LESS is Even More: Optimizing Digital Signatures from Code Equivalence

Luke Beckwith[1], Andre Esser[2], Edoardo Persichetti[3], Paolo Santini[4], Floyd Zweydinger[2]

[1] George Mason University, USA
[2] Technology Innovation Institute, UAE
[3] Florida Atlantic University, USA [4] Marche Polytechnic University, Italy

Abstract. LESS is a signature scheme based on the code equivalence problem that has advanced to the second round of the NIST PQC standardization process. While promising, the scheme suffers from relatively large signatures and moderate to slow signing and verification times. Chou, Santini, and Persichetti recently introduced a variant of LESS relying on *canonical forms* to significantly reduce signature sizes. However, the overall performance impact of this approach remained largely unclear. In this work, we provide the first implementation of the new LESS variant and show that, in its original form, it performs poorly due to the overhead of computing canonical forms in a naïve way. We then introduce a series of algorithmic and implementation-level optimizations that reduce this overhead to about 10%, showing that the signature size reduction comes at minor cost. In addition, we present further improvements to the signature scheme as a whole, as well as a re-parameterization. The resulting scheme achieves speedups of 2.5× to 10× over the Round 1 NIST submission, while maintaining the reduced signature sizes.

Keywords: Digital Signatures · Post-Quantum

1 Introduction

The transition to PQC has led to an increased focus on cryptographic diversity, ensuring that security does not rely on a single hard problem. Code-based cryptography is one of the most promising areas of investigation in this context, enjoying conservative security guarantees and a rapidly improving performance profile. LESS, a code-based signature scheme, emerged as an intriguing candidate in the NIST PQC signature standardization effort [NIS23], thanks to a clean and elegant design. Constructed via the Fiat-Shamir transformation from a Zero-Knowledge Proof of Knowledge (ZK-PoK), it was recently selected to advance to the second round of the competition. Still, the Round 1 LESS submission exhibited somewhat large signatures, and relatively slow signing and verification.

A recent work [CPS25] shows how to significantly reduce the LESS signature size, introducing so-called "canonical forms" to lower the communication cost of the underlying ZK-PoK. For instance, signatures for NIST Category 1 can be reduced from 5-8 KB down to 1–2.5 KB. However, this reduction comes at a price, as computing canonical forms introduces additional overhead during signature and verification. Indeed, the authors in [CPS25] estimated that each canonical form computation could in principle be as expensive as multiple Gaussian eliminations. Given that LESS was already among the slowest candidates in Round 1, with Gaussian elimination being precisely the bottleneck in computation, it remained unclear whether this trade-off would result in a competitive scheme or if it merely shifted the bottleneck from signature size to performance.

In this work, we present a full implementation of the improved LESS scheme and introduce a series of optimizations targeting both the computation of canonical forms and the signature scheme as a whole. Through algorithmic enhancements, implementation-level improvements, and re-parameterization, we accelerate LESS's signing and verification by a factor ranging from 2.5× to over 10× compared to the initial submission. The Round 2 LESS submission already incorporates many of the improvements presented in this work; yet, we present here additional ideas that further improve its performance by a factor of about 2×. In this context, we improve the computation of canonical forms significantly, showing that with an overhead of only $7 - 15\%$ in our optimized implementation the improvement is not a trade-off, but instead comes only at a minor cost. Our results position LESS as a strong candidate, achieving both compact signatures and competitive performance relative to other zero-knowledge-based constructions.

Our Contributions. The work of [CPS25] is able to reduce signatures to a very competitive size, but raised several questions about the practicality of this tweak. The focus of this work, therefore, is to address the computational aspects of the protocol. As we mentioned, a key bottleneck in the Round 1 LESS implementation was the Gaussian elimination algorithm required to compute the Reduced-Row Echelon Form (RREF) of the matrices involved in the protocol, which accounted for a very significant portion of the total signing and verification time. By adding canonical forms, the computational cost is further dominated by the combined overhead of computing both canonical forms and RREF.

As an initial step, we provide a full implementation of the canonical form function proposed in [CPS25], establishing a baseline for our benchmarks. We find that here the RREF and CF computations jointly make up for about $88 - 94\%$ of the total signing / verification time, while the CF is $\times 9$ slower then the RREF. We then introduce optimizations on different levels to ultimately bring the contribution of those routines down to just $55 - 60\%$ of the overall signing and verification time. Whereby the overwhelming majority is now only contributed by the RREF, as the computational overhead of the CF is $\leq 10\%$ as the CF is now $\times 2.4$ faster then the RREF.

Variable-time RREF. We introduce several optimizations to the RREF computation during verification, where constant-time execution is not required. The first-round LESS implementation used the same RREF approach for both signing and verification, despite verification having different constraints. Since the matrices being transformed into systematic form are of the form $(\mathbf{I}_k \mid \mathbf{A})\mathbf{Q}$ for some monomial[1] matrix $\mathbf{Q}$, they already include many pivot columns. Reusing those pivot columns allows to save a great part of the otherwise necessary computations. Additionally we implement early aborts and apply rescaling and row additions only when necessary.

Canonical Form Computation. For a given matrix $\mathbf{A} \in \mathbb{F}_q^{k\times(n-k)}$, the computation of its canonical form $\mathrm{CF}(\mathbf{A})$ requires finding the matrix having the most compact representation, among matrices of the form $\mathbf{Q}_r\mathbf{A}\mathbf{Q}_c$, where $\mathbf{Q}_r \in M_k$ and $\mathbf{Q}_c \in M_{n-k}$ are monomial matrices. To accelerate this process, we derive structural properties of $\mathrm{CF}(\mathbf{A})$ in dependence on the entries of $\mathbf{A}$. This allows us to fix certain entries of $\mathbf{Q}_r$ and $\mathbf{Q}_c$ a priori, which previously required testing multiple candidates.

These optimizations introduce a variable-time computation, which could leak information about the supposedly secret matrices $\mathbf{Q}_r$ and $\mathbf{Q}_c$. To mitigate this, we introduce a "blinding" technique: by first applying random monomial matrices to $\mathbf{A}$, we obtain a transformed matrix $\tilde{\mathbf{A}} = \mathbf{Q}'_r\mathbf{A}\mathbf{Q}'_c$. Note that, due to the properties of canonical forms, the canonical form of $\tilde{\mathbf{A}}$ remains the same as that of $\mathbf{A}$, ensuring that any side-channel information obtained during computation does not reveal details about $\mathbf{Q}_r$ and $\mathbf{Q}_c$. Therefore, all improvements on the CF computation translate 1-to-1 to the signing routine.

[1] A permutation with scaling factors.

Built-in Side-channel Resistance The newly introduced "blinding" technique does not only allow us to speedup the computations, by leveraging variable-time computations, but it is also an additional side-channel resistance layer built into the scheme, which allows each subsequent function, and not just the canonical form, to be variable-time. As a consequence, LESS is one of the few schemes which sacrifices a little of its computation time $\leq 1\%$ so that large parts of the implementation can be written in variable-time fashion. This makes the overall implementation effort much easier. All in all, one only needs to be careful to write constant-time code while sampling random monomial matrices, applying those to the generator matrix and while applying the subsequent RREF. Additionally, we further strengthen LESS's side-channel resistance by switching from a Fisher-Yates approach to an odd-even sorting networks for the sampling of random permutations.

$\mathbb{F}_q$*-Arithmetic.* Another key component involved in the RREF and CF computations is the arithmetic over $\mathbb{F}_q$. We provide a significantly improved AVX2-based implementation for this arithmetic. For subtraction and addition, we have reduced the number of instructions from six to three, resulting in a $\times 2$ improvement compared to the Round-1 LESS submission, by leveraging the specialized `blendv` AVX2 instruction. Similarly, for $\mathbb{F}_q$ multiplication, we have reduced the number of instructions, leading to a 23.7% improvement in timings. Furthermore, we have optimized the arithmetic by utilizing the AVX512 instruction set available on newest generation x86 CPUs, which results in an additional 26% improvement in timings compared to our AVX2 implementation, while processing twice as many field elements. Additionally, we provide an optimized NEON implementation for ARM processors.

Re-parameterization. While the authors of [CPS25] demonstrated the potential of canonical forms using the Round 1 LESS parameters, we show that this enhancement opens up new possibilities for parameter selection. In the original Round 1 LESS parameters, the Fiat-Shamir construction relied on a large number of repetitions, which were crucial for achieving smaller signatures. This was made possible by a skewed challenge distribution ensuring that the much smaller ephemeral responses appear more often than larger ones. However, the introduction of canonical forms significantly reduces the communication cost, eliminating the need for such a high number of repetitions. This allows us to maintain competitive signature sizes while using far fewer rounds. Note that any reduction in the amount of rounds leads to a linear improvement in both signing and verification times.

The combined effect of the improvements outlined above is reflected in the timing results as detailed in Table 5 for NIST Category 1 parameters. The canonical form computation now accounts for only 10.1% of the total time, with the previously dominant RREF computations contributing 40.0%. The majority of the computation time is now dedicated to computing commitments through the KECCAK function.

2 Preliminaries

Notation. Let $\mathbb{F}_q$ be the finite field with q elements. In concrete instantiations, throughout we use $q = 127$. Naturally the vector and matrix space over this field are defined as $\mathbb{F}_q^n$ and $\mathbb{F}_q^{k\times n}$. We denote vectors with bold lower case letters, e.g., $\mathbf{v} \in \mathbb{F}_q^n$ and matrices with bold upper case letters, e.g., $\mathbf{G} \in \mathbb{F}_q^{k\times n}$. Furthermore we refer by $\mathbf{v}[i]$ to the i-th element of a vector $\mathbf{v}$ and $\mathbf{G}[i, j]$ describes the element in the i-th row and j-th column of a matrix $\mathbf{G}$. We describe with $\mathbf{G}[i :]$ the i-th row, and with $\mathbf{G}[: j]$ the j-th column, of the matrix $\mathbf{G}$. Additionally $\mathbf{G}[i :] \cdot \mathbf{G}[j :]$ denotes the element-wise multiplication of the two rows.

Permutation and Monomial Maps. We denote by S_n the symmetric group on n elements, comprised of *permutations* of n objects. For a vector $\mathbf{v} = (v_1, \cdots, v_n)$, it holds that $\pi(\mathbf{v}) = \left(v_{\pi^{-1}(1)}, \cdots, v_{\pi^{-1}(n)}\right)$. As is well known, each permutation can be represented also as an $n \times n$ matrix such that every row and every column has a unique element equal to 1 while all the other elements are 0. If $\mathbf{P}$ is the permutation matrix associated to $\pi \in S_n$, we then have $\pi(\mathbf{v}) = \mathbf{v} \cdot \mathbf{P}$. Sometimes we slightly abuse notation and write $\mathbf{P} \in S_n$ to imply that $\mathbf{P}$ is a permutation matrix.

The symmetric group can be generalized to a group M_n, known as the monomial group, formed by *monomial maps*, that is, functions of the form $\mu := (\pi, \mathbf{a})$ with $\pi \in S_n$ and $\mathbf{a} \in (F_q^*)^n$, acting as $\mu(\mathbf{v}) = \left(a_1 \cdot v_{\pi^{-1}(1)}, \cdots, a_n \cdot v_{\pi^{-1}(n)}\right)$. As for permutations, we can represent a monomial map via a matrix $\mathbf{Q} \in \mathbb{F}_q^{n \times n}$ such that every row and column has a unique non-zero element from $\mathbb{F}_q^*$. Again, for such matrices, we will sometimes be flexible with notation and write $\mathbf{Q} \in M_n$. We naturally extend the action of monomial maps to matrices $\mathbf{A}$, i.e., $\mu(\mathbf{A})$ indicates the matrix resulting from the action of μ on the columns of $\mathbf{A}$. In other words, if $\mathbf{Q}$ is the monomial associated with μ, we have $\mu(\mathbf{A}) = \mathbf{G} \cdot \mathbf{Q}$.

Orderings and Sorting. Let $\leq_{\mathbb{F}_q}$ be a total ordering on the prime field $\mathbb{F}_q$. Let $\textsc{Sort} : (\mathbb{F}_q)^* \to (\mathbb{F}_q)^*$ be a routine that sorts a vector of arbitrary length according to this ordering. We extend this ordering to the vector space $\mathbb{F}_q^n$ by letting $\mathbf{v} \leq_{\mathbb{F}_q^n} \mathbf{v}'$ if and only if either $\mathbf{v}_i = \mathbf{v}'_i$, $\forall i \in [n]$ or $\exists i \in [n], \text{s.t.}\, \mathbf{v}_i < \mathbf{v}'_i$, 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 a $k \times n - k$ 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 a $k \times n - k$ matrix and sorts them according to $\leq_{\mathbb{F}_q^k}$.

Code Equivalence. A linear code $\mathscr{C}$ of length n and dimension k, or $[n, k]$-code for short, is a k-dimensional subspace of $\mathbb{F}_q^n$. We represent such a code via a basis $\mathbf{G} \in \mathbb{F}_q^{k \times n}$ called the *generator matrix* of $\mathscr{C}$, with the vectors taken as rows i.e. $\mathscr{C} = \{\mathbf{mG} \mid \mathbf{m} \in \mathbb{F}_q^k\}$. Since bases for vector spaces are not unique, one typically identifies a standard one, known as the *systematic form*, as the one yielding a matrix of the form $(\mathbf{I}_k \mid \mathbf{A})$. We call two linear codes $\mathscr{C}$ and $\mathscr{C}'$ *linearly equivalent*, if there exist a monomial map $\mu \in M_n$ that maps elements from one into the other; for convenience, we indicate this with the notation $\mathscr{C}' = \mu(\mathscr{C})$. Given two linear codes, identifying such a monomial map μ (if it exists), is known as the code equivalence problem.

Definition 1 (Code Equivalence Problem)**.** Given two $[n, k]$-codes $\mathscr{C}$ and $\mathscr{C}'$ over $\mathbb{F}_q$, find a monomial map $\mu \in M_n$ such that $\mathscr{C}' = \mu(\mathscr{C})$.

This problem is also widely known as the *Linear Equivalence Problem (LEP)*, while the special case where the monomial map is a permutation is referred to as the *Permutation Equivalence Problem (PEP)*.

Definition 1 can also be written with respect to the generator matrices $\mathbf{G}$ and $\mathbf{G}'$ of $\mathscr{C}$ and $\mathscr{C}'$. Then, the target monomial map has to satisfy $\mathbf{G}' = \mathbf{S}\mu(\mathbf{G})$, where $\mathbf{S} \in \mathbb{F}_q^{k \times k}$ describes an appropriate change of basis. Often, this change of basis is fixed to transforming $\mu(\mathbf{G})$ into row-reduced echelon form (RREF), leading to $\text{RREF}(\mathbf{G}') = \text{RREF}(\mu(\mathbf{G}))$. Furthermore, when we restrict to generator matrices in systematic form the equivalence identity reads $\mathbf{G}' = \text{RREF}(\mu(\mathbf{G}))$, or $\mathbf{G}' = \text{RREF}(\mathbf{GQ})$ if using matrix notation.

3 LESS

The LESS signature scheme [BMPS20] stands out as the only signature scheme that was originally designed so that security is based entirely on the linear equivalence problem. Successive modifications bring the concrete security of LESS over to related problems, which however still reduce to LEP.

The final version of LESS is obtained starting from a simple zero-knowledge protocol, a 3-pass Sigma protocol in the style of Goldreich-Micali-Wigderson (GMW) [GMW91], from which a signature scheme is obtained via the Fiat-Shamir transformation [FS86]. As the base Sigma protocol only yields a soundness error of 1/2, it needs to be iterated, say t times, until a satisfactory soundness is reached. The protocol features several tweaks, which were progressively added over several works [BBPS21, PS23, CPS25]. Below, we briefly summarize only the latter, which is specific to the case of Linear Equivalence; for a general treatment of protocols based on group actions, we point the reader to [BPS+23].

3.1 Canonical Forms

Roughly speaking, for what concerns proving the existence of an equivalence between two codes, canonical forms can be seen as a manner to trade witness size (in this case, the amount of bits to prove that two codes are equivalent) with computational complexity. We briefly describe here just the main idea, at a very high level, and refer the interested reader to [CPS25] for mathematical details. Let $\mathbf{G}, \mathbf{G}' \in \mathbb{F}_q^{k\times n}$ be generator matrices for $\mathscr{C}, \mathscr{C}'$, respectively, and assume there exists a change of basis $\mathbf{S} \in \mathbb{F}_q^{k\times k}$ and a monomial map $\mu \in M_n$ such that $\mathbf{G}' = \mathbf{S} \cdot \mu(\mathbf{G})$. Then, the pair $(\mathbf{S}, \mu)$ is a witness to the equivalence relation. As we have already observed, providing $\mathbf{S}$ is actually not necessary and only μ is needed: one can indeed check that $\mu(\mathbf{G})$ and $\mathbf{G}'$ have the same RREF. In other words, the computation of the RREF is employed as an invariant, allowing to omit the change of basis $\mathbf{S}$ from the witness of the equivalence relation.

In the same spirit, the framework proposed in [PS23], and later formalized in [CPS25], allows to further reduce the size of the witness by omitting also some "information" about μ. The authors show how the action of any monomial map can be fully expressed by first stating which coordinates are moved to the k leftmost positions, and then specifying, separately, how the k coordinates on the left and the $n-k$ coordinates on the right are shuffled and scaled. The former information is represented via a permutation $\pi_{is} \in S_n$, while the latter via a special monomial map τ, which can be factored into two disjoint maps $\tau_r = \mathbf{Q}_r \in M_k$ and $\tau_c = \mathbf{Q}_c \in M_{n-k}$. With this formulation, in order to prove that two codes are linearly equivalent, it is enough to provide π_{is}, which can be efficiently encoded into the size-k set J. A verifier can then compute a matrix $\pi_{is}(\mathbf{G})$ which is equal to $\mathbf{G}'$ up to i) a change of basis, and ii) a monomial such as τ. In conclusion, while RREF expresses invariance under changes of basis, canonical forms are invariant under monomials like τ.

The resulting Sigma protocol is depicted in Figure 1. To be consistent with the LESS specification, we rely on the same notation, which requires the following additional functions. The function RREF* returns the systematic form of the input, together with the permutation[2] π used to bring its RREF into systematic form (basically, sort columns so that the identity is on the leftmost positions). The function CosetRep, on the other hand, first factors the input monomial map as we have discussed, and then returns the size-k set J. The name of this function comes from a group theory perspective, where each set J uniquely identifies one of the $\binom{n}{k}$ cosets forming a partition of the group M_n, and a canonical form can be seen as an efficiently computable representative for each class.

[2] note that this is the identity of the corresponding submatrix is invertible.

Public Data	System parameters $q, n, k \in \mathbb{N}$, hash function 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:		
$\widetilde{\mu} \xleftarrow{\$} M_n$		
$\Big((\mathbf{I}_k \mid \mathbf{A}), \pi\Big) \leftarrow \mathrm{RREF}^*\big(\widetilde{\mu}(\mathbf{G})\big)$		
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 $\mathsf{ch} = 0$:		
$\mathsf{rsp} \leftarrow \widetilde{\mu}$		
Else:		
$\mathsf{rsp} \leftarrow J = \textsc{CosetRep}(\pi \circ \widetilde{\mu} \circ \mu^{-1})$	$\xrightarrow{\mathsf{rsp}}$	**If** $\mathsf{ch} = 0$:
		$\big(\mathbf{I}_k \mid \mathbf{A}\big) \leftarrow \mathrm{RREF}^*\big(\mathsf{rsp}(\mathbf{G})\big)$
		Else:
		$\mathbf{A} \leftarrow \mathbf{G}'^{-1}_{\mathsf{rsp}} \cdot \mathbf{G}'_{\{1,\cdots,n\}\setminus\mathsf{rsp}}$
		Accept if $\textsc{Hash}\big(\mathrm{CF}(\mathbf{A})\big) = \mathsf{cmt}$

Figure 1: Description of the Sigma protocol based on canonical forms.

Failure probability As we shall see later on, in practice computing canonical forms boils down to scaling and reordering rows and columns so that the input matrix exhibits some special properties (e.g., rows and columns following a lexicograph ordering). Formally, a proper canonical form function would require that the resulting matrix can be uniquely and efficiently computed for any input. Actually, the canonical form functions put forward in [CPS25] relax this condition and accept that, for some input matrices, a canonical form cannot be computed: in such cases, the function returns a failure. In this context, we speak of failure probability and measure it as the probability that a uniformly random input leads to a failure. The expressions for the failure probabilities are rather convoluted and, for the sake of space, we do not report them here. We refer the interested reader to [CPS25, Section 4]. As argued in [CPS25], this relaxation seems necessary to unlock the possibility of devising canonical form functions that run in polynomial time, for most of the input matrices. We remark that existence of such a failure probability does not lead to any practical issue. First, for cryptographically relevant parameters, the failure probability is upper bounded by a constant which is always extremely small (e.g., approximately 2^{-80} for the NIST Category 1 parameters). Moreover, we emphasize that existence of a small failure probability does not lead to security issues. Indeed, as highlighted in Figure 1, the canonical form computation during commitment preparation is implemented via rejection sampling: upon a failure, we restart by sampling a completely new random monomial $\widetilde{\mu}$. Therefore, an attacker only learns that some CF computation of an unknown (rejected) transformation failed.

High Level Description of the Canonical Form Function. We now describe at a high level the Canonical Form function CF, which computes canonical forms of the non-IS part of generator matrices. For this, let $\mathbf{G} = (\mathbf{I}_k \mid \mathbf{A})$ and $\mathbf{G}^* = (\mathbf{I}_k \mid \mathbf{A}^*)$ be two generator matrices of two equivalent codes; it is shown in [CPS25] that, thanks to canonical forms, this implies $\mathbf{A}^* = \mathbf{Q}_r \mathbf{A} \mathbf{Q}_c$. Let us first assume an easy case $(\mathbf{Q}_r, \mathbf{Q}_c) = (\mathbf{P}_r, \mathbf{P}_c) \in S_k \times S_{n-k}$, e.g. the applied isometry is a permutation of rows and columns. As we shall see, the algorithm to compute canonical forms, in this case, will become a subroutine for the more general case in which the considered transformations are monomials. In this case we are able to compute

the canonical representative of $\mathbf{A}$ and $\mathbf{A}^*$ with SortCF below. First we sort each row by $\prec_{\mathbb{F}_q^k}$, which yield SortRows($\mathbf{A}$) = SortRows($\mathbf{P}_r\mathbf{A}$), as $\mathbf{P}_r$ only permutes a row, but does not alter it. To account for the column permutation $\mathbf{P}_c$, we sort the columns via SortCols according to $<_{\mathbb{F}_q^k}$. This leads in SortCols($\mathbf{A}$) = SortCols($\mathbf{A}\mathbf{P}_c$). Together with the previous results this yields that SortCF($\mathbf{A}$) = SortCF($\mathbf{P}_r\mathbf{A}\mathbf{P}_c$) = SortCF($\mathbf{A}^*$).

Algorithm 1: SortCF

Input: $\mathbf{G} \in \mathbb{F}_q^{k\times n-k}$ non-IS of Generator Matrix.
Output: $\mathbf{G}' \in \mathbb{F}_q^{k\times n-k}$.

```
G' ← SortRows(G)
return SortCols(G')
```

It follows that SortCF computes the canonical representative of generator matrices, in the case the applied isometries are just permutations.

Scaling Rows. For the next step we assume that $(\mathbf{Q}_r, \mathbf{Q}_c) = (\mathbf{Q}_r, \mathbf{P}_c) \in M_k \times S_{n-k}$. The trick to tackle the scaling of the rows via $\mathbf{Q}_r$, is to ensure that the sum of each row is 1. Let ScaleRow($\mathbf{v}$) = $(\sum_i \mathbf{v}_i)^{-1} \cdot \mathbf{v}$ and $\mathbf{v}' = \alpha\mathbf{v}$, for some $\alpha \in \mathbb{F}_q^*$. Now:

$$\text{SortRow}(\text{ScaleRow}(\mathbf{v}')) = \text{SortRow}(\text{ScaleRow}(\alpha\mathbf{v}))$$
$$\text{SortRow}((\sum \alpha\mathbf{v}_i)^{-1}(\alpha\mathbf{v}))\text{SortRow}((\sum \mathbf{v}_i)^{-1}\mathbf{v}) = \text{SortRow}(\text{ScaleRow}(\mathbf{v}))$$

Thus SimpleScaleCF simply computes ScaleRow for each row, and subsequently applies SortCF.

Algorithm 2: SimpleScaleCF

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 ∈ [k] do
    G[i :] ← ScaleRow(G[i :])
SortCF(G)
return true
```

It follows that SimpleScaleCF($\mathbf{Q}_r\mathbf{A}\mathbf{P}_c$) = SimpleScaleCF($\mathbf{A}$). This description ignores the fact, that $\sum_i \mathbf{v}_i$ could be equal to zero, thus the inverse does not exist. That is the reason why the canonical form function is a probabilistic algorithm, and canonical forms do not exist for every matrix. For a detailed analysis we refer to [CPS25].

Scaling Columns. For the last case we have $(\mathbf{Q}_r, \mathbf{Q}_c) \in M_k \times M_{n-k}$. Compared to the two previous cases, we additionally have to deal with the scaling of columns. To handle this, let $\mathbf{w}' = \alpha\mathbf{w}$, for some $\alpha \in \mathbb{F}_q^*$, a scaled column and additionally assume that $\mathbf{w} \in (\mathbb{F}_q^*)^k$. Now we can use the first coordinate $\mathbf{w}_0'^{-1}$ to scale $\mathbf{w}'$.

$$\mathbf{w}_0'^{-1} \cdot \mathbf{w}' = (\alpha\mathbf{w}_0)^{-1} \cdot (\alpha\mathbf{w}) = \mathbf{w}_0^{-1} \cdot \mathbf{w}$$

Thus we removed the scaling by α, by the same approach as with the rows, yielding the same as scaling $\mathbf{w}$ with $\mathbf{w}_0^{-1}$. The usage of the first dimension $\mathbf{w}_0$ as a scaling factor, seems arbitrary and indeed it is. For this define ScaleCol($\mathbf{w}, \beta$) = $\beta \cdot \mathbf{w}$. From the example above we know that ScaleCol($\mathbf{w}, \mathbf{w}_i^{-1}$) = ScaleCol($\mathbf{w}', \mathbf{w}_i'^{-1}$), $\forall i = 0, \dots k-1$. As our goal is to find the unique shortest canonical form representative, we need to enumerate all possible scaling/correcting factors $\mathbf{w}_i$ for all rows, to find the one which yields the minimal matrix according to $<_{\mathbb{F}_q^{k\times(n-k)}}$.

Algorithm 3: SIMPLECF

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: $\mathbf{B} \in \mathbb{F}_q^{k\times(n-k)}$, $\mathbf{M} \in \mathbb{F}_q^{k\times(n-k)}$: temporary buffer

```
/* set M to the largest matrix                                    */
```
$\mathbf{M} \leftarrow (q-1)^{k\times(n-k)}$
for $i \in [k]$ **do**
  for $j \in [k]$ **do**
    $\mathbf{B}[i:] \leftarrow \text{SCALECOL}(\mathbf{G}[j:], \mathbf{G}[i:]^{-1})$
  if $\textit{SIMPLESCALECF}(\mathbf{B}) \prec_{\mathbb{F}_q^{k\times(n-k)}} \mathbf{M}$ **then**
    $\mathbf{M} \leftarrow \mathbf{B}$
return $\mathbf{M}$

Putting all together the canonical form function is computed by first multiplying the columns, to remove any isometry applied to the columns. Next the rows are scaled to one, to remove any isometry applied to the rows. And finally any permutation on the rows and columns is removed via sorting. The uniqueness is ensured by enumerating all scalings for the columns to find the minimal resulting matrix.

3.2 Data Sizes

When computing data sizes, one has to take into accounts all the tweaks incorporated in the protocol. These include:

- *commitment recoverability*, so that the signature contains just the sequence of responses, and the challenge bits;
- *fixed-weight* challenge string, according to a skewed distribution, resulting in exactly w out of t non-zero challenges;
- *multiple statements*, expanding the public key from a single pair $(\mathbf{G}, \mathbf{G}')$ to a tuple $(\mathbf{G}_0, \mathbf{G}_1, \ldots, \mathbf{G}_s)$;
- *GGM trees* to efficiently transmit the $t - w$ seeds used to represent ephemeral challenges.

These lead to the following public key and signature sizes.

Public Key Size. Instantiating LESS with s statements in the public key leads to a challenge space of size $s+1$, and hence a soundness error of $1/(s+1)$ for a single execution of the Sigma protocol. While this reduces the number of necessary parallel repetitions to achieve a low soundness error, it increases the public key size (in bytes) to

$$\text{SIZE}_{\text{pk}} = (s-1) \cdot \left(\lceil k(n-k)\lceil \log_2(q)\rceil/8\rceil + \lceil n/8\rceil\right) + \lambda/8. \tag{1}$$

Note that $\mathbf{G}_0$ is generated from a seed and all $\mathbf{G}_i$, $i = 1, \ldots, s$ are assumed to be in systematic form $\mathbf{G}_i = (\mathbf{I}_k \mid \mathbf{A}_i)$. Additionally to the non-systematic part for every $\mathbf{G}_i$ the pivot columns are stored, which requires an additional n bits.

Signature Size. The protocol responses, after the introduction of canonical forms, consist only of the partial permutations that define the indices of the first k columns, rather than the full monomial maps. Moreover, for challenge 0 this size-k subset is derived from a seed of length λ bits. Whenever the challenge is non-zero, the encoding of such a subset can be realized using n bits.[3] Furthermore, we ensure $n \approx 2\lambda$ to meet the well-known lower bound for non-ephemeral responses in group action based Sigma protocols.

[3]Precisely only $\log \binom{n}{k} \leq n$ bits are required, however, for simplicity and efficiency we use a slightly suboptimal encoding.

If w out of t parallel protocol executions have a challenge $\mathsf{ch} \neq 0$ there are w non-ephemeral responses of size $n \approx 2\lambda$ and $t - w$ responses which are formed by PRNG seeds. For the transmission of the $t - w$ seeds a GGM tree is used, which reduces the communication cost whenever $w < t/2$. Namely, an upper bound for the number of seeds that have to be transmitted using a GGM tree is given by

$$w \log_2(t/w) + u - 1, \tag{2}$$

where $u \leq \log_2(t)$ is the weight of t in binary. Note that when $t = 2^d$ for an integer d the upper bound yields $w \log_2(t/w)$. In total the signature size in bytes amounts to

$$\textsc{Size}_{\mathtt{sig}} = \underbrace{\lambda/2}_{\text{cmt \& salt}} + \underbrace{w \cdot \lceil n/8 \rceil}_{\text{non-eph. resp.}} + \underbrace{\min\{\lfloor w \log_2(t/w) + u - 1 \rfloor, t - w\} \cdot \lambda/8}_{\text{ephemeral responses}} + \underbrace{1}_{\#\text{seeds}} . \tag{3}$$

4 Optimizations

In this section we describe various improvements aimed at obtaining a competitive implementation of LESS. We begin by addressing canonical form computation, via a variable-time version of the CF function. We then introduce a blinding technique in Section 4.3, which allows to implement large parts of the scheme, including the canonical form computation, in variable time without leaking secret information. In total, only the RREF, Blinding and the sampling of random permutations need to be computed in constant time.

4.1 RREF

The RREF algorithm is a direct instantiation of Gaussian elimination. It systematically transforms a given matrix into row echelon form through a series of elementary row operations: row swapping, scalar multiplication, and adding or subtracting multiples of rows. In each round, the first step is finding a suitable pivot element. This is then used in subsequent row operations to ensure all other elements in the pivot columns are zero. Once the RREF operation completes, all pivot columns will contain a single non-zero entry.

4.1.1 Pivot Reusage

The RREF operation in LESS is not applied to a randomly generated matrix. The input is a matrix $\mathbf{G}_0$, which was already in RREF but a monomial map was applied. Since the application of a monomial map is equivalent to permuting and scaling the columns of $\mathbf{G}_0$, after applying the map, the k columns of $\mathbf{G}_0$ that were previously pivot columns 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. For LESS parameters where $k = n/2$, 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 almost 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[4] 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 an average of only 1 non-constant-time operation before the signature limit is reached. It is also worth noting that the variance in latency is related only to the ephemeral monomial, not the long-term secret monomial.

We further extend the technique by avoiding to rescale the pivot element of already reduced columns. This is possible because the canonical form computation is independent of the non-zero entries in the identity part of the generator matrix.

4.1.2 Variable-Time Implementation

The reuse of pivots requires only minor changes to the RREF algorithm. First, the locations of the previous pivot columns must be determined. The pivot column location is tracked using a n-entry Boolean array where a 1 signifies the column is a pivot column. We determine the new location of these pivots by applying the same monomial permutation that was applied to the generator matrix to this array. Then, prior to performing the row scaling operation we check if the current pivot column can be reused. If so, the row scaling and row reductions steps are skipped. Pivot elements are only reused if the pivot reuse limit has not been reached.

In some circumstances, a reusable pivot may be corrupted. For example, if $(\mathbf{G}[0,1])$ is a reusable pivot, but $(\mathbf{G}[0,0])$ is used as the pivot element for the reduction of the first column, then the zero elements of $(\mathbf{G}[:1])$ will be overwritten during the row reduction step. To limit the likelihood of reusable pivots being corrupted, a pre-processing step is performed where row swaps are used to position reusable pivots on the diagonal. In this example, row 0 and 1 would be swapped so that the reusable pivot on column 1 is in position$(\mathbf{G}[1,1])$. Thus the element would only be corrupted if $(\mathbf{G}[0,0]) == 0$ and $(\mathbf{G}[1,0]) \neq 0$. The RREF algorithm with pivot reuse is described in Algorithm 4.

We were also able to improve the RREF implementation for the VERIFY algorithm, by using a variable-time scalar multiplication via table lookups. More on this in Section 4.4.

4.2 Canonical Form

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 $\mathbf{v} \in \mathbb{F}_q^n$. In order to get a practical canonical form we apply the following optimizations. The core idea of the newly proposed canonical form computation is to skip as many computational steps as possible, for which we know beforehand that they do not lead to anything useful. The goal of the canonical form is to compute the "shortest" representative, according to $\prec_{\mathbb{F}_q^{k\times(n-k)}}$, for the input matrix $\mathbf{G}$. Below, we present our new CF algorithm, with early abort.

We first we keep track of the rows with no zeros in it, as for rows with zeros, line 7 would not be defined. With this information we can reduce the number of SCALECF calls from k down to the number of rows not containing a zero. Additionally, we scan the input matrix $\mathbf{G}$ for the number of zeros per row. As we know that these rows with the most zeros will be the shortest according to $\prec_{\mathbb{F}_q^{k\times(n-k)}}$, once scaled. Let us denote this set with $|J|$. Then, instead of scaling the full matrix $\mathbf{G}$, we only scale the submatrix $\mathbf{B} \in \mathbb{F}_q^{|J|\times(n-k)}$. But instead of fully computing SCALECF, we can do better. We want to skip the expensive computation of SORTCF, more precisely the computation of SORTCOLS.

[4]The signature limit refers to the number of signatures that can safely be generated with a single keypair.

Algorithm 4: RREFPivotReuse

Input: $\mathbf{G} \in \mathbb{F}_q^{k \times n}$, p : an array which specifies if a column was a previous pivot of $\mathbf{G}$, l : number specifying how many pivot elements may be reused
Output: $\mathbf{G} \in \mathbb{F}_q^{k \times n}$ in RREF
Data:

```
   /* Swap previous pivot elements to the diagonal                                   */
 1 rc ← 0
 2 for col ∈ [k] do
 3 |   if p[col] = 1 and rc < l then
   |   |   /* Find row containing reusable pivot element                           */
 4 |   |   for row ∈ [k] do
 5 |   |   |   if G[row col] ≠ 0 then
 6 |   |   |   |   i ← row
 7 |   |   G[col :], G[i :] ← G[i :], G[col :]
 8 |   |   rc ← rc + 1
 9 rc ← 0
10 for row_to_reduce ∈ [k] do
11 |   i, j ← 0, 0
12 |   for j ∈ [row_to_reduce, n] do
13 |   |   for i ∈ [row_to_reduce, k] do
14 |   |   |   if G[i, j] > 0 then
15 |   |   |   |   go to 16
16 |   if row_to_reduce ≠ i then
   |   |   /* Row swap during pivot search corrupts reusable pivot in column i    */
17 |   |   p[i] = 0
   |   /* Swap the rows at row_to_reduce with i                                     */
18 |   G[row_to_reduce :], G[i :] ← G[i :], G[row_to_reduce :]
   |   /* If reusable pivot, skip                                                  */
19 |   if p[row_to_reduce] ≠ 1 and rc < l then
   |   |   /* Reduce the pivot row                                                */
20 |   |   G[row_to_reduce :] ← G[row_to_reduce, j]^{-1} · G[row_to_reduce :]
   |   |   /* Reduce the rest                                                      */
21 |   |   for row ∈ [k] \ row_to_reduce do
22 |   |   |   G[row :] ← G[row :] − G[row, j] · G[row_to_reduce :]
23 |   else
24 |   |   rc ← rc + 1
25 return G
```

Thus SubScaleCFPreprocess, scales the rows normally, but at the same time keeps track of the shortest row $\mathbf{L}$ and returns this row. Note that the $\prec_{\mathbb{F}_q^{n-k}}$ comparison in line 10 implicitly computes SortRows. Additionally, it is important to note that SortCols does not change $\mathbf{L}$ if it is the shortest row, among all inputs. And finally we fully compute the scaling of the full matrix $\mathbf{G}$ via ScaleCF only once for the previously scanned row r, which yields the shortest row. Thus we reduced the number of ScaleCF calls from k down to 1. Note that it happens with a small probability, that the scaling via the r-th row does not yield a valid canonical form. In this case we fall back to the original version.

Overall this improvement yields a speedup of 60% for LESS-252 and LESS-400 and 36% for LESS-548. Note that for this benchmark both version of the canonical form utilized the same arithmetic as well as the same sorting subroutines. This is to ensure that Table 1 solely reflects the algorithmic improvements of the canonical form function.

Table 1: Timings (Kcycles) for the computation of CF and Improved CF. Average of 2^{15} runs, measured on a AMD Ryzen 5 7600X.

	LESS-252	LESS-400	LESS-548
CF	4300	8500	9700
Improved CF	73	138	255

Algorithm 5: IMPROVED CF

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: $\mathbf{B} \in \mathbb{F}_q^{k \times (n-k)}, \mathbf{L} \in \mathbb{F}_q^{n-k}$: temporary buffer.

$J \leftarrow \{i \,|\, \mathsf{max}(\mathsf{cnt}_0(\mathbf{G}[i])), \forall i \in [k]\}$
$Z \leftarrow \{i \,|\, \mathsf{cnt}_0(\mathbf{G}[i]) > 0, \forall i \in [k]\}$
$\mathbf{L} \leftarrow (\mathbf{q}-\mathbf{1})^{n-k}$
$r \leftarrow 0$
for $i \notin Z$ **do**
  for $j \in [|J|]$ **do**
    $\mathbf{B}[j:] \leftarrow \mathbf{G}[J[j]:] \cdot \mathbf{G}[i:]^{-1}$
  $b, \mathbf{L} \leftarrow$ SUBSCALECFPREPROCESS$(\mathbf{B}, \mathbf{L}, |J|)$
  if b **then**
    $r \leftarrow i$;
for $j \in [k]$ **do**
  $\mathbf{B}[j:] \leftarrow \mathbf{G}[J[j]:] \cdot \mathbf{G}[r:]^{-1}$
if SCALECF$(\mathbf{B}) = \mathsf{false}$ **then**
  return CF$(\mathbf{G})$
return $\mathbf{B}$

Algorithm 6: SUBSCALECFPREPROCESS

Input: $\mathbf{G} \in \mathbb{F}_q^{z \times (n-k)}$, $\mathbf{L} \in \mathbb{F}_q^{n-k}$, $z \in \mathbb{Z}$.
Output: true/false.
Data: $\mathbf{T} \in \mathbb{F}_q^{n-k}$: temporary buffer.

$r \leftarrow \mathsf{false}$
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**
    $s = \sum_{j=0}^{n-k-1} \mathbf{G}[i,j]^{-1}$
  if $s = 0$ **then**
    continue
  $\mathbf{T} \leftarrow s \cdot \mathbf{G}[i:]$
  if $\mathbf{T} \prec_{\mathbb{F}_q^{n-k-1}} \mathbf{L}$ **then**
    $\mathbf{L} \leftarrow \mathbf{T}$
    $r \leftarrow \mathsf{true}$
return $r, \mathbf{L}$

In the following sections we introduce further improvements to the subroutines used within the CF computation.

4.2.1 Histogram on Sorting Rows

As already described in a previous section, the runtime of canonical form computation depends heavily on the SORT subroutine. A possible constant-time instantiation of this algorithm is given in Algorithm 7. To further improve on this we propose to compute a histogram instead of sorting vectors. A histogram $\mathbb{F}_q^{n-k} \to \mathbb{Z}^q$ counts the number of occurrences of each symbol within the vector, see Algorithm 8. Note that a histogram is still enough to compare elements $\mathbf{v}, \mathbf{w} \in \mathbb{F}_q^n$ via the ordering $\mathbf{v} \prec_{\mathbb{F}_q^n} \mathbf{w}$.

Algorithm 7: Sort

Input: $\mathbf{v} \in \mathbb{F}_q^n$
Output: $\mathbf{v} \in \mathbb{F}_q^n$ sorted
Data: $C \in \mathbb{F}_{2^{32}}^q$: temporary buffer

$C \leftarrow \mathbf{0}^q$
for $i \in [n]$ **do**
  $C[\mathbf{v}[i]] \leftarrow C[\mathbf{v}[i]] + 1$
$j \leftarrow 0$
for $i \in [q]$ **do**
  while $C[i] > 0$ **do**
    $\mathbf{v}_j = i$
    $C[i] \leftarrow C[i] - 1$
    $j \leftarrow j + 1$
return $\mathbf{v}$

Algorithm 8: Histogram

Input: $\mathbf{v} \in \mathbb{F}_q^n$
Output: $\mathbf{w} \in \mathbb{Z}^q$ sorted

$\mathbf{w} \leftarrow \mathbf{0}^q$
for $i \in [n]$ **do**
  $\mathbf{w}[\mathbf{v}[i]] \leftarrow \mathbf{w}[\mathbf{v}[i]] + 1$
return $\mathbf{w}$

Another positive side-effect is that a histogram acts like a compression function, which reduces the size of each element to compare from k to q. Unfortunately, it is not possible to optimize the computation of a histogram via vectorization for all proposed parameters. Only LESS-548 profits from a parallel histogram computation. In this case the input vector $\mathbf{v}$ is processed in multiples of 4, which are put into 4 different buckets. In a subsequent step, these 4 buckets are summed together via vector instructions.

Table 2: Number of cycles to compute Sort and Histogram. Average of 2^{20} runs, measured on a AMD Ryzen 5 7600X.

	LESS-252	LESS-400	LESS-548
Sort	1280	1777	2135
Histogram	600	860	1154

4.2.2 Sorting Columns

The second most time-consuming subroutine in the computation of a canonical form is SortCols. This function takes the columns of a matrix and sorts them according to the ordering $\mathbf{w} \leq_{\mathbb{F}_q^n} \mathbf{w}'$ with $\mathbf{w}, \mathbf{w}' \in \mathbb{F}_q^n$. As all matrices in our implementation are stored in a row-major form, accessing columns is rather inefficient. Thus we first transpose an input matrix, apply SortRows, and subsequently we are transposing the matrix again. Note that SortRows does not need to apply a Sort function for each row. Our 3 proposed optimized implementations leverage the respective instructions of the underlying CPUs. In case of AVX2 and AVX512, specialized functions are provided, which transpose 32×32 (resp. 64×64) submatrices via the `_mm{256|512}_unpack{lo|hi}_epi{8|16|32|64}` instructions. The NEON implementation implements an optimized transpose of a 16×16 matrix via the `vtrn{1|2}q_ u{8|16|32|64}` instructions.

4.3 Blinding

As demonstrated, the variable-time implementation of canonical forms (see Algorithm 5) is considerably faster than its constant-time version. To justify its use from a side-channel security perspective also during signing, we introduce a *blinding* procedure which is applied 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. Before we analyze this technique, let us clarify the following points. The application of monomial maps does not change the result of a canonical form,

e.g. $\mathrm{CF}(\mathbf{M}_l \cdot \mathbf{G} \cdot \mathbf{Q}_r) = \mathrm{CF}(\mathbf{G})$, hence it does not alter the signature. Additionally, it lifts previously secret data, the non-Information Set of the RREF of $\mathbf{G}$, to be public data. Hence, it removes the needs for constant-time code for large amount of the code, which greatly simplifies it. Justification for this follows. The cost for this are two additional monomial multiplications per round, which only add up to $< 1\%$ of the total runtime.

Algorithm 9: 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: $\mathbf{Q}_l \in \mathbb{F}_q^{k \times (n-k)}$, $\mathbf{Q}_r \in \mathbb{F}_q^{(n-k) \times k}$

1 $\mathbf{Q}_r \leftarrow$ SampleMonomial(state)
2 $\mathbf{Q}_l \leftarrow$ SampleMonomial(state)
3 **return** $\mathbf{Q}_l \cdot \mathbf{G} \cdot \mathbf{Q}_r$

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

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

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.

4.4 Arithmetic

We provide 4 different implementations of LESS: a reference C implementation, and 3 optimized versions for Intel chips supporting the AVX2/AVX512 instruction set and ARM chips supporting the NEON instruction set. Besides the optimization described in Section 4.2.2, these implementations primarily optimize the arithmetic, which is crucial for the efficient implementation of LESS.

4.4.1 Reference Arithmetic

Let us first start with a description of the implementation of the arithmetic used in the reference code. Each $\mathbb{F}_q$ elements is represented in a single byte, thus additions do not overflow. The reduction is implemented using the following fact. Every number $x \in \mathbb{Z}_{2^{16}}, x = \sum_{i=0}^{15} x_i \cdot 2^i$ can be reduced $\mod 127$ by computing $(\sum_{i=7}^{15} x_i \cdot 2^{i-7}) + (\sum_{i=0}^{6} x_i \cdot 2^i)$ as $2^i = 2^{i-7} \mod 127$ for $i \geq 7$. Thus the full reduction can be implemented as shown in Figure 2.

```
1 uint8_t t = (x >> 7) + (x & q)
2 uint8_t sub_q = t - q;
3 uint8_t mask = -(sub_q >> 7);
4 return (mask & q) + sub_q;
```

Figure 2: Implementation of a constant time reduction in plain C. The value x is reduced mod q, if $x \leq 127^2$. Note that the size of the input type of x is 2 bytes.

4.4.2 AVX2 Arithmetic

The core idea of the vectorized AVX2 arithmetic is to combine the reference C code with the `_mm256_blendv_epi8`$(\mathtt{a}, \mathtt{b}, \mathtt{m})$ instruction. This instruction operates on 32×8 bit elements and returns in each limb either $\mathtt{b}[i]$ if the highest bit in $\mathtt{m}[i]$ is set, else $\mathtt{a}[i]$. With this we can simply implement a vectorized addition or subtraction as shown in Figure 3 respectively Figure 4. Note that in case no overflow/underflow in line 1 happens, e.g. $\mathtt{a}[i] + \mathtt{b}[i] < 127$, respectively $\mathtt{a}[i] - \mathtt{b}[i] \geq 0$, the `_mm256_blendv_epi8` instruction will correctly select $\mathtt{x1}[i]$ as the output value. In case an overflow/underflow happens, note the signed instruction is used to compute the correction factor `x2`, the correction factor is instead returned. Note that in all algorithms `c7f` = `_mm256_set1_epi8(0x7f)`.

```
1 __m256i x1 = _mm256_add_epi8(a, b);
2 __m256i x2 = _mm256_sub_epi8(x1, c7f);
3 return _mm256_blendv_epi8(x2, x1, x2);
```

Figure 3: AVX2 addition $\mathtt{a} + \mathtt{b}$

```
1 __m256i x1 = _mm256_sub_epi8(a, b);
2 __m256i x2 = _mm256_add_epi8(x1, c7f);
3 return _mm256_blendv_epi8(x1, x2, x1);
```

Figure 4: AVX2 subtraction $\mathtt{a} - \mathtt{b}$

One could replace `_mm256_blendv_epi8` with `_mm256_min_epu8`, as the first is one of the heavy instructions, consisting of many micro operations. This will be useful for older CPUs, like Intel SandyBridge or Intel Haswell, on which `_mm256_min_epu8` is faster. In our test system containing a Ryzen 7600X, the `_mm256_blendv_epi8` instruction actually performs better.

For the scalar multiplication, note that Intel does not provide an AVX2 8-bit multiplication, but only a 16-bit vector multiplication. Thus, we extend every 8-bit limb into a 16-bit limb, which is done via the `_mm256_cvtepu8_epi16` instruction. Next, we compute the multiplication as described in Figure 2 but for 32 elements in parallel. Subsequently, the elements are again compressed from 16-bit limbs into 8-bit limbs via `_mm256_packs_epi16`. The subsequent reduction is computed as in the addition and subtraction.

```
 1 a_lo = _mm256_mullo_epi16(a_lo, b);
 2 a_hi = _mm256_mulhi_epi16(a_hi, b);
 3 __m256i v1 = _mm256_srai_epi16(a_lo, 7);
 4 __m256i v2 = _mm256_srai_epi16(a_hi, 7);
 5 __m256i w1 = _mm256_packs_epi16(a_lo&c007f, a_hi&c007f);
 6 __m256i w2 = _mm256_packs_epi16(v1, v2);
 7 v1 = _mm256_add_epi8(w1, w2);
 8 v2 = _mm256_permute4x64_epi64(v1, 0b11011000);
 9 w1 = _mm256_sub_epi8(v2, c7f);
10 return _mm256_blendv_epi8(w1, v2, w1);
```

Figure 5: AVX2 scalar mutliplication $\mathtt{a} \cdot b$, where `a` is extended into `a_lo` and `a_hi` via `_mm256_cvtepu8_epi16`. Thus `a_lo` (`a_hi`) contains the lower (upper) 16 8-bit limbs extended to 16-bit limbs. The variable `c007f` is set to `_mm256_set1_epi16(0x7f)`.

4.4.3 AVX512 Arithmetic

Newer x86 CPUs even support AVX512 which introduced 512-bit vectors and instructions. This extension finds slowly broad adoption in consumer hardware, as such it was introduced in the ZEN4 architecture in 2022 by AMD. Whereas Intel introduced AVX512 in their Skylake-X and Cannon Lake processors around 2018. In contrast to AVX2, which is only split into two instruction sets (AVX and AVX2), AVX512 is split into 19 different instructions sets (AVX512F (Foundation), AVX512CL (Conflict Detection), ...). Additionally not all CPUs support the same subset of AVX512 instruction sets. We assume for our implementation the support of AVX512VBMI (Vector Byte Manipulation Instructions) which was introduced with Intels Cannon Lake, or AMDs ZEN4 architecture.

The intrinsics shown in the previous chapter can be easily replaced with their AVX512 counterpart to get an implementation to work on 512-bit vectors. Additionally, we provide a variable-time implementation of a scalar multiplication, which is used to speed up the canonical form computations as well as the Gaussian elimination in the verification. The idea is to use the `_mm512_permutex2var_epi8(a, idx, b)` instruction, which computes a table lookup. The two inputs `a` and $\mathtt{b} \in \mathbb{F}_{2^8}^{64}$ are interpreted as a 128 byte lookup table `t`, where the lowest address is `a`$[0]$ and the highest `b`$[63]$. Finally the instruction returns a vector $\mathtt{c} \in \mathbb{F}_{2^8}^{64}$ with $\mathtt{c}[i] = \mathtt{t}[\mathtt{idx}[i]], \forall i = 0, \ldots, 63$. Note that only the lower 7 bits in each 8 bit limb of `idx` are used.

```
1 __m512i table1 = _mm512_load_si512(__gf127_lookuptable + 128*a +  0);
2 __m512i table2 = _mm512_load_si512(__gf127_lookuptable + 128*a + 64);
3 return _mm512_permutex2var_epi8(table1, x, table2);
```

Figure 6: AVX512 scalar multiplication of $\mathbf{x} \cdot a, \mathbf{x} \in \mathbb{F}_q^{64}, a \in \mathbb{F}_q$. `__gf127_lookuptable` is a lookup table, which contains $i \cdot j \mod q$ at position $i \cdot 128 + j$. Each row in the table is padded with zeros to a size of 128 for easier access.

4.4.4 Neon Arithmetic

ARM introduced its SIMD solution in 2004 with the ARMv7 Instruction Set Architecture (ISA), which was extended for ARMv8 to process 128-bit registers. This ISA is implemented by Apple on the M1 Max chip, which is our test system.

The arithmetic proposed in Section 4.4.2 can be translated straightforwardly into the 128-bit NEON world by replacing the instructions with their corresponding counterparts. In the same fashion as Section 4.4.3, we propose a variable-time table lookup-based scalar multiplication to speed up the Gaussian elimination. In contrast to the AVX512 instructions, NEON does not support a 128-byte table lookup instruction but only a 64-byte instruction `vqtbx4q_u8`$(\mathtt{z}, \mathtt{t}, \mathtt{idx})$. Where $\mathtt{t} \in \mathbb{F}_{2^8}^{64}$ is the lookup table, $\mathtt{idx} \in \mathbb{F}_{2^8}^{16}$ the indices and $\mathtt{z} \in \mathbb{F}_{2^8}^{16}$ is a fallback value which is returned if any of the indices is ≥ 64. In total this `vqtbx4q_u8` returns $\mathtt{c} \in \mathbb{F}_{2^8}^{16}$, with $\mathtt{c}[i] = \mathtt{idx}[i] < 64 \,?\, \mathtt{t}[\mathtt{idx}[i]] : \mathtt{z}[i], i = 0, \ldots, 15$. We extend this to a full 128-byte table lookup in two steps as shown in Figure 7. First a lookup with the full `x` as table indices is performed. Note that for every entry in `x` which is ≥ 64 a $\mathbf{0} \in \mathbb{F}_{2^8}$ is returned. In the second step we perform another table lookup, but using $\mathtt{x}[i] - 64$ as the indices. Note that for every position in `x`, previously exceeding 64, is now smaller, and visa versa. Thus, an xor of both results gives the final solution.

```
uint8x16_t zero = vdupq_n_u8(0);
uint8x16_t m = vdupq_n_u8(64);
uint8x16x4_t table1 = vld1q_u8_x4(__gf127_lookuptable + 128*a +  0);
uint8x16x4_t table2 = vld1q_u8_x4(__gf127_lookuptable + 128*a + 64);
uint8x16_t ret1 = vqtbx4q_u8(zero, table1, x);
return vqtbx4q_u8(ret1, table2, vsubq_u8(x, m));
```

Figure 7: NEON scalar multiplication of $\mathbf{x} \cdot a, \mathbf{x} \in \mathbb{F}_q^{64}, a \in \mathbb{F}_q$. `__gf127_lookuptable` is a lookup table, which contains $i \cdot j \mod q$ at position $i \cdot 128 + j$. Each row in the table is padded with zeros to a size of 128 for easier access.

4.5 Sampling Random Permutations

In order to securely implement the LESS protocol, it is important that the ephemeral monomial map $\tilde{\mu}$ is sampled in constant time. Whereas a random element of $\mathbb{F}_q$ can be computed via rejection sampling, sampling a random permutation in constant time is a more delicate task. For this reason we employ a Batcher [Bat68] Odd-Even Permutation Network for non power of twos as described in [Knu73] Algorithm 5.2.2M. While these networks are normally used to sort a list of integers in constant time, we are using them to randomly permute all elements. Furthermore, [Ber20] showed that this yields in the end a random permutation. This is done by leveraging a normal sorting network, but instead of comparing and swapping each node, we flip a coin if we swap the two inputs. Note the the Batchers Odd-Even Permutation Network needs in total $n/4 \cdot \lfloor \log_2(n) \rfloor \cdot (\lfloor \log_2(n) \rfloor + 1)$ bits of randomness.

Algorithm 10: RandomPermutation

Input: $\mathbf{v} \in \mathbb{Z}^n$, state of an already initialized XOF
Output: $\mathbf{v} \in \mathbb{Z}^n$ permuted

$R \leftarrow \text{XOF}(\text{state}, n/4 \cdot \lfloor \log_2(n) \rfloor \cdot (\lfloor \log_2(n) \rfloor + 1))$
$c, t \leftarrow 0, \lceil \log_2(n) \rceil - 1$
for $p \in [2^t, 2^{t-1}, \ldots, 1]$ **do**
  $r, d \leftarrow 0, p$
  for $q \in [2^t, 2^{t-1}, \ldots, p]$ **do**
    for $i \in [n - d]$ **do**
      if $i \wedge p$ *and* $R[c]$ **then**
        $\texttt{swap}(\mathbf{v}[i], \mathbf{v}[i + d])$
        $c \leftarrow c + 1$
    $r, d \leftarrow p, q - p$
return $\mathbf{v}$

In Algorithm 10 `state` describes the already initialized state of a XOF. Putting it all together yields Algorithm 11 to sample random monomials in constant time. Note that SampleUniformNumber is a function which returns a random number $a \in \mathbb{Z}_{q-1}$ using the same state as RandomPermutation.

Algorithm 11: SampleMonomial

Input: state
Output: μ : a monomial matrix represented by two lists of length n, called $\mu.\pi$ and $\mu.u$, containing the permutation and coefficients.

$\mu.\pi \leftarrow [n]$
for $i \in [n]$ **do**
  $\mu.u[i] \leftarrow 1 + \text{SampleUniformNumber}(\text{state}, q - 1)$
$\mu.\pi \leftarrow \text{RandomPermutation}(\mu.\pi, \text{state})$
return μ

5 Performance

In this section we propose multiple instantiations of LESS, offering different performance profiles. We provide detailed insights in the parameter selection process and comprehensive benchmarks for all suggested instantiations.

5.1 Parameters

The introduction of canonical forms leads to a new and improved performance profile of LESS: the signature size is significantly reduced at the potential expense of the canonical form computation. While the cumulative improvements made to the implementation in Section 4 reduce the computational overhead to a bare minimum, LESS still offers new performance trade-offs, necessitating a re-parameterization process. In this section we detail the selection of optimal problem and protocol parameters, and propose different instantiations of LESS.

Problem Parameters. The security of LESS rests on the hardness of the Linear Equivalence Problem (LEP) from Definition 1. With respect to the parameters of this problem, we resort to the ones suggested in LESS NIST Round 1 submission (see Table 3).

Table 3: Suggested LEP parameters for LESS instantiations.

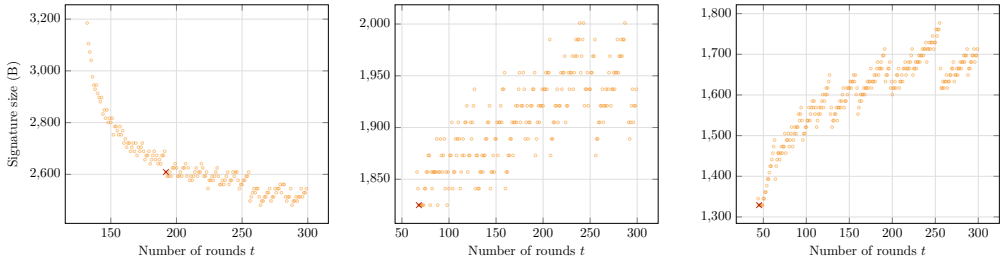
	n	k	q	security parameter λ
NIST-1	252	126	127	128
NIST-3	400	200	127	192
NIST-5	548	274	127	256

Selection Rationale. In the context of the NIST PQC standardization effort security is defined with respect to AES: a parameter set is said to match the security definition of Category 1, 3 or 5, if the scheme instantiated with those parameters is at least as hard to break as AES-128, -192 or -256 respectively.

The suggested LESS parameters from Table 3 have been selected based on an established lower bound for attacks that are based on finding low-weight codewords in the corresponding codes. More precisely parameters are chosen such that this lower bound yields a complexity of 2^λ for $\lambda \in [128, 192, 256]$ for NIST security categories 1, 3 and 5 respectively.
Recent Attack Improvements. Together with introducing the concept of canonical forms, Chou, Persichetti and Santini [CPS25] also proposed a new meet-in-the-middle style attack on LEP running in time roughly $T_{\mathrm{CF}} = \sqrt{\binom{n}{k}}$. Note that for the parameters from Table 3 we have $T_{\mathrm{CF}} \approx 2^{\frac{n}{2}} \approx 2^\lambda$. Observe that this is no coincidence, as in another recent work, Budroni, Esser, Franch and Natale [BEFN25] have shown that the lower bound for attacks based on low-weight codeword finding converges from below to T_{CF} for growing field sizes. Therefore the chosen parameters are inherently resistant against this new meet-in-the-middle attack. Furthermore, in the same work, the authors propose an improved attack that, in certain cases, comes close to the known lower bound. However, since it still relies on finding low-weight codewords, it remains fundamentally constrained by this lower bound and does not invalidate the security assumptions made.

Protocol Parameters. The Sigma protocol from which LESS is constructed involves further parameters that affect the performance of the protocol and, hence, the metrics of the signature scheme. Those parameters include the number of generator matrices in the public key s, as well as the Hamming weight of the challenge string w (see Section 3). The selection of these parameters allows for interesting trade-offs between signature size, computational efficiency and the size of the public key.

Parameter Selection. As given by Equation (1), the public key size grows linearly with the number of equivalence relations s in the public key. On the upside, a larger s leads to a reduced number of parallel repetitions t (compare to Equation (2)) and hence a smaller signature size. Furthermore, the fewer number of rounds also leads to faster signing and verification. For NIST Category 1 parameters we consider $s \in [2, 4, 8]$, while for NIST Category 3 and 5 we restrict to $s \in [2, 4]$. The choice of $s = 2$ leads to the smallest public key size, while the choice of $s = 8$ in case of NIST-1 is the largest choice for s such that $\text{Size}_{\texttt{sig}} + \text{Size}_{\texttt{pk}} < 100\text{KB}$ — two scenarios relevant for practical applications. The choice $s = 4$ is meant to offer intermediate instantiations with small signatures and improved timings, but moderately large public keys.



(a) $s = 2$ (b) $s = 4$ (c) $s = 8$

Figure 8: Signature size as function of the number of rounds for NIST-1 parameters.

For each security category and choice of s we now determine the smallest challenge weight w such that for a given amount of rounds t the soundness criterion (see Equation (2)) is satisfied. Figure 8 illustrates the signature sizes of the possible instantiations for NIST Category 1 parameters as a function of the number of rounds. On the left, instantiations for the choice of $s = 2$ are depicted. Here we observe that the signature size is decreasing in the number of rounds. This is related to the GGM tree and the optimal value of w: Since the optimal w is found to be $w < t/2$ it follows that the GGM tree reduces the amount of seeds transmitted to (roughly) $w \log(t/w) < t - w$. Among all instantiations, we choose $t = 192$ leading to balanced profile with a signature size of 2609 Bytes, depicted as a red cross in Figure 8. On the other hand for $s \in [4, 8]$ the optimal challenge weight satisfies $w > t/2$, excluding improvements due to the GGM tree and, hence, leading to optimal signatures for minimal amount of rounds, again marked as red crosses in the Figure 8.

For NIST Category 3 and 5 the picture looks similar, with $s = 2$ allowing for a trade-off between the number of rounds and the signature size, while for $s = 4$ signature size optimizes for a small number of rounds. We provide the full parameters and the resulting public key and signature sizes in Table 4.

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

NIST Cat.	**Parameter Set**	**Code Params** n	k	q	**Prot. Params** t	w	s	**pk** (B)	**sig** (B) Worst	Avg
1	LESS-252-192	252	126	127	192	36	2	13940	2609	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	6353	5585
	LESS-400-102				102	61	4	105174	4131	3867
5	LESS-548-345	548	274	127	345	75	2	65793	10712	9464
	LESS-548-137				137	79	4	197315	7436	7116

Table 5: Timings (Mcycles) for the optimized AVX2 implementation of LESS. Values collected on an AMD Ryzen 5 7600X. Clock cycle values collected via `rtdscp`, as averages of 2^{14} runs.

NIST Cat.	Parameter Set	Original CF			Improved CF		
		Keygen	Sign	Verify	Keygen	Sign	Verify
1	LESS-252-192	0.9	1001.8	970.7	0.7	117.0	106.8
	LESS-252-68	1.7	353.5	351.9	1.6	41.4	39.3
	LESS-252-45	3.6	233.5	233.3	3.6	26.5	26.3
3	LESS-400-220	1.1	2135.6	2162.7	1.0	325.7	317.1
	LESS-400-102	4.8	1001.7	970.7	4.8	162.5	152.9
5	LESS-548-345	4.2	5026.9	4960.3	4.1	1234.9	1172.0
	LESS-548-137	10.7	1981.3	1973.3	10.6	485.8	481.1

Table 6: Timings (Mcycles) for the optimized AVX512 and NEON implementations of LESS. Values collected on an AMD Ryzen 5 7600X/Apple M1, as an averages of 2^{14} runs.

NIST Cat.	Parameter Set	NEON			AVX512		
		Keygen	Sign	Verify	Keygen	Sign	Verify
1	LESS-252-192	1.0	181.6	164.6	0.8	112.2	80.6
	LESS-252-68	2.8	64.4	59.5	2.0	40.2	29.7
	LESS-252-45	6.4	52.8	40.§	4.5	25.4	19.7
3	LESS-400-220	1.0	687.9	644.5	2.2	341.9	250.2
	LESS-400-102	4.8	318.8	300.1	5.9	156.6	119.0
5	LESS-548-345	4.1	2861.8	2742.0	4.6	1224.4	923.0
	LESS-548-137	10.6	1149.0	1098.0	12.5	479.2	380.5

Comparing to earlier parameters suggested in [CPS25], we are able to decrease the number of rounds by a factor ranging from 1.3 up to 3.9 for $s = 2$, at the cost of a marginally increased signature between 5% to 12%. With respect to $s = 4$, we actually decrease the signature size slightly (1-5%) while reducing t by a factor between 3.7 and 8.8.

5.2 Timings

Table 5 reflects the runtime improvements of the new variable time canonical form implementation deployed in the LESS signature scheme. The improvements range from ×9 for Category 1 to ×5 for Category 5.

Table 7 contains the public key size, signature size, as well as performance metrics for all NIST Round 2 candidates, which are based on Zero-Knowledge proof systems. Only Category I parameters and the fastest AVX2 optimized implementation are compared. The table clearly shows that LESS positions as a strong candidate, with the smallest signature sizes. For an additional in depth comparison we recommend either PQSort [5] or the NIST Signature Zoo [6]. Both websites include data for the other security categories as well as better comparison tools.

[5] https://pqsort.tii.ae/
[6] https://pqshield.github.io/nist-sigs-zoo/

Table 7: Comparison of different NIST Round 2 Submissions. Clock cycle values collected via `rtdscp` on an AMD Ryzen 5 7600X. Results are shown only for Category I parameter sets, with benchmarks limited to the fastest AVX2 implementation.

Name	Variant	Signature Size (kB)			Performance (Mcc)		
		pk	sig	pk + sig	Keygen	Sign	Verify
CROSS [BBB+25a]	Speed	0.054	11.98	12.03	0.026	1.36	0.83
	Balanced	0.054	9.12	9.17	0.026	2.88	1.73
	Size	0.054	8.96	9.01	0.026	5.71	3.45
FAEST [BBB+25b]	EM-128f	0.032	5.06	5.09	0.015	1.38	1.15
	EM-128s	0.032	3.90	3.93	0.014	9.49	7.98
Mirath [AAB+25]	Ia_fast	0.07	3.73	3.80	0.190	8.22	7.23
	Ia_short	0.07	3.08	3.15	0.191	136.91	102.77
MQOM [BBFR25]	gf256_fast_r5	0.08	3.62	3.70	0.33	5.09	4.37
	gf256_short_r5	0.08	3.16	3.24	0.30	13.55	13.59
PERK [ABB+25a]	I-fast5	0.15	8.03	8.05	0.10	6.28	4.36
	I-short5	0.24	5.78	6.02	0.10	31.59	21.93
RYDE [ABB+25b]	F	0.07	3.60	3.67	0.038	12.3	11.22
	S	0.07	2.99	3.06	0.038	131.81	127.82
SDitH [MBB+25]	fast	0.07	4.48	8.55	14.1	7.78	7.17
	short	0.07	3.71	3.78	1.41	22.20	21.37
LESS	252-192	13.94	2.29	16.23	0.7	117.0	106.8
	252-68	41.79	1.75	43.54	1.6	42.4	39.3
	252-45	97.48	1.31	98.79	3.6	26.5	26.3

Acknowledgments

The work of Paolo Santini has been supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU. Andre Esser has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID MA 2536/12.

References

[AAB+25] Gora Adj, Nicolas Aragon, Stefano Barbero, Magali Bardet, Emanuele Bellini, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Andre Esser, Thibauld Feneuil, Philippe Gaborit, Romaric Neveu, Matthieu Rivain, Luis Rivera-Zamarripa, Carlo Sanna, Jean-Pierre Tillich, Javier Verbel, and Floyd Zweydinger. Mirath signature scheme, 2025. https://pqc-mirath.org/.

[ABB+25a] Najwa Aaraj, Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Victor Dyseryn, Andre Esser, Thibauld Feneuil, Philippe Gaborit, Mukul Kulkarni, Victor Mateu, Marco Palumbi, Lucas Perin, Matthieu Rivain, Jean-Pierre Tillich, and Keita Xagawa. Perk, 2025. https://pqc-perk.org/.

[ABB+25b] Nicolas Aragon, Magali Bardet, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Thibauld Feneuil, Philippe Gaborit, Antoine Joux, Romaric Neveu, Matthieu Rivain, Jean-Pierre Tillich, and Adrien Vincotte. Ryde, 2025. https://pqc-ryde.org/.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30*

April - 2 May 1968, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[BBB+25a] Marco Baldi, Alessandro Barenghi, Michele Battagliola, Sebastian Bitzer, Patrick Karl, Felice Manganiello, Alessio Pavoni, Gerardo Pelosi, Paolo Santini, Jonas Schupp, Edoardo Signorini, Freeman Slaughter, Antonia Wachter-Zeh, and Violetta Weger and. Cross: Codes and restricted objects signature scheme, 2025. https://www.cross-crypto.com/.

[BBB+25b] Carsten Baum, Ward Beullens, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. Faest, 2025. https://faest.info/.

[BBFR25] Ryad Benadjila, Charles Bouillaguet, Thibauld Feneuil, and Matthieu Rivain. Mq on my mind, 2025. https://mqom.org/.

[BBPS21] Alessandro Barenghi, Jean-François Biasse, Edoardo Persichetti, and Paolo Santini. LESS-FM: Fine-Tuning Signatures from the Code Equivalence Problem. In *Post-Quantum Cryptography: 12th International Workshop, PQCrypto 2021, Daejeon, South Korea, July 20–22, 2021, Proceedings 12*, pages 23–43. Springer, 2021.

[BEFN25] Alessandro Budroni, Andre Esser, Ermes Franch, and Andrea Natale. Two is all it takes: Asymptotic and concrete improvements for solving code equivalence. *IACR Cryptol. ePrint Arch.*, page 227, 2025.

[Ber20] Daniel J. Bernstein. Verified fast formulas for control bits for permutation networks. *IACR Cryptol. ePrint Arch.*, page 1493, 2020.

[BMPS20] Jean-François Biasse, Giacomo Micheli, Edoardo Persichetti, and Paolo Santini. LESS is More: Code-based Signatures Without Syndromes. In *Progress in Cryptology - AFRICACRYPT 2020: 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20–22, 2020, Proceedings 12*, pages 45–65. Springer, 2020.

[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*, 2023.

[CPS25] Tung Chou, Edoardo Persichetti, and Paolo Santini. On linear equivalence, canonical forms, and digital signatures. *Designs, Codes and Cryptography*, pages 1–43, 2025.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology — CRYPTO' 86*, pages 186–194. Springer, 1986.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *Journal of the ACM (JACM)*, 38(3):690–728, 1991.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[MBB+25] Carlos Aguilar Melchor, Slim Bettaieb, Loïc Bidoux, Thibauld Feneuil, Philippe Gaborit, Nicolas Gama, Shay Gueron, James Howe, Andreas Hülsin, David Joseph, Antoine Joux, Mukul Kulkarni, Edoardo Persichetti, Tovohery H. Randrianarisoa, Matthieu Rivain, and Dongze Yue and. The syndrome decoding in the head (sd-in-the-head) signature scheme, 2025. https://sdith.org/.

[NIS23] NIST. Call for Additional Digital Signature Schemes for the Post-Quantum Cryptography Standardization Process, 2023. https://csrc.nist.gov/projects/pqc-dig-sig/standardization/call-for-proposals.

[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.