Wavelet: Code-based postquantum signatures with fast verification on microcontrollers

Gustavo Banegas[1], Thomas Debris-Alazard[1], Milena Nedeljković[2] and Benjamin Smith[1]

[1] Inria and Laboratoire d'Informatique de l'École polytechnique, Institut Polytechnique de Paris, Palaiseau, France,
gustavo@cryptme.in,thomas.debris@inria.fr,smith@lix.polytechnique.fr
[2] École polytechnique, Institut Polytechnique de Paris, Palaiseau, France
milena.nedeljkovic@polytechnique.edu

Abstract. This work presents the first full implementation of Wave, a postquantum code-based signature scheme. We define Wavelet, a concrete Wave scheme at the 128-bit classical security level (or NIST postquantum security Level 1) equipped with a fast verification algorithm targeting embedded devices. Wavelet offers 930-byte signatures, with a public key of 3161 kB. We include implementation details using AVX instructions, and on ARM Cortex-M4, including a solution to deal with Wavelet's large public keys, which do not fit in the SRAM of a typical embedded device. Our verification algorithm is $\approx 4.65\times$ faster then the original, and verifies in 1 087 538 cycles using AVX instructions, or 13 172 ticks in an ARM Cortex-M4.

Keywords: Post-quantum Cryptography · Code-based Signature · Fast Verification · Implementation · Embedded devices.

1 Introduction

Ensuring the long-term security of constrained devices is a critical, and perennial, problem. The secure and efficient implementation of public-key cryptographic algorithms for microcontrollers is vital, but difficult given the constrained resources available on these devices. The difficulties are magnified when it comes to implementing *post-quantum* cryptosystems, which generally have much larger keys and more intensive computing requirements than their elliptic-curve equivalents.

In this article, we focus on the problem of implementing post-quantum signature schemes with efficient verification on microcontrollers. Our scheme, Wavelet, is a variant of Wave [DST19a], a post-quantum trapdoor based on the hardness of problems in coding theory. Wave has attractive properties, including *simple verification* and relatively *short signatures*. In a sea of broken code-based signature schemes (including [BBC$^+$13], [PT16], [SHM$^+$20], and [ABD$^+$21]), Wave maintains the promise of high security. Wave also presents some interesting practical challenges: for example, the Wave designers recommend working over the finite field $\mathbb{F}_3$, which is highly unusual in contemporary cryptographic software.

Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. This work was funded in part by the European Commission through H2020 SPARTA, https://www.sparta.eu/ and by the French Agence Nationale de la Recherche through ANR JCJC COLA (ANR-21-CE39-0011). Generic disclaimer: "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies."

WAVELET is a signature scheme based on WAVE over $\mathbb{F}_3$, but designed for efficient verification on constrained devices. WAVELET keys and signatures are easily converted to WAVE keys and signatures over $\mathbb{F}_3$, and vice versa; in particular, WAVELET inherits its security from WAVE over $\mathbb{F}_3$. There are two main benefits to WAVELET compared with WAVE: shorter signatures, and faster verification.

WAVE signatures are already remarkably short, but WAVELET signatures are even shorter. We achieve this on two levels: first, we truncate WAVE signatures from length n (the code length) to length k (the code dimension). This already represents a substantial improvement: for our parameters, truncated signatures are one-third shorter. But these truncated signatures can be compressed further using minimal-redundancy coding, at minimal computational cost.

The end result is that WAVELET signatures are the shortest among code-based signatures, and nearly half the length of WAVE signatures for the same code parameters. In practice, when targeting the 128-bit classical security level (and NIST postquantum Level 1 security), WAVELET signatures fit in under one kilobyte. This is competitive with lattice-based postquantum signatures: NIST Round-3 candidates Dilithium [ABD+] and Falcon [FHK+] targeting similar security levels offer 2420- and 666-byte signatures, respectively (see Table 1 for a more complete comparison).

WAVELET's signature verification is also faster—and not just because the signatures are shorter. Indeed, a subtle tweak to the public key allows us to avoid loading or operating on nearly half of its data during any given signature verification. This yields an important speedup by drastically reducing the number of field operations, but also by reducing the impact of memory latency—an effect that is further exaggerated in microcontrollers by the fact that the public key is typically too large for the RAM.

We have put this all into practice with a free and portable C implementation of WAVELET for x86_64, with verification on x86_64 (with and without AVX) and on Cortex-M4 platforms, targeting the 128-bit security level (and NIST Level 1). This is the first full implementation of a WAVE signature scheme[1] on any platform. We show how to use high-speed bitsliced $\mathbb{F}_3$-vector arithmetic to speed WAVE systems, and use external Flash via QSPI to handle very large public keys (over 3MB). Using this software, WAVELET signatures can be verified on an Intel® Core™ PC in 450μs, and on an ARM Cortex-M4 powered board in 402ms.

Table 1: Size of private/public keys and signatures for postquantum signature schemes.

Algorithm	Private Key (B)	Public Key (B)	Signature (B)
WAVELET	32 or 11043162	3236327	930
Falcon-512 [FHK+]	1281	897	666
Dilithium2 [ABD+]	2528	1312	2420
LMS [MCF19]	64	60	4756
SPHINCS+-128f [ABB+]	64	32	17088
GeMSS [CFMR+]	48	14520	417416
Picnic3-L1 [CDG+]	17	34	13802

Organization of the paper Section 2 gives an overview of WAVE, defines the *Supertubos* parameters, and explains the improvements that lead to our concrete scheme, WAVELET. Section 3 gives detailed versions of the algorithms in WAVELET. Section 4 gives useful binary

[1] A proof-of-concept implementation of the original WAVE trapdoor was published with [DST19a] at [DST19b], but it cannot be used to sign messages: in particular, it does not include the ternary hash function required to hash messages.

representations for ternary keys, describes the efficient compression and decompression of Wave and Wavelet signatures, and calculates key and (compressed) signature sizes. Section 5 gives details on the implementation of Wavelet in software. Section 6 shows the experimental results from our implementation on x86_64 (with and without AVX) and Cortex-M4 platforms. We conclude in Section 7, giving perspectives on future development.

Notation and conventions Vectors and matrices are written in bold. Vectors are in row notation. Indices start at 0: if $\mathbf{v}$ is a vector in $\mathbb{F}_q^N$, then its components are $(v_0, \ldots, v_{N-1})$; if $\mathbf{H}$ is a matrix in $\mathbb{F}_q^{M\times N}$, then its row vectors are $\mathbf{H}_0, \ldots, \mathbf{H}_{M-1}$ and $\mathbf{H}_{i,j}$ is its coefficient at position (i, j). The Hamming weight of $\mathbf{v}$ is $|\mathbf{v}| := \#\{0 \le i < N \mid \ v_i \neq 0\}$. The concatenation of $\mathbf{u}$ and $\mathbf{v}$ is denoted by $\mathbf{u} \parallel \mathbf{v}$.

The set of permutations of $\{0, \ldots, N-1\}$ is denoted by $\mathscr{S}_N$. We represent a permutation π in $\mathscr{S}_N$ by the sequence $(\pi(0), \ldots, \pi(N-1))$ (and we warn the reader that this is *not* cycle notation). Permutations act on vector coordinates:

$$\pi(\mathbf{v}) := (v_{\pi(0)}, \ldots, v_{\pi(N-1)}) \parallel (v_N, \ldots, v_{N'-1}) \quad \text{for } \pi \in \mathscr{S}_n \text{ and } \mathbf{v} \in \mathbb{F}_q^{N'} \text{ with } N' \ge N\,.$$

If $\mathbf{H}$ is a matrix in $\mathbb{F}_q^{M\times N}$, then $\pi(\mathbf{H})$ is the matrix with row vectors $\pi(\mathbf{H}_0), \ldots, \pi(\mathbf{H}_{M-1})$.

To ease analysis, the functions and subroutines in our algorithms do *not* modify their arguments. In practice, we (strongly) recommend implementing them to modify large vector and matrix arguments in-place, to save memory. Finally, if $\mathscr{E}$ is a finite set, then $x \xleftarrow{\$} \mathscr{E}$ means that x is sampled uniformly at random from $\mathscr{E}$.

2 Wave and Wavelet

We begin with a high-level view of Wave, before explaining the modifications that yield Wavelet. This will serve as a high-level tour of our main theoretical results.

2.1 Wave

Wave is a full-domain hash (FDH) signature scheme [BR96, Cor02]. Fix a prime power $q \neq 2$, and let $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ be a parity-check matrix for a code $W = \{\mathbf{x} \in \mathbb{F}_q^n \mid \mathbf{x}\mathbf{H}^\intercal = \mathbf{0}\}$ of dimension k and length n over $\mathbb{F}_q$. Fix a weight $w < n$. If W is a random code, then for carefully chosen w, the function mapping error vectors $\mathbf{e}$ in $\mathbb{F}_q^n$ of weight w to their syndromes $\mathbf{s} = \mathbf{e}\mathbf{H}^\intercal$ is a one-way function: inverting it corresponds to decoding a random linear code. But if we suppose that W is equipped with a trapdoor allowing us to invert the one-way function, then we get the FDH signature scheme of Figure 1.

```
Sign(m, sk):
    r ←$ {0,1}^λ
    s ← Hash(m, r)
    e ← InvAlg(s, sk)
    return(e, r)

Verify(m, (e', r), pk = (H, w)):
    s ← Hash(m, r)
    If e'H^T == s and |e'| == w:
        return True
    Else:
        return ⊥
```

Figure 1: Code-based signature using FDH paradigm (on a high level). The function `InvAlg` is a "decoder" that inverts the trapdoor, correctly finding an error $\mathbf{e}$ with $|\mathbf{e}| = w$.

The Wave trapdoor is built from two random linear codes U and V of length $n/2$ and dimensions k_U and k_V, respectively, over $\mathbb{F}_q$. These codes are combined way to form a code W of length n and dimension $k = k_U + k_V$, using a construction explained in §3.2.

The public key is a parity-check matrix $\mathbf{H}_{\text{pk}}$ in $\mathbb{F}_q^{(n-k)\times n}$ of the code W; the private key consists of U, V, and data allowing us to map decoding problems into U and V. The parameters k_U, k_V, n, and w are carefully balanced, and the construction of W from U and V carefully designed, so that the map $\mathbf{e} \mapsto \mathbf{s} = \mathbf{e}\mathbf{H}_{\text{pk}}{}^{\mathsf{T}}$ is a trapdoor: computing a solution $\mathbf{e}$ of the correct weight w for a given $\mathbf{s}$ is cryptographically hard *unless* we have the secret key, in which case we can use the decoding algorithm described in §3.3.

To construct a signature scheme from this trapdoor, it suffices to let $\mathbf{h}$ be the hash of a message m with a random salt r; the WAVE signature is the pair $(\mathbf{e}, r)$. To verify the signature, it suffices to check that $\mathbf{e}\mathbf{H}_{\text{pk}}{}^{\mathsf{T}}$ is the hash of m and r and that $|\mathbf{e}| = w$.

The theoretical security of WAVE is beyond the scope of this article. Details and proofs appear in [DST19a].

2.2 Parameters

The fundamental WAVE parameters are a classical security parameter λ, a field size $q \neq 2$, vector space dimensions n, k_U, and k_V, a Hamming weight w, and a parameter d used in the rejection sampling in the signing algorithm. Concrete values for these parameters targeting 128-bit classical security and NIST Level 1 postquantum security were specified in [DST19a], and are reproduced here in Table 2. We call this parameter set *Supertubos*.[2]

Table 2: General WAVE parameters for 128 bits of security.

Parameters	λ	q	n	w	$k = k_U + k_V$	d
Supertubos	128	3	8492	7980	5605 = 3558 + 2047	81

Our WAVELET implementation uses *Supertubos*, and the reader may keep these parameter values in mind to get a concrete picture of WAVELET's practical improvements. The WAVELET algorithms will work with any WAVE parameter set with $q = 3$, though the performance of the signature compression algorithm is sensitive to the ratio $w/n \approx 0.94$. Deriving new parameter sets that may be better suited to constrained evironments (or other applications) is left for future work.

2.3 Restricting to $q = 3$

The description of WAVE in [DST19a] defines a trapdoor over a general finite field $\mathbb{F}_q$ with $q \neq 2$, but the security analysis focuses on the case $q = 3$, and parameters are only specified for $q = 3$. The analysis of WAVE with $q > 3$ is an open question.

While [DST19a] recommends restricting WAVE to $q = 3$, WAVELET explicitly *requires* $q = 3$. Forcing $q = 3$ allows several important practical improvements, including

- shorter signatures through more effective compression, and more compact keypairs;
- simple field arithmetic and high-speed bitsliced vector operations (see §5.1); and
- interesting algorithmic optimizations, especially in signature verification.

Using $q = 3$ also imposes some interesting practical questions, since the data to be signed, and the keys and signatures, must ultimately be encoded in binary. The choice of binary encoding for each ternary object is addressed in §4.

But not all of these problems are a simple matter of data representation. For example: WAVE is a hash-and-sign signature scheme, and it requires a cryptographic hash function $\{0,1\}^* \to \mathbb{F}_3^{n-k}$. No concrete hash function is specified in [DST19a], and indeed we are not aware of any standard, high-security hash functions mapping from binary to ternary

[2] See `39.3410294768957, -9.36095797386207`.

domains. Instead, we must build one from a secure hash functions into a binary domain. The construction of a hash function into $\mathbb{F}_3^{n-k}$ is addressed in §3.1.

2.4 Truncated signatures and Wavelet public keys

The WAVE public key is a parity-check matrix $\mathbf{H}_{\text{pk}}$ for the code W; but in fact, the basic WAVE key-generation algorithm always constructs a row-reduced $\mathbf{H}_{\text{pk}} = (\mathbf{I}_{n-k}|\mathbf{R})$ where $\mathbf{R}$ is a random-looking $(n-k) \times k$ matrix over $\mathbb{F}_3$. The verification equations are

$$|\mathbf{e}| = w \qquad \text{and} \qquad \mathbf{h} = \mathbf{e}{\mathbf{H}_{\text{pk}}}^{\intercal} \qquad \text{where} \qquad \mathbf{h} = \texttt{Hash}(r \parallel m)\,. \tag{1}$$

Now let $\mathbf{s}$ in $\mathbb{F}_3^k$ consist of the last k coordinates of $\mathbf{e}$; let $\mathbf{u} := \mathbf{h} - \mathbf{s}\mathbf{R}^{\intercal}$ and $\mathbf{e}' := \mathbf{u} \parallel \mathbf{s}$. If $\mathbf{e}{\mathbf{H}_{\text{pk}}}^{\intercal} = \mathbf{h}$, then $\mathbf{e}'{\mathbf{H}_{\text{pk}}}^{\intercal} = \mathbf{h}$ too (indeed, typically $\mathbf{e} = \mathbf{e}'$), and $|\mathbf{e}| = |\mathbf{e}'| = |\mathbf{u}| + |\mathbf{s}|$. We can therefore transmit $(r, \mathbf{s})$ as the signature instead of $(r, \mathbf{e})$, thus saving $n-k$ elements of $\mathbb{F}_3$, and use the verification equation

$$|\mathbf{u}| + |\mathbf{s}| = w \qquad \text{where} \qquad \mathbf{u} = \mathbf{h} - \mathbf{s}\mathbf{R}^{\intercal} \qquad \text{with} \qquad \mathbf{h} = \texttt{Hash}(r \parallel m)\,. \tag{2}$$

For *Supertubos*, this reduces the uncompressed signature length—and the work involved in the vector-matrix product—by roughly one-third.

The verifier needs to compute $\mathbf{s}\mathbf{R}^{\intercal} = \sum_{i=0}^{k-1} s_i \mathbf{R}_i^{\intercal}$, where $\mathbf{R}_i^{\intercal}$ is the i-th row of $\mathbf{R}^{\intercal}$.[3] Since the s_i are in $\mathbb{F}_3$, no multiplication is really required: computing the sum amounts to adding $\mathbf{R}_i^{\intercal}$ into an accumulator if $s_i = 1$, subtracting it if $s_i = 2$, and ignoring it completely if $s_i = 0$. Skipping rows is an important optimization, and not just to minimize field operations: loading rows from memory (or external flash) is expensive.

But the vector $\mathbf{s}$ has unusually few 0s. Indeed, $|\mathbf{e}| = w$, which is quite close to n in the WAVE context, so we expect $|\mathbf{s}|$ to be proportionally close to k. At first, this may appear frustrating: there are not many rows to skip when computing $\mathbf{s}\mathbf{R}^{\intercal}$. However, we can turn this imbalance around to create a relatively sparse sum using the following propositions.

First, for each $\mathbf{s}$ in $\mathbb{F}_3^k$, we define a vector $\hat{\mathbf{s}}$ in $\mathbb{F}_3^k$ by

$$\left.\begin{aligned} \hat{s}_{2i} &:= s_{2i} + s_{2i+1} \\ \hat{s}_{2i+1} &:= s_{2i} - s_{2i+1} \end{aligned}\right\} \quad \text{for} \quad 0 \le i < k/2\,, \quad \text{and} \quad \hat{s}_{k-1} := s_{k-1} \text{ if } k \text{ is odd}\,. \tag{3}$$

Proposition 1. *Let* $\mathbf{s}$ *be a vector in* $\mathbb{F}_3^k$*, and let* $\hat{\mathbf{s}}$ *be defined as in* (3)*. Then*

$$|\hat{\mathbf{s}}| + |\mathbf{s}| = \frac{3}{2}(k + \epsilon) - 3\delta$$

where

$$\delta = \#\{0 \le i < k/2 \mid s_{2i} = s_{2i+1} = 0\} \quad \textit{and} \quad \epsilon = \begin{cases} 0 & \textit{if } k \textit{ is even;} \\ 1 & \textit{if } k \textit{ is odd and } s_{k-1} \neq 0; \\ -1 & \textit{if } k \textit{ is odd and } s_{k-1} = 0\,. \end{cases}$$

Proof. See Appendix D.

Proposition 1 shows that as $|\mathbf{s}|$ grows, $|\hat{\mathbf{s}}|$ shrinks. Indeed, in the limiting case where $|\mathbf{s}| = k$, we find $|\hat{\mathbf{s}}| = \lceil k/2 \rceil$. Now we just need to find a way to verify using a vector-matrix product involving $\hat{\mathbf{s}}$ instead of $\mathbf{s}$.

[3] One might also compute this as a vector of dot-products between $\mathbf{s}$ and the columns of $\mathbf{R}^{\intercal}$; but when using bitsliced arithmetic, it seems harder to exploit the weight of $\mathbf{s}$ to reduce the cost of this approach.

Proposition 2. *Let* $\mathbf{R}$ *be in* $\mathbb{F}_3^{(n-k)\times k}$. *For every* $\mathbf{s}$ *in* $\mathbb{F}_3^k$, *if* $\hat{\mathbf{s}}$ *is defined as in* (3), *then*

$$\mathbf{s}\mathbf{R}^\intercal = -\hat{\mathbf{s}}\mathbf{M} \tag{4}$$

where $\mathbf{M}$ *is the matrix in* $\mathbb{F}_3^{k\times(n-k)}$ *whose rows are*

$$\left.\begin{aligned}\mathbf{M}_{2i} &:= \mathbf{R}_{2i}^\intercal + \mathbf{R}_{2i+1}^\intercal \\ \mathbf{M}_{2i+1} &:= \mathbf{R}_{2i}^\intercal - \mathbf{R}_{2i+1}^\intercal\end{aligned}\right\} \quad \textit{for} \quad 0 \le i < k/2\,, \quad \textit{and} \quad \mathbf{M}_{k-1} := -\mathbf{R}_{k-1}^\intercal \textit{ if } k \textit{ is odd}.$$

Proof. See Appendix D.

Proposition 2 shows that if we replace the public key $\mathbf{R}$ with $\mathbf{M}$, then the validation equation (2) becomes

$$|\mathbf{u}| + |\mathbf{s}| = w\,, \quad \text{where} \quad \mathbf{u} = \mathbf{h} + \hat{\mathbf{s}}\mathbf{M} \quad \text{with} \quad \mathbf{h} = \mathtt{Hash}(r \parallel m)\,. \tag{5}$$

Clearly $\mathbf{R}^\intercal$ can be recovered from $\mathbf{M}$, and $\mathbf{s}$ from $\hat{\mathbf{s}}$, so solving (5) is equivalent to solving (2). But $\hat{\mathbf{s}}$, which can be rapidly computed from $\mathbf{s}$ on the fly using (3), has much lower weight by Proposition 1: roughly speaking, we expect $|\mathbf{s}|$ to be close to k, and so $|\hat{\mathbf{s}}|$ should be close to $k/2$. This lets us compute the product $\hat{\mathbf{s}}\mathbf{M}$, and hence verify (5), in much less time than (2) (for *Supertubos*, nearly half); and less time again than that required for the equivalent WAVE signatures using (1).

We therefore take $\mathbf{M}$ to be the public key in WAVELET; we verify WAVELET signatures $(r, \mathbf{s})$ using (5) and (3). Note that we do *not* replace $\mathbf{s}$ with $\hat{\mathbf{s}}$ in the signature: the weight condition $w = |\mathbf{u}| + |\mathbf{s}|$ is simpler than the corresponding condition involving $|\hat{\mathbf{s}}|$, and in any case the weight $|\mathbf{s}|$ can be easily computed at the same time as $\hat{\mathbf{s}}$.

Remark 1. The mapping $\mathbf{s} \mapsto \hat{\mathbf{s}}$ can be recognised as the first step in a fast Hadamard transform. We might naturally ask if pushing further with the Hadamard transform on $\mathbf{s}$ would produce even lower weights; it does not. Instead, the weight of the image vector creeps back up again. With the *Supertubos* parameters, the expected $|\mathbf{s}|/k$ is ≈ 0.94. One step of the fast Hadamard transform gives an expected $|\hat{\mathbf{s}}|/k$ of ≈ 0.55; but the next step brings this expected ratio up to ≈ 0.65, and the following step to ≈ 0.67.

2.5 Signature compression

A WAVELET signature is a pair $(r, \mathbf{s})$, with r in $\{0,1\}^{2\lambda}$ and $\mathbf{s}$ in $\mathbb{F}_3^k$. We must choose a binary encoding of $\mathbf{s}$, and at first glance it would appear that this requires (at least) $\log_2(3)k$ bits. However, this ignores the fact that $\mathbf{s}$ has very high weight, and thus an entropy significantly less than $\log_2(3) \approx 1.585$.

Concretely, if we use the *Supertubos* parameters where $n = 8492$ and $w = 7980$, then the entropy is ≈ 1.268. In theory, then, we can hope to encode $\mathbf{s}$ in as few as $1.268k \approx 7109$ bits. This is better than $1.585k \approx 8884$ bits, and much better than the $1.585n \approx 13460$ bits for WAVE signatures suggested in [DST19a] ("the order of 13 thousand bits").

We show in §4.2 that we can approach this level of efficiency using a simple Minimal Redundancy (MR) coding. This variable-length coding yields variable-length signatures, but on average we can compress $\mathbf{s}$ to around 7280 bits.

3 Wavelet algorithms

In this section, we present the details of our hashing, Key Generation, Signing, and Verification algorithms. The key generation and signing algorithms are *not* constant-time, and should only be run in trusted environments. We leave the development of countermeasures against side-channel attacks for WAVE and WAVELET for future work.

3.1 Hashing to ternary vectors

In order to sign any messages, we need to define a hash function into $\mathbb{F}_3^{n-k}$. The simplest option would be to hash into $\{0,1\}^{\lceil \log_2(3)(n-k) \rceil}$ using an XOF, view the output as an integer, and "ternarize" the result (i.e., read off its 3-adic coefficients) to get the entries of the output in $\mathbb{F}_3^{n-k}$. For realistic parameter sizes, though, this is too slow and intensive: for *Supertubos*, for example, this would mean repeated Euclidean division by 3 on a 4576-bit integer. This is inconvenient on a PC, and unrealistic on embedded platforms.

Our hash function, called `Hash` (Algorithm 1) combines the following three functions to define a cryptographic hash function $\{0,1\}^* \to \mathbb{F}_3^{n-k}$:

- `BinaryHash` : $\{0,1\}^* \to \{0,1\}^{2\lambda}$ is a cryptographic hash function.
- `Ternarize` : $\{0,1\}^* \times \mathbb{Z}_{\geq 0} \to \mathbb{F}_3^*$ (Algorithm 2) views its input $((x_0, x_1, \ldots), \tau)$ as the vector of coefficients in the little-endian binary expansion of an integer x, together with a length τ, and returns the ternary vector of length τ representing the little-endian ternary expansion of x mod 3^τ.
- `Expand`: $\{0,1\}^{2\lambda} \to \mathbb{F}_3^\tau$ (Algorithm 3) is a pseudorandom function. `Expand` applies an XOF (or a stream cipher) to its input to produce a long stream of pseudorandom bytes, which we view as integers in $[0, 255]$. The non-negative integers less than $3^5 = 243$ are in bijection with $\mathbb{F}_3^5$, so if a byte is less than 243 we convert it to an element of $\mathbb{F}_3^5$ with `Ternarize` and concatenate it to the output; otherwise we skip the byte. We continue processing bytes until we have produced τ elements of $\mathbb{F}_3$ (discarding the last few if τ is not a multiple of 5). This process generates a distribution of vectors in $\mathbb{F}_3^\tau$ that is computationally indistinguishable from the uniform distribution (with respect to the security parameter).[4]

Algorithm 1: Hashing from binary data to ternary vectors for Wavelet. We write μ for $\lfloor 2\lambda / \log_2(3) \rfloor$.

1 **Function** `Hash`(x)
  Input: $x \in \{0,1\}^*$
  Output: $\mathbf{s} \in \mathbb{F}_3^{n-k}$
2 $h \leftarrow$ `BinaryHash`(x) // `Hash` function $\{0,1\}^* \to \{0,1\}^{2\lambda}$
3 $\mathbf{t} \leftarrow$ `Ternarize`$(h,\ \mu)$ // `Ternarize`: $\{0,1\}^{2\lambda} \to \mathbb{F}_3^\mu$
4 $\mathbf{p} \leftarrow$ `Expand`$(h,\ n-k-\mu)$ // `Expand`: $\{0,1\}^{2\lambda} \to \mathbb{F}_3^{n-k-\mu}$
5 **return** $\mathbf{t} \parallel \mathbf{p}$

The collision- and preimage-resistance of `Hash` are derived from the properties of `BinaryHash`. `Ternarize` transcodes the output of `BinaryHash`; this composition maintains the security of `BinaryHash`, but is relatively slow to compute. The composition of `Expand` and `BinaryHash` has weaker preimage and collision resistance, but good pseudorandomness properties, and is relatively fast to compute. The concatenation of the two has the security of the strong hash, and the good pseudorandomness of both.

In practice, `BinaryHash` could be SHA3-512, while the bytestream in `Expand` could be generated with an XOF such as SHAKE-256. This minimises the code size, since both are built on the same Keccak permutation, while maintaining appropriate domain separation.

[4] A true uniform distribution on $\mathbb{F}_3^\tau$ cannot be produced in this way when τ is large, because there are only $2^{2\lambda}$ possible stream outputs—and even for large λ, the number of possible output vectors is limited by the size of the internal state of the XOF or stream cipher used to generate the stream. On the other hand, this process has the advantage of being simple and fast.

Algorithm 2: Converting integer values to ternary vectors of a specified length, corresponding to the little-endian ternary expansion of the input

Function `Ternarize`$(x,\ \tau)$
  Input: $x \geq 0$ and $\tau > 0$
  Output: $\mathbf{v} \in \mathbb{F}_3^\tau$ such that $x = \sum_{i=1}^{\tau} v_i 3^{i-1}$, where the v_i are lifted to {0,1,2}
  $(\mathbf{v}, t) \leftarrow ((), x)$ `// v: empty vector over` $\mathbb{F}_3$
  for $1 \leq i \leq \tau$ **do**
    $(\mathbf{v}, t) \leftarrow (\mathbf{v} \parallel (r), q)$ **where** $(q, r) = (\lfloor t/3 \rfloor, t \bmod 3)$
  return v

Algorithm 3: Expand a binary seed to a pseudo-random stream of ternary values. The random bytestream may be instantiated with an XOF or a stream cipher. The expected number of bytes drawn from the stream is $(256\tau)/(243 \times 5) \approx 0.21\tau$.

Function `Expand`$(h,\ \tau)$
  Input: $h \in \{0,1\}^{2\lambda}$ and $\tau > 0$
  Output: $\mathbf{p} \in \mathbb{F}_3^\tau$
  stream = random bytestream seeded with h `// E.g. secure XOF(`h`)`
  $(\mathbf{p}, r) \leftarrow ((), \tau)$ `// p: empty vector over` $\mathbb{F}_3$
  while $r > 0$ **do**
    $b \leftarrow$ next byte from stream, viewed as an integer in $[0, 255]$
    if $b < 243$ **then**
      $(\mathbf{p}, r) \leftarrow (\mathbf{p} \parallel$ `Ternarize`$(b, \min(5, r)), r)$
  return p

3.2 Key generation

Algorithm 4 generates a WAVELET keypair. It is essentially the WAVE key-generation procedure from [DST19a], with the public key transformation at the end. We have made no attempt to optimize this procedure beyond the use of bitsliced and vectorized $\mathbb{F}_3$-arithmetic.

Given an $(n/2 - k_U) \times k_U$ matrix $\mathbf{R}_U$, an $(n/2 - k_V) \times k_V$ matrix $\mathbf{R}_V$, and vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$ in $\mathbb{F}_3^{n/2}$, the subroutine `ParityCheckUV(`$\mathbf{R}_U$, $\mathbf{R}_V$, $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$`)` returns the $(n-k) \times n$ matrix

$$\mathbf{H}_{\text{sk}} = \begin{pmatrix} \mathbf{H}_U\mathbf{D} & -\mathbf{H}_U\mathbf{B} \\ -\mathbf{H}_V\mathbf{C} & \mathbf{H}_V\mathbf{A} \end{pmatrix} \quad \text{where} \quad \begin{cases} \mathbf{H}_U := (\mathbf{I}_{n/2-k_U} \mid \mathbf{R}_U)\,, \\ \mathbf{H}_V := (\mathbf{I}_{n/2-k_V} \mid \mathbf{R}_V)\,, \end{cases}$$

and

$$\mathbf{A} := \mathbf{Diag}(\mathbf{a})\,, \qquad \mathbf{B} := \mathbf{Diag}(\mathbf{b})\,, \qquad \mathbf{C} := \mathbf{Diag}(\mathbf{c})\,, \qquad \mathbf{D} := \mathbf{Diag}(\mathbf{d})\,;$$

here, $\mathbf{Diag}(\mathbf{x})$ denotes the $n/2 \times n/2$ diagonal matrix with diagonal entries given by $\mathbf{x}$.

The auxiliary function `GaussElim` is given in Algorithm 5. For this function, we need to "split" into two functions, that is, in our scheme we use it to generate a submatrix with certain rank and for this, we need to have the support as part of our secret key thus we changed a little to know which rows are the pivots.

Algorithm 6 gives the elimination of a single row, for this we check if the element is not 0 then we perform the operations. Later, we swap rows to put in the correct "rank", and do the sum and subtraction necessary to just let the (r, j)-th element as not zero.

Algorithm 4: WAVELET Key Generation. This algorithm also serves for classic WAVE key generation if we stop after Line 11 and return $\mathtt{sk}$ and $\mathtt{pk} = (\mathbf{I}_{n-k}|\mathbf{R})$.

1 **Function** `KeyGen()`
  Parameters: n, w, k_U, k_V, k
  Output: $(\mathtt{sk}, \mathtt{pk})$
2 $\mathbf{R}_U \xleftarrow{\$} \mathbb{F}_3^{(n/2-k_U)\times k_U}$
3 $\mathbf{R}_V \xleftarrow{\$} \mathbb{F}_3^{(n/2-k_V)\times k_V}$
4 $\mathbf{a} \xleftarrow{\$} (\mathbb{F}_3\backslash\{0\})^{n/2}$
5 $\mathbf{b} \xleftarrow{\$} \mathbb{F}_3^{n/2}$
6 $\mathbf{c} \xleftarrow{\$} (\mathbb{F}_3\backslash\{0\})^{n/2}$
7 $\mathbf{d} \leftarrow (d_i)_{0\le i<n/2}$ **where** $d_i \xleftarrow{\$} \mathbb{F}_3\backslash\{b_i c_i a_i^{-1}\}$ for $0 \le i < n/2$
8 $\mathbf{H}_{\mathrm{sk}} \leftarrow \mathtt{ParityCheckUV}(\mathbf{R}_U, \mathbf{R}_V, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$
9 $\pi_{\mathrm{sk}} \xleftarrow{\$} \mathscr{S}_n$
10 $((\mathbf{I}_{n-k} \mid \mathbf{R}), \pi_{\mathrm{sk}}) \leftarrow \mathtt{GaussElim}(\mathbf{H}_{\mathrm{sk}}, \pi_{\mathrm{sk}})$
11 $\mathtt{sk} \leftarrow (\mathbf{R}_U, \mathbf{R}_V, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \pi_{\mathrm{sk}})$
12 $\mathbf{M} = \mathbf{0}^{k\times(n-k)}$
13 **for** $0 \le i < (k-1)/2$ **do** `// Build first` $2\lfloor k/2 \rfloor$ `rows of` $\mathbf{M} = \mathbf{T}\mathbf{R}^\intercal$
14 $\mathbf{M}_{2i} \leftarrow (\mathbf{R}^\intercal)_{2i} + (\mathbf{R}^\intercal)_{2i+1}$ `// Sum of rows`
15 $\mathbf{M}_{2i+1} \leftarrow (\mathbf{R}^\intercal)_{2i} - (\mathbf{R}^\intercal)_{2i+1}$ `// Difference of rows`
16 **if** *k is odd* **then** `// Fill in last row if necessary`
17 $\mathbf{M}_{k-1} \leftarrow -(\mathbf{R}^\intercal)_{k-1}$ `// The negative is important`
18 $\mathtt{pk} \leftarrow \mathbf{M}$ `//` $\mathbf{M} \in \mathbb{F}_3^{k\times(n-k)}$
19 **return** $(\mathtt{sk}, \mathtt{pk})$

3.3 Signing

Algorithm 7 defines the WAVELET signing process, which is essentially the same as WAVE signing. We have made no attempt to optimize this procedure beyond the use of bitsliced and vectorized $\mathbb{F}_3$-arithmetic.

The main objective of the WAVE signing procedure is to find a vector $\mathbf{e} \in \mathbb{F}_3^n$ such that

$$|\mathbf{e}| = w\,, \quad \text{and} \quad \mathbf{e}(\mathbf{I}_{n-k} \mid \mathbf{R})^\intercal = \mathbf{h} \tag{6}$$

where $\mathbf{h}$ is the salted hash of the message to be signed, and $\mathbf{R} \in \mathbb{F}_3^{(n-k)\times k}$ is the public key. The public key is generated in such a way that

$$(\mathbf{I}_{n-k} \mid \mathbf{R}) = \mathbf{S}\pi_{\mathrm{sk}}(\mathbf{H}_{\mathrm{sk}}) \quad \text{with} \quad \mathbf{H}_{\mathrm{sk}} = \left(\begin{array}{cc} \mathbf{H}_U\mathbf{D} & -\mathbf{H}_U\mathbf{B} \\ -\mathbf{H}_V\mathbf{C} & \mathbf{H}_V\mathbf{A} \end{array}\right)$$

where $\mathbf{S}$ is a nonsingular matrix corresponding to the Gaussian elimination putting $\pi_{\mathrm{sk}}(\mathbf{H}_{\mathrm{sk}})$ in row-reduced form. Therefore, the solution $\mathbf{e}$ in (6) is also a solution of the system

$$\pi_{\mathrm{sk}}(\mathbf{H}_{\mathrm{sk}})\,\mathbf{e}^\intercal = \mathbf{S}^{-1}\mathbf{h}^\intercal = \pi_{\mathrm{sk}}(\mathbf{H}_{\mathrm{sk}})(\mathbf{h}, \mathbf{0}_k)^\intercal\,.$$

We therefore aim to find some vector $\mathbf{e}'$ such that

$$|\mathbf{e}'| = w \quad \text{and} \quad \mathbf{e}'\mathbf{H}_{\mathrm{sk}}{}^\intercal = \mathbf{h}' \quad \text{where} \quad \mathbf{h}' := \pi_{\mathrm{sk}}(\mathbf{H}_{\mathrm{sk}})(\mathbf{h}, \mathbf{0}_k)^\intercal \tag{7}$$

and output $\mathbf{e} = \pi_{\mathrm{sk}}(\mathbf{e}')$. Using the special structure of $\mathbf{H}_{\mathrm{sk}}$, we see that

$$\mathbf{H}_{\mathrm{sk}}\mathbf{e}'^\intercal = \mathbf{h}' \iff \begin{cases} \mathbf{e}_U^\intercal\mathbf{H}_U = \mathbf{h}^U\,, \\ \mathbf{e}_V^\intercal\mathbf{H}_V = \mathbf{h}^V\,. \end{cases} \tag{8}$$

Algorithm 5: GaussElim

Input: $\mathbf{H} \in \mathbb{F}_3^{R \times C}$ and $\pi \in \mathscr{S}_D$ with $D \leq C$
Output: $\mathbf{H} = (\mathbf{I}_R | \mathbf{M}) \in \mathbb{F}_3^{R \times C}$ where $\mathbf{M} \in \mathbb{F}_3^{R \times C-R}$, and $\pi' \in \mathscr{S}_D$ where the first R positions are the pivots.

Function `GaussElim(`$\mathbf{H}$, π`)`
  pivot $\leftarrow ()$
  nonpivot $\leftarrow ()$
  $(r, c) \leftarrow (0, 0)$
  while $r < R$ **and** $c < D$ **do**
    $(\mathbf{H}, P) \leftarrow$ `ElimSingle(`$\mathbf{H}$, r, $\pi(c)$`)` `// Algorithm 6`
    if P **then**
      $(\mathsf{pivot}, r) \leftarrow (\mathsf{pivot} \parallel (\pi(c)), r + 1)$
    else
      $\mathsf{nonpivot} \leftarrow \mathsf{nonpivot} \parallel (\pi(c))$
    $c \leftarrow c + 1$
  $\pi' \leftarrow \mathsf{pivot} \parallel \mathsf{nonpivot} \parallel (\pi(c), \ldots, \pi(D-1))$
  return $(\pi'(\mathbf{H}), \pi')$

where $\mathbf{h}' = \mathbf{h}^U \parallel \mathbf{h}^V$ and $\mathbf{e}' = \big(\mathbf{e}_U\mathbf{A} + \mathbf{e}_V\mathbf{B}\big) \parallel \big(\mathbf{e}_U\mathbf{C} + \mathbf{e}_V\mathbf{D}\big)$ (note that $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$ are generated in such a way that $(\mathbf{x}, \mathbf{y}) \mapsto \big(\mathbf{x}\mathbf{A} + \mathbf{y}\mathbf{B}, \mathbf{x}\mathbf{C} + \mathbf{y}\mathbf{D}\big)$ is a bijection).

To find $\mathbf{e}$, we solve both linear systems in (8) using the WAVE decoder originally described in WAVE [DST19a]. Pseudocode for the decoder is given in Algorithm 9 in Appendix A. There are two main steps:

1. First, compute any solution $\mathbf{e}_V$ of the undetermined linear system $\mathbf{e}_V\mathbf{H}_V^\intercal = \mathbf{h}^V$
2. Then, compute a *particular* solution $\mathbf{e}_U$ of the second undetermined linear system.

For the second step, the WAVE decoder uses a generalized version of Prange's decoder [Pra62]. Roughly speaking, given $\mathbf{H}_U$ and $\mathbf{h}^U$, when solving the linear system $\mathbf{e}_U{\mathbf{H}_U}^\intercal = \mathbf{h}^U$ (with $n/2$ unknowns and $n/2 - k_U$ equations), we are free to select values of $\mathbf{e}_U$ on k_U coordinates: let's say positions $(0, \ldots, k_U - 1)$ (though these k_U coordinates can be chosen almost anywhere). Since we are looking for a solution $\mathbf{e}' = (\mathbf{e}_U\mathbf{A} + \mathbf{e}_V\mathbf{B}, \mathbf{e}_U\mathbf{C} + \mathbf{e}_V\mathbf{D})$ of large weight, the best strategy is to choose $(e_U)_0, \ldots, (e_U)_{k_U-1}$ such that

$$(\mathbf{e}_U\mathbf{A} + \mathbf{e}_V\mathbf{B})_j \neq 0 \quad \text{and} \quad (\mathbf{e}_U\mathbf{C} + \mathbf{e}_V\mathbf{D})_j \neq 0 \quad \text{for all} \quad 0 \leq j < k_U\,.$$

This is always possible over $\mathbb{F}_3$ given our choice of $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$. The other $n - 2k_U$ coordinates of $\mathbf{e}'$ will be uniformly distributed over $\mathbb{F}_3$, because they are obtained from coordinates of $\mathbf{e}_U$ that we have no control over when solving the random square linear system. Therefore, we typically expect $|\mathbf{e}'| = 2k_U + 2/3(n - 2k_U)$. Choosing $w = 2k_U + 2/3(n - 2k_U)$ ensures that our decoder will succeed after a polynomial number of steps. Now, by carefully choosing k_U, our decoder solves a trapdoor problem: finding $\mathbf{e}$ of weight w such that $\mathbf{e}{\mathbf{H}_{\text{pk}}}^\intercal = \mathbf{h}$ is hard unless we know the hidden structure (via π_{sk}) of $\mathbf{H}_{\text{pk}}$.

Rejection Sampling. The procedure described above has a serious drawback: signatures $\mathbf{e}$ may leak information about π_{sk} (see [DST17, 5.1]). To avoid this, the outputs of the signing algorithm must to be very close to uniformly distributed over words of Hamming weight w. By carefully adjusting with some rejection sampling the Hamming weights of $\mathbf{e}_V$ and $\mathbf{e}_U$, we can meet this property by still producing solutions of weight w for which it is hard to solve the decoding problem for this weight. See [DST19a, 5.1] for more details.

Algorithm 6: Attempted Gaussian elimination on a single column of a matrix.

Input: $\mathbf{H} \in \mathbb{F}_3^{R \times C}$, row index $0 \leq r < R$, column index $0 \leq c < C$
Output: $(\mathbf{H}, P)$ where P is **True**/**False** if a pivot was/was not found in column c.

Function `ElimSingle`($\mathbf{H}$, r, c)
  $p \leftarrow r$
  while $p < R$ **and** $\mathbf{H}_{p,c} = 0$ **do** `// Search for pivot position`
    $p \leftarrow p + 1$
  if $p = R$ **then** `// Pivot not found`
    return $(\mathbf{H}, \mathbf{False})$
  if $\mathbf{H}_{p,c} = 2$ **then**
    $\mathbf{H} \leftarrow$ `NegateRow`($\mathbf{H}$, p) `// Negate` p`-th row`
  $\mathbf{H} \leftarrow$ `SwapRows`($\mathbf{H}$, p, r) `// Move pivot to` r`-th row`
  for i **in** $(0, \ldots, r-1, r+1, \ldots, R-1)$ **do**
    if $\mathbf{H}_{i,c} = 1$ **then**
      $\mathbf{H} \leftarrow$ `SubtractRow`($\mathbf{H}$, i, r) `// Subtract` r`-th row from` i`-th row`
    else if $\mathbf{H}_{i,c} = 2$ **then**
      $\mathbf{H} \leftarrow$ `SumRow`($\mathbf{H}$, i, r) `// Add` r`-th row to` i`-th row`
  return $(\mathbf{H}, \mathbf{True})$

3.4 Verification

Algorithm 8 verifies WAVELET signatures $(r, \mathbf{s})$ under public keys $\mathbf{M}$ following the derivation of §2.4, and using the verification equation (5):

$$|\mathbf{u}| + |\mathbf{s}| = w\,, \quad \text{where} \quad \mathbf{u} = \mathbf{h} + \hat{\mathbf{s}}\mathbf{M} \quad \text{with} \quad \mathbf{h} = \mathtt{Hash}(r \parallel m)\,.$$

We iterate over the signature vector $\mathbf{s}$, computing the entries of the sparser vector $\hat{\mathbf{s}}$ on the fly using (3), and accumulating the corresponding rows of the public key $\mathbf{M}$. Concretely, if w/n is around 0.94 (as it is in *Supertubos*), then we expect Algorithm 8 to use around $0.56k$ row vector operations. Notice that there is only a single pass over the rows of $\mathbf{M}$, so Algorithm 8 is compatible with applications where the public key is streamed.

4 Encodings for keys and signatures

4.1 Basic vector encodings and key sizes

Mathematically, WAVELET works over $\mathbb{F}_3$; but in reality, we must operate on binary values. In practice, we will work with two binary encodings for vectors over $\mathbb{F}_3$.

The *compact* representation views a group of s trits as the coefficients of the ternary expansion of an integer in $[0, 3^s)$, and outputs the binary coefficients of the same integer. This representation achieves 1.6 bits per trit, or 5 trits per byte[5]—which is very close to the optimum of $\log_2(3) \approx 1.585$ bits per trit.

For efficient $\mathbb{F}_3$-vector arithmetic (see §5), we use a *bitsliced* representation. On a β-bit architecture, we store a group of β trits in a pair of machine words: the i-th trit is encoded using the i-th bits of both words. This achieves 2 bits per trit, or 4 trits per byte.

[5]The compact representation packs 5 trits into a byte, or 20 into a 32-bit word, or 40 into a 64-bit word; so packing into words yields no improvement in space efficiency. Encoding to and decoding from words requires more operations on larger integers, too—so we prefer encoding to simple bytes.

Algorithm 7: The WAVELET signing algorithm. This algorithm also produces classic WAVE signatures if we stop at Line 7 and return $(\mathbf{e}, r)$.

1 **Function** `Sign`(m, `sk`)
  Input: $m \in \{0,1\}^*$, `sk` $= (\mathbf{R}_U, \mathbf{R}_V, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \pi_{\text{sk}})$
  Output: $\mathbf{e} \in \mathbb{F}_3^n$, $|\mathbf{e}| = w$
2 $\mathbf{H}_{\text{sk}} \leftarrow$ `ParityCheckUV(`$\mathbf{R}_U$, $\mathbf{R}_V$, $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$`)`
3 $r \xleftarrow{\$} \{0,1\}^{2\lambda}$ `// salt`
4 $\mathbf{h} \leftarrow$ `Hash(`$r \parallel m$`)` `// syndrome in` $\mathbb{F}_3^{n-k}$
5 $\mathbf{x} \leftarrow (\mathbf{h}, \mathbf{0}_k)\pi_{\text{sk}}(\mathbf{H}_{\text{sk}})^\intercal$ `// syndrome adjusted for decoding`
6 $\mathbf{y} \leftarrow$ `Decode(`$\mathbf{x}$, $\mathbf{R}_U$, $\mathbf{R}_V$, $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$`)`
7 $\mathbf{e} \leftarrow \pi_{\text{sk}}(\mathbf{y})$ `//` $(\mathbf{e}, r)$ `= WAVE signature`
8 $\mathbf{s} \leftarrow (e_{n-k}, \ldots, e_{n-1})$ `// Truncate to last` k `entries`
9 **return** $(r, \mathbf{s})$

The public key is essentially a $k \times (n-k)$ random matrix over $\mathbb{F}_3$. In theory, it can be encoded in $\log_2(3)k(n-k) \approx 1.585k(n-k)$ bits. In practice, the compact representation requires $1.6k(n-k)$ bits, while the bitsliced representation requires $\approx 2k(n-k)$ bits.

Technically, the private key can be stored in λ bits, since it suffices to store the seed used to randomly generate the matrices $\mathbf{R}_U$, $\mathbf{R}_V$, $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$ and the permutation π_{sk} in Algorithm 4. However, this means recomputing them every time we sign a message. In practice, then, we want to store the "expanded" private key $(\mathbf{R}_U, \mathbf{R}_V, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \pi_{sk})$. The matrices $\mathbf{R}_U$ and $\mathbf{R}_V$ have sizes $(\frac{n}{2} - k_U) \times \frac{n}{2}$, and $(\frac{n}{2} - k_V) \times \frac{n}{2}$, respectively, while $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$ each have length $\frac{n}{2}$. We therefore need to store $(n-k+4)(n/2)$ trits, along with the permutation π_{sk}, which is a sequence of n integers of size $\lceil \log_2(n) \rceil$.

Table 3 summarizes the performance of each encoding, and gives concrete key sizes using the *Supertubos* parameters. We prefer the compact representation for public key transmission (to minimize bandwidth), but for local storage we prefer the bitsliced representation: the 20% space overhead is a reasonable price to pay to avoid the cost of transcoding from the compact to the bitsliced representation before each vector operation.

4.2 Compressed signature encodings

A WAVELET signature is a pair $(r, \mathbf{s})$, where r is a 2λ-bit salt and $\mathbf{s}$ is a vector in $\mathbb{F}_3^k$. As we mentioned in §2.5, there are very few 0s in $\mathbf{s}$, so the entropy is relatively low, and we can hope to exploit this to compress signatures to substantially less than $\log_2(3)k$ bits.

For *Supertubos*, the probability that a given entry is 1 is 0.47; the probability of 2 is also 0.47; and the probability of 0 is only 0.06. This means 1.267 bits of entropy, which is about 20% smaller than $\log_2(3) \approx 1.585$; theoretically, we can compress an average *Supertubos* WAVELET signature down to 7102 bits (or 888 bytes).

Simple Huffman encoding [Huf52] brings us close to the theoretical limit, as it takes advantage of these frequencies. Huffman coding is a compression technique that uses the frequencies of the characters in a source alphabet to generate a binary tree whose leaves are the characters. The code for each character is defined by the path from the root to the character's leaf: when take the left edge, we add a 0 to the code, and when we take the right edge, we add a 1, or vice-versa. The idea is to build the tree bottom-up, putting the least frequent elements first so that they stay on the lower levels, giving them longer codes.

We will use the slightly more general framework of minimal redundancy prefix codes (MR codes) [MT97, Mof19]. We have two versions of the compression algorithm: the first, *2TMR*, takes $\mathbb{F}_3^2$ as its source alphabet (encoding two trits at a time); the second, *3TMR*, takes $\mathbb{F}_3^3$ as its source alphabet (encoding three trits at a time). Table 4 shows the binary

Algorithm 8: Verification for WAVELET signatures. The entries of $\hat{\mathbf{s}}$ are constructed from $\mathbf{s}$ on the fly using (3), and zero entries are skipped.

Function `STVerify`$(m,\ \sigma,\ \mathbf{M})$
  Input: $m \in \{0,1\}^*$, $\sigma = (r, \mathbf{s}) \in \{0,1\}^{2\lambda} \times \mathbb{F}_3^k$, $\mathbf{M} \in \mathbb{F}_3^{k \times (n-k)}$
  Output: True or False
  $\mathbf{x} \leftarrow \mathtt{Hash}(r \parallel m)$
  for $0 \le i < (k-1)/2$ **do** `// Handle first` $2\lfloor k/2 \rfloor$ `entries of s in pairs`
    $(\hat{s}_{2i}, \hat{s}_{2i+1}) \leftarrow (s_{2i} + s_{2i+1}, s_{2i} - s_{2i+1})$ `//` $(2i, 2i+1)$`-th entries of` $\mathbf{sT}$
    if $\hat{s}_{2i} = 1$ **then**
      $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{M}_{2i}$ `// Add` $2i$`-th row of` $\mathbf{M}$
    else if $\hat{s}_{2i} = 2$ **then**
      $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{M}_{2i}$ `// Subtract` $2i$`-th row of` $\mathbf{M}$
    if $\hat{s}_{2i+1} = 1$ **then**
      $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{M}_{2i+1}$ `// Add` $(2i+1)$`-th row of` $\mathbf{M}$
    else if $\hat{s}_{2i+1} = 2$ **then**
      $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{M}_{2i+1}$ `// Subtract` $(2i+1)$`-th row of` $\mathbf{M}$
  if *k is odd* **then** `// Handle last entry if necessary`
    if $s_{k-1} = 1$ **then**
      $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{M}_{k-1}$
    else if $s_{k-1} = 2$ **then**
      $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{M}_{k-1}$
  return $|\mathbf{s}| + |\mathbf{x}| = w$

Table 3: Binary encodings of vectors over $\mathbb{F}_3$.

Encoding	Size (bits per trit)	Rate (trits per byte)	*Supertubos* key sizes Public	Expanded private
(Optimal)	$\log_2(3) \approx 1.585$	≈ 5.047	$\approx$ *3131* kB	$\approx$ *1350* kB
Compact	1.6	5	$\approx$ 3161 kB	$\approx$ 1363 kB
Bitsliced	2	4	$\approx$ 3951 kB	$\approx$ 1703 kB

outputs for these encodings, derived using the usual MR code system based on the average symbol frequencies for *Supertubos* WAVELET signatures. Observe that 2TMR and 3TMR are *prefix-free*: no output code is a prefix of any other.

To compress, we break the signature vector $\mathbf{s}$ into a sequence of pairs (or triples), and output the concatenation of the corresponding codes. For decompression, we look at our input (a sequence of bytes) as a stream of bits; the prefix-free property of the 2TMR and 3TMR codes allows us to quickly and unambiguously recognise codewords, and output the corresponding pairs (or triples) of trits, using the algorithm of [MT97]. The decoding algorithm requires a few pre-computed lists for fast lookup; we give the relevant data, along with details on the compression and decompression algorithms, in Appendix C.

Since Huffman and MR coding is variable-length, compressed WAVELET signatures have variable length. An extra level of variation is introduced by the fact that the supposed symbol probabilities are not exact. For example, while we know that there are exactly $n - w$ zeroes in a full-length WAVE signature vector, when we move to truncated WAVELET signatures the number of zeroes follows a binomial distribution centred on $k - wk/n$.

Figure 2 shows the lengths of 22000 random *Supertubos* WAVELET signatures compressed

Table 4: 2TMR and 3TMR encodings for *Supertubos* WAVELET signatures

2TMR		**3TMR**					
Pair	Code	Triple	Code	Triple	Code	Triple	Code
$(2,2)$	00	$(2,2,2)$	000	$(2,0,2)$	110101	$(1,0,1)$	111110
$(2,1)$	01	$(2,2,1)$	001	$(0,2,2)$	110110	$(0,1,1)$	1111110
$(1,2)$	100	$(2,1,2)$	010	$(2,1,0)$	110111	$(2,0,0)$	111111100
$(1,1)$	101	$(1,2,2)$	011	$(2,0,1)$	111000	$(0,2,0)$	1111111010
$(2,0)$	1100	$(2,1,1)$	100	$(0,2,1)$	111001	$(0,0,2)$	1111111011
$(1,0)$	1101	$(1,2,1)$	1010	$(1,2,0)$	111010	$(1,0,0)$	1111111100
$(0,2)$	1110	$(1,1,2)$	1011	$(1,0,2)$	111011	$(0,1,0)$	1111111101
$(0,1)$	11110	$(1,1,1)$	1100	$(0,1,2)$	111100	$(0,0,1)$	1111111110
$(0,0)$	11111	$(2,2,0)$	110100	$(1,1,0)$	111101	$(0,0,0)$	1111111111

with the 2TMR and 3TMR encodings. We see that experimentally, 2TMR-compressed *Supertubos* WAVELET signature vectors fit in 950 bytes on the average (within 5% of the theoretical optimum), and almost always less than 970 bytes; 3TMR-compressed *Supertubos* WAVELET signature vectors fit in 910 bytes on the average (within 2.5% of the optimum), and almost always less than 930 bytes. Once we have included 32 bytes worth of salt, *Supertubos* WAVELET signatures fit comfortably within one kilobyte.

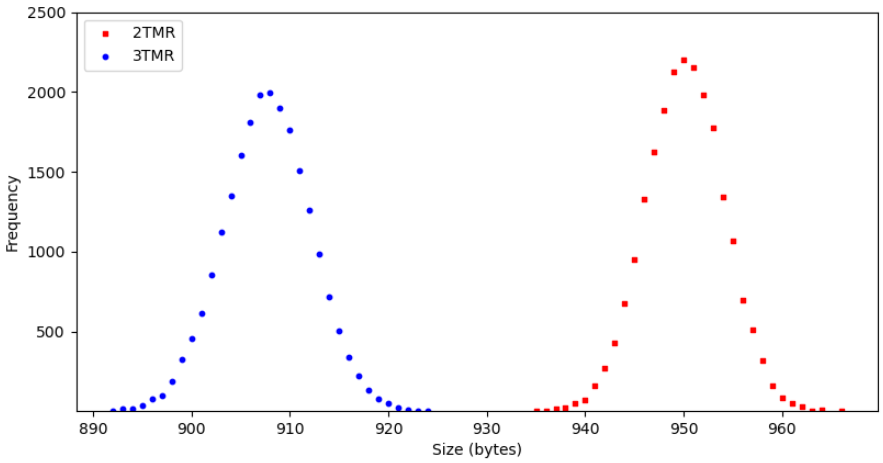

Figure 2: Frequencies of 2TMR and 3TMR encoding lengths (in bytes) for 22 000 random *Supertubos* WAVELET signatures.

5 Software implementations

This section describes the techniques we used to put the algorithms of §3 into practice on an Intel® Core™ platform with AVX, and also on an ARM Cortex-M4 platform.

5.1 Efficient ternary vector arithmetic

Our implementation uses the bitsliced $\mathbb{F}_3$-vector arithmetic described in [Coo13]; the underlying $\mathbb{F}_3$-arithmetic is the *Type-2* representation of [KAT08, §4.2]. We use $\oplus$, &, |, and $\neg$ to denote logical XOR, AND, OR, and NOT, respectively.

Each element a in $\mathbb{F}_3$ is represented using a pair of bits:

$$\mathbb{F}_3 \ni a \longleftrightarrow (a_h, a_l) \in \{0,1\}^2 ;$$

the encoding of [KAT08, §4.2] uses

$$0 \longleftrightarrow (0,0) , \qquad 1 \longleftrightarrow (0,1) , \qquad 2 \longleftrightarrow (1,1) .$$

Additions and subtractions can be computed at a cost of 7 logical operations each using the identities

$$\big((a+b)_h, (a+b)_l\big) = \big((a_l \oplus b_h) \& (a_h \oplus b_l) , (a_l \oplus b_l) | ((a_h \oplus b_l) \oplus b_h)\big) , \tag{9}$$

$$\big((a-b)_h, (a-b)_l\big) = \big((a_l \oplus (b_h \oplus b_l)) \& (a_h \oplus b_l) , (a_l \oplus b_l) | (a_h \oplus b_h)\big) ; \tag{10}$$

pure negations can be computed using

$$\big((-a)_h, (-a)_l\big) = \big(a_h \oplus a_l, a_l\big) . \tag{11}$$

This representation lends itself well to bitslicing [Bih97]. Working on a platform with β-bit machine words, we break $\mathbb{F}_3$-vectors down into a sequence of subvectors $\mathbf{a}$ in $\mathbb{F}_3^\beta$, each of which is encoded as a pair of β-bit words:

$$\mathbb{F}_3^\beta \ni \mathbf{a} \longleftrightarrow (\mathbf{a}_h, \mathbf{a}_l) := \big(\big((a_0)_h, \ldots, (a_{\beta-1})_h\big), \big((a_0)_l, \ldots, (a_{\beta-1})_l\big)\big) .$$

The same sequences of logical operations defining the addition, negation, and subtraction formulæ above, applied to the machine words $\mathbf{a}_h$ and $\mathbf{a}_l$, thus compute addition, negation, and subtraction in parallel on β bits at a time.

AVX instructions. Going further in this direction, many computer architectures have special instructions to vectorize logical operations across multiple words. In our x86_64 implementation of verification, we use AVX2 instructions to further speed up the bitsliced $\mathbb{F}_3$-vector arithmetic above by moving from $\beta = 64$ to $\beta = 256$. More precisely, we use Intel® intrinsics including _mm256_xor_si256, _mm256_or_si256, and _mm256_and_si256, as well as special instructions to load and store the 256-bit words.

5.2 Key Generation

Algorithm 4 requires a pseudorandom bit generator to sample the matrices, vectors, and permutation. In our implementation we use the SHAKE256 XOF, seeded with a λ-bit random string (which can be used as a compressed representation of the private key if desired). To generate the random permutation in $\mathscr{S}_n$, we construct the list of integers from 0 to $n-1$, and then shuffle it using the modern Fisher-Yates Shuffle algorithm [Knu97].

We also used SHAKE256 to generate the randomness required in our other algorithms—though for security, each of these uses a different random string as initial state.

5.3 Signing

Apart from the use of `Hash` (Algorithm 1), our signing implementation is heavily based on the version of `Decode` included in the proof-of-concept Wave trapdoor code from [DST19b]. The implementation of [DST19b] generates trapdoor challenges, using random trits in place of hash function output. (Indeed, it neither specifies nor uses a hash function, so it cannot be used to sign messages as is.) Apart from some use of bitsliced arithmetic as in §5.1 for vector-matrix multiplication, we made absolutely no attempt to optimize or improve the signing algorithm.

Indeed, we privileged correctness and safety (and coherence with [DST19a]) over speed for signing: any modifications to the original algorithm must be accompanied by a subtle and difficult verification of the distributions produced by the decoder. A verified *and* optimized signing algorithm is an important future goal for WAVE development.

For completeness, we include pseudocode for `Decode` as Algorithm 9 in Appendix A. Our implementation uses precomputed tables for the distribution of rejection with `SampleV` and `SampleU`, but these values could be recomputed on the fly using the parameters proposed in Table 2 and in Appendix B.

To hash into $\mathbb{F}_3^{n-k}$, our implementation uses Algorithm 1 with SHA3-512 as `BinaryHash` and SHAKE-256 as the XOF in `Expand`.

To sign with Algorithm 7, we need to (re)generate the parity-check matrix. While it is possible to avoid this by storing the parity check matrix with the private key, this imposes an unacceptably high space overhead (this matrix is much bigger than the expanded private key, which is already very large).

Remark 2. We remind the reader that the signing process requires great care. One needs to respect the parameters and distributions involved in the rejection sampling to produce an output that is statistically indistinguishable from a random word of weight w. Any deviation from this distribution could result in an attack on the signature scheme.

5.4 Verification

In WAVE, the verification process consist into check the weight of a vector, and compare the hash of the message. However, we change this to simplify the verification process.

In summary, Algorithm 8 consists of a for loop with additions and subtractions of vectors. For those operations, we use the efficient ternary vector arithmetic. After the loop, we need to check the weight of $\mathbf{s}$ and $\mathbf{x}$. To check the weight of a vector using Type 2 encoding, we only need to count the number of ones in the low part of the encoding, that is, given a vector $\mathbf{a} = (\mathbf{a_l}, \mathbf{a_h})$ we only need to compute $\texttt{BinaryWeight}(\mathbf{a_l})$.

The Hamming weight, or `BinaryWeight` function can be implemented by counting the number of 1's in the binary representation of an integer, or in the case the amount of 1 bits in the register. The binary weight for AVX2 can be computed using [MKL18]. However, the generic code for other architectures needs to be done without AVX instructions. Listing 1 shows the method for 32-bit words; the technique extends easily to 64-bit words.

Listing 1: BinaryWeight for 32 bit architetures.

```
static uint32_t BinaryWeight (uint32_t n)
  {
      uint32_t m1 = (~(wave_word)0) / 3;
      uint32_t m2 = (~(wave_word)0) / 5;
      uint32_t m4 = (~(wave_word)0) / 17;

      n -= (n >> 1) & m1;
      n = (n & m2) + ((n >> 2) & m2);
      n = (n + (n >> 4)) & m4;
      n += n >>  8;
      n += n >> 16;
      return n & 0x7F;
  }
```

5.5 Cortex-M4 implementation

Implementing WAVELET verification for an ARM Cortex-M4 platform presents several interesting challenges.

QSPI and Flash. WAVELET has relatively compact signatures, but large public keys—generally far too large for the RAM available in many small devices. In our case, the nRF52840-dev Kit provides 256 kB RAM. However, it also has 64 MB of external QSPI (Queued Serial Peripheral Interface) flash memory, which allows reads of up to 16 MB/sec using the `nrf_drv_qspi_read` instruction.[6] For this board, therefore, we can flash the public key into the external flash and read it later; this is a useful approach for applications where a device must verify signatures from a signer known at installation time, e.g. for secure software updates.

While this allows us to get the entire WAVELET public key onto the device, we still cannot load the entire matrix into SRAM. In our Cortex-M4 implementation of Algorithm 8 we load one row at a time, as and when it is used in a computation. To fetch the data we have a base address, from which we can easily compute the address for each required row.

Streaming vs Flashing public keys. Another way to handle big keys is to stream them; this approach has been used for other postquantum signatures on Cortex-M4 [GHK+21]. However, as noted in [CC21], this design choice depends on the application. In our case, we have two reasons to keep the public key in the flash. First, streaming the data is slower: streaming speeds in [GHK+21] were around 500 kbits/s, which is slower than QSPI. Second, streamed keys require validation, which would complicate and slow verification.

6 Experimental results

In our tests, we ran the code on an Intel® Core™ i7-10610U processor under Arch Linux (Kernel 5.14.11), compiling with GCC version 11.1.0. We counted cycles using `cpucycles.h` from SUPERCOP[7], disabling Turbo Boost and SpeedStep. We first ran our tests 100 times to clean the cache, then 100 more times to get the average number of cycles.

Table 5 shows the results of our implementation for KeyGen, Sign, and WAVELET Verification. We also include timings for an implementation of classic WAVE verification, based on the same underlying arithmetic, for comparison. We implemented both verifications twice: once with AVX instructions and one without them. (KeyGen and Sign do not use AVX instructions, since we focused on optimizing verification.)

We also tested verification in a Nordic nRF52840 Development Kit, which provides an ARM Cortex-M4 microcontroller running at 64 MHz, with 256 kB RAM, 1 MB flash and 64 MB of external memory. We compile the code using GNU Arm Embedded Toolchain version 11.2.0. We used the timer library provided by Nordic Semiconductors to measure running time in ms and in "ticks". Table 6 presents these results.

Timings for signature compression and decompression appear in Table 7 (Intel® Core™) and Table 8 (ARM Cortex-M4). Timings were measured in the same way as above. We see that (de)compression times are negligible in comparison with the other operations.

Our code is available from `https://github.com/waveletc/wavelet`.

7 Conclusion

This work presents the first complete implementation of the WAVE postquantum signature scheme. Our variant, WAVELET, includes important optimizations—from the highest level to the lowest—yielding much shorter signatures and a much faster verification algorithm.

To interface this fundamentally ternary signature scheme with the binary world, we defined a practical hash function and a highly efficient compression algorithm for signatures,

[6] Function documented at `https://infocenter.nordicsemi.com/index.jsp?topic=/com.nordic.infocenter.sdk5.v14.2.0/group__nrf__drv__qspi.html`.

[7] `https://bench.cr.yp.to/supercop.html`

Table 5: WAVE and WAVELET signature scheme timings on an Intel® Core™.

Operation	Optimization	Time (ms)	Cycles
Key Generation	`-O3`	3144.00	7 403 069 461
Sign	`-O3`	702.77	1 644 281 062
WAVE Verification	`-O3`	2.20	5 098 394
	AVX instr	2.19	5 064 048
WAVELET Verification	`-O3`	1.19	2 733 457
	AVX instr	0.45	1 087 538

Table 6: WAVELET verification time on an ARM Cortex-M4 (nRF52840-DK).

Operation	Optimization	Time (ms)	Ticks
WAVELET Verification_static	`-O2s`	402	13172

Table 7: WAVELET signature compression and decompression timings on an Intel® Core™.

Encoding	Operation	Optimization	Time (ms)	Cycles
2TMR	Compression	`-O3`	0.041	79 529
	Decompression	`-O3`	0.043	81 543
3TMR	Compression	`-O3`	0.065	126 230
	Decompression	`-O3`	0.057	109 546

Table 8: WAVELET signature compression and decompression timings on an ARM Cortex-M4 (nRF52840-DK).

Encoding	Operation	Optimization	Time (ms)	Ticks
2TMR	Compression	`-O2s`	3	123
	Decompression	`-O2s`	7	231
3TMR	Compression	`-O2s`	3	105
	Decompression	`-O2s`	5	171

taking advantage of their unusually high weight. This yields even shorter signatures at very little cost. For signatures targeting the 128-bit classical security level (and NIST Level 1 postquantum security), the *Supertubos* parameter set yields WAVELET signatures that fit in less than a kilobyte. WAVELET signatures are therefore the smallest code-based signatures at this security level, and are competitive with signatures across other post-quantum paradigms.

We have also demonstrated the feasibility of WAVELET verification on ARM Cortex-M4 microcontroller platforms. While WAVELET public keys are generally much bigger than the available SRAM, our implementation exploits large external flash where it is available. While memory access latency remains a major bottleneck, *Supertubos* WAVELET signatures decompress and verify in under half a second on the Nordic nRF52840 Development Kit board. Our verification algorithm should also be compatible with streaming public keys, similar to the approach of [GHK+21], though we have not attempted to implement this.

Future Work. Our results highlight some interesting outstanding problems, both theoretical and practical. On the theoretical side, WAVE for $q > 3$ needs more analysis.

The derivation of alternative Wave parameter sets—targeting NIST Levels 3 and 5, for example, or tuning key and signature sizes for better performance in embedded and other environments—is also a priority.

On the practical side, key generation and signing must be improved and optimized. Another crucial problem is to develop a signing algorithm that is secure against side-channel attacks, or at least constant-time. For the current signing algorithm, with its complexity and dependence on rejection sampling, this is a highly nontrivial problem.

References

[ABB+] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mentel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. SPHINCS+ stateless hash-based signatures.

[ABD+] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS/Dilithium. URL: `https://pq-crystals.org/`. `https://pq-crystals.org/`.

[ABD+21] Nicolas Aragon, Marco Baldi, Jean-Christophe Deneuville, Karan Khathuria, Edoardo Persichetti, and Paolo Santini. Cryptanalysis of a code-based full-time signature. *Des. Codes Cryptogr.*, 89(9):2097–2112, 2021.

[BBC+13] Marco Baldi, Marco Bianchi, Franco Chiaraluce, Joachim Rosenthal, and Davide Schipani. Using LDGM codes and sparse syndromes to achieve digital signatures. In Philippe Gaborit, editor, *Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings*, volume 7932 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2013.

[Bih97] Eli Biham. A fast new DES implementation in software. In Eli Biham, editor, *Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1997.

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures - how to sign with RSA and Rabin. In Ueli M. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer, 1996.

[CC21] Ming-Shing Chen and Tung Chou. Classic McEliece on the ARM Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):125–148, 2021.

[CDG+] Melissa Chase, David Derler, Steven Goldfeder, Daniel Kales, Jonathan Katz, Vladimir Kolesnikov, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Xiao Wang, and Greg Zaverucha. Picnic: A family of post-quantum secure digital signature algorithms. `https://eprint.iacr.org/2017/279`.

[CFMR+] Antoine Casanova, Jean-Charles Faugère, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret, and Jocelyn Ryckeghem. GeMSS: A GrEat Multivariate Short Signature. URL: `https://www-polsys.lip6.fr/Links/NIST/GeMSS.html`. `https://www-polsys.lip6.fr/Links/NIST/GeMSS.html`.

[Coo13] Kris Coolsaet. Fast vector arithmetic over $\mathbb{F}_3$. *Bulletin of the Belgian Mathematical Society – Simon Stevin*, 20(2):329–344, 2013.

[Cor02] Jean-Sébastien Coron. Optimal security proofs for PSS and other signature schemes. In Lars R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 272–287. Springer, 2002.

[DST17] Thomas Debris-Alazard, Nicolas Sendrier, and Jean-Pierre Tillich. The problem with the SURF scheme. preprint, November 2017. arXiv:1706.08065.

[DST19a] Thomas Debris-Alazard, Nicolas Sendrier, and Jean-Pierre Tillich. Wave: A new family of trapdoor one-way preimage sampleable functions based on codes. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part I*, volume 11921 of *Lecture Notes in Computer Science*, pages 21–51. Springer, 2019.

[DST19b] Thomas Debris-Alazard, Nicolas Sendrier, and Jean-Pierre Tillich. Wave implementation. `https://wave.inria.fr/en/implementation/`, 2019. Accessed: 2021-09-14.

[FHK+] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-Fourier lattice-based compact signatures over NTRU. URL: `https://falcon-sign.info`. `https://falcon-sign.info`.

[GHK+21] Ruben Gonzalez, Andreas Hülsing, Matthias J. Kannwischer, Juliane Krämer, Tanja Lange, Marc Stöttinger, Elisabeth Waitz, Thom Wiggers, and Bo-Yin Yang. Verifying post-quantum signatures in 8 kB of RAM. In Jung Hee Cheon and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021, Daejeon, South Korea, July 20-22, 2021, Proceedings*, volume 12841 of *Lecture Notes in Computer Science*, pages 215–233. Springer, 2021.

[Huf52] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[KAT08] Yuto Kawahara, Kazumaro Aoki, and Tsuyoshi Takagi. Faster implementation of eta-T pairing over GF(3^m) using minimum number of logical instructions for GF(3)-addition. In Steven D. Galbraith and Kenneth G. Paterson, editors, *Pairing-Based Cryptography - Pairing 2008, Second International Conference, Egham, UK, September 1-3, 2008. Proceedings*, volume 5209 of *Lecture Notes in Computer Science*, pages 282–296. Springer, 2008.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition, 1997.

[MCF19] David McGrew, Michael Curcio, and Scott Fluhrer. RFC 8554: Leighton-Micali hash-based signatures. IETF Request for Comments, April 2019.

[MKL18] Wojciech Muła, Nathan Kurz, and Daniel Lemire. Faster population counts using AVX2 instructions. *The Computer Journal*, 61(1):111–120, 2018.

[Mof19] Alistair Moffat. Huffman coding. *ACM Computing Surveys*, 52(4), August 2019.

[MT97] Alistair Moffat and Andrew Turpin. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, 1997.

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

[PT16] Aurélie Phesso and Jean-Pierre Tillich. An efficient attack on a code-based signature scheme. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings*, volume 9606 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2016.

[SHM+20] Yongcheng Song, Xinyi Huang, Yi Mu, Wei Wu, and Huaxiong Wang. A code-based signature scheme from the Lyubashevsky framework. *Theor. Comput. Sci.*, 835:15–30, 2020.

A The Wave decoder

Algorithm 9 is the WAVE decoder, which finds the error vector $\mathbf{e}$ with Hamming weight w such that $\mathbf{e}\mathbf{H}_{\mathrm{sk}} = \mathbf{s}$. It uses the precomputed data and rejection sampling parameters from Appendix B. It uses `ElimSingle` (Algorithm 6) from key generation, and auxiliary functions `FreeSetV` (Algorithm 10) and `FreeSetU` (Algorithm 11).

Algorithm 9: The WAVE decoder

Function `Decode`($\mathbf{s}$, $\mathbf{R}_U$, $\mathbf{R}_V$, $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$)
  Input: $\mathbf{s} \in \mathbb{F}_3^{n-k}$, $\mathbf{R}_U \in \mathbb{F}_3^{(n/2-k_U)\times k_U}$, $\mathbf{R}_V \in \mathbb{F}_3^{(n/2-k_V)\times k_V}$, $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d} \in \mathbb{F}_3^{n/2}$
  Output: $\mathbf{e} \in \mathbb{F}_3^n$ such that $|\mathbf{e}| = w$ and $\mathbf{e}\mathbf{H}_{\mathrm{sk}}{}^{\mathsf{T}} = \mathbf{s}$
    where $\mathbf{H}_{\mathrm{sk}} =$ `ParityCheckUV`($\mathbf{R}_U$, $\mathbf{R}_V$, $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$)
  $\mathbf{s}_U \leftarrow (s_0, \ldots, s_{n/2-k_U-1})$ // $\mathbf{s}_U \in \mathbb{F}_3^{n/2-k_U}$
  $\mathbf{s}_V \leftarrow (s_{n/2-k_U}, \ldots, s_{n-k-1})$ // $\mathbf{s}_V \in \mathbb{F}_3^{n/2-k_V}$
  $\mathbf{H} \leftarrow (\mathbf{I}_{n/2-k_V} \mid \mathbf{R}_V)$ // $\mathbf{H} \in \mathbb{F}_3^{(n/2-k_V)\times n/2}$
  repeat // rejection sampling
    $\ell \leftarrow$ `SampleV`()
    $\mathbf{x}_V \xleftarrow{\$} \mathbb{F}_3^d$
    $\mathbf{x}'_V \xleftarrow{\$} \{\mathbf{x} \in \mathbb{F}_3^{k_V-d} : |\mathbf{x}| = \ell\}$
    $\left(\left(\mathbf{I}_{n/2-k_V} | \mathbf{A}\right), \mathbf{s}_0, \pi_V\right) \leftarrow$ `FreeSetV`($\mathbf{H}, \mathbf{s}_V$)
    $\mathbf{e}_V \leftarrow \pi_V^{-1}\left(\left(\mathbf{s}_0 - (\mathbf{x}_V \parallel \mathbf{x}'_V)\,\mathbf{A}^{\mathsf{T}}\right) \parallel (\mathbf{x}_V \parallel \mathbf{x}'_V)\right)$
    $t \leftarrow |\mathbf{e}_V|$
  until `AcceptSampleV`(t)
  $\mathbf{H} \leftarrow (\mathbf{I}_{n/2-k_U} \mid \mathbf{R}_U)$ // $\mathbf{H} \in \mathbb{F}_3^{(n/2-k_U)\times n/2}$
  $\mathscr{E}_V \leftarrow$ `Support`($\mathbf{e}_V$) // `Support(x) = list of coordinates where` $x_i \neq 0$
  $\mathbf{a}_V \leftarrow (\mathbf{b} \odot \mathbf{e}_V \parallel \mathbf{d} \odot \mathbf{e}_V)$ // for $\mathbf{x}, \mathbf{y} \in \mathbb{F}_3^{n/2}$, $\mathbf{x} \odot \mathbf{y} := (x_i y_i)_{0 \le i \le n/2-1}$
  repeat // rejection sampling
    $k_{\neq 0} \leftarrow$ `SampleU`(t)
    $\left(\left(\mathbf{I}_{n/2-k_U} | \mathbf{A}\right), \mathbf{s}_1, \pi_U\right) \leftarrow$ `FreeSetU`($k_{\neq 0}$, $\mathscr{E}_V$, $\mathbf{H}$, $\mathbf{s}_U$)
    $(z_i)_{0 \le i \le n/2-1} \leftarrow \pi_U(\mathbf{e}_V)$
    repeat
      $\mathbf{x}_U \xleftarrow{\$} \mathbb{F}_3^d$
      $\mathbf{x}'_U \xleftarrow{\$} \left(\mathbb{F}_3 \backslash \{0\}\right)^{k_U - k_{\neq 0} - d}$
      $\mathbf{x}''_U \leftarrow (x_j)_{0 \le j < k_{\neq 0}}$ **where**
        $x_j \leftarrow \frac{b_u}{a_u} z_{n/2-k_{\neq 0}+j} + \frac{d_u}{c_u} z_{n/2-k_{\neq 0}+j}$ where $u := \pi_U^{-1}(j)$
      $\mathbf{e}_U \leftarrow \pi_U^{-1}\left(\left(\mathbf{s}_1 - (\mathbf{x}_U \parallel \mathbf{x}'_U \parallel \mathbf{x}''_U)\,\mathbf{A}^{\mathsf{T}}\right) \parallel (\mathbf{x}_U \parallel \mathbf{x}'_U \parallel \mathbf{x}''_U)\right)$
      $\mathbf{a}_U \leftarrow (\mathbf{a} \odot \mathbf{e}_U \parallel \mathbf{c} \odot \mathbf{e}_U)$
      $\mathbf{e} \leftarrow \mathbf{a}_V + \mathbf{a}_U$
    until $|\mathbf{e}| = w$
  until `AcceptSampleU`($m_1(\mathbf{e})$, t) **where**
    $m_1(\mathbf{x}) := \#\{0 \le i \le \frac{n}{2} - 1 : x_i \neq x_{\frac{n}{2}+i}\}$
  return $\mathbf{e}$

The `FreeSetV` and `FreeSetU` algorithms are superficially similar, but there are important differences in their input and permutation sampling. In particular, `FreeSetU` uses an auxiliary function `RandPermSet`(n, $k_{\neq 0}$, $(i_0, \ldots, i_{t-1})$), which samples a random

permutation $\pi \in \mathscr{S}_n$ subject to the constraints

$$\{\pi(0), \ldots, \pi(t - k_{\neq 0} - 1)\} \subseteq (i_0, \ldots, i_{t-1})$$

and

$$\{\pi(t - k_{\neq 0}), \ldots, \pi(n - k_{\neq 0} - 1)\} = \{1, \ldots, n\} \backslash \{i_0, \ldots, i_{t-1}\}\,.$$

Algorithm 10: FreeSetV

Input: $\mathbf{H} \in \mathbb{F}_3^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_3^{n-k}$
Output: $\mathbf{H} = (\mathbf{I}_{n-k}|\mathbf{M}) \in \mathbb{F}_3^{(n-k)\times n}$ where $\mathbf{M} \in \mathbb{F}_3^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_3^{n-k}$, $\pi \in \mathscr{S}_n$

Function `FreeSetV`$(\mathbf{H}, \mathbf{s})$
  repeat
    $\pi \xleftarrow{\$} \mathscr{S}_n$
    pivot $\leftarrow ()$
    nonpivot $\leftarrow ()$
    $(r, c) \leftarrow (0, 0)$
    while $r < n - k$ **and** $c < n$ **do**
      $(\mathbf{H} \parallel \mathbf{s}^\intercal, P) \leftarrow$ `ElimSingle`$(\mathbf{H} \parallel \mathbf{s}^\intercal,\ r,\ \pi(c))$ `// Algorithm 6`
      if P **then**
        $(\mathsf{pivot}, r) \leftarrow (\mathsf{pivot} \parallel (\pi(c)), r + 1)$
      else
        $\mathsf{nonpivot} \leftarrow \mathsf{nonpivot} \parallel (\pi(c))$
      $c \leftarrow c + 1$
  until $\#\mathsf{nonpivot} \leq d$
  $\pi \leftarrow \mathsf{pivot} \parallel \mathsf{nonpivot} \parallel (\pi(c), \ldots, \pi(n-1))$
  return $(\pi(\mathbf{H}), \mathbf{s}, \pi)$

B Rejection sampling parameters

There are two rejection sampling steps in the signing algorithms of Wave (specifically, in the Wave decoder). We split the description of their parameters in two parts: the *V-part* and the *U-part*. We write

$$k'_V := k_V - d \quad \text{and} \quad k'_U := k_U - d$$

where k_U, k_V and d are given in Table 2.

The V-part. This first rejection sampling step involves several functions on $\{0, \ldots, n/2\}$, namely

$$f_V^{\mathrm{rs}}(i) := \frac{1}{\max\limits_{0 \leq j \leq n/2} \frac{q_1^{\mathrm{unif}}(j)}{q_1(j)}} \, \frac{q_1^{\mathrm{unif}}(i)}{q_1(i)}\,,$$

$$q_1^{\mathrm{unif}}(i) := \frac{\binom{n/2}{i}}{\binom{n}{w} 2^{w/2}} \sum_{\substack{p=0 \\ w+p \equiv 0 \mod 2}}^{i} \binom{i}{p} \binom{n/2 - i}{(w+p)/2 - i} 2^{3p/2}\,,$$

and

$$q_1(i) := \sum_{t=0}^{i} \frac{\binom{n/2-k'_V}{i-t} 2^{i-t}}{3^{n/2-k'_V}} p_V(t)\,,$$

Algorithm 11: FreeSetU

Input: $\mathscr{E} \subseteq \{0,\dots,n-1\}$, $k_{\neq 0} \in \{0,\dots \#\mathscr{E}\}$, $\mathbf{H} \in \mathbb{F}_3^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_3^{n-k}$
Output: $\mathbf{H} = (\mathbf{I}_{n-k}|\mathbf{M}) \in \mathbb{F}_3^{(n-k)\times n}$ where $\mathbf{M} \in \mathbb{F}_3^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_3^{n-k}$, $\pi \in \mathscr{S}_n$

Function `FreeSetU(`$k_{\neq 0}$, $\mathscr{E}$, $\mathbf{H}, \mathbf{s}$`)`
  repeat
    $\pi \leftarrow$ `RandPerm(`n, $k_{\neq 0}$, $\mathscr{E}$`)`
    pivot $\leftarrow ()$
    nonpivot $\leftarrow ()$
    $(r, c) \leftarrow (0, 0)$ **while** $r < n - k$ **and** $c < n$ **do**
      $(\mathbf{H} \parallel \mathbf{s}^\mathsf{T}, P) \leftarrow$ `ElimSingle(`$\mathbf{H} \parallel \mathbf{s}^\mathsf{T}, r, \pi(c)$`)` `// Algorithm 6`
      if P **then**
        $(\mathsf{pivot}, r) \leftarrow (\mathsf{pivot} \parallel (\pi(c)), r + 1)$
      else
        $\mathsf{nonpivot} \leftarrow \mathsf{nonpivot} \parallel (\pi(c))$
      $c \leftarrow c + 1$
  until $\#\mathsf{nonpivot} \leq d$
  $\pi \leftarrow \mathsf{pivot} \parallel \mathsf{nonpivot} \parallel (\pi(c), \dots, \pi(n-1))$
  return $(\pi(\mathbf{H}), \mathbf{s}, \pi)$

where the function $p_V(\cdot)$ is a system parameter for the V-part of the rejection sampling.

The U-part. This second rejection sampling step involves several functions on the domain

$$\{(s,t) : t_{\min} \le t \le t_{\max}, 0 \le s \le \min(t, n-w), s \equiv w \pmod 2\}$$

where

$$t_{\min} := 1953 \qquad \text{and} \qquad t_{\max} := 2745\,;$$

these functions are

$$f_U^{\mathrm{rs}}(s,t) := \frac{1}{\max\limits_{0\le u\le v} \frac{q_2^{\mathrm{unif}}(u,v)}{q_2(u,v)}} \, \frac{q_2^{\mathrm{unif}}(s,t)}{q_2(s,t)} \tag{12}$$

where

$$q_2^{\mathrm{unif}}(s,t) := \frac{\binom{t}{s}\binom{n/2-t}{\frac{w+s}{2}-t}2^{\frac{3s}{2}}}{\sum\limits_p \binom{t}{p}\binom{n/2-t}{\frac{w+p}{2}-t}2^{\frac{3p}{2}}}$$

and

$$q_2(s,t) := \sum_{\substack{t+k_U'-n/2\le k_{\neq 0}\le t \\ k_0 = k_U' - k_{\neq 0}}} \frac{\binom{t-k_{\neq 0}}{s}\binom{n/2-t-k_0}{\frac{w+s}{2}-t-k_0}2^{\frac{3s}{2}}}{\sum\limits_p \binom{t-k_{\neq 0}}{p}\binom{n/2-t-k_0}{\frac{w+p}{2}-t-k_0}2^{\frac{3p}{2}}} \, p_U(k_{\neq 0}, t)$$

where the function $p_U(\cdot, t)$ is a system parameter for the U-part.

Precomputed Data. The values for $(p_V(t))_{1\le i\le n/2}$ and $(p_U(\cdot,t))_{t_{\min}\le t\le t_{\max}}$ used in our implementation can be found at `https://github.com/waveletc/wavelet` in the file `precomputedData.txt`. We precomputed and stored the necessary values of f_V^{rs} and $(f_U^{\mathrm{rs}}(\cdot,t))_{t_{\min}\le t\le t_{\max}}$ to 128-bit precision, to save recomputing f_V^{rs} and f_U^{rs} each time they are called. This requires significant storage; if this is not available, then the values can always be computed on the fly using the formulas above.

C Signature compression and decompression algorithms

For efficient signature compression and decompression using 2TMR and 3TMR, we follow the algorithms of Moffat and Turpin [MT97]. Tables 9 and 10 give the precomputed arrays required by the decompression algorithm. The notation $[x] * n$ means a list consisting of the number x, repeated n times.

Table 9: Pre-computed lists for fast 2TMR decoding using the algorithm of [MT97].

`Code_len`	$[2, 2, 3, 3, 4, 4, 4, 5, 5]$
`First_symbol`	$[0, 0, 0, 2, 4, 7, 10]$
`First_code_r`	$[0, 0, 0, 4, 12, 30, 32]$
`First_code_l`	$[0, 0, 0, 16, 24, 30, 32]$
`Search_start`	$[2] * 16 \parallel [3] * 8 \parallel [4] * 5 \parallel [5] * 3$

Table 10: Pre-computed lists for fast 3TMR decoding using the algorithm of [MT97].

`Code_len`	$[3] * 5 \parallel [4] * 3 \parallel [6] * 11 \parallel [7, 9] \parallel [10] * 6$
`First_symbol`	$[0, 0, 0, 0, 5, 8, 8, 19, 20, 20, 21, 27]$
`First_code_r`	$[0, 0, 0, 0, 10, 10, 52, 126, 126, 508, 1018, 1024]$
`First_code_l`	$[0, 0, 0, 0, 640, 832, 832, 1008, 1016, 1016, 1018, 1024]$
`Search_start`	$[3] * 640 \parallel [4] * 192 \parallel [6] * 176 \parallel [7] * 8 \parallel [9] * 2 \parallel [10] * 6$

D Mathematical proofs

Proof of Proposition 1. Recall that for each $\mathbf{s}$ in $\mathbb{F}_3^k$, the vector $\hat{\mathbf{s}}$ in $\mathbb{F}_3^k$ is defined by

$$\left.\begin{aligned} \hat{s}_{2i} &:= s_{2i} + s_{2i+1} \\ \hat{s}_{2i+1} &:= s_{2i} - s_{2i+1} \end{aligned}\right\} \quad \text{for} \quad 0 \le i < k/2\,, \quad \text{and} \quad \hat{s}_{k-1} := s_{k-1} \text{ if } k \text{ is odd}\,.$$

We want to show that

$$|\hat{\mathbf{s}}| + |\mathbf{s}| = \frac{3}{2}(k + \epsilon) - 3\delta\,,$$

where

$$\delta = \#\{0 \le i < k/2 \mid s_{2i} = s_{2i+1} = 0\}$$

and

$$\epsilon = \begin{cases} 0 & \text{if } k \text{ is even}\,, \\ 1 & \text{if } k \text{ is odd and } s_{k-1} \neq 1\,, \\ -1 & \text{if } k \text{ is odd and } s_{k-1} = 0\,. \end{cases}$$

First, suppose k is even. Breaking $\hat{\mathbf{s}}$ into pairs $(\hat{s}_{2i}, \hat{s}_{2i+1})$ for $0 \le i < k/2$, we have

- $|(\hat{s}_{2i}, \hat{s}_{2i+1})| = 0 \iff |(s_{2i}, s_{2i+1})| = 0$, and there are δ such pairs;
- $|(\hat{s}_{2i}, \hat{s}_{2i+1})| = 2 \iff |(s_{2i}, s_{2i+1})| = 1$, and there are $k - |\mathbf{s}| - 2\delta$ such pairs;
- $|(\hat{s}_{2i}, \hat{s}_{2i+1})| = 1 \iff |(s_{2i}, s_{2i+1})| = 2$, and there are $|\mathbf{s}| - k/2 + \delta$ such pairs.

Summing over the pairs, we find

$$|\hat{\mathbf{s}}| + |\mathbf{s}| = (0+0)\delta + (2+1)(k - |\mathbf{s}| - 2\delta) + (1+2)(|\mathbf{s}| - k/2 + \delta) = 3k/2 - 3\delta\,.$$

If k is odd and $s_{k-1} = 0$ then $\hat{s}_{k-1} = s_{k-1} \neq 0$, so

$$|\hat{\mathbf{s}}| + |\mathbf{s}| = |(\hat{s}_0, \ldots, \hat{s}_{k-2})| + |(s_0, \ldots, s_{k-2})|\,,$$

and applying the even-length argument above to the subvectors yields

$$|\hat{\mathbf{s}}| + |\mathbf{s}| = 3(k-1)/2 - 3\delta\,.$$

Similarly, if k is odd and $s_{k-1} \neq 0$, then $\hat{s}_{k-1} = s_{k-1} \neq 0$, so

$$|\hat{\mathbf{s}}| + |\mathbf{s}| = 2 + |(\hat{s}_0, \ldots, \hat{s}_{k-2})| + |(s_0, \ldots, s_{k-2})|\,,$$

and applying the even-length argument again yields

$$|\hat{\mathbf{s}}| + |\mathbf{s}| = 2 + 3(k-1)/2 - 3\delta = 3(k+1)/2 - 3\delta\,.$$

Proof of Proposition 2. Let $\mathbf{R}$ be in $\mathbb{F}_3^{(n-k)\times k}$. We want to show that for every $\mathbf{s}$ in $\mathbb{F}_3^k$, if $\hat{\mathbf{s}}$ is defined as above, then

$$\mathbf{s}\mathbf{R}^\intercal = -\hat{\mathbf{s}}\mathbf{M}$$

where $\mathbf{M}$ is the matrix in $\mathbb{F}_3^{k\times(n-k)}$ whose rows are

$$\left.\begin{array}{l}\mathbf{M}_{2i} := \mathbf{R}_{2i}^\intercal + \mathbf{R}_{2i+1}^\intercal \\ \mathbf{M}_{2i+1} := \mathbf{R}_{2i}^\intercal - \mathbf{R}_{2i+1}^\intercal\end{array}\right\} \quad \text{for} \quad 0 \le i < k/2\,, \quad \text{and} \quad \mathbf{M}_{k-1} := -\mathbf{R}_{k-1}^\intercal \text{ if } k \text{ is odd}$$

(note the minus sign). To show this, we simply sum the identities

$$\hat{s}_{2i}\mathbf{M}_{2i} + \hat{s}_{2i+1}\mathbf{M}_{2i+1} = -(s_{2i}\mathbf{R}_{2i}^\intercal + s_{2i+1}\mathbf{R}_{2i+1}^\intercal)\,,$$

for $0 \le i < k/2$, and add

$$\hat{s}_{k-1}\mathbf{M}_{k-1} = -s_{k-1}\mathbf{R}_{k-1}^\intercal$$

if k is odd.