a parent that should not be
                                                      // published.
startNode $\leftarrow 0$, pubNodes $\leftarrow 0$, path $\leftarrow \emptyset$
for level $\in [1, \lceil \log_2(t) \rceil + 1]$ do
    for $i \in$ [npl[level]] do
        node $\leftarrow$ startNode $+ i$
        parent $\leftarrow$ Parent(node) $+$ off[level $- 1$]$/2$
        // Reveal node if it is to publish but its parent is not
        if $\mathcal{T}'$[node] $= 0$ and $\mathcal{T}'$[parent] $= 1$ then
            path[pubNodes] $\leftarrow \mathcal{T}$[node]
            pubNodes $\leftarrow$ pubNodes $+ 1$
    startNode $\leftarrow$ startNode $+$ npl[level]
return path
```

Finally, the RebuildGGM procedure receives the output of GGMPath, 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 GGMPath 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. The pseudo-code of the RebuildGGM procedure is shown by Algorithm 14. The algorithm re-generates the round-seeds $\mathsf{seed}^{(i)}$ given the path, b, and salt.

Algorithm 14: RebuildGGM

```
Input:  path: subset of nodes
        salt: a $2\lambda$-bit string, chosen uniformly at random
        $U$: the size-$w$ subset of $\{0, \cdots, t-1\}$ containing the indices of leaves that must
        not be revealed

Output: $(\mathsf{seed}^{(i)}_{i:b^{(i)}=0}$: the leaves corresponding to $b^{(i)} = 0$
Data:   $t$: number of leaves (corresponds to the number of protocol rounds)
        $\lambda$: security parameter (a seed is $\lambda$ bits long)
        npl[...]: number of nodes per level
        lpl[...]: number of leaves per level
        off[...]: offsets required to move between two levels in the unbalanced tree
// Use tree $\mathcal{T}'$ to reconstruct which nodes were revealed.
// $\mathcal{T}'$: binary tree data structure with stencil matching the one of $\mathcal{T}$ (i.e., the one built by executing BuildGGM
// algorithm), and binary valued nodes. All nodes with 0 value correspond to nodes of $\mathcal{T}$ that were meant to be
// revealed; nodes with value 1 correspond to nodes of $\mathcal{T}$ that should have not be revealed.
// ComputeNodesToPublish($\cdot$) builds it knowing the sequence of indices in $b$.
$\mathcal{T}' \leftarrow$ ComputeNodesToPublish($U$)
$\mathcal{T} \leftarrow \emptyset$
startNode $\leftarrow 0$, pubNodes $\leftarrow 0$, path $\leftarrow \emptyset$
for level $\in [1, \lceil \log_2(t) \rceil + 1]$ do
    for $i \in$ [npl[level]] do
        node $\leftarrow$ startNode $+ i$
        parent $\leftarrow$ Parent(node) $+$ off[level $- 1$]$/2$
        leftChild $\leftarrow$ LeftChild(node) $-$ off[level]
        rightChild $\leftarrow$ leftChild $+ 1$
        // If node is in path, copy it to tree
        if $\mathcal{T}'$[node] $= 0$ and $\mathcal{T}'$[parent] $= 1$ then
            $\mathcal{T}$[node] $\leftarrow$ path[pubNodes]
            pubNodes $\leftarrow$ pubNodes $+ 1$
        // Expand it if node is in the tree and not a leaf, with co-domain $\{0,1\}^\lambda \times \{0,1\}^\lambda$
        if $\mathcal{T}'$[node] $= 0$ and $i <$ npl[level] $-$ lpl[level] then
            state $=$ XOF.init($\mathcal{T}$[node] | salt | node)
            $\mathcal{T}$[leftChild], $\mathcal{T}$[rightChild] $\leftarrow$ XOF(state, $2\lambda$)
    startNode $\leftarrow$ startNode $+$ npl[level]
return Leaves($\mathcal{T}$)$[i]_{i:b^{(i)}=0}$
```

4.2 Auxiliary Functions

In our constructions, we require several accessory functions, which we report here.

Sampling Uniform Numbers. In this paragraph we describe the algorithm to sample a uniformly distributed number. The special feature of Algorithm 15 is that the modulus, to which the output needs to be reduced, can be passed as an argument. Thus, if n is a prime, this algorithm samples uniformly distributed finite field elements.

```
Algorithm 15: SampleUniformNumber
Input: state of an already initialized XOF, modulus $n$
Output: $x \in \mathbb{Z}_n$
static $c \leftarrow 0$
static $\mathbf{b} \leftarrow \mathbf{0}^{64}$
$m \leftarrow 2^{\lceil \log_2(n) \rceil}$
do
    if  $c = 0$ then
        $\mathbf{b} \leftarrow \mathrm{XOF}(\mathsf{state}, 8)$
        $c = \lfloor (64)/\log_2(m) \rfloor$
    $x \leftarrow \mathbf{b} \mod m$
    $\mathbf{b} \leftarrow \mathbf{b} \gg \log_2(m)$
    $c \leftarrow c - 1$
while $x \geq n$
return $x$
```

Additionally, note that the two variables c and $\mathbf{b}$ are static, which means that they hold their values between function calls and persist for the program's lifetime. Therefore, it is mandatory to reset those variables upon the sampling of a new seed for this algorithm, as otherwise this would lead to wrong results.

Compressing Generator Matrices. At the end of the key generation procedure, all the generator matrices that comprise the public key are compressed in order to reduce their size. This procedure, illustrated in Algorithm 16, 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 or not. Note that, for some of the parameter sets, the value of n is not divisible by 8; in this case, we pad with enough 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 17. This algorithm first parses the pivot columns encoded as a bit vector of length n. Next, the non pivot columns will be parsed and put into place. And last but not least, the pivot columns will be placed.

Checking Correctness of Coset Representatives. The LESS verification algorithm needs to make sure that the coset representatives, which are revealed as part of the signature, are of the correct form. That is, they are vectors of length n with Hamming weight k. Algorithm 18 implements this.

Algorithm 16: CompressRREF

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

$\mathbf{b} \leftarrow \mathbf{p}$
$\mathbf{b} \leftarrow \mathbf{0}^{8-(n\ mod\ 8)} || \mathbf{b}$// fill the last byte with zeros
for $i \in [k]$ do
    if $\mathbf{p}[i] \neq 1$ then
        for $j \in [n]$ do
            $\mathbf{b} \leftarrow \mathsf{LSB}_{\lceil\log_2(q)\rceil}(\mathbf{G}[i,j]) \mid\mid \mathbf{b}$
return $\mathbf{b}$
```

Algorithm 17: ExpandToRREF

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

/* extract the pivot column */
$\mathbf{p} \leftarrow \mathsf{LSB}_n(\mathbf{b})$
$\mathbf{b} \leftarrow \mathbf{b} \gg (\lceil n/8\rceil \cdot 8)$
$\mathbf{G} \leftarrow \mathbf{0}^{k\times n}$ for $i \in [k]$ do
    for $j \in [n]$ do
        if $\mathbf{p}[j] \neq 1$ then
            $\mathbf{G}[i,j] \leftarrow \mathsf{LSB}_{\lceil\log_2(q)\rceil}(\mathbf{b})$
            $\mathbf{b} \leftarrow \mathbf{b} \gg \lceil\log_2(q)\rceil$
for $i \in [k]$ do
    pivot_idx $\leftarrow 0$
    for $j \in [n]$ 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 $\mathbf{G}$
```

Algorithm 18: CheckCosetRep

```
Input: $\mathbf{b} \in \{0,1\}^n$ $n$-bits representing the compressed canonical form action.
Output: true (accept) or false (reject).

$j \leftarrow 0$
for $i \in [n]$ do
    /* count the number of ones in $\mathbf{b}$ */
    $j \leftarrow j + \mathbf{b}[i]$
if $j = k$ then
    return true
return false
```

Ordering and Sorting. Let $\textsc{Sort} : (\mathbb{F}_q)^* \to (\mathbb{F}_q)^*$ be a routine that sorts a vector of arbitrary length. Given the prime field $\mathbb{F}_q$, we assume that there is a total ordering $\leq_{\mathbb{F}_q}$. We extend this ordering to the vector space $\mathbb{F}_q^n$, defined as $\mathbf{v} \leq_{\mathbb{F}_q^n} \mathbf{v}'$ if and only if either $\mathbf{v}_i = \mathbf{v}'_i, \forall i \in [n]$ or $\exists i \in [n], \text{s.t.}\, \mathbf{v}_i \neq \mathbf{v}'_i$, and $\mathbf{v}_j = \mathbf{v}'_j, \forall j < i$. We extend this definition to a strict partial ordering $\prec_{\mathbb{F}_q^n}$ over $\mathbb{F}_q^n$. Given $\mathbf{v}, \mathbf{v}' \in \mathbb{F}_q^n$, we define $\mathbf{v} \prec_{\mathbb{F}_q^n} \mathbf{v}'$ if and only if $\mathbf{w} \leq_{\mathbb{F}_q^n} \mathbf{w}'$ and $\mathbf{w} \neq \mathbf{w}'$, where $\mathbf{w} = \textsc{Sort}(\mathbf{v}), \mathbf{w}' = \textsc{Sort}(\mathbf{v}')$. This definition is extended to matrices in a row-wise fashion.

Equipped with these definitions of ordering, we are able to define the following sorting routines. First, we define $\textsc{SortRows} : \mathbb{F}_q^{k\times(n-k)} \to \mathbb{F}_q^{k\times n-k}$, which takes as input the rows of $k \times n - k$ matrix and sorts them according to $\prec_{\mathbb{F}_q^{n-k}}$. And secondly, we define $\textsc{SortCols} : \mathbb{F}_q^{k\times n-k} \to \mathbb{F}_q^{k\times n-k}$, which takes as input the columns of $k \times n - k$ matrix and sorts them according to $\leq_{\mathbb{F}_q^k}$.

Canonical Forms Auxiliary Functions. We are now able to define the supporting algorithms for computing canonical forms. First, Algorithm 19 which scales each row of the input matrix in place. Next, Algorithm 20 sorts the rows and columns of the input matrix according to the orderings defined in the paragraph just above.

Algorithm 19: ScaleCF

```
Input: $\mathbf{G} \in \mathbb{F}_q^{k\times n-k}$ non-IS of Generator Matrix.
Output: $\mathbf{G}' \in \mathbb{F}_q^{k\times n-k}$.
for $i \in [k]$ do
    $s = \sum_{j=0}^{n-k} \mathbf{G}[i,j]$
    $s' = \sum_{j=0}^{n-k} \mathbf{G}[i,j]^{-1}$
    if $s \neq 0$ then
        $s \leftarrow s^{-1}$
    if $s = 0$ then
        $s \leftarrow s'$
    if $s = 0$ then
        return $\perp$
    $\mathbf{G}[i:] \leftarrow s \cdot \mathbf{G}[i:]$
return SortCF($\mathbf{G}$)
```

Algorithm 20: SortCF

```
Input: $\mathbf{G} \in \mathbb{F}_q^{k\times n-k}$ non-IS of Generator Matrix.
Output: $\mathbf{G}' \in \mathbb{F}_q^{k\times n-k}$.
$\mathbf{G}' \leftarrow$ SortRows($\mathbf{G}$)
return SortCols($\mathbf{G}'$)
```

4.3 Implementation Optimizations

We now describe the main implementation optimizations that can be applied to our algorithms to improve performance across all platforms.

Partially Sorting Rows within Canonical Forms. A first optimization strategy consists of only partially sorting each row, within the computation of Algorithm 20. Indeed, the algorithm needs to sort the rows of the input matrix under the ordering $\prec_{\mathbb{F}_q^{n-k}}$. This is done by first sorting the entries within each row, by normal sorting algorithm. And afterwards sort this output by $<_{\mathbb{F}_q^{n-k}}$. The core idea of the optimization is to replace the initial sorting of each row, by a histogram computation. A Histogram: $\mathbb{F}_q^{n-k} \rightarrow \mathbb{Z}^q$ in our case is the number of occurrences of each field element within a vector. Note that this is enough to be able to compare two rows, as the histogram is a compressed representation of the actual sorted vector. Additionally, this shrinks the size of the temporary buffer for Category 3 and 5 parameters, as only q elements need to be stored instead of $n-k$. See Algorithm 21 for a possible implementation.

Algorithm 21: Histogram

```
/* A possible constant time sorting implementation based on histogram computations */
Input: $\mathbf{v} \in \mathbb{F}_q^n$
Output: $\mathbf{w} \in \mathbb{Z}^q$ sorted
$\mathbf{w} \leftarrow \mathbf{0}$
for $i \in [n]$ do
    $\mathbf{w}[\mathbf{v}[i]] \leftarrow \mathbf{w}[\mathbf{v}[i]] + 1$
return $\mathbf{w}$
```

Reusing Pivots during RREF Computation. Within the Key Generation, Sign and Verify algorithms, a core operation is to compute the RREF of a matrix, via the RREF and RREF* functions. A useful observation is that these algorithms compute the RREF after applying a monomial map to a matrix $\mathbf{G}_0$ which is already in RREF. Since the application of a monomial map is equivalent to permuting and scaling the columns of $\mathbf{G}_0$, after applying the map, k columns of $\mathbf{G}_0$ will have only a single non-zero entry. During conversion to RREF, if the current pivot element is in one of these columns, the expensive row reduction step can be skipped. On average, $k/2$ of the k pivot columns of $\widetilde{\mu}^{(i)}(\mathbf{G}_0)$ were pivots in $\mathbf{G}_0$, which means that half of the row reductions can be skipped. Thus, on average, this optimization reduces RREF latency by 50%.

It is important to note that the number of pivots that can be reused varies based on the permutation behind the secret monomial map. Thus, a direct application will result in non-constant-time performance, which may introduce a vulnerability to timing attacks. This is not an issue for verification, so it can be applied directly. For signature generation, instead, the variance in latency would reveal information about the secret ephemeral monomials. However, one can limit the number of pivots reused to some value $P < \frac{k}{2}$ such that the probability of non-constant-time operation is below an acceptable threshold. So long as there are at least P pivots that can be reused, the operation will be constant-time. This quasi-constant-time approach allows some of the performance benefit to be achieved with a strong argument for security. The probability that a monomial has exactly p reusable pivots is $\binom{k}{p} \cdot \binom{k}{k-p} / \binom{k}{n}$, since the permutation would be selecting p columns that were pivot columns and $k-p$ that were non-pivot columns. We can calculate the probability of $\widetilde{\mu}^{(i)}(\mathbf{G}_0)$ having at least P reusable pivots and then select P such that the likelihood of non-constant-time performance is below the maximum signature limit considered by NIST of 2^{64}. This leads to a value of 25, 51, and 79 for P for security categories 1, 3, and 5, respectively. This yields a performance improvement of $20\% - 35\%$ for RREF with only an average of 1 non-constant-time operation before the signature limit is reached. It is worth noting that the variance in latency is related only to the ephemeral monomial, not the long-term secret monomial.

Partially Reduced RREF. The technique of reusing pivots and thus not reducing already reduced columns can be extended to also skip the reduction of the pivot element itself. This is possible because the canonical form computation depends only on the number of 0 and not on the exact values of the other non-zero entries within the generator matrix.

Non-Constant-Time Computation of Canonical Forms. Let us first define $\mathsf{cnt}_x(\mathbf{v}) = |\{i \,|\, \mathbf{v}_i = x, \forall i \in [n]\}|$ for $x \in \mathbb{F}_q$ and $\mathbf{v} \in \mathbb{F}_q^n$. It is easy to see that it would be very costly to implement the algorithm CF in constant time. However, there are many operations for which one knows beforehand that the outcome would not lead to a shorter matrix $\mathbf{M}$. For example, we know that the rows containing the most zeros scaled with rows containing no zeros lead to the shortest matrices. This idea is implemented in Algorithm 22. This algorithm first computes the sets Z and J, which contain, respectively, the indices of the rows where a zero is present, and the most zeros among all rows. Then, only for those rows we compute ScaleCF.

```
Algorithm 22: Improved CF (non CT)
Input: $\mathbf{G} \in \mathbb{F}_q^{k \times n-k}$ non-IS of Generator Matrix.
Output: $\mathbf{M} \in \mathbb{F}_q^{k \times n-k}$.
Data: $\mathbf{B} \in \mathbb{F}_q^{k \times (n-k)}$: temporary buffer.
$J \in \mathbb{Z}^k$: track rows which contain the maximal numbers of zeros, $Z \in \mathbb{Z}^k$: track rows which contain zeros
/* set $\mathbf{M}$ to the largest matrix */
$\mathbf{M} \leftarrow (q-1)^{k \times (n-k)}$
/* track the rows which contain the maximum amount of zero */
$J \leftarrow \{i \,|\, \mathsf{max}(\mathsf{cnt}_0(\mathbf{G}[i])), \forall i \in [k]\}$
$Z \leftarrow \{i \,|\, \mathsf{cnt}_0(\mathbf{G}[i]) > 0, \forall i \in [k]\}$
for $i \notin Z$ do
    for $j \in [|J|]$ do
        $\mathbf{B}[j:] \leftarrow \mathbf{G}[J[j]:] \cdot \mathbf{G}[i:]^{-1}$
    if SubScaleCF($\mathbf{B}, \mathbf{M}[0:], |J|$) then
        for $j \in [k]$ do
            $\mathbf{B}[j:] \leftarrow \mathbf{G}[j:] \cdot \mathbf{G}[i:]^{-1}$
        $\mathbf{B} \leftarrow$ ScaleCF($\mathbf{B}$)
        if $\mathbf{B} \prec_{\mathbb{F}_q^{k \times (n-k)}} \mathbf{M}$ then
            $\mathbf{M} \leftarrow \mathbf{B}$
return $\mathbf{M}$
```

In addition to the idea described above, we can remove certain operations from ScaleCF, as we only need to compute the full ScaleCF in the case where we know that one of the rows in $\mathbf{B}$ is shorter than the shortest row we already know, which is $\mathbf{M}[0:]$. For this reason, we introduce the algorithm SubScaleCF, which computes the same as ScaleCF but only for the $|J| \times (n-k)$ sub-matrix $\mathbf{B}$. Additionally, it returns a true/false flag indicated that one of the rows in $\mathbf{B}$ is already shorter then the shortest the algorithm currently knows. Only in this case the full ScaleCF algorithm will be computed.

Blinding. As a non-constant-time implementation of canonical forms (see Algorithm 22) is considerably faster than a constant-time version, we apply a procedure called *blinding* to the input of the canonical form. This is done by applying to the matrix $\mathbf{G}$ a monomial map both on the left and on the right, via multiplication with the matrices $\mathbf{Q}_r \in M_k$ and $\mathbf{Q}_c \in M_{n-k}$, respectively.

Algorithm 23: SubScaleCF

```
Input: $\mathbf{G} \in \mathbb{F}_q^{z \times (n-k)}$, $\mathbf{M} \in \mathbb{F}_q^{n-k}$, $z \in \mathbb{Z}$.
Output: true/false.
Data: $\mathbf{T} \in \mathbb{F}_q^{n-k}$: temporary buffer.
for $i \in [z]$ do
    $s = \sum_{j=0}^{n-k} \mathbf{G}[i,j]$
    if $s \neq 0$ then
        $s \leftarrow s^{-1}$
    if $s = 0$ then
        $s = \sum_{j=0}^{n-k} \mathbf{G}[i,j]^{-1}$
    if $s = 0$ then
        return 0
    $\mathbf{T} \leftarrow s \cdot \mathbf{G}[i:]$
    if $\mathbf{T} \prec_{\mathbb{F}_q^{n-k}} \mathbf{M}$ then
        return true
return false
```

Algorithm 24: blind

```
Input: $\mathbf{G} \in \mathbb{F}_q^{k \times n-k}$ non-IS of Generator Matrix, state of an already initialized XOF.
Output: $\mathbf{G}' \in \mathbb{F}_q^{k \times n-k}$
Data: $\mathbf{M}_l \in \mathbb{F}_q^{k \times (n-k)}$, $\mathbf{M}_r \in \mathbb{F}_q^{(n-k) \times k}$
$\mathbf{M}_r \leftarrow$ SampleMonomial(state)
$\mathbf{M}_l \leftarrow$ SampleMonomial(state)
return $\mathbf{M}_l \cdot \mathbf{G} \cdot \mathbf{M}_r$
```

We assume, conservatively, that the side-channel analysis of CF entirely reveals its input. With information-theoretic arguments we argue that, thanks to blinding, any type of side-channel information leakage is avoided. For the sake of simplicity, we show this by focusing on a single round of the Sigma protocol. We first consider what would happen when blinding is not applied. Let $\mathbf{A}$ denote the input to CF (on the signer's side) and $\mathbf{B} = \mathrm{CF}(\mathbf{A})$. Let us now assume that, in this round, the challenge is $\neq 0$, and denote by $\mathbf{A}'$ the input to CF, on the verifier's side. An attacker would know both $\mathbf{B}$ and $\mathbf{A}'$, since these values can be computed from a honest signature, but does not know $\mathbf{A}$. Any information about $\mathbf{A}$ may provide useful information about the ephemeral map employed by the signer which, in turn, can be combined with the signer's response to gather information about the secret map. Even though we are not aware of attacks that make use of this information, in such a case there is obviously some (potentially dangerous) information leakage.

Note that the attacker knows that $\mathbf{A} = \mathbf{Q}_r \cdot \mathbf{A}' \cdot \mathbf{Q}_c$ for some (unknown) $\mathbf{Q}'_r \in M_k$ and $\mathbf{Q}'_c \in M_{n-k}$. Hence, without blinding the attacker can recover both $\mathbf{Q}'_r$ and $\mathbf{Q}'_c$.
Now, we consider blinding: on the signer's side, the input to CF is

$$\widetilde{\mathbf{A}} = \mathbf{Q}_r \cdot \mathbf{A} \cdot \mathbf{Q}_c = \underbrace{\mathbf{Q}_r \cdot \mathbf{Q}'_r}_{\widetilde{\mathbf{Q}}_r} \cdot \mathbf{A}' \cdot \underbrace{\mathbf{Q}'_c \cdot \mathbf{Q}_c}_{\widetilde{\mathbf{Q}}_c}.$$

Since $\mathbf{Q}_c$ and $\mathbf{Q}_r$ are drawn uniformly at random from M_k and M_{n-k}, $\widetilde{\mathbf{Q}}_c$ and $\widetilde{\mathbf{Q}}_r$ are uniformly random as well: this prevents any sort of information leakage. In other words: the input to CF, on the signer's side, is one among all matrices leading to the canonical form $\mathbf{B}$. The attacker learns nothing more than this but, as we already stressed, this information is already publicly revealed.

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

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

- The Fiat-Shamir transformation.
- Protocol-level modifications.
- The Sigma protocol of Figure 2 is secure.

The first 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. Then, 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 4, and briefly summarized, one by one, in Section 3, where we refer to the appropriate literature for further details about their impact on security. Finally, the third item implies that the underlying protocol is complete, sound and Zero Knowledge. These properties are trivially satisfied for our employed protocol; see [BMPS20] for the basic case of a Sigma protocol based on LEP, and [PS23, CPS23] for the extended notions of equivalence.

In particular, for the current version of LESS, special soundness is based on the hardness of the following problem.

Problem 1 (**Canonical Forms Linear Equivalence Problem (CF-LEP)**). *Given two generator matrices $\mathbf{G}, \mathbf{G}' \in \mathbb{F}_q^{k\times n}$, find two size-k sets $J, J' \subseteq \{1, \cdots, n\}$ such that*

$$\mathrm{CF}\big(\mathbf{G}_J^{-1} \cdot \mathbf{G}_{\{1,\cdots,n\}\setminus J}\big) = \mathrm{CF}\big(\mathbf{G}'^{-1}_{J'} \cdot \mathbf{G}'_{\{1,\cdots,n\}\setminus J'}\big)$$

As shown in [CPS23], when random codes are concerned (as in LESS), CF-LEP is equivalent to the standard LEP in terms of hardness, since one can always reduce, in polynomial time, one problem to the other and vice versa.

5.1 Known Attacks

Broadly speaking, most 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 B.

Information-Set Decoding. An *Information-Set Decoding (ISD)* algorithm is an iterative procedure that 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 B.2), an attacker is interested in finding a certain number ℓ of distinct codewords with Hamming weight w. Let $C_{\mathrm{ISD}}(q,n,k,v)$ 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 v, out of $N(v)$, is

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

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

$$f\big(\ell, N(v)\big) \approx \begin{cases} \ell & \text{, if } \ell \ll N(v), \\ \ell \cdot \ln(\ell) & \text{, else.} \end{cases} \tag{2}$$

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

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

where

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

and

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

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

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

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 in terms of the number 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 B.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 B.1.

Attacks based on Low-weight Codeword Finding. Many 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 $\leq v$, that is

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

It then holds that $A = \mu(A')$ and, when $v \ll n$, we further have $|A| \ll |\mathscr{C}| = q^k$. Roughly speaking, since A and A' only contain a few codewords, reconstructing the map μ 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 $\mathbf{c} \in \mathscr{C}$, $\mathbf{c}' \in \mathscr{C}'$ such that $\mathbf{c} = \mu(\mathbf{c}')$. This idea was first proposed in [Beu21] and later refined in [BBPS23]. We give more details on those approaches in Appendix B.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 μ.

ii) Find those collisions.

iii) Use the collisions to reconstruct the secret monomial μ.

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 a lower overall 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 $v \in \{1, \cdots, n\}$ finding lists $L_1 \subseteq \mathscr{C}$ and $L_2 \subseteq \mathscr{C}'$ with weight-v codewords and such that $L_1 \cap \mu(L_2) := \{(\mathbf{c}, \mathbf{c}') \in L_1 \times L_2 \mid \mathbf{c}' = \mu(\mathbf{c})\}$ 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 of producing these lists is

$$f\big(\ell, N(v)\big) \cdot C_{\text{ISD}}(v).$$

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

$$|L_1 \cap \mu(L_2)| = \frac{|L_1| \cdot |L_2|}{N(v)} = \frac{\ell^2}{N(v)},$$

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

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

Following our conservative design, we assume that such a single collision is sufficient and let $\ell = \sqrt{N(v)}$. Further, since this implies $\ell \ll N(v)$ we have (compare to Equation (2))

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

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*

$$\frac{1}{\sqrt{N(v)}} \cdot C_{\text{ISD}}(v) > 2^\lambda.$$

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

Attack based on Canonical Forms. In [CPS23], together with the introduction of canonical forms, a new meet-in-the-middle style algorithm to solve LEP is introduced. The algorithm enumerates permutations $\pi \in S_{k,n}$ and stores the canonical form of $\pi(\mathcal{C})$ (or, respectively, $\pi(\mathcal{C}')$) in two lists. A collision between both lists implies two permutations π and π' with $\text{CF}(\pi(\mathscr{C})) = \text{CF}(\pi'(\mathscr{C}'))$ from which the monomial map μ can be recovered. The list sizes to expect a collision in case of LESS parameters using $k = n/2$ is of the order $\sqrt{\binom{n}{k}} \sim 2^{n/2}$. Therefore, the complexity of this algorithm is generally higher than the lower bound used for parameter selection in Criterion 2.

6 Performance (2.B.2)

In this section, we report the performance figures for LESS. We begin by presenting a summary of the byte lengths for the various protocol objects, and the resulting key and signature sizes, in Table 3, (rounded up to account for bytes encoding). We remind the reader that (n, k, q) are the coding theory parameters for the codes used in LESS, whereas (t, w, s) are protocol-level parameters; m denotes the Hamming weight of the integer t, in its binary decomposition.

		NIST Category 1	NIST Category 3	NIST Category 5
$\ell_{\mathsf{tree\_seed}}$	(bytes)	16	24	32
$\ell_{\mathsf{sec\_seed}}$	(bytes)	32	48	64
$\ell_{\mathsf{pub\_seed}}$	(bytes)	16	24	32
ℓ_{salt}	(bytes)	32	48	64
ℓ_{digest}	(bytes)	32	48	64
$\ell_{\mathbf{G}_i}$	(bytes)	$\left\lceil k(n-k)\lceil \log_2(q) \rceil / 8 \right\rceil + \lceil n/8 \rceil$		
ℓ_{mono}	(bytes)	$\lceil n/8 \rceil$		
ℓ_{path}	(bytes)	$\min \{\lfloor w \log_2(t/w) + m - 1 \rfloor ; t - w\} \cdot \ell_{\mathsf{tree\_seed}}$		
ℓ_{sk}	(bytes)	$\ell_{\mathsf{sec\_seed}}$		
ℓ_{pk}	(bytes)	$(s-1) \cdot \ell_{\mathbf{G}_i} + \ell_{\mathsf{pub\_seed}}$		
ℓ_{sig}	(bytes)	$\ell_{\mathsf{salt}} + \ell_{\mathsf{path}} + w \cdot \ell_{\mathsf{mono}} + \ell_{\mathsf{digest}} + 1$		

Table 3: Choice data sizes (in bytes). We remark that the provided formula for ℓ_{path} refers to the worst case.

For $\ell_{\mathbf{G}_i}$, it is essential to note that, in addition to storing the generator matrix itself, we must also save the pivot columns, hence the additional n bits in the formula.

Next, we present our choice of parameters for LESS, displayed in Table 3. In addition to the design criteria and security arguments presented in the previous sections, we include in our thought process some considerations related 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 very large data sizes and thus remain around the psychological threshold of 100kB. With this in mind, as anticipated in Section 2, we explore two different directions: a first configuration, where we keep the smallest possible public key (corresponding to the case $s = 2$), and a second configuration, where we shorten the signature size, at the cost of larger public keys (corresponding to the case $s > 2$). To uniquely identify our different parameter sets, we use the nomenclature LESS-n-t, where n is the code length and t the number of rounds in the iterated protocol. Note that the table 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 PRNG seed.

Average Signature Size. We remark that the signature size given in Table 3 shall be interpreted as a worst case, since the provided formula for ℓ_{path} refers to the maximum number of nodes which need to be revealed. For completeness, we provide also the average size of a signature, computed by taking the average over 2^{10} signatures for random messaged of length 80.

NIST	**Parameter**	**Code Params**			**Prot. Params**			**pk**	**sig** (B)	
Cat.	**Set**	n	k	q	t	w	s	(B)	Worst case	Avg case
1	LESS-252-192	252	126	127	192	36	2	13940	2625	2289
	LESS-252-68				68	42	4	41788	1825	1745
	LESS-252-45				45	34	8	97484	1329	1313
3	LESS-400-220	400	200	127	220	68	2	35074	6329	5585
	LESS-400-102				102	61	4	105174	4131	3867
5	LESS-548-345	548	274	127	345	75	2	65793	10680	9464
	LESS-548-137				137	79	4	197315	7436	7116

Table 4: Parameter sets for LESS, and resulting data sizes.

6.1 Performance in Software

The two most demanding operations in the LESS Sign and Verify algorithms are the computation of the RREF and the CF. Together, they account for more 90% of the overall computation time. 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 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 double-precision multiplication 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), typically performing 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 execution times obtained for our reference code, measured in Megacycles (Mcycles). The values were collected on an 13th Gen Intel(R) Core(TM) i7-1355U, clocked at 5.0 GHz. The compiler used was `gcc` version 11.4 on an `Ubuntu 22.05`machine. Clock cycle values were collected via `rtdscp`, as averages of 128 runs, with `-O3 -march=native -flto -ftree-vectorize -funroll-loops` 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://github.com/less-sig/LESS

NIST Cat.	**Parameter Set**	**KeyGen** (Mcycles)	**Sign** (Mcycles)	**Verify** (Mcycles)
1	LESS-252-192	1.3	376.4	303.5
	LESS-252-68	3.8	135.6	107.3
	LESS-252-45	8.6	89.5	71.9
3	LESS-400-220	4.0	1087.8	865.1
	LESS-400-102	11.5	502.7	399.4
5	LESS-548-345	10.0	3719.2	2959.0
	LESS-548-137	29.6	1476.6	1188.1

Table 5: Timings for the reference implementation of LESS benchmarked on a 13th Gen Intel(R) Core(TM) i7-1355U. Clock cycle values collected via `rtdscp`, as averages of 128 primitive runs.

In addition to the reference implementation, we report two additional optimized implementations. One for Intel/AMD based systems supporting the AVX2 instruction set architecture. And one for ARM based systems supporting the NEON instruction set. Both instruction sets support **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD) instructions, which allow vectorization.

The test system for the AVX2 implementation is the same as for the reference code. The numbers in Section 6.1 are averages of 128 runs, with `-O3 -mavx -mavx2 -mbmi -bmi2 -flto -ftree-vectorize -funroll-loops` compilation options.

NIST Cat.	**Parameter Set**	**KeyGen** (Mcycles)	**Sign** (Mcycles)	**Verify** (Mcycles)
1	LESS-252-192	0.5	127.5	122.7
	LESS-252-68	1.4	45.9	44.8
	LESS-252-45	3.1	30.6	30.0
3	LESS-400-220	1.7	401.5	392.7
	LESS-400-102	4.5	184.0	181.3
5	LESS-548-345	4.4	1424.2	1396.0
	LESS-548-137	12.0	560.0	564.6

Table 6: Timings for the optimized AVX implementation of LESS. Values collected on an 13th Gen Intel(R) Core(TM) i7-1355U. Clock cycle values collected via `rtdscp`, as averages of 128 primitive runs.

The test system for the NEON implementation is the Apple M1 Max chip.

6.2 Performance in Hardware

In this section, we report preliminary performance and area results for 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.

NIST Cat.	**Parameter Set**	**KeyGen** (Mcycles)	**Sign** (Mcycles)	**Verify** (Mcycles)
1	LESS-252-192	1.0	260.2	247.8
	LESS-252-68	2.8	92.2	93.9
	LESS-252-45	6.3	61.3	63.6
3	LESS-400-220	2.8	725.7	639.0
	LESS-400-102	8.9	336.6	334.1
5	LESS-548-345	7.1	2397.0	2338.5
	LESS-548-137	20.0	951.9	960.7

Table 7: Timings for the optimized NEON implementation of LESS. Values collected on an Apple M1 Max.

RREF in Hardware. The most computationally expensive operation in LESS is the conversion of generator matrices to RREF. To provide insight into the performance improvement possible through hardware acceleration, Table 8 provides the results of the RREF unit presented in [BWMG23]. This architecture aims at high performance, so scaling and row addition operations are performed on an entire row in one clock cycle. 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-252-{192,68,45}	125	29.4	20.4	25.5	16.1	129.3
LESS-400-{220,102}	111	46.3	33.8	40	40.4	364.3
LESS-548-{345,137}	100	66.6	47.4	55	75.7	756.6

Table 8: RREF Implementation Results on Artix-7

7 Known Answer Tests (2.B.3)

The LESS KAT archive is available at the following link:

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

8 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 the literature.

Solid Security Foundations. The security of LESS reduces to 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).

Smaller Signatures. Thanks to the use of canonical forms, the signature sizes for LESS were dramatically reduced from the previous round, and now range between 2.5 kB and 1.2 kB (previously 8.4 kB to 5.2 kB) for Category 1, which is an improvement up to a factor ×4.3. In Category 5, the signature size was reduced from 32.5 kB (26.1 kB) to 9.98 kB (6.75 kB), which is an improvement of up to a factor ×3.9. In contrast to other schemes, this is not accompanied by an increase of computation time, as we explain next.

Faster Timings. We implemented various computational optimizations in our procedures; these turned out to be so effective that not only they allowed to counter the extra cost coming from the introduction of canonical forms, but they also resulted in a very meaningful speed up. Indeed, we were able to improve the runtime of the optimized implementation by a factor between ×11.1 for Category 3 and ×1.3 for Category 1, compared to Round 1. But this is not all. A distinctive property of the LESS signature scheme, in fact, is that it gets faster as the signatures get smaller (within the same security category); this is in sharp contrast with the dichotomy “short” vs “fast” typical of various other post-quantum schemes. This phenomenon is directly linked to the structure of the scheme: the number t of rounds of the underlying ZK protocol, in fact, affects both the number of objects published in the signature (and thus the total size of a signature) and the number of operations performed (which speeds up the whole signature computation). Reducing t is possible by enlarging the public key, as we note below.

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. Furthermore, LESS public keys can grow quickly, if parameters are tuned in order to minimize signature size (and speed up computation) as described above. 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) and the computation of the canonical forms. This makes it so that instances with larger public keys are also the fastest ones, leading to a potentially unpleasant trade-off. 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

[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.

[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.

[BPS$^+$23] Giacomo Borin, Edoardo Persichetti, Paolo Santini, Federico Pintore, and Krijn Reijnders. A guide to the design of digital signatures based on cryptographic group actions. Cryptology ePrint Archive, Paper 2023/718, 2023.

[BWMG23] Luke Beckwith, Robert Wallace, Kamyar Mohajerani, and Kris Gaj. A High-Performance Hardware Implementation of the LESS Digital Signature Scheme. In Thomas Johansson and Daniel Smith-Tone, editors, *Post-Quantum Cryptography*, pages 57–90, Cham, 2023. Springer Nature Switzerland.

[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.

[CPS23] Tung Chou, Edoardo Persichetti, and Paolo Santini. On linear equivalence, canonical forms, and digital signatures. *Cryptology ePrint Archive*, 2023.

[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. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 759–789. Springer, 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.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 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. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 351–378. Springer, 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* $\mathscr{C}$ 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 intuitively represented 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.* The generator matrix then defines the code as a mapping between the vectors $\mathbf{u} \in \mathbb{F}_q^k$ and the corresponding words $\mathbf{uG}$. Obviously, there exists 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. $\mathscr{C} = \{\mathbf{x} \in \mathbb{F}_q^n : \mathbf{Hx}^T = 0\}$. 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., $\mathscr{C}^\perp = \{\mathbf{y} \in \mathbb{F}_q^n \;:\; \forall \mathbf{x} \in \mathscr{C},\;\; \mathbf{x}\cdot\mathbf{y}^T = 0\}$. It is then easy to see that a parity-check matrix of a linear code is a generator of its dual, and vice versa. 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., $\mathscr{C} \subseteq \mathscr{C}^\perp$, are called *weakly self-dual*, and codes that are equal to their dual, i.e., $\mathscr{C} = \mathscr{C}^\perp$, are called simply *self-dual.*

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

$$N_v = \binom{n}{v}(q-1)^v q^{k-n}.$$

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

B Known Attacks

B.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 $\mathscr{C}$ is defined as the code $\{\mathbf{c}\otimes\mathbf{a} \mid c \in \mathscr{C}\}$, where $\mathbf{a} = (a_1, \dots, a_{q-1})$ 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[1]. 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 $q < 5$.

B.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 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 $\mathscr{C}, \mathscr{C}' \subseteq \mathbb{F}_q^n$, such that $\mathscr{C}' = \mathscr{C}\mathbf{Q}$ 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 $A' = A\mathbf{Q} = \{\mathbf{c}\mathbf{Q} \mid \mathbf{c} \in A\}$, 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 $L_1 = L_2\mathbf{Q} = \{\mathbf{c}\mathbf{Q} \mid \mathbf{c} \in L_1\}$. 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.

[1] 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.

Beullens' Algorithm. Beullens proposed an algorithm to retrieve the hidden $\mathbf{Q}$ by exploiting 2-dimensional subcodes with support v [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 v (L_1 with subcodes from $\mathscr{C}$, L_2 with subcodes from $\mathscr{C}'$), 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 v.

For finding 2-dimensional subcodes with support size v 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 v are constructed. They first find codewords of weight v', and then search for combinations among them that form 2-dimensional codes of support v. This allows a direct application of advanced ISD techniques to find weight-v' codewords.[2] Also by ensuring that the overlap between the pairs of codewords used in the subcode construction is small, i.e. ensuring that $2v' - v$ 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.

B.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

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

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

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

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.